SherLog: Error Diagnosis by Connecting Clues from Run-time Logs

Ding Yuan
University of Illinois at Urbana-Champaign
dyuan3@cs.uiuc.edu

Haohui Mai
University of Illinois at Urbana-Champaign
mai4@cs.uiuc.edu

Weiwei Xiong
University of Illinois at Urbana-Champaign
wxiong2@cs.uiuc.edu

Lin Tan*
University of Waterloo
lintan@uwaterloo.ca

Yuanyuan Zhou
University of California, San Diego
yyzhou@cs.ucsd.edu

Shankar Pasupathy
NetApp, Inc
Shankar.Pasupathy@netapp.com

Abstract

Computer systems often fail due to many factors such as software bugs or administrator errors. Diagnosing such production run failures is an important but challenging task since it is difficult to reproduce them in house due to various reasons: (1) unavailability of users' inputs and file content due to privacy concerns; (2) difficulty in building the exact same execution environment; and (3) non-determinism of concurrent executions on multi-processors.

Therefore, programmers often have to diagnose a production run failure based on logs collected back from customers and the corresponding source code. Such diagnosis requires expert knowledge and is also too time-consuming, tedious to narrow down root causes. To address this problem, we propose a tool, called SherLog, that analyzes source code by leveraging information provided by run-time logs to infer what must or may have happened during the failed production run. It requires neither re-execution of the program nor knowledge on the log's semantics. It infers both control and data value information regarding to the failed execution.

We evaluate SherLog with 8 representative *real world* software failures (6 software bugs and 2 configuration errors) from 7 applications including 3 servers. Information inferred by SherLog are very useful for programmers to diagnose these evaluated failures. Our results also show that SherLog can analyze large server applications such as Apache with thousands of logging messages within only 40 minutes.

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

General Terms Reliability

Keywords Log, Failure Diagnostics, Static Analysis

* This work was done when she was at UIUC as a graduate student. Currently she is an assistant professor at the University of Waterloo.

ASPLOS'10, March 13–17, 2010, Pittsburgh, Pennsylvania, USA.

1. Introduction

1.1 Motivation

Many applications require high reliability and availability [23]. Unfortunately, software failure is a major contributor to system down time and security holes. As software systems have grown in size, complexity and cost, it has become increasingly difficult to deliver bug-free software to end users. As a result, many software failures (including crashes, hangs, incorrect results and other software anomalies) occur during production runs.

Besides software defects, administrator errors are another major cause for system failures because, being human, administrators usually make mistakes when performing configuration related tasks. A recent study [29] has shown a significant fraction (48.6%) of failures in enterprise networks are caused by mis-configurations.

When a system fails in production run, regardless of the root cause (software bugs, mis-configurations or even hardware faults), support engineers and programmers are frequently called upon to diagnose and solve the issue within a tight time schedule. Since such errors directly impact customers' business, vendors make diagnosing and fixing them as the highest priority. In order to provide timely solutions to users, vendors often devote extensive time and human resources, which often results in high supporting cost to solve end user's problems. In addition, it also causes interruption on on-going effort to develop new features or products.

To solve a product-run issue, support engineers[1] first need to understand what have happened during the failure run in order to narrow down the root cause. The best solution to achieve this is of course to reproduce the failure in house. Much effort has been conducted in system and hardware support for reproducing software failures on uni-processor and multi-processors, including TTVM [30], R2 [24], DMP [17], Kendo [38], FDR [46], BugNet [36], VMWare [43] , just to name a few.

Unfortunately, despite the above applaudable effort in failure reproduction, in reality, many circumstances make such failure reproduction impossible or forbiddingly expensive. First, customers' privacy concerns can make failure reproduction infeasible. For example, financial companies will not be able to release their databases to vendors to troubleshoot a problem. Even an ordinary desktop user may feel uncomfortable to send back Microsoft their inputs

[1] Note that most tier-2 or tier-3 support engineers are software engineers who have designed and developed the released software.

typed into a browser before a crash. Second, it is hard to have the exact same execution environment (including hardware, network, third-party application layers, OS or library versions, etc.) as what customers have due to resource or license constraints. Third, how to provide a low-overhead logging mechanism for failure reproduction on multi-processors is still a challenging open research problem [19, 35, 46].

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

1.2 A Motivating Example

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

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

The log produced by `rmdir` contains the following text (we number each message for the convenience of our description):

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

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

While the root cause may sound simple once known, it was a "mystery" the first time this failure is reported and being diagnosed. The only information available is just the log messages. By examining the logs and the corresponding source code, an engineer may find that code statements $r1, m1$ could generate the first 2 messages, and $r2, m2$ generate the third message. However, simple combinatorics would indicate a total of 8 combination possibilities, and different combinations would take engineers to different paths for narrowing down the root cause.

However, by conducting a deeper analysis, we can find out that six of the eight paths are infeasible because they are self-contradictory; and only two paths, namely $\{m1, r1, r2\}$ or $\{m1, m1, m2\}$ are remaining for the next step of follow-up. For example, $\{r1, r1, r2\}$ would not be feasible because it implies $m1$ not being executed, which requires `verbose` not being set, contradictory with the constraint for $r1$ to be executed.

In addition, to diagnose a failure, engineers often need to know more than just log-printing code locations. Specifically, they may need to know what code has been executed and in what order, i.e., the execution path, as well as values of certain variables. As log messages are generally sparsely printed, there may be tens of thousands of possible paths that lead to the same log message combination (Examples in Section 2). Engineers would need to manually reason about what code segments and control flows must have been executed (referred to as *Must-Path*) and what may have been executed (referred to as *May-Path*).

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

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

1.3 Current State of the Art

Most existing work on log analysis focus on using logs to classify errors or group similar failures into the same class, or automatically detecting recurring failures that match to some known issues [3, 8, 14, 27, 47]. For example, Windows Error Reporting system [22] and the Mozilla Quality Feedback Agent [34] automatically collect raw failure information (e.g. core-dumps and some heap data). It still relies on manual effort from the developers to inspect these runtime information to diagnose the error.

