Improving Software Diagnosability via Log Enhancement

Ding Yuan‡† Jing Zheng† Soyeon Park† Yuanyuan Zhou† Stefan Savage†

†University of California, San Diego, ‡University of Illinois at Urbana-Champaign

{diyuan,j3zheng,soyeon,yyzhou,savage}@cs.ucsd.edu

Abstract

Diagnosing software failures in the field is notoriously difficult, in part due to the fundamental complexity of trouble-shooting *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 during diagnosis while imposing negligible overheads.

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]: Diagnostics

General Terms Reliability

Keywords Log, Software Diagnosability, Static Analysis

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 or at run-time [9, 14, 38, 48], 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.

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 [13, 20, 28, 33, 40, 44, 49, 50, 52, 55]. 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" [18]). 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 [1, 2, 18, 47].

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 [3, 8,

[1] Some systems, such as Windows Error Reporting [26] and Mozilla's Quality Feedback Agent [41] 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.

ASPLOS'11 March 5–11, 2011, Newport Beach, California, USA.

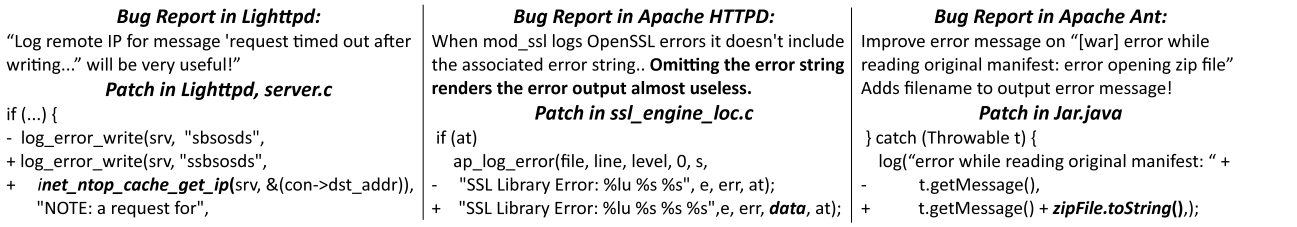


Figure 1. Example of real-world patches just for the purpose of enhancing log messages.

16, 29, 56], 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 [57] 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 the 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 that is automatically collected 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 [51] and the thread ID with each message [57]), 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. For example, there are more than 900 different error log messages in apache httpd, capturing various failure types, and over its five year history, we have identified 5,409 "enhancements" in the form of patches to these messages to improve their fidelity. 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 paper, 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 nor failures itself. Rather it is a tool for reducing the burden of failure diagnosis by enhancing the information that programmers should have captured when writing log messages. Such additional log information can significantly narrow down the number 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 do not try 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 collecting 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 *historical* 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 we can perform 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 the above 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 possible partial execution paths and run-time 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 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 in the target software prior to release, oblivious to what failure might occur in production.

