IMPROVING FAILURE DIAGNOSIS VIA BETTER DESIGN AND ANALYSIS OF LOG MESSAGES

BY

DING YUAN

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

Doctoral Committee:

Professor Jiawei Han, Chair
Professor Yuanyuan Zhou, Director of Research
Professor Vikram S. Adve
Associate Professor Samuel T. King
Professor Stefan Savage, University of California, San Diego
Doctor Yinglian Xie, Microsoft Research Silicon Valley
Abstract

As software is growing in size and complexity, accompanied by vendors’ increased time-to-market pressure, it has become increasingly difficult to deliver bulletproof software. Consequently, software systems still fail in the production environment.

Once a failure occurs in production systems, it is important for the vendors to troubleshoot it as quickly as possible since these failures directly affect the customers. Consequently, vendors typically invest significant amounts of resources in production failure diagnosis. Unfortunately, diagnosing these production failures is notoriously difficult. Indeed, constrained by both privacy and expense reasons, software vendors often cannot reproduce such failures. Therefore, support engineers and developers continue to rely on the logs printed by the runtime system to diagnose the production failures.

Unfortunately, the current failure diagnosis experience with log messages, which is colloquially referred as “printf-debugging”, is far from pleasant. First, such diagnosis requires expert knowledge and is also too time-consuming, tedious to narrow down root causes. Second, the ad-hoc nature of the log messages is frequently insufficient for detailed failure diagnosis.

This dissertation makes three main contributions towards improving the diagnosis of production failures. The first contribution is a practical technique to automatically analyze the log messages and the source code to help the programmers debugging the failure. Given a production software failure and its printed log messages, programmers need to first map each log message to the source code statement and manually work backwards to infer what possible conditions might have led to the failure. This manual detective work to find the cause of the failure remains a tedious and error-prone process. To address this problem, this dissertation designs and evaluates a technique, named SherLog, that analyzes source code by leveraging information provided by run-time logs to reconstruct what must or may have happened during the failed production run.

The second contribution is the understanding of the quality of log messages for failure diagnosis. The efficacy of failure diagnosis, either manually or with automatic log inference tools such as SherLog, is predicated by the quality of log messages. However, there is little empirical data about how well existing logging practices work and how they can yet be improved. This dissertation provides the first comprehensive study on the logging effectiveness. By examining developers
own modifications to their logging code in the revision history, this study found that developers often do not make the log messages right in their first attempts, and thus need to spend a significant amount of efforts to modify the log messages as after-thoughts. It further provides many interesting findings on where programmers spend most of their efforts in modifying the log messages.

The third main contribution of this dissertation is to improve the quality of the log messages, which is informed and inspired by the characteristic study, to enhance postmortem failure diagnosis. In particular, this dissertation invents two techniques: LogEnhancer and Errlog for this purpose. LogEnhancer is motivated by a simple observation from our characteristic study: log messages often do not contain sufficient information for diagnosis. LogEnhancer solves this problem by systematically and automatically “enhance” each existing log printing statement in the source code by collecting additional informative variable values. However, LogEnhancer does not insert new log messages. The second technique, Errlog, is to insert new log messages into the program to significantly reduce the diagnosis time with only negligible logging overhead penalty. This technique is driven by an empirical study on 250 real-world failures. A controlled user study suggests that Errlog and LogEnhancer can effectively cut the diagnosis time by 60.7%.
To my dear parents, Jian Song and Wei Yuan, 
and my dear wife, Guangji Xu
Acknowledgments

No words can express my appreciations to my advisor, Yuanyuan (YY) Zhou. Without her never-fading trust, encouragement, and guidance throughout my PhD study, this dissertation would not exist. YY is the one guided me from an infant step-by-step into someone who can walk by himself in research, from working on a well-defined technical problem to discovering a problem to solve. YY taught me not only how to do research, but also how to become a better person. I have seldom seen such an inspiring person like YY. She has always been the one to discover my potentials and encourage me to follow my heart and pursue my dreams. Working with YY always makes me feel that there are lots for me to learn, from how to solve a problem to how to identify a good one, from how to make an impact in academia to how to make an impact in industry, from how to mentor students to how to raise child right, from how to keep up one’s faith during adversity to how to stay humble during temporary prosperity. Unfortunately every great journey must come to an end and I can only learn a limited amount of these qualities. In the future, I hope I can become a caring advisor like her to my own students. She also helped me coping with the two-body problem, allowing me to work remotely whenever I wish to so I can spend more time with my family. There is an old saying in Chinese: “having a great mentor is a true blessing for one’s entire life”. Thank you YY, for helping me to realise my dream via such a fun journey in the past six years!

I also would like to thank Stefan Savage. Stefan has given me tremendous amounts of encouragements and insightful feedbacks in our collaborations on two projects. Through our collaboration, I am fortunate enough to observe how he can escalate an idea to a much higher level. I also thank him for a lot of the deep conversations we had where he gave me many insightful career advices and encouragements to follow my intuitions.

I wish to express my appreciations to the other members of my committee — Vikram S. Adve, Jiawei Han, Sam King, and Yinglian Xie for their encouragement and advice on my research. In particular, Jiawei gave me lots of guidance and advice during the early stage of my PhD study. His advice was also influential in my career choices. Yinglian has taught me a lot of things about doing research and so much beyond that. Sam has challenged me and taught me to think deeper about research questions. I also thank Vikram for valuable discussions and advices.

I am very grateful to the members in the Opera research group. I want to thank Zhenmin,
Qingbo, Shan, Joe, Spiros, Lin, Soyeon, Weihang, and Xiao for setting up great examples for me from the very beginning of my PhD study, and in particular for the mentoring I received from them. I thank Lin for mentoring me on my first project that gave me an enjoyable experience in research from the beginning. I thank Xiao for the technical discussions and mentoring. Shan gave me a great amount of advices, not only when she was with the group, but also after she left the group. I want to thank Soyeon for working together with me through many deadlines. I have learnt so much from her, in particular in paper writing. It has been my great privilege to work with Ryan, Jing, Yang, Haohui, Weiwei, Michael, Xiaoming, Zuoning, Dongcai, Kyu Hyung and Gopal. Thank you guys for collaborating with me on projects and bearing with me through many paper submissions/resubmissions, offering enlightening discussions and incessant assistance. I also thank Yoann for all the fun discussions we had, and for his unique way of improving my research: challenging almost everything in my papers. I learnt so much from you, Yoann (but not Ocaml I guess). Thanks Weiwei for the countless discussions we had on technology, career, and life. I also thank other members in the Opera group, including Chongfeng, Jiaqi, Tianwei, Yao, Tianyin, Xinxin, Zhuoer, and Gen. You guys and YY really made the group like a home for me and gave me lots of fun in our read groups, games of Frisbees and basketball, and all the parties we had.

I would also like to thank Geoff Voelker for all his help to me. I have learnt many things from him, including presentations, research, teaching, and of course, hiking. In particular, he sets up an example of a great teacher who exemplifies how complex concepts can be explained in a clear and intriguing way via simple insights and a sense of humor.

I am also thankful to many other people in UIUC and UCSD. Thank you Sarita Adve, Darko Marinov, Josep Torrellas, Ralph Johnson, Jiawei, Jeff Erickson, Vikram, Sam, Kevin Chang, and Tarek Abdelzaher for the enlightening lectures you taught. In particular, I am very grateful to Sarita for her mentoring when I was her Teaching Assistant as well as her recognitions of my TA work. Thank you the entire UCSD sysnet group. Thanks also go to the Sheila Clark, Mary Beth Kelley, and Virginia McIlwain for helping with many administrative stuffs.

Thanks also go to all my other friends in UIUC and UCSD, including Minhao Yu, Tao Cheng, Yan Gao, Chengdu Huang, Yue Lu, Yong Yang, Marti Motoyama, Wanmin Wu, Jack Sampson, Qing Zhang, Xuehai Qian, Md Kamruzzaman, John McMullough, Kirill Levchenko, Zach Tatlock, Chris Kanich, Yuntao Jia, Shuo Tang, Shuyi Chen, Tian Xia, Jing Gao, Lu Su, Ying Huang, Jianqing Zhang, He Liu, Michael Vrable, just to name a few. Thank you all for making my PhD journey so enjoyable.

I also want to thank my collaborators outside UIUC and UCSD. I thank my mentors in Microsoft Research, Yinglian Xie, Junfeng Yang, and Rina Panigrahy for giving me a great experience as a summer intern. Yinglian and Junfeng have been not only mentors to me, but more like
wonderful friends. I also thank Shankar Pasupathy and Lakshmi Bairavasundaram from NetApp for the help on research. In particular I am very grateful to Shankar, who shared with me insights coming his industrial experiences that were so influential to me forming my thesis topic.

Last, but definitely not the least, my deepest appreciations go to my family. I would like to thank my parents for their unconditional love and support for me to pursue my dream. Their love and encouragement have always been the inspirations for me to exert myself in my life of explorations. Thanks go to my wife and my best friend, Guangji, whose unfading supports and encouragements have always doubled my joy after achievements and yet halved my feeling of frustrations. She is also the one who keeps me true to myself. Finally I would like to thank my parents-in-law for their supports and encouragements.

My PhD study is supported by grants from NSF, the Saburo Muroga Fellowship, and NetApp.
# Table of Contents

List of Tables .................................................................................. xi

List of Figures ............................................................................... xiii

Chapter 1 Introduction ................................................................. 1
  1.1 Background on Failure Diagnosis ........................................ 2
  1.1.1 Unique Challenges in Diagnosing Production Failures ....... 2
  1.1.2 The Key Role of Logging ................................................ 3
  1.2 Dissertation Contributions .................................................. 4
  1.2.1 Postmortem Log Inference .............................................. 4
  1.2.2 Understanding the Logging Effectiveness ....................... 6
  1.2.3 Improving the Quality of Logging ................................. 6
  1.3 Outline ............................................................................... 8

Chapter 2 Postmortem Log Inference .............................................. 9
  2.1 A Motivating Example ....................................................... 9
  2.2 Contribution Highlights ................................................... 12
  2.3 Design Overview .............................................................. 14
  2.4 Log Parsing ....................................................................... 17
  2.4.1 Challenges .................................................................. 17
  2.4.2 SherLog’s Approach ..................................................... 18
  2.5 Path Inference ................................................................... 19
  2.5.1 Overall Design and Process ......................................... 20
  2.5.2 Overview of Saturn Static Analysis System .................. 23
  2.5.3 Detailed Design .............................................................. 23
  2.5.4 Reporting Inferred Paths ............................................ 25
  2.6 Value Inference .................................................................. 26
  2.7 Evaluation ......................................................................... 28
  2.7.1 Overall Results ............................................................. 29
  2.7.2 SherLog Diagnosis Case Studies .................................... 30
  2.7.3 Performance of SherLog .............................................. 34
  2.8 Discussion and Limitations ............................................... 34
  2.9 Summary .......................................................................... 36
Chapter 5  
5.4  Exception Identification .................................. 98  
5.4.1  Exception Identification .................................. 98  
5.4.2  Log Printing Statement Insertion ......................... 100  
5.4.3  Run-time Logging Library ................................ 101  
5.5  In-lab Experiment .............................................. 102  
5.5.1  Coverage of Existing Log Points ......................... 102  
5.5.2  Additional Log Points ........................................ 104  
5.5.3  Performance Overhead ...................................... 105  
5.5.4  Real World Failures ........................................ 107  
5.6  User Study ..................................................... 108  
5.7  Limitations and Discussions ................................ 111  
5.8  Summary ....................................................... 113  

Chapter 6  Previous Works ................................. 114  
6.1  Diagnosis without Reproducing the Failure ............... 114  
6.1.1  Log Analysis ................................................ 114  
6.1.2  Trace Analysis ............................................... 115  
6.1.3  Bug-Type Specific Diagnostic Information Collection 116  
6.1.4  Debugging with Core Dumps .............................. 116  
6.2  Diagnosis with Failure Reproduction .................... 117  
6.3  Failure Characteristic Studies ............................. 118  
6.4  Other Related Work .......................................... 118  
6.4.1  Static Analysis for Bug Detection ..................... 118  
6.4.2  Detecting Bugs in Exception Handling Code ............. 119  

Chapter 7  Conclusions and Future Work .................. 120  
References ......................................................... 123
## List of Tables

2.1 Applications and real failures evaluated in our experiments. ..... 28
2.2 Detailed result for SherLog. ................................................. 29
2.3 Performance of SherLog. .................................................... 34

3.1 The major findings on real world logging characteristics and their implications. ..... 38
3.2 Open-source software studied. ............................................. 39
3.3 Modifications in each revision to the log messages. .................... 44
3.4 Log messages(%) that have been modified. ............................... 44
3.5 Type of log modifications as after thoughts. .............................. 45
3.6 Verbosity-level modifications with error event logging and with non-error event logging .................................................. 46
3.7 Reconsiderations of error criticality and verbosity-level modification .... 47
3.8 Reconsideration of logging trade-off and verbosity-level modification .... 48
3.9 Modifications to improve variable logging. ................................ 50
3.10 Variable replacement and deletion. The margin of errors are shown at 95% confidence level. ................................................. 51
3.11 Improving static content of log messages. The margin of errors are shown at 95% confidence level. ................................................. 52
3.12 Reasons for moving or deleting log messages ............................. 54
3.13 Verbosity-level inconsistency detected. ................................... 55

4.1 Evaluated applications by LogEnhancer. .................................. 72
4.2 The number of uncertain branches and uncertain variable values per log point. 76
4.3 Analysis performance. ......................................................... 77
4.4 Real-world failures evaluated by LogEnhancer. ............................ 78
4.5 Performance overhead added by LogEnhancer’s logging. ................ 81
4.6 Comparison between LogEnhancer and core dump. ........................ 82

5.1 Software used in the characteristic study and the number of log points (i.e. logging statements). ................................................. 87
5.2 The number of sampled failures and the subset with failure-related log messages. 88
5.3 Error manifestation characteristics of examined software. ................ 91
5.4 Logging practices when general errors are detected. ........................ 91
5.5 Logging practices for common exceptions. .................................. 93
5.6 The number of hard-to-check failures that could have been caught during testing, assuming 100% test coverage with each criteria. ......................................... 96
5.7 Generic exception patterns searched by Errlog. .................................................. 97
5.8 The new software projects used to evaluate Errlog, in addition to the five examined in the characterization study. ................................................................. 102
5.9 Additional log points added by Errlog. ................................................................. 104
5.10 Optional logging for untested branch decisions. ................................................... 105
5.11 The performance overhead added by Errlog’s logging. ...................................... 105
5.12 Noisy log points exercised during correct executions. ....................................... 106
5.13 Analysis time ........................................................................................................ 107
5.14 Errlog’s effect on the randomly sampled 238 real-world failure cases. .............. 107
5.15 Effectiveness of the log messages added by Errlog for diagnosis using Sherlog. . 109
5.16 Real-world failures used in the user study. ......................................................... 109
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Classic Fault-Error-Failure model.</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Log printing code examples from open-source software.</td>
<td>4</td>
</tr>
<tr>
<td>1.3</td>
<td>The relationship of the three contributions in this dissertation.</td>
<td>5</td>
</tr>
<tr>
<td>2.1</td>
<td>Simplified rmdir.c source code.</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>SherLog's error report for the rmdir bug.</td>
<td>14</td>
</tr>
<tr>
<td>2.3</td>
<td>Must-Path, May-Path, and Pruned-Path.</td>
<td>15</td>
</tr>
<tr>
<td>2.4</td>
<td>The components of SherLog.</td>
<td>16</td>
</tr>
<tr>
<td>2.5</td>
<td>User defined rule to handle error strings returned by strerror().</td>
<td>19</td>
</tr>
<tr>
<td>2.6</td>
<td>SherLog's analysis for rmdir.</td>
<td>20</td>
</tr>
<tr>
<td>2.7</td>
<td>Modeling the for loop as tail-recursive call.</td>
<td>21</td>
</tr>
<tr>
<td>2.8</td>
<td>SherLog connects maximum partial message sequence.</td>
<td>22</td>
</tr>
<tr>
<td>2.9</td>
<td>SherLog report for ln in coreutils 4.5.1.</td>
<td>30</td>
</tr>
<tr>
<td>2.10</td>
<td>ln.c in GNU Coreutils-4.5.1.</td>
<td>31</td>
</tr>
<tr>
<td>2.11</td>
<td>SherLog diagnosis report for Squid 2.3.STABLE4.</td>
<td>32</td>
</tr>
<tr>
<td>2.12</td>
<td>store.c in Squid 2.3.STABLE4.</td>
<td>32</td>
</tr>
<tr>
<td>2.13</td>
<td>SherLog report for CVS in coreutils 4.5.1.</td>
<td>33</td>
</tr>
<tr>
<td>2.14</td>
<td>parseinfo.c in CVS 1.11.23.</td>
<td>33</td>
</tr>
<tr>
<td>3.1</td>
<td>Churn rates comparison between logging and entire code.</td>
<td>43</td>
</tr>
<tr>
<td>3.2</td>
<td>Error verbosity level modifications from PostgreSQL</td>
<td>47</td>
</tr>
<tr>
<td>3.3</td>
<td>Example from PostgreSQL of a verbosity level demotion from default level (non-erroneous) to verbose level, together with developers' commit log.</td>
<td>49</td>
</tr>
<tr>
<td>3.4</td>
<td>Example of adding variable values to log message.</td>
<td>50</td>
</tr>
<tr>
<td>3.5</td>
<td>Logging a wrong variable causing Apache vulnerable to denial of service attack.</td>
<td>51</td>
</tr>
<tr>
<td>3.6</td>
<td>Example of inconsistency between log messages and the code in OpenSSH.</td>
<td>52</td>
</tr>
<tr>
<td>3.7</td>
<td>The use of the programming language support to hold location information for log messages in Squid.</td>
<td>52</td>
</tr>
<tr>
<td>3.8</td>
<td>Example of a log message clarification from PostgreSQL.</td>
<td>53</td>
</tr>
<tr>
<td>3.9</td>
<td>Deleting Logging from a signal handler in Squid.</td>
<td>54</td>
</tr>
<tr>
<td>4.1</td>
<td>Example of real-world patches just for the purpose of enhancing log messages.</td>
<td>56</td>
</tr>
<tr>
<td>4.2</td>
<td>Highly simplified code for rm in coreutils-4.5.4.</td>
<td>58</td>
</tr>
<tr>
<td>4.3</td>
<td>Remaining uncertainty if dp was printed at line 16.</td>
<td>60</td>
</tr>
<tr>
<td>4.4</td>
<td>The constrained expression for is_dir at line 16.</td>
<td>66</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td>4.5</td>
<td>Code and Control-flow graph of tail-recursive calls for loop.</td>
<td></td>
</tr>
<tr>
<td>4.6</td>
<td>Run-time recording for Delayed Collection.</td>
<td></td>
</tr>
<tr>
<td>4.7</td>
<td>Variable values at each log point.</td>
<td></td>
</tr>
<tr>
<td>4.8</td>
<td>Number of Uncertain Branches reduced by LogEnhancer.</td>
<td></td>
</tr>
<tr>
<td>4.9</td>
<td>Number of variables used in branches.</td>
<td></td>
</tr>
<tr>
<td>4.10</td>
<td>Ranking of uncertain variable values in Apache.</td>
<td></td>
</tr>
<tr>
<td>4.11</td>
<td>Apache bug example. Patched code is highlighted.</td>
<td></td>
</tr>
<tr>
<td>4.12</td>
<td>Apache configuration error.</td>
<td></td>
</tr>
<tr>
<td>4.13</td>
<td>Normal execution overhead on fully loaded Apache for different verbose modes.</td>
<td></td>
</tr>
<tr>
<td>5.1</td>
<td>A real world example from Apache on the absence of error log message.</td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td>A real world example from squid to demonstrate the challenge of failure diagnosis in the absence of error messages, one that resulted in a long series of exchanges (45 rounds) between the user and developers.</td>
<td></td>
</tr>
<tr>
<td>5.3</td>
<td>Benefit of logging on diagnosis time (median).</td>
<td></td>
</tr>
<tr>
<td>5.4</td>
<td>Fault manifestation for the sampled failures, and the number of cases with failure-related log messages.</td>
<td></td>
</tr>
<tr>
<td>5.5</td>
<td>Incorrect error handling in Apache resulting in a hang.</td>
<td></td>
</tr>
<tr>
<td>5.6</td>
<td>SVN’s good logging practices for checking and logging function return errors.</td>
<td></td>
</tr>
<tr>
<td>5.7</td>
<td>Logging practices for exception signals.</td>
<td></td>
</tr>
<tr>
<td>5.8</td>
<td>Logging for unexpected cases falling into default.</td>
<td></td>
</tr>
<tr>
<td>5.9</td>
<td>Squid’s good logging practice for resource leak.</td>
<td></td>
</tr>
<tr>
<td>5.10</td>
<td>Example showing the need of control and data flow analysis.</td>
<td></td>
</tr>
<tr>
<td>5.11</td>
<td>Coverage of existing log points by Errlog.</td>
<td></td>
</tr>
<tr>
<td>5.12</td>
<td>User study result, with error bars showing 95% confidence interval.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ixv</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

“As a personal choice, we tend not to use debuggers beyond getting a stack trace or the value of a variable or two. One reason is that it is easy to get lost in details of complicated data structures and control flow; we find stepping through a program less productive than thinking harder and adding output statements and self-checking code at critical places. Clicking over statements takes longer than scanning the output of judiciously-placed displays. It takes less time to decide where to put print statements than to single-step to the critical section of code, even assuming we know where that is. More important, debugging statements stay with the program; debugger sessions are transient.”

— Brian W. Kernighan and Rob Pike, “The Practice of Programming”

Real computer systems inevitably experience failures. While considerable effort is spent trying to eliminate such problems before deployment via testing, or at run-time by tolerating the failures, the size and complexity of modern systems, combined with real time and budgetary constraints on developers have made it increasingly difficult to deliver “bullet-proof” software to customers. In addition, up to 42% of the production failures today are caused by operators’ own mistakes instead of the software’s own defects [Gra86, OGP03, NOB+04, JHP+09, YMZ+11]. Consequently, many software failures still occur in fielded systems providing production services.

Such production failures, once occurred, often have direct impact on customers and end users and require immediate resolutions. For example, it is estimated that the downtime caused by system failures will cost Amazon.com, Inc. millions of U.S. dollars per hour [Amaa, Amab]. The downtime caused by the production failures will be even more catastrophic for those mission-critical software services. For example, the 2003 northeast blackout, caused by a software bug, resulted in over fifty-five million people in U.S. and Canada out of power for at least 7 hours [Sec]. Later in an investigation report, officials state that “failure to provide effective real-time diagnostic support” as one of the primary reasons behind such catastrophic damage.

However, diagnosing such production failures is notoriously difficult. In part it is due to the fundamental complexity of trouble-shooting any complex software system, but it is further exca-
erbated by the paucity of information that is typically available in the production setting. Indeed, for reasons of both overhead and privacy, product support engineers may not be given access to the failed system or its input data in the production environment. Therefore, it is common that only the run-time log generated by a system (e.g., syslog) can be shared with the vendors.

This dissertation investigates how to improve the diagnosis of production failures, focusing on the utility of log messages. It proposes to automatically improve the quality of the log messages before the software is deployed, which is based on comprehensive characterization studies on the logging efficacy, as well as automatic log analysis techniques for postmortem diagnosis.

1.1 Background on Failure Diagnosis

Before discussing the diagnosis of failures, it is useful to first understand how a failure happens. In his seminal work two decades ago, J.C. Laprie decomposed the structural elements of system failures—fault, error and failure—into a model that is widely used today [Lap95]. As shown in Figure 1.1, a fault is a root cause, which can be a software bug, a hardware malfunction, or an operator error. A fault can produce abnormal behaviors referred to as errors. However, some of these errors will have no user-perceivable side-effects or may be transparently handled by the system. It is only the subset of remaining errors which further propagate and become visible to users that are referred to as failures, including crash, hang, incorrect result, incomplete functionality, unnecessary result, etc.

![Figure 1.1: Classic Fault>Error>Failure model.](image)

With this failure model, the failure diagnosis process is to reconstruct the fault-propagation chain, typically in a backward manner starting with the observed failure symptom, to pinpoint the fault. When a failure is reproducible, such as those occurred in the testing environment, some classic debugging tools such as interactive debuggers [GDB, VSD, Ecl] can be applied.

1.1.1 Unique Challenges in Diagnosing Production Failures

While there have been significant advances in postmortem debugging technology, the production environment imposes requirements—low overhead and privacy sensitivity—that are challenging to overcome in commercial settings.
For example, while in principle, deterministic replay—widely explored by the research community [VLW+11, GWT+08, MCT08, VMW, OAA09, AS09, GAM+07, DKC+02, LWV+10, LMC87, DLFC08, KSC00, KDC05, DLCO09, NPC05, CKS+08, VGK+10, XBH03, PZX+09, ZTG06, CBZ11, SN11, LVN10], allows a precise postmortem reproduction of the execution leading to a failure, in practice it faces a range of deployment hurdles including privacy concerns (by definition, the replay trace must contain all input, including any sensitive data), integration complexity (vendors need to recreate the same execution environment as in the production settings, including purchasing licenses for third-party software and hardware), and high overhead (such systems must log most non-deterministic events). Consequently, it is frequently the case that production failures cannot be reproduced by the vendors for diagnosis, and some debugging techniques including interactive debuggers cannot be applied.

By contrast, the other major postmortem debugging advance, cooperative debugging, has a broader commercial deployment, but is less useful for debugging individual failures. In this approach, exemplified by systems such as Windows Error Reporting [GKG+09] and the Mozilla Quality Feedback Agent [Moz], failure reports are collected (typically in the form of limited memory dumps due to privacy concerns) and statistically aggregated across large numbers of system installations, providing great utility in triaging which failures are most widely experienced (and thus should be more carefully debugged by the vendor). Unfortunately, since memory dumps do not capture dynamic execution states, they offer limited fidelity for exploring the root cause of any individual failure. Finally, sites with sensitive customer information can be reticent to share arbitrary memory contents with a vendor.

### 1.1.2 The Key Role of Logging

Consequently, software engineers continue to rely on traditional system logs (e.g., syslog) as a principal tool for troubleshooting failures in the field. Figure 1.2 shows real-world examples of such log printing statements.

What makes these logs so valuable is their ubiquity and commercial acceptance. It is an industry-standard practice to request logs when a customer reports a failure and, since log data typically focuses narrowly on issues of system health, they are generally considered far less sensitive than other data sources. Moreover, since system logs are typically human-readable, they can be inspected by a customer to establish their acceptability. Indeed, large-scale system vendors such as Network Appliance, EMC, Cisco and Dell report that such logs are available from the majority of their customers and many even allow logs to be transmitted automatically and without review [Del08, Net07]. Indeed, presumably commercial vendors would not engage in logging if its costs outweighed the benefits.
Figure 1.2: Log printing code examples from open-source software. Each log printing statement typically has a verbosity level, static text, and variable values. Different types of verbosity levels typically include: fatal (i.e., abort a process after logging), error (i.e., record error events), info (i.e., record important but normal events), debug (i.e., verbose logging only for debugging). Under default production setting, open-source software typically only log error events in addition to a few (less than 10% [YPH+12]) book-keeping messages (e.g., info) due to performance concerns.

Given the key role log message plays in diagnosing production failures, the effectiveness of such diagnosis is fundamentally constrained by two challenges:

- **Large manual inference efforts**: given the log messages printed from a failed execution, programmers currently need to manually examine the source code to infer why these messages were printed. Such inference effort is frequently tedious and error-prone.

- **Quality of the log message**: the effectiveness of failure diagnosis is ultimately predicated on what gets logged. For example, if a failure prints no log messages, engineers would have no clue to diagnose the underlying cause. As it will be shown in this dissertation, failures with log message printed will have a much shorter diagnosis time (2.2X) than those without.

Therefore these two challenges can significantly increase software’s downtime and severely impact software system’s overall availability.

### 1.2 Dissertation Contributions

This dissertation makes three contributions towards expediting the diagnosis of production failures: postmortem log inference, understanding the logging effectiveness, and improving the quality of logging. These three components of this dissertation motivate and complement each other as shown in Figure 1.3. The tools designed in this dissertation have been requested for release by many commercial vendors including Google, NetApp, Cisco, EMC, Huawei, etc.

#### 1.2.1 Postmortem Log Inference

My dissertation research started with observing the problem experienced by virtually all programmers in debugging: given the log messages printed from a failed execution, programmers currently...
need to manually examine the source code to infer why these messages were printed. Such inference requires them to follow the complex logic of the program’s source code, thus is tedious, error-prone, and often cannot be carried deep. However, such manual inference efforts can be well automated by computer programs. Using program analysis, we can automatically infer the failed execution if an execution path would print the exact same sequence of log messages. This observation motivated the design, implementation, and evaluation of SherLog [YMX+10], a tool that analyzes the source code by leveraging information provided by run-time logs. It uses symbolic execution to infer the feasible execution paths that would print the same sequence of log messages. It also infers the value of key variables.

The key challenge faced by SherLog’s design is to be both scalable and precise. Precise program analysis is expensive, yet it is required by debugging as incorrect information might lead programmers into a wild goose chase. The key insight toward achieving these seemingly conflicting requirements is that failure diagnosis, unlike other common program analysis tasks such as bug finding, is highly target-driven. Inferring the root cause of a failed execution often involves examining only a small slice of the program. Driven by this insight, SherLog only analyzes those functions that either directly print log messages or indirectly print logs through its callee. This design allows SherLog to only focus on analyzing a small number of functions that are highly relevant to the failed execution. Therefore, SherLog can afford to further use heavy-weight, precise program analysis, namely symbolic execution, to infer the root cause.

SherLog is evaluated on 8 real world failures from GNU Coreutils, tar, CVS, Apache and Squid, including 6 software bugs and 2 configuration errors. The result shows that all of the failures can be accurately diagnosed by SherLog. The performance evaluation shows that analysis for simple applications such as GNU Coreutils could finish within 5 minutes. For server applications such as Apache and Squid, SherLog is applied on logs from hours of execution with thousands of messages, and the analysis still finishes within 40 minutes.
1.2.2 Understanding the Logging Effectiveness

However, the effectiveness of failure diagnosis is ultimately predicated on the quality of the log messages. Despite their widespread use in failure diagnosis, log messages are still designed in an ad-hoc fashion. Therefore, a natural question is “how effective are the log messages for diagnosis?” While asking this question is trivial, answering it is not. It is challenging to objectively judge the efficacy of log messages since there is often no rigorous specification of software’s logging behavior. This dissertation addresses this challenge by studying developers’ own modifications to their log messages as after-thoughts. It shows that developers often do not make the log messages right in their first attempts, and thus need to spend significant efforts to modify the problematic log messages as after-thoughts. This study further provides several interesting findings on where developers spend most of their efforts in modifying the log messages, which can give insights for programmers, tool developers, and language and compiler designers to improve the current logging practice. To demonstrate the benefit of this study, a simple checker is built based on one of the findings and effectively detected 138 pieces of new problematic logging code from studied software (30 of them are already confirmed and fixed by developers).

1.2.3 Improving the Quality of Logging

Motivated by the findings on logging inefficacies, this dissertation further presents techniques to improve the quality of log messages. In particular, it proposes two techniques that together can make log messages more informative for diagnosis.

Enhancing Existing Log Messages: One of the disturbing problem found in the study described above is that existing log messages often do not contain sufficient information. For example, in many cases, an error log message may simply print “system failed” without providing any further information for diagnosis. Only until later when the failure occurred would developers discover they need more information. Indeed, the study described above identified 1546 such “enhancements” in the form of patches to add additional variable values into the log messages as after-thoughts.

Motivated by this problem, this dissertation presents a tool named LogEnhancer to enhance the quality of log messages. It systematically and automatically modifies each log message in a given piece of software to collect additional causally-related information. Such information can ease the diagnosis of future failures and thereby improve the diagnostic power of logging in general. The insight of LogEnhancer is again from observing how programmers use the log messages in practice. Given a message being printed, programmers would start from the log printing statement in the source code and infer backwards to understand why it was printed. In such process, complexity
arises when there are too many uncertainties, e.g., too many execution paths that are causally-related but cannot be disambiguated given only the printed log message. Therefore, an ideal log message should record information that can disambiguate such uncertainties. LogEnhancer automates this process by analyzing the source code in a backward manner to identify the variable values, if known, would resolve such uncertainties.

Evaluating the effectiveness of LogEnhancer’s log enhancements is challenging. Fortunately, existing log messages already contain variable values that are added manually by developers, therefore LogEnhancer’s inference should at least cover those. Indeed, LogEnhancer can automatically infer over 95% of the existing variable values to each log message, indicating it can be at least as good as manual logging efforts. Furthermore, LogEnhancer inferred additional variable values in each log message, which are extremely useful in trouble-shooting failures.

**Log Message Placement:** While LogEnhancer enhances the quality of existing log messages, it assumes programmers already appropriately placed log message. But what if there are no relevant messages printed for a failure in the first place? Where is the best place to log? Motivated by such questions, this dissertation further describes a study that characterises the efficacy of log messages during failure diagnosis. It examines 250 randomly sampled user-reported failures from five software systems (Apache httpd, Squid, PostgreSQL, SVN, and GNU Coreutils) and identifies both the source of the failure and the particular information that would have been critical for its diagnosis. Surprisingly, it shows that the majority (77%) of these failures manifest through a small number of concrete error patterns (e.g., error return codes, switch statement “fall-thoughts”, etc.). Unfortunately, more than half (57%) of the 250 examined failures did not log these detectable errors, and their empirical “time to debug” suffers dramatically as a result (taking 2.2X longer to resolve on average in this study).

Driven by this result, this dissertation further shows that it is possible to fully automate the insertion of such proactive logging statements parsimoniously, yet capturing the key information needed for postmortem debugging. It describes the design and implementation of a tool, Errlog, and shows that it automatically inserts messages that cover 84% of the error cases manually logged by programmers across 10 diverse software projects. Further, the error conditions automatically logged by Errlog capture 79% of failure conditions in the 250 real-world failures used in the empirical study. Finally, using a controlled user study with 20 programmers, it is shown that the error messages inserted by Errlog can cut failure diagnosis time by 60.7%.

---

1The data used can be found at: http://opera.ucsd.edu/errlog.html
1.3 Outline

The remainder of this dissertation is organized as follows. Chapter 2 presents SherLog, the post-mortem log inference tool. Chapter 3 then describes the characteristic study on logging efficacy. Next, Chapter 4 describes how LogEnhancer enhances the quality of existing log messages, and Chapter 5 presents the problem of log placement and how to address it by Errlog. Chapter 6 describes previous work on failure diagnosis and other related topics, before it concludes.

