SMARTLOG: Place Error Log Statement by Deep Understanding of Log Intention

Zhouyang Jia, Shanshan Li, Xiaodong Liu and Xiangke Liao
College of Computer Science
National University of Defense Technology, Changsha, China
{jiazhouyang, shanshanli, liuxiaodong, xkliao}@nudt.edu.cn

Yunhuai Liu
Big Data Research Center
Peking University, Beijing, China
yunhuai.liu@pku.edu.cn

***Abstract*—Failure-diagnosis logs can dramatically reduce the system recovery time when software systems fail. Log automation tools can assist developers to write high quality log code. In traditional designs of log automation tools, they define log placement rules by extracting syntax features or summarizing code patterns. These approaches are, however, limited since the log placements are far beyond those rules but are according to the intention of software code. To overcome these limitations, we design and implement SmartLog, an intention-aware log automation tool. To describe the intention of log statements, we propose the Intention Description Model (IDM). SmartLog then explores the intention of existing logs and mines log rules from equivalent intentions. We conduct the experiments based on 6 real-world open-source projects. Experimental results show that SmartLog improves the accuracy of log placement by 43% and 16% compared with two state-of-the-art works. For 86 real-world patches aimed to add logs, 57% of them can be covered by SmartLog, while the overhead of all additional logs is less than 1%.**

I. Introduction

Logs, especially those reporting errors, are essential for system running and operation [1]. High-quality error logs can greatly ease the failure diagnosis and reduce the system recovery time, and thus different log handling technologies are proposed such as log enhancement [2], log characterization [3], [4], [5] and postmortem analysis [6], [7], [8], [9]. Among them, log automation attracts intensive research interests [1], [10] due to its great impact on the effectiveness of the others. It helps the developers write high-quality log statements, based on which log enhancement and postmortem analysis are conducted.

Log automation is often encapsulated as a log automation tool embedded to a software development environment, e.g., Errlog [1] in Eclipse. During development, the log automation tool scans the code and reminds developers to place log statements. In this paper, we focus on the failure-diagnosis log statement (e.g., $printf(\text{“}Memory\ page\ fault\text{”})$), also referred as error log statement (ELS). Throughout the paper, we overload the term ELS to mean either a set of error log statements or a single error log statement — the actual meaning should be evident from the context. As the content of ELS can be improved by log enhancement work [2], the fundamental issue in the design of log automation tools is to make log decisions for code snippets, i.e., for each given code snippet, whether an ELS is needed. This is also referred as where-to-log or log placement problem in literature [10].

For practical concerns, logs should not be too few or too many. On one hand, logging too little may miss critical runtime information that negatively affects the postmortem analysis [1]. On the other hand, too many logs consume additional system resources at runtime [11], and degrade the system performance [12]. For these reasons, strategic log placement schemes desire to record valuable information yet not to induce too much overhead. This can be measured by the precision and recall rates. A high precision rate means that most of the logs are useful, and a high recall rate means that most of places that need logs are logged.

Existing ELS placement strategies are based on the syntax features or code patterns of the software source code. For example, pattern-based approaches (e.g., Errlog [1]) identify code patterns from existing codes and insert logs. On the other hand, feature-based tools (e.g., LogAdvisor [10]) extract syntax features and apply machine learning techniques to make log decisions. The ELS is often placed under a certain check condition, which is often a branch statement, e.g., if or $switch$. We refer to the check condition (including the related variables and function calls) as $log\ context$, and the semantics of log context as $log\ intention$. A better ELS placement strategy should take the log intention into consideration. Notice that we avoid to use the terminology $log\ semantics$, since it may refer to the semantics of log statement itself, which introduces ambiguity.

To reveal the limitations of existing tools, we conduct comprehensive experiments based on 6 mature open-source projects. Errlog's overall precision is 72% (i.e., 72% of placed logs are useful for failure diagnosis), while its recall rate is 16%, meaning that 84% of the places that require ELS are missed. LogAdvisor produces 32% recall rate, while its precision rate decreases to 62%. When considering log intention, the recall rate can be improved to 58% while the precision rate can achieve 84%. These gaps motivate us to design and implement SmartLog, an intention-aware log automation tool. More details of the experimental results are in Section II.

SmartLog design encounters several practical challenges. First, software code is not simple texts but contain complex logics such as loops, conditions, and calls. These logics reflect the intention of the program. It is quite hard to determine whether an ELS is needed before we deeply understand the program intention. Second, given the intention of programs,

978-1-5386-4969-5/18/$31.00 © 2018 IEEE

SANER 2018, Campobasso, Italy
Technical Research Papers
Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on September 09,2023 at 01:42:18 UTC from IEEE Xplore. Restrictions apply.

it is not enough to make log decision either. There is not yet common knowledge or general agreement about under which intention an ELS is needed. Developer's experiences can help, while they are not always consistent either.

