2018 25th Asia-Pacific Software Engineering Conference (APSEC)

NotOnlyLog: Mining Patch-Log Associations from Software Evolution History to Enhance Failure Diagnosis Capability

Shuqi Chi*, Shanshan Li*, Yong Guo*, Wei Dong*, Zhouyang Jia*, Haochen He* and Qing Liao†
*Department of Computer Science, National University of Defense Technology, Changsha, China
Email: {chishuqi16,shanshanli,yguo,wdong,jiazhouyang,hehaochen13} @nudt.edu.cn
†Department of Computer Science and Technology,Harbin Institute of Technology(Shenzhen), Shenzhen, China
Email: liaoqing@hit.edu.cn

***Abstract*—Log messages are widely used in the diagnosis of software failures. Existing studies of failure diagnosis based on log messages tend to use rule-based methods or execution-path-based methods. Rule-based methods generate bug-fixing rules using either human expertise, which is time consuming, or machine learning methods, which may lack the precision of failure diagnosis. To remedy these problems, researchers propose execution-path-based methods that reconstruct execution paths by analyzing source code and run-time logs. These methods, however, may lead to path explosion. To fill this gap, our work focuses on solving the path explosion problem in execution-path-based methods. We assume that run-time logs may have a relationship with their corresponding patches in real-world bug reports. We conduct empirical studies on seven open-source software packages and obtain two findings: 1) 80% of similar bugs have similar patches, and 2) 70% of faulty code is found to lie near the code where the first failure message is printed.**

Based on these two observations, we design and implement a practical tool NotOnlyLog for bug diagnosis. NotOnlyLog is able to mine the relationships between failure logs and their corresponding patches, in order to reduce both the number and length of uncertain execution paths in bug diagnosis. We evaluate the performance of NotOnlyLog on nine real-world bugs from three large open-source projects. Our experimental results show that, compared with SherLog, NotOnlyLog can achieve a reduction of 86.9% in the number of execution paths.

***Keywords*-Bug Diagnosis,Software Evolution, Log**

I. Introduction

As the complexity and scale of software increases, the quantity of software bugs is growing at a startling rate. System run-time logs [1] are very important sources of information for software failure diagnosis. Run-time logs often contain vital information for use in troubleshooting run-time failures, and are often the sole data source available to software developers in order to understand and diagnose these failures.

Many studies have been conducted to help system maintainers with failure diagnosis. Existing studies in this area tends to be based on either rule-based methods [2]–[6] or execution-path-based methods [7]–[9]. Rule-based methods concentrates on generating rules or patterns, using either human expertise or machine learning methods. For example, Logsurfer [5] uses rules formulated by people with strong domain knowledge to diagnose software failures. However, the frequent changing nature of the source code requires too expensive expertise cost to keep generating new rules in time [7]. Distalyzer [3] uses machine learning techniques to generate rules, which are extracted from run-time logs,to compare system behaviors, in order to reveal which component of the software is broken. However, developers still need a long time to determine the root cause of these failures. Execution-path-based methods [7], [8] usually target at reconstructing the execution paths when system failures occur. These methods can always provide the system maintainers with more detailed information on how the failures happened and thus give maintainers a deeper insight into what went wrong. As a representative work, SherLog [7] performs a static, context-sensitive approach to reconstruct the failure execution paths by analyzing both run-time logs and source code. However, execution-path-based methods are not able to infer every choice at branch instructions in failed execution, and this could result in path explosion. And the possibility of path explosion grows while the scale and complexity of software increase. To remedy this problem, Pensieve [8] uses event chains instead of execution paths to profile system traces, while it sacrifices accuracy, and can lead to some infeasible or inaccurate result.To solve path explosion of execution-path-based methods and keep its accuracy, we conduct empirical studies on software evolution and bug fixing history to figure out the ways to prune execution paths.

Bugs often have corresponding run-time logs and patches. We believe that there may be some relationships between these logs and the fixing patches. As Figure 1 shows, the run-time failure messages of MDEV-13591 and MDEV-8195, two bugs in MariaDB [10], contain similar information, and both of their corresponding patches have the same variable "crypt_data". This indicates that similar bugs may have similar root causes. In order to understand the relationships between logs and patches, we perform a study on seven popular open-source software packages, including MariaDB, Squid, CUBRID, Httpd, OpenStack, OpenSSH and Nginx. We obtained two key findings:1) 80% of similar bugs have similar patches, and 2) 70% of patches is found to have

978-1-7281-1970-0/18/$31.00 ©2018 IEEE
DOI 10.1109/APSEC.2018.00033

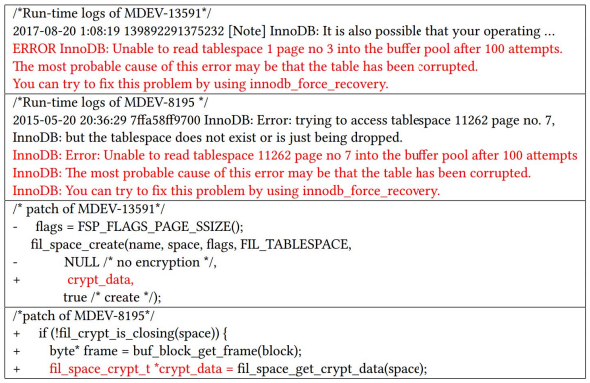

Figure 1. A representative example of bugs (MDEV-13591 and MDEV-8195) and patches

a distance of less than 2 to the first failure message, where distance denotes the number of functions between the statements in patches to the statements which print first error messages in system execution paths. Based on these findings, we leverage the relationships extracted from patches and logs from historical bugs and combine this with the execution-path-based method to prune the execution paths for more precise bug diagnosis.