The materials in some chapters have been published as conference and journal papers. The materials in Chapter 2 have been presented in the International Conference on Architecture Support for Programming Language and Operating Systems (ASPLOS’10) [YMX+10]. The materials in Chapter 3 were published in International Conference on Software Engineering (ICSE’12) [YPZ12]. The materials in Chapters 4 have been published in ASPLOS’11 [YZP+11] and ACM Transactions on Computer Systems [YZP+12]. The materials in Chapter 5 have been published in Symposium on Operating Systems Design and Implementation (OSDI’12) [YPH+12].

While I am the leading author of all the works presented in this dissertation, they are indeed the results of a group effort. Please see the conference and journal publications for the list of my collaborators who contributed to this dissertation [YMX+10, YPZ12, YZP+11, YZP+12, YPH+12].
Chapter 2

Postmortem Log Inference

This Chapter describes the design, implementation, and evaluation of SherLog, a postmortem log inference tool that connects clues from the runtime logs.

Once a production failure occurs, the common practice in industry is that customers send vendors logs that are generated by the vendor’s system, and such logs are frequently the sole data source (in addition to their source code) to troubleshoot the occurred failure. Based on what are in the logs and source code, engineers manually infer what may have happened, just like a detective who tries to solve a crime by connecting all seemingly unrelated on-site evidence together. Therefore, to resolve a production run failure quickly, it typically requires experienced engineers to manually examine logs collected from customers. In some difficult and urgent cases, customers allow vendors to provide a newer instrumented version to collect more detailed logs, especially at the suspect code locations after support engineers’ first round of log analysis. Such handshakes usually can iterate a couple of times, but not more because it requires customers’ close collaboration, and can distract customers away from attending their own core business. Therefore, each iteration of the log analysis needs to be effective to quickly zoom into the right root cause within only a few round of interaction with customers.

2.1 A Motivating Example

Even though human intelligence still greatly exceeds machines’, there are a few tasks, especially those dull and tedious ones, that machine automation does excellent job in offloading from humans. It not only saves time and human cost, but also provides better accuracy and more comprehensiveness. Failure diagnosis is such a task.

Let’s consider a real world failure example in rmdir of GNU core utilities version 4.5.1. Executing rmdir -p with a directory name ended by slash will mistakenly cause the program to fail. When executing rmdir -vp dir1/dir2/, where dir1/ and dir1/dir2/ are existing directories, the buggy program only removes dir1/dir2/, without removing dir1/ as it should have.

The log produced by rmdir contains the following text (we number each message for the convenience of our description):
remove_parents (char *path) {
  char *slash;
  while (1) {
    slash = strrchr (path, '/');
    if (slash == NULL)
      break;
    /* Remove any characters after the slash, */
    slash[0] = 0;
    /* printing out removing. */
    if (verbose)
      error(0,0,_("removing directory, %s"), path);
    fail = rmdir (path);
  }
  /* printing out removing. */
  if (verbose)
    error(0,0,_("removing directory, %s"), path);
  fail = rmdir (path);
  if (fail) {
    ...;
    error (0,errno,"%s", quote(path));
    break;
  }
  return fail;
}

main (argc, argv) {
  while (optc = next_cmd_option) {
    switch (optc) {
      case 'p':
        empty_paths = 1;
        break;
      case 'v':
        verbose = 1;
        break;
      ...
    }
  }
  for (; optind < argc; optind++) {
    char* dir = argv[optind];
    if (verbose)
      error(0,0,_("removing directory, %s"), dir);
    fail = rmdir (dir);
    if (fail) {
      ...;
      error (0,errno,"%s", quote(path));
      break;
    }
  }
  return fail;
}

Figure 2.1: Simplified rmdir.c source code. Bold statements are those print out log messages.

rmdir: removing directory, dir1/dir2/ [msg 1]
rmdir: removing directory, dir1/dir2  [msg 2]
rmdir: 'dir1/dir2': No such file or directory [msg 3]

Figure 2.1 shows the highly simplified code of rmdir.c. With the -p option, empty_paths will be set to 1, causing remove_parents to be called after removing each argument directory, which will remove all the super directories along the hierarchy. Line 5-24 of remove_parent shows the loop that removes all the super directories. In each iteration, path moves one level up along the directory hierarchy by removing all the characters after the last slash of the current directory's path. The error is triggered when initially path ends with a trailing slash, so the first iteration will
simply remove the last slash, resulting in the same directory which has just been removed in line 50 in `main`. The fix is simply to remove all the trailing slashes before entering the loop starting at line 5 in `remove_parents`.

While the root cause may sound simple once known, it was a “mystery” the first time this failure is reported and being diagnosed. The only information available is just the log messages. By examining the logs and the corresponding source code, an engineer may find that code statements \( r_1, m_1 \) could generate the first two messages, and \( r_2, m_2 \) generate the third message. However, simple combinatorics would indicate a total of 8 combination possibilities, and different combinations would take engineers to different paths for narrowing down the root cause.

However, by conducting a deeper analysis, we can find out that six of the eight paths are infeasible because they are self-contradictory; and only two paths, namely \( \{m_1, r_1, r_2\} \) or \( \{m_1, m_1, m_2\} \) are remaining for the next step of follow-up. For example, \( \{r_1, r_1, r_2\} \) would not be feasible because it implies \( m_1 \) not being executed, which requires `verbose` not being set, contradictory with the constraint for \( r_1 \) to be executed.

In addition, to diagnose a failure, engineers often need to know more than just log-printing code locations. Specifically, they may need to know what code has been executed and in what order, i.e., the execution path, as well as values of certain variables. As log messages are generally sparsely printed, there may be tens of thousands of possible paths that lead to the same log message combination. Engineers would need to manually reason about what code segments and control flows must have been executed (referred to as `Must-Path`) and what may have been executed (referred to as `May-Path`).

Unfortunately, above analysis is usually beyond human’s manual analysis effort as it is a tedious and repetitive process, with each step requiring examining its constraints and also checking them against existing assumptions for feasibility analysis. Many other real world cases are much more complicated than the simple example shown here. Many logs contain hundreds and even thousands of messages, and each message can provide some information about what could have happened in the failed production run. Not all possible combinations are feasible, it requires a non-trivial, lengthy analysis to prune those infeasible ones and sort those “must-have-happened” and “must-not-have-happened” from “may-have-happened”. Such analysis results would provide engineers useful clues to narrow down the root cause. Later sections will show more real world failure examples that are caused by either software bugs or mis-configurations.

Therefore, it is highly desirable to use an automated log inference engine to automatically infer from the available data (logs and source code) on what have happened in the failed production run before narrowing down the root cause. This is exactly the objective of this work.
2.2 Contribution Highlights

SherLog is motivated by the problem above. It is a postmortem error diagnosis tool that leverages logs as starting points in source code analysis to automatically infer what must or may have happened during a failed execution.

To be general and practical, SherLog meets the following design goals:

- **No need to re-execute the program:** For practicality, SherLog only assumes the availability of the target program’s source code and logs generated from failed executions. These two assumptions are quite reasonable as diagnosis are done by programmers themselves and it has been a common practice for customers to send logs back to vendors. For example, most storage vendors collect failure logs from more than 50-75% of their customers [JHP+09, Net07, Del08].

  All inferences are performed statically using path- and context-sensitive program analysis and a satisfiability constraint solver to prune infeasible combinations. This also allows SherLog to be applicable to even system code such as operating systems for which dynamic debugging is very cumbersome and challenging.

- **No assumption on log semantics:** For generality, SherLog assumes no semantic information on logs. For example, SherLog does not know which log message is a configuration parameter, etc. It simply treats every log message as a text string to match against source code. Every match provides a “hint point” for backward and forward inference on execution paths, variable values, etc. All such derived information is then combined together to prune infeasible ones and reveal a bigger, global picture regarding what may have happened during the failed execution.

- **Able to infer both control and data information:** Since both execution path and variable values provide programmers useful clues to understand what have happened during the failed production run, SherLog infers both control flow and data flow information. This is accomplished via the following three steps (Figure 2.4): first, the log file is automatically parsed by SherLog’s parser to match each message to its corresponding logging statement(s) in source code (referred to as “Logging Points”). Variable values printed out in logging messages are also parsed to refine our analysis. Then SherLog’s path inference engine infers the execution path based on information from our log parser, along with the constraints that an inferred path needs to satisfy. Finally, SherLog allows user to query the value of variables along an inferred execution path, and returns the inferred value of a variable at its definition and modification points.

- **Scalability:** Real world software often have thousands or millions lines of code. Therefore, SherLog needs to be scalable to handle such large software. To improve scalability, SherLog uses function summaries and skip log-irrelevant code to limit the inference space.

- **Accuracy:** Information reported by SherLog needs to be accurate. Incorrect information can
lead programmers to a wild goose chase, and thus waste effort and delay diagnosis. SherLog’s path-sensitive analysis faithfully models the C program’s semantic down to bit level, with a SAT solver to prune infeasible paths. Caller-visible constraint information is propagated from callee back to caller to ensure context-sensitivity.

• **Precision:** The reported information also needs to be precise as too many possibilities do not help programmers narrow down the root cause. For this reason, SherLog ranks the results based on probabilities. Must-Paths are ranked higher than May-Paths. Along each path, the necessary constraints, are ranked higher than sufficient but not necessary constraints. In value inference where multiple values of a variable might be possible, SherLog ranks concrete values higher than symbolic ones.

• **Capability of automatically generating log parsers:** Real-world software have hundreds or thousands of code lines that can print out log messages, many of which have distinctive log formats, therefore it would be tedious and error-prone to manually specify how to parse each log message. By extending the standard C format strings, SherLog can automatically generate a parser matching majority of the Logging Points. And with only a few annotations by developers, SherLog can be easily customized to developers’ specific logging mechanisms.

SherLog is evaluated on 8 real world failures from GNU coreutils, tar, CVS, Apache and Squid, including 6 software bugs and 2 configuration errors. The result shows that all of the errors can be accurately diagnosed by SherLog. The performance evaluation shows that analysis for simple applications such as GNU coreutils could finish within 5 minutes. For server applications such as Apache and Squid, we used logs from hours’ of execution with thousands of messages, and the analysis still finishes within 40 minutes. The maximum memory usage for analyzing the 8 real world failures is 1.3 GB.

For the real world example shown on Figure 2.1, the information generated by SherLog is presented by Figure 2.2. For the first path, \{m1,r1,r2\}, SherLog infers necessary constraints such as `strrchr()` returns non-NULL value at line 6, `rmdir()` returns failure (non-zero) at line 17. The developer can query the value of `path` in `remove_parents`, and SherLog would report the value of `path` at each definition point in `remove_parents`. All these information can help the developer getting much closer to the root cause.
Figure 2.2: SherLog’s error report for the `rmdir` bug. The Must-Paths are automatically generated to aid diagnosis. SherLog also infers the necessary constraints, i.e., constraints that must have been satisfied to execute a Must-Path.

### 2.3 Design Overview

SherLog takes two things as input: (1) a log recorded during the failed execution, mostly from production runs at customer sites; and (2) source code of the target program.

Note that SherLog’s objective is not to detect bugs. Instead, its objective is to infer information to help programmers understand what have happened during the failed execution. Such information includes the likely execution paths (and their constraints) as well as certain variable values along these paths. So at a high level, SherLog’s objective is similar to program slicer [AHLW95] and core-dump analyzer [GKG09]. But they differ in both inputs and information inferred.

**The Ideal Goal:** Ideally, it would be great if SherLog could output the exact, complete execution path and state as what actually happened during the failed execution. Unfortunately, it is impossible to achieve this goal since SherLog does not have the input data, the execution environment including file systems and databases, etc. All it has is just logs generated by the program from the

---

3 Even though SherLog targets for diagnosing production-run failures, it can help diagnosing failures occurred during in-house testing as well.
failed execution. Therefore, with limited available information, SherLog has to lower its goal.

**A Realistic Goal:** If only looking at the source code, there are many possibilities in terms of both execution paths and states during the failed execution. Fortunately, logs significantly help in narrowing down the possibilities, even though it may not pin-point the execution path and state that actually happened during the failed execution. Therefore, a realistic goal is to find the following information (illustrated in Figure 2.3):

```plaintext
A log:  
msg1
msg2

1 main() {  
2   ...  
3   if (A)  
4       log(msg1);  
5   else    
6       log(msg2);  
7   if (!A)  
8       log(msg1);  
9       if (var)  
10      b1();  
11      else    
12      b2();  
13   }  
14   b1() {  
15      c();  
16   }  
17 }  
18   b2() {  
19      c();  
20      if (A)  
21          log(msg2);  
22   }  
23   c() {  
24      if (A)  
25          log(msg2);  
26   }
```

**A log:**
- **msg1**
- **msg2**

**Must-Path & May-Path:**
- Node 1: `A ≠ 0`
- Node 4: `{A ≠ 0}
- Node 9: `var = 0`
- Node 10: `{A ≠ 0 && var = 0}
- Node 16: `{A ≠ 0 && var = 0}
- Node 20: `{A ≠ 0 && var = 0}
- Node 25: `{A ≠ 0}

**Pruned-Path:**
- Node 1: `A ≠ 0`
- Node 4: `{A ≠ 0}
- Node 7: `{A ≠ 0 && A = 0}
- **Unsatisfiable**

**Figure 2.3:** Must-Path, May-Path, and Pruned-Path. The Must-Paths are shown in solid lines, while May-Paths are presented in dashed lines. Pruned-Path is analyzed by SherLog but determined infeasible by our SAT solver. Logging Points are highlighted in bold circles and fonts. Constraints along the path are shown in `{}` on the right of a circle.

- **Must-Have:** partial execution paths that were definitely executed (referred to as **Must-Path**), and variable values that were definitely held during the failed execution.
- **May-Have:** partial execution paths that may have been executed (referred to as **May-Path**) and their corresponding constraints for them to happen, as well variable values that may have been held during the failed execution.
- **Must-Not-Have:** execution paths that were definitely NOT executed (referred to as **Pruned-Paths**).
**SherLog Architecture:** To accomplish above objectives, SherLog first needs to parse logs and using logs to identify starting points in source code for information inference. Then using the initial information provided by logs, it tries to statically “walk” through the code to infer the above information. As illustrated in Figure 2.4, SherLog has three main analysis steps: (1) *Log Parsing:* for each log message\(^2\), identifying which lines in the source code prints it (referred to as *Logging Points*), and what program variable values are revealed in this log message (referred to as *Logging Variables*); (2) *Path Inference:* starting from the information provided by the log parsing step to infer Must-Paths, May-Paths and Pruned-Paths. (3) *Value Inference:* infer variables values on the paths produced by the previous step.

This section provides a brief summary of each component. The details are explained in the following three sections.

![Figure 2.4: The components of SherLog.](image)

**Log Parsing:** In this step, the first operation is to identify the possible source code locations that can output each log message. This is a string matching problem and can be solved via many methods, but each with different efficiency. Instead of doing a brute-force matching from log messages to source code, SherLog uses an innovative approach that starts from the source code itself to obtain all possible log message formats in regular expressions to match any run-time logs produced by this program. To support complex, customized logging facility, SherLog also provides extension hooks to allow programmers to adapt specific logging utilities and format.

**Path Inference:** Path inference starts from log points and try to reconstruct paths (sometimes partial paths) that can connect them together. To find precise, accurate and relevant information, it starts from information available from logs, and use a SAT solver to find all feasible paths and their constraints. To help programmers narrow down the possibilities, SherLog also separates Must-Paths from Maybe-Paths. To scale to large software, it uses function summaries and skip functions that are not relevant to log messages.

**Value Inference:** Theoretically it is impossible to provide sound and complete solution for value inference in static analysis, without environmental information such as inputs and libraries. How-

---

\(^2\)Each line in the log is referred to as a log message.
ever the observation made by SherLog is that in most cases only the values involved in constraints of the error execution path, as well as those printed in the log messages are important in error diagnosis. The former are important because they are the ones leading the execution to the error site, while the latter are important since programmers often only prints values that directly caused the error. SherLog makes best efforts to approximate a sound solution only to these variables to achieve overall balance between effectiveness and efficiency. It does so by symbolically execute the program faithfully following the path inferred in the path inference step, taking all the constraints along this path as facts for variable values, and propagating the value information across function calls.

All three components of SherLog are built on top of the Saturn [XA07] Static Analysis Framework, because Saturn offers both the flexibility in expressing our analysis algorithm and precise static analysis of C programs.

2.4 Log Parsing

The objectives for SherLog’s log parser (referred as LogParser) include parsing log messages, connecting them to source code, and providing the initial set of information for inference in later steps, namely Path Inference and Value Inference.

To achieve above objectives, LogParser performs two tasks. First, it needs to identify Logging Points, i.e., possible code locations that produce each log messages. Second, LogParser extracts variable values that are directly contained in log messages. Such values are not only useful for programmers to understand the failure, but also can be used as additional constraints to guide our next step of inference.

2.4.1 Challenges

If programmers simply use a printf to output each log message entirely, identifying Logging Points is relatively straightforward. LogParser can just use phrases from each log message to search against the entire source code and then compare each returned result against the entire log message to find the true match.

Unfortunately, most real-world software usually use a modularized logging facility, that is typically implemented as complicated wrappers around standard C library printing calls to support customized logging format. In the example shown in Introduction, the error call at line 53 and 21 will finally call strerror() function, to get the constant string “No such files or directory” corresponding to the value of errno. The third argument for error call at line 48 and 15, _("removing
directory, %s") is actually a wrapper to a call to dcgettext() function for internationalization support. The quote(dir) call at line 53 wraps the directory string with a pair of single quotes.

Above complex logging facility makes the naive method described earlier ill fitted. For example, it is hard for this method to find the match for msg 3 because all the sub-strings in this message are generated dynamically from either user input or functions such as strerror().

### 2.4.2 SherLog’s Approach

To address the above challenge, SherLog’s LogParser uses a general mechanism that starts from source code and its abstract syntax tree (AST) to automatically generate regular expression based parsers to parse log messages and extract variable values. In addition, it also provides an easy way for programmers to support complex, nested logging facility. Here, I first describe the basic LogParser, and then discuss how to support complex logging mechanisms.

**Basic LogParser:** The basic LogParser requires programmers to indicate only the logging functions and which parameter in this function is the format string. For example, in the rmdir example, error() is the logging function and its third parameter is the format string. In this example, users provide 3 annotations, i.e., \{error(), 3, 4\}, meaning error() is the logging function, the 3rd parameter of error() is the format string (“removing directory, %s”), and parameters starting from the 4th parameter feed the wild cards, e.g., %s, in the format string.

Then LogParser scans the entire source code and find out all possible format strings from the source code that can produce log messages. This is done through traversing the Abstract Syntax Tree (AST) of the program to extract all the format string arguments from all calls to the logging function. It is easy to walk up the AST to get the actual values for the format string arguments to the logging function call. In the rmdir example, the parser first identifies both “removing directory, %s” and “%s” as format strings from error calls.

Now from all obtained format strings, LogParser builds a collection of regular expressions that can be used to match against log messages. Each regular expression is associated with a code location, i.e., a Logging Point. For each log message, LogParser matches it against the collection of regular expressions. The corresponding code locations for any matching regular expressions are possible Logging Points for this message. In the rmdir example described in Figure 2.1, LogParser finds out that msg1 matches two possible Logging Points: code Line 48 (m1) and Line 15 (r1).

For a matching regular expression, SherLog also maps the value back to the format string argument to obtain the associated variable values. For example, if [msg 1] is generated by Logging Point m1, LogParser can obtain the value of variable dir at m1 as dir1/dir2/. Similarly, if it is generated by r1, LogParser knows that the value of variable path at r1 is dir1/dir2/.

**Supporting complex logging mechanisms:** The basic LogParser described above can work well
for programs with relatively simple, flat logging mechanisms. Unfortunately, some real world programs use more complex, hierarchical logging facility with nested log formatting from some assisting functions. Such complication requires us to extend our basic LogParser.

In the rmdir example, error() sometimes calls strerror() to get a string that corresponds to the value stored in errno. Now we cannot simply use the format string "%s" in line 53 or 21 as a wild-card to match msg3. Doing so would infer incorrect value of dir at line 53 or path at line 21 being "'dir1/dir2': No such file or directory". To correctly parse msg3, We need to separate the string constant "No such file or directory" from the rest of the message.

To address this problem, LogParser provides a simple extension mechanism to allow programmers to annotate complex format string. Programmers can define a new format string '%s': 

```
\{serrno\}
```

This new format string can be used to match [msg3], with %s mapped to dir1/dir2, and serrno mapped to a Regex instance No such file or directory.

```
rule = [{"specifier": serrno; "regex": Regex; "val_func": ErrMsgToErrno}]
```

Figure 2.5: User defined rule to handle error strings returned by strerror().

Programmers can use the above APIs to construct a group of Regex with the constant error messages returned by strerror or any other third-party libraries whose source code is not available to SherLog. One such error message is No such file or directory. ErrMsgToErrno is an optional value mapping function to extract variable values. In this case, ErrMsgToErrno reversely maps each error message to its corresponding error number errno. Thus, we know that if the Regex maps to No such file or directory, the value of errno would be ENOENT. It can then be used as an additional constraint in our path/value inference.

These user defined regular expressions are then added into the collection to map against log messages. In rmdir case, SherLog infers that the value of dir at line 53 or path at line 21 were dir1/dir2, and the value of errno were ENOENT.

### 2.5 Path Inference

Given a log, SherLog’s goal is to reconstruct paths that could have generated the log or parts of the log, along with the constraints that the path needs to satisfy. To make SherLog useful and practical for real world systems, SherLog has to meet several design goals: (1) Precise: be as concrete as possible to narrow down the possibilities for programmers. (2) Accurate: information reported by SherLog should be correct. (3) Scalable: can handle large software. (4) Relevant: most information reported should be related to the occurred failure.
To achieve the above goals, SherLog’s Path Inference engine uses a log-driven, summary-based approach. Being log-driven allows SherLog to focus on relevant information. Summary-based approach enable SherLog to scale to large software such as server applications. To be accurate, SherLog uses inter-procedural path- and context-sensitive analysis, and reports the constraints associated with each feasible path. To improve precision, SherLog separates Must-Path from May-Path so programmers can first focus on those paths that have definitely happened. Additionally, when it is impossible to narrow down the complete path between two log points, SherLog tries to reconstruct the maximum partial paths.

This section first provides a high-level overview of the design and process, and then describe SherLog’s approach in more detail.

### 2.5.1 Overall Design and Process

SherLog is summary-based, that each function $f$’s body is analyzed separately, and at the call-site of $f$, only a summary of $f$ is used. The summary of each function $f$ is the subsequences of logging messages $f$ might print out, along with the constraints that needs to be satisfied for $f$ to print out those messages. If there is a path $P$ in $f$ that calls $a$ and $b$, $a$ prints [msg 1] under the constraint $C_a$ and $b$ prints [msg 2] under the constraint $C_b$, then $f$ might print [msg 1, 2] given that the constraint $C_a \wedge C_b \wedge C_P$ is satisfiable, where $C_P$ is the constraint along $P$. Then the analysis is a recursive process, that given a message sequence, initially the Log Parser provides the Logging Points in the program that print each message, under the constraints that the log variables must hold the values revealed in the log. SherLog starts with the functions that directly call these Logging Points, and recursively propagating the subsequences a function connects from callee to caller, along with constraints. The caller would concatenate all the sub-sequences the callees connects into a longer sub-sequence. At the end, if SherLog can infer a complete execution path of the
program that prints the entire message sequence, then SherLog should find \texttt{main} has summary that prints the entire log. Otherwise, SherLog only inferred partial paths. The path SherLog reports is the call chains and the constraints it satisfies. Figure 2.6 shows the analysis order of functions for the \texttt{rmdir} case.

![Figure 2.7: Modeling the \texttt{for} loop as tail-recursive call.](image)

Loops are modeled as a function with a tail-recursive call to itself in SherLog, that at the end of a loop body, a call instruction is inserted with the target to the current loop’s head. Thus, loop is treated with no difference than a function. I will use the term function to also refer to loop in the rest of this Chapter. This way, repeating logging messages printed by loop iterations would be precisely connected by analyzing function/loop repeatedly until it can not connect more logging messages. Figure 2.7 shows the control flow graph of the \texttt{for} loop at line 45.

SherLog only analyzes functions that directly or indirectly calls the Logging Points. In the \texttt{rmdir} case, only the functions shown in Figure 2.6 are analyzed, while functions like \texttt{rmdir()} on line 17 in Figure 2.1 is skipped. For functions SherLog analyzes, we use precise path-sensitive analysis to faithfully follow the program’s semantic. For functions that doesn’t print logging messages, SherLog simply ignores their side-effects. This is a trade-off in our design decision to balance scalability with precision. This allows us analyzing functions that \textit{must} affect the execution that generates the logging sequence precisely, while not wasting time on those functions that \textit{may} have effects. Skipping the side-effects of functions is generally unsound, that might introduce both false-positives and false-negatives in the paths we inferred. However, we find this approach works well in practice since partial side-effects of these skipped functions could still be recovered from the constraints along a path. For example, without analyzing \texttt{rmdir()} on line 17, SherLog still infers that it returns non-zero value along the path \{m1, r1, r2\}; without analyzing the while loop at line 32, SherLog still infers that \texttt{verbose} needs to be non-zero along the two paths. Later in our value inference, we allow user to further query why the return value of \texttt{rmdir()} is non-zero.

Modeling all functions in the program would result in a full-symbolic execution approach which is unlikely to scale well to large applications.

Once the analysis is finished, SherLog reports the paths that connects the longest sub-sequence of logging messages involving the error message. By default, SherLog assumes the last message is
the error message unless otherwise specified by user. There might be multiple paths that connects
the same sequence of Logging Points. In this case, SherLog would “diff” all these paths and first
report the common records along these paths as Must-path before output the rest.

Figure 2.6 shows how SherLog infers the two paths \{m_1, r_1, r_2\} and \{m_1, m_1, m_2\} in rmdir
example. For \{m_1, r_1, r_2\}, log parser could map message 1, 2 and 3 to error() at line 48, 15 and
21 respectively. SherLog first analyzes the while loop at line 5, as it will directly call two Logging
Points error()@15 and error()@21. This would result in a path error()@15->error()@21, with
the constraints \(C_w\) as “strrchr \(\neq\) NULL \&\& verbose \(\neq\) 0 \&\& rmdir()@17 \(\neq\) 0”. Thus a summary of
this while loop will be generated and stored, including the path, the messages it connects ([msg2]
and [msg3]), and a converted constraint \(C_r\) as “verbose \(\neq\) 0”. Because this summary is to be used
by the caller of the while loop, so the constraint must be converted to filter out caller invisible
fields such as “strrchr \(\neq\) NULL \&\& rmdir()@17 \(\neq\) 0”. Then SherLog analyzes the caller of this
while loop, remove_parents, directly using the summary of the while loop at line 5 to propagate
message connection information to the caller. This will generate a summary for function remove_parents
similar as the one for the while loop. Next SherLog analyzes the for loop in main, which
finds a path error()@48 -> remove_parents@55, along with the constraint \(C_p\) as “verbose \(\neq\) 0 \&
\& rmdir()@50 = 0”. It then uses the previously computed summary of remove_parents, and further
connects message 1 with message 2 and 3 by solving the constraint \(C_p \& C_r\) along this path.

Figure 2.6 also shows the inference of path \{m_1, m_1, m_2\} in the rmdir example. This path
involves the for loop at line 45 being repeated twice. At the beginning, SherLog analyzes this loop
the first time by connecting message 2 and 3. The next time, it will first connect message 1, then
use the previously computed summary of itself to further connect 1 with 2 and 3. SherLog would
analyze the for loop one more time and find out it can no longer produce any new summaries.
Thus SherLog would stop the analysis.

![Figure 2.8: SherLog connects maximum partial message sequence.](image)

This design also makes SherLog practical to connect maximum partial paths if the complete
path is unavailable, e.g., because of multi-thread executions or system restart. Figure 2.8 shows
the case where message 1, 2 and 3 are generated by thread 1, while message 4, 5 and 6 by thread
2. The bottom-up design of SherLog still connects message 1, 2 and 3 with path 1 and 4, 5, 6 with
path 2, although SherLog couldn’t statically further connect path 1 and 2. If message 6 is the actual
error message, SherLog would report path 2 to user which is the longest path involving a message of interest.

### 2.5.2 Overview of Saturn Static Analysis System

This section provides a brief background of the Saturn Static Analysis framework that SherLog is built on before further moving into the detail implementation of SherLog. Saturn is a static analysis framework of C programs. User writes analysis in a logic programming language to express the program analysis algorithm. Saturn is summary-based, and models loops as tail-recursive function calls. It is also constraint-based, that the analysis captures all the conditions as constraints along a path that reaches the program point of interest. A SAT solver can be used to report whether these constraints are satisfiable. Saturn models all C program’s construct such as integers, structures, pointers, etc., faithfully [XA07] by statically name every memory location accessed by each statement uniquely within current function being analyzed [ABD+07]. Thus, each bit accessed by the function is represented by a distinct boolean variable. Saturn also supports alias analysis, with an option to turn it on for the analysis. Currently SherLog doesn’t turn on alias analysis, and assumes non-aliasing among pointers.

### 2.5.3 Detailed Design

SherLog formalizes the path inference problem as a constrained sequence matching problem. Given a log file $L$, let’s use a sequence of integers $M = [1, 2, \cdots n]$ to match the sequence of logging messages, an integer $i$ corresponds to the $i$th message in the log. Log parser generates a set of Logging Points for each log message:

$$\text{Possible Logging Point: } s : i \rightarrow \{l_{i,j}\}$$

where $l_{i,j}$ is a candidate Logging Point for message $i$. The problem of path inference is to find all paths $P$ in the program (function/loop call chains) that connects sequences of Logging Points $\mathcal{L}_{i,k} = [l_{i,j}, l_{i+1,j+1}, \cdots, l_{k,j_k}]$ where $[i, k] \subseteq [1, n]$, under the constraint that $P$ is a feasible path in the program. Intuitively, SherLog is searching for all the possible continuous sub-sequences of the logging messages that can be printed in the program.

SherLog’s summary-based analysis decouples the problem of searching for Logging Point sequences in the entire program into searching from function by function. The summary SherLog is computing for each function will be:

$$\text{sum}(F, i, k, P_{i,k}, C_F)$$
This tuple \( \text{sum} \) indicates that, in function \( F \), there is a path (call-chain) \( P_{i,k} \), which will connect the sequence \([i, k]\), under the constraint \( C_F \). Path Inference engine is to find all possible \( \text{sum} \) tuples for each function \( F \). All the computed summary tuples are stored in a Berkely DB database as key/value pairs \( F/\text{sum} \).

More formally, if function \( F \) calls \( F' \) at program point \( PP \), SherLog searches the summary database for the key \( F' \). For each \( \text{sum}(F', i, k, P_{i,k}, C_{F'}) \), SherLog generates a predicate \( \text{logp}(PP, i, k, P'_{i,k}, C'_{F'}) \), indicating at program point \( PP \) there is sub-sequence matched by callee, with the following rule:

\[
\text{logp}(PP, i, k, P'_{i,k}, C'_{F'}) : - \\
P P \text{ is the call site to } F', \\
\text{sum}(F', i, k, P'_{i,k}, C'_{F'}), \\
P'_{i,k} = \text{concatenate}(F'@PP, "\rightarrow", P_{i,k}), \\
\text{convert}(C_{F'}, C'_{F'})
\]

This rule propagates the sequence matching behavior of callee to the caller’s context. For the path string \( P_{i,k} \), SherLog attaches the current function name \( F' \) and call-site to the path string within \( F' \), thus recursively construct a path string such as ".remove_parents@55->while@5->error@15.". The \text{convert} predicate is to convert the constraint from callee’s context into the caller’s context. SherLog implements \textit{strongest observable necessary condition} [DDA08] for constraint conversion, which filters all the caller-unobservable conditions involving local variables and keeping only the caller-observable conditions such as return values, function arguments and globals. It guarantees the converted constraint is a necessary condition of the original one to be conservative.

With the \text{logp} defined, the sequence matching behavior of \( F \) can now be summarized as:

\[
\text{sum}(F, i_1, k_m, P_{i_1,k_m}, C_F) : - \\
PP_1, PP_2, \ldots, PP_m \text{ is a path in } F \text{ with constraint } C_p, \\
\text{logp}(PP_1, i_1, k_1, P_{i_1,k_1}, C'_{F_1}), \ldots, \\
\text{logp}(PP_m, i_m, k_m, P_{i_m,k_m}, C'_{F_m}), \\
k_1 + 1 = i_2, k_2 + 1 = i_3, \ldots, k_{m-1} + 1 = i_m, \\
P_{i_1,k_m} = \text{concatenate}(P_{i_1,k_1} \ldots P_{i_m,k_m}), \\
C_F = C_p \land C'_{F_1} \ldots \land C'_{F_m}, \text{SAT}(C_F)
\]

Here \( PP_1 \) to \( PP_m \) are call sites within \( F \) whose targets have summary entries. SherLog ignores the side effects of functions that doesn’t have summary entries, i.e., not printing logs. This rule
connects all the sub-sequences matched by $F_1$, $F_2$ to $F_m$ together to form a longer sub-sequence. Note that SherLog propagates the constraints from the callee to ensure inter-procedural path- and context- sensitivity. The \textit{sum} entry is only added to the summary database if the constraint is satisfiable, i.e., path is feasible. Note that if the sequence generated by $F_i$ and $F_{i+1}$ are discontinued, then SherLog simply ignores this path.

At the beginning SherLog initializes the \textit{logp} for each Logging Point $l_{ij}$ as:

\[
\logp(l_{ij}, i, i, \text{""}, \text{log variables}=\text{log values})
\]

The path of the initial Logging Points is empty string. Then it's a iterative process that SherLog gradually adds more summaries into the summary database. The analysis stops when the sub-sequences each function matches stabilizes. Although the sub-sequences each function generates is always guaranteed to converge, since SherLog is not backtracking along the sequence, the constraints might not. For example, a log can be analyzed infinite numbers of times but only printing the logging message in the first iteration. SherLog handles this by setting a threshold $T$ that a function/loop can be maximumly analyzed $T$ times.