To overcome these challenges, we employ several instrumental techniques. We augment a traditional keyword-based approach with a filtering algorithm to accurately recognize ELS in existing projects. We then apply program dependence analysis techniques to obtain log intentions, and express them by a newly designed Intention Description Model (IDM). Each IDM instance expresses the intention of an ELS. The IDM instances are further normalized using Generalized IDM (GIDM) to evaluate the equivalence across different log intentions. The Aprior data mining algorithm [13] is applied on GIDM instances to mine the ELS placement rules, and these rules will finally forge the SmartLog. To summarize, our main contributions are as follows:

- We conduct a large amount of real-world experiments to reveal the limitations of exiting works. We also give comprehensive analysis on the root cause why these works are limited. They are mainly based on the syntax level analysis of software code, without deep understanding of the intentions of the program.
- To accurately describe the log intention, we propose two models, the IDM and GIDM. To the best of our knowledge, GIDM is the first in literature that can be used universally and generically evaluate the semantic equivalence across different log contexts.
- Based on IDM and GIDM, we design and implement SmartLog. Compared with the state-of-the-art tools Errlog and LogAdvisor, SmartLog can improve the F_{score} by 43% and 16%. For 86 real-world patches aimed to add logs, 57% of them can be covered by SmartLog, while the overhead is less than 1%.

The remainder of this paper is organized as follows. We present the limitations of the existing works in Section II. In Section III, we give the basic models. The SmartLog design will be in Section IV, and Section V will be our experiment evaluations. We present threats to validity and the related work in Section VI and VII, and finally conclude our work in Section VIII.

II. Motivation

In this section, we evaluate the precision and recall rates of the state-of-the-art log automation tools based on the experiments on the six mature open-source projects. We present the results of these tools, and then give analysis on these results and reveal the rationale behind the results, which motivates us to design and implement SmartLog.

Our evaluation is based on six projects, namely Apache Httpd [14], Subversion [15], MySQL [16], PostgreSQL [17], GIMP [18] and Wireshark [19]. All the projects are with long development history and keep active. Among these projects, Httpd is the main daemon program for Apache HTTP servers, Subversion is a version control software, MySQL and PostgreSQL are two database management systems, GIMP is a cross-platform image editor, and Wireshark is a network packet analysis software. These six projects cover network, server, software development, multimedia processing, and database managements. We believe they can provide sufficient generality for our investigations.

TABLE I
Evaluation of recall rate

Project	Errlog	LogAdvisor	Semantic Eq.
Httpd	5.3/50 (11%)	19.5/50 (39%)	30.7/50 (61%)
Subversion	4.2/50 (8%)	23.1/50 (46%)	36.4/50 (73%)
MySQL	10.5/50 (21%)	6.0/50 (12%)	24.1/50 (48%)
PostgreSQL	9.6/50 (19%)	14.3/50 (29%)	33.3/50 (67%)
GIMP	8.0/50 (16%)	20.2/50 (40%)	26.7/50 (53%)
Wireshark	10.4/50 (21%)	13.7/50 (27%)	23.0/50 (46%)
Total	48.0/300 (16%)	96.8/300 (32%)	174.2/300 (58%)

We employ LogAdvisor and Errlog as the representatives of the state-of-the-art approaches since they are widely advocated. $LogAdvisor$ is a feature-based log automation tool. It identifies and selects a number of syntax features from the software code, and then applies machine learning and noise handling techniques to make log decisions. The syntax features include the structural features, textual features and syntactic features. $Errlog$ is a pattern-based tool. It provides a set of error patterns and applies the patterns to the source code. Logs will be placed when matching any patterns, e.g., function return errors, exception signals and unexpected cases.

A. Evaluation of Recall Rate

In this part, we evaluate the recall rate of LogAdvisor and Errlog. Since there is no general agreement about whether an ELS is needed, one practical approximation to ground truth is the advanced developers' experiences and logging behaviors in software projects with high code quality and long evolution history.

For each project, we randomly select 100 function-call statements which are followed by ELS. We therefore have 600 function calls over 6 projects[1]. Half of them are used as the training set and the others are used as the test set. Since different approaches will be fairly evaluated on the same training and test sets, 600 samples are enough for evaluating the accuracy gap between different approaches. For Errlog, we apply its code patterns for test set directly since it does not require any training. We remove the ELS from the test set and check how many of them can be recovered by LogAdvisor and Errlog (we simulate LogAdvisor and Errlog according to their papers). We independently run the experiments 10 times (each time with a different training set) and compute the averaged recall rate R. Table I shows LogAdvisor logging 19.5 calls for Httpd on average, and the overall recall rate is 32%. While recall rate of Errlog is 16%.

As LogAdvisor is based on syntax features of software code, its performance is mainly due to the syntax level code

[1]In this paper, the datasets and source code are avalible in https://github.com/ZhouyangJia/SmartLog.

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on September 09,2023 at 01:42:18 UTC from IEEE Xplore. Restrictions apply.

```
/*Example 1:httpd/.../mod_proxy_ftp.c*/
switch (proxy_ftp_command(...)) {
    case -1:
    case 421:
    case 550:
        ap_proxyerror(..."Failed to ...");
    break;
}
```

```
/*Example 2: httpd/.../mod_proxy_ftp.c*/
rc = proxy_ftp_command(...);
if (rc == -1 || rc == 421) {
    return ftp_proxyerror("Error ...");
}
if (rc == 550) {
    ap_log_rerror(...,"RETR failed...");
}
```

Figure 1. Example of semantic equivalent code snippets w.r.t. log placement.