Extracting and using the relationships contained in historical bugs and patches to help the process of bug diagnosis has many challenges. Firstly, given two bugs, it is non-trivial to determine the similarity of their logs that report errors. This is because that the failure messages are often, if not all the time, hidden in massive normal messages. Secondly, given two bugs with similar failure logs, it is hard to determine the similarity of their patches. On one hand, we need to choose proper features that can effectively determine patch similarity. On the other hand, the patches are code snippets without complete semantic information, making it more difficult to extract the features, e.g., we do not know the types of the variables used in the patches. Thirdly, we do not have sufficient run-time logs and corresponding patches for each bug for relationship mining, since run-time logs are not required by many software.

To address the challenges outlined above, we studied the log messages together with their producing mechanism, the features of patches, and the feasibility of reproducing logs from patches. To distinguish failure messages from normal ones, we manually studied the subject software about how messages are printed. We found that most failure messages were printed in special patterns, which was different from normal messages. Based on this observation, we develop a keyword-based method to generate rules to separate the failure messages from normal messages. To choose proper features of patches to determine patch similarity for similar logs, we studied the patches of similar logs manually. We found that patches of many similar logs underwent similar modifications to same variables or functions. Hence, we take features of patches from these modifications. In order to facilitate extracting features, we study the source code where the patches located to complete the semantic information of patches, in order to extract features efficiently. To obtain sufficient run-time logs for our study, we leverage a control-flow analysis on patches to identify the likely messages, in order to reproduce run-time logs. Solving those problems enables us to mine relationships and use them to facilitate failure diagnosis. We design and implement NotOnlyLog, a practical tool that mines the relationships between failure logs and fixing patches in order to reduce the number and length of uncertain execution paths, which we reconstruct based on SherLog [7].

In summary, our work makes the following contributions:

- Empirical studies on the relationships between the fixed bugs of several open-source software and the patches used to fix them . Our results show that the likelihood of similar bugs having similar patches is around 80%. This proves that our work can help to fix newly discovered bugs by identifying a similar bug that has been fixed already. We also find that 70% of faulty code is found to lie near the code where the first failure message is printed. This helps us locate bugs in a smaller range in code.
- A proactive tool to assist with failure diagnosis. We design and implement NotOnlyLog to extract the relationships between fixed bugs and the patches used to fix them, then use this relationship to help diagnose subsequent unfixed bugs. NotOnlyLog can also reproduce sufficient run-time logs from patches for each bug for relationship mining.
- Validation of the effectiveness of NotOnlyLog. NotOnlyLog can outperform the state-of-the-art work in many aspects. For example, it can achieve a reduction of 86.9% in the number of execution paths.

The rest of this paper is organized as follows. Section II summarizes the findings of the empirical studies. Section III presents the design overview and implementation details of NotOnlyLog. Section IV evaluates the tool's effectiveness and precision. Section V discusses the limitation of our method. Section VI discusses the threats to validity. Section VII presents the related work. Finally, we conclude this paper in Section VIII.

II. Motivation

Considering that specific bugs are often found to have corresponding run-time logs and patches, we believe that there may be some relationships between these logs and the patches used to fix the issues. To verify this assumption and to determine the relationships between run-time logs and their corresponding patches, we conduct empirical studies on software evolution histories in an attempt to answer the following Research Questions (RQ):

Table I
SUBJECT SOFTWARE."BUGS" MEANS THE TOTAL BUGS WE COLLECTED. "PATCHES" MEANS TOTAL PATCHES FOR BUGS."VERSIONS" MEANS HOW MANY VERSIONS OF SOFTWARE INVOLVED.

Software	Patches	Bugs	Versions	Logs
Squid	1803	1803	176	6006
MariaDB	4450	4450	140	1466
CUBRID	600	600	77	420
Httpd	2540	2600	18	103
OpenStack	78000	80003	18	103
OpenSSH	540	2586	78	161
Nginx	367	1028	80	200

- RQ1: How pervasive do bugs with similar logs have similar patches? It is common that bugs are fixed during software evolution. During the process of bug fixing, it is unavoidable that bugs with similar logs may appear at different periods. This research aims to gain a general insight into the relationships between patches for bugs with similar logs.
- RQ2: Where are these patches located in code? In bug diagnosis, reconstructing the execution paths from the whole log results in extra time overhead and a proliferation of useless information, which will finally lead to an explosion in the number of execution paths. In this question, we aim to study the distribution of patches in code so that we can reconstruct fewer and shorter execution paths.
- RQ3: Do we have sufficient run-time logs to mine relationships? To facilitate our process of bug diagnosis, we need run-time logs and corresponding patches for each bug. In this question, we need to study if we can collect sufficient run-time logs for each bug from bug tracking systems.

A. Experimental Setup

This empirical study is conducted on seven popular open-source software packages written in C/C++: namely, Squid , Httpd, OpenStack, OpenSSH, nginx, MariaDB and CUBRID. Each of these has a long history of development to ensure the system remains multifunctional and robust. Through use of these software packages, we can collect many bugs together with their patches.

We collect software release information, bugs and their corresponding patches from their bug-tracing systems, e.g., Bugzilla, Trac and JIRA, using a crawler based on Web-Magic [11], a popular crawler frame written in the Java language. The Versions column, below, shows the number of versions involved in this study. Naturally, the failure logs are submitted mixed with descriptions provided by customers; we need to extract these logs for our study. More detailed information is presented in Table I.

Table II
PERVASIVENESS OF SIMILAR MODIFICATIONS AMONG BUGS WITH SIMILAR LOGS

Software	Same variable	Same function	Different	total
Squid	17(85%)	0(0%)	3(15%)	20
MariaDB	28(61%)	12(26%)	6(13%)	46
CUBRID	21(67%)	6(19%)	4(14%)	31

B. RQ1:How Pervasive do Bugs with Similar Logs have Similar Patches?

To determine the pervasiveness of the phenomenon in which similar bugs have similar patches, we evaluate the proportion of bugs with similar logs and similar patches to bugs with similar logs.

