Improving Software Diagnosability via Log Enhancement

DING YUAN, University of Illinois at Urbana-Champaign and University of California, San Diego
JING ZHENG, SOYEON PARK, YUANYUAN ZHOU, and STEFAN SAVAGE,
University of California, San Diego

Diagnosing software failures in the field is notoriously difficult, in part due to the fundamental complexity of troubleshooting *any* complex software system, but further exacerbated by the paucity of information that is typically available in the production setting. Indeed, for reasons of both overhead and privacy, it is common that only the run-time log generated by a system (e.g., syslog) can be shared with the developers. Unfortunately, the ad-hoc nature of such reports are frequently insufficient for detailed failure diagnosis. This paper seeks to improve this situation within the rubric of existing practice. We describe a tool, *LogEnhancer* that automatically "enhances" existing logging code to aid in future post-failure debugging. We evaluate *LogEnhancer* on eight large, real-world applications and demonstrate that it can dramatically reduce the set of potential root failure causes that must be considered while imposing negligible overheads.

Categories and Subject Descriptors: D.2.5 [**Testing and Debugging**]: Diagnostics; D.4.5 [**Operating Systems**]: Reliability

General Terms: Reliability, Languages

Additional Key Words and Phrases: Log, failure diagnostics, debugging, software diagnosability, program analysis

ACM Reference Format:
Yuan, D., Zheng, J., Park, S., Zhou, Y., and Savage, S. 2012. Improving software diagnosability via log enhancement. ACM Trans. Comput. Syst. 30, 1, Article 4 (February 2012), 28 pages.
DOI = 10.1145/2110356.2110360 http://doi.acm.org/10.1145/2110356.2110360

1. INTRODUCTION

Complex software systems inevitably have complex failure modes; errors only triggered by some combination of latent software bugs, environmental conditions and/or administrative errors. While considerable effort is spent trying to eliminate such problems before deployment, 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 end-users. Consequently, many software failures still occur in fielded systems providing production services.

An earlier version of this article appeared in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*.
This research is supported by NSF CNS-0720743 grant, NSF CCF-0325603 grant, NSF CNS-0615372 grant, NSF CNS-0347854 (career award), NSF CSR Small 1017784 grant and NetApp Gift grant.
Author's addresses: D. Yuan (corresponding author), J. Zheng, S. Park, Y. Zhou, S. Savage, Department of Computer Science and Engineering, University of California, San Diego, La Jolla, CA 92093; email: diyuan@cs.ucsd.edu.

DOI 10.1145/2110356.2110360 http://doi.acm.org/10.1145/2110356.2110360

1.1. Production Failure Reporting

Production failures are problematic at two different levels. First, they demand tremendous urgency; a production failure can have direct impact on the customer's business, and system vendors must make the diagnosis and remediation their highest priority. Unfortunately, this goal conflicts with a second problem: the substantial difficulty in analyzing such failures. Indeed, diagnosing rare failures can be challenging even in a controlled setting, but production deployments are particularly daunting since often support engineers are given insufficient information to identify the root cause, let alone reproduce the problem in the lab.

To address this problem, a range of research efforts have focused on techniques for capturing external and non-deterministic inputs, thereby allowing post-mortem deterministic replay [Chen et al. 2008; Crameri et al. 2011; Devietti et al. 2009; Dunlap et al. 2008; Guo et al. 2008; LeBlanc and Mellor-Crummey 1987; Lee et al. 2010; King et al. 2005; Montesinos et al. 2008; Narayanasamy et al. 2005; Olszewski et al. 2009; Park et al. 2009; Subhraveti and Nieh 2011; Veeraraghavan et al. 2011; Vlachos et al. 2010; VMWare; Xu et al. 2003; Zhang et al. 2006]. However, these approaches have been slow to gain traction in the commercial world for several reasons, including high overhead, environmental complexity (e.g., interactions between multiple licensed software and hardware from different vendors), and substantive privacy concerns.

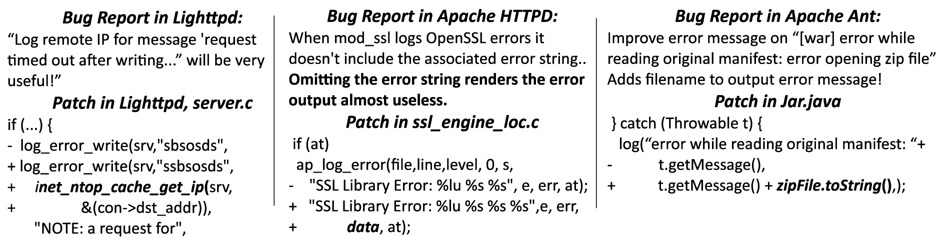
A more established vehicle for diagnosis is the "core dump", which captures memory context and execution state in a file. However, core dumps have their own drawbacks. They are typically collected only at the time of the crash failure. They only capture program state but no execution history information (which is frequently critical for diagnosis), and the comprehensive capture of process state can once again preclude sharing such files due to privacy concerns.[1]

Consequently, the sine qua non of production failure debugging remains the log file. Virtually all software systems, whether commercial or open source, log important events such as error or warning messages, as well as some *historic* intermediate progress/bookkeeping information generated during normal execution. It is a common industry practice for support engineers to request such logs of their customers upon failure, or even for customers to allow their systems to transmit such logs automatically (i.e., "call home" [Dell 2008]). Since these logs focus on the system's own status and "health", they are usually considered to be less sensitive than other data. Moreover, since they are typically human-readable, customers can inspect them first (either after a failure or during initial contract negotiations). Consequently, most modern systems today from EMC, NetApp, Cisco, Dell are able to collect logs from at least 50% of their customers, and many of them even have enabled the capability to automatically send logs to the vendor [Cisco; Dell 2008; EMC 2005; NetApp 2007].

1.2. Diagnosing via Log Messages

Thus in many cases, log messages are the sole data source available for vendors to diagnose reported failures. Support engineers then attempt to map log message content to source code statements and work backwards to infer what possible conditions might have led to the failure. While a range of research projects have shown that statistical machine learning techniques can be used to detect anomalies or catch recurring failures that match known issues [Aguilera et al. 2003; Barham et al. 2004; Cohen

[1]Some systems, such as Windows Error Reporting [Glerum et al. 2009] and Mozilla's Quality Feedback Agent [Mozilla QFA] attempt to mitigate the privacy issues through data minimization (typically limiting the scope of captured state to the execution context and stack trace) but at the cost of yet reduced debugging effectiveness. Indeed, these systems succeed because they aggregate large numbers of failures with common causes, rather than due to their ability to substantively aid in the debugging of any singular failure instance.



Bug Report in Lighttpd:

"Log remote IP for message 'request timed out after writing..." will be very useful!"

Patch in Lighttpd, server.c

```
if (...) {
-  log_error_write(srv,"sbsosds",
+ log_error_write(srv,"ssbsosds",
+    inet_ntop_cache_get_ip(srv,
+              &(con->dst_addr)),
     "NOTE: a request for",
```

Bug Report in Apache HTTPD:

When mod_ssl logs OpenSSL errors it doesn't include the associated error string.. **Omitting the error string renders the error output almost useless.**

Patch in ssl_engine_loc.c

```
 if (at)
  ap_log_error(file,line,level, 0, s,
-   "SSL Library Error: %lu %s %s", e, err, at);
+   "SSL Library Error: %lu %s %s %s",e, err,
+           data, at);
```

Bug Report in Apache Ant:

Improve error message on "[war] error while reading original manifest: error opening zip file" Adds filename to output error message!