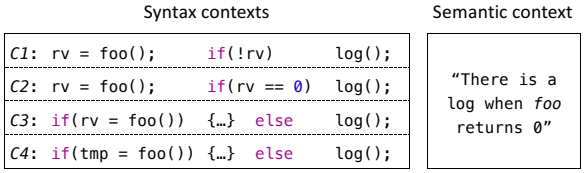


Figure 2. Syntax contexts and the equivalent semantic form.

analysis. Consider a real example in Figure 1. The function $proxy\_ftp\_command$ should be logged under different return values. In Example 1, a log will be triggered when the return value equals to -1, 421 or 550. In Example 2, when rv equals to -1, 421 or 550, there is a log too. The semantics of these two log contexts are equivalent. But they are different in syntax, sharing few common structural or textural except for the function name. Traditional feature-based tools can hardly make a right log decision on Example 2 by simply learning Example 1.

For Errlog, only code snippets following several pre-defined code patterns will be logged. The shortcomings of these pattern-based rules are clear, i.e., many logs are missing.

Impacts of log-context semantics. Log contexts with syntax equivalence can lead to semantic equivalence, but not always the reverse. One semantic context can have different implementations, i.e., different syntax contexts. For the example in Figure 2, a context semantics "There is a log when foo returns 0" has four different implementations from $C1$ to $C4$.

The context equivalence of different error log statements can be classified into three categories. (i) Syntax equivalent. The contexts are exactly the same except some white spaces and blank lines. (ii) Syntax non-equivalent but semantic equivalent. It means that the contexts have the same behaviors in terms of log placement when given the same input. It happens when the contexts are with different variables ($C3$ and $C4$), different check conditions ($C1$ and $C2$), or different branches ($C2$ and $C3$). (iii) Semantic non-equivalent.

Log automation tools with semantic-equivalence identification capability can dramatically increase the recall rate. As shown in Table I, we find the overall R can be improved to 58% through manual analysis.

Limitations of pattern-based rules. To further understand the limitation of code patterns employed by Errlog, we conduct more investigation on existing ELS. The existing ELS are often

```
/*Example 1: postgresql-9.3.5/.../dml.c*/
if (!attname)
    elog(ERROR, "cache lookup failed ...",...);
attno = get_attnum(childId, attname);
if (attno == InvalidAttrNumber)
    elog(ERROR, "cache lookup failed ...",...);
```

```
/*Example 2: gimp/.../dialog.c*/
gtk_ui_manager_add_ui_from_string(..., &error);
if (error){
    g_warning ("error ...);
    g_clear_error (&error);
}
```

Figure 3. Example of different log types.

used in exception or assert mechanism. For example, a C# project may put an ELS in a catch block. These exception logs are crucial for recording certain errors and have been well handled by Errlog and existing work [10].

Logging for function call is another important logging type, and these logs are much more flexible and harder to learn. We study these logs and find that they can be classified into three types. (i) Log for sensitive argument. In Figure 3 Example 1, function get_attnum is sensitive for its argument $attname$, which should be logged when being empty. (ii) Log for return value. In Example 1, the return value (i.e., $attno$) decides whether or not a log is needed. (iii) Log for pointer argument. In Example 2, logs are placed based on $error$, a pointer argument, which may be assigned in the callee function. These logs, however, are hard to be covered by Errlog patterns.

B. Evaluation of Precision Rate

In this part, we evaluate the precision of LogAdvisor and Errlog. For failure diagnosis, a logging tool is supposed to insert ELS when error happens. Beside ELS, programs also need normal log statements (NLS) such as showing results, audit and debugging. Showing results is a normal function of the program, audit is used for monitoring purposes, and debugging is for developing. In our experiments, we implement the keyword method [10] to identify ELS, as suggested by LogAdvisor. The basic idea of keyword method is that a function call is a log statement if and only if its function name contains keywords log/$logging$, $trace$ or $write$/$writeline$.

In each project, we randomly select 100 functions calls (i.e., 600 for 6 projects), and each of them has an ELS or NLS. We equally split these 600 calls into two sets, one is for training and the other is for test. We remove all ELS from test set and check whether LogAdvisor or Errlog can place them back. We compare it with the total log statements that the two tools placed and the ratio is the precision rate.

We independently run the experiment 10 times (each time with a different training set) and calculate the averaged precision rate. The results are shown in Table II. Errlog performs quite well in precision. For Httpd, Errlog places 8.2 log statements and 7.2 of them are ELS, thus the precision rate is 88%. We believe this is mainly due to the effective patterns employed by Errlog.

In contrast, LogAdvisor places 28.0 log statements in Httpd, while only 14.9 are ELS. The precision rate over six projects is 62%. We believe such performance is mainly due to

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on September 09,2023 at 01:42:18 UTC from IEEE Xplore. Restrictions apply.

TABLE II
EVALUATION OF PRECISION RATE

Project	Errlog	LogAdvisor	Error Log Stmt.
Httpd	7.2/8.2 (88%)	14.9/28.0 (53%)	13.7/18.1 (76%)
Subversion	7.5/9.0 (83%)	16.9/30.2 (56%)	13.8/16.8 (82%)
MySQL	2.4/6.3 (38%)	7.4/24.0 (31%)	6.7/8.8 (76%)
PostgreSQL	9.4/10.7 (88%)	23.1/25.3 (91%)	21.9/23.0 (95%)
GIMP	1.5/1.9 (79%)	18.1/25.5 (71%)	17.5/19.8 (88%)
Wireshark	4.5/9.0 (50%)	19.1/26.5 (72%)	15.0/18.5 (81%)
Total	32.5/45.1(72%)	99.5/159.5(62%)	88.6/105 (84%)

the keyword method. LogAdvisor employs machine learning techniques, and therefore its performance highly depends on the accuracy of the labels in the training set. When the keyword method is applied, many of the labelled statements are incorrect, and further mislead the learning process.