So far, few work has been conducted to automatically infer from both logs and source code to find out what have happened during a failed execution.

1.4 Our Contribution

In this paper we present SherLog, a postmortem error diagnosis tool that leverages logs as starting points in source code analysis to automatically infer what must or may have happened during a failed execution.

To be general and practical, we have the following design goals:

- **No need to re-execute the program:** For practicality, our tool only assumes the availability of the target program's source code and logs generated from failed executions. These two assumptions are quite reasonable as diagnosis are done by programmers themselves and it has been a common practice for customers to send product logs back to vendors. For example, most storage

```
 1 remove_parents (char *path)
 2 {
 3  char *slash;
 4
 5  while (1) {
 6    slash = strrchr (path, '/');
 7    if (slash == NULL)
 8      break;
 9
10     /* Remove any characters after the slash, */
11     slash[0] = 0;
12
13     /* printing out removing. */
14     if (verbose)
15       error (0,0,_("removing directory, %s"), path
               ); //r1
16
17     fail = rmdir (path);
18
19     if (fail) {
20       ...;
21       error (0,errno,"%s", quote(path)); //r2
22       break;
23      }
24   } //end while
25   return fail;
26 } //end remove_parent
```

```
30 main (argc, argv) {
31   ...
32   while (optc = next_commandline_option) {
33     switch (optc) {
34       ...
35       case 'p':
36         empty_paths = 1;
37         break;
38       case 'v':
39         verbose = 1;
40         break;
41         ...
42      }
43   }
44
45   for (; optind < argc; optind++) {
46     char* dir = argv[optind];
47     if (verbose)
48       error (0,0,_("removing directory, %s"),
                dir); //m1
49
50     fail = rmdir (dir);
51
52     if (fail)
53       error (0,errno,"%s", quote(dir)); //m2
54     else if (empty_paths)
55       remove_parents (dir);
56    } //end for
57 } //end main
```

Figure 1: Simplified rmdir.c source code. Bold statements are statements which print out console log messages.

vendors can collect failure logs from more than 50-75% of their customers [15, 28, 37].

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

- **No assumption on log semantics:** For generality, our work assumes no semantic information on logs. For example, our tool does not know which log message is a configuration parameter, etc. It simply treats every log message as a text string to match against source code. Every match provides a "hint point" for backward and forward inference on execution paths, variable values, etc. All such derived information is then combined together to prune infeasible ones and reveal a bigger, global picture regarding what may have happened during the failed execution.
- **Able to infer both control and data information**: Since both execution path and variable values provide programmers useful clues to understand what have happened during the failed production run, we infer both control flow and data flow information. This is accomplished via the following three steps (Figure 4): First, the log file is automatically parsed by our log parser to match each message to its corresponding logging statement(s) in source code (referred to as "Logging Points"). Variable values printed out in logging messages are also parsed to refine our analysis. Then SherLog's path inference engine infers the execution path based on information from our log parser, along with the constraints that an inferred path needs to satisfy. Finally, SherLog allows user to query the value of variables along an inferred execution path, and returns the inferred value of a variable at its definition and modification points.
- **Scalability:** Real world software often have thousands or millions lines of code. Therefore, our inference engine needs to be scalable to handle such large software. To improve scalability, we use function summaries and skip log-irrelevant code to limit the inference space.
- **Accuracy:** Information reported by SherLog needs to be accurate. Incorrect information can lead programmers to a wild goose chase, and thus waste effort and delay diagnosis. SherLog's path-sensitive analysis faithfully models the C program's semantic down to bit level, with a SAT solver to prune infeasible paths. Caller-visible constraint information is propagated from callee back to caller to ensure context-sensitivity.
- **Precision:** The reported information also needs to be precise as too many possibilities do not help programmers narrow down the root cause. For this reason, SherLog ranks the results based on probabilities. Must-Paths are ranked higher than May-Paths. Along each path, the *necessary* constraints, are ranked higher than *sufficient but not necessary* constraints. In value inference where multiple values of a variable might be possible, SherLog ranks concrete values higher than symbolic ones.
- **Capability of automatically generating log parsers:** Real-world software have hundreds or thousands of code lines that can print out log messages, many of which have distinctive log formats, therefore it would be tedious and error-prone to manually specify how to parse each log message. By extending the standard C format strings, SherLog can automatically generate a parser matching majority of the Logging Points. And with only a few annotations by developers, SherLog can be easily customized to developers' specific logging mechanisms.

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

For the real world example shown on Figure 1, the information generated by SherLog is presented by Figure 2. For the first path,

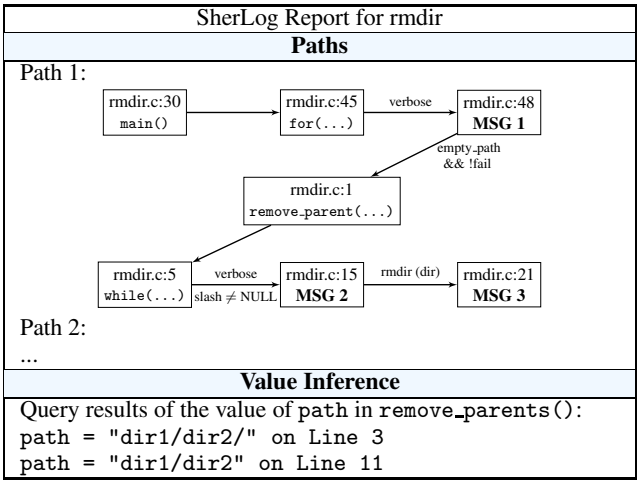


Figure 2: SherLog's error report for the `rmdir` bug. The Must-Paths are automatically generated to aid diagnosis. SherLog also infers the necessary constraints, i.e., constraints that must have been satisfied to execute a Must-Path.