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

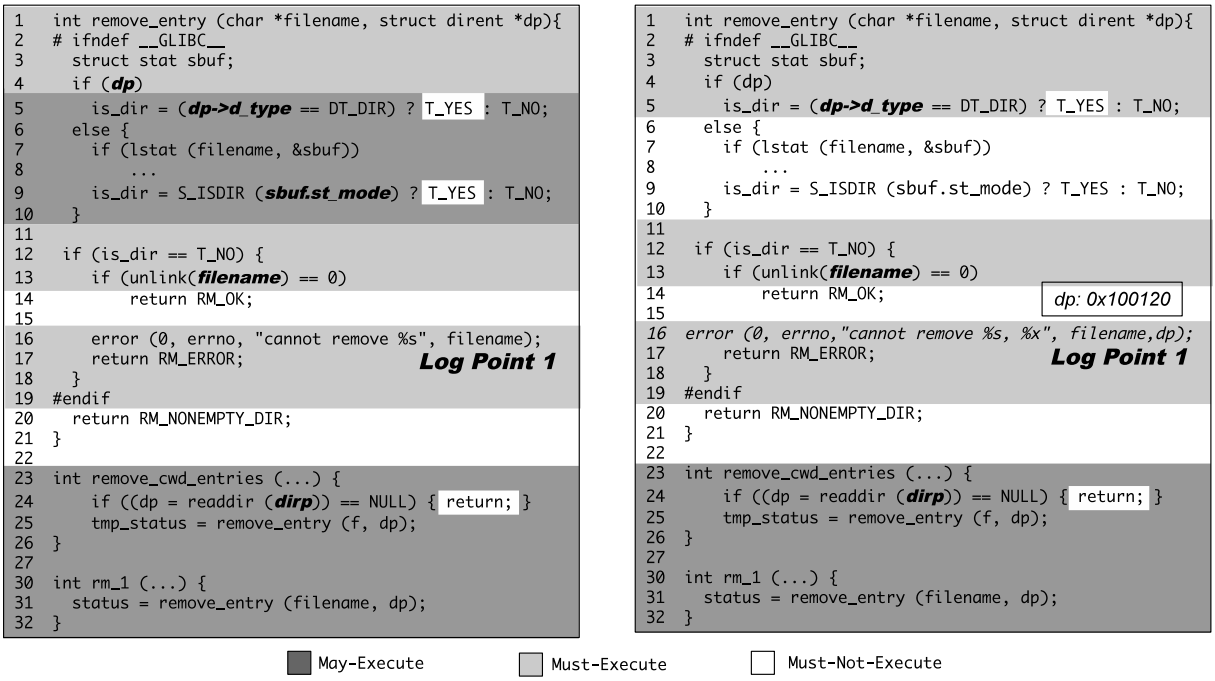


Figure 2. Highly simplified code for `rm` in coreutils-4.5.4. Different colors highlight which information can be inferred given the log message was printed on line 16. For example, "Must-Execute" reflects code paths that can be completely inferred based on the given log message. Variable values that cannot be inferred are also highlighted.