Impacts of recognizing ELS. In practice, accurate and effective recognition of ELS is not a trivial task. In large-scale software, there could be hundreds of logging functions. Manually inputting all of them will be practically infeasible. The problem becomes harder in multi-developer projects. Even an experienced developer will take plenty of time to identify all logging functions written by others. Additionally, a logging function can be used as both ELS and NLS. For example, a logging function $printf$ can be used as $printf(\text{“}Memory\ page\ fault\text{”})$ or $printf(\text{“}Hello\ World\text{”})$. The former is an ELS while the latter is a NLS. This nature may generate a lot of false positive labels by simple keyword method.

Log automation tools with the capability of recognizing ELS can improve the precision rate. As shown in Table II, with consideration on recognition of ELS, we find the overall precision rate can be improved to 84% through manual analysis.

III. PRELIMINARY AND MODELS

In this section, we first give some preliminary information of SmartLog, and then propose an Intention Description Model, which can help us capture the log intention. This model will be normalized to a Generalized Intention Description Model (GIDM) which is more efficient to evaluate the semantic equivalence across log contexts.

A. Preliminary

SmartLog learns from the real-world projects representing how the experienced developers write ELS. From their experiences, SmartLog answers the simple yet critical question, “For any given code snippet, whether an error log statement is needed, and if so, under which condition?"

Towards this goal, we are facing several key challenges. First, there is no simple method to accurately recognize ELS. To address this issue, we design a two-phase ELS recognition algorithm in Section IV-B. Second, logs are placed according to the semantics of their contexts. Given similar context semantics, however, there are various ways to implement it. For this we propose two models, IDM and GIDM, to evaluate the equivalence of context semantics in Section IV-C and Section IV-D. Third, given the semantics of a log context, it is still not trivial to make the log decision. Intuitive log placement rules may not be efficient in practice. For example, in network software, developers sometimes do not log even for error-prone functions in the light of performance concerns. To solve this problem, we employ data mining techniques in Section IV-E.

B. Intention Description Model

Log intention is mainly determined by four factors. First, the ELS is supposed to log for an error, which is often, if not all the times, triggered by a function, e.g., $fopen$, new, or $malloc$. We call such functions as error-prone functions. Second, ELS placements are often controlled by certain variables, which are dependent on the error-prone function. These variables are often checked by a branch statement before the log decision is made. The branch condition is the second major factor. Third, for sensitive arguments, developers usually check them before the invocations of the error-prone functions. On the other hand, the check of pointer arguments and return values will be after the invocations. The position of check condition is the third factor. The fourth factor is the log decision, i.e., whether the log statement is placed under such circumstances. Keeping the above thoughts in mind, we have Intention Description Model.

Definition Intention Description Model (IDM) is a 4-tuple $m =< f, c, B/A, Y/N >$ where f is the error-prone function name, c is the check condition which contains a Boolean formula, B/A is the position of the check condition where B for before and A for after, and Y/N is the label of the log decision where Y for yes and N for no log.

An error-prone function may introduce several IDM instances. It happens when the arguments or return values of the error-prone function are checked in different branch statements. In contrast, each IDM instance is associated with one branch statement only. For example, the Figure 3 Example 1 can be described with two instances,

$m_1 =< \text{get\_attnum}, \neg\text{attname}, \text{B}, \text{Y} >$
$m_2 =< \text{get\_attnum}, \text{attno==InvalidAttrNumber}, \text{A}, \text{Y} >$

In Section IV-C, we will show how to extract all the IDM instances.

C. Generalized Intention Description Model

The IDM is able to describe the log intentions. It is, however, limited by its complex check condition c. We desire an atomic, unique, ubiquitous and exclusive model, which will be more general to describe the intentions.

Definition Generalized Intention Description Model (GIDM) is a 4-tuple $\hat{m} =< f, \hat{c}, B/A, Y/N >$, where f is an error-prone function name, B/A is the position of check condition, Y/N is a label to record the log decision, and $\hat{c}$ is a Boolean formula that can only contain two logical operators,

- $\neg$: expressions and their negative forms, e.g., a and $\neg a$;
- $\wedge$: conjunctions of expressions and their negative forms, e.g., $a \wedge \neg b \wedge d$.

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on September 09,2023 at 01:42:18 UTC from IEEE Xplore. Restrictions apply.

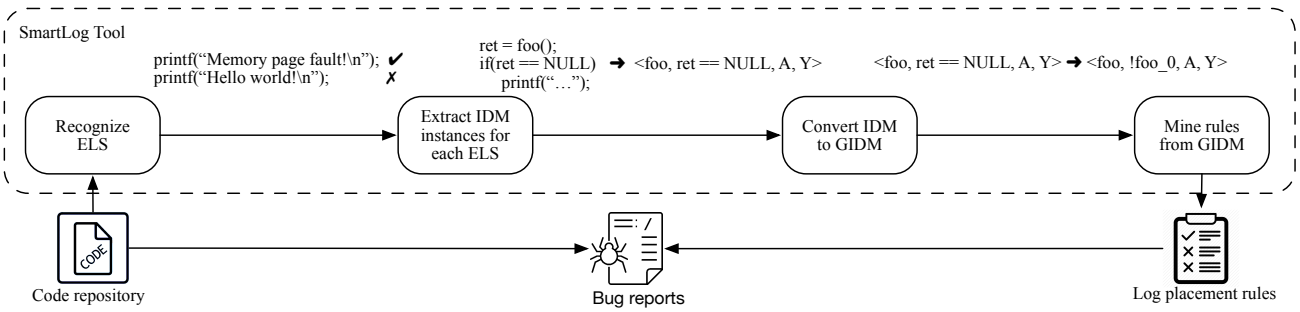


Figure 4. Architecture of SmartLog.