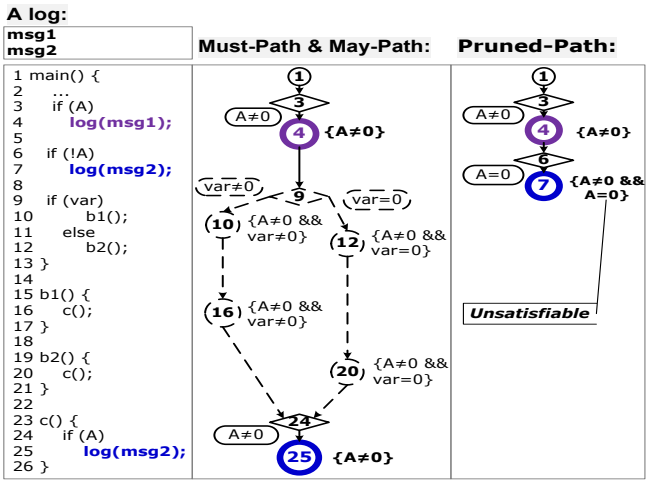


Figure 3: Must-Path, May-Path, and Pruned-Path. The Must-Paths are shown in solid lines, while May-Paths are presented in dashed lines. Pruned-Path is analyzed by SherLog but determined infeasible by our SAT solver. Logging Points are highlighted in bold circles and fonts. Constraints along the path are shown in $\{\}$ on the right of a circle.

$\{m1, r1, r2\}$, we have necessary constraints such as `strrchr()` returns non-NULL value at line 6, `rmdir()` returns failure (non-zero) at line 17. The developer can query the value of `path` in `remove_parents`, and SherLog would report the value of `path` at each definition point in `remove_parents`. All these information can help the developer getting much closer to the root cause.

2. SherLog Overview

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

Note that SherLog's objective is not to detect bugs. Instead, its objective is to infer information to help programmers understand what have happened during the failed execution. Such information includes the likely execution paths (and their constraints) as well as certain variable values along these paths. So at a high level, SherLog's objective is similar to program slicer [2] and core-dump analyzer [45]. But they differ in both inputs and information inferred.

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

A Realistic Goal: If we look at only the source code, there are many possibilities in terms of both execution paths and states during the failed execution. Fortunately, logs significantly help us narrow down the possibilities, even though it may not pin-point the execution path and state that actually happened during the failed execution. Therefore, a realistic goal is to find the following information (illustrated in Figure 3):

- *Must-Have*: partial execution paths that were definitely executed (referred to as *Must-Path*), and variable values that were definitely held during the failed execution.
- *May-Have*: partial execution paths that may have been executed (referred to as *May-Path*) and their corresponding constraints for them to happen, as well variable values that may have been held during the failed execution.
- *Must-Not-Have*: execution paths that were definitely NOT executed (referred to as *Pruned-Paths*).

SherLog Architecture: To accomplish above objectives, SherLog first needs to parse logs and using logs to identify starting points in source code for information inference. Then using the initial information provided by logs, it tries to statically "walk" through the code to infer the above information. As illustrated in Figure 4, SherLog has three main analysis steps: (1) *Log Parsing:* for each log message[3], identifying which lines in the source code prints it (referred to as *Logging Points*), and what program variable values are revealed in this log message(referred to as *Logging Variables*); (2) *Path Inference:* starting from the information provided by the log parsing step to infer Must-Paths, May-Paths and Pruned-Paths. (3) *Value Inference:* infer variables values on the paths produced by the previous step.

In this section, we provide a brief summary of each component. The details are explained in the following three sections.

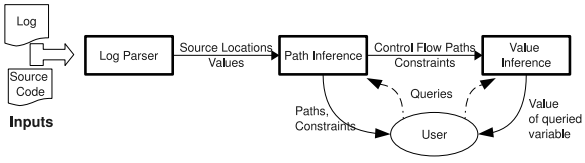


Figure 4: The components of SherLog.

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

[2] Even though we target for diagnosing production-run failures, SherLog can help diagnosing errors occurred during in-house testing as well.

[3] Each line in the log is referred to as a log message.

customized logging facility, SherLog also provides extension hooks to allow programmers to adapt specific logging utilities and format.

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

Value Inference: Theoretically it is impossible to provide sound and complete solution for value inference in static analysis, without environmental information such as inputs and libraries. However the observation made by SherLog is that in most cases only the values involved in constraints of the error execution path, as well as those printed in the log messages are important in error diagnosis. The former are important because they are the ones leading the execution to the error site, while the latter are important since programmers often only prints values that directly caused the error. SherLog makes best efforts to approximate a sound solution only to these variables to achieve overall balance between effectiveness and efficiency. We do so by symbolically execute the program faithfully following the path inferred in the path inference step, taking all the constraints along this path as facts for variable values, and propagating the value information across function calls.

We build all three components of SherLog on top of the Saturn [49] Static Analysis Framework, because Saturn offers both the flexibility in expressing our analysis algorithm and precise static analysis of C programs.

3. Log Parsing

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

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

3.1 Challenges

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

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

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

3.2 Our Approach

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

Basic LogParser: Our basic LogParser requires programmers to indicate only the logging functions and which parameter in this function is the format string. For example, in the `rmdir` example, `error()` is the logging function and its third parameter is the format string.

For the `rmdir` example, the user provides 3 annotations, i.e., {`error(), 3, 4`}, meaning `error()` is the logging function, the 3rd parameter of `error()` is the format string ("`removing directory, %s`), and parameters starting from the 4th parameter feed the wild cards, e.g., `%s`, in the format string.

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

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

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

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

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

To address this problem, LogParser provides a simple extension mechanism to allow programmers to annotate complex format string. Programmers can define a new format string `‘%s’: %{serrno}`, along with a rule shown in figure 5 to handle the newly defined specifier `%{serrno}`. This new format string can be used to

match [msg3], with %s mapped to dir1/dir2, and serrno mapped to a Regex instance No such file or directory.

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

Figure 5: User defined rule to handle error strings returned by strerror().

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

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

4. Path Inference

Given a log, we want to recreate paths that could have generated the log or parts of the log, along with the constraints that the path needs to satisfy. To make SherLog useful and practical for real world systems, SherLog has to meet several design goals: (1) *Precise*: be as concrete as possible to narrow down the possibilities for programmers. (2)*Accurate*: information reported by SherLog should be correct. (3) *Scalable*: can handle large software. (4)*Relevant*: most information reported should be related to the occurred failure.