2. Overview

To explain how *LogEnhancer* works, we first examine how diagnosis is performed manually today. Figure 2(A) 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
```

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(A) 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(A) 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.

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. 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 data 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(A)). 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 how this piece of new

information would help in Figure 2(B)). In this case dp is non-zero, and thus the code at line 5 is now in a "must-execute" path, while lines 6–10 "must not" have executed. In turn, it removes the need to consider the value of sbuf.st_mode since is_dir now only depends 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 frames, and dp->d_type (or, some *equivalent value* that can be used to infer dp->d_type's value at line 5; we will discuss how to find an equivalent value in Section 3.2).

The goal of *LogEnhancer* is to automate exactly the kind of analysis we described above — identifying *causally-related* variable values for each "log point" and enhancing the log messages to incorporate these values. Moreover, because it is automatic, *LogEnhancer* can be applied comprehensively through out the entire 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. It can be used to enhance *every existing* log printing statement in the target software's source code or to enhance any *newly inserted* log printing statement. 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(), so simply issuing

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

is sufficient for *LogEnhancer* to do its work.

Once invoked, *LogEnhancer* leverages the standard make process to compile all program source code into the CIL intermediate language [45], 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 at the log points and then re-compiles the modified source to generate a new binary.

During production-runs, when an 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 added: *(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 enhanced log from production run can be manually examined by developers at 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 main 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 the program execute 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 starting from is_dir). This process is repeated recursively for each potential caller (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 sub-steps: (i) Identify values that are certain from the constraint (i.e., the conditions for the log message to be printed), and prune them out using a SAT solver; (ii) Parse the uncertain values into a *conditional value* format, e.g., [dp]:dp->d_type, indicating the value dp->d_type is only meaningful under condition dp!=NULL; (iii) Identify the values that would be overwritten before the target log point; (iv) Find equivalent values that can be used to infer those overwritten key values; (v) From the uncertain value set, find the minimum set by eliminating redundant values that can be inferred by remaining uncertain values. Finally, *LogEnhancer* builds an *Uncertain Value Table* to identify the selected variable values to be recorded for each log point.

(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. 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* does 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 below.

(1) How far and deep can LogEnhancer go in analysis? To avoid path explosion, *LogEnhancer* places a number of limits on how far its analysis is applied within a program. Given the problem of inferring causally-related information, our design focuses on analyzing only the functions that *must* have a causal relationship with the log point (i.e., functions that are on the call-stack or whose 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 variables with most diagnostic value are commonly on the execution path to a

log point and such variables 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 effects of removing redundant values).

(3) What about privacy concerns? Just as with existing log messages, the information we record focuses narrowly on the system's own "health". Because we are only recording a small number of variable values per message, it is much easier, compared with core dumps, for users to to check that no private information is revealed. It is also easier to combine our system with automatic privacy information filtering techniques (e.g., [12] that 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 [57] 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 limitations of 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 a 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 particular values. The sub-steps (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 sub-steps and conservatively treat any shared data as overwritten ones at the log point.

Note this limitation does not mean 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 mis-configurations, not by concurrency bugs [36].

Also, our objective is to collect more diagnostic information, not to to pinpoint the exact root cause (although it would be extremely nice, it is too ideal to be realistic). So even for concurrency bugs, the causally-related key variable values from intra-thread/process analysis is still useful to reduce the diagnostic space.

Additionally, unlike static analysis for bug detection, inaccurate data flow analysis does *not* introduce false bug reports since all the recorded values are from the production-run execution. The only consequence is that some logged variable values may be less useful for diagnosis or that we are still missing some causally-related variable values. However, the recorded variable values are still valid, just as if programmers manually added those variables into logging statements.

Addressing these issues would require more complicated thread-aware code analysis. For each variable that is causally-related to the log message, in addition to analyzing the intra-thread or intra-process data flow, we would also need to analyze any inter-thread or inter-process modifications. Although theoretically we can still use the same Uncertainty Identification algorithm to recursively follow intra-thread/process *and* inter-thread/process data-flow, we imagine practical scalability and precision issues might arise. Given an uncertain variable value `V` in function `F`, any modifications to `V` that might be executed concurrently with `F` need to be considered. Without precise information about which functions might be executed concurrently and with the complications caused by pointer aliasing, we might end up analyzing many data-flows that are not causally related to the log point at all. This might add significant overhead to our analysis, and more importantly, end up recording a huge number of irrelevant variables. Annotations can be used in expressing which functions are concurrent [19, 24], while techniques presented in RacerX can help to automatically infer this information [23]. Previous work [42, 43] also show that 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. Leveraging these techniques to handle inter-thread/process data-flows remains as our future work.

