Be Conservative: Enhancing Failure Diagnosis with Proactive Logging

Ding Yuan[†*], Soyeon Park[*], Peng Huang[*], Yang Liu[*], Michael M. Lee[*], Xiaoming Tang[*], Yuanyuan Zhou[*], Stefan Savage[*]

[*]*University of California, San Diego,* [†]*University of Illinois at Urbana-Champaign*

{diyuan,soyeon,ryanhuang,yal036,mmlee,x2tang,yyzhou,savage}@cs.ucsd.edu

Abstract

When systems fail in the field, logged error or warning messages are frequently the only evidence available for assessing and diagnosing the underlying cause. Consequently, the efficacy of such logging—how often and how well error causes can be determined via postmortem log messages—is a matter of significant practical importance. However, there is little empirical data about how well existing logging practices work and how they can yet be improved. We describe a comprehensive study characterizing the efficacy of logging practices across five large and widely used software systems. Across 250 randomly sampled reported failures, we first identify that more than half of the failures could not be diagnosed well using existing log data. Surprisingly, we find that majority of these unreported failures are manifested via a common set of generic error patterns (e.g., system call return errors) that, if logged, can significantly ease the diagnosis of these unreported failure cases. We further mechanize this knowledge in a tool called *Errlog*, that proactively adds appropriate logging statements into source code while adding only 1.4% performance overhead. A controlled user study suggests that *Errlog* can reduce diagnosis time by 60.7%.

1 Introduction

Real systems inevitably experience failure—whether due to hardware faults, misconfigurations or software bugs. However, resolving *why* such a failure has occurred can be extremely time-consuming, a problem that is further exacerbated for failures in the field. Indeed, failures in production systems are the *bête noire* of debugging; they simultaneously require immediate resolution and yet provide the least instrumented and most complex operational environment for doing so. Even worse, when a system fails at a *customer site*, product support engineers may not be given access to the failed system or its data—a situation referred to colloquially as "debugging in the dark".

This paper addresses a simple, yet critical, question: why is it so difficult to debug production software systems? We examine 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, we show that the majority (77%) of these failures manifest through a small number of concrete error patterns (e.g., error return codes, switch statement "fall-throughs", 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 our study).

Driven by this result, we further show that it is possible to fully automate the insertion of such proactive logging statements parsimoniously, yet capturing the key information needed for postmortem debugging. We describe the design and implementation of our 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 we studied. Finally, using a controlled *user study* with 20 programmers, we demonstrate that the error messages inserted by *Errlog* can cut failure diagnosis time by 60.7%.

2 Background

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 principal, deterministic replay—widely explored by the research community [3, 11, 29, 31]—allows a precise postmortem reproduction of the execution leading to a failure, in practice it faces a range of deployment hurdles including high overhead (such systems must log most non-deterministic events), privacy concerns (by definition, the replay trace should contain all input) and integration complexity (particularly in distributed environments with a range of vendors).

By contrast, the other major postmortem debugging advance, cooperative debugging, has broader commercial deployment, but is less useful for debugging individual failures. In this approach, exemplified by systems such as Windows Error Reporting [15] and the Mozilla Quality Feedback Agent [23], 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 triag-

[1] The data we used can be found at: http://opera.ucsd.edu/errlog.htm

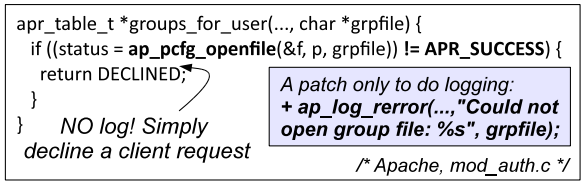

Figure 1: A real world example from `Apache` on the absence of error log message. After diagnosing this failure, the developer released a patch that only adds an error-logging statement.

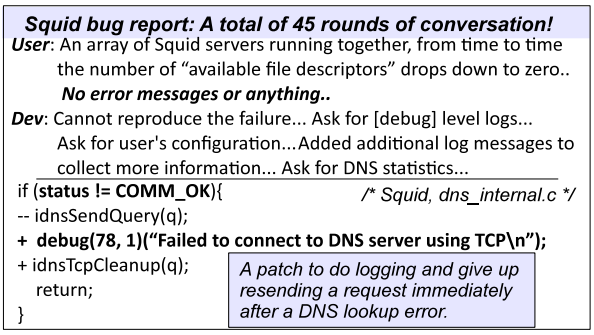

Figure 2: A real world example from `squid` to demonstrate the challenge of failure diagnosis in the absence of error messages, one that resulted in a long series of exchanges (45 rounds) between the user and developers.

ing 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 state, 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.

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. 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 their data typically focuses narrowly on issues of system health, logs 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 [10].

Even though log messages may not directly pinpoint the root cause (e.g. hardware errors, misconfigurations, software bugs) of a failure, they provide useful clues to narrow down the diagnosis search space. As this paper will show later, failures in the field *with* error messages have much shorter diagnosis time than those without.