To achieve the above goals, SherLog's Path Inference engine uses a log-driven, summary-based approach. Being log-driven allows SherLog to focus on relevant information. Summary-based approach enable SherLog to scale to large software such as server applications. To be accurate, SherLog uses inter-procedural path- and context-sensitive analysis, and reports the constraints associated with each feasible path. To improve precision, SherLog separates Must-Path from May-Path so programmers can first focus on those paths that have definitely happened. Additionally, when it is impossible to narrow down the complete path between two log points, SherLog tries to reconstruct the maximum partial paths.

We first give a high-level overview of the design and process, and then describe our approach in more detail.

4.1 Overall Design and Process

SherLog is summary-based, that each function f's body is analyzed separately, and at the call-site of f, only a summary of f is used. The summary of each function f is the subsequences of logging messages f might print out, along with the constraints that needs to be satisfied for f to print out those messages. If there is a path P in f that calls a and b, a prints [msg 1] under the constraint C_a and b prints [msg 2] under the constraint C_b, then f might print [msg 1, 2] given that the constraint $C_a \wedge C_b \wedge C_P$ is satisfiable, where C_P is the constraint along P. Then the analysis is a recursive process, that given a message sequence, initially the Log Parser provides the Logging Points in the program that print each message, under the constraints that the log variables must hold the values revealed in the log. SherLog starts with the functions that directly call these Logging Points, and recursively propagating the subsequences a function connects from callee to caller, along with constraints. The caller would concatenate all the sub-sequences the callees connects into a longer sub-sequence. At the end, if we can infer a complete execution path of the program that prints the entire message sequence, then we should find main has summary that prints the entire log. Otherwise, we only inferred partial paths. The path SherLog reports is the call chains and the constraints it satisfies. Figure 6 shows the analysis order of functions for the rmdir case.

Loops are modeled as a function with a tail-recursive call to itself in SherLog, that at the end of a loop body, a call instruction is inserted with the target to the current loop's head. Thus, loop is treated with no difference than a function. We will use the term function to also refer to loop in the rest of paper. This way, repeating logging messages printed by loop iterations would be precisely connected by analyzing function/loop repeatedly until it can not connect more logging messages.

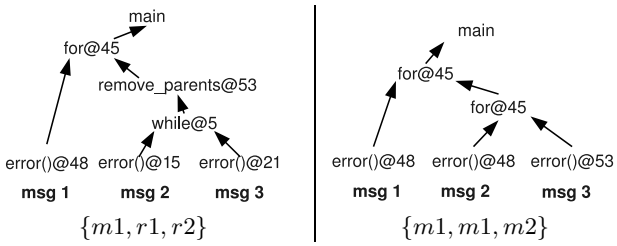


Figure 6: SherLog's analysis for rmdir.

SherLog only analyzes functions that directly or indirectly calls the Logging Points. In the rmdir case, only the functions shown in Figure 6 are analyzed, while functions like rmdir() on line 17 in Figure 1 is skipped. For functions SherLog analyzes, we use precise path-sensitive analysis to faithfully follow the program's semantic. For functions that doesn't print logging messages, SherLog simply ignores their side-effects. This is a trade-off in our design decision to balance scalability with precision. This allows us analyzing functions that *must* affect the execution that generates the logging sequence precisely, while not wasting time on those functions that *may* have effects. Skipping the side-effects of functions is generally unsound, that might introduce both false-positives and false-negatives in the paths we inferred. However, we find this approach works well in practice since partial side-effects of these skipped functions could still be recovered from the constraints along a path. For example, without analyzing rmdir() on line 17, SherLog still infers that it returns non-zero value along the path $\{m1, r1, r2\}$; without analyzing the while loop at line 32, SherLog still infers that verbose needs to be non-zero along the two paths. Later in our value inference, we allow user to further query why the return value of rmdir() is non-zero. Modeling all functions in the program would result in a full-symbolic execution approach which is unlikely to scale well to large applications.

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

Figure 6 shows how SherLog infers the two paths $\{m1, r1, r2\}$ and $\{m1, m1, m2\}$ in rmdir example. For $\{m1, r1, r2\}$, log parser could map message 1, 2 and 3 to error() at line 48, 15 and 21 respectively. SherLog first analyzes the while loop at line 5, as it will directly call two Logging Points error()@15 and error()@21. This would result in a path error()@15->error()@21, with the constraints C_w as "strrchr $\neq$ NULL $\wedge$ verbose $\neq 0 \wedge$ rmdir()@17 $\neq$ 0". Thus a *summary* of this while loop will be generated and stored, including the path, the messages it connects ([msg2] and [msg3]),

and a converted constraint C_r as "verbose $\neq$ 0". Because this summary is to be used by the caller of the while loop, so the constraint must be converted to filter out caller invisible fields such as "strrchr $\neq$ NULL $\wedge$ rmdir()@17 $\neq$ 0". Then SherLog analyzes the caller of this while loop, `remove_parents`, directly using the summary of the while loop at line 5 to propagate message connection information to the caller. This will generate a summary for function `remove_parents` similar as the one for the while loop. Next SherLog analyzes the for loop in `main`, which finds a path `error()@48 ->remove_parents@55`, along with the constraint C_p as "verbose $\neq$ 0 $\wedge$ rmdir()@50 = 0". It then uses the previously computed summary of `remove_parents`, and further connects message 1 with message 2 and 3 by solving the constriant $C_p \wedge C_r$ along this path.

Figure 6 also shows the inference of path $\{m1, m1, m2\}$ in the `rmdir` example. This path involves the `for` loop at line 45 being repeated twice. At the beginning, SherLog analyzes this loop the first time by connecting message 2 and 3. The next time, it will first connect message 1, then use the previously computed summary of itself to further connect 1 with 2 and 3. SherLog would analyze the `for` loop one more time and find out it can no longer produce any new summaries. Thus SherLog would stop the analysis.