\textbf{External Code Modeling:} In order to correctly model the program behavior, SherLog needs to understand the side effects of some external routines, such as \texttt{abort} and \texttt{exit}, whose source code are not available. The summary-based design eases this process that SherLog can write a \textit{summary} of the routine and stores into the summary database. Currently, SherLog manually modeled roughly 20 routines including library calls as well as system calls. These routines include \texttt{strrchr}, \texttt{stat}, \texttt{exit}, \texttt{abort}, \texttt{setjmp}, \texttt{longjmp}, etc.

An interesting case is the handling of \texttt{setjmp}/\texttt{longjmp} which are commonly used in C program to model exception handling. At each call site to \texttt{longjmp(jmp_env)}, SherLog first treats that node in the Control Flow Graph as termination node. Then it creates two \textit{summary} entries for the \texttt{setjmp(jmp_env)} call. One has the summary same as the caller of \texttt{longjmp} as it reaches the call-site, with the constraint that the return value of \texttt{setjmp} being the none-zero. The other summary entry simply indicate \texttt{setjmp} would do nothing with a zero return value. This way the control flow from the call-site of \texttt{longjmp} will be redirected to the call-site of \texttt{setjmp}, through the propagation of the context information stored in summary entry.

\subsection*{2.5.4 Reporting Inferred Paths}

There might be multiple paths inferred by the static analysis engine that connects the same Logging Point sequence. In the example shown in Figure 2.3, SherLog finds two paths main->\texttt{log@4} ->b1@10->c016->\texttt{log@25} and main->\texttt{log@4}->b2@12->c@20->\texttt{log@25}, all connecting the same
Logging Point sequence: log@4, log@25. In real world applications the scarcity of Logging Points might result in tens of thousands paths connecting two Logging points. To be useful, SherLog needs to effectively summarize these paths before output them to the user.

SherLog defines all the paths that connect the same sequence of Logging Points \( L \) as the May-Paths for \( L \). The common call-records among all these May-Paths are defined as Must-Path for \( L \). So for each \( L \), we can only have one Must-Path. In the example shown in Figure 2.3, there is one feasible Logging Points sequence log@4, log@25, with two May-Paths of length 5, and one Must-Path main->log@4..->log@25 of length 3. In real-world error execution finding Must-Path from May-Paths can effectively localize the root cause, that most of the errors can be diagnosed by looking only at Must-Path. SherLog also ranks the call-records in all the May-Paths by their frequencies.

The final output of SherLog’s path inference would be the Must-Paths that connects the longest sequence of logging messages involving the error message, along with a randomly selected May-Path for each Must-Path. By default, SherLog assumes the last message is the error message. The user could query paths involving other messages or other May-Paths. For the rmdir example, there are two Logging Points sequences, \( \{m_1, r_1, r_2\} \) and \( \{m_1, m_1, m_2\} \), so we report two Must-Paths. Since each Must-Path has only one May-Path, so in the end SherLog reports two Must-Paths same as May-Paths: main->for@45->error@48->remove_parents@55->while@5->error@15 and main->for@45->error@48->for@45->error@48->error@53. The constraints with each function/loop record along each path can be inspected.

### 2.6 Value Inference

Once the path inference engine infers an execution path \( P \), SherLog can further infer the value-flow of a variable \( v \) along \( P \) by re-executing \( P \) symbolically. Value Inference is built on top of Saturn’s memory model. Each memory location accessed by the function, e.g., local variables, globals, formal arguments, etc., is statically and symbolically named. Along each path, Saturn models the assignment relationship among memory locations as guarded points-to graph, that location \( A \) points to location \( B \) under a certain condition \( C \). Note that the points-to relationship here refers only to the relationship between the Saturn’s statically named locations, not to be confused with C program’s pointer information. A predicate \( value(PP, l, val, C) \) is used to model this behavior, indicating at program point \( PP \), location \( l \) points-to the \( val \) under constraint \( C \). Consider the following example:

1: a = argc;
2: if (c == 1)
In this program, a, argc, c and integer 1 all have static names for their location. By executing the assignment and branch instruction following C’s semantic [ABD+07], Saturn would infer that at line 4, location a points-to the location of integer constant 1 if c == 1 is true, that \textit{value}(line 4, a, 1, c == 1). Otherwise, a points to argc, that \textit{value}(line 4, a, argc, c \neq 1). We refer any constant value of a variable as concrete value.

Given a path \( P = \{F_1, F_2, ..., F_n\} \), where each \( F_i \) is either a function or loop, the value inference symbolically executes each \( F_i \) following their orders in \( P \). Within each function \( F_i \) body, it infers the guarded points-to information for all the variables accessed by \( F_i \). At the call-site in \( F_i \) to \( F_{i+1} \), SherLog propagates the context information from caller to callee. For all the \textit{value}(\( PP, l, val, C \)), where \( val \) is concrete and \( l \) is observable by \( F_{i+1} \), this information is propagated to \( F_{i+1} \) given \( C \) is satisfiable. Next when analyzing \( F_{i+1} \), SherLog initializes this points-to information, converting caller’s location to callee’s location and similar constraint conversion as in Path Inference. Thus, SherLog is propagating the constant value information along \( P \). Note that the constraint \( C \) SherLog is solving in \( F_i \) includes the constraints inferred in path inference, to guarantee that the value SherLog infers is only along the queried path.

If variable \( \text{var} \)’s value within function \( F_i \) is queried along the path \( P \), the analysis stops in \( F_i \). SherLog would output all the inferred guarded points-to information of \( \text{var} \) in \( F_i \), at each point where \( a \) is modified. SherLog also outputs any the constraints involving \( \text{var} \) if \( \text{var} \) can be found in the constraint of \( F_i \).

In the \texttt{rmdir} example, the user might query the value of variable \texttt{path} in \texttt{remove_parents}, along the path "main->for@45->error@48 ->remove_parents@55->while@5->error@15->error@21". SherLog starts with \texttt{main}, then the body of the \texttt{for} loop at line 45. At line 48, SherLog infers logging variable \texttt{dir} points-to a concrete value “dir1/dir2/”. At line 55, this information is propagated to \texttt{remove_parents} as context information. Within \texttt{remove_parents}, value of variable path is initialized as “dir1/dir2/”. At line 6, SherLog would infer \texttt{slash} points to the last ‘/’ in “dir1/dir2/” after calling \texttt{strrchr}. Then at line 11, the assignment would change the path from “dir1/dir2/” to “dir1/dir2”, removing the last slash. The output of SherLog for this query is shown in Figure 2.2.

Like Path Inference, SherLog’s Value Inference also skips the analysis on functions that doesn’t print logs, which might cause Value Inference return incorrect result. It also does not model complicated C’s features such as pointer arithmetic. Instead, SherLog tries best effort in ensuring the correctness of values involved in constraints of the path and those printed in the log messages, guaranteeing that our inference result would always conform with the constraint of the path. User can also force SherLog to analyze functions that are skipped in Path Inference. For example, in the
rmdir case user can query the return value of `rmdir()` at line 17. SherLog still answer this query by propagating constants along the path, and into the function body of `rmdir()` after the call at line 17 within the while loop. The evaluations on real world errors confirmed the effectiveness of our value inference.

<table>
<thead>
<tr>
<th>Name</th>
<th>Program</th>
<th>App. description</th>
<th>Type</th>
<th>LOC</th>
<th>Msgs.</th>
<th>Root Cause Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rmdir</td>
<td>rmdir-4.5.1</td>
<td>GNU coreutils</td>
<td>bug</td>
<td>18K</td>
<td>3</td>
<td>missing to handle trailing slashes with -p option.</td>
</tr>
<tr>
<td>ln</td>
<td>ln-4.5.1</td>
<td>GNU coreutils</td>
<td>bug</td>
<td>20K</td>
<td>2</td>
<td>missing the condition check for -target-directory option.</td>
</tr>
<tr>
<td>rm</td>
<td>rm-4.5.1</td>
<td>GNU coreutils</td>
<td>bug</td>
<td>23K</td>
<td>4</td>
<td>missing a condition check causing option -i behaves as -ir</td>
</tr>
<tr>
<td>CVS1</td>
<td>CVS-1.11.23</td>
<td>version control server</td>
<td>config. error</td>
<td>148K</td>
<td>3</td>
<td>incorrectly setting the permission for locking directory.</td>
</tr>
<tr>
<td>CVS2</td>
<td>CVS-1.11.23</td>
<td>version control server</td>
<td>config. error</td>
<td>148K</td>
<td>2</td>
<td>using wrong configuration file from a newer version.</td>
</tr>
<tr>
<td>Apache</td>
<td>apache-2.2.2</td>
<td>web server</td>
<td>bug</td>
<td>317K</td>
<td>1,309</td>
<td>incorrectly handles EOF in response stream when set up as proxy server.</td>
</tr>
<tr>
<td>Squid</td>
<td>Squid-2.3</td>
<td>web proxy cache server</td>
<td>bug</td>
<td>69K</td>
<td>197</td>
<td>Treating certain icon files wrongly by not caching them</td>
</tr>
<tr>
<td>TAR</td>
<td>tar-1.19</td>
<td>archive tool</td>
<td>bug</td>
<td>79K</td>
<td>2</td>
<td>Tar fails to update a non-existing tarball, instead of first creating it.</td>
</tr>
</tbody>
</table>

Table 2.1: Applications and real failures evaluated in our experiments. “Type” indicates the type of the root cause, either software bug or configuration error. “LOC” is the number of lines of code. “Msgs” is the number of logging messages in the failure’s log.

### 2.7 Evaluation

SherLog is evaluated on 8 real world failures from 7 applications (including 3 servers), which are summarized in Table 2.1. This suite covers a wide spectrum of representative real-world failures and applications. Six (6) of the failures are caused by software semantic bugs and 2 by configuration errors, which no prior static analysis work could diagnose.

For evaluation purpose, each failure is manually reproduced and diagnosed with the run-time log collected, and the information essential for diagnosing each failure summarized. Results generated by SherLog are compared against the summary. If SherLog could infer a subset of the summarized information it is considered *useful*. If all the information essential for diagnosing the failure are inferred correctly by SherLog, it is considered *complete*.
The experiments are conducted on a Linux machine with 8 Intel Xeon 2.33GHz CPUs, and 16GB of memory. SherLog is a single threaded program. A 30 seconds time-out threshold is set, so that each function/loop will not be analyzed more than 30 seconds.

<table>
<thead>
<tr>
<th>Name</th>
<th>Log Parser</th>
<th># Paths</th>
<th>Path Length</th>
<th>Effective</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Regex</td>
<td># Paths</td>
<td>Must</td>
<td>May</td>
</tr>
<tr>
<td>rmdir</td>
<td>4</td>
<td>10</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>ln</td>
<td>17</td>
<td>23</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>rm</td>
<td>17</td>
<td>25</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>CVS1</td>
<td>695</td>
<td>1,173</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>CVS2</td>
<td>695</td>
<td>1,173</td>
<td>1</td>
<td>120</td>
</tr>
<tr>
<td>Apache</td>
<td>997</td>
<td>1,259</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Squid</td>
<td>1,134</td>
<td>1,209</td>
<td>1</td>
<td>57</td>
</tr>
<tr>
<td>TAR</td>
<td>171</td>
<td>228</td>
<td>5</td>
<td>24</td>
</tr>
</tbody>
</table>

Table 2.2: Detailed result for SherLog. Regex is the number of regular expressions (format strings) generated by the Log Parser. “Logging points” show the number of logging points in the program that matches these regular expressions. “# Paths” is the number of each type of paths. “Path Len” is the length of the paths. Only the number of function calls is counted, with loops ignored. For multiple-must paths, the average length is reported. The length of May-Path is the length of a randomly chosen May-Path along the Must-Path. Msg is the length of the May path in terms of number of logging messages it connects, including the error message.

2.7.1 Overall Results

Table 2.2 shows the diagnostic results by the SherLog on the 8 failures. In all 8 cases, SherLog correctly and completely inferred all the diagnostic information.

Table 2.2 also shows the results of SherLog components. The Log Parser results confirm that large applications often print hundreds of or thousands of different kinds of messages, and it is common for multiple Logging Point to have the same format string (numbers in column Log Pts are bigger that those in column Regex), which makes it hard for developers to manually reason about the exponential number of Logging Point combination possibilities for a given log. In most cases, SherLog reports one Must-Path containing 2-9 function calls, which is much more precise information than the 18K-317K LOC and the exponential number of Logging Point combinations. This result demonstrates that SherLog is effective in zooming into the paths that are relevant to the failure. Table 2.2 also shows the number of logging messages SherLog connects. SherLog cannot connect Logging Points across threads (Apache and Squid), processes communicated by message passing (CVS 1 and CVS 2), or functions called by function pointer (TAR).
2.7.2 SherLog Diagnosis Case Studies

Next, I will use 3 failures as case studies to demonstrate the effectiveness of SherLog in help diagnosing bugs and misconfigurations.

Case 1: ln

<table>
<thead>
<tr>
<th>SherLog Report for ln</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Symptoms</strong></td>
</tr>
<tr>
<td>A user uses ln to create a hard link for a file, but somehow ln tries to create a hard link for the current directory &quot;.&quot; and fails.</td>
</tr>
<tr>
<td><strong>Log Traces</strong></td>
</tr>
<tr>
<td>create hard link ‘./dir1/target’ to ‘target’ [MSG 1]</td>
</tr>
<tr>
<td>ln:’.‘.: hard link not allowed for directory [MSG 2]</td>
</tr>
<tr>
<td><strong>Paths</strong></td>
</tr>
<tr>
<td>ln.c:11 for(...)</td>
</tr>
<tr>
<td>ln.c:12 do_link(...)</td>
</tr>
<tr>
<td>ln:1 main()</td>
</tr>
<tr>
<td>ln.c:11 for(...)</td>
</tr>
<tr>
<td>ln.c:12 do_link(...)</td>
</tr>
</tbody>
</table>

**Figure 2.9: SherLog report for ln in coreutils 4.5.1**

A user uses ln of coreutils 4.5.1. to create a hard link for a file, but somehow ln tries to create a hard link for the current directory (".") and fails. The log is shown in Figure 2.9, while the highly simplified code is shown in Figure 2.10.

SherLog automatically infers control flow and data flow information (the bottom of Figure 2.9) that is useful for the developer to understand the root cause of this error. SherLog infers one Must-Path from the log messages. It shows that do_link() in Line 12 was executed twice, and it failed in the second time and printed out log message [MSG 2]. It also shows that the constraints target_directory_specified || n_files > 2 must be satisfied for this error to happen. Then the developer would naturally want to know why the second file name, variable file[1], was set to the current directory. So he or she can query SherLog for this information. From the source

...
Figure 2.10: ln.c in GNU Coreutils-4.5.1. The log messages are printed in function do\_link(), whose code is not shown due to space limit.

code and the log messages, SherLog infers that file[1] is set to the current directory on Line 5 if n\_files (the number of files to be linked) is 1. Combined with the path constraints target\_directory\_specified || n\_files > 2, she or he can infer that target\_directory\_specified is set, because n\_files > 2 is not true. At this point, the developer would know the root cause. Line 5 (and the if statement from Line 2 to 9) should only be executed if no link name is specified for ln, so ln would create a hard link in the current directory (by setting file[1] to be "."). If target\_directory\_specified is set, then Line 5 should not be executed, meaning the code forgets to check if target\_directory\_specified was set at Line 2. Replacing Line 2 with if (n\_files == 1 && !target\_directory\_specified) would fix the problem.

Case 2: Squid

User experiences some missing icons on some web-pages. The log messages of Squid proxy server seem to imply that Squid thinks the missing file is too small to be cached. SherLog reports 57 May-Paths that could connect this particular message, however, only 1 Must-Path is reported(Figure 2.11). The highly simplified source code is shown in Figure 2.12.

SherLog infers the condition for the error message printing is storeCheckTooSmall() returns a non-zero value. SherLog’s value inference engine for the return value of storeCheckTooSmall() shows that storeCheckTooSmall() returns 1 when the file’s flag is of ENTRY\_SPECIAL. By this step, the developers would realize there is a typo in storeCheckTooSmall(): it should return 0 instead of 1 for these files. Note that as discussed earlier, the function body of storeCheckTooSmall() was initially skipped at the path inference stage for better scalability because it does not print any log messages. But if a user query values related to such skipped code, SherLog is able to analyze them
SherLog Report for Squid

Symptoms

End users can not see certain FTP icons if they connect the FTP server via Squid 2.3.STABLE4 [Squ].

Log Traces

Starting Squid Cache version 2.3.STABLE4 for i686-pc-linux-gnu...
... storeCheckCachable: NO: too small

Paths

storeCheckCachable() \[\rightarrow\] storeCheckTooSmall(e) \[\neq\] 0 MSG 1

Value Inference

```
#return@storeCheckTooSmall =
    { 0 if !EBIT_TEST(e->flags, ENTRY_SPECIAL) || ...
    1 if EBIT_TEST(e->flags, ENTRY_SPECIAL) || ...
```

Figure 2.11: SherLog diagnosis report for Squid 2.3.STABLE4.

```
1 int storeCheckCachable(StoreEntry * e) { ...
2    else if (storeCheckTooSmall(e)) {
3        debug(20, 2) ("storeCheckCachable: NO: too small\n");
4    } ...
5 }
6 static int storeCheckTooSmall(StoreEntry * e) {
7    if (EBIT_TEST(e->flags, ENTRY_SPECIAL))
8        return 1;
9    ...
10 }
```

Figure 2.12: store.c in Squid 2.3.STABLE4.

just as in this case study.

Case 3: CVS Configuration Error

Some users in a corporate network cannot perform any operations with CVS. The error can be difficult to diagnose because this failure affects only a portion of the users, all of which access the same repository. The result of inferred executions along with its constraints are shown in Figure 2.13.

SherLog helps to locate relevant source code which is shown in Figure 2.14. The function parse_config(), which matches keywords by calling strncmp() one by one, signals fail-
Some users in a corporate network cannot perform any CVS operations.

```
cvs [status aborted]: unrecognized auth response from whoami.
uphoria.net:
cvs pserv: /repository/CVSROOT/config: unrecognized keyword 'UseNewInfoFmtStrings'
```

Figure 2.13: SherLog report for CVS in coreutils 4.5.1

```
1 int parse_config (char *cvsroot)
2 {
3  ...   
4  while (getline (&line, ...) >= 0) {
5    ...   
6    if (strcmp (line, "RCSBIN") == 0) { ... }
7    else if (strcmp (line, "SystemAuth") == 0) { ... }
8    else if (ignore_unknown_config_keys) {
9      else {
10        error (0, 0, "%s: unrecognized keyword '%s'",
11           infopath, line);
12        goto error_return;
13      }
14    }
15  }
```

Figure 2.14: parseinfo.c in CVS 1.11.23.

When the keyword does not match any of them. Therefore, the constraints in Figure 2.13 captures all supported keywords by this particular CVS version. They imply that the keyword UseNewInfoFmtStrings is not supported since it is not in the constraints. It turns out the keyword is a new feature added in CVS 1.12. Therefore users of earlier versions should not use this option. Upgrading the CVS package solves the problem.
2.7.3 Performance of SherLog

<table>
<thead>
<tr>
<th>Name</th>
<th>Parser</th>
<th>Path</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Time</td>
<td>Memory</td>
</tr>
<tr>
<td>rmdir</td>
<td>0.02s</td>
<td>2.25m</td>
<td>174 MB</td>
</tr>
<tr>
<td>ln</td>
<td>0.02s</td>
<td>2.32m</td>
<td>194 MB</td>
</tr>
<tr>
<td>rm</td>
<td>0.01s</td>
<td>2.00m</td>
<td>511 MB</td>
</tr>
<tr>
<td>CVS1</td>
<td>0.32s</td>
<td>39.56m</td>
<td>1,317 MB</td>
</tr>
<tr>
<td>CVS2</td>
<td>0.19s</td>
<td>38.96m</td>
<td>1,322 MB</td>
</tr>
<tr>
<td>Apache</td>
<td>0.67s</td>
<td>28.38m</td>
<td>321 MB</td>
</tr>
<tr>
<td>Squid</td>
<td>0.81s</td>
<td>38.02m</td>
<td>1,520 MB</td>
</tr>
<tr>
<td>TAR</td>
<td>0.08s</td>
<td>6.55m</td>
<td>210 MB</td>
</tr>
</tbody>
</table>

Table 2.3: Performance of SherLog. “s” stands for seconds, “m” stands for minutes. Memory measures the maximum memory footprint at any given time during the execution. Value inference’s performance is measured by querying multiple relevant variable values in one pass along the inferred path.

Table 2.3 shows the performance of SherLog on each case. All diagnostic components of SherLog finishes within 40 minutes. Among the three components, Path Inference takes the longest time, because it needs to scan through the entire program, and analyzing log-related functions multiple times. Once the paths are inferred, value inference only analyzes the functions/loops along the inferred path, which greatly reduces the analysis time. Log Parser’s timing overhead is negligible.

The maximum memory consumption is mainly determined by the size of the function/loop SherLog is analyzing. For example, in CVS, function regex_compile spans more than 1,100 lines of code, resulting in more than 1GB memory usage.

2.8 Discussion and Limitations

How broadly applicable SherLog is? SherLog assumes that the logging messages are relevant to an error’s symptom and root cause. Although this assumption conforms with the motivation of logging, we found that this is not always true based on our experience. In some cases, the error symptoms are not captured by any logging message, while in some other cases, the logging messages are too general to offer path information. However, with the increasing complexity of software, developers are likely to design more informative and discriminative log messages to help them diagnose problems encountered by their customers in order to retain their customers. Thus, approaches such as SherLog would be able to help in more cases.

Lessons for better logging messages design: Although designing good logging messages for
failure diagnosis largely depends on developers’ domain knowledge and is application specific, we do observe some general guidelines can be followed. 1). Recording thread ID in concurrent programs. 2). Recording the exact location in the source code where the error message is printed out, for example, using \_\_FILE\_ and \_\_LINE\_ macros in GNU C language. 3). Recording relevant variable’s value in the error messages. How to further design better logging messages requires sophisticated analysis of the program, which remains as our future work.

**Handling logs from concurrent execution:** SherLog focuses on connecting continuous log message sequences and will stop at the boundary of two sequences of messages that were printed from different threads. Some mature software such as Apache HTTPD or Log4j [ApL] usually record thread IDs in logging messages. Therefore it is easy to separate the log messages out for each thread. SherLog can then be applied to each thread’s logs to find path and variable information in this thread. To infer information across threads is interesting and challenging, which remains as my future work.

**Locating relevant log messages:** SherLog assumes our users have certain understanding of the log to be able to identify the error symptom messages out of millions lines of various messages. The common practice in large programs is to divide logging messages into different severity levels, from informational to fatal error messages. In this case, users only need to focus on messages above certain level. In addition, current log analysis work [XHF+09] have demonstrated the effectiveness of locating a small number of suspicious log messages from millions lines of log messages. These approaches could be used together with SherLog.

**Comparing logs across applications:** SherLog works the best if the log of a failed run contains both an unambiguous symptom and enough messages along the execution for SherLog to infer the path and the root cause. For the open-source applications evaluated, none of them achieves both goals satisfiably. For server applications such as Apache, their logs are thoughtfully designed to help human to immediately know the symptom. These applications have fine-grained error messages categorization and each error message uniquely identifies a symptom. However, other than the 1 or 2 messages at the symptom site indicating the failure, Apache does not print any logging messages along the execution. On the other hand, GNU programs print out more logging messages along the execution of a program, but the error symptoms could be ambiguous or missing in the log.

   For applications that print out the file name and line number of each Logging Point, are printed in the message, there would be no need to map each log message to its Logging Points. However, the Log Parser would still need to recover the values of Logging Variables.
2.9 Summary

This Chapter describes the design and implementation of SherLog — a practical and effective diagnosis technique which can analyze logs from a failed production run and source code to automatically generate useful control-flow and data-flow information to help engineers diagnose the failure without reproducing the failure or making any assumption on log semantics. SherLog has been evaluated on 8 real world software failures, i.e., 6 bugs and 2 configuration errors, from 7 open source applications including 3 servers. For all of the 8 failures, SherLog infers useful and precise information for developers to diagnose the problems. In addition, our results demonstrates that SherLog can analyze large server applications such as Apache with thousands of logging messages within 40 minutes.
Chapter 3

Understanding Logging Effectiveness

Although SherLog can conduct deeper inference than manual efforts, it is still limited by the quality of log messages, just like manual inference by programmers. For example, if a log message does not contain enough information, SherLog and programmers will have limited starting information to disambiguate between different potential causal paths that led to a failure. Therefore, the next part of this dissertation will focus on how to improve the quality of log messages.

This Chapter will describe a characteristic study that answers the question: “how effective is the current logging practices?” Indeed, improving current logging practice will significantly benefit from a deep understanding of the real-world logging characteristics. Specifically, one first needs to assess whether the current logging practice is good enough, and if not, what are the common issues and their consequences. Not only it could provide programmers with useful guidelines and motivations for better logging, but also shed lights to new tools and programming language support for systematic logging, better testing of logging, logging improvement, etc. The next two Chapters will further describe how to improve the quality of log messages via automatic tools.

Specifically, this study will answer the questions: (i) how pervasive is software logging in reality? (ii) what is the benefit of software logging? (iii) is current logging practice good enough? (iv) if not, how are developers modifying logs?

To answer these questions, this Chapter studies logging practice in the four large, widely used open-source software systems, including Apache, OpenSSH, PostgreSQL, and Squid, each with at least over ten years of developing history. To understand the pervasiveness of logging, the density and the churn rate [ME98] of logging code are studied. As for the benefit of logging to diagnosis of production run failure, 250 real-world failure cases are sampled to compare the diagnosis time of the cases with logs and those without logs.

Answering question (iii) and (iv) is more challenging because judging the logging efficacy requires deep domain expertise. This study addresses this challenge by examining developers’ own modifications to their log messages. It systematically and automatically analyzes the revision history of each log message, and further separates those modifications that are indeed modifying problematic logging code from those merely consistency updates together with other non-logging code changes. Then it analyzes each category separately, with more focus on the former.
3.1 Overview of the Study

Table 3.1 summarizes the major findings of this study and their implications. Overall, this study makes the following contributions:

<table>
<thead>
<tr>
<th>Density of software logging (Section 3.3.1)</th>
<th>Implications</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) On average, every 30 lines of code contains one line of logging code. Similar density is observed in all the studied software.</td>
<td>Logging is pervasive during software development.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Benefit of log messages in failure diagnosis (Section 3.3.2)</th>
<th>Implications</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2) Log messages can speed up the diagnosis time of production-run failures by 2.2 times [YPH+ 12].</td>
<td>Logging is beneficial for diagnosing production-run failures.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Churns of logging code (Section 3.3.3)</th>
<th>Implications</th>
</tr>
</thead>
<tbody>
<tr>
<td>(3) The average churn rate of logging code is almost two times (1.8), compared to the entire code.</td>
<td>Logging code is being actively maintained by developers.</td>
</tr>
<tr>
<td>(4) In contrast to its relatively small density, logging code is modified in a significant number (18%) of all the committed revisions.</td>
<td>Logging code takes a significant part of software evolution despite its relatively small presence.</td>
</tr>
<tr>
<td>(5) 33% of modifications on logging code are after-thoughts. The remaining ones are updates together with other non-logging code changes within the same patch to make them consistent. More than one third (36%) of the studied log messages have been modified at least once as after-thoughts.</td>
<td>The current logging practice is ad hoc, introducing problems to the log quality. Developers take their efforts to address them as after-thoughts.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Overview of log modifications (Section 3.4)</th>
<th>Implications</th>
</tr>
</thead>
<tbody>
<tr>
<td>(6) Developers seldom delete or move logging code, accounting for only 2% of all the log modifications. Almost all (98%) of the log modifications are to the content of the log messages.</td>
<td>Developers are conservative in deleting/moving log messages, possibly due to the lack of documentations to explain the purpose of log message.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Modification to logging content (Section 3.5, 3.6, 3.7)</th>
<th>Implications</th>
</tr>
</thead>
<tbody>
<tr>
<td>(7) Developers spend significant efforts on adjusting the verbosity level of log messages, accounting for 26% of all the log improvements. Majority (72%) of them reflect the changes in developers’ judgement about the criticality of an error event.</td>
<td>Tools that systematically exposing error conditions would help testing the logging behavior. Testing and code analysis tools need to provide more information (e.g., error conditions) for developers to decide the proper verbosity.</td>
</tr>
<tr>
<td>(8) In 28% of verbosity level modifications, developers reconsider the trade-off between multiple verbosities. It might indicate that developers are often confused when estimating the cost (e.g., excessive messages, overhead) and benefit afforded by each verbosity level.</td>
<td>The scalar design of current verbosity level may not be a good way to help developers with such logging decision. More intelligent logging methods, such as adaptive logging, can help balancing the trade-offs.</td>
</tr>
<tr>
<td>(9) One fourth (27%) of the log modifications are to variable logging. Majority (62%) of them are adding new variables that are causally related to the log message. It indicates that developers often need additional runtime information to understand how a failure occurred.</td>
<td>Logging tools that automatically infer which variables to log (e.g., LogEnhancer as described in Chapter 4) can help informative logging. Given failing and passing test cases, Delta debugging [Zel02] can also be used to infer the relevant variables to log.</td>
</tr>
<tr>
<td>(10) 45% of log modifications are modifying static content (text) in log messages. More than one third (39%) of them are fixing inconsistencies between logs and actual execution information intended to record. Software can leverage programming language support to eliminate certain inconsistency, as Squid does.</td>
<td>Developers should pay more attention to update the log messages as code changes. Tools combining natural language processing and static code analysis can be designed to detect such inconsistencies.</td>
</tr>
</tbody>
</table>

Table 3.1: The major findings on real world logging characteristics and their implications.
• The first characteristic study (to the best of our knowledge) to provide the quantitative evidences that logging is an important software development practice.

• Despite the importance of effective logging, unfortunately, developers often are not able to make log messages right at the first time. Therefore, many of the log messages need to be modified as after-thoughts. By examining developers’ own modifications, this study identifies the particular aspects in logging choices (i.e., when, what and where to log) where developers spend most efforts to get them right. Such findings can shed lights on various new tools and program language and compiler supports to improve log messages.

• To demonstrate the potential benefits of the findings, a simple checker is developed to detect problematic verbosity level assignment (inspired by our Finding 8). This checker detected 138 new problematic cases in the latest version of the four studied software. 30 of them have already been confirmed and fixed by the developers as a result of our bug reports. This result confirms that our findings are indeed beneficial to tool developers to systematically help programmers to improve their log messages.

While I believe that the open-source software examined in this study well represent the characteristics of current logging practice, this study do not intend to draw any general conclusions. The findings and implications should be taken with the examined often-source software and the methodology in mind (see the discussion about threats to validity in Section 3.2).

### 3.2 Methodology

This Section describes the software projects used in this study and the study methodology. It also discusses the potential threats to validity of this study.

<table>
<thead>
<tr>
<th>Software</th>
<th>Description</th>
<th>Market share</th>
<th>LOC</th>
<th>Verbose levels</th>
<th>Log messages</th>
<th>Lines of log code</th>
<th>LOC per log</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache httpd 2.2.16</td>
<td>Web server</td>
<td>top 1</td>
<td>249K</td>
<td>8</td>
<td>1838</td>
<td>1100 (warn)</td>
<td>6758</td>
</tr>
<tr>
<td>OpenSSH 5.8p2</td>
<td>Secure shell server</td>
<td>top 1</td>
<td>81K</td>
<td>8</td>
<td>3407</td>
<td>2241 (info)</td>
<td>4672</td>
</tr>
<tr>
<td>PostgreSQL 8.4.7</td>
<td>Database server</td>
<td>top 2*</td>
<td>614K</td>
<td>13</td>
<td>6052</td>
<td>5818 (warn)</td>
<td>20733</td>
</tr>
<tr>
<td>Squid 3.1.16</td>
<td>Caching web proxy</td>
<td>top 1</td>
<td>155K</td>
<td>10</td>
<td>3474</td>
<td>1268 (info)</td>
<td>4103</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td><strong>1.1M</strong></td>
<td></td>
<td></td>
<td>14771</td>
<td>10427</td>
<td><strong>36266</strong></td>
</tr>
</tbody>
</table>

Table 3.2: Open-source software studied. The third column shows the popularity of the software in its own product category [netb, Ope, Top, Wes04]. *: Among only open-source database servers, it has the second largest market share [Top].
3.2.1 Software Used in the Study

This study focuses on four large, widely used software programs with over at least 10 years of developing history, namely Apache httpd, OpenSSH, PostgreSQL and Squid. Table 3.2 provides the descriptions. Each of them is popular as it is ranked either first or second in market share for its product’s category. The lines of code (LOC) is measured using \texttt{sloccount} [SLO], which excludes non-functional code such as comments, white-spaces, etc.

All of the software projects in this study include at least one server component. Servers are chosen because, first, their availability and reliability are often important since they tend to be used as infrastructure software providing critical services (e.g., e-commerce services), and thus many users and other applications are depending on them.

Second, runtime logs are particularly important for diagnosing server failures, which are hard to be reproduced due to the privacy and environment setting issues. They typically run for a long period of time, handle large amounts of data from users, perform concurrent execution, and are sometimes deployed in a large distributed environment forming server farms, all of which make failure reproduction and diagnosis difficult.

3.2.2 Study Methodology

Various aspects of the logging practice are included in this study. To study the density of logging code, both the lines of code (LOC) of the entire program and the logging code are measured (shown in Table 3.2). Note that the LOC of logging code is larger than the number of log printing statements since a logging statement might occupy more than one code lines.

The code churn rate is measured in \textit{Churned LOC/ Total LOC} [NB05]. The churn rate of logging code is thus measured in \textit{churned LOC of logging code / LOC of logging code}. Each revision in the recent five year’s history (2006-2011) of software is analyzed to measure the churned code. The churn rate of each year is first measured, then the average of these five one-year churn rates is taken.

To study the modifications on logging code, this study only focuses on the modifications to the log printing behavior, including verbosity levels, static content, variable values and log message location. Non-behavioral modifications, such as renaming log printing functions or verbosity levels (e.g., from \texttt{warn} to \texttt{warning}), indent changes, etc., are excluded from our analysis.

Once all the modifications to the logging code are collected, they can be divided into two types: some are merely for consistent update with other non-logging code changes \textit{within the same revision}, and others are modifying the logging behavior as after-thoughts. To separate the modification types, one possible policy is to check whether the revisions only include changes
solely to logging code but not to other code. However, this is too conservative in that developers
tend to batch multiple patches into one revision.