What are the metrics for measuring similarity between two logs? According to previous works [12],1) Error messages need to be similar if two logs are similar. 2) The possibility is quite low that messages in one log are exact copies to that of another log without modifications. These observations indicate that error messages are critical for a log, since they address the potential error path, concrete values of identifiers and timestamps in messages might be different since they are in different executions.

According to the points above, we define similar logs as: Given two logs A and B, if the content of their error messages are same except for concrete values of identifiers, timestamps, which are different in different executions, log A and log B are similar. In this RQ, we select those messages contains keywords like "error" , "fail", "exception" and their synonyms from WordNet [13], which are usually used by developer to indicate system error, as error messages for logs.

For research purposes, we manually study 97 pairs of bugs with similar logs from three open-source projects. Table II presents more detailed data. On average, more than 80% of bugs with similar logs are found to have modifications to same variables or functions in their patches. For convenience, we say that these two patches are similar and give the definition as: Given two patches A and B, if the modified lines of A shares the same variable or functions to that of B, two patches are similar.

That is to say, if we know that same modifications have occurred in two bugs with similar logs, we may need to modify same variables or functions for bugs with similar logs in future. This result shows the potential effectiveness of mining for relationships between bugs with similar logs and their corresponding patches.

C. RQ2:Where are Patches Located in Code?

Considering that log messages can be treated as points in code, we may be able to identify a software execution path if those points are connected. In order to identify the general distribution of where these patches are located in code, we evaluate the distance between the patch and the point at

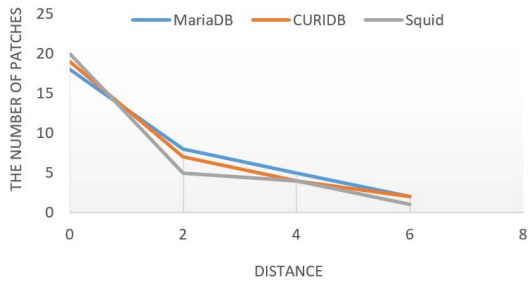

Figure 2. The distribution of patches.

which the first failure message emerges. Since patches are not only used to fix the corresponding bugs only, they may include adding new functions, modifications to code related to the bug-fixing code, we take the smallest distance to represent the whole patch.

For research purposes, we manually study 95 patches randomly selected from three large open-source projects. Results are presented in Figure 2. On average, for each project, more than 70% of patches are found to have a distance of less than two messages from the first failure message. And the closer to the first failure message, the higher the possibility of containing patches. That is to say, it is likely that we do not need to reconstruct execution paths from all messages, since the probability that the faulty code lies within a distance of two messages from the first failure message is high. This result indicates that we may only need to take several messages into consideration during the process of reconstructing execution paths.

D. RQ3: Do we have sufficient run-time logs to mine relationships?

To know whether we have sufficient run-time logs from bug tracking systems to mine relationships, we collected data from several open-source projects. As Table I shows, more than half of these software provide less than 200 run-time logs. Considering the variety of bugs in software [14], these run-time logs are insufficient for us to mine relationships.

We can also see from Table I that MariaDB provides 7 times more run-time logs than Nginx. To understand the reason of this phenomenon, we studied the bug submission instructions of these two software. As we can see in Table III, MairaDB provides several detailed instructions on what to submit when customers find a bug, including the failure log files, while Nginx only give one instruction without requiring log files. This may be the reason why Nginx has received less run-time logs than MariaDB.

III. Design and Implementation

A. Overview

In order to help bug diagnosis, we design and implement NotOnlyLog, a practical tool that mines the relationships

Table III
Instructions for Bug Submission

MariaDB	1.The environment (Operating system, hardware and MariaDB version) where the bug happened. 2.Any related errors or warnings from the server error log file. Normally it is hostname.err file 3.The content of your my.cnf file or alternatively the output from mysqld –print-defaults. .....
Nginx	Please describe the bug in full detail including all of the conditions that make it happen and include the relevant part of configuration as well. It would substantially reduce time and effort to understand and resolve the bug.

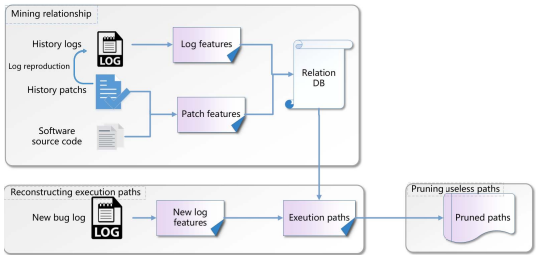

Figure 3. The architecture of NotOnlyLog

between failure logs and fixing patches to reduce the number of uncertain execution paths. These paths are reconstructed using a static context-sensitive analysis approach based on SherLog [7].

NotOnlyLog requires three inputs: (1) historical bugs and logs, which are used to mine relationships; (2) run-times logs of new bugs, which are used to match the relationships, in order to find culprits; (3) software source code. Consequently, NotOnlyLog outputs execution paths (pruned) which contain software faults.

As illustrated in Figure 3, NotOnlyLog consists of three modules:

- Mining relationships. This module analyzes run-time logs and corresponding patches for bugs to mine the relationships between them. In Section III-B, we will explain the relationships and further introduce the design and implementation details of this module.
- Reconstructing execution paths. This module reconstructs the execution paths from run-time logs. Detailed design and implementation will be provided in Section III-C.
- Pruning useless paths. This module applies the relationships mined in the first module to prune the execution paths reconstructed in the previous module. More details are provided in Section III-D.

NotOnlyLog mines the relationships between run-time logs and their corresponding patches in software evolution. When a new bug appears, NotOnlyLog reconstructs its likely execution paths and prunes them by applying the mined

relationships. In short, NotOnlyLog works as tool to help developers understand and locate bugs.

B. Mining Relationship

As noted in Section II-B, bugs with similar logs often have similar patches to source code. These patches may modify the same variables or functions. The objective of this section is to mine the relationship between run-time logs and their corresponding patches in software evolution.