The main difference between GIDM instances and IDM instances is the check condition. IDM allows c in any form, while GIDM's $\hat{c}$ are more restricted. As such, GIDM has much less forms, and same GIDM instances are more likely to be equivalent in semantic. For example, all log contexts in Figure 2 will be described by the same GIDM instance $\hat{m} =< foo, \neg foo\_0, A, Y >$.

In Section IV-D, we will show how to convert an IDM instance to a number of GIDM instances.

IV. SmartLog Design & Implementation

In this section, we present the detailed design and implementation of SmartLog. We first give a schematic overview of SmartLog, and then describe each component individually. With these components available, we show how we implement SmartLog in practice and how to utilize it in daily software developments.

A. SmartLog Architecture

As illustrated in Figure 4, SmartLog has five steps. First, we analyze source code and recognize ELS, which will be labelled for later analysis. We then extract all the IDM instances from the source code, and convert them to GIDM instances. We collect the statistics of GIDM instances such as the total number of occurrences, the ratio of Y log decisions, the correlation between error-prone functions, check conditions and log decisions. With these statistics, we apply data mining techniques to mine ELS placement rules and assemble these rules to forge the SmartLog tool. In the next, we will introduce these components in details.

B. ELS Recognition

To learn the logging practices from the real-world projects, the first task is to recognize ELS. Each ELS is composed of two parts, i.e., logging function name and arguments, and thus we propose a two-phase recognition algorithm. We first use traditional keyword approach to find all the potential logging functions that may output logs, and collect all their invocations used for both ELS and NLS. We then apply a filtering algorithm to label the calls used for ELS.

Figure 5 gives the pseudo-code of the recognition algorithm. As we find the logging functions usually contain semantics of $error$, $exit$ or $output$, we first build a keyword list Ψ with all

```
Error Log Statements Recognition Algorithm
Input: Software source code
Output: Error log label in the source code

 %% Identify all the potential logging functions
 1. Build Ψ synonymous keyword list for
    {error, exit, output}
 2. For each function f, e.g. f = "fopen", do
 3.  Conduct word segmentation for f's name
 4.  Let F_p={f| ∃g ∈ one-gram(f) for which g ∈ Ψ}
 5. End for

 %% Choose calls of error log statement
 6. Build synonymous keyword list Ψ̂ for
    {error, exit, limitation, negation}
 7. For ∀f ∈ F_p
 8.  For ∀call of f
 9.   Label call if its argument contains keywords in Ψ̂
10.   Label call if it has data dependence with
      another statement containing keywords in Ψ̂
11.  End for
12. End for
```

Figure 5. Error log statements recognition algorithm.

synonymous words of $error$, $exit$, and $output$ using WordNet tool [20] (line 1). For any function f, we apply one-gram word segmentation [21] to segment its function name (line 3). If the segmented names contain at least one keyword in Ψ, f will be considered as a potential logging function and put in set F_p. f can be called multiple times, which may output both error logs (e.g., $printf(\text{“}Memory\ page\ fault\text{”})$) and non-error logs (e.g., $printf(\text{“}Hello\ World\text{”})$). In the second phase, we will remove the invocations of non-error ones.

To achieve this, the observation is that the arguments of ELS usually contain special semantics of $error$, $exit$, $limitation$ or $negation$. For this we build another keyword list $\hat{\Psi}$ that contains the synonyms of these four semantics (line 6). An invocation of the potential logging functions will be labelled as an ELS if its arguments contain any keyword in $\hat{\Psi}$ (line 9). Notice that Ψ is used to select a potential logging function, while $\hat{\Psi}$ is used to choose its invocations which are used for ELS. For example, $printf$ is a potential logging function (a synonym of $output$ in Ψ). And $printf(\text{“}Memory\ page\ fault\text{”})$ is an ELS, since the arguments contain $fault$ (a synonym of $error$ in $\hat{\Psi}$).

In addition, a call will also be labelled as an ELS if its arguments are data dependent to another statement which has

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on September 09,2023 at 01:42:18 UTC from IEEE Xplore. Restrictions apply.

keyword in $\hat{\Psi}$ (line 10). For the following example, line 2 has a potential logging function $printf$, and its argument msg is data dependent to line 1, which contains the keyword $Fatal$, line 2 thus will be labelled as an ELS.

```
1. msg = "Fatal:OOM";
2. printf("%s", msg);
```

C. Intention Description Model Extraction

With accurate labels of ELS, we are able to extract the IDM instances from software source code.

Figure 6 gives the pseudo-code of the IDM extraction algorithm. The extraction algorithm is based on the program dependence analysis technique [22]. Given a function body F. the IDM extraction algorithm first scans F's source code sequentially (line 1). For each function call f (line 2), the algorithm scans F backwards from f, looking for all branch statements with input dependence (line 3-5). Here input dependence means that the branch statement shares the common variable or alias variable with the arguments of f. For the example of Figure 1 case 3, there is an input dependence between the branch statement $if(!attname)$ and the function call $get\_attnum(childId,\ attname)$.