Patch in Jar.java

```
 } catch (Throwable t) {
  log("error while reading original manifest: "+
-          t.getMessage(),
+          t.getMessage() + zipFile.toString(),);
```

Fig. 1. Example of real-world patches just for the purpose of enhancing log messages.

et al. 2005; Ha et al. 2007; Xu et al. 2009], the detective work of mapping log messages to source statements and then sifting through potential causes of individual crashes remains a heavily manual activity.

Recent work on the SherLog system [Yuan et al. 2010] addresses the first part of this problem by automating this manual inference process. SherLog is a post-mortem diagnosis tool that uses failure log messages as starting points to automatically infer what source code paths may have been executed during a failed execution. Although SherLog can conduct deeper inference than manual efforts, it is still limited by the amount of information available in log messages, just like manual inference by programmers. If a log message does not contain enough information, automatic log inference engines have limited starting information to disambiguate between different potential causal paths that led to a failure. It is precisely this limitation that motivates the work in this paper. In Section 4.2 we will show three real-world cases to demonstrate how automatic log inference engines like SherLog can perform better after log messages are enhanced with more causally-related information (automatically by *LogEnhancer*).

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 [Schmidt 2009] and the thread ID with each message [Yuan et al. 2010]), 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 1 shows three such enhancements, each of which expanded the log messages to capture distinct pieces of diagnostic state. In our work we propose to systematically and automatically add such enhancements to log messages, and thereby improve the diagnostic power of logging in general.

1.3. Our Contributions

In the remainder of this article, we present a tool called *LogEnhancer*, that modifies each log message in a given piece of software to collect additional causally-related information to ease diagnosis in case of failures.[2] To be clear: *LogEnhancer* does not detect bugs or failures itself. Rather it is a tool for reducing the burden of failure diagnosis by enhancing the information that programmers should have captured when

[2]We target for production failure diagnosis even though our work can also be useful for in-house testing and debugging.

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, we enhance log content in a very specific fashion, using program analysis to identify *which* state should be captured at each log point (a logging statement in source code) to minimize causal ambiguity. In particular, we say that 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 we identify which candidate variable values, if known, would resolve this ambiguity. Note we are 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.

We 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). We also develop 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).

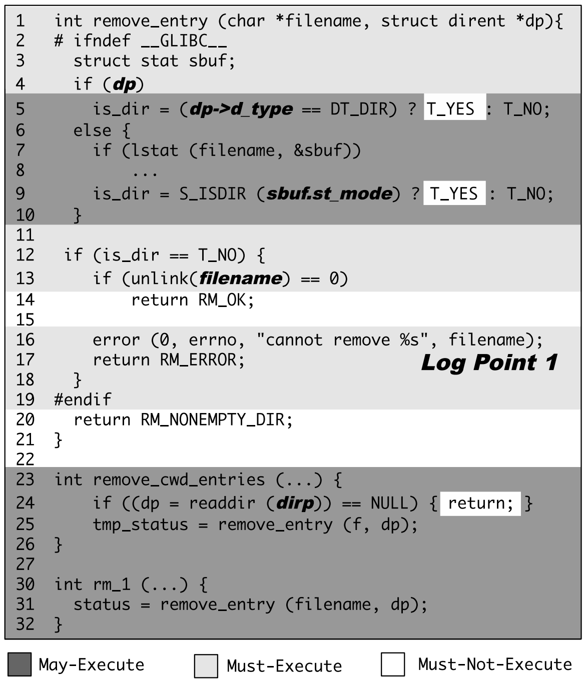
Finally, we evaluate *LogEnhancer* with 8 large, real-world applications (5 servers and 3 client applications). We find that *LogEnhancer* automatically identifies 95% of the same variable values that developers have added to their log messages over time. Moreover, it identifies an additional set of 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. We also selected 15 representative, real-world failures (with 13 caused by bugs and 2 caused by mis-configurations) from these applications 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 runtime states, helping both manual diagnosis and automatic log inference engines like SherLog to narrow down and identify the root causes. Finally, we show that both log size and run-time overhead are small, and almost negligible (with the delayed collection).

To the best of our knowledge, our work is the first attempt to systematically and automatically enhance log messages to collect causally-related information for diagnosis in case of failures. It can be used to enhance every existing log message (oblivious to what failure might occur in production) in the target software prior to release.

2. OVERVIEW

To explain how *LogEnhancer* works, we first examine how diagnosis is performed manually today. Figure 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
```



```
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
16      error (0, errno, "cannot remove %s", filename);
17      return RM_ERROR;
18    }
19  #endif
20    return RM_NONEMPTY_DIR;
21  }
22
23  int remove_cwd_entries (...) {
24      if ((dp = readdir (dirp)) == NULL) { return; }
25      tmp_status = remove_entry (f, dp);
26  }
27
30  int rm_1 (...) {
31    status = remove_entry (filename, dp);
32  }
```

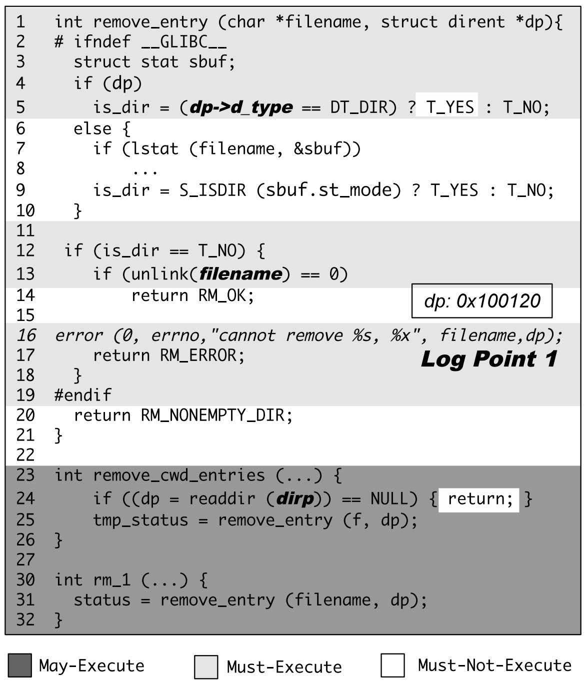
■ May-Execute ▒ Must-Execute □ Must-Not-Execute

Fig. 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.

2.1. 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 (`is_dir == T_NO`) is taken and the conditional at line 13 (`unlink(filename == 0)`) is not taken. This suggests that `rm` treated `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 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 2 as "May-Execute").

However, further inference of why `is_dir` equals `T_NO` is considerably more complicated. There are two possibilities for the branch at line 4, depending on the value of `dp`, and both paths may set `is_dir` to be `T_NO`. Further, since `dp` is a parameter, we must find the caller of `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. In Section 4.2 we will show that automatic log inference engines such as SherLog can do no better than manual inference in this case.