Remembering to log

However, the utility of logging is ultimately predicated on what gets logged; how well have developers anticipated the failure modes that occur in practice? As we will show in this paper, there is significant room for improvement.

Figure 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 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 programmer neglected to include a statement to log the error event, resulting in a long and painful diagnosis.

One of the main objectives of this paper is to provide empirical evidence concerning the value of error logging. However, while we hope our results will indeed motivate developers to improve this aspect of their coding, we also recognize that automated tools can play an important role in reducing this burden.

Log automation vs log enhancement

Recently, Yuan et. al [37, 36] have studied how developers modify logging statements over time and proposed methods and tools to improve the quality of *existing* log messages by automatically collecting additional diagnostic information in each message. Unfortunately, while such approaches provide clear enhancements to the fidelity provided by a given log message, they 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, mak-

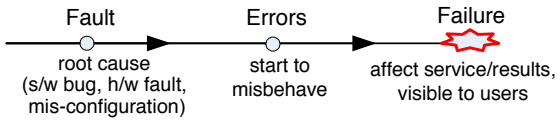

Figure 3: Classic Fault-Error-Failure model.

ing 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.

Fundamentally, any attempt to add new log messages needs to balance utility and overhead. To reach this goal, our 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, we focus instead on how to improve the quality and utility of the system logs that are already being used in practice. For similar reasons, we also choose 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.

3 Where to Log?

Before we decide where to add log points, it is useful to 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 [20]. As shown in Figure 3, a *fault* is a root cause, which can be a software bug, a hardware malfunction, or a misconfiguration. 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*, such as crash, hang, incorrect result, incomplete functionality, etc.

To further inform our choice of where to place log statements, we divide errors into two categories:

(i) 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.

(ii) 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. Therefore, some errors will always remain latent and undetected until they eventually produce a failure.

Appl.	LOC	#Default log points*	
		Total	Err+Warn
Apache	249K	1160	1102 (95%)
Squid	121K	1132	1052 (92%)
Postgres	825K	6234	6179 (99%)
SVN	288K	1836	1806 (98%)
Coreutils	69K	1086	1080 (99%)

Table 1: Applications used in our 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 verbosities.

Appl.	#Failures		
	population*	sampled	**with logs**
Apache	838	65	24 (37%)
Squid	680	50	20 (40%)
Postgres	195	45	24 (53%)
SVN	321	45	25 (56%)
Coreutils	212	45	15 (33%)
Total	**2246**	**250**	**108 (43%)**

Table 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.

To dive in one step further, detected errors can be handled in three different ways: *(i) Early termination*: a program can simply exit when encountering an error. *(ii) Correct error handling*: a program recovers from an error appropriately, and continues execution. *(iii) Incorrect error handling*: a program does not handle the error correctly and results in an unexpected failure.

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 [30, 17, 16]. 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.

4 Learning from Real World Failures

This section describes our empirical study of how effective existing logging practices are in diagnosis. To drive our study, we randomly sampled 250 real world failures reported in five popular systems, including four servers (`Apache httpd`, `squid`, `PostgreSQL`, and `SVN`) and a utility toolset (`GNU Coreutils`), as shown in Table 1.

The failure sample sets for each system are shown in Table 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. We randomly sampled those non-duplicate, fixed failures that were reported within the recent five years. We carefully studied the reports, discussions, related source code and patches to understand the root cause and its propagation leading to each failure.

In our study, we focus primarily on the *presence* of a failure-related log message, and do not look more deeply into the content of the messages themselves. 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. Moreover, Yuan et. al.'s recent LogEnhancer work shows promise in automatically enhancing each existing log message by recording the values of causally-related variables [37].

Threats to Validity: As with all characterization studies, there is an inherent risk that our findings may be specific to the programs studied and may not apply to other software. While we cannot establish representativeness categorically, we took care to select 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 we are 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 [28]. However, certain failures might not be reported to Bugzilla. Both `Apache` and `Postgres` have separate mailing lists for security issues; Configuration errors (including performance tunings) are usually reported to the user-discussion forums. Therefore our 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.

Another concern is that we might miss those very hard failures that never got fixed. However, as the studied applications are well maintained, *severity* is the determining factor of the likelyhood 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 we miss 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. Our failure study took 4 inspectors 4 months of time.

4.1 Failure Characterization

Across each program we extract its embedded log messages and then analyze how these messages relate to the failures we identified manually. We decompose these results through a series of findings for particular aspects of logging behavior.

• **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 1. Verbose or bookkeeping messages are usually not enabled under the default verbosity mode due to overhead concerns. 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)*, as shown in Figure 4, supporting our 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). We then divide the failure set 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 our sample set is relatively large we believe our 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).

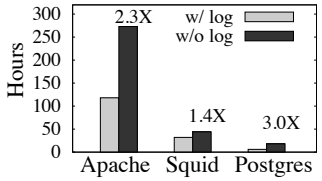

Figure 4: Benefit of logging on diagnosis time (median).