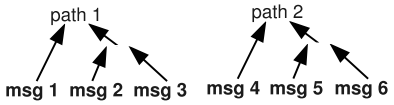


Figure 7: SherLog connects maximum partial message sequence.

This design also makes SherLog practical to connect maximum partial paths if the complete path is unavailable, e.g., because of multi-thread executions or system restart. Figure 7 shows the case where message 1,2 and 3 are generated by thread 1, while message 4, 5 and 6 by thread 2. The bottom-up design of SherLog still connects message 1,2 and 3 with path 1 and 4,5,6 with path 2, although SherLog couldn't statically further connect path 1 and 2. If message 6 is the actual error message, SherLog would report path 2 to user which is the longest path involving a message of interest.

4.2 Overview of Saturn Static Analysis System

We give a brief background of the Saturn Static Analysis framework that SherLog is built on before we move into the detail implementation. Saturn is a static analysis framework of C programs. User writes analysis in a logic programming language to express the program analysis algorithm. Saturn is summary-based, and models loops as tail-recursive function calls. It is also constraint-based, that the analysis captures all the conditions as constraints along a path that reaches the program point of interest. A SAT solver can be used to report whether these constraints are satisfiable. Saturn models all C program's construct such as integers, structures, pointers, etc., faithfully [49] by statically name every memory location accessed by each statement uniquely within current function being analyzed [4]. Thus, each bit accessed by the function is represented by a distinct boolean variable. Saturn also supports alias analysis, with an option to turn it on for the analysis. Currently SherLog doesn't turn on alias analysis, and assumes non-aliasing among pointers.

4.3 Detailed Design

We formalize the path inference problem as a constrained sequence matching problem. Given a log file L, we use a sequence of integers $\mathscr{M} = [1, 2, \cdots n]$ to match the sequence of logging messages, an integer i corresponds to the ith message in the log. Log parser generates a set of Logging Points for each log message:

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

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

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

$$sum(F, i, k, P_{i,k}, C_F)$$

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

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

$$\begin{aligned} logp(PP, i, k, P'_{i,k}, C'_F) :- \\ & PP \text{ is the call site to } F', \\ & sum(F', i, k, P_{i,k}, C_{F'}), \\ & P'_{i,k} = \text{concatenate}(F'@PP, \texttt{"->"}, P_{i,k}), \\ & \text{convert}(C_{F'}, C'_{F'}) \end{aligned}$$

This rule propagates the sequence matching behavior of callee to the caller's context. For the path string $P_{i,k}$, we attach the current function name F' and call-site to the path string within F', thus recursively construct a path string such as `"...remove_parents@55->while@5->error@15..."`. The $convert$ predicate is to convert the constraint from callee's context into the caller's context. We implement *strongest observable necessary condition* [18] for constraint convertion, which filters all the caller-unobservable conditions involving local variables and keeping only the caller-observable conditions such as return values, function arguments and globals. It guarantees the converted constraint is a necessary condition of the orginal one to be conservative.

With the $logp$ defined, we can now summarize the sequence matching behavior of F as:

$$\begin{aligned} sum(F, i_1, k_m, P_{i1,km}, C_F) :- \\ & PP_1, PP_2, \ldots, PP_m \text{ is a path in } F \text{ with constraint } C_p, \\ & logp(PP_1, i_1, k_1, P_{i1,k1}, C'_{F1}), \cdots\cdots \\ & logp(PP_m, i_m, k_m, P_{im,km}, C'_{Fm}), \\ & k_1 + 1 = i_2, k_2 + 1 = i_3, \ldots, k_{m-1} + 1 = i_m, \\ & P_{i1,km} = \text{concatenate}(P_{i1,k1} \ldots P_{im,km}), \\ & C_F = C_p \wedge C'_{F1} \ldots \wedge C'_{Fm}, \text{SAT}(C_F) \end{aligned}$$

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

At the beginning we initialize the $logp$ for each Logging Point l_{ij} as:

$$logp(l_{ij}, i, i, \text{“”}, \text{log varialbes=log values})$$

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

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

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

4.4 Reporting Inferred Paths

There might be multiple paths inferred by the static analysis engine that connects the same Logging Point sequence. In the example shown in Figure 3, we find two paths `main->log@4->b1@10 ->c@16->log@25` and `main->log@4->b2@12->c@20->log@25`, all connecting the same Logging Point sequence: `log@4,log@25`. In real world applications the scarcity of Logging Points might result in tens of thousands paths connecting two Logging points. To be useful, SherLog needs to effectively summarize these paths before output them to the user.

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

The final output of SherLog's path inference would be the Must-Paths that connects the longest sequence of logging messages involving the error message, along with a randomly selected May-Path for each Must-Path. By default, SherLog assumes the last message is the error message. The user could query paths involving other messages or other May-Paths. For the `rmdir` example, there are two Logging Points sequences, $\{m1, r1, r2\}$ and $\{m1, m1, m2\}$, so we report two Must-Paths. Since each Must-Path has only one May-Path, so in the end SherLog reports two Must-Paths same as May-Paths: `main->for@45->error@48 ->remove_parents@55->while@5->error@15` and `main->for@45 ->error@48->for@45->error@48->error@53`. The constraints with each function/loop record along each path can be inspected.

5. Value Inference

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

```
1: a = argc;
2: if (c == 1)
3:   a = 1;
4:
```

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

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

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

In the `rmdir` example, the user might query the value of variable `path` in `remove_parents`, along the path `"main->for@45->error@48 ->remove_parents@55->while@5->error@15->error@21"`. SherLog starts with `main`, then the body of the `for` loop at line 45. At line 48, SherLog infers logging variable `dir` points-to a concrete value "dir1/dir2/". At line 55, this information is propagated to `remove_parents` as context information. Within `remove_parents`, value of variable `path` is initialzed as "dir1/dir2/". At line 6, SherLog would infer `slash` points to the last '/' in "dir1/dir2/" after calling `strrchr`. Then at line 11, the assignment would change the path from "dir1/dir2/" to "dir1/dir2", removing the last slash. The output of SherLog for this query is shown in Figure 2.