```
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
16  error (0, errno,"cannot remove %s, %x", filename,dp);
17      return RM_ERROR;
18    }
19  #endif
20    return RM_NONEMPTY_DIR;
21  }
22
23  int remove_cwd_entries (...) {
24      if ((dp = readdir (dirp)) == NULL) { return; }
25      tmp_status = remove_entry (f, dp);
26  }
27
30  int rm_1 (...) {
31    status = remove_entry (filename, dp);
32  }
```

Fig. 3. Remaining uncertainty if dp was printed at line 16. The code is the same as Figure 2.

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 is_dir must equal T_NO. However, why is_dir holds this value depends on data flow. Specifically, the value of is_dir was previously assigned at either line 5 or 9, and has data dependencies on either the value of dp->d_type or 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 2). Similarly, the values of dp->d_type and sbuf.st_mode are also uncertain, as is the context in which remove_entry() was called. While the ambiguity 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.

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 (we illustrate it in Figure 3, given this new information). In this case dp is nonzero, 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; we discuss this optimization in Section 3.2).

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 have described: 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 comprehensively to a program, thereby capturing the information needed to diagnose unanticipated failures that may occur in the future.

2.2. 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 [Necula et al. 2002], 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 recompiles 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 4.2 shows three such examples.

2.3. Architecture Overview

The complexity in our system 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_NO` 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 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, for instance, `[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.

2.4. 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 our analysis. We make 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. We briefly outline the issues surrounding our assumptions and their impact in the following.

(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 we can go as far and deep as we would like, it is impractical to do so for large real world software. Therefore, we need 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, we do 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 we limit our analysis in this fashion, we still identify 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 our 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 our 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 can we address the privacy concerns?* Just as existing log message contents, the information we record focuses narrowly on system's own "health." Because we are recording only 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., Castro et al. [2008] 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 [Yuan et al. 2010] to collect back the inferred and less-sensitive information (e.g. the execution path during the occurred failure).

(4) *How do we handle inter-thread or inter-process data dependencies?* Due to the limitation of our static analysis, we do not analyze data dependencies across threads or processes. Any values that are causally related to the log point through these dependencies thus would be missed. In most cases, such dependencies do not interfere with our 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, we may not log enough information to figure out why certain shared key variables have some particular values. The substeps (iii)–(v) in our 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, we might disable these substeps and conservatively treat any shared data as overwritten ones at the log point.

Note this limitation is not that we cannot handle concurrent programs. For concurrent programs, we still analyze 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 our evaluated applications are concurrent, including Apache, CVS, Squid, PostgreSQL and lighttpd. Section 4 shows our evaluation results on these applications. Note that a majority of failures in real world are caused by semantic bugs, and even misconfigurations, not by concurrency bugs [Li et al. 2006].

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 intrathread or intraprocess data flow, we also need to analyze any interthread or interprocess modifications. Although theoretically we can still use the same Uncertainty Identification algorithm to recursively follow intrathread/process *and* interthread/process data-flow, we imagine 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, we might end up analyzing huge number of dataflows that are not causally related to the log point. This might add exponential overhead to our analysis, and more importantly, include huge number of noisies in the set of variable values we decide record to enhance each log message. Annotations can be used in expressing which functions are concurrent [Detlefs et al. 1998; Flanagan et al. 2002], while techniques presented in RacerX can help to automatically infer this information [Engler and Ashcraft 2003]. Previous work [Naik and Aiken 2007; Naik et al. 2006] 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 our future work.

(5) *What if there is no log message?* If a software program generates no log message at all, *LogEnhancer* offers no value. Fortunately, this is usually not the case in most commercial and open source software. As mentioned in our Introduction, most commercial and open source software already contains significant amount of logs as logging has become a standard practice. Hence we focus on enhancing existing log messages, and *assume* that such log messages exist.

3. DESIGN AND IMPLEMENTATION

LogEnhancer's source code analysis is implemented using Saturn static analysis framework [Aiken et al. 2007]. Saturn models C programs precisely and allows user to express the analysis algorithm in a logic programming language. It is summary-based, which conducts its analysis for each function separately, and then a summary is generated for each function. At the calling sites of a function, its summary is used instead of going deep into this function. Saturn also provides a SAT solver.

In this section, due to space limit, we will not repeat the all the details of Saturn. Except for the data-flow analysis described in Section 3.1, all the discussions on analysis processes, design and implementation issues are specific to *LogEnhancer*.

3.1. 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. Our 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.

3.1.1. 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

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

Fig. 4. The constrained expression for `is_dir` at line 16. C_{yes} and C_{no} are constraints for `is_dir` to hold value `T_YES` and `T_NO` respectively.

all the memory locations (heap and stack) accessed by f, and initializes each location `V` as `V=V: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=exp;`, the value of `V` would be updated to `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 shows the CE of `is_dir` in `rm` at log point 1.

Each variable involved in the CE is a *live-in* variable to the function f, i.e., variable whose value is first read before written in f [Aho et al. 2006]. 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, `is_dir` is not a live-in variable, and its value can be represented by a small set of live-in values such as `dp, T_YES`, etc., as shown in Figure 4.

3.1.2. Control-Flow Analysis. At each program point P, *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 we can prune out such constraint. The satisfiable constraint thus contains all the causally related control and data-flow conditions to reach LP. Thus if we know all the variable values involved in this constraint C, we 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 `exit, abort, _exit` and their wrappers are identified and adjusted on the CFG. `longjmps` are correlated with `setjmps` through function summary in a similar manner as described in Yuan et al. [2010].

In the `rm` example, the control-flow constraint within `remove_entry` to reach log point 1 would be `is_dir==T_NO && unlink(filename)!=0`. Then `is_dir` is replaced by its CE, as shown in Figure 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.

$$C_r = \texttt{(dp \&\& dp->d\_type!=DT\_DIR || !dp \&\& !S\_ISDIR(sbuf.st\_mode)) \&\& unlink(filename)}$$

3.1.3. 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 callsite 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 = `(readdir(dirp)!=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 `= (readdir(dirp) && readdir(dirp)->d_type!=DT_DIR || !readdir(dirp)) && f==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==Sym` is the substitution for `unlink (filename)`. `Sym` is a symbolic value and `f` is the substitution of `filename` in caller. `f==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, $C'_r \wedge C_c$ is stored as the summary for `remove_cwd_entries` to reach log point 1.

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—we refer them as "sibling functions". 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 we limit the analysis to follow only one level into such functions due to scalability concern. However, there is no theoretical limit, and we 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), we simply plug in its parameter into our constraint so we may choose to record the parameter.

3.1.4. Handling Loops. Loops are modeled as tail-recursive functions [Aiken et al. 2007], 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==Sym`, to relax the value constraint since we are not following the multiple iterations as in runtime. This way, constraint from the loop body can be conservatively captured.

3.1.5. Efficiency and Scalability. Uncertainty Identification scans the program linearly, a key to our scalability to large applications. We further use *preselection* and *lazy SAT solving* for optimization. The former preselects 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.

3.1.6. Pointer Aliasing. Intra-procedural pointer aliasing is precisely modeled by Saturn's constrained expression model [Aiken et al. 2007]. 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 nonaliased, which might cause us to miss some causally-related variable values. Note that for Value Selection we will enable inter-procedural alias analysis for all types of pointers for conservative liveness checking.

3.2. 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, we refer an expression without any Boolean operator (`&&`, `||`, `!`) as a *uni-condition*. For example, `dp!=NULL` is a unicondition (note `!=` is not one of the three Boolean operators). A constraint is thus a formula of uniconditions combined together using Boolean operators.

(1) *Pruning determinable values.* Some variable values can be inferred knowing that a given log point is executed. We 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 previous step, all remaining values are uncertain. However, not every value is meaningful under all circumstances. In `rm`, `dp->d_type` is meaningful only if `dp!=NULL`. Recording a nonmeaningful value could result in an invalid pointer dereference or reading a bogus value. Therefore, for each value that is not pruned, we also identify 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 we cannot delay the recording until the log point. To identify such dead values, we perform conservative liveness analysis, that is, if one variable value might be modified before the log point, we conservatively mark it as "dead". To be conservative, we run Saturn's global pointer alias analysis [Hackett and Aiken 2006] before the liveness checking. Any pointers passed into a library call where source code is unavailable are conservatively treated as dead after the call (we manually exclude some common C libraries such as `strlen`); Any `extern` values not defined inside the program are also conservatively treated as dead.
However, we do not give up on recording dead values so easily. For each dead value, we try 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 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`, we simply record `EV` and `UV`.
(4) *Ranking.* Finally, we rank all selected values based on the amount of uncertain branch conditions they contribute. Ranking can be used to prioritize our runtime 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 3.1.2, the constraint to reach the log point 1 in the `rm` example is:

```
Cr = (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 in our setting, we do not set any threshold on the number of variables we record at log point. So the ranking is only used when presenting the recorded values to users.