• **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 2. Next, we further zoom in to understand why those cases did not have log messages and whether it is hard to log them in advance.

• **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, which

[2]Throughout the entire paper, we assume the default verbosity mode (i.e., no verbosity), which is the typical setting for production runs.

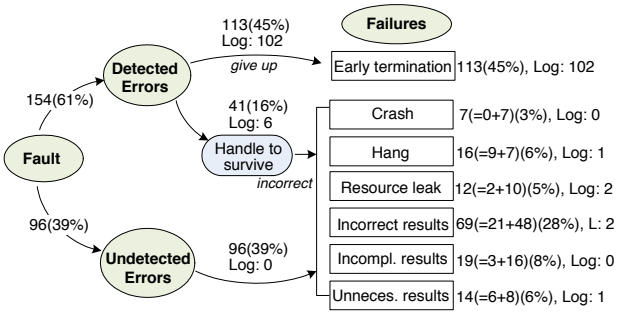

Figure 5: Fault manifestation for our sampled failures. (=x+y): x failures from detected errors and y failures from undetected errors. "Log: N": N cases have failure-related log messages.

Appl.	Detected Error		Undetected Error	
	Early terminat.	Handle incorrect.	Generic except.	Semantic except.
Apache	23	18	9	15
Squid	23	9	10	8
Postgres	24	4	5	12
SVN	26	0	7	12
Coreutils	17	10	8	10
Total	**113(73%)**	**41(27%)**	**39(41%)**	**57(59%)**
	154		**96**	

Table 3: Error manifestation characteristics of examined software. All detected errors were caught by generic exception checks such as those in Table 5. Some undetected errors could have been detected in the same way.

Appl.	Early Termination		Handle Incorrectly	
	no log	w/ log	no log	w/ log
Apache	3	20	14	4
Squid	4	19	8	1
Postgres	0	24	4	0
SVN	1	25	0	0
Coreutils	3	14	9	1
Total Detected	**11(10%)**	**102(90%)**	**35(85%)**	**6(15%)**
	113		**41**	

Table 4: Logging practices when general errors are detected.

shows how our sampled failures map to the error manifestation model presented in Section 3. Table 3 breaks them down by application, where the behavior is generally consistent. This indicates that programmers did reasonably well in anticipating many possible errors in advance.

However, as shown in Figure 5 programmers do not comprehensively *log* these detected errors. 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.

Further drilling down, we 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 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 1 documents one such omission in `Apache`). Logging overhead is not a big concern since the programs subsequently terminate.

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

Table 5: Logging practices for common exceptions.

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 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. We surmise this is because 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.

• **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. The `squid` example shown in Figure 2 documents such an example.

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 3 in which the programs could have detected the error via generic checks (e.g., system call error returns). Therefore we have:

• **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.*

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.

4.2 Logging Generic Exceptions

Table 5 documents these generic exception patterns, many of which are checked by the studied programs but are not logged. We explain some of them and highlight good practices that we encountered.

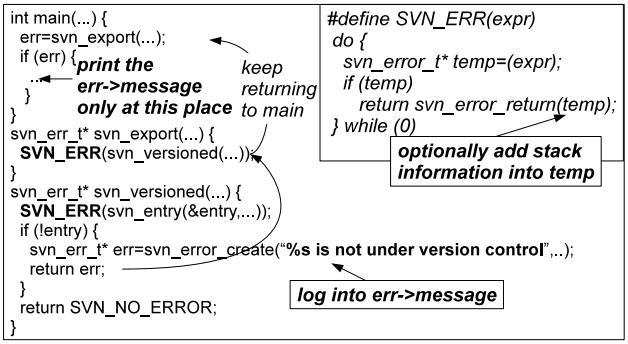

Figure 6: SVN's good logging practices for checking and logging function return errors.

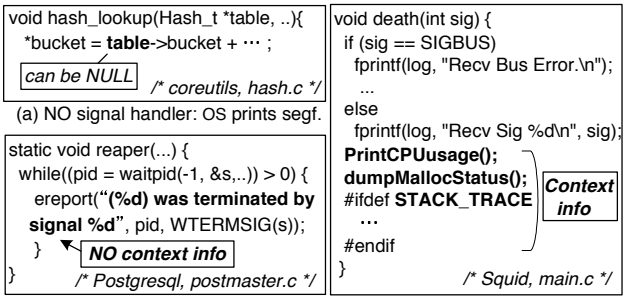

Figure 7: Logging practices for exception signals.

(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. However, a significant percentage (28%) of them did not log such errors.

Good practice: SVN uniformly logs function return errors. First, as shown in Figure 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 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 our study, about 14% of detected errors were caught by the programs' own signal handlers, and fortunately all were logged.

However, all examined software (except for squid) only logs signal names. Figure 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.