(5) What if there is no log message? In this work, we are trying to improve the world as it is. As mentioned in our introduction, most commercial and open source software already contains significant number of logging statements as logging has become a standard practice. Hence we focus on enhancing existing log messages, and *assume* that such log messages exist. 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.

3. Design and Implementation

LogEnhancer's source code analysis is implemented using the Saturn static analysis framework [5]. Saturn models C programs precisely and allows user to express the analysis algorithm in a logic programming language. It is summary-based, meaning it conducts its analysis for each function separately and then generates a summary for each function. At the calling sites of a function, the summary is used instead of going deep into the function. Saturn also provides a SAT solver.

In this section we will not repeat the all the details of Saturn. Except for the Data-flow analysis described in Section 3.1, all the analysis processes, design and implementation issues are unique to *LogEnhancer* and solved by us.

3.1 Uncertainty Identification

For each log message in the target software, the goal of Uncertainty Identification is to identify uncertain control or data flows that are causally-related to this 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 instruction 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 a *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, a summary of f' is generated which captures the causally-related constraint within f' to eventually reach LP.

Data-flow analysis and memory model: *LogEnhancer* directly uses Saturn's memory model for data-flow analysis. Saturn models every memory location accessed by function f at every program point P in the form of a constrained expression (CE). A CE is represented in the format of `V=E:C`, indicating the value of `V` equals the expression `E` under condition `C`. At the beginning of each function f, Saturn first statically enumerates all the memory locations (heap and stack) accessed by f, and initializes each location `V` as `V=V:True`, indicating the value of `V` is unknown (symbolic). This is possible because we model the loops as tail-recursive functions, thus each function body is loop-free (see Handling Loops later this section). At an assignment instruction P, `V=exp;`, the value of `V` is 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 3 shows the CE of `is_dir` in `rm` at log point 1.

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

Figure 3. 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.

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 [4]. Thus we can represent all memory locations accessed by f with a 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 3.

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, therefore, we can prune out such a constraint. The satisfiable constraint thus contains all the causally-related control and data-flow conditions to reach LP. This constraint C will be stored as a part of this function's summary, along with the location of LP. It 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. `longjmp`s are correlated with `setjmp`s through function summaries in a manner similar to that described in [57].

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 3. 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` to reach log point 1:

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

Inter-Procedural analysis: The above process is then recursively repeated in the caller by traversing the call-graph in bottom-up order. In `rm`, after analyzing `remove_entry`, *LogEnhancer* next analyzes its caller `remove_cwd_entries` in the same manner: a linear scan to compute the CE for each memory location and control-flow constraint for each program point. At line 25, it finds a call-site to a function with a summary (`remove_entry`), indicating that reaching this point might eventually lead to log point 1, so it 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 produce the following:

$$C'_r \texttt{ = (readdir(dirp) \&\& readdir(dirp)->d\_type!=DT\_DIR || !readdir(dirp)) \&\& f==Sym}$$

Here, `readdir(dirp)` is the substitution for `dp` in C_r; `S_ISDIR (sbuf.st_mode)` is pruned since 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 from the log point. It ignores functions that are not in the call chains for the log point—we refer them as "sibling functions". Sibling functions may also be causally-related to the log point. Therefore, if a sibling function's return value appears in the constraint for the log point, *LogEnhancer* also analyzes the 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 descend only one level into such functions due to scalability concerns. 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.

Handling Loops Loops are modeled as tail-recursive functions so that 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. Each loop is handled similarly to ordinary functions except that it is traversed *twice*, to explore both loop entering and exiting directions. A variable `V` modified within the loop body is propagated to its caller as `V==Sym` to relax the value constraint, since we are not following the multiple iterations as in run-time. In this way, constraint from the loop body can be conservatively captured.

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

Pointer Aliasing: Intra-procedural pointer aliasing is precisely modeled by Saturn's constrained expression model [5]. Inter-procedural pointer aliasing analysis is only performed on function pointers to ensure that *LogEnhancer* can traverse deep along the call-chain. The other types of pointers are assumed to be non-aliased, which might cause us to miss some causally-related variable values. Note that for Value Selection we enable inter-procedural alias analysis for all types of pointers for *conservative* liveness checking.

3.2 Value Selection