Like Path Inference, SherLog's Value Inference also skips the analysis on functions that doesn't print logs, which might cause Value Inference return incorrect result. It also does not model complicated C's features such as pointer arithmetic. Instead, we do best effort in ensuring the correctness of values involved in constraints of the path and those printed in the log messages, guaranteeing that our inference result would always conform with the constraint of the path. User can also force SherLog to analyze

functions that are skipped in Path Inference. For example, in the `rmdir` case user can query the return value of `rmdir()` at line 17. SherLog still answer this query by propagating constants along the path, and into the function body of `rmdir()` after the call at line 17 within the while loop. Our evaluations on real world errors confirmed the effectiveness of our value inference.

6. Evaluation Methodology

We evaluate SherLog on 8 real world failures from 7 applications (including 3 servers), which are summarized in Table 1. This suite covers a wide spectrum of representative real-world failures and applications. Six (6) of the failures are caused by software semantic bugs and 2 by configuration errors, which no prior static analysis work could diagnose.

For evaluation purpose, we manually reproduced and diagnosed each failure, collected the run-time log, and summarized the information essential for diagnosing each error. Results generated by SherLog are compared against our summary. If SherLog could infer a subset of the summarized information we consider it *useful*. If all the information essential for diagnosing the failure are inferred correctly by SherLog, we consider SherLog *complete*.

Our experiments are conducted on a Linux machine with 8 Intel Xeon 2.33GHz CPUs, and 16GB of memory. SherLog is a single threaded program. We set a 30 seconds time-out threshold, so that each function/loop will not be analyzed more than 30 seconds.

	Log Parser		# Paths		Path Len			Effective	
Name	Regex	log Pts	Must	May	Must	May	Msg	Useful	Comp.
rmdir	4	10	2	2	4.5	4.5	3	√	√
ln	17	23	1	1	5	5	2	√	√
rm	17	25	1	10	7	13	4	√	√
CVS1	695	1,173	1	2	2	4	2	√	√
CVS2	695	1,173	1	120	5	12	1	√	√
Apache	997	1,259	1	1	8	8	10	√	√
Squid	1,134	1,209	1	57	9	15	108	√	√
TAR	171	228	5	24	3	7	1	√	√

Table 2: Detailed result for SherLog. Regex is the number of regular expressions (format strings) generated by the Log Parser. Log Pts. is the number of logging points in the program that matches these regular expressions. # Paths is the number of each type of paths. Path Len is the length of the paths. We only count number of function calls, ignoring loops. For multiple-must paths, we report the average length. The length of May-Path is the length of a randomly chosen May-Path along the Must-Path. Msg is the length of the May path in terms of number of logging messages it connects, including the error message. Comp. stands for completeness.

7. Experimental Results

7.1 Overall Results

Table 2 shows the diagnostic results by the SherLog on the 8 errors. In all 8 cases, SherLog correctly and completely inferred all the diagnostic information.

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

7.2 SherLog Diagnosis Case Studies

We use 3 errors as case studies to demonstrate the effectiveness of SherLog in help diagnosing bugs and mis-configurations.

Case 1:ln

SherLog Report for ln

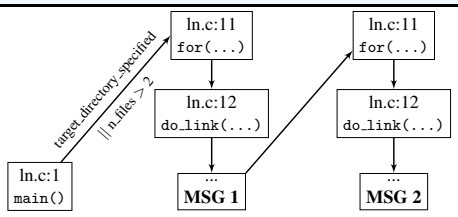
Synptoms

A user uses `ln` to create a hard link for a file, but somehow `ln` tries to create a hard link for the current directory `"."` and fails.

Log Traces

```
create hard link ‘./dir1/target’ to ‘target’ [MSG 1]
ln:‘.’: hard link not allowed for directory [MSG 2]
```

Paths



Value Inference

Query results of the value of `file[1]` in `main()`:

`file[1] set to "." on Line 6, if n_files == 1`

Figure 8: SherLog report for ln in coreutils 4.5.1

```
1  int main (int argc, char **argv) { ...
2    if (n_files == 1) {
3        static char *dummy[2];
4        dummy[0] = file[0];
5        dummy[1] = ".";
6        file = dummy;
7        n_files = 2;
8        dest_is_dir = 1;
9    } ...
10   if (target_directory_specified || n_files>2) {
       ...
11     for (i = 0; i <= last_file_idx; ++i)
12       errors += do_link(file[i], target_directory);
13   }
14 }
```

Figure 9: ln.c in GNU Coreutils-4.5.1. The log messages are printed in function `do_link()`, whose code is not shown due to space limit.

A user uses `ln` of coreutils 4.5.1. to create a hard link for a file, but somehow `ln` tries to create a hard link for the current directory (“.”) and fails. The log is shown in Figure 8, while the highly simplified code is shown in Figure 9.

SherLog automatically infers control flow and data flow information (the bottom of Figure 8) that is useful for the developer to understand the root cause of this error. SherLog infers one Must-Path from the log messages. It shows that `do_link()` in Line 12 was executed twice, and it failed in the second time and printed out log message [MSG 2]. It also shows that the constraints `target_directory_specified || n_files > 2` must be satisfied for this error to happen. Then the developer would naturally want to know why the second file name, variable `file[1]`, was set to the current directory. So he or she can query SherLog for this information. From the source code and the log messages, SherLog infers that `file[1]` is set to the current directory on Line 5 if `n_files` (the number of files to be linked) is 1. Combined with the path constraints `target_directory_specified || n_files > 2`, she or he

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

Table 1: Applications and real errors evaluated in our experiments. Type indicates the error type, either software bug or configuration error. LOC is the number of lines of code. #MSG is the number of logging messages in the error's log.

SherLog Report for Squid

Symptoms

End users can not see certain FTP icons if they connect the FTP server via Squid 2.3.STABLE4 [39].