Statement cov.*	10 (18%)	Decision cov.	12 (21%)
Condition cov.	2 (4%)	Weak mutation	4 (7%)
Mult. cond. cov.	2 (4%)	Loop cov.	1 (2%)
Concurr. cov.	1 (2%)	Perf. profiling	1 (2%)
Functional cov.	34 (60%)	**Total failures**	57

Table 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.

(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.

(4) Other exceptions: Programs also perform other types of generic exception checks such as bound-checks, input vadility checks, resource leak checks, etc., (Table 5) but they often forget to log detected errors, losing opportunities to gather evidences for postmortem diagnosis.

4.3 Logging for Hard-to-check Failures

As shown earlier in Table 3, 57 failures are hard to detect via generic exception checks. We refer them as *hard-to-check errors*. When a production failure occurs, it is usually due to an unusual input or environment triggering some code paths that are not covered during in-house testing. Table 6 shows that 21% of the 57 hard-to-check failure cases execute some branch edges that we surmise have never been executed during testing (otherwise, the bugs on those edges would definitely have been exposed)[3]. 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 them.*

5 *Errlog*: A Practical Logging Tool

Driven by the findings in our study, we further build an automatic logging tool called *Errlog*, 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 [2] static analysis framework.

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 do we

[3]Due to software's complexity, cost of testing, and time-to-market pressure, complex systems can rarely achieve 100% test coverage.

	Exception Pattern	How to identify in source code
DE	Function return error	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.
	Failed memory safety check	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.
	Abnormal exit/abort	Search for "abort, exit, _exit". The constraint *EC* for this pattern is "true".
	Exception signals	Intercept and log abnormal signals. Our logging code uses memory buffer and is re-entrant.
LE	Unexpected cases falling into default	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.
	Invalid input check	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.
AG	Resource leak	*Errlog* monitors resource (memory and file descriptor) usage and logs them with context information. *Errlog* uses exponential-based sampling to reduce the overhead (Section 5.3).

Table 7: Generic exception patterns searched by *Errlog*. These patterns are directly from our findings in Table 5 in Section 4.

avoid significant performance overhead without missing important diagnostic information?

To address the first challenge, *Errlog* follows the observations from our characterization study. It identifies potential exceptions by mechanically searching in the source code for the seven generic exception patterns in Table 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.

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. 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.

5.1 Exception Identification

In this step, *Errlog* scans the code and generates the following predicate: *exception(program_point P, constraint EC)*, where *P* is the program location of an exception check, and *EC* is the constraint that causes the exception to happen. In the example shown in Figure 2, *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 we should log the exception and whether the developer has already logged the exception.

Search for generic exceptions Table 7 shows the generic exception patterns *Errlog* automatically identifies, which are directly from the findings in our characterization study.

5.1.1 Learning Program-Specific Exceptions

Errlog-LE further attempts to automatically identify program-specific exceptions *without any program-specific knowledge*. If a certain 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 2 is a `squid`-specific exception that is frequently followed by an error message. Similar to previous work [12] 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. For example, the exception condition in Figure 8 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-

```
tmp=pcre_malloc(...);
if (tmp == NULL)
  goto out_of_memory;
  ... ...
out_of_memory:
  error ("out of memory");
```

Figure 8: Example showing the need of control & data flow analysis.

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 8, *Errlog* first identifies the condition that leads to the error message being `tmp==NULL`. By analyzing the data-flow of `tmp`, *Errlog* 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 condition `status!=COMM_OK` in Figure 2 is learnt because `status` is a formal parameter of the function.

Identifying helper logging functions *Errlog* only requires developers to provide the name of the default logging function. However, in all the large software we studied, 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 `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. Our experiments are conducted without this option.

5.1.2 Identifying Untested Code Area (optional)

Errlog-AG further inserts log points for code regions not covered by in-house testing. We use the test coverage tool `GNU gcov` [14] and the branch decision coverage criteria. For each untested branch decision, *Errlog* instruments a log point. For multiple nested branches, *Errlog* only inserts a log point at the top level. This option is not enabled in our experiments unless otherwise specified.

5.2 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 8, 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. $C_P \wedge EC \wedge \neg C_L$ is *not* satisfiable.
2. $C_P \wedge \neg C_L$ is satisfiable.

The first condition is equivalent to i), while the second condition is equivalent to ii).

If no such log point exists, *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.

Log placement If no logging statement is found for an exception E from the analysis above, *Errlog* inserts its own logging library function, "`Elog(logID)`", into the basic block after the exception check. If no such check exists, *Errlog* also adds the check.

Each logging statement records (i) a log ID unique to each log point, (ii) the call stack, (iii) casually-related variable values identified using LogEnhancer [37] [4], (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 `errno` is also recorded. No static text string is printed at runtime. *Errlog* will compose a postmortem text message by mapping the log ID and `errno` to a text string describing the exception. For example, *Errlog* would print the following message for an `open` system-call error: *"open system call error: No such file or directory: ./filepath ..."*.