Instead, this study uses a few simple heuristics following the observations on the common
logging practice: developers often log right after checking a certain condition (e.g., an error con-
dition), which is usually implemented with a branch statement (e.g., if, while, etc.). In a revision,
if such a branch statement is modified together with following logging code, it may not be intro-
duced for logging adjustment, but for changing program semantic together with logging. Similarly,
if a variable or a function is renamed consistently both in the logging code and non-logging code
within the same revision, it is also not modifying the logging behavior.

Since the automatic analysis may not be accurate, I further manually verify our analysis re-
sult on 400 randomly sampled modifications (100 from each program). The manual verification
suggests the accuracy of our analysis is 94%.

This study further zooms into the details of those log modifications. For some types of such
modifications, to reason about why the previous logging was problematic or insufficient, some
modification cases are randomly sampled and their relevant source code, comments, commit log,
bug reports, and discussions in mailing list (if any) are carefully examined. If the reason cannot
be clearly understood, such cases are always conservatively classify as the “other” category when
presenting the results. The confidence interval of the sampling is reported together with the result
whenever sampling is used.

3.2.3 Threats to Validity

As with all the previous characteristics study, this study is also subject to a validity problem.
Potential threats to the validity of this characteristic study are representativeness of the software
and examination methodology.

To address the former, this study selected diverse open-source software in terms of function-
ality, including Web server, database server, caching proxy, shell server, and together with their
client utilities, all of which are widely used in their product category as shown in Table 3.2. They
have at least 10 years of history in their code repositories and more than 14771 static log points in
source code. Overall, I believe that they well represent large software which embed the current log-
ging practices. However, this study may not reflect the characteristics of logging practices in other
types of non-server/client software, such as scientific applications, operating systems, commercial
software, or software written in other programming languages.

As for the examination methodology, this study tries to minimize the authors’ own subjective
judgement on the quality of log messages by systematically analyzing developers’ own modifi-
cations to the log messages. Also, it examines developers’ commit logs, comments, related bug
reports, etc., together with the source code to reason about the modifications. Furthermore, this study includes all of the aspects typically considered by developers for logging, including verbosity level, static content and variables to record, and log placement in source code, except whether to insert a log message at the first place, a problem that will be further discussed in Chapter 5. As a limitation, for some logging problems unknown even to the developers, this study may also miss them, since the modification is not in revision history. However, if the problem is general enough, it should have been fixed in at least one of the program included in this study.

This study does not study the additions of new logging code. This is because the goal of this Chapter is to reveal issues with the current logging practices, therefore this Chapter only focuses on the modifications (including deletions) to the previously existing logging code. The problem of “where to insert new logging code” will be discussed in detail in Chapter 5.

However, adding new logging code might also reflect issues with existing logs where it is a revival of the existing logging code that has been deleted once. While this Chapter studies the deletions in such addition/deletion chains, it will miss the additions where they might have important meanings as well. However, the results in Table 3.5 and Table 3.12 suggest that the deletions of log messages rarely occur (less than 2% among all of the modifications). Therefore, I expect that such deletion/addition chains are also rare.

Overall, while I cannot draw any general conclusions that can be applied to all software logging, I believe that this study provides insights about efficacy and pitfalls of software logging, particularly in open source server applications written in C/C++.

### 3.3 Importance of Log Messages

This Section studies the pervasiveness of software logging in reality and the benefit of software logging to production-run failure diagnosis.

#### 3.3.1 Code Density of Software Logging

**Finding 1:** On average, every 30 lines of code contains one line of logging code. Similar density is observed in all the software we studied.

**Implications:** Logging is a pervasive practice during software development.

The code density of software logging is shown in the “LOC per log” column in Table 3.2. It is calculated using the LOCs of logging code and the entire code. Even in the software with least log density (Squid), there is still one line of logging code per 38 lines of code.
3.3.2 Benefit of Log Messages in Failure Diagnosis

**Finding 2 (Benefit of log messages):** Log messages reduces median diagnosis time of production-run failures between 1.4 and 3 times (on average 2.2X speedup).

**Implications:** Logging is beneficial for failure diagnosis.

250 user reported failures are randomly sampled from Apache, Squid, and PostgreSQL, etc. The failure resolution time between the set of failures where user provided *any* log messages is compared with the ones without *any* log messages. This result is borrowed from Chapter 5 (Figure 5.3), which provides a more detailed discussion on this finding and other aspects of these 250 failures. Jiang *et al.* [JHP+09] revealed a similar finding on the benefit of logging by studying production failures in commercial systems.

![Figure 3.1: Churn rates comparison between logging and entire code.](image)

3.3.3 Churns of Logging Code

**Finding 3:** As shown in Figure 3.1, the average churn rate of logging code is almost two times (1.8) compared to the entire code. Interestingly, except for PostgreSQL, all the software show that logging code have higher churn rates than the entire code base.

**Implications:** Developers are actively maintaining logging code like other non-logging code for software functionality. Logging is at least as important as other part of code in the maintenance perspective.

Such churns on logging code are also scattered across many revisions, indicating the logging code is *continuously* maintained as a significant part of software evolution:
**Finding 4:** In contrast to the relatively small density of logging code (Finding(1)), a significant number (18%) of all the committed revisions modify logging code.

**Implications:** Logging code takes a significant part of software evolution despite its relatively small presence.

Overall, Finding 3 and 4 together implicate that logging code is being continuously and actively modified. Next I will classify these modifications to understand what these modifications are.

Table 3.3: Modifications in each revision to the log messages.

<table>
<thead>
<tr>
<th>Software</th>
<th>total</th>
<th>after-thoughts</th>
<th>following other code change</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>condition</td>
</tr>
<tr>
<td>apache</td>
<td>3035</td>
<td>810 (27%)</td>
<td>1941</td>
</tr>
<tr>
<td>openssh</td>
<td>3459</td>
<td>1446 (42%)</td>
<td>1703</td>
</tr>
<tr>
<td>postgres</td>
<td>15455</td>
<td>5389 (35%)</td>
<td>9153</td>
</tr>
<tr>
<td>squid</td>
<td>5536</td>
<td>1431 (26%)</td>
<td>2951</td>
</tr>
<tr>
<td>Total</td>
<td>27485</td>
<td>9076 (33%)</td>
<td>15748</td>
</tr>
</tbody>
</table>

Table 3.3 shows the detailed classification of modifications to logging code, which is from the automatic analysis tool on committed revisions (with 94% accuracy) as described in Section 3.2.2. This tool identifies a modification as a *consistent update* with the other changes on non-logging code if the *same patch* contains one of the following three types of changes: (i) modification on the conditions that the logging code is dependent on; (ii) re-declaration of the logged variable that is also changed in logging code; (iii) modification on a function name that is also referred in the logging code as static text. Otherwise, this tool classify the modification on logging code as an after-thought. Table 3.4 shows the number of log messages that have been modified at least once as after-thoughts by these 9076 modifications.

Table 3.4: Log messages(%) that have been modified.

<table>
<thead>
<tr>
<th>log msgs</th>
<th>apache</th>
<th>openssh</th>
<th>postgres</th>
<th>squid</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>modified</td>
<td>605</td>
<td>628</td>
<td>3128</td>
<td>1106</td>
<td>5367</td>
</tr>
<tr>
<td>total</td>
<td>1838</td>
<td>3407</td>
<td>6052</td>
<td>3474</td>
<td>14771</td>
</tr>
<tr>
<td>percentage</td>
<td>40%</td>
<td>18%</td>
<td>52%</td>
<td>30%</td>
<td>36%</td>
</tr>
</tbody>
</table>
Finding 5: 33% modifications on logging code are after-thoughts. The remaining ones are consistent updates with the other changes on non-logging code in the same patch. As a result, 36% of the total 14771 log messages have been modified at least once as after-thoughts.

Implication: In current practice, logging is conducted in a subjective and arbitrary way, introducing problems to the log quality. Developers take efforts to improve them as after-thoughts.

In remainder of this Chapter, I will use modifications to only refer to these modifications that are not consistent updates with other non-logging code changes, unless otherwise specified. This Chapter focuses on studying these modifications as they are likely to reflect more directly developers’ concerns over the previously problematic log messages.

3.4 Overview of Log Modifications

In Table 3.5, it further breaks down the 9076 modifications based on what developers modified: the location of logging code within the source code, verbosity level, static content of a log message, and variables to log. For location change, this study considers the logging code’s relative location within a basic block, including both move and deletion.

<table>
<thead>
<tr>
<th>Software</th>
<th>total</th>
<th>location</th>
<th>verbosity</th>
<th>text</th>
<th>variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>apache</td>
<td>810</td>
<td>35 (4%)</td>
<td>118 (15%)</td>
<td>429 (52%)</td>
<td>228 (28%)</td>
</tr>
<tr>
<td>openssh</td>
<td>1446</td>
<td>33 (2%)</td>
<td>550 (38%)</td>
<td>264 (18%)</td>
<td>599 (41%)</td>
</tr>
<tr>
<td>postgres</td>
<td>5389</td>
<td>17 (1%)</td>
<td>1148 (21%)</td>
<td>3000 (56%)</td>
<td>1224 (23%)</td>
</tr>
<tr>
<td>squid</td>
<td>1431</td>
<td>65 (5%)</td>
<td>573 (40%)</td>
<td>364 (25%)</td>
<td>429 (30%)</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>9076</td>
<td>150 (2%)</td>
<td>2389 (26%)</td>
<td>4057 (45%)</td>
<td>2480 (27%)</td>
</tr>
</tbody>
</table>

Table 3.5: Type of log modifications as after thoughts.

Finding 6: Developers seldom delete or move the logging code, accounting for only 2% of all the log modifications. Almost all (98%) modifications are to the content of the log messages, namely verbosity level, static text and variables.

Implications: Given the lack of specifications on logging behaviors, developers would not delete/move log messages unless they cause serious problems (Section 3.8).
<table>
<thead>
<tr>
<th>software</th>
<th>total</th>
<th>error</th>
<th>non-error</th>
</tr>
</thead>
<tbody>
<tr>
<td>apache</td>
<td>118</td>
<td>84 (71%)</td>
<td>34 (29%)</td>
</tr>
<tr>
<td>openssh</td>
<td>550</td>
<td>398 (72%)</td>
<td>152 (28%)</td>
</tr>
<tr>
<td>postgres</td>
<td>1148</td>
<td>831 (72%)</td>
<td>317 (28%)</td>
</tr>
<tr>
<td>squid</td>
<td>573</td>
<td>399 (70%)</td>
<td>174 (30%)</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>2389</strong></td>
<td><strong>1712 (72%)</strong></td>
<td><strong>677 (28%)</strong></td>
</tr>
</tbody>
</table>

Table 3.6: Verbosity-level modifications with error event logging and with non-error event logging

### 3.5 Verbosity Levels Modification

This Section analyzes the 2389 modifications to verbosity levels (Table 3.5), which indicate developers often do not assign the right verbosity level at the first time.

In Table 3.6, it further breaks down the verbosity level modification into those for error event logging and non-error event logging. In the former (72%), at least one verbosity level before or after the modification is an error-level (e.g., `error`, `fatal`, etc.). These indicate that developers might have misjudged *how critical the event to log is* at the first place. Please recall that in these modifications developer *did not change the conditions* (typically the error condition) leading to the log messages, but only the verbosity level themselves, indicating they are likely *after-thoughts*. In the other 28% verbosity level modifications, developers change between non-error (also non-fatal) levels, such as `info` and `debug`, which are supposedly to record non-error events.

**Finding 7:** Majority (72%) of the verbosity-level modifications reflect the changes in developers’ judgement about the criticality of an error event (Table 3.6).

**Implications:** Tools that systematically exposing error conditions would help test the logging behaviors. For example, fault injection tools [HTI97] can be used to inject faults to trigger an error and consequently its error logging. Similarly, software model checking [BR02] can be extended to explore the execution paths reaching the logging code.

### 3.5.1 Reconsidering Error Criticality

Table 3.7 breaks down the verbosity-level modifications for error event logging. More than half (56%) of the cases are changing levels between non-fatal and fatal. This class is different from others in that they are introduced to change the system’s execution behavior as well as logging behavior. Specifically, with the modifications, developers changed their decision either to enforce a system to abort after logging, or allow it to continue to run.

The modification from a non-fatal to a fatal level is to prevent a non-survivable error from propagating, which can lead to serious system malfunctions or security issues. On the other hand,
the modification from a fatal to a non-fatal level is to avoid an unnecessary system termination on a survivable error for better system availability.

For example, in Figure 3.2 (A), PostgreSQL developers originally record an error event (i.e., an access to an uninitialized pointer) at ERROR level, which could potentially introduce security holes if not aborted right away. Later they provide this patch only to promote the level to a PANIC (fatal in this software) that will abort the entire database back-end. As an opposite example, in Figure 3.2 (B), PostgreSQL developers prevent non-critical cases from taking down the entire database by demoting the original PANIC to ERROR.

In Table 3.7, some others (38%) are changing verbosity levels between an error level and non-error levels. In those cases, developers may reconsider their original judgements about whether the event to record is an error or not, because recording a real error with non-default verbosity level such as debug would cause missing important error messages for failure diagnosis, and recording
a non-error event with error level might either confuse the users and developers or cause unnecessary production run overhead. For example, Figure 3.2 (C) shows that PostgreSQL developers originally missed to report a configuration error by logging it with info which is not a default verbosity level for production run in PostgreSQL. After suffering from diagnosing it without logs, they committed a patch only to change it to error.

3.5.2 Reconsidering Logging Trade-offs

As shown in Table 3.6, 28% of the verbosity-level modifications come from non-error event logging. In general, non-error events are logged with one of multiple verbose levels such as debug1, debug2,..., or sometimes even with a default levels, e.g. info in Squid. Table 3.8 decomposes the verbosity modifications for non-error events.

<table>
<thead>
<tr>
<th>Software</th>
<th>between verbose</th>
<th>verbose to default</th>
<th>default to verbose</th>
<th>between default</th>
</tr>
</thead>
<tbody>
<tr>
<td>apache</td>
<td>23 (67%)</td>
<td>3 (9%)</td>
<td>8 (24%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>openssh</td>
<td>116 (76%)</td>
<td>11 (7%)</td>
<td>25 (16%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>postgres</td>
<td>132 (42%)</td>
<td>108 (34%)</td>
<td>59 (19%)</td>
<td>18 (5%)</td>
</tr>
<tr>
<td>squid</td>
<td>115 (66%)</td>
<td>38 (22%)</td>
<td>21 (12%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>Total</td>
<td>386 (57%)</td>
<td>160 (24%)</td>
<td>113 (17%)</td>
<td>18 (3%)</td>
</tr>
</tbody>
</table>

Table 3.8: Reconsideration of logging trade-off and verbosity-level modification

**Finding 8:** For non-error event logging, developers reconsider the trade-off between multiple verbosity levels. It might indicate that developers are often confused when estimating the cost (e.g., excessive messages, logging overhead) and benefit afforded by each verbosity level.

**Implications:** The scalar design of current verbosity level may not be a good way to help developers with such logging decision. Adaptive logging in runtime, similar to adaptive sampling [HC04], can help balancing the trade-off by dynamically backing-off the logging rate.

In Table 3.8, more than half (57%) of the non-error verbosity level modifications are changing between two verbose levels. In all the studied software, verbose levels are not enabled by default, meaning that they are mostly used during in-house testing. Therefore, the logging overhead may be less of concern when developers make such adjustments. Instead, developers probably are more concerned about balancing the amount of logs: too excessive logging would rather make noises for failure diagnosis, but insufficient logging would miss important runtime information.
One of the possible causes for such many adjustments within verbose levels might be because no clear division among multiple verbose levels is given in terms of purpose of use, benefit, and cost, resulting in confusing developers when deciding among the verbose levels. For example, in Squid, there are 7 debug levels out of total 10 verbosity levels, but no guidance for which cases they should be used. Indeed, in their header file, developers wrote a comment saying “level 2-8 are still being discussed amongst the developers”. This study surmises that developers would decide which level to use arbitrarily at the first place and often revisit the decision later.

Figure 3.3: Example from PostgreSQL of a verbosity level demotion from default level (non-erroneous) to verbose level, together with developers’ commit log.

For non-error logging with default level (e.g., bookkeeping with info), developers may need to carefully consider more factors since it would directly affect production run. For example, Figure 3.3 shows that PostgreSQL users complained about the excessive log messages, and thus developers demote the previous default level (LOG) to verbose level (DEBUG). Interestingly the developers originally assigned a default level because the event was in some new code that potentially is buggy, but it resulted in excessive logging at a user site. In addition, of course developers may need to consider logging overhead in production run.

Overall, setting the verbosity level by considering all those factors may not be easy at the first place, or need further adjustment as software and environment changes. Unfortunately, the current scalar design of verbosity level does not provide enough information for developers. To help developers, systematic and dynamic logging tools or assists, such as adaptive logging [HC04], are needed. Instead of using a statically assigned verbosity, adaptive logging exponentially decreases logging rate when a certain logging statement is executed many times, only recording its \(2^n\) dynamic occurrences. Such strategy will reduce both the amount of logs and performance overhead, while preserving the first several occurrences of each log message.

### 3.6 Modifications to Variable Logging

Table 3.9 shows how developers improved variable logging. Majority of them are adding new variables to original logging code, which could provide more dynamic information for failure
<table>
<thead>
<tr>
<th>Software</th>
<th>Total</th>
<th>add var.</th>
<th>replace var.</th>
<th>delete var.</th>
<th>change format(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>apache</td>
<td>228</td>
<td>81</td>
<td>68</td>
<td>15</td>
<td>64</td>
</tr>
<tr>
<td>openssh</td>
<td>599</td>
<td>348</td>
<td>106</td>
<td>24</td>
<td>121</td>
</tr>
<tr>
<td>postgres</td>
<td>1224</td>
<td>839</td>
<td>184</td>
<td>102</td>
<td>99</td>
</tr>
<tr>
<td>squid</td>
<td>429</td>
<td>278</td>
<td>45</td>
<td>26</td>
<td>80</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>2480</td>
<td>1546 (62%)</td>
<td>403 (16%)</td>
<td>167 (7%)</td>
<td>364 (15%)</td>
</tr>
</tbody>
</table>

Table 3.9: Modifications to improve variable logging. *: e.g. from integer format to float format.

diagnosis. For example, in Figure 3.4, a user of PostgreSQL reported a production-run failure with an error message printed by the software. Unfortunately, developers could not diagnose the failure due to the lack of runtime information. Only after a couple of rounds of back-and-forth discussion with users they resolved this. From the lessons, later they committed a patch only to a new variable causally-related to the logging point.

*Bug Report from user:*

**User:** Error when setting client encoding to UTF-8, with error message: 
   *failed to commit client_encoding*

**Dev:** Cannot reproduce the bug… Asking for more details…

... ... ... ...

**Dev:** Fixed the bug. **Motivated by this report, should always include the parameter value we failed to assign.**

**Patch:**

if (!(*conf->assign_hook) (newval, true, PGC_S_OVERRIDE))
  - elog(ERROR, "failed to commit %s", conf->gen.name);
  + elog(ERROR, "failed to commit %s as %d", conf->gen.name, **newval**);

Figure 3.4: Example of adding variable values to log message.

**Finding 9:** One fourth (27%) of the log modifications are to variable logging. Majority (62%) of them are adding new variables that are causally related to the log message. It indicates that developers often need additional run-time information to understand how a failure occurred.

**Implication:** Logging tools that automatically infer which variables to log (e.g., Log-Enhancer that will be described in Chapter 4) can help informative logging. Given failing and passing test cases, Delta debugging [Zel02] can be used to log those variable values that are specific to a failing run.

Interestingly, once variables are introduced into logging code, they are seldom (7%) deleted, as shown in Table 3.9. Probably it is because recording unnecessary variables often would not introduce serious concerns besides one or two useless variable values in the log.
However, there can also be certain variables that should not be logged, considering security and privacy concerns, and developers should be careful to avoid them. Figure 3.5 shows that Apache developers deleted a variable including a client’s URI from the logging code, since recording it could “make the server vulnerable to denial-of-service attack”.

```
Patch:
ap_log_cerror(APLOG_INFO, 0, c,
    "Connection to child %ld closed with %s shutdown"
- "(client %s)", c->id, type,
+ c->id, type);
```

```
Commit log:
"It is VERY IMPORTANT that you not log any raw data from the network, such as the request-URI or request header fields. Doing so makes the server vulnerable to a denial-of-service attack."
```

Figure 3.5: Logging a wrong variable causing Apache vulnerable to denial of service attack.

<table>
<thead>
<tr>
<th>wrong var.</th>
<th>inconsistency</th>
<th>readability</th>
<th>redundancy</th>
<th>other</th>
</tr>
</thead>
<tbody>
<tr>
<td>46% (±6%)</td>
<td>11 (±4%)</td>
<td>23 (±5%)</td>
<td>2 (±2%)</td>
<td>18 (±5%)</td>
</tr>
</tbody>
</table>

Table 3.10: Variable replacement and deletion. The margin of errors are shown at 95% confidence level.

To further understand why variables are deleted or replaced in the logging code, I manually study 154 such modifications that are randomly sampled. As a result, Table 3.10 shows that (i) as the most dominant case, the original logging code records wrong variables at the first place, either only by mistakes or by not being aware of security or privacy concerns; (ii) other non-logging code was evolved but the variable logging was not updated together, becoming inconsistent; (iii) an error number such as errono was printed without interpretation, requiring replacement to readable description; (iv) a log message includes redundant variables, preferred to be deleted. The remaining cases, where I cannot understand the modifications from their source code, commit logs, or comments, belong to the “other” category.

### 3.7 Modifications to Static Content

45% of the log modifications are modifying the static content (text) in log messages. In general, static content (text) in log messages can include the description of an event to record, the information of source code location (e.g., function and file name, line number) where the logging is conducted, and so on. Since it is challenging to automatically analyze the text written in natural language, I randomly sampled 200 modifications and studied them, which are shown in Table 3.11.
Inconsistency

<table>
<thead>
<tr>
<th>Inconsistency</th>
<th>Clarification</th>
<th>Spell/grammar</th>
<th>Incorrect content</th>
<th>Others</th>
</tr>
</thead>
<tbody>
<tr>
<td>39% (±6.6%)</td>
<td>36% (±6.5%)</td>
<td>18% (±5.2%)</td>
<td>5% (±2.9%)</td>
<td>2% (±1.9%)</td>
</tr>
</tbody>
</table>

Table 3.11: Improving static content of log messages. The margin of errors are shown at 95% confidence level.

In some cases (39%), developers modified the out-of-date log messages that are inconsistent with the actual execution information, which could mislead and confuse the developers or users (please note that those consistent updates of both log and code in the same patch are excluded from our analysis by our analysis tool). Majority (76%) of them are related to function name changes. For example, in Figure 3.6, OpenSSH developers changed a function name from “restore_uid” to “permanently_set_uid” but forgot to update the logging code to record this name. Later, they were confused with the out-of-date log message while trying to resolve a failure. Finally, they committed this patch just to fix the inconsistent logging code.

```c
permanently_set_uid(struct passwd *pw) {
    if (temporarily_use_uid_effective)
        fatal("permanently_set_uid: temporarily_use_uid effective");
    + fatal("permanently_set_uid: temporarily_use_uid effective");
}
```

Figure 3.6: Example of inconsistency between log messages and the code in OpenSSH. This patch is just to fix this inconsistency.

Such inconsistency can be partly avoided by using programming language support. For example, C programming language provides a macro “\_FUNCTION\_” as part of the ANSI-C99 standard, which holds the function name within which the code is currently executing. As a good logging practice, as shown in Figure 3.7, Squid started to use this in its logging code to automatically recognize a function name and log it. This eliminates the need for developers to manually record or update a location information, avoiding the inconsistent update problem at the first place.

```c
/* HERE is a macro that you can use like this:
 * debugs(1, HERE << "some message"); */
#define HERE __FILE__"("<< __LINE__ <<") "<< __FUNCTION__"<<": "
if (fd < 0) {
    - debugs(3, "BlockingFile::open: failure (" << errno << ")");
    + debugs(3, HERE "": got failure (" << errno << ");
```

Figure 3.7: The use of the programming language support to hold location information for log messages in Squid.

To detect other inconsistent updates (e.g., an event to log and its description), it would be
beneficial to use natural language processing together with static source code analysis, similar to iComment [TYKZ07] which uses natural language processing to automatically analyze comment and source code in order to detect inconsistency.

**Finding 10:** More than one third (39%) of modifications to static content are fixing inconsistency between logs and actual execution information intended to record. Software can leverage programming language support to eliminate some of the inconsistency, as Squid does.

**Implication:** Tools combining natural language processing and static code analysis can be designed to detect such inconsistency.

---

**Bug Report from user: Confusing message in log file**

"I changed the postgresql.conf file, and see the following messages:

<table>
<thead>
<tr>
<th>configuration file change ignored</th>
</tr>
</thead>
</table>

so I expect both newly enabled "archive_command" and "shared_buffers" not to take effect…. But in fact, "archive_command" does take effect."

**Patch:**

ereport (ERROR,
- "parameter "%s" cannot be changed after server start;",
- gconf->name
- "configuration file change ignored",
+ "at attempted change of parameter "%s" ignored.",
+ gconf->name
+ "This parameter cannot be changed after server start"

---

Figure 3.8: Example of a log message clarification from PostgreSQL.

In some other cases (36%), developers modified static content of log messages to clarify the event description in it. As an example, Figure 3.8 shows that a log message in PostgreSQL was unclear and thus it misled a user to believe that all his configuration changes would lose effect, which was not true. At the end, the modification was made only to clarify the content of the log message.

### 3.8 Location Change

As discussed in Finding 6, developers seldom delete or move logging code once it is written. To understand under which cases developers delete/move logging code, 57 such cases are further randomly sampled from the 150 location modifications and manually examined. The Table 3.12 summarizes the results with the sampling errors at the 95% confidence level.
Interestingly, 26% of the location changes were required because the original logging code was misplaced and resulted in software failures. For example, logging in signal/interrupt handlers is dangerous since the non-reentrant I/O operations during logging might corrupt system states and open up vulnerabilities. Figure 3 shows a patch to delete such a problematic logging code from a signal handler in Squid. In addition, logging variables before their initialization would result in system crash or misbehavior; logging non-error events with "fatal" verbosity level will unnecessarily terminate the software execution. To identify these problems above, in-house testing tools and static analysis tools [ECC] can be extended to explore logging place. For example, developers can use static analysis to detect logging statements within interrupt handlers and the use of uninitialized variables.

In some cases, developers delete some misleading log messages (e.g., an error message printed under a non-error situation). From several commit logs, it is shown that some developers tend to actively log certain events simply with the "error" verbosity level for the purpose of in-house testing, then forgot to completely delete them before the production release. In other cases, log messages are moved out of a loop body or combined into one that can summarize them, probably in order to avoid overhead and noises from excessive logging. Finally, the “others” category includes the cases that cannot be clearly understood.

### 3.9 Verbosity Level Checker

To show the feasibility of automatic logging assistance from the findings of this study, I designed a simple verbosity-level checker which identifies certain type of problematic verbosity assignment. This is motivated by the significant number of verbosity-level adjustments (Finding 7).

This checker is based on the observation that if the logging code within the similar code snippets have inconsistent verbosity levels, at least one of them is likely to be incorrect [ECH+01, LLMZ04, GYY+10, YMNCC04]. First, the tool identifies all the code clones in the source code (Table 3.12: Reasons for moving or deleting log messages

<table>
<thead>
<tr>
<th>Reason</th>
<th>Percentage</th>
<th>Variation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software failure</td>
<td>26% (±9%)</td>
<td></td>
</tr>
<tr>
<td>Misleading log msg.</td>
<td>21% (±8%)</td>
<td></td>
</tr>
<tr>
<td>Reduce noises</td>
<td>40% (±10%)</td>
<td></td>
</tr>
<tr>
<td>Others</td>
<td>12% (±7%)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.9: Deleting Logging from a signal handler in Squid.
CP-Miner [LLMZ04] is used to detect code clones. Then, it further checks each pair of clones to see whether they contain logging code and their verbosity levels are consistent.

<table>
<thead>
<tr>
<th>Inconsistency</th>
<th>apache</th>
<th>openssh</th>
<th>postgres</th>
<th>squid</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>4</td>
<td>89</td>
<td>33</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.13: Verbosity-level inconsistency detected.

As a result, this checker detected 138 inconsistent pairs of logging code, as shown in Table 3.13. I reported 45 cases to the developers. 30 of them are confirmed and fixed, 10 are confirmed as false positives where the cloned code snippets are in different contexts so they should have different verbosity levels, and the remaining ones are not being responded.

This result shows that based on the finding, even a simple checker can effectively help for better logging. It confirms that the first important step towards systematic and automatic supports for better software logging is to understand the current manual efforts for logging, which is exactly the goal of this work.

### 3.10 Summary

This Chapter describes a study that characterizes the practice of software log messages using four pieces of large open-source software. It first quantifies the pervasiveness and the benefit of software logging. By further studying developers’ own modifications on their log messages, it is shown that they often cannot get the log messages right after the first attempts. In particular, developers spend significant efforts in modifying the verbosity level, static text, and variable values of log messages in various ways, but rarely change the message locations. By identifying these common log-modification efforts that are done manually, we reveal many opportunities for tools, compiler and programming language support to improve the current logging practices. Such benefit of the findings is confirmed by a simple checker, which is motivated by identifying developers’ large amount of manual efforts in modifying the verbosity level, that can effectively detect 138 new pieces of problematic logging code.

In the next Chapter, I will describe another tool, LogEnhancer, that is also motivated by the result from this study.
Chapter 4

Improving Log Quality I: Enhancing Existing Logging Statement

In Chapter 3, we have seen the inefficiencies of current log messages. In particular, it shows that log messages often do not contain sufficient information — developers from four software projects have spent large efforts (in the form of 1546 patches) just to add additional variable values. If a log message does not contain enough information, developers or automatic log inference engines such as SherLog will have limited starting information to disambiguate between different potential causal paths that led to a failure.

![Figure 4.1: Example of real-world patches just for the purpose of enhancing log messages.](image)

Figure 4.1: Example of real-world patches just for the purpose of enhancing log messages.

At its essence, the key problem is that existing log messages contain too little information. Despite their widespread use in failure diagnosis, it is still rare that log messages are systematically designed to support this function. In many cases, logging statements are inserted into a piece of software in an ad hoc fashion to address a singular problem. For example, in many cases, an error log message may simply contain “system failed” without providing any further context for diagnosis. While there are a number of “rules of thumb” for designing better logging messages (e.g., such as logging the error symptoms [Sch] and the thread ID with each message, these still do not capture the specific information (e.g., state variable values) that are frequently necessary to infer a problem’s root cause. Instead, developers update log messages to add more information as they discover they need it. Figure 4.1 shows three such enhancements, each of which expanded the log messages to capture distinct pieces of diagnostic state.

This Chapter describes a tool called LogEnhancer, that can systematically and automatically
add such enhancements to log messages, and thereby improve the diagnostic power of logging in general. LogEnhancer modifies each log message in a given piece of software to collect additional causally-related information to ease diagnosis in case of failures\(^1\). To be clear: LogEnhancer does not detect bugs nor failures itself. Rather it is a tool for reducing the burden of failure diagnosis by enhancing the information that programmers should have captured when writing log messages. Such additional log information can significantly narrow down the amount of possible code paths and execution states for engineers to examine to pinpoint a failure’s root cause.

In brief, LogEnhancer enhances log content in a very specific fashion, using program analysis to identify which states should be captured at each log point to minimize causal ambiguity. In particular, the “uncertainty” around a log message reflects the control-flow paths or data values that are causally-related but cannot be inferred from the original log message itself. Using a constraint solver LogEnhancer identifies which candidate variable values, if known, would resolve this ambiguity. Note LogEnhancer is not trying to disambiguate the entire execution path leading to the log message. For example, a branch whose directions have no effect for the execution to reach the log message will not be resolved since it is not causally-related.

LogEnhancer explore two different policies for deciding how to collect these variable values: delayed collection, which captures only those causally-related key values that are “live” at the log point or can be inferred directly from live data, and in-time collection, which, in addition to those recorded in delayed collection, also records historic causally-related key values before they are overwritten prior to the log point. The latter approach imposes additional overhead (2-8% in our experiments) in exchange for a richer set of diagnostic context, while delayed collection offers the reverse trade-off, annotating log messages with only variable values “live” at log points, while imposing minimal overhead (only at the time an existing message is logged). LogEnhancer also has a variant of the delayed collection method that derives equivalent information from a core dump (thus allowing a similar analysis with unmodified binaries when core files are available).

Finally, LogEnhancer is applied on 8 large, real-world applications (5 servers and 3 client applications). The evaluation results show that LogEnhancer automatically identifies 95% of the same variable values that developers have added to their log messages over time. Moreover, it identifies an additional set of key variable values (10-22) which, when logged, dramatically reduce the number of potential causal paths that must be considered by a factor of 35. Furthermore, 15 representative, real-world failures (with 13 caused by bugs and 2 caused by mis-configurations) from the above applications are selected to demonstrate how the enhanced log messages can help diagnosis. In all these cases, the enhanced log messages would quickly reduce the number of possibilities of partial execution paths and run-time states, helping both manual diagnosis and automatic\(^1\)

\(^1\)LogEnhancer targets for production failure diagnosis even though it can also be useful for in-lab debugging.
log inference engines like SherLog to narrow down and identify the root causes. Finally, the evaluation results show that both log size and run-time overhead are small, and almost negligible (with the delayed collection).