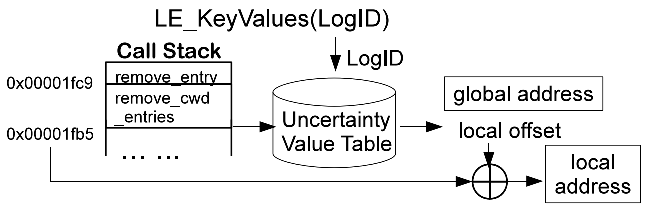


Fig. 5. Runtime recording for Delayed Collection.

3.3. Runtime Value Collection

This section describes how *LogEnhancer* modifies the application's original source code to collect the variable values. We describe two value collection policies: delayed collection and in-time collection.

Delayed Collection. We instrument 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 [DWARF]. 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 5 shows the runtime process of `LE_KeyValues()`. It is triggered only at the log point, that is, 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 `LE_KeyValues()`, so the delayed recording does not add any overhead during normal execution, that is, when no log message being printed. We also record the dynamic call stack.

By default, *LogEnhancer* records only basic type values. For example, for a pointer value, we only record the address stored in this pointer. To further provide meaningful diagnostic information, we add two extensions. First, if the variable is of type `char*` and is not NULL, then we record the string with a maximum of 50 characters (of course, if the string is shorter than 50, we record only the string). Second, if the variable is a field within a structure, in addition to that field, we also record 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 we are already very cautious in our design to record only meaningful and valid values to ensure memory safety, due to the limitation of static analysis, we might still access an invalid memory location (e.g., caused by multi-threading). To be conservative, we further ensure memory safety by intercepting SIGSEGV signals without crashing the running application. For applications such as apache who also intercepts SIGSEGV signals, we add a wrapper to filter out those obviously caused by our log recording. In our experiments, we have never encountered such signal.

We also implement 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

Table I. Evaluated Applications

Application	Version	Lines of code	Log Points	
			All	Default verbose level
ln	4.5.1	20K	26	14 (ERROR)
rm	4.5.4	18K	28	25 (ERROR)
tar	1.22	66K	210	176 (ERROR)
apache	2.2.2	228K	1,654	1,093 (WARNING)
cvs	1.11.23	111K	1,088	762 (ERROR)
squid	2.3.S4	70K	1,116	402 (ERROR)
postgresql	8.4.1	825K	4,876	4,403 (WARNING)
lighttpd	1.4.26	53K	127	127 (ERROR)

Lines of code is measured by SLOCCount [SLOCCount]. 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 verbose level" shows the number of log points under the default verbose-level. The description of the default verbose level is in the brackets.

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 bookkeeping 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:

```
- if (X)
+ if (LE_InTime(&X, Lint32) && X)
```

`LE_InTime()` always returns 1. It simply copies `Lint32` number of bytes starting from `&X`. Note that `LE_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 `LE_KeyValue()` and `LE_InTime()` are first stored into buffers in memory (both currently 40 KB), respectively. At error messages, both buffers were flushed to disk. `LE_KeyValue()`'s buffer is also flushed when it becomes full, whereas `LE_InTime()` simply recycles the shadow buffer from the beginning. Each thread has its own private buffer.

4. EVALUATION

We use *LogEnhancer* to enhance each of the total 9,125 log messages in 8 different real-world applications as shown in Table I. 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, we enhance all messages printing into the error log file. Currently we do not enhance other types of log files such as access logs. 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, we try to evaluate *LogEnhancer* both quantitatively and qualitatively using three sets of experiments.

(1) *Value selection.* First, we investigate how well our algorithm captures the variables that are useful for failure diagnosis by comparing against manual selection

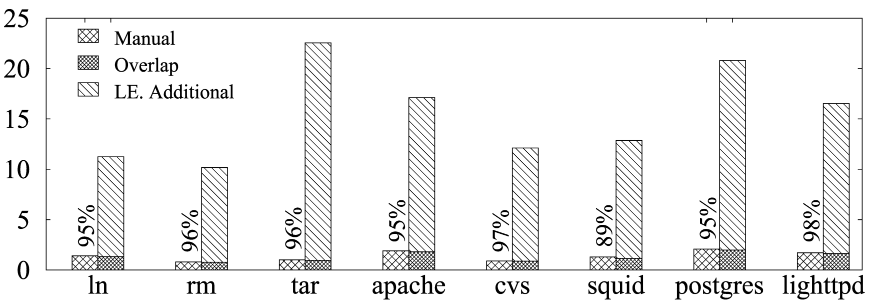


Fig. 6. Variable values at each log point. We compare the number of variables per message logged manually by developers with the ones inferred automatically by *LogEnhancer*. "Overlap" shows the number of variable values that are selected by both programmers and *LogEnhancer*. The percentages of overlap are marked beside each bar. "LE-additional" shows the additional variable values only identified by *LogEnhancer*.

(variables that have already been recorded in existing logging statements by programmers). Then, we 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 we select 15 real world failure cases caused by 13 bugs and 2 mis-configurations to show the effectiveness of the information collected by *LogEnhancer* in failure diagnosis. In particular, we 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 offline, *LogEnhancer* currently runs as single process, single thread (even though the analysis can potentially be parallelized to reduce the analysis time [Aiken et al. 2007]).

4.1. Effectiveness in Variable Recording

Figure 6 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 applications except `squid`, *LogEnhancer* achieves a coverage over 95%.[3] 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 bookkeeping 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

[3]Many variable values are converted to human readable strings when printing to log message. For example "inet_ntoa" converts an IP address into string. We count the value as covered by *LogEnhancer* only if by recording the non-textual value we can *deterministically* infer text string.

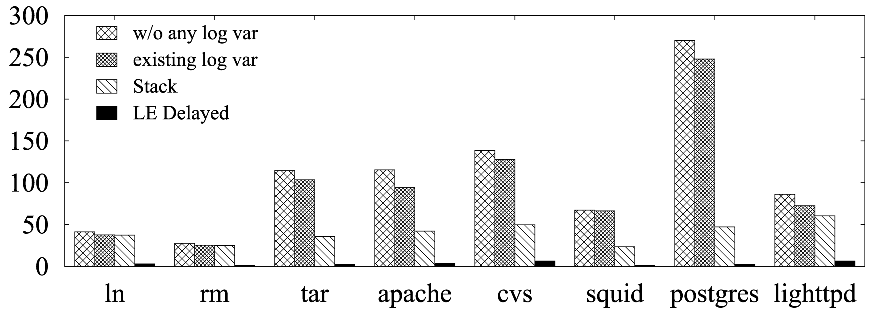


Fig. 7. Number of Uncertain Branches reduced by *LogEnhancer*. We compare the amount of uncertain branches that are causally related to each log point given different types of information recorded: without any variables (the original uncertainty space); existing variables included by developers; call stack in addition to existing variables; variables inferred by *LogEnhancer* and call stack using the delayed collection method.