5.3 Run-time Logging Library

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

To address this issue, *Errlog*'s run-time logging library borrows the idea of adaptive sampling [19]. 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 2^nth dynamic occurrences. 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^nth dynamic occurrences. We further differentiate system call return errors by the value of `errno`. For efficiency, *Errlog* logs into

[4] LogEnhancer [37] is a static analysis tool to identify useful variable values that should be logged with each *existing* log message.

App.	*Errlog*-DE					*Errlog*-LE				*Errlog*-AG	
	func. ret.	mem. safe.	abno. exit	sig-nals	**Total**	switch-default	input check	learned errors	**Total**	res. leak	**Total**
Apache	30	41	9	22	102 (0.09X)	117	389	360	968 (0.83X)	24	992 (0.86X)
Squid	393	112	29	3	537 (0.47X)	116	147	17	817 (0.72X)	26	843 (0.74X)
Postgres	619	166	28	9	822 (0.13X)	432	7	1442	2703 (0.43X)	65	2768 (0.44X)
SVN	33	6	1	3	43 (0.02X)	53	1	8	105 (0.06X)	31	136 (0.07X)
Coreutil cp	34	4	9	2	49 (0.73X)	13	5	0	67 (1.00X)	4	71 (1.06X)
CVS	1109	360	23	3	1495 (1.30X)	52	49	645	2241 (1.95X)	32	2273 (1.97X)
OpenSSH	714	31	26	3	774 (0.32X)	112	31	63	980 (0.40X)	23	1003 (0.41X)
lighttpd	171	16	30	3	220 (0.27X)	67	27	6	320 (0.39X)	37	357 (0.44X)
gzip	45	3	32	3	83 (0.85X)	40	3	16	142 (1.45X)	14	156 (1.59X)
make	339	6	16	3	364 (2.72X)	29	12	10	415 (3.10X)	6	421 (3.14X)
Total	3487	745	203	54	4489 (0.30X)	1031	671	2567	8758 (0.58X)	262	9020 (0.60X)

Table 8: Additional log points added by *Errlog*. The "total" of LE and AG include DE and DE+LE, respectively, and are compared to the number of existing log points (Table 1 and 9). Note that most of these log points are *not* executed during normal execution.

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.

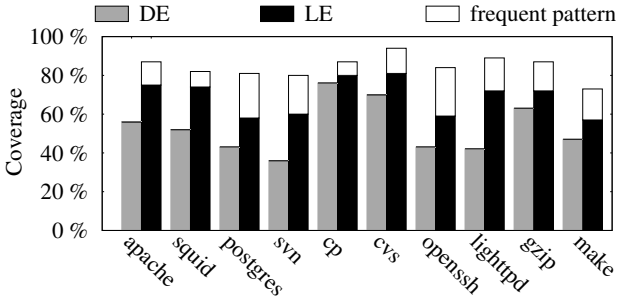

Figure 9: Coverage of existing log points by *Errlog*. For *Errlog*-LE, we break down the coverages into log points identified by generic exceptions and those learned by frequent logging patterns. AG has similar coverages as LE.

6 In-lab Experiment

We evaluate *Errlog* using both in-lab experiments and a controlled user study. This section presents the in-lab experiments. In addition to the applications we used in our characterization study, we also evaluate *Errlog* with 5 more applications as shown in Table 9.

6.1 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 our Section 4 suggests that the current logging practices miss many logging opportunities, we do 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 afterthoughts. This is confirmed by our Finding 2: 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 9 shows that *Errlog*, especially with *Errlog*-LE, can automatically cover an average of 84% of existing log points across all evaluated software. 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.

App.	description	LOC	#Default Log Points	
			Total	Err+Warn
CVS	version cont. sys.	111K	1151	1139 (99%)
OpenSSH	secure connection	81K	2421	2384 (98%)
lighttpd	web server	54K	813	792 (97%)
gzip	comp/decom. files	22K	98	95 (97%)
make	builds programs	29K	134	129 (96%)

Table 9: The *new* software projects used to evaluate *Errlog*, in addition to the five examined in our characterization study.

6.2 Additional Log Points

In addition to the existing log points, *Errlog* also adds new log points, shown in Table 8. Even though *Errlog*-LE adds 0.06X–3.10X additional log points, they only cause an average of 1.4% overhead (Section 6.3) because most of them are not triggered when the execution is normal.

Logging for untested branch decision Table 10 shows *Errlog*-AG's optional logging for untested branch decisions, which is not included in the results above. For `Apache`, `Postgres`, `SVN` and `Coreutils`, we used the test cases released together with the software.

App.	Uncovered decisions	# log points
Apache	57.0% (2915)	655
Postgres	51.7% (51396)	11810
SVN	53.7% (14858)	4051
Coreutils	62.3% (9238)	2293

Table 10: Optional logging for untested branch decisions.