Value Selection selects, from all constraints identified by the previous step, what key variable values to record at each log point. In this section, we refer an expression without any Boolean operator (i.e., `&&, ||, !`) as a *uni-condition*. For example, "`dp!=NULL`" is a uni-condition (note `!=` is not one of the three Boolean operators). A constraint is thus a formula of uni-conditions combined together using Boolean operators.

(1) Pruning Determinable Values: Some variable values can be inferred knowing that a given log point is executed. We call them *determinable values*. For example, in constraint `a==0 && b!=0`, it

can be determined that "a" must equal zero, while b's value is still uncertain. A determinable value V is identified if: (i) V is involved in a *necessary* uni-condition uc of constraint C, i.e., $\neg uc \wedge C$ is unsatisfiable; (ii) uc is in the form of V==CONSTANT. A determinable value can be pruned out since it need not be recorded.

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

(3) Liveness Checking and Equivalent Value Identification (EVI): A value can also be *dead* (overwritten or gone altogether along 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, i.e., if a 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 [30] 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 the dead value. More specifically, a value EV is *equivalent* to another value V if and only if: (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.

3.3 Run-time Value Collection

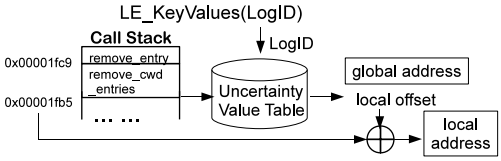


Figure 4. Run-time recording for Delayed Collection.

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. The addresses of these variables are obtained from the compiled binary by parsing the DWARF debugging information [22]. Local variables' addresses are the offsets from the stack frame base pointer. Heap values' addresses are represented the same way as they are in the original code. Each live value is represented by its address and the condition for it to be meaningful is stored into an Uncertain Value Table (UVT) that corresponds to the log point. At the end of our analysis, each UVT is output to a separate file. These files are released together with the target application.

Figure 4 shows the run-time process of LE_KeyValues(). It is triggered only at the log point, i.e., when a log message is being printed. When triggered, it first uses the LogID of the log point to load the corresponding UVT into 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 adding this offset to the dynamic stack frame base pointer obtained by walking the stack. Note that the UVT is only loaded into memory during the execution of LE_KeyValues(), so the delayed recording does not add any overhead during normal execution, i.e., 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 which also intercept SIGSEGV signals, we add a wrapper to filter out those caused by our log recording. In our experiments, we have never encountered such a signal.

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 the 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.

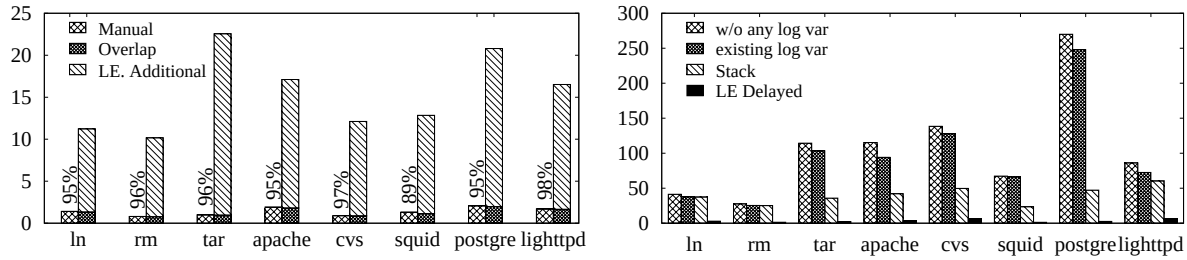
All recorded values from LE_KeyValues() and LE_InTime() are first stored into buffers in memory (both currently 40 KB) respectively. At *error* messages, both buffers are flushed to disk. LE_KeyValues()'s buffer is also flushed when it becomes full, whereas LE_InTime() simply recycles the shadow buffer from the beginning. Each thread has its own private buffer.

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 equivalent values) from a core dump at a log point (if there is such core dump). When a core dump is generated, our Core Dump Digger derives equivalent information to delayed collection from the core dump. Note that not every log point has a core dump, especially those for book-keeping or warning messages.

4. Evaluation

We use *LogEnhancer* to enhance *all 9,125* log messages in 8 different real-world applications as shown in Table 1. Five of them are server applications, including 2 web servers (apache httpd, lighttpd), a database server (postgresql), a concurrent version control server (cvs), and a web cache (squid). For server applications where there are multiple log files, 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 verbosity mode, all applications only print error messages (some of them also prints warning messages). Therefore, during normal execution with the default verbosity mode, there are few log message printed besides a few messages indicating system start/stop.

For any diagnostic tools like *LogEnhancer*, the most effective evaluation method is of course user study: having it used by real programmers for a period of time who then report their experi-



(A) Variables Recording at each log point (B) Number of Uncertain Branches

Figure 5. Overall Result of *LogEnhancer.* In (A), 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.* In (B), 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.

Application	Version	LOC	Log Points	
			All	Default
ln	4.5.1	20K	26	14 (ERR)
rm	4.5.4	18K	28	25 (ERR)
tar	1.22	86K	210	176 (ERR)
apache	2.2.2	317K	1,654	1,093 (WARN)
cvs	1.11.23	148K	1,088	762 (ERR)
squid	2.3.S4	69K	1,116	402 (ERR)
postgresql	8.4.1	1,029K	4,876	4,403 (WARN)
lighttpd	1.4.26	56K	127	127 (ERR)

Table 1. Evaluated applications. LOC is lines of code of the entire application. "All" shows the total number of log points for the most verbose level. "Default" shows the default verbose-level of log message printed (in bracket) and the number of log points at this level.

ences. Unfortunately, this would be a time-consuming process and 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, all conducted on a Linux machine with eight 2.33GHz Xeon processors and 16GB of memory. Since the analysis is done off-line, *LogEnhancer* currently runs as single process, single thread (even though the analysis can potentially be parallelized to reduce the analysis time).

(1) Value selection. First, we investigate how well our algorithm captures the variable whose values are useful for failure diagnosis by comparing against manual selection (variable values 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 manually added by programmers over time (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 misconfigurations to show the usefulness 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 for both in-time and delayed collection methods.

4.1 Effectiveness in Variable Recording

Figure 5 (A) 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 are 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 well as manual effort.

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