Such branch statement will output an IDM instance (Goto line 10). The Boolean formula in the condition of this branch statement will be record as c. If there is an ELS in the body of this branch statement, we will get an instance $m =< f, c, B, Y >$. Otherwise, $m =< f, c, B, N >$ (line 11).

The extraction algorithm then forwards scans F from f, looking for all branch statements with flow dependence. These branch statements will generate IDM instances similarly (line 6-8). Here flow dependence means that a variable written by f (e.g., return value or pointer argument) or its alias variable is in the condition of the branch statement.

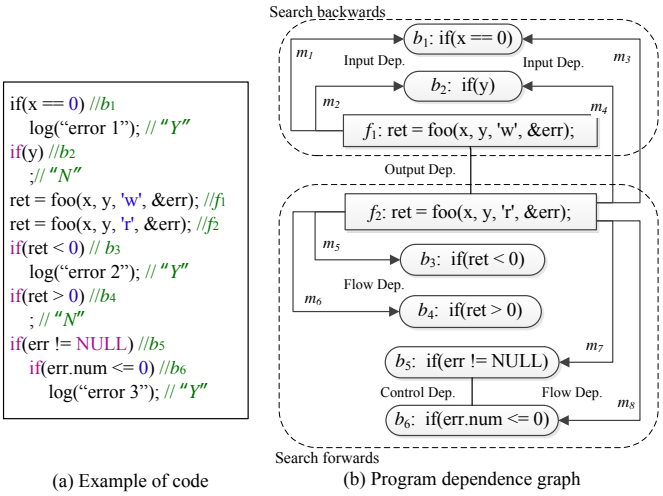
Figure 7 gives an example code snippet and its program dependence graph. For function call f_1, we first search backwards and find input-dependent branch statements b_1 and b_2, thus we have two instances. Then we search the body of these branch statements and find there is an ELS in b_1, while b_2 has no log statement. Therefore, our two IDM instances are $m_1 =< f_1, x{==}0, B, Y >$ and $m_2 =< f_1, y, B, N >$. Table III lists all the IDM instances. Searching backwards and forwards from f_2, we get m_3/m_4 and m_5/m_6/m_7/m_8, respectively.

```
Intention Description Model Extraction Algorithm
Input: Function body F
Output: the model instances M = {m|m =< f, c, B/A, Y/N >}
Initial: M = Φ

 1. Scan F's source code sequentially.
 2. For each function call f in F do
 3.   From f, scan the source code backwards
 4.   For all branch stmt with input dependence with f
 5.     Let p = 'B', goto line 10
 6.   From f, scan the source code forwards
 7.   For all branch stmt with flow dependence with f
 8.     Let p = 'A', goto line 10
 9. Return M
 %% This is the end of the main algorithm

 %% The next is a sub-function
10. c records Boolean formula in branch statement
11. If an error log is in branch, let l = 'Y'
    Otherwise, let l = 'N'
12. Let m =< f, c, p, l >, add m to M;
13. Return
```

Figure 6. Pseudo-code of IDM extraction algorithm.



(a) Example of code (b) Program dependence graph

Figure 7. Example code of program dependence graph.

D. Model Normalization

IDM is able to describe the semantic of log context, but is not general enough due to its various forms of check condition c. To better evaluate the semantic equivalence, we prefer models with the following features, (i) in variable level, all variables are uniformly named so that the models are independent from the variable name; (ii) in expression level, all expressions are simple and easy to evaluate the equivalence; and (iii) in statement level, there are only if branches and other branch statements like $switch$ and $else$ are convert to semantically equivalent if statements.

Towards this goal, we propose the Generalized Intention Description Model (GIDM). An IDM instance will be converted to one or multiple GIDM instances by the following steps.

Variable level transformation: (i) For any variable v, if v is a return value, it will be renamed to the function name with a "0" attached. For example, in Table III, the variable ret in m_5 will be renamed to foo_0. (ii) For any variable v, if v is input dependent or flow dependent on an argument of the error-prone function, v will be renamed to the function name with an index indicating the position of the argument. Taking m_1 of Table

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on September 09,2023 at 01:42:18 UTC from IEEE Xplore. Restrictions apply.

TABLE III
IDM INSTANCES AND NORMALIZED GIDM INSTANCES

m	f	c	B/A	Y/N	$\hat{c}$
m_1	f_1	$x == 0$	B	Y	$!foo\_1$
m_2	f_1	y	B	N	foo_2
m_3	f_2	$x == 0$	B	Y	$!foo\_1$
m_4	f_2	y	B	N	foo_2
m_5	f_2	$ret < 0$	A	Y	$foo\_0 < 0$
m_6	f_2	$ret > 0$	A	N	$0 < foo\_0$
m_7	f_2	err!=NULL	A	N	foo_4
m_8	f_2	err!=NULL && $err.num$ <=0	A	Y	foo_4 && $foo\_4.num$ <=0

III for example, the variable x is the first argument of foo, and will be renamed to foo_1. (iii) For any variable v, if v is a member of classes or data structures, a suffix indicating the member name will be added, e.g., $foo\_4.num$ in m_8. (iv) Constant expressions will be computed directly. (v) Irrelative variables (i.e., they are not sensitive arguments, return values or point arguments of the error-prone function) will be left unchanged.