Software	Adaptive sampling*			No sampling	
	DE	LE	AG	DE	LE
Apache	<1%	<1%	2.7%	<1%	<1%
Squid	<1%	1.8%	2.1%	4.3%	9.6%
Postgres	1.5%	1.9%	2.0%	12.6%	40.1%
SVN	<1%	<1%	<1%	<1%	<1%
cp	<1%	<1%	<1%	6.3%	6.3%
CVS	<1%	<1%	<1%	<1%	2.3%
Openssh scp	2.0%	4.6%	4.8%	5.2%	27.1%
lighttpd	<1%	<1%	2.2%	<1%	<1%
gzip	<1%	<1%	<1%	<1%	<1%
make	3.9%	4.0%	4.8%	4.2%	6.8%
Average	1.1%	1.4%	2.1%	3.5%	9.4%

Table 11: The performance overhead added by *Errlog*'s logging. *: By default, *Errlog* uses adaptive sampling. We also show the overhead without using sampling only to demonstrate the effect of adaptive sampling.

	func. ret.	mem. safe.	switch default	input check	learned errors	Total
Log pts.	5	8	5	7	10	35

Table 12: Noisy log points exercised during correct executions.

6.3 Performance Overhead

We evaluate *Errlog*'s logging overhead during the software's normal execution. Server performance is measured in peak-throughput. Web servers including `Apache httpd`, `squid`, and `lighttpd` are measured with `ab` [4]; `Postgres` is evaluated with `pgbench` [24] using the select-only workload; `SVN` and `CVS` with a combination of check-out, merge, copy, etc.; `OpenSSH`'s `scp` with repeatedly transferring files; `gzip` and `cp` with processing large files; `make` with compiling `PostgreSQL`.

Table 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%. The maximum runtime memory footprint imposed by *Errlog* is less than 1MB.

`scp` and `make` have larger overhead than others in Table 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.

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. Therefore we simply treat the log points that are executed during our performance testing as noisy messages as we are not aware of any failures in our performance testing. Among the five applications we used in our failure study, only a total of 35 log points (out of 405 error condition checks) are executed, between 3-12 for each application. Table 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 we use adaptive sampling, the size of run-time log is small (less than 1MB).

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

Table 13: *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.

Sampling overhead comparison We also evaluate the efficiency of adaptive sampling by comparing it with "no sampling" in Table 11. "No sampling" logging records *every* occurrence of executed log points into memory buffer and flushes it to disk when it becomes full. We do not evaluate "no sampling" on *Errlog*-AG 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. 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% overhead to 1.9%. In comparison, the effect of sampling is less obvious for `make`, 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. *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.

6.4 Real World Failures

Table 13 shows *Errlog*'s effect to the real-world failures we studied in Section 4. In this experiment we turn on the logging for untested code region in *Errlog*-AG. Originally, 41% of the failures had log messages. With *Errlog*, 75% and 79% of the failures (with *Errlog*-LE and AG, respec-

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

Table 14: Real-world failures used in our user study.

tively) have failure-related log messages.

Effectiveness of *Errlog* for Diagnosis We evaluate the usefulness of the added log messages in diagnosis using SherLog [35], a log-inference engine. Given log messages related to a failure, SherLog reconstructs the execution paths must/may have taken to lead to the failure. Our evaluation shows that 80% of the new messages can help SherLog to successfully infer the root causes.

7 User Study

We conduct a controlled user study to measure the effectiveness of *Errlog*. Table 14 shows the five real-world production failures we used. Except for "`apache` crash", the other four failed silently. Failures are selected to cover diverse root causes (bugs and misconfigurations), symptoms, and reproducibilities. We test on 20 programmers (no co-author of this paper is included), who indicated that they have extensive and recent experience in C/C++.

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). 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. Below we discuss some of the potential biases and how we addressed them in our user study:

Bias in case selection: We did not select some very hard-to-diagnose failures and only chose two 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.

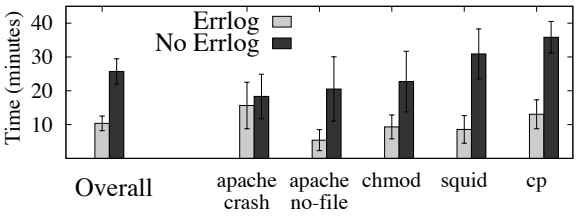

Figure 10: User study result, with error bars showing 95% confidence interval.

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, we do provide each user a brief tutorial of the relevant code. Moreover, studies [34] 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 our experiment is a single-blind trial (where we, the experimenters, know the ground truth), there is a risk that the subjects are influenced by the interaction. Therefore we give the users written instructions for each failure, with the only difference being the presence/absence of the log message; we also minimize our interactions with the user during the trial.

Results Figure 10 shows our study result. 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% ($p=5.47 \times 10^{-10}$), 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 we 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 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.

8 Limitations and Discussions

(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 13). 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 [26]. *Errlog* could be further used with low-overhead run-time invariants checking [13] to log the violations to the invariants.