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) Bookkeeping 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* covered only 88.7% of the existing log variables in Squid: majority (64% as shown in Table I) 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) Redundant multivariables (8%) that are always updated together while only one is used in branch. *LogEnhancer* only identifies the one used in branch while the missed values can be inferred from the identified one.

In addition to automatically select most of existing log variables (manually included by programmers), *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 7. From the 108 original uncertain branches per log point, existing log variables can reduce it to 97, whereas the *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 we record 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 II shows the detailed number of uncertain branches.

Table II also shows the number of variable values identified by *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

Table II. The Number of Uncertain Branches and Uncertain Variable Values per Log Point

Application	Uncertain Branches								Number of variables		
	w/o recording any variable				*LogEnhancer* (delay)				all	live	logged
	avg	med	max	min	avg	med	max	min			
ln	41	43	78	7	2.9	1	8	0	11.3	9.8	10.1
rm	28	27	57	6	1.4	1	9	0	10.2	9.3	9.5
tar	114	35	1419	2	2.2	1	20	0	22.6	19.5	21.6
apache	115	78	626	1	3.5	2	35	0	17.2	14.7	15.9
cvs	139	62	3836	1	6.5	3	38	0	12.2	8.7	10.6
squid	67	19	4409	1	1.3	0	17	0	13.0	11.6	12.5
postgre	270	61	22070	1	1.2	0	48	0	20.9	14.7	18.1
lighttpd	86	88	222	5	6.4	6	40	0	20.7	15.2	18.8

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.

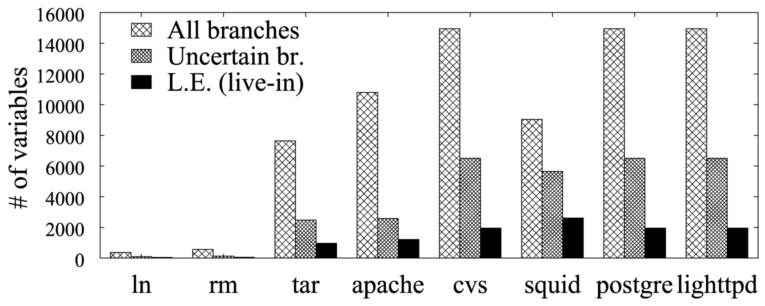


Fig. 8. Number of variables used in branches.

normal-run overhead. Among these 14.6 variables, 12.9 of them were not overwritten before log point (i.e., they are “live”), and 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 8 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 through out 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 II).

Ranking of variable values. Figure 9 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 we can eliminate an average

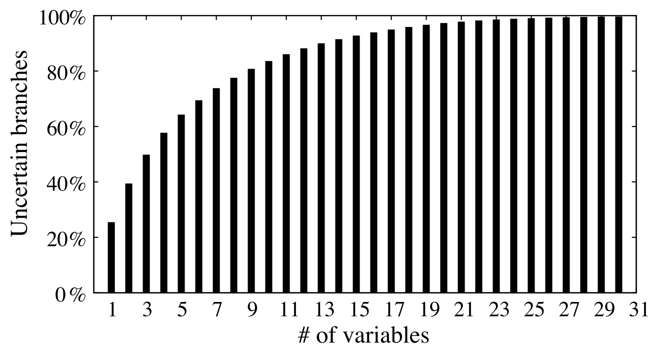


Fig. 9. Ranking of uncertain variable values in Apache. We show the accumulated number of uncertain branches each variable involved.

Table III. Analysis Performance

Analysis Time and Memory Usage					
ln	3 minutes	579MB	rm	2 minutes	172MB
tar	1.5 hours	263MB	apache	2.1 hours	1.3GB
cvs	3.0 hours	1.7GB	squid	3.8 hours	2.3GB
postgres	10.7 hours	1.5GB	lighttpd	20 minutes	532MB

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 III 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 we expect *LogEnhancer* to be used offline prior to software release, the analysis time is less critical. Besides, the summary-based design allows it to be parallel or incrementally applied [Aiken et al. 2007]. The memory usage in all cases is below 2.3GB.

4.2. Real-World Failures

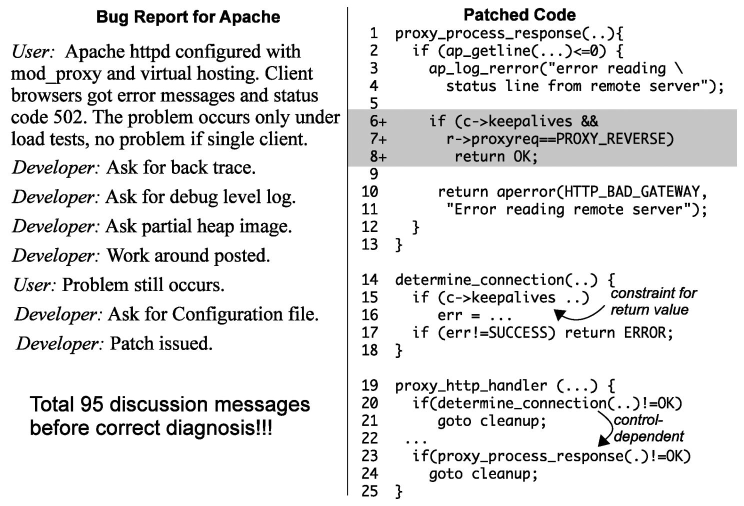
We evaluated *LogEnhancer* by analyzing 15 real-world failures, including 13 software bugs and 2 configuration errors, to see how our enhanced log messages would help failure diagnosis. In all these cases, the original log messages were insufficient to diagnose the failure due to many remaining uncertainties,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

We also compared the inference results of SherLog [Yuan et al. 2010] (referred to as SherLog in this section) before and after *LogEnhancer*'s enhancement.

Case 1 *rm*. For the `rm` failure described in Figure 2, *LogEnhancer* recorded the call stack 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

Table IV. Real-World Failures Evaluated by Us

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



Bug Report for Apache

User: Apache httpd configured with mod_proxy and virtual hosting. Client browsers got error messages and status code 502. The problem occurs only under load tests, no problem if single client.

Developer: Ask for back trace.

Developer: Ask for debug level log.

Developer: Ask partial heap image.

Developer: Work around posted.

User: Problem still occurs.

Developer: Ask for Configuration file.

Developer: Patch issued.

Total 95 discussion messages before correct diagnosis!!!

Patched Code

```
1   proxy_process_response(..){
2     if (ap_getline(...)<=0) {
3       ap_log_rerror("error reading \
4         status line from remote server");
5
6+      if (c->keepalives &&
7+        r->proxyreq==PROXY_REVERSE)
8+         return OK;
9
10        return aperror(HTTP_BAD_GATEWAY,
11         "Error reading remote server");
12    }
13  }

14  determine_connection(..) {
15    if (c->keepalives ..)
16       err = ...
17    if (err!=SUCCESS) return ERROR;
18  }

19  proxy_http_handler (...) {
20    if(determine_connection(..)!=OK)
21       goto cleanup;
22   ...
23    if(proxy_process_response(.)!=OK)
24      goto cleanup;
25  }
```

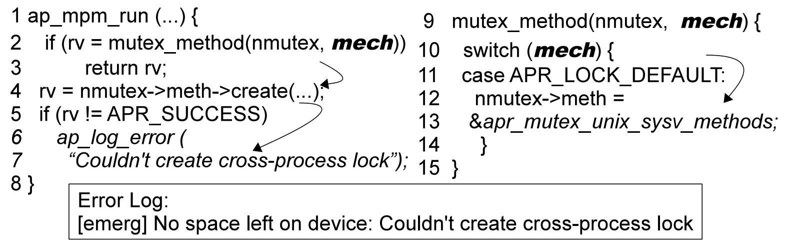
Fig. 10. Apache bug example. Patched code is highlighted.