There are four main categories of manually-identified variables that are missed by *LogEnhancer*, together contributing to 97% of the few missed cases. *(1) Book-keeping values logged immediately after initialization (37%).* For example, 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. *(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 multi-variables (8%)* that are always updated together while only one used in branch. *LogEnhancer* only identifies the one used in branch and the missed values can be inferred from the identified one.

In addition to automatically selecting 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 5 (B). From the 108 original uncertain branches per log point, existing log variables can reduce it to 97, whereas *LogEnhancer*'s delayed recording scheme can reduce this number to 3, meaning that, on average for each log point, there are only 3 unresolved branches for programmers to consider to fully understand why the log point was reached. The remaining uncertain branches are caused by uncertain values that are dead at log points,

[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-text value we can *deterministically* infer the text string.

Apps.	uncert. br. (avg/med/max/min) w/o any var	L.E. (delay)	# of Var all	live	logged
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

Table 2. Detailed result showing the number of uncertain branches and uncertain variable values per log point. The large difference between average and median in "w/o any var" is caused by small number of log points inside some library functions, that have a large number of uncertain branches accumulated from many possible call stacks. Once we differentiate call stacks in "Stack" approach, this difference between average and median significantly reduces.

which can 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 is reduced from 97 to 40 on average. Table 2 shows the detailed number of uncertain branches.

Table 2 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 these can be recorded at log points ("logged") without introducing normal-run overhead. Among these 14.6 variables, 12.9 are not overwritten before log point (i.e., they are "live"), and the remaining 1.7 are recovered from Equivalent Value Identification (EVI). On average 49% of the dead values can be recovered by EVI. The remaining 51% of dead values can be collected only via in-time collection, at the cost of some overhead to normal execution.

Analysis Time Table 3 shows the analysis time of *LogEnhancer* on each application. For all applications except postgresql, *LogEnhancer* finishes the entire analysis within 2 minutes to 4 hours. For postgresql, it takes 11 hours since there are 4,876 logging points in 1M lines of code. Since we expect *LogEnhancer* to be used off-line prior to software release, the analysis time is less critical. Additionally, the summary-based design allows it to be parallel or incrementally applied [5]. The memory usage in all cases is below 2.3GB.

Analysis Time and Memory Usage								
ln	3m	579MB	rm	2m	172MB	tar	1.5h	263MB
apache	2.1h	1.3GB	cvs	3.0h	1.7GB	squid	3.8h	2.3GB
postgre	10.7h	1.5GB	lighttpd	19.5m	532M			

Table 3. Analysis performance.

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 entirely eliminated. Due to space limitations, in this section we will show 3 cases in detail to demonstrate the effectiveness of *LogEnhancer*. The other 12 cases are summarized in Table 4.

We also compared the inference results of SherLog [57] before and after *LogEnhancer*'s enhancement.

Fail.	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 non-existing tar-ball.
lighttpd	Proxy fails when connecting to multiple backends.

Table 4. Real-world failures evaluated.

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 infer that the failed execution 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 a type for `dp->d_type`. In this case, just as if `dp` is NULL, the program should use `lstat` to determine the directory type. So the fix is straightforward as shown below:

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

Without *LogEnhancer*'s enhancement, SherLog inferred a total of 13 possible call paths (not even complete execution paths, only function call sequences) that might have been taken to print 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 6 shows a bug report in apache. With only the error log message at line 2, 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 only two pieces of information are key to identifying the root cause: One is the value of `keepalives` and the other is the request type, `proxyreq`, both of which are unfortunately buried deep in large and mostly irrelevant data structures.

LogEnhancer automatically identifies `c -> keepalive` and `r-> proxyreq` to be collected for this log message. `keepalive` is identified since it is used at line 12 as the constraint for the execution to reach the log point. `proxyreq` is identified in similar manner. So if the developers had used *LogEnhancer* to enhance their log messages automatically, *LogEnhancer* would have helped them by saving a lot of time in discussions 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 `keepalives` in this function.

Without *LogEnhancer*'s enhancement, SherLog inferred 63 possible call paths and not be able to infer the value of `keepalive` or `proxyreq`. With *LogEnhancer*'s enhancement, SherLog can narrow down to only one possible call path, and infer the value of `keepalive` and `proxyreq`.

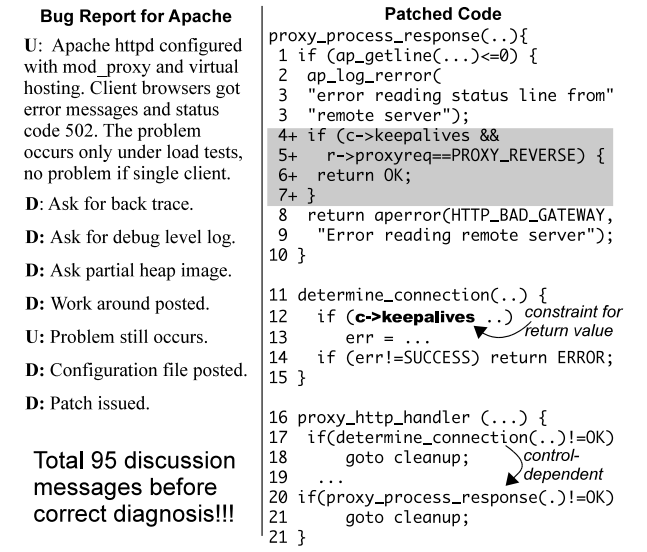


Figure 6. Apache bug example. "U" stands for apache user while "D" for developer. Patched code is highlighted.

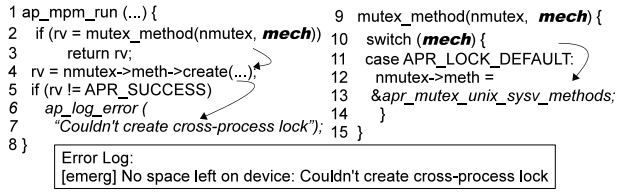


Figure 7. Apache configuration error. The dependencies to identify variable `mech` are marked as arrows.

Case 3: Apache configuration error.

A misconfiguration failure in Apache occurs with the log message shown in Figure 7. It warns of no space on disk, while users' file system and disk are 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 identifies `mech` as a key value to collect at line 6, since it is used at line 10 in function `mutex_method`, whose `nmutex` is causally related to the log point at line 6. 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 multi-threaded mode, apache should use fnctl-based locking 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 5 shows the *LogEnhancer*'s recording overhead during applications' normal execution under the default verbosity mode. Few log messages are printed in the default verbosity 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%.

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

Table 5. Overhead of *LogEnhancer* in normal execution (default verbosity mode). The first number is the overhead for the delayed collection, and the second is for the in-time collection. `rm` and `ln`'s results are not reported since the execution times are too short. `Tar` is measured in response time, while the servers are measured in throughput degradation when fully loaded.