Expression level transformation: (i) The operators $>$ and $\geq$ will be changed to $<$ and $\leq$ for unification by switching the operands, e.g., in m_5, operands of $\hat{c}$ is changed. (ii) The conditional operator $a?b{:}c$ in C language is changed to $(a \wedge b) \vee (\neg a \wedge c)$ (notice that this equivalence only happens in conditional expression of branch statement, otherwise they are non-equivalent). (iii) When there is 0 or NULL on one side of operator $==$ or $!=$, it will be removed by adding a negation according to the operator. For example, the expression $x == 0$ and $err! = NULL$ in m_1 and m_7 has the same meaning as $!x$ and err. (iv) The assign expression is replaced by its assigned value. (v) The operands of commutative operator (e.g., $+$, $==$ or $||$) will be rearranged in lexicographical order.

Statement level transformation: (i) The $switch$ statement will be transferred to an equivalent if statement. For example, the statement $switch(flag)\ case\ 1{:}\ log(\text{“}error\text{”})$; will be transferred to $if(flag{==}1)\ log(\text{“}error\text{”})$. (ii) When the error log statement is located in $else$ branch of if statement, the Boolean formula is negated.

Splitting (one-to-many transformation): The complex Boolean formulas connected by multiple logical operators can be recursively divided into several independent sub-formulas by following rules: (i) from $if(a \vee b)$ to $if(a)$ and $if(b)$; (ii) from $if(a \wedge (b \vee c))$ to $if(a \wedge b)$ and $if(a \wedge c)$. By this, a complex instance can be split into several sample instances, which share the same error-function f, check position B/A, log decision Y/N, but have different sub-formulas in check conditions $\hat{c}$.

For example, in Table III, the c of IDM instances will be normalized to $\hat{c}$ of GIDM instances.

E. Mine Log Placement Rules

With normalized GIDM, we can collect the statistics of GIDM instances. With these statistics, we then apply data mining techniques to mine the log placement rules. Although the GIDM instances are labelled, we still do not have the ground truth about whether the log is needed. We only know, for some circumstances, someone adds logs, and others do not. We have no knowledge about who is right. For this consideration, we employ the frequent item set mining algorithm, Aprior algorithm [13], instead of supervised learning.

Intuitively, Aprior algorithm helps mining the frequent item set and learn the association rules between the error-prone function f, the normalized check condition $\hat{c}$, the position B/A, and the log decision Y/N. Let $Pr(A)$ denote the probability of an event A to occur, $Pr(A, B)$ denote the probability of both events A and B occur, $Pr(B|A)$ denote the conditional probability of an event B given that another event A to occur. In Aprior algorithm, $Pr(B|A)$ is also called the $confidence$. Define $\rho(A, B, C)$ as the $correlation$ between the events A, B and C,

$$\rho(A, B, C) = \frac{Pr(A, B, C)}{Pr(A) * Pr(B) * Pr(C)}.$$

The log placement rule is a 3-tuple $r =< f, \hat{c}, B/A >$ meaning “f should be logged under $\hat{c}$ before (B)/after (A) its invocations". SmartLog mines the log placement rules from GIDM instances satisfying following properties:

- $Pr(Y|f,\ \hat{c},\ B) \geq \mathrm{Pr}_{th}$ or $Pr(Y|f,\ \hat{c},\ A) \geq \mathrm{Pr}_{th}$
- $\rho(\hat{c},\ B,\ Y|f) \geq 1$ or $\rho(\hat{c},\ A,\ Y|f) \geq 1$

where Pr_{th} is a pre-defined threshold. The property $Pr(Y|f,\ \hat{c}, B) \geq \mathrm{Pr}_{th}$ means that, when f, $\hat{c}$ and B appear together, the probability of log decision Y is greater than Pr_{th}. The property $\rho(\hat{c},\ B,\ Y|f) \geq 1$ means that $\hat{c}$, B and Y are not negatively associated when f appears.

For example, the following example lists 4 GIDM instances and the number of their appearance. Assuming Pr_{th} = 0.5, both $Pr(Y|foo,\ foo\_0 < 0,\ A)$ = 0.52 and $Pr(Y|foo,\ foo\_0 == 0,\ A)$ = 0.91 are satisfying the first properties. As for the second properties, $\rho(foo\_0 < 0, A, Y|foo)$ = 0.79 while $\rho(foo\_0 == 0, A, Y|foo)$ = 1.36. In this situation, SmartLog only mines the rule that “foo should be logged after its invocation when $foo\_0 == 0$ (the return value equals to 0)".

$< foo, foo\_0 < 0, A, Y >$:10 $< foo, foo\_0 == 0, A, Y >$:10
$< foo, foo\_0 < 0, A, N >$:9 $< foo, foo\_0 == 0, A, N >$:1

V. EVALUATION

In this section, we evaluate SmartLog upon 6 projects in Section II. We first assess the impact of parameter in SmartLog, and compare the accuracy of SmartLog with existing log placement tools. In the next, we evaluate the effectiveness of SmartLog.

A. Accuracy Assessment

SmartLog obtains GIDM instances from source code, and further applies them into mining algorithm. In GIDM, SmartLog mines the error-prone function f, check condition $\hat{c}$, and check position B/A, of which the $confidence$ is greater than a pre-defined threshold and the $correlation$ is greater than 1. We first

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on September 09,2023 at 01:42:18 UTC from IEEE Xplore. Restrictions apply.