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 follows:

```
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 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 10 shows a bug report in apache. With only the error log message at line 3, the developer could not diagnose the failure. So he asked the user for all kinds of run-time information in a total of *95* message exchanges. Actually



```
1 ap_mpm_run (...) {
2   if (rv = mutex_method(nmutex, mech))
3        return rv;
4   rv = nmutex->meth->create(...);
5   if (rv != APR_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     }
15 }
```

```
Error Log:
[emerg] No space left on device: Couldn't create cross-process lock
```

Fig. 11. Apache configuration error. The dependencies to identify variable `mech` are marked as arrows.

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->keepalives` in this function.

Without *LogEnhancer*'s enhancement, SherLog inferred 63 possible call paths and not be able to infer the value of `c->keepalives` nor `r->proxyreq`. With *LogEnhancer*'s enhancement, SherLog can narrow down to only one possible call path, and infer the value of `c->keepalives` and `r->proxyreq`.

Case 3 *Apache configuration error.* A misconfiguration in Apache resulted in a failure with the log message shown in Figure 11. 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 `create()` at line 4. However, developers had no other clues why this call failed.

LogEnhancer in this case identifies `mech` as a key value to collect, since it is used at line 10 in function `mutex_method`, whose `nmutex` is causally related to the log point. If apache had been enhanced by *LogEnhancer*, the log message would record the value of `mech` being `APR_LOCK_DEFAULT` and the value of `nmutex->meth` being `apr_mutex_unix_sysv_methods`. This indicates that apache was using the default lock setting which caused the failure. In a multithreaded mode, apache should use fnctl-based lock instead. To fix this, users should explicitly add "AcceptMutex fcntl" into the configuration file.

Note that, without *LogEnhancer*'s enhancement, SherLog cannot infer the value of `mech` from the original log message and would not be able to narrow down to the lock setting configuration as the root cause.

4.3. Overhead

Execution time. Table V shows the *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

Table V. Overhead of *LogEnhancer*

Applications and **Slow-down**											
rm	0.0%	<1.0%	tar	0.0%	1.5%	apache	0.0%	3.9%	postgre	0.0%	7.6%
ln	0.0%	<1.0%	cvs	0.0%	1.7%	squid	0.0%	8.2%	lighttpd	0.0%	3.4%

The first number is the overhead for the delayed collection, and the second is for the in-time collection.

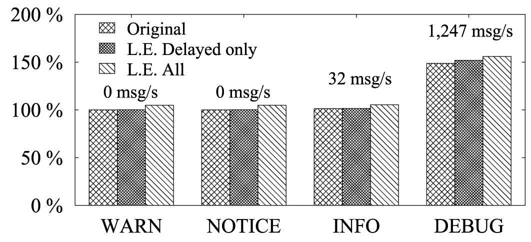


Fig. 12. Normal execution overhead on fully loaded Apache for different verbose modes. These overheads are normalized over the unmodified code under default verbose level (WARN).

Table VI. Comparison between *LogEnhancer* and Core Dump

Failure	Time (ms)		Size (bytes)		
	LogEnhancer	coredump	original log	*LogEnhancer*	coredump
ln	0.45	630	45	62	55M
rm	0.45	610	51	176	55M
tar 2	0.39	630	95	93	55M
cvs	0.44	60	52	53	772K
apache 1	0.41	670	196	354	3.2M

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%.

Figure 12 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 3.3, 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 our experiment. If a log point is executed at run-time, `LE_KeyValues()` 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 VI compares *LogEnhancer*'s recording time and data size with core dump at a failure. We reproduced 5 failures in Table IV and forced a core dump to be generated at each log point using `gcore` [GCORE] library call. The log size of *LogEnhancer* does not include the size of the original log.

On average, *LogEnhancer* only needs 0.43 millisecond to print the log message. This time includes the time to print the original log message and the additional variable values. The size of additionally recorded data is only 53-354 bytes on average. In comparison, core dumps require 1000 times recording time, and 55MB in size. The large overhead of core dump makes it impractical to be generated at each log message. Note that as the core dump's size/time largely depends on the memory footprint size at the time of the failure, the purpose of our comparison here is merely to show the magnitude differences between the two.

Table VI shows that *LogEnhancer*'s log size is 53-354 bytes, which is in the same magnitude as original log message. A large portion of this log is the call stack encode in clear text. We can further compress this portion since calling contexts are likely to be the same for a log point.

5. RELATED WORK

Our work builds on previous efforts in a number of areas. In this section, we discuss each in turn.

5.1. Log Analysis for Failure Diagnosis

Existing log analysis work focuses on post-mortem diagnosis using logs. Some learn statistical signatures to detect anomalies [Aguilera et al. 2003; Barham et al. 2004; Cohen et al. 2005; Ha et al. 2007; Xu et al. 2009] while others inferring partial execution paths and run-time states [Yuan et al. 2010]. Xu et al. [2009] use statistical techniques to efficiently learn a decision tree based signature from large amount of console logs. This signature can be used to effectively detect and diagnose anomalies.

SherLog [Yuan et al. 2010] uses static analysis to infer the partial execution paths that can connect the runtime log messages. It infers both control and data value information post-mortemly, providing a similar user-experience as an interactive debugger without dynamically reexecuting the program.

LogEnhancer is different but complementary with these log analysis work like SherLog [Yuan et al. 2010] in several aspects.

(1) *LogEnhancer* has a completely different focus: it aims to improve software's diagnose-ability by adding more causally related information in log messages to make failure diagnosis easier. Such information benefits not only manual diagnosis but also automatic log analysis engines like SherLog.
(2) *LogEnhancer* logs only those variables that cannot be inferred (manually or automatically with SherLog) from what are already available in log messages. SherLog cannot resolve the 108 causally related uncertain branches to each log message that are identified by *LogEnhancer* given only existing log messages.
(3) Although *LogEnhancer* leverages summary-based static analysis similar to SherLog, the different objectives lead to several major new design and implementation issues. For example, *LogEnhancer* needs to perform uncertain control/data identification, value selection, liveness analysis, equivalent variable identification, and finally instrumenting the source code and log those select variables at runtime. None of these would be needed in a log inference engine like SherLog.
(4) As the real world case studies in Section 4.2 have shown, SherLog can significantly benefit from *LogEnhancer*'s log enhancement information.

5.2. Logging Design

Existing guidelines for logging design are purely empirical [Kernighan and Pike 1999; Schmidt 2009]. Kernighan and Pike [1999] argued the importance of well-designed logging in diagnosis. Schmidt [2009] summarized some empirical logging practices. To the best of our knowledge, *LogEnhancer* is one of the first to automatically enhance log messages.

5.3. Use of End Execution State for Failure Diagnosis