4.1 Overview of LogEnhancer

```
1 int remove_entry (char *filename, struct dirent *dp){
2   # ifndef __GLIBC__
3   struct stat sbuf;
4   if (dp)
5       is_dir = (dp->d_type == DT_DIR) ? T_YES : T_NO;
6   else {
7       if (lstat (filename, &sbuf))
8           ...
9       is_dir = S_ISDIR (sbuf.st_mode) ? T_YES : T_NO;
10   }
11
12   if (is_dir == T_NO) {
13       if (unlink(filename) == 0)
14           return RM_OK;
15       error (0, errno, "cannot remove \%s", filename);
16       return RM_ERROR;
17   }
18 #endif
19   return RM_NONEMPTY_DIR;
20 }
21
22 int remove_cwd_entries (...){
23   if ((dp = readdir (dirp)) == NULL) { return; }
24   tmp_status = remove_entry (f, dp);
25 }
26
27 int rm (...){
28   status = remove_entry (filename, dp);
29 }
30```

Figure 4.2: Highly simplified code for rm in coreutils-4.5.4. Different colors highlight which information can be inferred given the log message, for example, “Must-Execute” reflects code paths that can be completely inferred based on the given log message. Variable values that cannot be inferred are also highlighted.

To explain how LogEnhancer works, let’s first examine how diagnosis is performed manually today. Figure 4.2 shows a simplified version of a real world failure case in the rm program from the GNU core utilities. This is a particularly hard-to-diagnose failure case since it has complex environmental requirements and only manifests on FreeBSD systems using NFS that do not have GLIBC installed. In particular, when executing `rm -r dir1` for an NFS directory dir1 in such an environment, rm fails with the following error message:

```
rm: cannot remove ‘dir1/dir2’: Is a directory
```
4.1.1 Manual Diagnosis

Upon receiving such a failure report, a support engineer’s job is to find the “log point” in the source code and then, working backwards, to identify the causally-related control flow and data flow that together could explain why the message was logged. Pure control flow dependencies are relatively easy to reason about, and upon inspection one can infer that the error message (printed at line 16) can only be logged if the conditional at line 12 (\texttt{is\_dir == T\_NO}) is taken and the conditional at line 13 (\texttt{unlink(filename == 0)}) is not taken. This suggests that \texttt{rm} treated \texttt{filename} (dir1/dir2 in this case) as a non-directory and subsequently failed to “unlink” it. Indeed, purely based on control flow, one can infer that lines 14–15, and 20–22 could not have been executed (highlighted in Figure 4.2 as “Must-Not-Execute”), while lines 1–4, 11-13, and 16–19 must have been executed (similarly labeled in the figure as “Must-Execute”). Already, the amount of ambiguity in the program is reduced and the only remaining areas of uncertainty within the function are on lines 5–10, and lines 23-32 (also highlighted in Figure 4.2 as “May-Execute”).

However, further inference of why \texttt{is\_dir} equals \texttt{T\_NO} is considerably more complicated. There are two possibilities for the branch at line 4, depending on the value of \texttt{dp}, and both paths may set \texttt{is\_dir} to be \texttt{T\_NO}. Further, since \texttt{dp} is a parameter, we must find the caller of \texttt{remove\_entry}. Unfortunately, there are two callers and we are not sure which one leads to the failure. In other words, given only the log message, there remain several uncertainties that prevent us from diagnosing the failure. Note that this challenge is not a limitation of manual diagnosis, but of how much information is communicated in a log message. Section 4.3.2 will show that automatic log inference engines such as SherLog can do no better than manual inference in this case.

In addition to control flow, backward inference to understand a failure also requires analyzing data flow dependencies, which can be considerably more subtle. We know from our control flow analysis that the conditional at line 12 is satisfied and therefore \texttt{is\_dir} must equal \texttt{T\_NO}. However, \texttt{why} \texttt{is\_dir} holds this value depends on data flow. Specifically, the value of \texttt{is\_dir} was previously assigned at either line 5 or 9, and has data dependencies on either the value of \texttt{dp\_t\_type} or \texttt{sbuf\_st\_mode}, respectively. Determining which dependency matters goes back to control flow: which branch did the program follow at line 4?

Unfortunately, the error message at line 16 simply does not provide enough information to answer this question conclusively. The conditional at line 4 is uncertain — either path (line 5, or line 7 to 10) could have been taken (indicated as “may-execute” in Figure 4.2). Similarly, the values of \texttt{dp\_t\_type} and \texttt{sbuf\_st\_mode} are also uncertain, as is the context in which \texttt{remove\_entry()} was called. While the amount of ambiguities is modest in this small example, it is easy to see how the number of options that must be considered can quickly explode when diagnosing a system of any complexity.
Figure 4.3: Remaining uncertainty if dp was printed at line 16. The code is the same as Figure 4.2.

However, a complete execution trace is not necessary to resolve this uncertainty. Indeed, if the program had simply included the single value of dp in the logging statement at line 16, the situation would have been far clearer (it is illustrated in Figure 4.3, given this new information). In this case dp is non-zero, and thus the code at line 5 is now in a “must-execute” path, while lines 6–10 “must not” have executed. In turn, it removes the need to consider the value of sbuf.st_mode since is_dir can now only depend on dp->d_type.

The remaining uncertainties then include: (1) which function (remove_cmd_entries or rm_1) called remove_entry? (2) What was the value of dp->d_type at line 5? Resolving these would require logging some additional information such as the call stack frame, and dp->d_type (or, some equivalent value that can be used to infer dp->d_type’s value at line 5; this optimization will be discussed in Section 4.2.2).

If this ambiguous error was reported frequently, developers might add exactly these values to the associated log statement to aid in their diagnosis. However, relying on this, software development cycle is both slow and iterative, and is unlikely to capture such state for rare failure modes. The goal of LogEnhancer is to automate exactly the kind of analysis we described above — identifying causally-related variable values for each “log point” and enhancing the log messages to incorporate these values. Moreover, because it is automatic, LogEnhancer can be applied com-
prehensively to a program, thereby capturing the information needed to diagnose unanticipated failures that may occur in the future.

4.1.2 Usage

LogEnhancer is a source-based enhancement tool that operates on a program’s source code and produces a new version with enhanced data logging embedded. It can be used to enhance every existing log message in the target software’s source code or to enhance any newly inserted log message. The only real configuration requirement is for the developer to identify log points (i.e., typically just the name of the logging functions in use). For example, the cvs revision control system uses GLIBC’s standard logging library `error()` and simply issuing

```
LogEnhancer --logfunc="error" CVS/src
```

is sufficient for LogEnhancer to do its work.

Upon being invoked, LogEnhancer leverages the standard make process to compile all program source code into the CIL intermediate language [NMRW02], then identifies log points (e.g., statements in cvs that call `error()`), uses program analysis to identify key causally-related variables, instruments the source code statically to collect the values of these variables and then re-compiles the modified source to generate a new binary.

During production-runs, when a log message is printed, the additional log enhancement information (variable values and call stack) will be printed into a separate log file. LogEnhancer can also be optionally configured to record additional log enhancement information only when `error` messages are printed.

In the `rm` example, at the log point at line 16, the following information will be collected: (1) `dp`: helps determining the control flow in line 4; (2) The call stack: helps knowing which call path leads to the problem; (3) `dp->d_type` or `sbuf.st_mode` depending on the value of `dp` helps determining why `is_dir` was assigned to `T_NO`; (4) `filename`: since it’s used in `unlink` system call, whose return value determines the control flow to log point at line 16; (5) `dirp` in function `remove_cwd_entries` if this function appears on the call stack.

During diagnosis, LogEnhancer’s enhancement result can be manually examined by developers along each log message, or can be fed to automatic inference engines such as SherLog, which automatically infer execution paths and variable values. Section 4.3.2 shows three such examples.

4.1.3 Architecture Overview

The complexity in LogEnhancer is largely in the analysis, which consists of three principal tasks:
1. **Uncertainty Identification**: This analysis identifies “uncertain” control-flow and variable values that are causally-related and whose state could not be resolved using only the original log message. Starting from each log point and working backwards, we identify the conditions that *must* have happened to allow control flow to each log point (e.g., `is_dir == T` and `unlink(filename)` are such conditions in `rm`). Using these conditions as clues, we continue to work backwards to infer *why* these conditions occurred through data-flow analysis (e.g., `dp`, `dp->d_type` and `sbuf.st_mode` are identified through data-flow analysis of `is_dir`). This process is repeated recursively for each potential caller as well (e.g., the data dependency on `dirp` from `remove.cwd.entries` is identified in this step). To prune out infeasible paths, *LogEnhancer* uses a SAT solver to eliminate those combinations with contradictory constraints.

2. **Value Selection**: This analysis is to identify key values that would “solve” the uncertain code paths or values constrained by the previous analysis. It consists of the following substeps: (i) Identify values that are certain from the constraint, and prune them out using a SAT solver; (ii) Parse the uncertain values into the *conditional value* format, e.g., `[dp]:dp->d_type`, indicating the value `dp->d_type` is only meaningful under condition `dp!=NULL`; (iii) Identify the values that would be overwritten before the target log point; (iv) Find equivalent values that can be used to infer those overwritten key values; (v) From the uncertain value set, find the minimum set by eliminating redundant values that can be inferred by remaining uncertain values. (vi) Rank the uncertain values based on the amount of relevant branch conditions involved. Finally, *LogEnhancer* builds an *Uncertain Value Table* for each log point to store the identified uncertain variable values to be recorded.

3. **Instrumentation**: Before each log point, *LogEnhancer* inserts a procedure `LE_KeyValues(LogID)` to record the variable values in the Uncertain Value Table corresponding to the `LogID`, where `LogID` is a unique identifier for each log point. At run-time, `LE_KeyValues()` collects these variable values from the stack and heap *only at the log point* (delayed collection). For in-time collection, *LogEnhancer* further instruments source code to keep a “shadow copy” of any key values that will be overwritten before the log point and cannot be inferred via equivalent values live at the log point.

### 4.1.4 LogEnhancer’s Assumptions

No tool is perfect, and *LogEnhancer* is no exception. There is an inevitable trade-off between the completeness and scalability offered by *LogEnhancer*’s analysis. *LogEnhancer* makes certain simplifying assumptions to make implementation practical and to scale to large real world programs,
at the cost of a few incomplete (missing certain variable values) and/or unsound (logging non-causally-related variable values) results. However, LogEnhancer would not impact the validity of diagnosis since all values recorded by LogEnhancer are obtained right from the failed execution. I briefly outline the issues surrounding LogEnhancer’s assumptions and their impact below.

(1) **How far and deep can LogEnhancer go in analysis?** Without any limitation, any program analysis or model checking approach would hit the path explosion problem on real world software. Similar to most previous work on program analysis and model checking, even though theoretically LogEnhancer can go as far and deep as it would like, it is impractical to do so for large real world software. Therefore, LogEnhancer needs to set some limits in the depth of our inter-procedural data dependency analysis. Given the problem of inferring causally-related information, our design only focuses on analyzing functions that *must* have a causal relationship with the log point (i.e., functions that are on the call-stack or their return values are causally-related to a log point), while ignoring the side-effects of other functions. Moreover, LogEnhancer does not perform program analysis more than one level deep into functions that are not on the call stack at the log point. Each function is analyzed only once, ignoring the side-effects caused by recursive calls.

Although LogEnhancer limits its analysis in this fashion, it still identifies an average of 108 causally-related branches for each log point (with a max of 22,070 such branches for a single log point in PostgreSQL). Moreover, our experience is that the values with most diagnostic power are commonly on the execution path to a log point and such values are naturally collected using LogEnhancer’s style of analysis.

(2) **What and how many values are logged per message?** The core of LogEnhancer’s analysis is to first identify causally-related branches to each log point and then infer a compact set of values that resolve those branch choices. In the evaluation, 108 causally-related branches are identified for each log point on average, that can be resolved by 16.0 variable values (this includes the impact of removing redundant values).

(3) **How about privacy concerns?** Just as existing log message contents, the information LogEnhancer records focuses narrowly on system’s own “health”. Because LogEnhancer is only recording a small number of variable values per message, this makes it much easier (than core dumps) for users to examine to make sure that no private information is contained. It is also easier to combine with some automatic privacy information filtering techniques (e.g., [CCM08] can filter data that can potentially leak private user information). In addition, collected logs can be analyzed at customers’ sites by sending an automatic log analysis engine like SherLog to collect back the inferred and less-sensitive information (e.g. the execution path during the occurred failure).

(4) **How to handle inter-thread or inter-process data dependencies?** Limited by the capability of static analysis, LogEnhancer does not analyze data dependencies across threads or processes. Any
values that are causally-related through these dependencies thus would be missed. In most cases, such dependencies do not interfere with LogEnhancer’s analysis since most shared data do not make big impact on control flows and are not causally-related to a log message. However, in some rare cases, LogEnhancer may not log enough information to figure out why certain shared variables hold particular values. The substeps (iii) - (v) in the value selection might also be inaccurate on shared data since the inter-thread data-flow is not analyzed. Therefore, for applications with very intensive inter-thread data dependencies on control variables, LogEnhancer might disable these substeps and conservatively treat any shared data as overwritten ones at the log point.

Note this limitation is not that LogEnhancer cannot handle concurrent programs. For concurrent programs, it still analyzes the intra-thread/process data flow to identify key variables to log. Such variables are useful for diagnosing failures in programs (sequential and concurrent). Five of the evaluated applications are concurrent, including Apache, CVS, Squid, PostgreSQL and lighttpd. Section 4.3 shows the evaluation results on these applications. Note that a majority of failures in real world are caused by semantic bugs, and even mis-configurations, not by concurrency bugs [LTW+06].

Addressing this issue would require more complicated thread-aware code analysis. For each variable that is causally-related to the log message, in addition to analyze the intra-thread or intra-process data flow, LogEnhancer also needs to analyze any inter-thread or inter-process modifications. Although theoretically LogEnhancer can still use the same Uncertainty Identification algorithm to recursively follow intra-thread/process and inter-thread/process data-flow, some practical scalability and precision issues might arise. Given an uncertain variable value \( v \) in function \( F \), any modification to \( v \) that might be executed concurrently with \( F \) needs to be considered. Without precise information on which functions might be executed concurrently and pointer aliasing, LogEnhancer might end up analyzing huge number of data-flows that are not causally related to the log point. This might add exponential overhead to the analysis, and more importantly, include huge number of noisies in the set of variable values LogEnhancer decides record to enhance each log message. Annotations can be used in expressing which functions are concurrent [DLR+98, FLL+02], while techniques presented in RacerX can help to automatically infer this information [EA03]. Previous work [NA07, NAW06] also shows for memory safe languages like Java where pointer usages are are limited, it is possible to analyze the concurrency behavior of a program much more precisely. Addressing these issues remains as my future work.

(5) What if there is no log message? If a software program contains no log points at all, LogEnhancer offers no value. Fortunately, this is usually not the case in most commercial and open source software. Most commercial and open source software already contains significant amount of log points as logging has become a standard practice. Hence LogEnhancer focus on enhancing
existing log messages, and assume that such log messages exist. Furthermore, in Chapter 5, I will describe how the problem of “inserting new log points” can be addressed.

4.2 Design and Implementation

Similar to SherLog, LogEnhancer’s source code analysis is implemented using Saturn static analysis framework [ABD+07]. In this section I will not repeat the all the details of Saturn. Except for the Data-flow analysis described in Section 4.2.1, all the discussions on analysis processes, design and implementation issues are specific to LogEnhancer.

4.2.1 Uncertainty Identification

For each log printing statement in the target software, the goal of Uncertainty Identification is to identify uncertain control or data flows that are causally-related to the log point but cannot be determined assuming the log point is executed. LogEnhancer’s analysis starts from those variable values that are directly included in the conditions for the log point to be executed. It then analyzes the data-flow of these variable values to understand why these conditions hold.

Within each function \( f \), LogEnhancer starts from the beginning and goes through each statement once. At any program point \( P \) within \( f \), LogEnhancer simultaneously performs two kinds of analysis: (1) data-flow analysis that represents every memory location \( f \) accesses in the form of constrained expression (CE); (2) control-flow analysis that computes the control-flow constraint to reach \( P \). If the current \( P \) is a log point \( LP \), LogEnhancer takes the control-flow constraint \( C \), and converts each memory location involved in \( C \) to its CE. Thus both the control and data flow branch conditions related to the log point can be captured together in one constraint formula, and it is stored as the summary of \( f \) to reach \( LP \). The same process is recursively repeated into the caller of \( f \). At the end of the analysis, for every function \( f' \) along a possible call-chain to a log point \( LP \), the summary of \( f' \) is generated, which captures the causally-related constraint within \( f' \) to eventually reach \( LP \).

Data-flow analysis

LogEnhancer directly uses Saturn’s memory model for data-flow analysis. Saturn models every memory location (in stack or heap) accessed by function \( f \) at every program point \( P \) in the form of constrained expression (CE). A CE is represented in the format of \( v=E:C \), indicating the value of \( v \) equals to expression \( E \) under condition \( C \). At the beginning of each function \( f \), Saturn first statically enumerates all the memory locations (heap and stack) accessed by \( f \), and initializes each
location \( V \) as \( V = V \):\text{True} \), indicating the value of \( V \) is unknown (symbolic). This is only possible because we will model the loops as tail-recursive functions, thus each function body is loop free (see Handling Loops later in this section). At an assignment instruction \( P, \ V = \text{exp}; \), the value of \( V \) would be updated to \( \text{exp} : C \), where \( C \) is the control-flow constraint to reach \( P \). At any merge point on the control-flow graph (CFG), all the conditions of \( V \) from every incoming edge are merged. This will prune all non-causally-related conditions to reach \( P \). Figure 4.4 shows the CE of \text{is\_dir} in \text{rm} at log point 1.

\[
is\_dir = \begin{cases} 
T\_YES: & C_{\text{yes}} = \text{dp} && \text{dp->d\_type==DT\_DIR} || \!\text{dp} && \!\text{S\_ISDIR(sbuf.st\_mode)} \\
T\_NO: & C_{\text{no}} = \text{dp} && \text{dp->d\_type!=DT\_DIR} || \!\text{dp} && \!\text{S\_ISDIR(sbuf.st\_mode)} 
\end{cases}
\]

Figure 4.4: The constrained expression for \text{is\_dir} at line 16. \( C_{\text{yes}} \) and \( C_{\text{no}} \) are constraints for \text{is\_dir} to hold value \( T\_YES \) and \( T\_NO \) respectively.

Each variable involved in the CE is a \textit{live-in} variable to the function \( f \), i.e., variable whose value is first read before written in \( f \) [ALSU06]. Thus we can represent all memory locations accessed by \( f \) with a small and concise set of variable values (i.e. live-ins) to reduce the number of redundant values to record. For example, \text{is\_dir} is not a live-in variable, and its value can be represented by a small set of live-in values such as \text{dp}, \( T\_YES \), etc., as shown in Figure 4.4.

**Control-flow analysis**

At each program point \( P \), \textit{LogEnhancer} also computes the constraint for the control-flow to reach \( P \). At a log point \( LP \), every variable value involved in the control-flow constraint would be replaced by its constrained expression. Then this constraint is solved by a SAT solver to test its satisfiability. An unsatisfiable constraint indicates no feasible path can reach \( LP \), thus \textit{LogEnhancer} can prune out such constraint. The satisfiable constraint thus contains all the causally-related control and data-flow conditions to reach \( LP \). Thus if \textit{LogEnhancer} knows all the variable values involved in this constraint \( C \), it can deterministically know the execution path lead up to \( LP \). Then this constraint \( C \) will be stored as a part of this function’s summary, along with the location of \( LP \). This records that function \( f \) would reach \( LP \) under constraint \( C \). Non-standard control flows such as \text{exit}, \text{abort}, \_exit and their wrappers are identified and adjusted on the CFG. \text{longjmp}s are correlated with \text{setjmp}s through function summary in a similar manner as described in Chapter 2.

In the \text{rm} example, the control-flow constraint within \text{remove\_entry} to reach log point 1 would be \( \text{is\_dir==T\_NO} \ &\ !\ \text{unlink(filename)}!=0 \). Then \text{is\_dir} is replaced by its CE as shown in Figure 4.4. The SAT solver determines \( T\_YES \) cannot satisfy this control-flow constraint, thus \( T\_YES \) and its constraint are pruned. The remaining result is a simplified, feasible constraint \( C_r \), which
is stored as the summary of remove_entry indicating the conditions for remove_entry to reach log point 1:

\[
C_r = (dp \&\& dp->d_type!=DT_DIR \mid \mid !dp \&\& !S_ISDIR(sbuf.st_mode)) \&\& \\
\text{unlink(filename)}
\]

**Inter-Procedural analysis**

After analyzing function \( F \), the above process is then recursively repeated into the caller of \( F \) by traversing the call-graph in bottom-up order. In \( rm \), after analyzing remove_entry, LogEnhancer next analyzes its caller remove_cwd_entries in the same manner: a linear scan to compute the CE for each memory location and control-flow constraint for each program point. At line 25, it finds a call-site to a function with a summary (remove_entry), indicating that reaching this point might eventually lead to log point 1. Therefore LogEnhancer takes the control-flow constraint \((C_c = (\text{readdir(dirp)}!=\text{NULL}))\), and replaces every variable with its CE (in this case the CE for dirp).

Besides \( C_c \), for context sensitivity, LogEnhancer also takes the \( C_r \) from remove_entry and substitutes it to a constraint that is meaningful in remove_cwd_entries:

\[
C'_r = (\text{readdir(dirp)} \&\& \text{readdir(dirp)}->d_type!=DT_DIR \mid \mid !\text{readdir(dirp)}) \&\& f==\text{Sym}
\]

Here, readdir(dirp) is the substitution for dp in \( C_r \) since the dp in remove_entry is not visible in the caller’s context; S_ISDIR(sbuf.st_mode) is pruned also because it is not visible in the caller’s context; \( f==\text{Sym} \) is the substitution for unlink (filename). Sym is a symbolic value and \( f \) is the substitution of filename in caller. \( f==\text{Sym} \) indicates we should plug-in the CE of \( f \) to track the inter-procedural data-flow, while not enforcing any constraint on \( f \)’s value. Finally, \( C'_r \land C_c \) is stored as the summary for remove_cwd_entries to reach log point 1.

The above process ignores those caller-invisible values (sbuf.st_mode is pruned), and only substitutes visible ones into caller’s context (dp is substituted by readdir(dirp)). \( f==\text{Sym} \) is converted from unlink(filename) != 0, where Sym is a symbolic value that can equal to anything. This tells LogEnhancer to only track the data-flow of \( f \) in remove_cwd_entries, while not enforcing any constraint on its value. Since the return value of unlink cannot be converted into the caller, so LogEnhancer only tracks the data-flow of its parameter filename, which is converted into \( f \). It then enforces both constraints from caller and converted from callee, e.g., \( C'_r \land C_{\text{cwd\_entries}} \) by querying the SAT solver. If satisfiable, then a summary is generated for remove_cwd_entries, to be further propagated into its caller.

Such bottom-up analysis traverses upward along each call chain of the log point. It ignores functions that are not in the call chains to this log point—I will refer them as “sibling functions”.

67
Sibling functions may also be causally-related to the log point. Therefore, if one of sibling function’s return values appear in the constraint to the log point, LogEnhancer also analyzes such function and identifies the control- and data flow dependencies for its return value. This analysis is implemented as a separate analysis pass after the bottom-up analysis. Currently LogEnhancer limits the analysis to follow only one level into such functions due to scalability concern. However, there is no theoretical limit, and it can carry the analysis deeper. If a causally-related sibling function is a library call with no source code (e.g., unlink() in the rm example), LogEnhancer simply plugs in its parameter into its constraint so it may choose to record the parameter.

Handling Loops

Loops are modeled as tail-recursive functions [ABD+07] in a similar fashion as SherLog, so each function is cycle-free, which is a key requirement allowing us to statically enumerate all the paths and memory locations accessed by each function. Such a loop is handled similarly as ordinary functions except that it is being traversed twice, to explore both loop entering and exiting directions. Variable \( V \) modified within the loop body are propagated to its caller as \( V == \text{Sym} \), to relax the value constraint since LogEnhancer is not following the multiple iterations as in run-time. This way, constraint from the loop body can be conservatively captured.

```plaintext
void func () {
    err = 0;
    for (i = 0; i < N; i++) {
        if (a[i]==INV)
            err++;
    }
    Log point 1
    error ("Failed, %d", err);
}