Second, while log messages provide clues to narrow down the search, they may not pinpoint the root cause. Section 6.4 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 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 by concurrency bugs [27].

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, we differentiate messages by runtime execution contexts including stack frames and errno. We 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.

(4) How about verbose log messages? This paper only studies log messages under the default verbosity mode, which is the typical production setting due to overhead concerns [36]. Indeed, verbose logs can also help debugging production failures as developers might ask user to reproduce the failure with the verbose logging 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 logging and insufficient logging. However, given that Saturn's intra-procedural analysis precisely tracks pointer aliases [2], such impact is limited only to the inter-procedural analysis (where the error is propagated via return code to the callers to log). In practice, however, we found programmers seldom use aliases on an error return code.

9 Related Work

Log enhancement and analysis Some recent proposals characterize, improve, and analyze *existing* log messages for failure diagnosis [36, 37, 35, 32]. LogEnhancer [37] adds variables into each existing log message to collect more diagnostic information; Our previous work [36] studied developers' modifications to existing logging code and found that they often cannot get the logging right at the first attempt. SherLog [35] is a postmortem debugger that combines runtime logs and source code to reconstruct

the partial execution path occurred in the failed execution.

However, all of these studies only deal with *existing* log messages, and do not address the challenge of where to add new logs as discussed in Section 2.

The different objective makes our techniques very different from the systems listed above. For example, both SherLog and LogEnhancer start from an existing log message to backtrack the execution paths. In comparison, *Errlog* scans the entire source code to identify different exception patterns. *Errlog* also learns the program-specific errors, identifies the untested code areas, checks whether exceptions are already logged, and logs with adaptive sampling at runtime. All these techniques are unique to *Errlog* for its objective.

Detecting bugs in exception handling code Many systems aim to expose bugs in the exception handling code [17, 26, 22, 16, 33], including two [17, 26] 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 we need to empirically study the weakness in logging practices, and build a tool to automatically add *logging statements*. Only our exception identification part (Section 5.1) shares some similarities with [17, 26]. 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.

Bug-type specific diagnostic information collection Some studies [5, 19, 6, 38] proposed to collect runtime information for specific types of bugs. For example, DCop [38] collects runtime lock/unlock trace for diagnosing deadlocks. These systems are more powerful but are limited to debugging only specific types of fault, whereas *Errlog* applies to various fault types but may log only the erroneous manifestations (instead of root causes).

Logging for deterministic replay Other systems [31, 3, 11, 29] aim at deterministically replaying the failure execution, which generally requires recording all the inputs or impose high run-time logging overhead. Castro et al. propose replacing private information while preserving diagnosis power [8], but they require replaying the execution at the user's site. *Errlog* is complementary and targets the failures where reproduction is difficult due to privacy concerns, unavailability of execution environments, etc.

Tracing for failure diagnosis Execution trace monitoring [21, 18, 25, 1] has been used to collect diagnostic information, where both normal and abnormal executions are monitored. *Errlog* logs only exceptions so its runtime overhead is small during normal executions.

Static analysis *Errlog* uses the Saturn [2] symbolic execution framework. Similar static analysis [7, 9] was used for other purposes, such as bug detection. *Errlog* uses code analysis for a different objective: log insertion for future postmortem diagnosis. Therefore many design aspects are unique to *Errlog*, such as checking whether the exception is logged, learning domain-specific errors, etc.

10 Conclusions

This paper 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? We comprehensively investigated 250 randomly sampled failure reports, and found a number of exception patterns that, if logged, could help diagnosis. We further developed *Errlog*, a tool that adds proactive logging code with only 1.4% logging overhead. Our controlled user study shows that the logs added by *Errlog* can speed up the failure diagnosis by 60.7%.

Acknowledgements

We greatly appreciate anonymous reviewers and our shepherd, Florentina I. Popovici, for their insightful feedback. We are also grateful to the Opera group, Alex Rasmussen, and the UCSD System and Networking group for the useful discussions and paper proof-reading. We also thank all of the volunteers in the user study for their time spent in debugging. This research is supported by NSF CNS-0720743 grant, NSF CSR Small 1017784 grant and NetApp Gift grant.

References

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proc. of the 19th ACM Symposium on Operating Systems Principles*, pages 74–89, 2003.

[2] A. Aiken, S. Bugrara, I. Dillig, T. Dillig, B. Hackett, and P. Hawkins. An overview of the saturn project. In *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering*, pages 43–48, 2007.

[3] G. Altekar and I. Stoica. ODR: output-deterministic replay for multicore debugging. In *Proc. of the ACM 22nd Symp. on Operating Systems Principles*, pages 193–206, 2009.

[4] ab - Apache HTTP server benchmarking tool. `http://goo.gl/NLAqZ`.

[5] M. D. Bond and K. S. McKinley. Bell: bit-encoding online memory leak detection. In *Proc. of the 12th International Conference on on Architectural Support for Programming Languages and Operating Systems*, pages 61–72, 2006.