Figure 8 shows *LogEnhancer*'s performance during normal execution with other verbosity modes. Turning on debug level logging degrades the throughput by 13% even *without LogEnhancer*. With *LogEnhancer*, there is an additional 3-7% overhead (measured in throughput degradation) on top of the original.

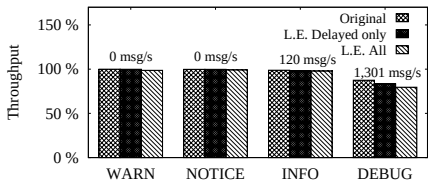


Figure 8. Normal execution throughput on fully loaded Apache for different verbosity modes. All numbers are normalized over the throughput of unmodified apache under default verbosity level (WARN).

Memory Overhead: As mentioned in Section 3.3, delayed collection does not introduce any memory overhead during normal execution (i.e., no log message printed). For in-time collection, the only memory overhead is the size of the 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 KB respectively.

Comparison with Core Dump Table 6 compares *LogEnhancer*'s recording time and data size with core dump at a failure. On average, *LogEnhancer* only needs 0.43 millisecond. The recorded data has only 66 bytes on average. In comparison, core dumps require 1000 times recording time, and 55MB in size. The large overhead of core dumps makes it impractical to collect the entire memory image with each log message.

Table 6 shows *LogEnhancer*'s average log size is 66 bytes per message, which is in the same magnitude as original log message.

Failure	**Time** and **Size (B)**			
	LogEnhancer		Core dump	
ln	0.45ms	41 (original 45)	630ms	55M
rm	0.45ms	113 (original 51)	610ms	55M
tar 2	0.39ms	39 (original 96)	630ms	55M
cvs	0.44ms	54 (original 52)	60ms	772K
apache 1	0.41ms	82 (original 196)	670ms	3.2M

Table 6. Comparison between *LogEnhancer* and core dump. We reproduced 5 failures in table 4 and forced a core dump to be generated at each log point using `gcore` [25] library call. The log size of *LogEnhancer* does not include the size of the original log. The size of original log (without *LogEnhancer*) is shown in the parenthesis.

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 remain the same for a log point. Other variable values in our log are encoded in binary format and are converted to human readable form post-mortemly.

5. Related Work

Log analysis for failure diagnosis Existing log analysis work focuses on post-mortem diagnosis using logs, learning statistical signatures [3, 8, 16, 29, 56] or inferring partial execution paths and run-time states [57]. Xu et al. [56] use statistical techniques to efficiently learn a decision tree based signature from large number of console logs. This signature can be used to effectively detect and diagnose anomalies.

In particular, the closest related work, SherLog [57], uses static analysis to infer the partial execution paths that can connect the run-time log messages. It infers both control and data value information post-mortemly, providing a similar user-experience as an interactive debugger without dynamically re-executing the program.

LogEnhancer is different from but complementary to log analysis work like SherLog [57] 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 is already available in 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 instrument the source code to log those selected variables at run-time. None of this would be needed in a log inference engine like SherLog.

(4) As the real world case studies in Section 4.2 have shown, automatic log inference engines like SherLog can significantly benefit from *LogEnhancer*'s log enhancement information. For example, in our second case study, the additional information added by *LogEnhancer* can help SherLog pinpoint the execution path from a total of 63 possibilities before the enhancement.

Logging design Existing guidelines for logging design are purely empirical [32, 51]. Kernighan and Pike [32] argued the importance of well-designed log messages in failure diagnosis. Schmidt summarized some empirical logging practices [51]. To the best of our knowledge, *LogEnhancer* is one of the first to automatically enhance log messages.

Use of core dump for failure diagnosis Several systems collect partial memory image [6, 26, 27, 46] when a system crashes. Windows Error Reporting [26] monitors the system for crashes or hangs, and records a "mini-dump". Crash Reporter [6], NetApp Savecore [46] and Google Breakpad [27] also collect compressed memory dumps.

Some core dump analyzers infer diagnostic information from the core dump. Their techniques are applicable on *LogEnhancer*'s recording result as well. PSE [39] and ESD [58] perform off-line diagnosis of program crashes from core dump. Weeratunge, et al. [54] diagnose Heisenbugs by diff-ing the core dumps from a failing run and passing run.

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.

Profiling for diagnosis Many of diagnostic tools collect run-time profiling such as low-level performance counters [10, 16] or execution traces [7, 13, 15, 29, 37, 52, 60]. Liblit et al. [37] sample profiling information from many users to offload the monitoring overhead, and isolate the most correlated information using statistical techniques. Chen et al. [13] 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.

DCop [59] records the acquisition of each lock involved in a deadlock to speed up the debugging process of deadlock failures. Our work is also complementary to DCop in that we can help the diagnosis of other kinds of failures that print log messages.

Logging for deterministic replay Other work [20, 34, 35, 40, 44, 49, 53, 55] attempts to deterministically replay failed execution, which generally requires high run-time logging overhead especially for multiprocessor systems. To reduce the overhead, recently SMP-Revirt [21] made clever use of page protection. Our work is complementary and mainly targets to the cases when failure reproduction is difficult due to privacy concerns, unavailability of execution environments, etc.

Other Static Analysis Work Compiler techniques similar to *LogEnhancer* are also used to address some other software reliability problems [11, 17, 31, 58]. KLEE [11] and ESD [58] use full symbolic execution engine to expose bugs in testing or infer paths from core dump. Carburizer [31] uses data-flow analysis to locate dependencies on data read from hardware. 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, run-time value collection, etc.

6. Conclusions

In this paper 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% 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. This information not only benefits manual diagnosis but also automatic log inference engines.

Acknowledgments