func: err = 0;
i = 0;
call: for_loop

for_loop: if (i < N)
    if (a[i]==INV)
        err++;
        if (err)
            return ();
    i++;
    if (err)
        return ();
call: for_loop
```

Figure 4.5: Code and Control-flow graph of tail-recursive calls for loop.

Figure 4.5 shows how a loop is split into a separate tail-recursive function (for_loop). By traversing this for_loop function twice, LogEnhancer can infer variables \( i, a[i], \text{err} \) into the constraint for reaching log point 1. The first pass it identifies the constraint \( C_1: ! (i < N) \& \& \text{err} \) for log point 1. The second pass at the tail recursive call-site, it uses this \( C_1 \) as the summary, substitute it into \( C_1' : i == \text{Sym} \& \& \text{err} == \text{Sym} \). LogEnhancer is relaxing the value constraints on the variables modified within the loop body, since it is not following the iteration as in run-time. Thus by plug in the CE of err, the second pass would infer the constraint being \( C_2: i == \text{Sym} \& \& a[i] == \text{Sym} \& \& \text{err} == \text{Sym} \).

68
Efficiency and Scalability

Uncertainty Identification scans the program linearly, a key to our scalability to large applications. LogEnhancer further uses pre-selection and lazy SAT solving for optimization. The former pre-selects only those functions that on the call-stack of any log point to analyze, and the latter queries the SAT solver lazily only at the time when function summaries are generated.

Pointer Aliasing

Saturn [ABD+07] provides precise intra-procedural pointer aliasing model. Inter-procedural pointer aliasing analysis is only performed on function pointers to ensure that LogEnhancer can traverse deep along the call-chain. The other types of pointers are assumed to be non-aliased, which might cause us to miss some causally-related variable values. For Value Selection LogEnhancer will enable inter-procedural alias analysis for all types of pointers for conservative liveness checking.

4.2.2 Value Selection

This step is to select, from all constraints identified by the previous step, what key variable values to record at each log point. The Value Selection consists of the following three steps. In this section, I will refer an expression without any Boolean operator (&&, ||, !) as a uni-condition. For example, \( dp!=NULL \) is a uni-condition (note != is not one of the three Boolean operators). A constraint is thus a formula of uni-conditions combined together using Boolean operators.

(1) Pruning determinable values: Some variable values can be inferred knowing that a given log point is executed. I will call them as determinable values. For example, in constraint \( a==0 \) \&\& \( b!=0 \), "a" can be determined that it must equal to zero, while b’s value is still uncertain. A determinable value \( V \) is identified if: (i) \( V \) is involved in a uni-condition \( uc \) that is the necessary condition to constraint \( C \) (i.e., \( \neg uc \wedge C \) is unsatisfiable); (ii) \( uc \) is in the form of \( V==CONSTANT \). A determinable value thus can be pruned out since it need not to be recorded.

(2) Identifying the condition for a value to be meaningful: After the above step, all remaining values are uncertain. However, not every value is meaningful under all circumstances. In \( \tau m \), \( dp->d_type \) is meaningful only if \( dp!=NULL \). Recording a non-meaningful value could result in an invalid pointer dereference or reading a bogus value. Therefore, for each value that is not pruned, LogEnhancer also identifies under what condition this value would be meaningful, representing it in a format as \([C]:V\), indicating value \( V \) is meaningful under condition \( C \). Our run-time recording will first check \( C \) before recording \( V \).

(3) Liveness checking and Equivalent Value Identification (EVI): A value can also be dead (overwritten or disappear together with its stack frame) prior to a given log point and LogEnhancer
cannot delay the recording until the log point. To identify such dead values, LogEnhancer performs conservative liveness analysis, i.e., if one variable value might be modified before the log point, it conservatively marks it as “dead”. To be conservative, LogEnhancer also runs Saturn’s global pointer alias analysis [HA06] before the liveness checking. Any pointers passed into a library call where source code is unavailable are conservatively treated as “dead” after the call (LogEnhancer excludes some common C libraries such as strlen); Any extern values not defined inside the program are also conservatively treated as dead.

However, LogEnhancer does not give up on recording dead values so easily. For each dead value, it tries to find some equivalent variable values which live until the log point and can be used to infer back the dead value. More specifically, a value EV is equivalent to another value V iff: (i) it is defined as $EV = V \text{ op } UV$, where UV are other live values, and (ii) both have the same control flow constraint. Therefore, if a dead value $V$ has an equivalent $EV$, it simply records $EV$ and $UV$.

(4) Ranking: Finally, LogEnhancer ranks all selected values based on the amount of uncertain branch conditions they contribute. Ranking can be used to prioritize our run-time recording and presenting the recorded values to users.

Since the constraint tracks the causal-relationship among the uncertain variable values, our ranking implementation is thus simple: values are ranked by the count of their appearances in the constraint formula. For example, as shown in Section 4.2.1, the constraint to reach the log point 1 in the rm example is:

$$C_r = (dp && dp->d_type!=DT_DIR || !dp && !S_ISDIR(sbuf.st_mode)) && unlink(filename)$$

With this constraint, $dp$ would be ranked the highest since it appeared twice, while all other values appeared only once.

Currently LogEnhancer does not set any threshold on the number of variables it records at log point. So the ranking is only used when presenting the recorded values to users.

### 4.2.3 Run-time Value Collection

This section describes how LogEnhancer modifies the original source code to collect the variable values. It describes two value collection policies: delayed collection and in-time collection.

**Delayed Collection:** LogEnhancer instruments the source code of the target application right before each log point by adding a function call `LE_KeyValues()` to record the values of identified live variables via their addresses. The addresses of these variables are obtained from the compiled binary by parsing the DWARF debugging information [DWA]. Local variables’ addresses are the offsets from the stack frame base pointer. Heap values’ addresses are represented the same way as
how they are referenced in the original code. Each live value represented by its address and the condition for it to be meaningful is stored into an Uncertain Value Table (UVT) that corresponds to a log point. At the end of our analysis, each UVT is output to a separate file. These files are released together with the target application.

Figure 4.6 shows the run-time process of \texttt{LE\_KeyValues()} . It is triggered only at the log point, i.e., when a log message is being printed. When triggered, it first uses the LogID of the log point to load the corresponding UVT into the memory. It then obtains the current call stack, using it to index into the UVT to find what values to record. For each value, the condition for it to be meaningful is first tested. A local variable’s dynamic address is computed by reading the offset from UVT and then add this offset to its dynamic stack frame base pointer obtained by walking the stack. Note that UVT is only loaded into the memory during the execution of \texttt{LE\_KeyValues()}, so the delayed recording does not add any overhead during normal execution, i.e., when no log message being printed. \texttt{LogEnhancer} also records the dynamic call stack.

By default, \texttt{LogEnhancer} records only basic type values. For example, for a pointer value, it only records the address stored in this pointer. To further provide meaningful diagnostic information, \texttt{LogEnhancer} adds two extensions. First, if the variable is of type \texttt{char*} and is not NULL, then it records the string with a maximum of 50 characters (of course, if the string is shorter than 50, it records only the string). Second, if the variable is a field within a structure, in addition to that field, it also records the values of other fields. This is because structures are often used to represent multiple properties of a single entity, such as a request in apache httpd.

Although for \texttt{LogEnhancer}’s design, it is already very cautious to only record meaningful and valid values to ensure memory safety, due to the limitation of static analysis, it might still access an invalid memory location (e.g., caused by multi-threading). To be conservative, \texttt{LogEnhancer} further ensures memory safety by intercepting SIGSEGV signals without crashing the running application. For applications such as apache who also intercepts SIGSEGV signals, \texttt{LogEnhancer} add a wrapper to filter out those obviously caused by our log recording. In the experiments, such signal has never been encountered.

\texttt{LogEnhancer} also implements a variation of the delayed method as a core dump analyzer
(referred as a **Core Dump Digger**) that automatically identifies the key values (or its equivalent values) from a core dump at a log point (if there is such core dump). Not every log point has a core dump, especially those book-keeping or warning messages.

**In-time Collection:** In addition to instrumentation at log points, the in-time collection method further saves a shadow copy of every dead value \( X \) that has no equivalent value by instrumenting the code in following way:

- \( \text{if} \ (X) \)
- \( \text{if} \ (\text{LE}_\text{InTime}(&X, \text{Lint32}) \&\& X) \)

\( \text{LE}_\text{InTime}() \) always returns 1. It simply copies \( \text{Lint32} \) number of bytes starting from \( &X \). Note that \( \text{LE}_\text{InTime}() \) can record \( X \) directly without checking any condition since it is within the same context as the use of \( X \).

All recorded values from \( \text{LE}_\text{KeyValue}() \) and \( \text{LE}_\text{InTime}() \) are first stored into buffers in memory (both currently 40 KB) respectively. At *error* messages, both buffers were flushed to disk. \( \text{LE}_\text{KeyValue}() \)’s buffer is also flushed when it becomes full, whereas \( \text{LE}_\text{InTime}() \) simply recycles the shadow buffer from the beginning. Each thread has its own private buffer.

### 4.3 Evaluation

<table>
<thead>
<tr>
<th>Application</th>
<th>Version</th>
<th>Lines of code</th>
<th>Log Points</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>All</td>
</tr>
<tr>
<td>In</td>
<td>4.5.1</td>
<td>20K</td>
<td>26</td>
</tr>
<tr>
<td>rm</td>
<td>4.5.4</td>
<td>18K</td>
<td>28</td>
</tr>
<tr>
<td>tar</td>
<td>1.22</td>
<td>66K</td>
<td>210</td>
</tr>
<tr>
<td>apache</td>
<td>2.2.2</td>
<td>228K</td>
<td>1,654</td>
</tr>
<tr>
<td>cvs</td>
<td>1.11.23</td>
<td>111K</td>
<td>1,088</td>
</tr>
<tr>
<td>squid</td>
<td>2.3.4</td>
<td>70K</td>
<td>1,116</td>
</tr>
<tr>
<td>postgresql</td>
<td>8.4.1</td>
<td>825K</td>
<td>4,876</td>
</tr>
<tr>
<td>lighttpd</td>
<td>1.4.26</td>
<td>53K</td>
<td>127</td>
</tr>
</tbody>
</table>

Table 4.1: Evaluated applications by *LogEnhancer*. LOC is lines of code. Note all the dependent library code that are scanned by *LogEnhancer* are counted. “All” shows the total number of log points for the most verbose level. “Default” shows the default verbose-level of log message printed (in bracket) and the number of log points at this level.

For evaluation purpose, *LogEnhancer* is applied to enhance *every of the total 9,125* log messages in 8 different real-world software projects as shown in Table 4.1. Five of them are server applications, including 2 web servers (apache httpd, lighttpd), a database server (postgresql), a concurrent version control server (cvs), and a web cache (squid). For server applications where there are multiple log files, *LogEnhancer* is applied to enhance all messages printing into the
error log file. In the default verbose mode, all applications only print error and/or warning messages. Therefore, during normal execution with the default verbose mode, there is few log message printed besides a few messages indicating system start/stop.

For any diagnosis tools like LogEnhancer, the most effective evaluation method is of course a user study by having it used by real programmers for a period of time and then report their experience. Unfortunately, this would be a time-consuming process and also it is hard to select samples to be representative. Given these constraints, LogEnhancer is evaluated both quantitatively and qualitatively using three sets of experiments:

1. **Value selection.** First, my dissertation investigates how well LogEnhancer’s algorithm captures the variables that are useful for failure diagnosis by comparing against manual selection (variables that have already been recorded in existing logging statements by programmers). Then, I also evaluate how many new variables are selected for logging in addition to those in this intersection set (i.e., how many new variable values would be logged by LogEnhancer) and how effective these additional logged values can help reducing the number of code paths to be considered in post-mortem diagnosis.

2. **Diagnostic effectiveness.** In the second set of experiments 15 real world failure cases are selected, including 13 bugs and 2 mis-configurations, to show the effectiveness of the information collected by LogEnhancer in failure diagnosis. In particular, I will also show how automatic log inference tools like SherLog can be improved given the information added by LogEnhancer into log messages.

3. **Logging overhead.** The third set of experiments evaluate the overhead introduced by LogEnhancer’s run-time logging.

All the experiments are conducted on a Linux machine with eight 2.33GHz Xeon processors and 16GB of memory. Since the analysis is done off-line, LogEnhancer currently runs as single process, single thread (even though the analysis can potentially be parallelized to reduce the analysis time [ABD+07]).

### 4.3.1 Effectiveness in Variable Recording

Figure 4.7 shows LogEnhancer’s comparison with existing log variables included manually by programmers into log messages over the years. On average, 95.1% (with minimum 89% and maximum 98%) of these log variables can be selected automatically by LogEnhancer. In all the
Figure 4.7: Variable values at each log point. The number of variables per message logged manually by developers is compared with the ones inferred automatically by LogEnhancer. “Overlap” shows the number of variable values that are selected by both programmers and LogEnhancer. The percentages of overlap are marked beside each bar. “LE-additional” shows the additional variable values only identified by LogEnhancer.

Applications except *squid*, LogEnhancer achieves a coverage over 95%². This high coverage is an evidence that our design matches with the intuition of programmers in recording key values to help diagnosis. It implies that LogEnhancer can do at least as good as manual effort.

The small fraction (4.9% on average) of existing log variables that are not automatically selected by LogEnhancer is mainly book-keeping information that is not very useful for inferring the execution path to the log point. For example, when CVS detects an invalid configuration entry, it outputs the line number of that entry in the configuration file. Since this line number is not used in any branches, it is thus missed by LogEnhancer. Note that the invalid entry string itself is identified by LogEnhancer. So even without the line number, by recording the configuration entry string itself is enough for users/developers to locate the error in the configuration file.

There are four main categories of manually-identified variables that are missed by LogEnhancer, together contributing to 97% of the few missed cases. (1) Book-keeping values logged immediately after initialization (37%): For example, in Squid, immediately after receiving a request, the length of the request is logged before it is actually used. All these log messages are verbose mode messages that do not indicate any error and are not enabled in default production settings. This explains why LogEnhancer only covered 88.7% of the existing log variables in Squid: majority (64% as shown in Table 4.1) of the log messages in Squid are verbose mode messages, and many of them are in such style. (2) The line number of invalid entry in configuration file (28%). (3) General configuration (host-names, PID, program names, etc.) (24%) that are not causally-related to the log point. Note causally-related configuration information would be identified LogEnhancer. (4) Many variable values are converted to human readable strings when printing to log message. For example “inet_ntoa” converts an IP address into string. I count the value as covered by LogEnhancer only if by recording the non-textual value I can deterministically infer text string.

---

²Many variable values are converted to human readable strings when printing to log message. For example “inet_ntoa” converts an IP address into string. I count the value as covered by LogEnhancer only if by recording the non-textual value I can deterministically infer text string.
Redundant multi-variables (8%) that are always updated together while only one is used in branch. \textit{LogEnhancer} only identifies the one used in branch while the missed values can be inferred from the identified one.

![Figure 4.8: Number of Uncertain Branches reduced by LogEnhancer. It compares the amount of uncertain branches that are causally related to each log point given different types of information recorded: without any variables (the original uncertainty space); existing variables included by developers; call stack in addition to existing variables; variables inferred by LogEnhancer and call stack using the delayed collection method.](image)

In addition to automatically select most of existing log variables (manually included by programmers), \textit{LogEnhancer} also selects an average of 14.6 additional new variable values for each log message. Recording these values (including the call stack) can eliminate an average of 108 uncertain branches for each log point as shown in Figure 4.8. From the 108 original uncertain branches per log point, existing log variables can reduce it to 97, whereas the \textit{LogEnhancer}’s delayed recording scheme can reduce this number to 3, meaning that, on average for each log point, there are only 3 un-resolved branches for programmers to consider to fully understand why the log point was reached. These remaining branches are caused by uncertain values that are dead at log points, and can only be recorded by our in-time collection (if overhead is not a concern). If \textit{LogEnhancer} records only the stack frames in addition to the original log messages, the number of uncertain branches are only reduced from 97 to 40 on average. Table 4.2 shows the detailed number of uncertain branches. Please note this evaluation is a best-effort evaluation. These Uncertain Branches are identified with the same methodology as \textit{LogEnhancer}’s analysis, and therefore suffer from the same limitations as \textit{LogEnhancer} and might not be objective. How to more objectively evaluate on Uncertain Branches remains as my future work.

Table 4.2 also shows the number of variable values identified by \textit{LogEnhancer} at different analysis stages. On average, 16.0 uncertain values are identified for each log point (“all”). 14.6 of them can be recorded at log points (“logged”) without introducing normal-run overhead. Among these 14.6 variables, 12.9 of them were not overwritten before log point (i.e., they are “live”), and
Table 4.2: The number of uncertain branches and uncertain variable values per log point. The large difference between average and median in “w/o any var” is caused by small number of log points inside some library functions, that have a huge number of uncertain branches accumulated from huge number of possible call stacks. Once we differentiate different call stacks, this difference between average and median significantly reduces.

![Figure 4.9: Number of variables used in branches.](image)

the rest 1.7 are recovered from Equivalent Value Identification (EVI). On average 49% of the dead values can be recovered by our EVI. The remaining 51% dead values can be collected only via in-time collection, with the cost of some overhead to normal execution.

**Effectiveness in reducing the number of variables to record:** Figure 4.9 shows the effectiveness of LogEnhancer’s value selection in reducing the number of variables to record. It compares the total number of variables used in all the branches throughout the entire program, the number of variables used in all uncertain branches for all log points, and the number of these uncertain variables if represented live-in form. For example, in Apache, there are 10,798 variables used in branch conditions in the entire program, however only 2,585 of them are in uncertain branches to some log points. Further, these variable values can be inferred by only recording 1,210 live-in variables with LogEnhancer. Consequently, on average LogEnhancer identified 17.2 uncertain
values for each log point in Apache (Table 4.2).

Figure 4.10: Ranking of uncertain variable values in Apache. We show the accumulated number of uncertain branches each variable involved.

**Ranking of variable values:** Figure 4.10 shows how the number of uncertain branches are reduced as the number of recorded variables increases in Apache, sorted based on each variable’s contribution in uncovering uncertain branches (i.e., its ranking). The contribution of each variable value is the average across all log points in Apache. By recording the single highest ranked variable LogEnhancer can eliminate an average of 25% of the uncertain branches to each log point. 50% of the uncertain branches can be eliminated by recording only 3 variables.

**Analysis performance:** Table 4.3 shows the analysis time of LogEnhancer on each application. For all applications except postgresql, LogEnhancer finishes the entire analysis within 2 minutes to 4 hours. For postgresql, it takes 11 hours since it has 4,876 logging points in a large code base. Since it is expected that LogEnhancer is used off-line prior to software release, the analysis time is less critical. Besides, the summary-based design allows it to be parallel or incrementally applied [ABD+07]. The memory usage in all cases is below 2.3GB.

<table>
<thead>
<tr>
<th>Application</th>
<th>Analysis Time</th>
<th>Memory Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>ln</td>
<td>3 minutes</td>
<td>579MB</td>
</tr>
<tr>
<td>tar</td>
<td>1.5 hours</td>
<td>263MB</td>
</tr>
<tr>
<td>cvs</td>
<td>3.0 hours</td>
<td>1.7GB</td>
</tr>
<tr>
<td>postgres</td>
<td>10.7 hours</td>
<td>1.5GB</td>
</tr>
<tr>
<td>rm</td>
<td>2 minutes</td>
<td>172MB</td>
</tr>
<tr>
<td>apache</td>
<td>2.1 hours</td>
<td>1.3GB</td>
</tr>
<tr>
<td>squid</td>
<td>3.8 hours</td>
<td>2.3GB</td>
</tr>
<tr>
<td>lighttpd</td>
<td>20 minutes</td>
<td>532MB</td>
</tr>
</tbody>
</table>

Table 4.3: Analysis performance.
4.3.2 Real World Failures

LogEnhancer is further evaluated by diagnosing 15 real-world failures, including 13 software bugs and 2 configuration errors, to see how the enhanced log messages would help failure diagnosis. In all these cases, the original log messages were insufficient to diagnose the failure due to many remaining uncertainties, while with LogEnhancer’s log enhancement these uncertainties were significantly reduced and almost eliminated. In this section, I will show 3 cases in detail to demonstrate the effectiveness of LogEnhancer. The other 12 cases are summarized in Table 4.4.

This section also compares the inference results of SherLog (describe in Chapter 2) before and after LogEnhancer’s enhancement.

Case 1: rm: For the rm failure described in Figure 4.2, LogEnhancer recorded the call stack at the moment of the error message being: ...remove.cwd.entries:25 -> remove.entry. In addition, LogEnhancer records the following variable values at log point 1: dp=0x100120, filename ="dir1/dir2", dp->d_type = DT_UNKNOWN. Programmers can now infer that the failed execution must took the path at line 5 and came from caller remove.cwd.entries. They can also tell that readdir returns a non-NULL value dp, but dp->d_type’s value is DT_UNKNOWN in the failed execution—which is exactly the root cause: the programmers did not expect such type for dp->d_type. In such case, just as if dp is NULL, the program should also use lstat to determine the directory type. So the fix is straightforward as below:

4: if (dp)
4: + if (dp && dp->d_type!=DT_UNKNOWN)

Without LogEnhancer’s enhancement, SherLog inferred a total of 13 possible call paths (not even complete execution paths, only function call sequences) that might have been taken to print

<table>
<thead>
<tr>
<th>Failure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rm</td>
<td>reports a directory cycle by mistake for a healthy FS.</td>
</tr>
<tr>
<td>cp</td>
<td>fails to replace hardlinks given “–preserve=links”.</td>
</tr>
<tr>
<td>ln</td>
<td>ln –target-directory failed by missing a condition check.</td>
</tr>
<tr>
<td>apache1</td>
<td>denies connection after unsuccessful login attempt.</td>
</tr>
<tr>
<td>apache2</td>
<td>OS checking procedure failed causing server to fail.</td>
</tr>
<tr>
<td>apache3</td>
<td>Server mistakenly refuses SSL connections.</td>
</tr>
<tr>
<td>apache4</td>
<td>A structure field wasn’t initialized properly causing unpredictable failure symptoms.</td>
</tr>
<tr>
<td>squid</td>
<td>wrong checking function caused access control failed.</td>
</tr>
<tr>
<td>cvs</td>
<td>login with OS account failed due to misconfiguration.</td>
</tr>
<tr>
<td>tar 1</td>
<td>failed since archive_stat.st_mode improperly set.</td>
</tr>
<tr>
<td>tar 2</td>
<td>tar failed to update non-existing tar-ball.</td>
</tr>
<tr>
<td>lighttpd</td>
<td>Proxy fails when connecting to multiple backends.</td>
</tr>
</tbody>
</table>

Table 4.4: Real-world failures evaluated by LogEnhancer.
the error message. Developers need to further manually determine among these which one actually lead to the failure. SherLog also failed to infer the value of dp and dp->d_type, leaving no clues for developers to infer branch direction at line 4. With LogEnhancer’s result, SherLog can pinpoint the only possible call path, and developers can easily examine the value of dp and dp->d_type.

**Case 2: Apache bug:** Figure 4.11 shows a bug report in apache. With only the error log message printed at line 3, the developer could not diagnose the failure. Therefore he asked the user for all kinds of run-time information in a total of 95 message exchanges. Actually only two pieces of information are key to identify the root cause. One is the value of c->keepalives and the other is the request type, r->proxyreq, which are unfortunately buried deep in huge amount of not very relevant data structures.

LogEnhancer automatically identifies c->keepalives and r->proxyreq to collect for this log message. c->keepalives is identified since it is in the constraint for the program to reach the log point. To reach the log point, proxy_process_response needs to be called by proxy_http_handler at line 23 and it is control-dependent on determine_connection’s return value. determine_connection’s return value is further data-dependent on the value of c->keepalives at line 15. Therefore c->keepalives is in the constraint to reach the log point. r->proxyreq is identified in the similar manner.

If the developers had used LogEnhancer to enhance their log messages automatically, LogEnhancer would have helped them saving a lot of time discussing back and forth with the user.
Interestingly, after such painful experience, the programmers added a patch whose sole purpose was to log the value of \(c->\text{keepalives}\) in this function.

Without \textit{LogEnhancer}’s enhancement, SherLog inferred 63 possible call paths and not be able to infer the value of \(c->\text{keepalives}\) nor \(r->\text{proxyreq}\). With \textit{LogEnhancer}’s enhancement, SherLog can narrow down to only one possible call path, and infer the value of \(c->\text{keepalives}\) and \(r->\text{proxyreq}\).

\textbf{Case 3: Apache configuration error:} A misconfiguration in Apache resulted in a failure with the log message shown in Figure 4.12. It warns no space on disk, while users’ file system and disk were perfectly healthy with plenty of free space available. From the source code, it is certain that the message was printed at line 6, as a result of an unsuccessful call to \textit{create()} at line 4. However, developers had no other clues why this call failed.

\textit{LogEnhancer} in this case identifies \texttt{mech} as a key value to collect, since it is used at line 10 in function \texttt{mutex\_method}, whose \texttt{nmutex} is causally related to the log point. If apache had been enhanced by \textit{LogEnhancer}, the log message would record the value of \texttt{mech} being \texttt{APR\_LOCK\_DEFAULT} and the value of \texttt{nmutex->meth} being \texttt{apr\_mutex\_unix\_sysv\_methods}. This indicates that apache was using the default lock setting which caused the failure. In a multi-threaded mode, apache should use \texttt{fioctl\_based lock} instead. To fix this, users should explicitly add “\texttt{AcceptMutex fcntl}” into the configuration file.

Note that, without \textit{LogEnhancer}’s enhancement, SherLog cannot infer the value of \texttt{mech} from the original log message and would not be able to narrow down to the lock setting configuration as the root cause.

```c
1  ap_mpm_run (...) {
2    if (rv = mutex_method(nmutex, mech)) {
3      return rv;
4    r = nmutex->meth->create(...);
5    if (rv ! AP\_SUCCESS)
6    ap_log_error (7 "Couldn’t create cross-process lock");
8 }
9  mutex_method(nmutex, mech) {
10    switch (mech) {
11      case APR\_LOCK\_DEFAULT: 12      nmutex->meth = 
13      apr\_mutex\_unix\_sysv\_methods; 14    }
```

Figure 4.12: Apache configuration error. The dependencies to identify variable \texttt{mech} are marked as arrows.

\subsection{4.3.3 Overhead}

\textbf{Execution time:} Table 4.5 shows the \textit{LogEnhancer}’s recording overhead during applications’ normal execution under the default verbose mode. For server applications, the overhead is measured
as throughput degradation when the server is fully loaded. For `rm`, `ln` and `tar`, the overhead is measured in increase of execution time. Few log messages are printed in the default mode during normal execution. Thus there is no overhead for LogEnhancer with the delayed collection method. The in-time collection incurs small (1.5-8.2%) overhead due to shadow copying. This number can be reduced by eliminating those shadow recording in frequently invoked code paths (e.g., inside a loop). For example in postgresql, by disabling two instrumentations in the `hash_seq_search` library function, the slow-down can be reduced to 1%.

Table 4.5: Performance overhead added by LogEnhancer’s logging. The first number is the overhead for the delayed collection, and the second is for the in-time collection.

<table>
<thead>
<tr>
<th>Applications</th>
<th>Slow-down</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rm</code></td>
<td>0.0%</td>
</tr>
<tr>
<td><code>ln</code></td>
<td>0.0%</td>
</tr>
<tr>
<td><code>tar</code></td>
<td>0.0%</td>
</tr>
<tr>
<td><code>cvs</code></td>
<td>0.0%</td>
</tr>
<tr>
<td><code>squid</code></td>
<td>0.0%</td>
</tr>
<tr>
<td><code>apache</code></td>
<td>0.0%</td>
</tr>
<tr>
<td><code>postgresql</code></td>
<td>0.0%</td>
</tr>
<tr>
<td><code>lighttpd</code></td>
<td>0.0%</td>
</tr>
</tbody>
</table>

Figure 4.13 shows LogEnhancer’s performance during normal execution with other verbose modes. Turning on debug level log causes 49.1% slow-down even without LogEnhancer. With LogEnhancer, there is only additional 3-6% overhead on top of the original. In other words, regardless verbose mode, the additional overhead imposed by LogEnhancer is small.

**Memory overhead:** As mentioned in Section 4.2.3, delayed collection does not introduce any memory overhead during normal execution since no log message is printed. For in-time collection, the only memory overhead is the size of the in-memory buffer, which is set to 40KB in the experiment. If a log point is executed at run-time, LEKeyValue() introduces additional memory overhead by loading the UVT into the memory. In all the 8 applications, the median and average sizes of UVT are 395 bytes and 354 kilobytes respectively.

**Comparison with core dump:** Table 4.6 compares LogEnhancer’s recording time and data size with core dump at a failure. 5 failures in table 4.4 are reproduced and a core dump is forced to be generated at each log point using gcore [GCoa] library call. The log size of LogEnhancer does not
Table 4.6: Comparison between LogEnhancer and core dump. 5 failures in table 4.4 are reproduced and a core dump is forced to be generated at each log point using gcore [GCoa] library call. The log size of LogEnhancer does not include the size of the original log. The size of original log (without LogEnhancer) is shown in the parenthesis.

<table>
<thead>
<tr>
<th>Failure</th>
<th>LogEnhancer Time (ms)</th>
<th>LogEnhancer Size (bytes)</th>
<th>Coredump Time (ms)</th>
<th>Coredump Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ln</td>
<td>0.45</td>
<td>630</td>
<td>45</td>
<td>62</td>
</tr>
<tr>
<td>rm</td>
<td>0.45</td>
<td>610</td>
<td>51</td>
<td>176</td>
</tr>
<tr>
<td>tar 2</td>
<td>0.39</td>
<td>630</td>
<td>95</td>
<td>93</td>
</tr>
<tr>
<td>cvs</td>
<td>0.44</td>
<td>60</td>
<td>52</td>
<td>53</td>
</tr>
<tr>
<td>apache 1</td>
<td>0.41</td>
<td>670</td>
<td>196</td>
<td>354</td>
</tr>
</tbody>
</table>

Table 4.6 shows that LogEnhancer’s log size is 53-354 bytes, which is in the same order of magnitude as original log message. A large portion of this log is the call stack encoded in clear text. LogEnhancer could further compress this portion since calling contexts are likely to be the same for a log point.

4.4 Summary

In this chapter, I described a tool, LogEnhancer, that systematically enhance every log message in software to collect causally-related diagnostic information. By applying LogEnhancer uniformly on 9,125 different log messages in 8 applications including 5 server applications, it shows that 95.1% of the variables included in the log messages by developers over time can be automatically identified by LogEnhancer. More importantly, LogEnhancer adds on average 14.6 additional values per log message, which can reduce the amount of uncertainty (number of uncertain branches) from 108 to 3 with negligible overhead. These information not only benefits manual diagnosis but also those automatic log inference engines.
Chapter 5

Improving Log Quality II: Where to Log?

While LogEnhancer enhances the quality of existing log messages, it assumes programmers already appropriately placed the log statements. But what if there are no relevant messages printed in the first place? How well have developers anticipated the failures that occur in practice? As this Chapter will show, there is significant room for improvement.

Figure 5.1: A real world example from Apache on the absence of error log message. After diagnosing this failure, the developer released a patch that only adds an error-logging statement.

```
apr_table_t *groups_for_user(..., char *grpfile) {
  if ((status = ap_pcfg_openfile(&f, p, grpfile)) != APR_SUCCESS) {
    return DECLINED;
  }
  /* NO log! Simply decline a client request */

  /* A patch only to do logging: */
  + ap_log_error(..., "Could not open group file: %s", grpfile);

  /* Apache, mod_auth.c */
```

Figure 5.1 shows one real world failure from the Apache web server. The root cause was a user’s misconfiguration causing Apache to access an invalid file. While the error (a failed open in ap_pcfg_openfile) was explicitly checked by developers themselves, they neglected to log the event and thus there was no easy way to discern the cause postmortem. After many exchanges with the user, the developer added a new error message to record the error, finally allowing the problem to be quickly diagnosed.

Figure 5.2 shows another real world failure example from the squid web proxy. A user reported that the server randomly exhausted the set of available file descriptors without any error message. In order to discern the root cause, squid developers worked hard to gather diagnostic information (including 45 rounds of back-and-forth discussion with the user), but the information (e.g., debug messages, configuration setting, etc.) was not sufficient to resolve the issue. Finally, after adding a statement to log the checked error case in which squid was unable to connect to a DNS server (i.e., status != COMM_OK), they were able to quickly pinpoint the right root cause—the original code did not correctly cleanup state after such an error.

In both cases, the programs themselves already explicitly checked the error cases, but the pro-
Motivated by this problem, the next objective of this dissertation is to provide empirical evidence concerning the value of error logging. It examines 250 randomly sampled user-reported failures from five software systems (Apache, squid, PostgreSQL, SVN, and Coreutils)\(^1\) and identify both the source of the failure and the particular information that would have been critical for its diagnosis. Surprisingly, the result shows that the majority (77%) of these failures manifest through a small number of concrete error patterns (e.g., error return codes, switch statement “fallthroughs”, etc.). Unfortunately, more than half (57%) of the 250 examined failures did not log these detectable errors, and their empirical “time to debug” suffers dramatically as a result (taking 2.2X longer to resolve on average).

In addition, while the empirical evidences themselves could motivate developers to improve this aspect of their coding, automated tools can further play an important role in reducing this burden. This Chapter will further show that it is possible to automate the insertion of such proactive logging statements parsimoniously, yet capturing the key information needed for postmortem debugging. It describes the design and implementation of a logging tool, Errlog, and show that it automatically inserts messages that cover 84% of the error cases manually logged by programmers across 10 diverse software projects. Further, the error conditions automatically logged by Errlog capture 79% of failure conditions in the 250 real-world failures from our empirical study. Finally, using a controlled user study with 20 programmers, the results demonstrate that the error messages

---

\(^1\)The data can be found at: http://opera.ucsd.edu/errlog.html
inserted by Errlog can cut failure diagnosis time by 60.7%.

Note again that the characteristic study and the Errlog tool do not attempt to detect bugs. Bug is only one type of root causes (i.e., faults) for production failures. Other types of faults include misconfigurations and hardware faults. Rather this study characterizes and improves the logging of errors, which are manifestations of faults, that can reduce the burden of postmortem diagnosis. See Figure 1.1 from Chapter 1 for the discussion on the difference between faults, errors, and failures. For example, the failure in Figure 5.1 was caused by a misconfiguration that manifested itself into a system call return error. Without logging this error, it is hard to troubleshoot this misconfiguration.

### 5.1 Log Automation vs. Log Enhancement

While Chapter 4 described LogEnhancer, a tool that can improve the quality of existing log messages by automatically collecting additional diagnostic information in each message, unfortunately, it cannot help with the all too common cases (such as seen above) when there are no log messages at all.

However, the problem of inserting entirely new log messages is significantly more challenging than mere log enhancement. In particular, there are two new challenges posed by this problem:

- **Shooting blind**: Prior to a software release, it is hard to predict what failures will occur in the field, making it difficult to know in advance where to insert log messages to best diagnose future failures.

- **Overhead concerns**: Blindly adding new log messages can add significant, unacceptable performance overhead to software’s normal execution (e.g., if a log message were inserted within a loop).

Fundamentally, any attempt to add new log messages needs to balance utility and overhead. To reach this goal, this work is heavily informed by practical experience. Just as system builders routinely design around the constraints of technology and cost, so too must they consider the role of cultural acceptance when engineering a given solution. Thus, rather than trying to create an entirely new logging technique that must then vie for industry acceptance, this dissertation focuses instead on how to improve the quality and utility of the system logs that are already being used in practice. For similar reasons, this dissertation also chooses to work “bottom-up”—trying to understand, and then improve, how existing logging practice interacts with found failures—rather than attempting to impose a “top-down” coding practice on software developers.
5.2 Where to Log?

For the ease of postmortem failure diagnosis, when developing the software, developers may need to prepare for possible failures. To help this, this Chapter first focuses on providing a good understanding of logging practices and solution to improve logging. Before presenting them, it is useful to understand how a failure happens. Let’s consider the failure model as shown in Figure 1.1 of Chapter 1, where it decomposed the structural elements of system failures—fault, error and failure. For example, the fault leading to Apache’s failure in Figure 5.1 was a user’s misconfiguration. This fault resulted into an error from the system call open. This error further manifested into a failure visible to users — Apache declined request unexpectedly.

To further inform the choice of where to place log statements, errors can be further divided into two categories:

- **Detected errors (i.e., exceptions)**: Some errors are checked and caught by a program itself. For example, it is a commonly accepted best practice to check library or system call return values for possible errors. Such errors are referred as detected errors or exceptions.

- **Undetected errors**: Many errors, such as incorrect variable values, may be more challenging to detect mechanistically. Developers may not know in advance what should be a normal value for a variable. Moreover, even when such invariants are known, overhead concerns may prevent checking all uses of a variable for invalidity. Therefore, some errors will always remain latent and undetected until they eventually produce a failure.

Please note that detecting error is not the same as detecting the fault. Different types of faults, bugs or misconfigurations, might manifest themselves into a small set of generic errors (e.g., system call return error) that can be checked and logged. For example, both a deadlock bug and a slow network might result in an empty socket that will trigger a read system call return error, which can be checked by the programmers proactively.

To dive in one step further, detected errors can be handled in three different ways:

- **Early termination**: a program can simply exit when encountering an error.

- **Correct error handling**: a program performs appropriate recovers from an error appropriately, and continues execution.

- **Incorrect error handling**: a program’s own error handling is incorrect and results in an unexpected failure (such bugs in the error handling code is referred as an error instead of fault in this Chapter).
These distinctions provide a framework for considering the best program points for logging. In particular, detected errors are naturally “log-worthy” points. Obviously, if a program is about to terminate then there is a clear causal relation between the error and the eventual failure. Moreover, even when a program attempts to handle an error, its exception handlers are frequently buggy themselves since they are rarely well tested [SC91, GRGAD+08, GDJ+11]. Consequently, logging is appropriate in most cases where a program detects an error explicitly—as long as such logging does not introduce undue overhead. Moreover, logging such errors has no runtime overhead in the common (no error) case.

5.3 Learning from Real World Failures

<table>
<thead>
<tr>
<th>Software</th>
<th>LOC</th>
<th>#Default log points*</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Total</td>
</tr>
<tr>
<td>Apache</td>
<td>249K</td>
<td>1160</td>
</tr>
<tr>
<td>Squid</td>
<td>121K</td>
<td>1132</td>
</tr>
<tr>
<td>Postgres</td>
<td>825K</td>
<td>6234</td>
</tr>
<tr>
<td>SVN</td>
<td>288K</td>
<td>1836</td>
</tr>
<tr>
<td>Coreutils</td>
<td>69K</td>
<td>1086</td>
</tr>
</tbody>
</table>

Table 5.1: Software used in the characteristic study and the number of log points (i.e. logging statements). *: the number of log points under the default verbosity mode. “Err+Warn”: number of log points with warning, error, or fatal verboisities. The percentages are computed over the corresponding values in the “Total” column.

This section describes the empirical study of how effective existing logging practices are in diagnosis. To drive the study, 250 real world failures are randomly selected. These failures were reported in five popular software systems, including four servers (Apache httpd, squid, PostgreSQL, and SVN) and a utility toolset (GNU Coreutils), as shown in Table 5.1.

The failure sample sets for each system are shown in Table 5.2. These samples were from the corresponding Bugzilla databases (or mailing lists if Bugzilla was not available). The reporting of a distinct failure and its follow-up discussions between the users and developers are documented under the same ticket. If a failure is a duplicate of another, developers will close the ticket by marking it as a “duplicate”. Once a failure got fixed, developers will often close the ticket as “fixed” and post the patch of the fix. The randomly sampled set is from those non-duplicate, fixed failures that were reported within the recent six years. For each case, the failure reports, discussions, related source code and patches are carefully studied by us to understand the root cause and its propagation leading to each failure.

In this study, it focuses primarily on the presence of a failure-related log message, and do not
look more deeply into the content of the messages themselves. The presence of log messages is
determined either by reproducing those reproducible failures or careful study of the source code
and bug report. Indeed, the log message first needs to be present before we consider the quality of
its content, and it is also not easy to objectively measure the usefulness of log content. The Log-
Enhancer work described in Chapter 4 shows promise in automatically enhancing each existing
log message by recording the values of causally-related variables (thus making any such message
more useful for postmortem diagnosis).

Threats to Validity: As with all characterization studies, there is an inherent risk that the findings
may be specific to the programs studied and may not apply to other software. While it is hard
to establish representativeness categorically, care is taken in selecting diverse programs—written
for both server and client environments, in both concurrent and sequential styles. At the very
least these software are widely used; each ranks first or second in market share for its product’s
category. However, there are some commonalities to our programs as all are written in C/C++ and
all are open source software. Should logging practice be significantly different in “closed source”
development environments or in software written in other languages then our results may not apply.

Another potential source of bias is in the selection of failures. Quantity-wise this study is on
a firmer ground, as under standard assumptions, the Central Limit Theorem predicts a 6% margin
of error at the 95% confidence level for our 250 random samples [Spa81]. However, certain fail-
ures might not be reported to Bugzilla. Both Apache and PostgreSQL have separate mailing lists for
security issues; Configuration errors (including performance tunings) are usually reported to the
user-discussion forums. Therefore this study might be biased towards software bugs. However,
before a failure is resolved, it can be hard for users to determine the nature of the cause, therefore
our study still cover many configuration errors and security bugs. In addition, those quickly diag-
nosed failures that do not incur public reports will not be reflected in this study (however, since the
focus is on improving diagnosis, this bias is completely acceptable).

### Table 5.2: The number of sampled failures and the subset with failure-related log messages. A
failure is classified as “with logs” if any log point exists on the execution path between the fault to
the symptom. *: the total number of valid failures that have been fixed in the recent five years in
the Bugzilla.

<table>
<thead>
<tr>
<th>Software</th>
<th>#Failures</th>
<th>#Failures</th>
<th>with logs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>population*</td>
<td>sampled</td>
<td></td>
</tr>
<tr>
<td>Apache</td>
<td>838</td>
<td>65</td>
<td>24 (37%)</td>
</tr>
<tr>
<td>Squid</td>
<td>680</td>
<td>50</td>
<td>20 (40%)</td>
</tr>
<tr>
<td>Postgres</td>
<td>195</td>
<td>45</td>
<td>24 (53%)</td>
</tr>
<tr>
<td>SVN</td>
<td>321</td>
<td>45</td>
<td>25 (56%)</td>
</tr>
<tr>
<td>Coreutils</td>
<td>212</td>
<td>45</td>
<td>15 (33%)</td>
</tr>
<tr>
<td>Total</td>
<td>2246</td>
<td>250</td>
<td>108 (43%)</td>
</tr>
</tbody>
</table>
Another concern is that some very hard failures that never got fixed might be missed by this study. However, as the studied software projects are well maintained, severity is the determining factor of the likelihood for a failure to be fixed. High severity failures, regardless of its diagnosis difficulty, are likely to be diagnosed and fixed. Therefore the failures that this study misses are likely those not-so-severe ones.

Finally, there is the possibility of observer error in the qualitative aspects of our study. To minimize such effects, two inspectors separately investigated every failure and compared their understandings with each other. To ensure correctness, this process was also repeated multiple times. This entire failure study took 4 inspectors 4 months of time.

Overall, while this study cannot make universal claims about logging practices in all software systems, I believe that this study provides insight about the efficacy and pitfalls of such practices in many development environments (and in particular for open source software written in C/C++).

5.3.1 Failure Characterization

Across each program this study extracts its embedded log messages and then analyze how these messages relate to the failures we identified manually. This study decomposes these results through a series of findings for particular aspects of logging behavior. Overall, it will show that even though appropriate logging can cut diagnosis time in half, large numbers of failure cases are not logged (even when the software detects the error that lead to the failure).

• **Finding 1:** Under the default verbosity mode\(^2\), almost all (97%) logging statements in our examined software are error and warning messages (including fatal ones). This result is shown in Table 5.1. This supports our expectation that error/warning messages are frequently the only evidence for diagnosing a system failure in the field.

• **Finding 2:** Log messages produce a substantial benefit, reducing median diagnosis time between 1.4 and 3 times (on average 2.2X faster).

This result is shown in Figure 5.3, which supporting this dissertation’s motivating hypothesis about the importance of appropriate logging. This result is computed by measuring each failure’s “duration” (i.e., the duration from the time the failure is reported to the time a correct patch is provided). The failure set is further divided into two groups: (1) those with failure-related log messages reported and (2) those without, and compare the median diagnosis time between the two groups. Obviously, some failures might be easier to diagnose than the others, but since the

\(^2\)This entire Chapter assumes the default verbosity mode (i.e., no verbosity), which is the typical setting for production runs. This is because verbose logging typically incurs a large overhead (e.g., over 90% throughput degradation in Squid). Therefore they are usually not enabled during production runs. Indeed, once a failure occur, developers might ask the user to reproduce the failure with verbose logging enabled. However, such hand-shake itself is undesirable for the users in the first place.
sample set is relatively large I believe the results will reflect any gross qualitative patterns (note, our results may be biased if the difficulty of logging is strongly correlated with the future difficulty of diagnosis, although we are unaware of any data or anecdotes supporting this hypothesis).

• **Finding 3:** *the majority (57%) of failures do not have failure-related log messages,* leaving support engineers and developers to search for root causes “in the dark”.

This result is shown in Table 5.2. Next, I will further zoom in to understand why those cases did not have log messages and whether it is hard to log them in advance.

![Figure 5.4: Fault manifestation for the sampled failures, and the number of cases with failure-related log messages. (=x+y): x failures from detected errors and y failures from undetected errors. “Log: N”: N cases have failure-related log messages.](image)

- **Finding 4:** *Surprisingly, the programs themselves have caught early error-manifestations in the majority (61%) of the cases.* The remaining 39% are undetected until the final failure point.

This is documented in Figure 5.4, which shows how the sampled failures map to the error manifestation model presented in Section 5.2. Table 5.3 breaks them down by application, where the behavior is generally consistent. This indicates that programmers did reasonably well in anticipat-
Table 5.3: Error manifestation characteristics of examined software. All detected errors were caught by generic exception checks such as those in Table 5.5. Some undetected errors could have been detected in the same way.

However, as shown in Figure 5.4 programmers do not comprehensively log these detected errors, leaving 30% of them unlogged. Fortunately, the result also indicates that log automation can be a rescue—at least 61% of failures manifest themselves through explicitly detected exceptions, which provide natural places to log the errors for postmortem diagnosis.

Table 5.4: Logging practices when general errors are detected.

Further drilling down, let’s consider two categories of failures for which programmers themselves detected errors along the fault propagation path: early termination and incorrect handling. As shown in Table 5.4, the vast majority (90%) of the first category log the errors appropriately (10% miss this easy opportunity and impose unnecessary obstacles to debugging; Figure 5.1 documents one such omission in Apache). Logging overhead is not a big concern in these cases since the programs subsequently terminate.

For the second category (i.e., those failure cases where programs decided to tolerate the errors but unfortunately did so incorrectly), the majority of the cases did not log the detected errors. Table 5.4 also shows that Postgres and SVN are much more conservative in surviving detected errors. Among their 54 detected errors, developers chose early termination in 93% (50/54) of the
detected errors. In comparison, for the other three applications, only 63% of the detected errors terminate the executions. The reason might be that data integrity is the first class requirement for Postgres and SVN—when errors occur, they seldom allow executions to continue at the risk of data damaging.