[6] M. D. Bond, N. Nethercote, S. W. Kent, S. Z. Guyer, and K. S. McKinley. Tracking bad apples: reporting the origin of null and undefined value errors. In *Proc. of the ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications*, pages 405–422, 2007.

[7] C. Cadar, D. Dunbar, and D. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX*

Conference on Operating Systems Design and Implementation, pages 209–224, 2008.

[8] M. Castro, M. Costa, and J.-P. Martin. Better bug reporting with better privacy. *SIGARCH Comput. Archit. News*, 36(1):319–328, Mar. 2008.

[9] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: a platform for in-vivo multi-path analysis of software systems. In *Proceedings of the 16th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 265–278, 2011.

[10] Dell. Streamlined Troubleshooting with the Dell system E-Support tool. *Dell Power Solutions*, 2008.

[11] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. In *Proc. of the Symp. on Operating Systems Design & Implementation*, 2002.

[12] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *Proc. of the 18th ACM Symposium on Operating Systems Principles*, pages 57–72, 2001.

[13] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 449–458, 2000.

[14] gcov – a test coverage program. `http://goo.gl/R9PoN`.

[15] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: ten years of implementation and experience. In *Proc. of the ACM 22nd Symposium on Operating Systems Principles*, pages 103–116, 2009.

[16] H. S. Gunawi, T. Do, P. Joshi, P. Alvaro, J. M. Hellerstein, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, K. Sen, and D. Borthakur. FATE and DESTINI: a framework for cloud recovery testing. In *Proc. of the Symp. on Networked Systems Design & Implementation*, pages 239–252, 2011.

[17] H. S. Gunawi, C. Rubio-González, A. C. Arpaci-Dusseau, R. H. Arpaci-Dussea, and B. Liblit. EIO: error handling is occasionally correct. In *Proc. of the USENIX Conference on File and Storage Technologies*, pages 207–222, 2008.

[18] Z. Guo, D. Zhou, H. Lin, M. Yang, F. Long, C. Deng, C. Liu, and L. Zhou. G^2: a graph processing system for diagnosing distributed systems. In *Proc. of the USENIX Annual Technical Conference*, pages 299–312, 2011.

[19] M. Hauswirth and T. M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 156–164, 2004.

[20] J.-C. Laprie. Dependable computing: concepts, limits, challenges. In *Proceedings of the 25th International Conference on Fault-tolerant Computing*, pages 42–54, 1995.

[21] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Proc. of the ACM SIGPLAN 2003 conference on Programming Language Design and Implementation*, pages 141–154, 2003.

[22] P. D. Marinescu and G. Candea. Efficient testing of recovery code using fault injection. *ACM Trans. Comput. Syst.*, 29(4):11:1–11:38, Dec. 2011.

[23] Mozilla quality feedback agent. `http://goo.gl/V9zl2`.

[24] pgbench - PostgreSQL wiki. `goo.gl/GKhms`.

[25] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: detecting the unexpected in distributed systems. In *Proc. of the 3rd Symp. on Networked Systems Design & Implementation*, pages 115–128, 2006.

[26] C. Rubio-González, H. S. Gunawi, B. Liblit, R. H. Arpaci-Dusseau, and A. C. Arpaci-Dusseau. Error propagation analysis for file systems. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 270–280, 2009.

[27] S. K. Sahoo, J. Criswell, and V. S. Adve. An empirical study of reported bugs in server software with implications for automated bug diagnosis. In *Proc. of the 32nd Intl. Conf. on Software Engineering*, pages 485–494, 2010.

[28] C. Spatz. Basic statistics, 1981.

[29] D. Subhraveti and J. Nieh. Record and transplay: partial checkpointing for replay debugging across heterogeneous systems. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 109–120, 2011.

[30] M. Sullivan and R. Chillarege. Software defects and their impact on system availability-a study of field failures in operating systems. In *Fault-Tolerant Computing, 1991. FTCS-21. Digest of Papers., 21st International Symposium*, pages 2–9, 1991.

[31] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. Doubleplay: parallelizing sequential logging and replay. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 15–26, 2011.

[32] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *Proc. of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 117–132, 2009.

[33] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. *ACM Trans. Comput. Syst.*, 24(4):393–423, Nov. 2006.

[34] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pages 26–36, 2011.

[35] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. SherLog: error diagnosis by connecting clues from run-time logs. In *Proceedings of the 15th edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, pages 143–154, 2010.

[36] D. Yuan, S. Park, and Y. Zhou. Characterizing logging practices in open-source software. In *Proc. of the Intl. Conf. on Software Engineering*, pages 102–112, 2012.

[37] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage. Improving software diagnosability via log enhancement. *ACM Trans. Comput. Syst.*, 30(1):4:1–4:28, Feb. 2012.

[38] C. Zamfir and G. Candea. Low-overhead bug fingerprinting for fast debugging. In *Proceedings of the 1st International Conference on Runtime Verification*, pages 460–468, 2010.