To achieve this goal, we need to solve two problem: identifying similar logs (see Section III-B2) and extracting the fixing patterns of patches (see Section III-B3). After that, we mine relationships by extracting the similar patterns of patches for similar logs.

In addition, considering that bug-tracking systems do not always require users to submit run-time logs, we may be not able to mine enough relationships from insufficient run-time logs and patches. To remedy this situation, we use a control flow analysis to reproduce logs from patches (see Section III-B1).

1) Log Reproduction: To overcome this problem that we cannot get sufficient run-time logs from bug-tracking systems, this section aims to investigate the generation of logs using patches. As explained in Section II-C, once we treat the messages as points in code and connect these points to build an execution path, patches are often found to lie near the first failure message in code. Inspired by this phenomenon, we reproduce logs from the context of the patches in the following steps:

(1) For a given patch, we first locate its position in source code.

(2) Starting from the located position, we perform a control flow analysis, and we get the control flow tree. According to RQ2 that patches is found to have a distance of less than 2 to the first failure message, we limited the length of call-site to 2 while analyze the control flow.

(3) We operate a preorder traversal to the control flow tree of the patched code to find the potential logging functions in the tree that could be executed if the patched code was executed.

(4) We extract all functions with parameters that are literal strings as candidates of logging functions, since logging function have various definitions, especially wrapped function, and we do not always know which function prints the logs.

(5) We put strings of one function into a sentence in the order of their number of parameter. After that, we order those sentences in the order which they are extracted to get logs.

2) Similar Logs Identification: The objective of this section is to identify similar logs. Naturally, since the run-time log is only a testimony of the system execution, we use logs that consist of several lines of messages as an alternative to bugs. In order to verify that two logs are similar, we need to solve one problem at first. Normal messages in logs may interfere with the measurement of similar bugs, since normal messages appear in almost every logs.

To address the problem above, since hundreds of messages could be printed during a system run, it is necessary to pick out several representative messages for this bug. It is common that a large number of messages will be printed in both successful executions and failure execution conditions; therefore, this kind of message is useless while the two logs are being compared. Consequently, we require only the messages that are printed under failure execution conditions. Accordingly, we manually studied the logging functions in MariaDB and Squid, finding that these software printed the messages of interest in a different way from common messages (MariaDB uses functions with “error” or “failed” in their names to print uncommon log messages, while Squid uses different log levels). Based on this observation, we can identify the uncommon messages heuristically by recognizing their corresponding logging functions. Given the above, we only need to compare the uncommon messages in the two logs to judge whether or not they are similar.

In order to calculate whether two logs are similar, we need to: (1) extracting the error messages, which is explained above, (2) processing the extracted messages, (3) calculating whether two logs are similar.

Processing the extracted messages: (1) text normalizing, splitting camel cases in text and removing special symbols, (2) stop words removing, removing the stop words in English for Google stop-words list, (3) port stemming, reducing inflected words to their root words(i.e. transform “goes” into “go”).

Calculating whether two logs are similar: (1) use Term Frequency–Inverse Document Frequency(TF–IDF) to weigh all the words in vectors, (2) calculate the cosine similarity between every two logs. In order to get the threshold for cosine similarity, we manually select 20 pairs of logs that error messages of each pair are exact same(except for the values of identifiers and timestamps). We calculate their cosine similarity and pick 0.5 as the threshold to judge similarity of logs.

That is to say, two logs are similar if their cosine similarity is bigger than 0.5.

3) Patch Extraction: To mine the similarity of patches, as detailed in our empirical study in RQ2, we extract the functions and variables appearing in patches in this section. Bug patches are usually code snippets that are often inadequate and confusing in the absence of enough context to infer all the types of variables involved. As Figure 1 shows, we cannot directly obtain the type of “crypt_data” from the patch of “MDEV-13591”. A patch also contains the filename and the line numbers, which indicate its location in the source code. We can thus use a heuristic method to extract the information that indicates the location of a patch. After that, we can perform a static analysis on the context

```
1   UNIV_INTERN bool buf_page_io_complete(
        buf_page_t* bpage,bool evict){
2     ......
3     if (!buf_page_decrypt_after_read(bpage)){
4     /* encryption error! */
5       ......
6       goto corrupt;
7     }
8     if (buf_page_get_zip_size(bpage)) {
9       ......
10      if (uncompressed&& !buf_zip_decompress(
11        (buf_block_t*) bpage,FALSE)) {
12        buf_pool->n_pend_unzip--;
13        goto corrupt;
14      }
15      ......
16  }
17    ......
18    corrupt:
19    fprintf(stderr,"InnoDB: Database page
        corruption disk or a failed\n"
20      "InnoDB: file read of page %lu.\n"
21    ......
22  }
```

```
23  ibool buf_page_decrypt_after_read(
24  /buf_page_t* bpage)
      /*!< in/out: buffer page read from disk */
25  {
26    ......
27    /* decrypt from crypt_buf to dst_frame */
28    fil_space_decrypt(bpage->space,
29      slot->crypt_buf,
30      size,
31      dst_frame);
32      ......
33  }
34  bool fil_space_decrypt(
35    fil_space_crypt_t* crypt_data,
      /*!< in: crypt data */
36    const byte* src_frame,
      /*!< in: input buffer */
37    ulint page_size, /*!< in: page size */
38    byte* dst_frame) /*!< out: output buffer */
39  {
40    ......
41    fil_crypt_get_key(key, &key_length,
        crypt_data, key_version);
42    ......
43    return true;/* page was decrypted */
44  }
```

Figure 4. The source code of bug MDEV-8143 in buf0buf.cc

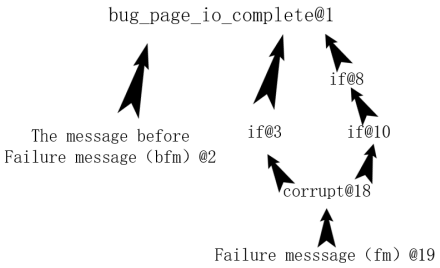

Figure 5. Reconstructed paths

of the patch code to infer the types of its code entities.