Several systems collect partial memory image when a system crashes. Windows Error Reporting [Glerum et al. 2009] monitors the system for crashes or hangs, and records a "mini-dump." Crash Reporter [Apple 2004], NetApp Savecore [NetAppSavecore] and Google Breakpad [GoogleBreakpad] also collect compressed memory dumps.

Some work infer diagnostic information from core dump or other end execution states. Their techniques are applicable on *LogEnhancer*'s recording result as well. PSE [Manevich et al. 2004] can perform off-line diagnosis of program crashes from core dump. Weeratunge et al. [2010] diagnose Heisenbugs by diff-ing the core dumps from failing run and passing run. ESD [Zamfir and Candea 2010] uses static analysis to infer a feasible path from the coredump and error report.

As discussed early in Introduction, our work is complementary to core dumps. *LogEnhancer* 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, our Core Dump Digger derives equivalent information as delayed collection from a core dump if a core dump is available.

Triage [Tucek et al. 2007] 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 checkpointing, Triage requires OS kernel modification and support. In comparison, our approach is much more light-weight. *LogEnhancer* does not need any special support from the OS or third party application/library.

5.4. Profiling for Diagnosis

Many of diagnostic tools collect run-time profiling such as low-level performance counters [Bhatia et al. 2008; Cohen et al. 2005] or execution traces [Ayers et al. 2005; Chen et al. 2008; Chilimbi et al. 2009; Ha et al. 2007; Liblit et al. 2003; Vlachos et al. 2010; Zhao et al. 2008]. Liblit et al. [2003] sample profiling information from many users to offload the monitoring overhead, and isolate the most correlated information using statistical techniques. Clarify [Ha et al. 2007] does instruction level profiling for normal and failure runs, and trains a classifier to classify each profile into to some known problems. Chen et al. [2008] propose hardware solution to accelerate instruction-level monitoring. Rather than collecting tailored, causally related information for each log message as *LogEnhancer*, these profiling tools collect general information. Our work is complementary to these work in that we collect causally related information specific to *each* log messages.

5.5. Logging for Deterministic Replay

Many works [Crameri et al. 2011; Devietti et al. 2009; Dunlap et al. 2008; Guo et al. 2008; King et al. 2005; LeBlanc and Mellor-Crummey 1987; Lee et al. 2010; Montesinos et al. 2008; Narayanasamy et al. 2005; Olszewski et al. 2009; Subhraveti and Nieh 2011; Veeraraghavan et al. 2011; VMWare; Xu et al. 2003; Zhang et al. 2006] targets deterministic replay of failed execution, which generally requires high runtime logging overhead especially for multiprocessor systems. To reduce the overhead, recently DoublePlay made clever use of spare cores on multiprocessor. Our work is complementary and mainly targets to the cases when failure reproduction is difficult due to privacy concerns, unavailability of execution environments, etc.

5.6. Other Static Analysis Work

Compiler techniques similar to *LogEnhancer* are also used to address some other software reliability problems [Cadar et al. 2008; Costa et al. 2007; Kadav et al. 2009; Zamfir and Candea 2010]. KLEE [Cadar et al. 2008] and ESD [Zamfir and Candea 2010] use full symbolic execution engine to expose bugs in testing or infer paths from core dump. Although *LogEnhancer* also uses 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. In addition, our work also has to use many other techniques and analysis such as control/data flow analysis, variable liveness analysis, equivalent variable analysis, runtime value collection, etc.

6. CONCLUSIONS

In this article we presented a tool, *LogEnhancer*, perhaps as the first work to systematically enhance every log message in software to collect causally related diagnostic information. We applied *LogEnhancer* uniformly on 9,125 different log messages in 8 applications including 5 server applications. Interestingly, we found 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.

ACKNOWLEDGMENTS

The authors would like to thank our ASPLOS shepherd Todd C. Mowry and the ASPLOS and TOCS reviewers for their insightful feedback, the UCSD Opera research group and the UCSD system and networking group for many lively discussions, and Michael Vrable for his careful article proofreading.

REFERENCES

AGUILERA, M. K., MOGUL, J. C., WIENER, J. L., REYNOLDS, P., AND MUTHITACHAROEN, A. 2003. Performance debugging for distributed systems of black boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*. ACM, New York, 74–89.

AHO, A. V., LAM, M. S., SETHI, R., AND ULLMAN, J. D. 2006. *Compilers: Principles, Techniques, and Tools* 2nd Ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

AIKEN, A., BUGRARA, S., DILLIG, I., DILLIG, T., HACKETT, B., AND HAWKINS, P. 2007. An overview of the saturn project. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'07)*. ACM, New York, NY, 43–48.

APPLE. 2004. Apple Inc., CrashReport. Tech. rep. TN2123.

AYERS, A., SCHOOLER, R., METCALF, C., AGARWAL, A., RHEE, J., AND WITCHEL, E. 2005. Traceback: First fault diagnosis by reconstruction of distributed control flow. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*. ACM, New York, NY, 201–212.

BARHAM, P., DONNELLY, A., ISAACS, R., AND MORTIER, R. 2004. Using magpie for request extraction and workload modelling. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design and Implementation*. USENIX Association, Berkeley, CA, 18–18.

BHATIA, S., KUMAR, A., FIUCZYNSKI, M. E., AND PETERSON, L. 2008. Lightweight, high-resolution monitoring for troubleshooting production systems. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, 103–116.

CADAR, C., DUNBAR, D., AND ENGLER, D. 2008. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, 209–224.

CASTRO, M., COSTA, M., AND MARTIN, J.-P. 2008. Better bug reporting with better privacy. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, NY, 319–328.

CHEN, S., KOZUCH, M., STRIGKOS, T., FALSAFI, B., GIBBONS, P. B., MOWRY, T. C., RAMACHANDRAN, V., RUWASE, O., RYAN, M., AND VLACHOS, E. 2008. Flexible hardware acceleration for instruction-grain program monitoring. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA'08)*. IEEE Computer Society, Los Alamitos, CA, 377–388.

CHILIMBI, T. M., LIBLIT, B., MEHRA, K., NORI, A. V., AND VASWANI, K. 2009. HOLMES: Effective statistical debugging via efficient path profiling. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*. IEEE Computer Society, Los Alamitos, CA, 34–44.

CISCO. Cisco system log management.

COHEN, I., ZHANG, S., GOLDSZMIDT, M., SYMONS, J., KELLY, T., AND FOX, A. 2005. Capturing, indexing, clustering, and retrieving system history. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*. ACM, New York, NY, 105–118.

COSTA, M., CASTRO, M., ZHOU, L., ZHANG, L., AND PEINADO, M. 2007. Bouncer: securing software by blocking bad input. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP'07)*. ACM, New York, NY, 117–130.

CRAMERI, O., BIANCHINI, R., AND ZWAENEPOEL, W. 2011. Striking a new balance between program instrumentation and debugging time. In *Proceedings of the 6th Conference on Computer Systems (EuroSys'11)*. ACM, New York, NY, 199–214.

DELL. 2008. Streamlined troubleshooting with the Dell system E-Support tool. Dell Power Solutions.

DETLEFS, D. L., LEINO, K. R. M., RUSTAN, K., LEINO, M., NELSON, G., AND SAXE, J. B. 1998. Extended static checking. Compac SRC Research rep. 159.

DEVIETTI, J., LUCIA, B., CEZE, L., AND OSKIN, M. 2009. Dmp: deterministic shared memory multiprocessing. In *Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09)*. ACM, New York, NY, 85–96.