Log Traces

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

Paths

store.c:1 `storeCheckCacheable()` —storeCheckTooSmall(e) $\neq 0$→ store.c:3 **MSG 1**

Value Inference

```
#return@storeCheckTooSmall =

   { 0   if !EBIT_TEST(e->flags, ENTRY_SPECIAL) || ...
   { 1   if EBIT_TEST(e->flags, ENTRY_SPECIAL) || ...
```

Figure 10: SherLog diagnosis report for Squid 2.3.STABLE4.

can infer that `target_directory_specified` is set, because `n_files > 2` is not true. At this point, the developer would know the root cause. Line 5 (and the if statement from Line 2 to 9) should only be executed if no link name is specified for `ln`, so `ln` would create a hard link in the current directory (by setting file[1] to be "."). If `target_directory_specified` is set, then Line 5 should not be executed, meaning the code forgets to check if `target_directory_specified` was set at Line 2. Replacing Line 2 with `if (n_files == 1 && !target_directory_specified)` would fix the problem.

Case 2: Squid

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

Figure 11: store.c in Squid 2.3.STABLE4.

User experiences some missing icons on some web-pages. The log messages of Squid proxy server seem to imply that Squid thinks the missing file is too small to be cached. SherLog reports 57 May-Paths that could connect this particular message, however, only 1 Must-Path is reported(Figure 10). The highly simplified source code is shown in Figure 11.

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

Case 3:CVS Configuration Error

SherLog Report for CVS

Symptoms

Some users in a corporate network cannot perform any CVS operations.

Log Traces

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

Paths

parseinfo.c:1 `parse_config()` → parseinfo.c:3 `while(...)` —(!strcmp(line, "RCSBIN") && !strcmp(line, "SystemAuth") ...)→ parseinfo.c:18 **MSG 1**

Value Inference

Not Needed.

Figure 12: SherLog report for CVS in coreutils 4.5.1

Some users in a corporate network cannot perform any operations with CVS. The error can be difficult to diagnose because this failure affects only a portion of the users, all of which access the same repository. The result of inferred executions along with its constraints are shown in Figure 12.

SherLog helps to locate relevant source code function `parse_config()`, which matches keywords by calling `strncmp()` one by one, and signaling failures when the keyword does not match any of them. Therefore, the constraints in Figure 12 captures all supported keywords by this particular CVS version. They imply that the keyword `UseNewInfoFmtStrings` is not supported since it is not in the constraints. It turns out the keyword is a new feature added in CVS 1.12. Therefore users of earlier versions should not use this option. Upgrading the CVS package solves the problem.

7.3 Performance of SherLog

Table 3 shows the performance of SherLog on each case. All diagnostic components of SherLog finishes within 40 minutes. Among the three components, Path Inference takes the longest time, because it needs to scan through the entire program, and analyzing log-related functions multiple times. Once the paths are inferred, value inference only analyzes the functions/loops along the inferred path, which greatly reduces the analysis time. Log Parser's timing overhead is negligible.

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

Name	Parser Time	Path Time	Path Mem.	Value Time	Value Mem.
rmdir	0.02s	2.25m	174 MB	15.54s	116 MB
ln	0.02s	2.32m	194 MB	37.75s	165 MB
rm	0.01s	2.00m	511 MB	38.87s	123 MB
CVS1	0.32s	39.56m	1,317 MB	188.53s	323 MB
CVS2	0.19s	38.96m	1,322 MB	39.19s	232 MB
Apache	0.67s	28.38m	321 MB	19.23s	217 MB
Squid	0.81s	38.02m	1,520 MB	22.01s	252 MB
TAR	0.08s	6.55m	210 MB	29.14s	155 MB

Table 3: Performance of SherLog. s stands for seconds, m stands for minutes. Mem. measures the maximum memory usage at any given time during the execution. Value inference's performance is measured by querying multiple relevant variable values in one pass along the inferred path.

8. Discussion and Limitations

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

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

Handling logs from concurrent execution: SherLog focuses on connecting continuous log message sequences.Some mature software such as Apache HTTPD or Log4j [32] usually record thread IDs in logging messages. So it is easy to separate the log messages out for each thread. We can then apply SherLog to each thread's logs to find path and variable information in this thread. To infer information across threads is interesting and challenging, which remains as our future work.

Locating relevant log messages: SherLog assumes our users have certain understanding of the log to be able to identify the error symptom messages out of millions lines of various messages. The common practice in large programs is to divide logging messages into different severity levels, from informational to fatal error messages. In this case, users only need to focus on messages above certain level. In addition, current log analysis work [27] have demonstrated the effectiveness of locating a small number of suspicious log messages from millions lines of log messages. These approaches could be used together with SherLog.

9. Related Work

Log analysis: Existing log analysis work focused on using statistical techniques to detect anomaly indicated by logs, or detect recurring failures that match known issues [3, 8, 14, 21, 27, 47]. These studies did not leverage source code for extracting control-flow and data-flow information, and thus cannot recreate the execution paths (or partial execution paths) and run-time variable values as SherLog does to help developers diagnose errors. Through studying commercial storage system logs, Jiang et. al. [27] pointed out that logs can be of great value in error diagnosis. They also proposed using statistical techniques to identify key events recorded in the log that can help pinpoint the root causes. Xu et. al. [47] applied machine learning techniques to learn common patterns from a large amount of console logs, and detect abnormal log patterns that violate the common patterns. To help parse logs more accurately, they analyzed the Abstract Syntax Tree of the program to extract the format strings used to print out the logs. However, their error detection and diagnosis are based on learning patterns solely from log messages, and thus cannot provide the capability of generating run-time control-flow and data-flow information. Magpie [8] addressed the problem of grouping log messages generated by the same request in a distributed environment. Once the log messages are grouped in per-request basis, they could compute the resource consumptions for each request, then use clustering techniques to learn the common patterns for each request, and identify anomalous requests.

Trace analysis: Trace analysis [6, 9, 11, 12, 31] analyzes execution traces generated at run-time by instrumented programs to detect or diagnose errors. While SherLog leverages logs produced by vendor's software as is, trace analysis requires to modify the software statically or dynamically to add instrumentations, which may not always be feasible, especially for production software. For example, Liblit et. al. [31] introduced "cooperative bug isolation", which collects run-time traces from instrumented programs by sampling from many users to offload the monitoring overhead. With some classification techniques, they can pinpoint the predicates recorded in the traces that are most correlated with the bugs. HOLMES [12] further investigated how path profiling based program sampling can help bug isolation. They also developed an iterative and bug-directed profiling technique to effectively isolate bugs with low overheads.