In more detail, we firstly employ a fuzzy analysis tool called srcML to build an AST (abstract syntax tree) for the patch and collect the code entities. Secondly, since patches are written in a standardized format, we can manually set a rule to extract the patch's location information, i.e., filename and line numbers. After that, we search the context of the lines in source code to fill in the type information missed in the first step. For the convenience of calculation, since variables and functions both have types and names, we record them in the following unified form:

$$<< Type1, Name1 >, < Type2, Name2 > \ldots > \quad (1)$$

If the extracting results of all the patches of similar logs have same items, we extract these items. We then express the relationship of the similar logs and the similar patches in the following form:

$$\{< message1, message2, \ldots >,$$
$$< Type1, Name1 >, < Type2, Name2 > \ldots\}$$

C. Reconstructing Execution Paths

In this section, we aim to reconstruct execution paths from the run-time log based on SherLog's method [7]. SherLog performs well for small programs (e.g. rm), but not for large systems, because it constructs hundreds of paths, and sometimes even has the problem of path explosion. It is still a tedious and tricky problem for maintainers to analyze these paths.

To solve this problem, we propose a method to effectively reduce the number of execution paths required to be reconstructed by restricting the search range. As noted in Section II-C, the faulty code usually lies near the code where the first failure message (i.e. the first message in the log printed by the failure message logging functions) is printed. Inspired by this, we propose a context-sensitive method based on that employed by SherLog in order to reconstruct fewer execution paths using reduced number of messages. Compared to SherLog, which uses all the messages in a log to reconstruct the execution paths, NotOnlyLog considers only three messages (the first failure message, along with a message before it and a message after according to the findings in Section II-C, since the closer to the first failure message, the higher the possibility of the existence of faulty code.) However, there are still some exceptional cases that we cannot get paths through 3 messages, we design a loop process to address them; this loop process will be explained in the next section.

As an example, we infer the paths in Figure 4 and present the results in Figure 5. Starting from the failure message and the message before, we conducted a bottom-up analysis and obtain two paths (the integer after "@" indicates the line number, "bfm" means the message "before failure message", and "fm" means "failure message" in Figure 5):

$$(1)Bug\_page\_io\_complete@1 \rightarrow bfm@2 \rightarrow if@3$$
$$\rightarrow corrupt@18 \rightarrow fm@19$$
$$(2)Bug\_page\_io\_complete@1 \rightarrow bfm@2$$
$$\rightarrow if@8 \rightarrow if@10 \rightarrow corrupt@18 \rightarrow fm@19$$

D. Pruning Useless Paths

In this section, we propose a method to reduce the number of execution paths reconstructed in Section III-C. The path-pruning process is accomplished by applying the relationships we have mined through the comparison of the run-time logs and patches from the software evolution history and obtain the code entities that has been modified to fix them. We first look for the similar bugs in the software evolution history. We believe that the bug to be fixed may also need to modify similar code entities. Thus, we perform a control analysis on each function in a path and scan the control flow to look for the relevant code entities. If these code entities cannot be found in the context of a path, this path is likely not to be the path containing the software bug, and we prune it. Finally, the paths left are more suspicious to contain faulty code.

However, we may find no path containing the code entities and we remove all the execution paths. To address this problem, we design a loop process that we add a message before

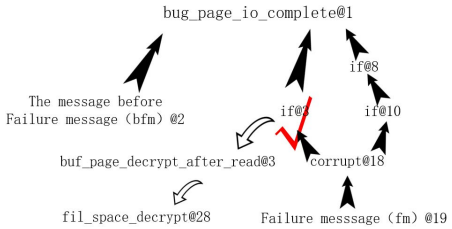

Figure 6. Pruned paths

the three messages and a message after and then reconstruct paths again. Once the paths containing the desired code entities are identified, the strategy stops, and the pruned paths are the final output of NotOnlyLog.

As shown in Figure 6, two paths have been reconstructed. To prune the constructed paths, we search the mined relationships for relationships with similar messages to bug "MDEV-8143". Consequently, we find a relationship, which is mined from "MDEV-13591" and "MDEV-8195" (the failure logs and patches of them is shown in Figure 1) contains the similar messages to the run-time log of bug "MDEV-8143". This relationship provides a variable "crypt_data" whose type is "fil_space_crypt_t" to help pruning the paths for bug "MDEV-8143".

As Figure 4 shows, the function "buf_page_decrypt_after_read" in the source code of MariaDB's bug "MDEV-8143" is called at line 3, while the function named "fil_space_decrypt" is called by "buf_page_decrypt_after_read" at line 28. Obviously, we can find the variable "crypt_data" in the body of function "fil_space_decrypt", and we cannot find "crypt_ data" in another path. Naturally, we then select the path containing "crypt_data" as the suspicious path. At this point, we have completed the path pruning process and there is one possible path left:

$$Bug\_page\_io\_complete@1 \rightarrow bfm@2 \rightarrow if@4 \rightarrow corrupt@18 \rightarrow fm@19$$

IV. Evaluation

In this section, we evaluate the performance of NotOnlyLog. Section IV-A evaluates the correctness of mined patch-log relationships. Section IV-B measures its effectiveness at pruning execution paths. Section IV-C evaluates the precision and effectiveness of its log reproduction work.

A. The Correctness of Mined Patch-Log Relationships

The correctness of mined patch-log relationships is essential to the precision of NotOnlyLog. In order to evaluate the correctness of mined patch-log relationships, we divide the collect data into two sets by their versions. We use the former versions as training set and the latter as test set. We mine relationships from one set (training set) and validity the mined relationships in the other set (test set). For each relationship mined from the training set, if all the bugs have similar messages to this relationship and contain the variables or functions in it, this relationship is correct. If there exists a bug, which has similar messages to this relationship, and contains no variable or function in it, this relationship is not correct. If no bug in the test set has similar messages to this mined relationship, the correctness is uncertain.