DUNLAP, G. W., LUCCHETTI, D. G., FETTERMAN, M. A., AND CHEN, P. M. 2008. Execution replay of multiprocessor virtual machines. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'08)*. ACM, New York, NY, 121–130.

DWARF. The DWARF Debugging Format. `http://dwarfstd.org`.

EMC. 2005. EMC seen collecting and managing log as key driver for 94 percent of customers.

ENGLER, D. AND ASHCRAFT, K. 2003. Racerx: Effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*. ACM, New York, NY, 237–252.

FLANAGAN, C., LEINO, K. R. M., LILLIBRIDGE, M., NELSON, G., SAXE, J. B., AND STATA, R. 2002. Extended static checking for java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02)*. ACM, New York, NY, 234–245.

GCORE. Man page for gcore (Linux section 1).

GLERUM, K., KINSHUMANN, K., GREENBERG, S., AUL, G., ORGOVAN, V., NICHOLS, G., GRANT, D., LOIHLE, G., AND HUNT, G. 2009. Debugging in the (very) large: Ten years of implementation and experience. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP'09)*. ACM, New York, NY, 103–116.

GOOGLEBREAKPAD. Google Inc., Breakpad. `http://code.google.com/p/google-breakpad/`.

GUO, Z., WANG, X., TANG, J., LIU, X., XU, Z., WU, M., KAASHOEK, M. F., AND ZHANG, Z. 2008. R2: an application-level kernel for record and replay. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, 193–208.

HA, J., ROSSBACH, C. J., DAVIS, J. V., ROY, I., RAMADAN, H. E., PORTER, D. E., CHEN, D. L., AND WITCHEL, E. 2007. Improved error reporting for software that uses black-box components. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*. ACM, New York, NY, 101–111.

HACKETT, B. AND AIKEN, A. 2006. How is aliasing used in systems software? In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT'06/FSE-14)*. ACM, New York, NY, 69–80.

KADAV, A., RENZELMANN, M. J., AND SWIFT, M. M. 2009. Tolerating hardware device failures in software. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP'09)*. ACM, New York, NY, 59–72.

KERNIGHAN, B. W. AND PIKE, R. 1999. *The Practice of Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

KING, S. T., DUNLAP, G. W., AND CHEN, P. M. 2005. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the USENIX Annual Technical Conference (ATEC'05)*. USENIX Association, Berkeley, CA, 1–1.

LEBLANC, T. J. AND MELLOR-CRUMMEY, J. M. 1987. Debugging parallel programs with instant replay. *IEEE Trans. Comput. 36*, 471–482.

LEE, D., WESTER, B., VEERARAGHAVAN, K., NARAYANASAMY, S., CHEN, P. M., AND FLINN, J. 2010. Respec: Efficient online multiprocessor replayvia speculation and external determinism. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'10)*. ACM, New York, NY, 77–90.

LI, Z., TAN, L., WANG, X., LU, S., ZHOU, Y., AND ZHAI, C. 2006. Have things changed now?: An empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability (ASID'06)*. ACM, New York, NY, 25–33.

LIBLIT, B., AIKEN, A., ZHENG, A. X., AND JORDAN, M. I. 2003. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*. ACM, New York, NY, 141–154.

MANEVICH, R., SRIDHARAN, M., ADAMS, S., DAS, M., AND YANG, Z. 2004. PSE: explaining program failures via postmortem static analysis. In *Proceedings of the 12th International Symposium on the Foundations of Software Engineering.* 63–72.

MONTESINOS, P., CEZE, L., AND TORRELLAS, J. 2008. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA'08)*. IEEE Computer Society, Los Alamitos, CA, 289–300.

MOZILLA QFA. Mozilla Quality Feedback Agent. `http://kb.mozillazine.org/Quality_Feedback_Agent`.

NAIK, M. AND AIKEN, A. 2007. Conditional must not aliasing for static race detection. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*. ACM, New York, NY, 327–338.

NAIK, M., AIKEN, A., AND WHALEY, J. 2006. Effective static race detection for java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'06)*. ACM, New York, NY, 308–319.

NARAYANASAMY, S., POKAM, G., AND CALDER, B. 2005. Bugnet: Continuously recording program execution for deterministic replay debugging. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA'05)*. IEEE Computer Society, Los Alamitos, CA, 284–295.

NECULA, G. C., MCPEAK, S., RAHUL, S. P., AND WEIMER, W. 2002. CIL: Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of the 11th International Conference on Compiler Construction (CC'02)*. Springer-Verlag, Berlin, 213–228.

NETAPP. 2007. Proactive health management with auto-support. NetApp white paper.

NETAPPSAVECORE. NetApp Inc., Savecore. ONTAP 7.3 Manual Page Reference, Volume 1, 471–472.

OLSZEWSKI, M., ANSEL, J., AND AMARASINGHE, S. 2009. Kendo: Efficient deterministic multithreading in software. In *Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09)*. ACM, New York, NY, 97–108.

PARK, S., ZHOU, Y., XIONG, W., YIN, Z., KAUSHIK, R., LEE, K. H., AND LU, S. 2009. Pres: Probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP'09)*. ACM, New York, NY, 177–192.

SCHMIDT, S. 2009. 7 more good tips on logging. `http://codemonkeyism.com/7-more-good-tips-on-logging/`.

SLOCCOUNT. Sloccount. `http://www.dwheeler.com/sloccount/`.

SUBHRAVETI, D. AND NIEH, J. 2011. Record and transplay: Partial checkpointing for replay debugging across heterogeneous systems. In *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'11)*. ACM, New York, NY, 109–120.

TUCEK, J., LU, S., HUANG, C., XANTHOS, S., AND ZHOU, Y. 2007. Triage: Diagnosing production run failures at the user's site. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP'07)*. ACM, New York, NY, 131–144.

VEERARAGHAVAN, K., LEE, D., WESTER, B., OUYANG, J., CHEN, P. M., FLINN, J., AND NARAYANASAMY, S. 2011. Doubleplay: Parallelizing sequential logging and replay. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*. ACM, New York, NY, 15–26.

VLACHOS, E., GOODSTEIN, M. L., KOZUCH, M. A., CHEN, S., FALSAFI, B., GIBBONS, P. B., AND MOWRY, T. C. 2010. Paralog: Enabling and accelerating online parallel monitoring of multithreaded applications.

In *Proceedings of the 15th Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS'10)*. ACM, New York, NY, 271–284.

VMWARE. Using the intergrated virtual debugger for visual studio. `http://www.vmware.com/pdf/ws65_manual.pdf`.

WEERATUNGE, D., ZHANG, X., AND JAGANNATHAN, S. 2010. Analyzing multicore dumps to facilitate concurrency bug reproduction. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'10)*. ACM, New York, NY, 155–166.

XU, M., BODIK, R., AND HILL, M. D. 2003. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA'03)*. ACM, New York, NY, 122–135.

XU, W., HUANG, L., FOX, A., PATTERSON, D., AND JORDAN, M. I. 2009. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP'09)*. ACM, New York, NY, 117–132.

YUAN, D., MAI, H., XIONG, W., TAN, L., ZHOU, Y., AND PASUPATHY, S. 2010. Sherlog: Error diagnosis by connecting clues from run-time logs. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'10)*. ACM, New York, NY, 143–154.

ZAMFIR, C. AND CANDEA, G. 2010. Execution synthesis: A technique for automated software debugging. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys'10)*. ACM, New York, NY, 321–334.

ZHANG, X., TALLAM, S., AND GUPTA, R. 2006. Dynamic slicing long running programs through execution fast forwarding. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT'06/FSE-14)*. ACM, New York, NY, 81–91.

ZHAO, Q., RABBAH, R., AMARASINGHE, S., RUDOLPH, L., AND WONG, W.-F. 2008. How to do a million watchpoints: Efficient debugging using dynamic instrumentation. In *Proceedings of the International Conference on Compiler Construction*.

Received July 2011; accepted October 2011