```c
apr_status_t apr_file_read(apr_file_t *file, void *buf, ...){
do{
    rv = read(file->des, buf, ...);       /* non-blocking */
} while (rv == -1 && errno==EINTR);
if(rv == -1 && errno == EWOULDBLOCK){
arv = apr_poll(file, NULL, 1);
if(arv != SUCCESS)
    return arv;
... /* Apache, readwrite.c */
}
```

Figure 5.5: Incorrect error handling in Apache resulting in a hang. The root cause is a deadlock bug that prevented the producer process from writing to the consumer process via a pipe. Apache tries to handle an “empty pipe error” from read() by keeping polling on the pipe to get cgi output. It would work well to tolerate temporary slow down on network, but in this corner case where a sender cannot fill in the empty pipe (with cgi output) until Apache first drains another pipe (to consume cgi error), Apache can hang and become no responsive.

- **Finding 5:** 41 of the 250 randomly sampled failures are caused by incorrect or incomplete error handling. Unfortunately, most (85%) of them do not have logs. This indicates that developers should be conservative in error handling code: at least log the detected errors since error handling code is often buggy. For example, Figure 5.5 shows an incorrect error handling in Apache. The details are explained in the Figure caption. Similar to the Squid example in Figure 5.2, the developers did not log the detected error, and consequently took a long time to diagnose the occurred failure.

Adding together the two categories, there are a total of 46 cases that did not log detected errors. In addition, there are also 39 failures shown in Table 5.3 in which the programs could have detected the error via generic checks (e.g., system call error returns) but did not. Overall, among all the failure cases without log messages (142 in Table 5.2), there were clear opportunities for 60% (85/142) to log failure-related error information.

- **Finding 6:** Among the 142 failures without log messages, there were obvious logging opportunities for 60% (85) of them. In particular, 54% (46) of them already did such checks, but did not log the detected errors.

**Room for improvement:** By logging these 85 cases with easy opportunities, 77% of the 250 randomly sampled failures could have error messages compared to 43% that currently are logged.
Logging Practice Recommendation: Overall, these findings suggest that it is worthwhile to conservatively log detected errors, regardless of whether there is error-handling code to survive or tolerate the errors.

5.3.2 Logging Generic Exceptions

<table>
<thead>
<tr>
<th>Generic Exception Conditions</th>
<th>Detected Errors</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>total</td>
<td>w/ logs</td>
</tr>
<tr>
<td>Function return errors</td>
<td>69 (45%)</td>
<td>50 (72%)</td>
</tr>
<tr>
<td>Exception signals(e.g., SIGSEGV)</td>
<td>22 (14%)</td>
<td>22 (100%)</td>
</tr>
<tr>
<td>Unexpected cases falling into default</td>
<td>27 (18%)</td>
<td>12 (44%)</td>
</tr>
<tr>
<td>Resource leak</td>
<td>1 (1%)</td>
<td>1 (100%)</td>
</tr>
<tr>
<td>Failed input validity check</td>
<td>17 (11%)</td>
<td>8 (47%)</td>
</tr>
<tr>
<td>Failed memory safety check</td>
<td>7 (4%)</td>
<td>7 (100%)</td>
</tr>
<tr>
<td>Abnormal exit/abort from execution</td>
<td>11 (7%)</td>
<td>8 (73%)</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>154</strong></td>
<td><strong>108 (70%)</strong></td>
</tr>
</tbody>
</table>

Table 5.5: Logging practices for common exceptions.

Table 5.5 documents these generic exception patterns, many of which are checked by the studied programs but are not logged. I will explain some of them and highlight good practices that are encountered in the study.

Figure 5.6: SVN’s good logging practices for checking and logging function return errors.

1) Function return errors: It is a common practice to check for function (e.g., system call) return errors. In our study, 45% of detected errors were caught via function return values as shown on Table 5.5. However, a significant percentage (28%) of them did not log such errors. For example,
in Figure 5.1, Apache developers checked if the `ap_pcfg_openfile()`’s return value reflected an error (which eventually calls `open`), but did not log it before terminating the program.

**Good practice:** SVN uniformly logs function return errors. First, as shown in Figure 5.6, almost all SVN function calls are made through a special macro `SVN_ERR`, which checks for error return. Second, if a function returns an error to its caller, it prepares an error message in a buffer, `err->message`. Every error is eventually returned back to `main` through the call path via `SVN_ERR` and then `main` prints out the error message. Consequently, as shown in Table 5.4, almost all exceptions detected by SVN are logged before early termination.

(2) **Exception signals:** In general, many server programs register their own signal handlers to catch fatal signals (e.g., `SIGSEGV`, `SIGTERM`). In this study, about 14% of detected errors were caught by the programs’ own signal handlers, and fortunately all were logged.

![Figure 5.7: Logging practices for exception signals.](image)

However, all examined software (except for `squid`) only logs signal names. Figure 5.7 compares the logging practices in three of them: (a) Coreutils does not have a signal handler. OS prints a generic “segmentation fault” message. (b) Postgres’s log does not provide much better information than the default OS’s signal handler. (c) **Good practice:** squid logs system status and context information such as CPU and memory usage, as well as the stack frames, when catching exception signals, providing useful diagnostic information.

(3) **Unexpected cases falling through into default:** Sometimes when programs fail to enumerate all possible cases in a switch statement, the execution may unexpectedly fall through into the base “default” case, and lead to a failure. In our study, 18% of detected errors belong to this category, but only 44% of them are logged. An example is given in Figure 5.8(a). Originally the ’\0’ case was not considered at all. Consequently, when this character was encountered, execution mistakenly fell through to default, resulting into a hang without any log messages. **Good practice:** In contrast,
the Squid code shown in Figure 5.8(b) logs the switch variable when the default case is executed, making diagnosis much easier.

Motivated from the significance of this category (18%), we recommend developers to log the default case of switch statement, whenever the default case is unexpected (if program styles where the default case is common, the associated logging overhead may impose too great a burden).

Figure 5.8: Logging for unexpected cases falling into default.

(4) Resource leak (not exhaustion): Resource leaks are more difficult to check and log. Although programmers can easily detect resource exhaustion (e.g., out of memory), the error message at an exhaustion point may not be useful to understand the root cause. Instead, it is more useful to monitor resource usage (balanced against the potential overhead of doing so).

Good practice: Figure 5.9 shows an example of a good logging practice for resource leak in

![Figure 5.9: Squid’s good logging practice for resource leak. Users can use a tool to access logs in memory on demand.](image)
Squid. First, Squid monitors the usage of file descriptors (e.g., reads/writes) and their types (e.g., socket, file, etc.), and logs them into a memory buffer. Second, Squid lets users use a special tool to access and print the in-memory logs on demand. Third, to check resource exhaustion, Squid compares the number of available file descriptors against a threshold (i.e., RESERVED_FD), and logs an overflow warning and then finally an exhaustion error.

(5) Other generic exception conditions: When programmers check input validity, they need to log the failed check. As shown in Table 5.5, 11% of the detected errors belong to this pattern, but not all of them were logged. Similarly, after memory safety checks such as array bound checks and null pointer dereference checks, failed cases also need to be logged. Programmers seem to follow this practice reasonably well.

Finally, when programmers decide to abort or terminate a program early for any reason, we recommend logging the events leading to the termination. In our study, 11 failures happened in this manner but only 8 have logs.

5.3.3 Logging for Hard-to-check Failures

<table>
<thead>
<tr>
<th>Coverage Criteria</th>
<th>Hard-to-check Failure Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statement cov.*</td>
<td>10 (18%)</td>
</tr>
<tr>
<td>Decision cov.</td>
<td>12 (21%)</td>
</tr>
<tr>
<td>Condition cov.</td>
<td>2 (4%)</td>
</tr>
<tr>
<td>Weak mutation</td>
<td>4 (7%)</td>
</tr>
<tr>
<td>Mult. cond. cov.</td>
<td>2 (4%)</td>
</tr>
<tr>
<td>Loop cov.</td>
<td>1 (2%)</td>
</tr>
<tr>
<td>Concurr. cov.</td>
<td>1 (2%)</td>
</tr>
<tr>
<td>Perf. profiling</td>
<td>1 (2%)</td>
</tr>
<tr>
<td>Functional cov.</td>
<td>34 (60%)</td>
</tr>
<tr>
<td><strong>Total failures</strong></td>
<td><strong>57</strong></td>
</tr>
</tbody>
</table>

Table 5.6: The number of hard-to-check failures that could have been caught during testing, assuming 100% test coverage with each criteria. *: can also be detected by decision coverage test.

As shown earlier in Table 5.3, 57 failures are hard to detect via generic exception checks. I will refer them as hard-to-check errors³. When a production failure occurs, it is usually due to an unusual input or environment triggering some code paths or state combinations that are not covered during in-house testing. Table 5.6 shows that 21% of the 57 hard-to-check failure cases execute some branch edges that likely have never been executed during testing (otherwise, the bugs on

³Please note that hard-to-check errors do not mean necessarily harder to diagnose than other errors. By studying the diagnosis time, it shows that in Apache and SVN, hard-to-check errors took slightly less time, and PostgreSQL and Squid showed the opposite.
those edges would definitely have been exposed). Therefore, if we log on those branch decisions that have not been covered during testing, i.e., cold paths, it would be useful for diagnosis. Of course, special care needs to be taken if some cold paths show up too frequently during runtime.

- **Finding 7:** Logging for untested code paths would collect diagnostic information for some of hard-to-check failures.

## 5.4 Errlog: A Practical Logging Tool

<table>
<thead>
<tr>
<th>Exception pattern</th>
<th>How to identify in source code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function return error</td>
<td>Mechanically search for libc/system calls. If a libc/system call’s error return value is not checked by the program, Errlog injects new error checking code. Such a check won’t incur too much overhead as it is masked by the overhead of a function call.</td>
</tr>
<tr>
<td>Failed memory safety check</td>
<td>Search for checks for null pointer dereference and out-of-bound array index. If no such safety check exists, Errlog does NOT add any check due to false positive concerns.</td>
</tr>
<tr>
<td>Abnormal exit</td>
<td>Search for “abort, exit, exit”. The constraint $EC$ is “true”.</td>
</tr>
<tr>
<td>Exception signals</td>
<td>Intercept and log abnormal signals. Our logging code uses memory buffer and is re-entrant.</td>
</tr>
<tr>
<td>Unexpected cases falling into default</td>
<td>Search for the “default” in a switch statement or a switch-like logic, such as if.. else if.. else..., where at least the same variable is tested in each if condition.</td>
</tr>
<tr>
<td>Invalid input check</td>
<td>Search for text inputs, using a simple heuristic to look for string comparisons (e.g., strcmp). The exception is the condition that these functions return “not-matched” status. In our study, 47% of the “invalid input checks” are from these standard string matching functions.</td>
</tr>
<tr>
<td>Resource leak</td>
<td>Errlog monitors resource (memory and file descriptor) usage and logs them with context information. Errlog uses exponential-based sampling to reduce the overhead (Section 5.4.3).</td>
</tr>
</tbody>
</table>

Table 5.7: Generic exception patterns searched by Errlog. These patterns are directly from our findings in Table 5.5 in Section 5.3.

Driven by the findings from the empirical study, an automatic logging tool called Errlog is built, which analyzes the source code to identify potential unlogged exceptions (abnormal or unusual conditions), and then inserts log statements. Therefore, Errlog can automatically enforce good logging practices. We implement our source code analysis algorithms using the Saturn [ABD+07] static analysis framework.

---

4Due to software’s complexity, cost of testing, and time-to-market pressure, complex systems can rarely achieve 100% test coverage.
Errlog faces three major challenges: (1) Where are such potential exceptions? (2) Has the program itself checked for the exception? If so, has the program logged it after checking it? (3) Since not every potential exception may be terminal (either because the program has mechanisms to survive it or it is not a true exception at all), how to avoid significant performance overhead without missing important diagnostic information?

To address the first challenge, Errlog follows the observations from the characterization study. It identifies potential exceptions by mechanically searching in the source code for the seven generic exception patterns in Table 5.5. In addition, since many other exception conditions are program specific, Errlog further “learns” these exceptions by identifying the frequently logged conditions in the target program. Moreover, it also optionally identifies untested code area after in-house testing.

For the second challenge, Errlog checks if the exception check already exists, and if so, whether a log statement also exists. Based on the results, Errlog decides whether to insert appropriate code to log the exception.

Since not every case identified by Errlog is a true exception, it may introduce logging overhead if this case happens frequently at runtime. To address the third challenge, Errlog provides three logging modes for developers to choose from, based on their preferences for balancing the amount of log messages versus performance overhead: Errlog-DE for logging definite exceptions, Errlog-LE for logging definite and likely exceptions, and Errlog-AG for aggressive logging. Moreover, Errlog’s runtime logging library uses dynamic sampling to further reduce the overhead of logging without losing too much logging information.

Usage  Users of Errlog only need to provide the name of the default logging functions used in each software project. For example, the following command is to use Errlog on the CVS version control system:

```
Errlog --logfunc="error" path-to-CVS-src
```

where error is the logging library used by CVS. Errlog then automatically analyzes the code and modifies it to insert new log statements. Errlog can also be used as a tool that recommends where to log (e.g., a plug-in to the IDE) to the developers, allowing them to insert logging code to make the message more meaningful. After logging statements are inserted, developers can review and revise them. For example, they may want to make the mechanically inserted text more readable.

### 5.4.1 Exception Identification

In this step, Errlog scans the code and generates the following predicate: \textit{exception(program\_point P, constraint EC)}, where \textit{P} is the program location of an exception check, and \textit{EC} is the constraint that causes the exception to happen. In the example shown in Figure 5.2, \textit{P} is the source code
location of “if (status!=COMM_OK)”, and $EC$ is status!=COMM_OK. $EC$ is used later to determine under which condition $Errlog$ should log the exception and whether the developer has already logged the exception.

**Search for generic exceptions** Table 5.7 shows the generic exception patterns $Errlog$ automatically identifies, which are directly from the findings in our characterization study.

**Learning Program-Specific Exceptions**

$Errlog$-LE further attempts to identify program-specific exceptions *without any program-specific knowledge*. The intuition is that if a condition is frequently logged by programmers in multiple code locations, it is likely to be “log-worthy”. For example, the condition status!=COMM_OK in Figure 5.2 is a *squid*-specific exception that is frequently followed by an error message. Similar to previous work [ECH+01] that statically learns program invariants for bug detection, $Errlog$-LE automatically learns the conditions that programmers log on more than two occasions. To avoid false positives, $Errlog$ also checks that the logged occasions outnumber the unlogged ones.

**The need for control and data flow analysis** It is non-trivial to correctly identify log-worthy conditions, since simple syntax based condition comparison is not sufficient.

For example, the exception condition in Figure 5.10 is that `pcre_malloc` returns `NULL`, not `tmp==NULL`. $Errlog$ first analyzes the control-flow to identify the condition that immediately leads to an error message. It then analyzes the data-flow, in a backward manner, on each variable involved in this condition to identify its source. However, such data-flow analysis cannot be carried arbitrarily deep as doing so will likely miss the actual exception source. For each variable $a$, $Errlog$’s data-flow analysis stops when it finds a *live-in* variable as its source, i.e., a function parameter, a global variable, a constant, or a function return value. In Figure 5.10, $Errlog$ first identifies the condition that leads to the error message being `tmp==NULL`. By analyzing the data-flow of `tmp`, it further finds its source being the return value of `pcre_malloc`. Finally, it replaces the `tmp` with `pcre_malloc()` and derives the correct error condition, `pcre_malloc()==NULL`. Similarly, the error condition `status!=COMM_OK` in Figure 5.2 is learnt because `status` is a function parameter.

**Identifying helper logging functions** In order to learn the frequent logging pattern, $Errlog$ needs
to know all log printing functions for each program. *Errlog* only requires developers to provide the name of the default logging function. However, in all the large software projects in the characteristic study, there are also many helper logging functions that simply wrap around the default ones. *Errlog* identifies them by recursively analyzing each function in the bottom-up order along the call graph. If a function $F$ prints a log message under the condition $\texttt{true}$, $F$ is added to the set of logging functions.

**Explicitly specified exceptions (optional)** *Errlog* also allows developers to explicitly specify domain-specific exception conditions in the form of code comments right before the exception condition check. Once the developers informs *Errlog* of the exception conditions, *Errlog* systematically checks if all appearances of such exception condition are logged, and inserts logging code if necessary. Our experiments are conducted without this option.

**Identifying Untested Code Area (optional)** *Errlog-AG* further inserts log points for code regions not covered by in-house testing. Test coverage tool GNU *gcov* [GCOB] and the branch decision coverage criteria are used. For each untested branch decision, *Errlog* instruments a log point. For multiple nested branches, *Errlog* only inserts a log point at the top level, and records relevant variable values to disambiguate the execution paths using LogEnhancer. This option is not enabled in our experiments unless otherwise specified.

### 5.4.2 Log Printing Statement Insertion

**Filter the exceptions already logged by a program** This is to avoid redundant logging, which can result in overhead and redundant messages. Determining if an exception $E$ has already been logged by a log point $L$ is challenging. First, $L$ may not be in the basic block immediately after $E$. For example, in Figure 5.10, the exception check and its corresponding log point are far apart. Therefore, simply searching for $L$ within the basic block following $E$ is not enough. Second, $E$ might be logged by the caller function via an error return code. Third, even if $L$ is executed when $E$ occurs, it might not indicate that $E$ is logged since $L$ may be printed regardless of whether $E$ occurs or not.

*Errlog* uses precise path sensitive analysis to determine whether an exception has been logged. For each identified exception $(P, EC)$, *Errlog* first checks whether there is a log point $L$ within the same function $F$ that: i) will execute if $EC$ occurs, and ii) there is a path reaching $P$ but not $L$ (which implies that $L$ is not always executed regardless of $EC$). If such an $L$ exists, then $EC$ has already been logged. To check for these two conditions, *Errlog* first captures the path-sensitive conditions to reach $P$ and $L$ as $C_P$ and $C_L$ respectively. It then turns the checking of the above two conditions into a satisfiability problem by checking the following using a SAT solver:
1. \( C_P \land EC \land \neg C_L \) is not satisfiable.

2. \( C_P \land \neg C_L \) is satisfiable.

The first condition is equivalent to i), while the second condition is equivalent to ii). A SAT solver is used to check for these conditions.

If no such log point exists, \textit{Errlog} further checks if the exception is propagated to the caller via return code. It checks if there is a return statement associated with \( EC \) in a similar way as it checks for a log point. It remembers the return value, and then analyzes the caller function to check if this return value is logged or further propagated. Such analysis is recursively repeated in every function.

\textbf{Log placement} If no logging statement is found for an exception \( E \) from the analysis above, \textit{Errlog} inserts its own logging library function, \( \text{Ellog}(\logID) \), into the basic block after the exception check. If no such check exists, \textit{Errlog} also adds the check at appropriate places based on the exception pattern. For example, for an \texttt{unlink} system call whose return value is not checked, \textit{Errlog} replaces the original call with the following expression: \( \text{tmp=unlink()}, \text{tmp==}-1 \ ? \text{Elog()} : \text{NULL, \_tmp} \).

Each logging statement records (i) a log ID unique to each log point, (ii) the call stack, (iii) casually-related variable values identified using \textit{LogEnhancer}, (iv) a global counter that is incremented with each occurrence of any log point, to help postmortem reconstruction of the message order. For each system-call return error, the \texttt{errno} is also recorded. No static text string is printed at runtime. \textit{Errlog} will compose a postmortem text message by mapping the log ID and \texttt{errno} to a text string describing the exception. For example, \textit{Errlog} would print the following message for an \texttt{open} system-call error: \texttt{"open system call error: No such file or directory: ./filepath ..."}. Developers can also manually edit the message to make it more meaningful.

\textbf{5.4.3 Run-time Logging Library}

Due to the lack of run-time information and domain knowledge during our static analysis, \textit{Errlog} may also log non-exception cases, especially with \textit{Errlog-LE} and \textit{Errlog-AG}. If these cases occur frequently at run time, the time/space overhead becomes a concern.

To address this issue, \textit{Errlog}’s run-time logging library borrows the idea of adaptive sampling [HC04]. It exponentially decreases the logging rate when a log point \( L \) is reached from the same calling context many times. The rationale is that frequently occurred conditions are less likely to be important exceptions; and even if they are, it is probably useful enough to only record its first, second, \( 2^n \)th dynamic occurrences, instead of every occurrence. To reduce the possibility of missing true exceptions, we also consider the whole context (i.e., the call stack) instead of just
each individual log point. For each calling context reaching each $L$ we log its $2^n$th dynamic occurrences. We further differentiate system call return errors by the value of `errno`, which stores the standard error code. For efficiency, `Errlog` logs into in-memory buffers and flushes them to disk when they become full, execution terminates, and when receiving user defined signals.

Note that comparing with other buffering mechanisms such as “log only the first/last N occurrences”, adaptive sampling offers a unique advantage: the printed log points can be postmortem ranked in the reverse order of their occurrence frequencies, with the intuition that frequently logged ones are less likely true errors.

At run-time, when the log point $P$ is triggered, `Errlog`’s logging library first unwinds the current call stack and use it together with the Log ID of $P$ to find an occurrence counter. It then increment the counter, and if it is the power of 2, logs the content into a buffer. Since some server applications fork a new process to serve new request, this counter is shared among all the forked and parent processes. Given the run-time log recorded by `Errlog`, developers can further rank different messages based on their recency to the failure or the number of times it is logged, with the intuition that frequently logged ones are less likely errors.

## 5.5 In-lab Experiment

`Errlog` is evaluated using both in-lab experiments and a controlled user study. This section presents the in-lab experiments. In addition to the five software projects used in the characterization study, `Errlog` is further evaluated with five more applications as shown in Table 5.8.

<table>
<thead>
<tr>
<th>App.</th>
<th>description</th>
<th>LOC</th>
<th>#Default Log Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVS</td>
<td>version cont. sys.</td>
<td>111K</td>
<td>1151</td>
</tr>
<tr>
<td>OpenSSH</td>
<td>secure connection</td>
<td>81K</td>
<td>2421</td>
</tr>
<tr>
<td>lighttpd</td>
<td>web server</td>
<td>54K</td>
<td>813</td>
</tr>
<tr>
<td>gzip</td>
<td>comp/decom. files</td>
<td>22K</td>
<td>98</td>
</tr>
<tr>
<td>make</td>
<td>builds programs</td>
<td>29K</td>
<td>134</td>
</tr>
</tbody>
</table>

Table 5.8: The new software projects used to evaluate `Errlog`, in addition to the five examined in the characterization study.

### 5.5.1 Coverage of Existing Log Points

It is hard to objectively evaluate the usefulness of log messages added by `Errlog` without domain knowledge. However, one objective evaluation is to measure how many of the existing log points,
added manually by developers, can be added by *Errlog* automatically. Such measurement could evaluate how much *Errlog* matches domain experts’ logging practice.

Note that while Section 5.3 suggests that the current logging practices miss many logging opportunities, it does not imply that existing log points are unnecessary. On the contrary, existing error messages are often quite helpful in failure diagnosis as they were added by domain experts, and many of them were added in the form of after-thoughts. This is confirmed by the benefit of logging: existing log messages would reduce the diagnosis time by 2.2X. Therefore, comparing with existing log points provides an objective measurement on the effectiveness of *Errlog*.

![Figure 5.11: Coverage of existing log points by *Errlog*. For *Errlog*-LE, we break down the coverages into log points identified by generic exceptions and those learned by frequent logging patterns. AG has similar coverages as LE.](image)

Figure 5.11 shows that *Errlog*, especially with *Errlog*-LE and *Errlog*-AG, can automatically cover an average of 84% of existing log points across all evaluated software. It indicates that *Errlog* can achieve almost comparable effectiveness as developers’ manual efforts in terms of logging quantity, and does so automatically. In comparison, *Errlog*-DE logs only definite errors and achieves an average of 52% coverage, still quite reasonable since on average it adds less than 1% overhead. Of course, a mechanically added log message may not be as meaningful as a manually added one. Therefore a good hybrid solution is to first use *Errlog* to automatically add log statements, and then developers can revise each to make it more meaningful.

Comparing the three modes, *Errlog*’s default mode, *Errlog*-LE, is the best to balance between coverage and overhead (Table 5.11). *Errlog*-DE logs only definite errors and achieves an average of 52% coverage (36%–76% across all applications), still quite reasonable since on average it adds less than 1% overhead. With *Errlog*-LE, 84% of the log points in all these software are covered. *Errlog*-AG performs the same as *Errlog*-LE for comparing against manual efforts.

Comparing different applications, with generic exception logging, *Errlog*-LE has the least coverage with PostgreSQL and SVN, 58% and 60%, respectively, since these softwares have more domain-specific log messages. Fortunately, over 20% of total existing log messages including many domain-specific ones could still be covered via *Errlog*’s learning from frequent logging pat-
terns. The similar results are shown with OpenSSH, where its own cryptography functions’ return errors are frequently logged but ours could not handle them automatically.

## 5.5.2 Additional Log Points

<table>
<thead>
<tr>
<th>App.</th>
<th>Erlog-DE</th>
<th>Erlog-LE</th>
<th>Erlog-AG</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>func. ret.</td>
<td>mem. safe.</td>
<td>abno. exit</td>
</tr>
<tr>
<td>Apache</td>
<td>30</td>
<td>41</td>
<td>9</td>
</tr>
<tr>
<td>Squid</td>
<td>393</td>
<td>112</td>
<td>29</td>
</tr>
<tr>
<td>Postgres</td>
<td>619</td>
<td>166</td>
<td>28</td>
</tr>
<tr>
<td>SVN</td>
<td>33</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>Coreutil cp</td>
<td>34</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>CVS</td>
<td>1109</td>
<td>360</td>
<td>23</td>
</tr>
<tr>
<td>OpenSSH</td>
<td>714</td>
<td>31</td>
<td>26</td>
</tr>
<tr>
<td>lighttpd</td>
<td>171</td>
<td>16</td>
<td>30</td>
</tr>
<tr>
<td>gzip</td>
<td>45</td>
<td>3</td>
<td>32</td>
</tr>
<tr>
<td>make</td>
<td>339</td>
<td>6</td>
<td>16</td>
</tr>
<tr>
<td>Total</td>
<td>3487</td>
<td>745</td>
<td>203</td>
</tr>
</tbody>
</table>

Table 5.9: Additional log points added by Erlog. The “total” of LE and AG include DE and DE+LE, respectively, and are compared to the number of existing log points (Table 5.1 and 5.8). Note that most of these log points are not executed during normal execution.

In addition to the existing log points, Erlog also adds new log points, shown in Table 5.9. Even though Erlog-LE adds 0.1X–3.1X additional log points, they only cause an average of 1.4% overhead (Section 5.5.3) because most of them are not triggered when the execution is normal.

Not surprisingly, Erlog adds the smallest number of log points to SVN (only 0.1X). As discussed in Section 5.3.2, SVN has a good exception checking and logging practices, but Erlog-LE still identifies new 105 potential exceptions, which makes SVN more diagnosable. Specifically, 30 of them are possible return errors from apr_palloc() memory allocation. According to the email discussions with SVN developers, they were not bothered to even check this, since they had no plan to tolerate it and thought that simply letting SVN crash would be better for data integrity. However, logging this would make postmortem diagnosis much easier, and Erlog does this automatically.

**Logging for untested branch decision** Table 5.10 shows Erlog-AG’s optional logging for untested branch decisions, which is not included in the results above. Postgres, SVN and Coreutils included their test cases into each release. For Apache, we contacted the developers and obtained their testing framework. Erlog logs each untested branch decision. When multiple branch decisions are untested in the same function, and their conditions have dependencies, Erlog further analyzes the function to combine the log points.
### Table 5.10: Optional logging for untested branch decisions.

<table>
<thead>
<tr>
<th>App.</th>
<th>Uncovered decisions</th>
<th># log points</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>57.0% (2915)</td>
<td>655</td>
</tr>
<tr>
<td>Postgres</td>
<td>51.7% (51396)</td>
<td>11810</td>
</tr>
<tr>
<td>SVN</td>
<td>53.7% (14858)</td>
<td>4051</td>
</tr>
<tr>
<td>Coreutils</td>
<td>62.3% (9238)</td>
<td>2293</td>
</tr>
</tbody>
</table>

### 5.5.3 Performance Overhead

<table>
<thead>
<tr>
<th>Software</th>
<th>Adaptive sampling*</th>
<th>No sampling</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DE</td>
<td>LE</td>
</tr>
<tr>
<td>Apache</td>
<td>&lt;1%</td>
<td>&lt;1%</td>
</tr>
<tr>
<td>Squid</td>
<td>&lt;1%</td>
<td>1.8%</td>
</tr>
<tr>
<td>Postgres</td>
<td>1.5%</td>
<td>1.9%</td>
</tr>
<tr>
<td>SVN</td>
<td>&lt;1%</td>
<td>&lt;1%</td>
</tr>
<tr>
<td>cp</td>
<td>&lt;1%</td>
<td>&lt;1%</td>
</tr>
<tr>
<td>CVS</td>
<td>&lt;1%</td>
<td>&lt;1%</td>
</tr>
<tr>
<td>Openssh scp</td>
<td>2.0%</td>
<td>4.6%</td>
</tr>
<tr>
<td>lighttpd</td>
<td>&lt;1%</td>
<td>&lt;1%</td>
</tr>
<tr>
<td>gzip</td>
<td>&lt;1%</td>
<td>&lt;1%</td>
</tr>
<tr>
<td>make</td>
<td>3.9%</td>
<td>4.0%</td>
</tr>
<tr>
<td>Average</td>
<td>1.1%</td>
<td>1.4%</td>
</tr>
</tbody>
</table>

* By default, Errlog uses adaptive sampling. The overhead without using sampling is only to demonstrate the effect of adaptive sampling.

Errlog’s logging overhead is measured using the software’s normal execution. Server performance is measured in peak-throughput. Web servers including Apache httpd, squid, and lighttpd are measured with ab [Apa]; PostgreSQL is evaluated with pgbench [PGB] using the select-only workload; SVN and CVS are evaluated with a combination of check-out, merge, copy, etc.; OpenSSH is evaluated by using scp to repeatedly transfer files; gzip and cp are evaluated with processing large files; make is evaluated by compiling PostgreSQL.

Table 5.11 shows Errlog’s logging overhead during the normal execution. For all evaluated software, the default Errlog-LE imposes an average of 1.4% run-time overhead, with a maximum of 4.6% for scp. The most aggressive mode, Errlog-AG, introduces an average of 2.1% overhead and a maximum of 4.8%. This is because of Errlog’s adaptive log sampling and the use of in-memory log buffer. The maximum runtime memory footprint imposed by Errlog is less than 1MB. Errlog allocates one chunk of memory (1MB) at a time and implements its own memory management. Additional chunks of memory are allocated on demand. In all of the experiments, 1MB is enough for all the applications, with the maximum usage being 228KB.
scp and make have larger overhead than others in Table 5.11. It is because scp is relatively CPU intensive (lots of encryptions) and also has a short execution time. Compared to I/O intensive workloads, the relative logging overhead added by Errlog becomes more significant in CPU intensive workloads. Moreover, short execution time may not allow Errlog to adapt the sampling rate effectively. make also has relatively short execution time.

<table>
<thead>
<tr>
<th>Log pts.</th>
<th>function ret.</th>
<th>memory safety</th>
<th>switch-default</th>
<th>input check</th>
<th>learned errors</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>8</td>
<td>5</td>
<td>7</td>
<td>10</td>
<td></td>
<td>35</td>
</tr>
</tbody>
</table>

Table 5.12: Noisy log points exercised during correct executions.

**Noisy messages** More log messages are not always better. However, it is hard to evaluate whether each log point captures a true error since doing so requires domain expertise. To address this challenge, those log points that are executed during the performance testing are simply treated as noisy messages, as there were no visible failures in the performance testing. Among the five software projects used in the characteristic study, only a total of 35 log points (out of 405 error condition checks) are executed, between 3-12 for each application. For example, given code “if(stat(..)==-1) {Elog(..);}”, the if(stat(..)==-1) is an error condition check. The log point Elog is only executed if the error condition check succeeds (stat returns -1). A switch statement is another example of error condition check. Examples of error checks include a system call or a switch statement that could potentially fall into error condition. Table 5.12 breaks down these 35 log points by different patterns. Examples of these include using the error return of stat system call to verify a file’s non-existence in normal executions. Since Errlog uses adaptive sampling, the size of run-time log is small (less than 1MB).

Please note that since these conditions that resulted in noisy messages are actually normal, they all occurred frequently in normal execution, thus their corresponding log buffers have multiple entries. Such high frequency messages will be ranked lower than those true exceptional messages that occurred much more rarely.

**Sampling overhead comparison** The efficiency of adaptive sampling is further evaluated by comparing it with “no sampling” in Table 5.11. “No sampling” logging records every occurrence of executed log points into memory buffer and flushes it to disk when it becomes full or execution ends. Errlog-AG is not evaluated with “no sampling” as it is more reasonable to use sampling to monitor resource usage.

Adaptive sampling effectively reduces Errlog-LE’s overhead from no-sampling’s 9.4% to 1.4%. The majority of the overhead is caused by a few log points on an execution’s critical paths. Sampling can exponentially reduce the logging rate of such frequently occurred log points. For example, in Postgres, the index-reading function, where a lock is held, contains a log point. By decreasing the logging rate, adaptive sampling successfully reduces no-sampling’s 40.1% over-
head to 1.9% compared to logging without sampling. In comparison, the effect of sampling is less obvious for make: reducing 6.8% to 4.0% for LE, where its short execution time is not sufficient for adaptive sampling to adjust its sampling rate.

**Analysis time** Since *Errlog* is used off-line to add log statements prior to software release, the analysis time is less critical. Table 5.13 shows *Errlog*’s analysis time. *Errlog* takes less than 41 minutes to analyze each evaluated software except for postgres, which took 3.8 hours to analyze since it has 1 million LOC. Since *Errlog* scans the source code in one-pass, its analysis time roughly scales linearly with the increase of the code size. Additionally, the summary-based design allows it to be parallel or incrementally applied [ABD+07]. The memory usage in each case is below 2GB.

<table>
<thead>
<tr>
<th>Analysis Time (minutes)</th>
<th>apache 38</th>
<th>cvs 41</th>
<th>squid 34</th>
<th>openssh 15</th>
<th>postgres 230</th>
<th>lighttpd 15</th>
<th>gzip 9</th>
<th>cp 2</th>
<th>make 12</th>
</tr>
</thead>
</table>

Table 5.13: Analysis time

### 5.5.4 Real World Failures

Table 5.14: *Errlog*’s effect on the randomly sampled 238 real-world failure cases. *: 12 of our 250 examined failure cases cannot be evaluated since the associated code segments are for different platforms incompatible with our compiler.

<table>
<thead>
<tr>
<th>App.</th>
<th>Tot. fails</th>
<th>w/ existing logs</th>
<th>Errlog-</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DE</td>
<td>LE</td>
<td>AG</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Apache</td>
<td>58</td>
<td>18 (31%)</td>
<td>28 (48%)</td>
<td>43 (74%)</td>
<td>48 (83%)</td>
<td></td>
</tr>
<tr>
<td>Squid</td>
<td>45</td>
<td>15 (33%)</td>
<td>23 (51%)</td>
<td>37 (82%)</td>
<td>37 (82%)</td>
<td></td>
</tr>
<tr>
<td>Postgres</td>
<td>45</td>
<td>24 (53%)</td>
<td>26 (58%)</td>
<td>32 (71%)</td>
<td>34 (76%)</td>
<td></td>
</tr>
<tr>
<td>SVN</td>
<td>45</td>
<td>25 (56%)</td>
<td>30 (67%)</td>
<td>33 (73%)</td>
<td>33 (73%)</td>
<td></td>
</tr>
<tr>
<td>Coreutils</td>
<td>45</td>
<td>15 (33%)</td>
<td>28 (62%)</td>
<td>34 (76%)</td>
<td>37 (82%)</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>238*</td>
<td>97 (41%)</td>
<td>135 (57%)</td>
<td>179 (75%)</td>
<td>189 (79%)</td>
<td></td>
</tr>
</tbody>
</table>

To show the concrete benefit of how these log messages added by *Errlog* can help with diagnosis, four real-world failures are chosen as case studies.

**Case 1:** For the Apache failure case shown in Figure 5.5, *Errlog* automatically adds a log statement for the `read()` system call after checking its return value. At runtime, different log buffers are
used for different calling context and \texttt{errno}. In the case of \texttt{errno == EINTR}, which the program can tolerate correctly, \texttt{Errlog} logs the first few and then decreases the logging rate exponentially. In the case of \texttt{errno==EWOULDBLOCK}, whose first dynamic occurrence along the particular calling context would lead to a deadlock, \texttt{Errlog} logs the error, which would help the programmers easily discover that the error handling code is incomplete and needs to single out such a case.

**Case 2:** In an SVN failure (issue #3346) report, the user complained that SVN terminated without any log messages or core dump. Even worse, developers could not reproduce it and thus it took long time to diagnose. They learned that logging the abort() can greatly help the diagnosis, so they released a patch whose entire change was just to replace each abort() in SVN with \texttt{svn_errno-assert()}, a new macro that logs before abort(). \texttt{Errlog} can avoid this painful diagnosis experience by automatically adding such a log, exactly like the patch.

**Case 3:** \texttt{Errlog} automatically adds an error message for the Squid failure described in Figure 5.2 similar to the message added in the patch. \texttt{Errlog-LE} first learns that Squid frequently logs (10 times) under the condition \texttt{if(status!=COMM_OK)}. Then it found 2 places that do not log this condition, including the one shown in Figure 5.2, and automatically inserts a logging statement under each of them. With this log message, developers can immediately know the failure was caused by a DNS lookup failure.

**Case 4:** As for the failure case described in Figure 5.1, \texttt{Errlog} first found the error return value of the \texttt{open} system call in function \texttt{ap_pcfg_openfile} was not logged. But since the error return value was further returned to its caller, \texttt{Errlog} follows along the call chain to see if the error condition was logged by any of its callers. Eventually since it found none of the callers logged the error, it would insert a log statement within the function \texttt{ap_pcfg_openfile}, immediately after the \texttt{open} system call returned non-zero.

**Effectiveness of \texttt{Errlog} for Diagnosis** The usefulness of the added log messages in diagnosis is further evaluated by using SherLog (Chapter 2). Given log messages related to a failure, SherLog reconstructs the execution paths must/may have taken to lead to the failure. Table 5.15 shows the numbers and also percentages of the failures that \texttt{Errlog} has added log messages can help SherLog to pinpoint the exact execution path leading to the failure. Of course, without these log messages, it has no log message to start its inference. It shows that 80% of the new messages can help SherLog to successfully infer the root causes.

### 5.6 User Study

A controlled user study is further conducted to measure the effectiveness of \texttt{Errlog}. Table 5.16 shows the five real-world production failures used in this user study. Except for “\texttt{apache crash}”,
the other four failed silently. Failures are selected to cover diverse root causes (bugs and misconfigurations), symptoms (crash, accepting wrong input, rejecting correct input, etc.), and reproducibilities. 20 programmers are selected in this study (no co-author of this work is selected), who indicated that they have extensive and recent experience in C/C++.

<table>
<thead>
<tr>
<th>Name</th>
<th>Repro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>apache crash</td>
<td>✓</td>
<td>A configuration error triggered a NULL pointer dereference.</td>
</tr>
<tr>
<td>apache no-file</td>
<td>✓</td>
<td>The name of the group-file contains a typo in the configuration file.</td>
</tr>
<tr>
<td>chmod</td>
<td>×</td>
<td>Fail silently on dangling symbolic link.</td>
</tr>
<tr>
<td>cp</td>
<td>✓</td>
<td>Fail to copy the content of /proc/cpuinfo.</td>
</tr>
<tr>
<td>squid</td>
<td>×</td>
<td>When using Active Directory as authentication server, incorrectly denies user’s authentication due to truncation on security token.</td>
</tr>
</tbody>
</table>

Table 5.16: Real-world failures used in the user study.

Each participant is asked to fix the 5 failures as best as she/he could. They are provided a controlled Linux workstation and a full suite of debugging tools, including GDB. Each failure is given to a randomly chosen 50% of the programmers with Errlog inserted logs, and the other 50% without Errlog logs. All participants are given the explanation of the symptom, the source tree, and instructions on how to reproduce the three reproducible failures—this is actually biased against Errlog since it makes the no-Errlog cases easier (it took us hours to understand how to reproduce the two Apache failures). They are allowed to use GDB, enable the verbose level logs, searching the internet (except that they were not allowed to view the particular bugzilla diagnosis report of each case). The criteria of a successful diagnosis is for the users to fix the failure. Further, there is a 40 minutes time limit per failure; failing to fix the failure is recorded as using the full limit. 40 minutes is a best estimation of the maximum time needed.

Note that this is a best-effort user study. The potential biases should be considered when interpreting our results. Next I will discuss some of the potential biases and how they are addressed in this user study:

**Bias in case selection:** Some very hard-to-diagnose failures are not selected, and only two are
unreproducible ones, since diagnosis can easily take hours of time. This bias, however, is likely against Errlog since our result shows that Errlog is more effective on failures with a larger diagnosis time.

Bias in user selection: The participants might not represent the real programmers of these software. Only four users indicated familiarities with the code of these software. However, each participant is provided with a brief tutorial of the relevant code. Moreover, studies [YYZ+11] have shown that many programmers fixing real-world production failures are also not familiar with the code to be fixed because many companies rely on sustaining engineers to do the fix. Sustaining engineers are usually not the developers who wrote the code in the first place.

Bias in methodology: As this experiment is a single-blind trial (where the experimenters know the ground truth), there is a risk that subjects are influenced by interactions. Therefore the participants are given written instructions for each failure, with the only difference being the presence/absence of the log message; the interactions are also intentionally minimized during the trial.

![Figure 5.12: User study result, with error bars showing 95% confidence interval.](image)

**Results** Figure 5.12 shows the result of the user study. On average programmers took 60.7% less time diagnosing these failures when they were provided with the logs added by Errlog (10.37±2.18 minutes versus 25.72 ± 3.75 minutes, at 95% confidence interval). An unpaired T-test shows that the hypothesis “Errlog saves diagnosis time” is true with a probability of 99.9999999% (p=5.47 × 10\(^{-10}\)), indicating the data strongly supports this hypothesis.

Overall, since factors such as individuals’ capability are amortized among a number of participants, the only constant difference between the two control groups is the existence of the log messages provided by Errlog. Therefore I believe the results reflect Errlog’s effectiveness.

Less formally, all the participants reported that they found the additional error messages provided by Errlog significantly helped them diagnose the failures. In particular, many participants reported that “(Errlog added) logs are in particular helpful for debugging more complex systems or unfamiliar code where it required a great deal of time in isolating the buggy code path.”
However, for one failure, “apache crash”, the benefit of Errlog is not statistically significant. The crash is caused by a NULL pointer dereference. Errlog’s log message is printed simply because SIGSEGV is received. Since users could reproduce the crash and use GDB, they could relatively quickly diagnose it even without the log.

In comparison, Errlog achieves maximum diagnosis time reduction in two cases: “squid” (by 72.3%) and “apache no-file” (by 73.7%). The squid bug is a tricky one: due to the complexity in setting up the environment and user privacy concerns, it is not reproducible by the participants. Without logs, most of the control group took time-consuming goose chases through the complicated code. In contrast, the error message from Errlog, caused by the abnormal return of snprintf, guided most of the users from the other group to quickly spot the unsafe use of snprintf that truncated a long security token.

In the “apache no-file” case (the one shown in Figure 5.1), apache cannot open a file due to a typo in the configuration file. Without any error message, some programmers did not even realize this was caused by a misconfiguration and started to debug the code. In contrast, the error message provided by Errlog clearly indicates the open system call cannot find the file, allowing most programmers in this group to quickly locate and fix the typo in the configuration file.

Comparing between reproducible failures versus unreproducible failures, we found log messages added by Errlog is more effective in unreproducible failures. For reproducible failures, overall Errlog reduces diagnosis time by 54.4%, where for irreproducible failures Errlog reduce the diagnosis time by 66.9%.

Overall, the error messages provided by Errlog has a large, statistically significant effect on programmers diagnosis time. While there are many factors that can affect the accuracy of a user study, I believe that these results still provide at least some qualitative evidence about the usefulness of Errlog in helping programmers diagnose field failures, whether caused by a misconfiguration or a software bug.

5.7 Limitations and Discussions

All work has limitations, and Errlog is of no exception. In this section, I will discuss some of the limitations of Errlog. Limitations to the characteristics study are already discussed in Section 5.3.

1) What failures cannot benefit from Errlog? Not all the failures can be successfully diagnosed with Errlog. First, Errlog fails to insert log messages for 21% of the randomly sampled failures (Table 5.14). The error conditions of these failures are subtle, domain-specific, or are caused by underlying systems whose errors are not even properly propagated to the upper level applications [RGGL+09]. To address this, Errlog could be further used with low-overhead run-time
invariants checking [ECGN00, SCGA13] to log the violations to the invariants, or with various testing tools [BR02, CDE08] to log untested region.

Second, while logs provide clues to narrow down the search, they may not pinpoint the root cause. Indeed, the failure diagnosis process involves reconstructing the propagation chain from the symptom to the root cause. A log message along this chain will provide valuable fingerprints but does not guarantee that the root cause can be deterministically inferred. Section 5.5.4 shows that for 20% of the failures, the added log messages are not sufficient for the diagnosis. Such examples include (i) concurrency bugs where the thread-interleaving information is required and (ii) failures with long propagation path from fault to error where key execution states are already lost at the log point. Note that a majority (>98%) of failures in the real world are caused by semantic bugs, misconfigurations, and hardware errors but not concurrency bugs [SCA10].

However, this does not mean Errlog can only help diagnosing easy failures. Log messages collect more diagnostic information, not to pinpoint the exact root cause. Evidences provided by logs along the fault propagation chain, despite how complicated this chain is, will likely help narrowing down the search space. Therefore even for concurrency bugs, an error message is still likely to be useful to reduce the diagnosis search space.

(2) What is the trade-off of using adaptive sampling? Adaptive sampling might limit the usage of log messages. If the program has already exercised a log point, it is possible that this log will not be recorded for a subsequent error. Long running programs such as servers are especially vulnerable to this limitation. To alleviate this limitation, Errlog differentiates messages by runtime execution contexts including stack frames and errno. It can also periodically reset the sampling rate for long running programs.

In addition, adaptive sampling might preclude some useful forms of reasoning for a developer. For instance, the absence of a log message no longer guarantees that the program did not take the path containing the log point (assuming the log message has already appeared once). Moreover, even with the global order of each printed message, it would be harder to postmortem correlate them given the absence of some log occurrences.

To address this limitation, programmers can first use adaptive sampling on every log point during the testing and beta-release runs. Provided with the logs printed during normal executions, they can later switch to non-sampling logging for those not-exercised log points (which more likely capture true errors), while keep using sampling on those exercised ones for overhead concerns.

(3) Can Errlog completely replace developers in logging? The semantics of the auto-generated log messages are still not comparable to those written by developers. The message semantic is especially important for support engineers or end users who usually do not have access to the source code. Errlog can be integrated into the programming IDE, suggesting logging code as developers
program and allowing them to improve inserted log messages and assign proper verbosity levels, instead of using adaptive sampling.

(4) **How about verbose log messages?** This Chapter only studies log messages under the default verbosity mode, which is the typical production setting due to overhead concerns. Indeed, verbose logs can also help debugging production failures as developers might ask user to reproduce the failure with the verbose logging enabled. The study presented in Chapter 3 also suggests some error messages can be mistakenly assigned with a verbose mode thus can only show up with more verbosity enabled. However, such repeated failure reproduction itself is undesirable for the users in the first place. How to effectively insert verbose messages remains as our future work.

(5) **What is the impact of the imprecisions of the static analysis?** Such imprecisions, mainly caused by pointer aliasing in C/C++, might result in redundant and/or insufficient logging. However, given that Saturn’s intra-procedural analysis precisely tracks pointer aliases [ABD+07], such impact is limited only to inter-procedural analysis (where the error is propagated via return code to callers to log). In practice, however, we found programmers seldom use aliases on an error return code.

(6) **Will too many messages mislead users?** 35 out of 405 messages inserted by *Errlog* are noisy messages and might be printed during normal execution (Section 5.5.3). When presenting the printed log messages to the users, *Errlog* will rank these frequently appeared log messages lower than those with lower frequencies. However, such message might still mislead users. Indeed, in the user study, there is one case in which the user was distracted by such messages (however, even with these noisy messages diagnosis time was still reduced by 62%). Prior to software release, programmers can disable these noisy messages. For those log messages frequently occur during production runs, a configuration setting can be used to disable the logging of them.

(7) **How about programs written in other languages such as Java?** *Errlog* currently only works for software written in C language, but can be extended to others, e.g., Java. Although Java programs offer better exception mechanisms, it is still up to programmers whether to log the exceptions. The “catch” block in exception handling provides natural places to insert log messages.

### 5.8 Summary

This Chapter answers a critical question: where is the proper location to print a log message that will best help postmortem failure diagnosis, without undue logging overhead? It comprehensively investigated 250 randomly sampled failure reports, and found a number of exception patterns that, if logged, could help diagnosis. It further described *Errlog*, a tool that adds proactive logging code with only 1.4% logging overhead. A controlled user study shows that the logs added by *Errlog* can speed up the failure diagnosis by 60.7%.
Chapter 6

Previous Works

This Chapter describes the previous works that are related to this dissertation. Section 6.1 and Section 6.2 discuss existing failure diagnosis techniques. Section 6.3 describes other characteristic studies on software bugs or failures. Section 6.4 further discusses other related work.

6.1 Diagnosis without Reproducing the Failure

This section discusses failure diagnosis techniques that do not require reproducing the failures.

6.1.1 Log Analysis

Many studies propose techniques to analyze the log messages to infer diagnostic information. Majority of them focus on using statistical techniques to detect anomalous log messages, or detect recurring failures that match known issues [XHF+09, AMW+03, FWZW08, BBS+11, MPP09, MP08, NKN12, TPK+09, TPK+08, TKGN10]. Through studying commercial storage system logs, Jiang et al. [JHP+09] point out that logs can be of great value in failure diagnosis. They also propose using statistical techniques to identify key events recorded in the log that can help pinpoint the root causes. Xu et al. [XHF+09] apply machine learning techniques to learn common patterns from a large amount of console logs, and detect abnormal log patterns that violate the common patterns. To help parse logs more accurately, they analyze the Abstract Syntax Tree of the program to extract the format strings used to print out the logs. However, their error detection and diagnosis are based on patterns solely learnt from log messages, and thus cannot provide the capability of generating source-level control-flow and data-flow information. DISTALYZER [NKN12] uses machine learning techniques to compare the logs from failed executions and correct executions to help debug performance problems in distributed systems. Mariani and Pastore [MP08] analyze logs to learn correct dependency between log messages from normal execution, and used the information to identify anomalies in failed execution. Tan et al. use heuristics to parse the logs from Hadoop systems to derive control-flow and data-flow information, and further provide visualized representation of the diagnostic information [TKGN10, TPK+08, TPK+09]. Synop-
tic [BBS+11, BABE11] analyzes logs to extract the causal-relationship of the events. It requires developers to write regular expressions in order to parse the log.

There are also some commercial log analysis tools. These tools treat software as black-box and provide search and visualization capabilities on log messages. Splunk [Spl] and Log Insight [Log] index the text messages in the logs and provide search capability and visualize the analysis result. There also exist commercial open-source [Cha] tools for visualizing the data in logs based on standardized logging mechanisms, such as log4j [ApL].

The research proposed in this dissertation is different and complementary to these previous works. The improved quality of log messages can benefit these previous log analysis techniques with more informative logs. For SherLog, it is also different from these proposals as these studies did not leverage source code for extracting control-flow and data-flow information, and thus cannot provide source-level debugging information such as reconstructed execution paths (or partial execution paths) and run-time variable values as SherLog does.

6.1.2 Trace Analysis

Many diagnosis tools analyze performance counters [BKFP08, CZG+05, BWG+10, KTGN10] or execution traces [BLH08, BDIM04, SaB+10, AMW+03, FPK+07, CR, LAZJ03, CLM+09, ASM+05, HRD+07, MRMS10, MRHBS06, ZRA+08] to diagnose failures. Chopstix [BKFP08] collects low-level OS events such as scheduling, CPU utilization, L2 cache misses, etc. It then reconstructs these events off-line for analysis, and demonstrates its analysis is useful in diagnosing some elusive problems. Cohen et al. [CZG+05] use statistical techniques to extract signatures from low-level performance counters to diagnose recurring failures. Because of the semantic gap between low-level events to program’s logic, errors caused by semantic bugs cannot be diagnosed by these approaches since no anomalies can be observed in low-level events. Bodik et al. [BWG+10] propose to use machine learning to learn fingerprints from a large number of performance metrics, therefore for each performance anomaly, they can automatically classify it into some known bugs. Liblit et al. [LAZJ03] introduce “cooperative bug isolation”, which collects run-time traces from instrumented programs by sampling from many users to offload the monitoring overhead. With some classification techniques, they can pinpoint the predicates recorded in the traces that are most correlated with the bugs. Clarify [HRD+07] does instruction level profiling for normal and failure runs, and trains a classifier to classify each profile into to some known problems. HOLMES [CLM+09] further investigates how path profiling based program sampling can help bug isolation. It also designs an iterative and bug-directed profiling technique to effectively isolate bugs with low overheads. Magpie [BDIM04] addresses the problem of grouping traces generated by the same request in a distributed environment. Once the traces are grouped in per-request basis,
Magpie could compute the resource consumptions for each request, then uses clustering techniques to learn the common patterns for each request, and identifies anomalous requests.

Log message is a different and complementary source of run-time information compared with trace. Log message, typically programmed by the programmers, provides richer semantic information than trace. In addition, in typical production settings, logs are only printed when errors occur, therefore logging introduces minimal overhead during the normal executions. Tracing cannot avoid such overhead as the traces are collected in both normal and error execution. The diagnostic information provided by logs is also complementary to the debugging information provided by trace analysis techniques.

### 6.1.3 Bug-Type Specific Diagnostic Information Collection

Some other works [BM, HC04] propose to collect runtime information for a few, specific types of bugs. For example, Bell [BM] and SWAT [HC04] collect resource allocation information to debug resource leaks. Bond et al. [BNK+07] collect data-flow information for variables with uninitialized or NULL value. Zamfir and Candea [ZC10b] propose “bug fingerprints” and a tool called DCop to collect runtime locking and unlocking traces to help diagnose deadlocks. This dissertation’s work is complementary to these works. They can collect more information related to certain specific types of faults whereas log messages apply to various faults/bug types but may log only erroneous manifestation (instead of root causes). As shown in Chapter 5, majority of the randomly sampled failures manifest through a small set of generic exception patterns that can be logged.

### 6.1.4 Debugging with Core Dumps


Once these core dumps are collected, some techniques can infer diagnostic information from the core dumps. PSE [MSA+04] can perform off-line diagnosis of program crashes from core dump. Weeratunge, et al. [WZJ10] diagnose Heisen bugs by diff-ing the core dumps from failing runs and passing runs. Dimmunix [JTZC08] can also be used to diagnose deadlock by analyzing the back-trace from core dumps. ESD [ZC10a] uses static analysis to infer a feasible path from the core dump and error report.

As discussed early in Chapter 1, this dissertation’s work on log analysis and improvements is complementary to core dumps. Core dumps are not available in all types of failures, such as
incorrect computation. Log can collect historic, intermediate information prior to failures and also provide diagnostic information when no core dump is available. It also significantly reduces overhead and data size by recording only causally-related information. Furthermore, the Core Dump Digger in LogEnhancer derives equivalent information as delayed collection from a core dump if a core dump is available.

6.2 Diagnosis with Failure Reproduction

Ever since the beginning of software programming, debugging has been a necessary procedure. Early debugging efforts are mostly centered on using output devices, such as printers and the later CRT terminals to indicate when an error occurred, which is essentially the “printf-debugging” process. Programmer would then reproduce the failure and step through the code line by line until they could determine the location of the problem [Yas].

The compiler support for Symbol maps and the ability of setting breakpoints enabled the invention of interactive debugger [Yas]. Modern interactive debuggers [GDB, VSd, Ecl, KDB, Oca] allow programmers to set breakpoints, step through the execution (some even in backward manner), examining the stack trace and the variable values, etc. Interactive debuggers have been widely used in the developing and testing environment.

When a failure is reproducible, another commonly used debugging technique is backward slicing [Wei82, ADS91, KC05], where only the relevant executions are selected and presented to the programmers. Therefore programmers only need to focus on the relevant slice of the execution to infer the root cause. Based on the dynamic slices, some studies further propose to use statistical approaches to predict the fault location [LH06, LZH+07, XSZ08].

Many systems [VLW+11, GWT+08, KDC05, MCT08, OAA09, AS09, DKC+02, LWV+10, KSC00, GAM+07, LMC87, DLFC08, VMW, NPC05, CKS+08, VGK+10, XBH03, PZX+09, ZTG06, CBZ11, SN11, LVN10] target to deterministically replay failed execution, which generally requires high run-time logging overhead especially for multiprocessor systems. For example, ReVirt [DKC+02] records all the system level events at the Virtual Machine Monitor level, and can replay the execution of guest OS instruction-by-instruction. King and Chen [KC05] further design a system to analyzes the events from ReVirt to “backtrack” to the root cause of a security failure. To reduce the overhead, recently DoublePlay made clever use of spare cores on multiprocessor. Triage [TLH+07] performs diagnosis at the user’s site at the moment of the failure. Since Triage operates at the user’s failure site, it could replay the failure multiple times by reloading from the recent checkpoints to infer various diagnostic information. To support check-pointing, Triage requires OS kernel modification and support.
Some other systems propose to Deterministic Multithreading (DMT), that constrain a program to repeat the same thread interleaving, or schedules, when given the same input [CWG+11, CWcTY10, AWHF10, BHCG10, DLCO09, BAD+10, BYLN09, LCB11]. As a consequence, even with concurrent programs running on multi-core systems, a failed execution can always be deterministically replayed in the production environment (but not vendor’s site). Such systems can be highly beneficial in particular to those field engineers (engineers who are sent to the field to troubleshoot production failures) and user-site diagnostic systems such as Triage [TLH+07].

Unfortunately, as discussed in Chapter 1, production failures are often very hard to be replayed at the vendors’ site due to privacy concerns, unavailability of execution environments, etc. Consequently these effective diagnosis techniques often cannot be used on production failures. Therefore the contributions of this dissertation is complementary to these techniques and mainly targets to the cases when failure reproduction is difficult.

### 6.3 Failure Characteristic Studies

In the past, many studies characterized system failures or faults [Gra86, CYC+01, SC91, PTS+11]. Their findings have provided useful guidelines for improving software reliability from different aspects, including bug detection [CYC+01, SC91, PTS+11], bug avoidance [YYZ+11, YMZ+11], fault tolerance [GKIY03], testing [OWB05], failure recovery [CC00, SCA10], etc. Jiang et al. studied the statistical correlations among root cause, impact and diagnosis time of storage system failures [JHP+09]. They also confirmed that failures with log messages invariably took shorter resolution time than cases that do not have logs.

Unfortunately, few previous work have studied the logging aspects of failures. To our best knowledge, the studies in this dissertation are the first to focus on the weakness and needs of error logging practices.

### 6.4 Other Related Work

#### 6.4.1 Static Analysis for Bug Detection

Compiler techniques similar to SherLog and LogEnhancer are also used to address some other software reliability problems, such as bug detections [CDE08, KRS09, CCZ+07]. KLEE [CDE08] uses full symbolic execution engine to expose bugs in testing. Carburizer [KRS09] uses data-flow analysis to locate dependencies on data read from hardware.

Although SherLog and LogEnhancer also use symbolic execution, due to the very different
objectives, it starts from each log message and walks backward along the call chain to conduct “inference”, instead of walking forward to explore every execution path.

6.4.2 Detecting Bugs in Exception Handling Code

Many studies have shown that the exception handling code is frequently buggy [SC91, GRGAD+08, RGGL+09, MC11, GDJ+11, YTEM06, BDT06]. This is also observed in the Chapter 5 of this dissertation, and Errlog proposes to automatically insert log statements in a small set of generic exception patterns. Many systems aim to expose bugs in the exception handling code [GRGAD+08, RGGL+09, MC11, GDJ+11, YTEM06, BDT06], including two [GRGAD+08, RGGL+09] that statically detect the unchecked errors in file-system code. Errlog is different and complementary to these systems. Errlog has a different goal: easing the postmortem failure diagnosis, instead of detecting bugs. Therefore it starts from an empirical study of the weaknesses in logging practices, and then builds a tool to automatically add logging statements. In addition, some exception patterns such as fall-through in switch statements, signal handling, and domain-specific errors are not checked by prior systems. These additional exceptions detected by Errlog might benefit the prior systems for detecting more bugs in the corresponding error handling code.
Chapter 7
Conclusions and Future Work

Motivated by the important role log messages play in diagnosing production failures, this dissertation makes three main contributions on improving the use and design of log messages. These three contributions include 1) a new type of log inference; 2) understanding the logging efficacy, and 3) improving the quality of log messages.

For log inference, this dissertation observes the tedious manual efforts involved in the “printf-debugging” process — where programmers examine the code and log to understand why the log messages were printed. Motivated by this problem, this dissertation presents a tool, named SherLog, that reconstructs the partial execution paths and variable states by analyzing the source code and run-time logs. It uses symbolic execution to infer the feasible execution paths that would print the same sequence of log messages. The key insight behind its precision and scalability is that failure diagnosis, unlike other common program analysis tasks such as bug finding, is highly target-driven. Inferring the root cause of a failed execution often involves examining only a small slice of the program. Driven by this insight, SherLog only analyzes those functions that either directly print log messages or indirectly print logs through its callee, therefore it can afford to analyze them precisely. Evaluated on 8 real world failures from 7 large, widely used software projects, SherLog can infer useful and precise information for developers to diagnose the problems.

However, the effectiveness of any log inference, either manual or automated, is fundamentally limited by the quality of the log messages. Motivated by this observation, this dissertation further presents one of the first studies that characterizes the efficacy of current logging practice. To address the challenge of objectively judge the efficacy of log messages, this dissertation studies developers’ own modifications to their log messages as after-thoughts. It shows that developers often do not make the log messages right in their first attempts, and thus need to spend significant amount of effort to modify the problematic log messages as after-thoughts. It further provides several interesting findings on where developers spend most of their efforts in modifying the log messages, which can give insights for programmers, tool developers, and language and compiler designers to improve the current logging practice. The practical benefit of the findings is confirmed by a simple checker, which is motivated by identifying developers’ large amount of manual efforts in modifying the verbosity level, that can effectively detect new problematic logging code.
To further improve the quality of log messages, this dissertation further proposes two techniques that together will make the logging data more informative:

- **Enhancing existing log messages with LogEnhancer.** The common problem with existing log messages is that they often do not contain sufficient information. This dissertation presents a technique, named LogEnhancer, to enhance the quality of each logging statement. It uses static analysis to systematically and automatically modify each log message in a given piece of software to collect additional causally-related information. Such information can ease the diagnosis of future failures and thereby improve the diagnostic power of logging in general. In the evaluation, LogEnhancer can automatically infer over 95% of the existing variable values to each log message, indicating it can be at least as good as manual efforts. Furthermore, LogEnhancer inferred additional variable values in each log message, which are extremely useful in trouble-shooting failures.

- **Where to log?** While LogEnhancer enhances the quality of existing log messages, it assumes programmers already appropriately placed log message. But what if there are no relevant messages printed in the first place? Where is the best place to log? Motivated by such questions, this dissertation further studies the problem of absence of log messages for real-world failures. By studying 250 user reported failures, it shows that over half of the failures do not have any log messages printed during the failure. It further found that majority of these unreported failures were manifested via a common set of generic error patterns (e.g., system call return error), that if logged, can significantly ease the diagnosis of these unreported failure cases. Such findings further motivated the design of a tool, Errlog, that automatically adds appropriate logging statements into source code to improve postmortem diagnosis, adding with only 1.4% logging overhead. A controlled user study shows that the logs added by Errlog can speed up the failure diagnosis by 60.7%.

Together, these contributions made by this dissertation can expedite the diagnosis of software failures. Of course, it also opens many new problems for production failure diagnosis that should be addressed by future work.

Along the contribution of log inference, this dissertation only focuses on diagnosing failures within the same software. Modern systems are getting increasingly distributed, and when a failure occurs in distributed systems, engineers often cannot realize which component is to blame. Therefore they need to examine log files from thousands of components and multiple sources, identify the relevant events, and infer their correlations. How to provide log inference that can correlate messages across multiple components remains as an open problem to be solved by future work.

Along the research direction of log improvement, this dissertation only focuses on providing more evidences in the log files. While such additional evidences are inevitably helpful for diag-
nosis, but in the end, engineers need *answers*. Therefore, ideally, log messages should not only provide the evidences of a failure, but also the resolutions. In fact, through the study of logging practices in this dissertation I have observed some ad-hoc practices by the developers toward this goal, such as when detecting a configuration error, developers might print an error message that provide hint to the users of the possible resolutions. Many of such practices are in the form of after-thoughts. The open question is that whether we can systematically and automatically design the log messages to provide resolutions to the users once a failure occurs.

A broader and more open-ended question that worth thinking is: *how to design the software to make failures more diagnosable?* Currently, while there are a lot of investigations on how to design software that is reliable (i.e., with fewer bugs), there is only limited understanding on how to design software to make it *diagnosable*. The log improvement presented in this dissertation is one of the first attempts toward making software diagnosable. However, I believe there are a lot more opportunities. For example, it would be interesting to investigate from the diagnosis point of view, whether error handling is a good design or we should just let software terminate on every error. More broadly, what are the good designs of software that make failures more diagnosable?

Of course, there is always the problem of too much information — what if there are too many log messages. How to rank the log messages and remove the redundant/low entropy messages remain as an interesting problem. On the other hand, for some cases there is not enough information. Chapter 5 shows that even with *Errlog*, there are still 21% of the failures that are hard to provide meaningful logging data. How to further log those failures also remains as future work.
References


Man page for gcore (Linux section 1).


Google Inc., Breakpad.


[LZH⁺07] Chao Liu, Xiangyu Zhang, Jiawei Han, Yu Zhang, and Bharat K. Bhargava. Failure indexing: A dynamic slicing based approach. In *Proceedings of the 23rd IEEE International Conference on Software Maintenance*, ICSM’07, pages 382–393, 2007.


[Sch] Stephan Schmidt. 7 more good tips on logging. http://codemonkeyism.com/7-more-good-tips-on-logging/.