Table IV
THE CORRECTNESS OF THE MINED PATCH-LOG RELATIONSHIPS. "UNCERTAIN" DEPICTS THE NUMBER AND PROPORTION OF THE RELATIONSHIPS WHICH WE CANNOT VALIDATE.

Software	Relationships	Correct	Incorrect	Uncertain
MariaDB	73	39(53.4%)	5(6.8%)	29(39.8%)
Squid	36	16(44.4%)	2(5.5%)	18(50.1%)
CUBRID	30	10(33.3%)	2(6.7%)	18(60%)

Table IV displays the correctness of mined patch-log relationships. Because of the variety of bugs [14], we cannot find every similar bug in the test set, and thus cannot validate the correctness of every relationship. As for the relationships that we can validate, more than 80% of them are correct. To understand why 20% of these relationships are not correct, we manually study all of the incorrect relationships and the corresponding bugs. In many cases, the bugs from which we mined rules are fixed by modifying source files, while the bugs used to validity the correctness are fixed by modifying script files. We guess that the bugs may not be fixed thoroughly at the first time, and they need to fix the left problem later.

B. The Effectiveness of Path Pruning

The evaluation of NotOnlyLog and SherLog is on 150 real-world bugs from three real-world open-source projects (including two database systems and one proxy server). Our experiments are conducted on a Linux machine with eight Intel Xeon 2.33GHz CPUs, and 8GB of memory. In order to evaluate the effectiveness of NotOnlyLog, we conduct an experimental comparison between NotOnlyLog and SherLog on the number of reconstructed paths and time overhead. As the key technology, we evaluate the execution paths reconstructed by NotOnlyLog. We mine relationships from 6853 bugs(1803 from Squid, 4450 from MariaDB, 600 from CUBRID), and we get 378 relationships(98 for Squid, 198 for MariaDB, 82 for CUBRID).

There are 125 bugs of 150 bugs which have similar logs to the mined relationships. After we reconstruct paths and prune them, we manually examine whether the real patches are contained in those paths and the result is shown in TableVI. 85.6% of the patches could be found in paths. This result shows that we could prune the paths of SherLog and keep its function in bug diagnosis for most bugs. We

Table V
COMPARISON ON PERFORMANCE OF SHERLOG AND NOTONLYLOG. "MUST-PATH" INDICATES THE NUMBER OF EXECUTION PATHS WHICH MUST BE EXECUTED. "MAY-PATH" INDICATES THE NUMBER OF PATHS WHICH MAY BE EXECUTED. "AVG#LENGTH" INDICATES THE AVERAGE LENGTH OF PATHS (CALL SITE DEPTH) PRODUCED. "PATH-MID" INDICATES THE NUMBER OF PATHS RECONSTRUCTED DIRECTLY FROM LOG MESSAGES. "PATH-FINAL" INDICATES THE NUMBER OF PATHS AFTER PRUNING."M" MEANS MINUTES.

Version	Bugid	MSG	SherLog				NotOnlyLog			
			Must-path	May-path	Time	AVG#Length	Path-mid	Path-final	Time	Avg#Length
MariaDB-10.1.4	MDEV-8143	11	1	50	56.6m	13.5	4	2	31.2m	3
MariaDB-10.1.13	MDEV-9793	17	1	20	53.4m	10	2	2	25.6m	5
MariaDB-10.1.3	MDEV-7878	50	2	68	59.8m	9.5	2	2	23.6m	6
CUBRID-10.1.0.0026	CBRD-20300	3	1	2	40.2m	6	1	1	15.6m	3
CUBRID-10.1.0.0025	CBRD-20239	2	1	48	45.4m	11.8	1	1	13.4m	3
CUBRID-10.1.0.7545	CBRD-21203	3	1	134	48.6m	12.6	11	5	28.4m	4
Squid-3.5.23	Squid-4004	48	2	88	32.2m	18.7	15	15	28.2m	10.5
Squid-3.0.PRE5	Squid-1940	7	2	176	26.8m	19.3	10	10	20.6m	10
Squid-3.2.0.11	Squid-3329	28	2	253	35.4m	17.9	72	72	42.2m	13.7

Table VI
THE CORRECTNESS OF NOTONLYLOG. "PATCH IN PATHS" MEANS NUMBER OF BUGS WHOSE REAL PATCHES ARE CONTAINED IN THE PATHS PRODUCED BY NOTONLYLOG, "PATCH NOT IN PATHS" MEANS NUMBER OF BUGS WHOSE REAL PATCHES ARE NOT CONTAINED IN THE PATHS PRODUCED BY NOTONLYLOG

	Patch in paths	Patch not in paths	total
Number	107(85.6%)	18(14.4%)	125

manually study why 14.4% of patches are not contained in the pruned paths. 10 of them are not contained in paths because of the long distance between their patches and their first error messages. We will discuss how to fix this problem in V. Besides, another 4 of them are not contained in paths, because the paths which contains patches are pruned by NotOnlyLog. And this will be discussed in Section V. The left 6 patches are not contained in paths because their patches are not modifications to source code (i.e. script files), which SherLog is not able to fix either.

Limited by the space, we select 9 bugs from the 107 bugs above to perform a deeper analysis. Table V displays the performance comparison of SherLog and NotOnlyLog on execution path reconstruction of 9 bugs. Overall, NotOnlyLog produces fewer and shorter paths than SherLog. On average, NotOnlyLog can decrease the number of execution paths produced by SherLog by 86.9%.

Moreover, the average length of the reconstructed execution paths generated by NotOnlyLog are basically half of that generated by SherLog. Since developers may need to scan every path suggested to complete diagnosis, the shorter execution path created by NotOnlyLog can reduce the diagnosing effort.