We thank the anonymous reviewers and our paper shepherd Todd C. Mowry for their insightful feedback, the UCSD Opera research group and Michael Vrable for useful discussions and paper proofreading. 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.

References

[1] Cisco system log management.

[2] EMC seen collecting and managing log as key driver for 94 percent of customers.

[3] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP'03*.

[4] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition), Page 528.*

[5] A. Aiken, S. Bugrara, I. Dillig, T. Dillig, P. Hawkins, and B. Hackett. An overview of the Saturn project. In *PASTE'07*.

[6] Apple Inc., CrashReport. Technical Report TN2123, 2004.

[7] A. Ayers, R. Schooler, C. Metcalf, A. Agarwal, J. Rhee, and E. Witchel. Traceback: First fault diagnosis by reconstruction of distributed control flow. In *PLDI'05*.

[8] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *OSDI'04*.

[9] E. D. Berger and B. G. Zorn. Diehard: probabilistic memory safety for unsafe languages. In *PLDI'06*.

[10] S. Bhatia, A. Kumar, M. Fiuczynski, and L. Peterson. Lightweight, high-resolution monitoring for troubleshooting production systems. In *OSDI'08*.

[11] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI'08*.

[12] M. Castro, M. Costa, and J.-P. Martin. Better bug reporting with better privacy. In *ASPLOS'08*.

[13] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos. Flexible hardware acceleration for instruction-grain program monitoring. In *ISCA'08*.

[14] L. Chew and D. Lie. Kivati: fast detection and prevention of atomicity violations. In *EuroSys'10*.

[15] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani. HOLMES: Effective statistical debugging via efficient path profiling. In *ICSE'09*.

[16] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, indexing, clustering, and retrieving system history. In *SOSP'05*.

[17] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: securing software by blocking bad input. In *SOSP'07*.

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

[19] D. L. Detlefs, K. R. M. Leino, K. Rustan, M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. In *TR SRC-159, COMPAQ SRC*, 1998.

[20] J. Devietti, B. Lucia, M. Oskin, and L. Ceze. Dmp: Deterministic shared-memory multiprocessing. In *ASPLOS'09*.

[21] G. Dunlap, D. Lucchetti, M. Fetterman, and P. Chen. Execution replay of multiprocessor virtual machines. In *VEE*, 2008.

[22] The DWARF Debugging Format. `http://dwarfstd.org`.

[23] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *SOSP'03*.

[24] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *PLDI'02*.

[25] Man page for gcore (Linux section 1).

[26] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: ten years of implementation and experience. In *SOSP '09*.

[27] Google Inc., Breakpad.

[28] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: An application-level kernel for record and replay. In *OSDI'08*.

[29] J. Ha, C. J. Rossbach, J. V. Davis, I. Roy, H. E. Ramadan, D. E. Porter, D. L. Chen, and E. Witchel. Improved error reporting for software that uses black-box components. In *PLDI'07*.

[30] B. Hackett and A. Aiken. How is aliasing used in systems software? In *FSE'06*.

[31] A. Kadav, M. J. Renzelmann, and M. M. Swift. Tolerating hardware device failures in software. In *SOSP'09*.

[32] B. W. Kernighan and R. Pike. *The Practice of Programming*. Addison-Wesley, 1999.

[33] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX ATC'05*.

[34] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. Comput.*, 36(4), 1987.

[35] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. Respec: efficient online multiprocessor replay via speculation and external determinism. In *ASPLOS'10*.

[36] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Have things changed now? An empirical study of bug characteristics in modern open source software. In *ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, October 2006.

[37] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI'03*.

[38] B. Lucia, L. Ceze, and K. Strauss. Colorsafe: architectural support for debugging and dynamically avoiding multi-variable atomicity violations. In *ISCA'10*.

[39] R. Manevich, M. Sridharan, S. Adams, M. Das, and Z. Yang. PSE: Explaining program failures via postmortem static analysis. *SIGSOFT Softw. Eng. Notes*, 29(6):63–72, 2004.

[40] P. Montesinos, L. Ceze, and J. Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *ISCA'08*.

[41] Mozilla Quality Feedback Agent. `http://support.mozilla.com/en-US/kb/quality+feedback+agent`.

[42] M. Naik and A. Aiken. Conditional must not aliasing for static race detection. In *POPL'07*.

[43] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. In *PLDI'06*.

[44] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *ISCA'05*.

[45] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *CC'02*.

[46] NetApp Inc., Savecore. ONTAP 7.3 Manual Page Reference, Volume 1, Pages 471-472.

[47] NetApp. Proactive health management with auto-support. *NetApp White Paper*, 2007.

[48] G. Novark, E. D. Berger, and B. G. Zorn. Exterminator: automatically correcting memory errors with high probability. In *PLDI'07*.

[49] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient Deterministic Multithreading in software. In *ASPLOS'09*.

[50] S. Park, W. Xiong, Z. Yin, R. Kaushik, K. H.Lee, S. Lu, and Y. Zhou. Pres:probabilistic replay with execution sketching on multiprocessors. In *SOSP*, 2009.

[51] S. Schmidt. 7 more good tips on logging. `http://codemonkeyism.com/7-more-good-tips-on-logging/`.

[52] E. Vlachos, M. L. Goodstein, M. A. Kozuch, S. Chen, B. Falsafi, P. B. Gibbons, and T. C. Mowry. Paralog: enabling and accelerating online parallel monitoring of multithreaded applications. In *ASPLOS'10*.

[53] VMware. Using the integrated virtual debugger for visual studio. http://www.vmware.com/pdf/ws65_manual.pdf.

[54] D. Weeratunge, X. Zhang, and S. Jagannathan. Analyzing multicore dumps to facilitate concurrency bug reproduction. *SIGARCH Comput. Archit. News*, 38(1):155–166, 2010.

[55] M. Xu, R. Bodik, and M. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *ISCA'03*.

[56] W. Xu, L. Huang, M. Jordan, D. Patterson, and A. Fox. Mining console logs for large-scale system problem detection. In *SOSP'09*.

[57] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. SherLog: Error diagnosis by connecting clues from run-time logs. In *ASPLOS'10*.

[58] C. Zamfir and G. Candea. Execution synthesis: a technique for automated software debugging. In *EuroSys'10*.

[59] C. Zamfir and G. Candea. Low-overhead bug fingerprinting for fast debugging. In *Runtime Verification*, volume 6418 of *Lecture Notes in Computer Science*, pages 460–468. 2010.

[60] Q. Zhao, R. Rabbah, S. Amarasinghe, L. Rudolph, and W.-F. Wong. How to do a million watchpoints: efficient debugging using dynamic instrumentation. In *CC'08*.