Error diagnosis without error reproduction: Some error diagnostic work uses static analysis thus do not need to reproduce the failures [26, 33, 40, 41]. PSE [33] performs off-line diagnosis of program crashes caused by one particular type of errors, i.e., NULL pointer dereferences. Different from PSE, SherLog is general to diagnose all types of errors as long as some error related information is contained in the log. In addition, PSE requires the availability of core-dumps, which are generally not available for non-crash errors. Static slicing [26, 40, 41] produces a smaller slice of a program according to a slicing criterion, which can be used for error diagnosis. SherLog can be viewed as a static slicer that uses logs as the slicing criterion to balance between scalability and precision.

Error diagnosis requiring error reproduction: Traditional error diagnosis and fault localization often relies on reproducing the error on the vendors' site [1, 2, 7, 16, 25, 42, 44, 50]. As discussed in section 1.1, this assumption may not hold for real-world software errors. Agrawal et. al. adapt execution slices and data flow tests to localize the root causes of bugs [2]. Ball et. al. propose an algorithm to localize the root causes from model checking error traces [7].

Static analysis and software bug detection: Many analysis tools are used or can be used to statically *detect* software bugs [5, 10, 13, 20, 48, 49]. SherLog is different from these studies because it leverages run-time logs and its goal is error diagnosis as opposed to error detection.

10. Conclusion

We designed and implemented a practical and effective diagnosis technique, SherLog, which can analyze logs from a failed production run and source code to automatically generate useful control-

flow and data-flow information to help engineers diagnose the error without reproducing the error or making any assumption on log semantics. We evaluate SherLog on 8 real world software failures, i.e., 6 bugs and 2 configuration errors, from 7 open source applications including 3 servers. For all of the 8 failures, SherLog infers useful and precise information for developers to diagnose the problems. In addition, our results demonstrates that SherLog can analyze large server applications such as Apache with thousands of logging messages within 40 minutes.

Acknowledgments

We thank the anonymous reviewers for their insightful feedback, the Opera group and Yoann Padioleau 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) and NetApp Gift grant.

References

[1] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Debugging with dynamic slicing and backtracking. *Software—Practice and Experience*, 23(6):589–616, June 1993.

[2] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong. Fault localization using execution slices and dataflow tests. In *ISSRE'95*.

[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. Aiken, S. Bugrara, I. Dillig, T. Dillig, P. Hawkins, and B. Hackett. The Saturn Program Analysis System.

[5] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *SP '02: Proceedings of the 2002 IEEE Symposium on Security and Privacy*.

[6] 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*.

[7] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: localizing errors in counterexample traces. *ACM SIGPLAN Notices*, 38(1):97–105, Jan. 2003.

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

[9] E. Bodden, P. Lam, and L. Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In *FSE'08*.

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

[11] F. Chen and G. Roşu. Parametric trace slicing and monitoring. In *TACAS'09*.

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

[13] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea. Selective Symbolic Execution. In *HotDep'09*.

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

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

[16] R. A. DeMillo, H. Pan, and E. H. Spafford. Critical slicing for software fault localization. In *ISSTA*, pages 121–134, 1996.

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

[18] I. Dillig, T. Dillig, and A. Aiken. Sound, complete and scalable path-sensitive analysis. *SIGPLAN Not.*, 2008.

[19] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution replay of multiprocessor virtual machines. In *VEE'08*.

[20] D. Engler, B. Chelf, and A. Chou. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI'00*.

[21] K. Fisher, D. Walker, K. Q. Zhu, and P. White. From dirt to shovels: Fully automatic tool generation from ad hoc data. In *POPL'08*.

[22] 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*, pages 103–116, New York, NY, USA, 2009. ACM.

[23] J. Gray. Why do computers stop and what can be done about it?, 1985.

[24] 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*.

[25] R. Gupta, M. L. Soffa, and J. Howard. Hybrid slicing: integrating dynamic information with static analysis. *ACM Transactions on Software Engineering and Methodology*, 6(4):370–397, Oct. 1997.

[26] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *PLDI '88*.

[27] W. Jiang. Understanding storage system problems and diagnosing them through log analysis. *Ph.D. Dissertation*.

[28] W. Jiang, C. Hu, S. Pasupathy, A. Kanevsky, Z. Li, and Y. Zhou. Understanding customer problem troubleshooting from storage system logs. In *FAST'09*.

[29] S. Kandula, R. Mahajan, P. Verkaik, S. Agrawal, J. Padhye, and P. Bahl. Degailed diagnosis in enterprise networks. In *SIGCOMM'09*.

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

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

[32] Apache Logging Services - Log4j. `http://logging.apache.org/log4j`.

[33] 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.

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

[35] S. Narayanasamy, C. Pereira, and B. Calder. Recording shared memory dependencies using strata. In *ASPLOS'06*.

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

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

[38] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *ASPLOS'09*.

[39] Squid Archives. `http://www.squid-cache.org/Versions/v2/2.3/bugs/#squid-2.3.stable4-ftp_icon_not_found`.

[40] M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. In *PLDI'07*.

[41] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[42] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. Triage: Diagnosing production run failures at the user's site. In *SOSP'07*.

[43] VMWare. Using the intergrated virtual debugger for visual studio. `http://www.vmware.com/pdf/ws65_manual.pdf`.

[44] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration debugging as search: finding the needle in the haystack. In *OSDI'04*.

[45] Windows Error Reporting(Dr.Watson). `http://www.microsoft.com/whdc/maintain/StartWER.mspx`.

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

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

[48] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. In *OSDI'04*.

[49] Y.Xie and A.Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *Transactions on Programming Language and Systems*, 29(3):1–16, 2007.

[50] A. Zeller. Isolating cause-effect chains from computer programs. In *FSE'02*.