C. Evalution on Log Reproduction in NotOnlyLog

This section evaluates the precision and effectiveness of NotOnlyLog's log reproduction work.

1) The Precision of Log Reproduction: We randomly select 120 bugs with logs submitted by customers from Squid, MariaDB, and CUBRID. We then manually identify the precision of the reproduced logs. Table VII presents

Table VII
THE PRECISION OF LOG REPRODUCTION

Software	Bugs(message found)	Bugs(total)	Precision
Squid	29	40	72.5%
MariaDB	27	40	67.5%
CUBRID	31	40	77.5%

Table VIII
THE EFFECTIVENESS OF LOG REPRODUCTION. "BUGS+" MEANS NUMBER OF BUGS FOR WHICH WE REPRODUCE LOGS, "RULES+" MEANS THE NUMBER OF NEW RULES MINED AFTER WE REPRODUCE LOGS

Software	Bugs	Rules	Bugs+	Rules+	EF
MariaDB	1466	198	100	9	4.5%
Squid	600	98	100	13	13.3%
CUBRID	420	82	100	15	18.3%

that failure messages can be reproduced for more than 65% of these bugs. This result indicates that we can reproduce log messages for a majority of fixed bugs with patches and that we can mine more relationships if we reproduce log messages for some of the bugs with no log messages submitted in bug tracking systems.

2) The Effectiveness of Log Reproduction: We first mine rules from the bugs with run-time logs submitted by customers. Then, we reproduce logs for 100 random bugs without submitted logs for each software. After that, we mine rules from all the bugs. We use Effectiveness(EF) to evaluate the effectiveness of log reproduction of NotOnlyLog.

$$EF = \frac{new\ rules\ mined\ after\ log\ reproduction}{rules\ mined\ before\ log\ reproduction}$$

Table VIII displays the EF of NotOnlyLog's log reproduction work on three software. For each software, the number of rules increases after log reproduction. We also find that the effectiveness of log reproduction is more significant on the software with smaller number of bugs with run-time logs.

V. DISSCUSION

In Section IV-B, we analyzed the result of NotOnlyLog find three problems that cause the imprecision of NotOnlyLog. They are: 1) The distance between patches of some

bugs and their first error message is longer than 2, and NotOnlyLog missed the real patches while reconstructing paths. 2) The bugs with similar logs to the mined relationships does not contain the corresponding variables or functions, and the paths, which real patches lie, are pruned by NotOnlyLog. 3) The patches of some bugs are modifications to files that are not source files (may be script files).

To fix the first problem, we need to expand the messages, which are used to reconstruct paths. In addition, this will inevitably increase the length and number of the reconstructed paths. Fortunately, average number of paths reconstructed by NotOnlyLog is 20, and that is not too much work for developers to check. We add an interface in NotOnlyLog for developers to expand the messages used to reconstruct paths, once they could not find the root causes of the bugs. This problem is solved after we expand the messages used to reconstructed paths in our experiment.

To fix the second problem, we provide the developers with both the paths before and after pruning. Since the paths before pruning are the reconstructed by NotOnlyLog already. There is no extra time overhead. And this problem is fixed after we provide both the paths before and after pruning.

As for the last problem, we are not able to fix it at present, since NotOnlyNot is not able to analysis script files. We will continue to figure out how to fix this in the future.

VI. THREATS TO VALIDITY

In this section, we will discuss the threats to the validity of NotOnlyLog.

The Variety of Bugs. NotOnlyLog is based on the relationships between similar bugs and their corresponding patches, which are extracted from historical data. From our empirical study, we find that 20% of bugs are similar, which means NotOnlyLog cannot prune execution paths for every bug. The performance of NotOnlyLog may degrade when fixing bugs without similar bugs detected previously, compared with the bugs whose similar bugs have been recorded. However, as we reported in Section IV-B, the length of execution paths generated by NotOnlyLog is shorter than SherLog, the overall performance of NotOnlyLog can be still better than the state-of-the-art work.

Limitation of Subject Software. Both our empirical studies and evaluation experiments have been conducted on three open-source systems. Each of them has a long period of development in its corresponding application field. However, as we were limited by the complicity of implementation of NotOnlyLog, we have analyzed only C/C++-based systems, causing uncertainty regarding the effectiveness of our tool for other programming languages (e.g. Python and Java). We plan to implement our tool on more extensive types of systems to overcome this limitation in the future.

VII. RELATED WORKS

Log-based Bug Diagnosis. Most log-based studies are based on rules or execution paths. Rule-based works [2], [4], [5] usually aim at generating heuristic rules and use them to help with bug diagnosis. Logsurfer [5] uses human-formulated rules with strong domain knowledge to diagnose software failures. Distalyzer [3] uses machine learning techniques to generate rules extracted from run-time logs to compare system behaviors, in order to reveal which component of the software is broken. Execution-path-based work [7], [8] uses a variety of algorithms to construct execution paths. As a representative study, SherLog [7] performs a static, context-sensitive approach to reconstruct the failure execution paths by analyzing both run-time logs and source code. NotOnlyLog is a execution-path-based method that can improve the efficiency of bug diagnosis.

Fault Localization. Fault localization work typically takes a faulty program as input and then produces a list of code locations ranked by the possibility of the occurrence of faults. A fault localization tool may use spectrum-based or mutation-based techniques. Spectrum-based techniques usually use test cases to locate faulty statements [15]–[19]. Op2 [19] runs test cases on the target program and ranks these statements by the ratio of failed test cases to passed ones. However, it may be unrealistic to build those test cases as the scale of systems is growing rapidly. Mutation-based techniques use "mutants" to locate faults [20], [21]. MUSE [21], on a given test case, mutates statements and select the mutants that change the state of the testing results. However, mutation-based fault localization also relies on test cases that may lead to uncertain performance. NotOnlyLog does not rely on test cases and provides a deeper insight into failures.

Log Analysis and Enhancement. Due to the trade-off between logging mechanism and run-time overhead, it is impossible for developers to print all system states at every moment. Many research studies have been proposed to improve the expressiveness of logs. Some of them concentrate on how to enrich log content [22], [23], and some focus on improving the effectiveness of logs by ensuring that log statements are properly placed [24]–[26]. SmartLog [24] performs an intention-ware study and learns from the advanced logging experiences to guide its placement of log statements. And LogEnhancer [23] identifies important variables and uses them to update existing log statements. NotOnlyLog analyzes run-time logs, so its performance will improve with the enhancement of logging mechanisms.

VIII. CONCLUSION

Previous studies that reconstruct execution paths to assist in bug diagnosis may lead to path explosion. Inspired by the finding that 80% of bugs with similar logs have similar patches and that 70% of faulty code is found to lie near the code where the first failure message is printed, we design and implement a tool called NotOnlyLog. It is a practical tool that mines the relationships between failure logs and their patches to reduce the number of uncertain execution paths,

which are reconstructed by a static context-sensitive analytic approach. NotOnlyLog can achieve a 86.9% reduction in the number of execution paths compared with SherLog.

ACKNOWLEDGMENT

The work described in this paper was supported by National Natural Science Foundation of China (Project No.61872373, 61690203, U1711261 and 61872375).

REFERENCES

[1] D. Yuan, S. Park, and Y. Zhou, "Characterizing logging practices in open-source software," in *International Conference on Software Engineering*, 2012, pp. 102–112.

[2] J. L. Hellerstein, S. Ma, and C.-S. Perng, "Discovering actionable patterns in event data," *IBM Systems Journal*, vol. 41, no. 3, pp. 475–493, 2002.

[3] K. Nagaraj, J. Neville, and C. Killian, "Structured comparative analysis of systems logs to diagnose performance problems," 2011.

[4] S. Ma and J. L. Hellerstein, "Mining partially periodic event patterns with unknown periods," in *International Conference on Data Engineering, 2001. Proceedings*, 2001, pp. 205–214.

[5] J. E. Prewett, "Analyzing cluster log files using logsurfer," *Proc.annual Conf.on Linux Clusters*, 2003.

[6] J. G. Lou, Q. Fu, S. Yang, Y. Xu, and J. Li, "Mining invariants from console logs for system problem detection," *Proc of Usenix Atc*, pp. 231–244, 2010.

[7] D. Yuan, H. Mai, W. Xiong, L. Tan, S. Pasupathy, and S. Pasupathy, "Sherlog: error diagnosis by connecting clues from runtime logs," in *Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, 2010, pp. 143–154.

[8] Y. Zhang, S. Makarov, X. Ren, D. Lion, and D. Yuan, "Pensieve: Non-intrusive failure reproduction for distributed systems using the event chaining approach," in *The Symposium*, 2017, pp. 19–33.

[9] Q. Fu, J. G. Lou, Q. Lin, R. Ding, D. Zhang, and T. Xie, "Contextual analysis of program logs for understanding system behaviors," in *Working Conference on Mining Software Repositories*, 2013, pp. 397–400.

[10] MariaDB, "Mariadb. retrieved may 16 from http://mariadb.org/about/," 2018.

[11] WebMagic, "Webmagic in action. retrieved may 16 from https://trac.edgewall.org/," 2018.

[12] X. W. Foyzul Hassan, "Hirebuild: an automatic approach to history-driven repair of build scripts," in *International Conference on Software Engineering, 2018. Proceedings*, 2018, pp. 1078–1089.

[13] WordNet, "Wordnet. retrieved may 16 from http://wordnetweb.princeton.edu/," 2018.

[14] L. Lu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Lu, "A study of linux file system evolution," *Acm Transactions on Storage*, vol. 10, no. 1, pp. págs. 10–17, 2013.

[15] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Ieee/acm International Conference on Automated Software Engineering*, 2005, pp. 273–282.

[16] A. Rui, P. Zoeteweij, and A. J. C. V. Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and Industrial Conference Practice and Research Techniques - Mutation*, 2007, pp. 89–98.

[17] W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The dstar method for effective software fault localization," *IEEE Transactions on Reliability*, vol. 63, no. 1, pp. 290–308, 2014.

[18] A. Rui, P. Zoeteweij, and A. J. C. V. Gemund, "Spectrum-based multiple fault localization," in *Ieee/acm International Conference on Automated Software Engineering*, 2010, pp. 88–99.

[19] L. Naish, J. L. Hua, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *Acm Transactions on Software Engineering Methodology*, vol. 20, no. 3, pp. 1–32, 2011.

[20] M. Papadakis and Y. L. Traon, *Metallaxis-FL: mutation-based fault localization*. John Wiley and Sons Ltd., 2015.

[21] S. Moon, Y. Kim, M. Kim, and S. Yoo, "Ask the mutants: Mutating faulty programs for fault localization," in *IEEE Seventh International Conference on Software Testing, Verification and Validation*, 2014, pp. 153–162.

[22] B. Chen, "Characterizing and improving logging practices in java-based open source software projects - a large-scale case study in apache software foundation," 2018.

[23] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving software diagnosability via log enhancement," in *Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011, pp. 3–14.

[24] Z. Jia, S. Li, X. Liu, X. Liao, and Y. Liu, "Smartlog: Place error log statement by deep understanding of log intention," in *IEEE International Conference on Software Analysis, Evolution and Reengineering*, 2018, pp. 61–71.

[25] D. Yuan, S. Park, P. Huang, Y. Liu, M. M. Lee, X. Tang, Y. Zhou, and S. Savage, "Be conservative: enhancing failure diagnosis with proactive logging," in *Usenix Conference on Operating Systems Design and Implementation*, 2012, pp. 293–306.

[26] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang, "Learning to log: helping developers make informed logging decisions," in *Ieee/acm IEEE International Conference on Software Engineering*, 2015, pp. 415–425.