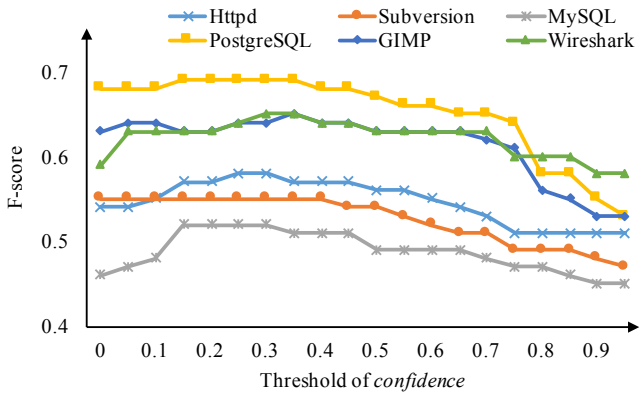


Figure 8. Impact of threshold in mining algorithm.

evaluate the impact of the threshold, and under the optimized threshold, we compare SmartLog with existing works.

1) Experiment Setup: We collect all instances of GIDM as dataset, each of them is labelled as Y (there is an ELS) or N (otherwise). Then the dataset is equally split into two sets, one for training and the other for test. In each experiment, we remove all ELS from the test set, and evaluate the accuracy of different logging approaches for recovering those removed logs. As mentioned before, we use P (i.e., precision rate) and R (i.e., recall rate) to evaluate the accuracy. For the ease of comparison, we further calculate F_{score} which is the harmonic mean of P and R. All experiments are independently executed 10 times, and we compute the averaged P, R and F_{score}.

2) Impact of Parameter: In SmartLog, only one parameter needs to be specified: $confidence$ (belongs to $[0, 1]$). As shown in Figure 8, the horizontal axis is $confidence$, from to 0 to 1 with step length of 0.05. The vertical axis is F_{score}. Taking MySQL for example, the F_{score} increases along with $confidence$ from 0 to 0.15, then reaches the highest value 0.52. When the $confidence$ is greater than 0.3, F_{score} begins to decrease. Similarly, all the six projects can reach their best F_{score} between 0.3 and 0.35, thus we choose 0.33 as the optimized threshold.

3) Comparison Experiments: We compare SmartLog with two baseline approaches: Errlog and LogAdvisor. By Errlog, all the instances in test set will be logged when belonging to several code patterns. As for LogAdvisor, it first learns from the syntax features around training instances. Then, for each test instance, LogAdvisor collects its syntax features, and makes log decision by checking whether similar features have appeared around training instances. Table IV shows the number of instances we extracted from each project, and P and R of different approaches. The P of Errlog is higher than LogAdvisor's in all six projects. This is consistent with our experiments in Table II. Errlog, however, misses many logs since its strict patterns, resulting in an extremely low R rate. LogAdvisor gets more balanced performance. The R rate is much better than Errlog's, since it can learn beyond fix code patterns. LogAdvisor, therefore, has a higher F_{score} than Errlog. SmartLog achieves better P and R. On one hand, the capability of semantic-equivalence identification increases the R rate. On the other hand, with consideration on recognition of ELS, the P rate is improved.

Despite the improvement of SmartLog, there are still many FPs and FNs. We manually sample and study 100 FPs and 100 FNs to reveal the root causes. The result show that (i) 87% of FPs are caused by return statement. In test set, when a log-worthy context appears, developers may choose to return an error code or error message instead of logging immediately. (ii) 13% of FPs have neither error log nor return statement. These FP may be caused by the incompleteness of logs in target projects. Another reason is that these contexts might have been well handled by other approaches, e.g., safe free.

As for FNs, (i) 42% of them are caused by different semantics, especially, the irrelative variables (Section IV-D). When a test instance has an irrelative variable, SmartLog cannot infer its semantics from training set, therefore fails to log. (ii) 12% have semantic equivalence, but SmartLog fails to obtain the semantics. One case is caused by function call, for example, both $if(length(str) == 0)$ and $if(!str)$ are checking whether str is empty. Another case is caused by pointer, for example, both $if(*str ==' \backslash 0')$ and $if(!str)$ are checking whether str is empty. SmartLog fails to deal with both above cases. (iii) for other 46%, the functions have no log statement in training set at all, thus SmartLog cannot learn.

B. Effectiveness Evaluation

In this section, we evaluate the effectiveness of additional logs from user's aspect including $utility$ and $overhead$.

1) Experiment Setup: In Section V-A, we assess the accuracy of SmartLog by using system-centric metrics, i.e., $precision$ and $recall$. As for user-centric aspect, we prefer $utility$ and $overhead$. In practice, the $utility$ of SmartLog is that to what extent the tool can help developer on adding ELS. While the $overhead$ is the extra cost of running time caused by new logs. A high $recall$ rate, in general, means a high $utility$, since the tool can place as many log statements as possible. On the other hand, the $precision$ rate is related to $overhead$, since unnecessary logs will slow down the target projects in field. In this section, we first provide the result of SmartLog, and then evaluate the $utility$ and $overhead$. We also compare SmartLog with existing tools.

TABLE IV
COMPARISON OF DIFFERENT LOGGING APPROACHES

Project	# Inst.	Errlog		LogAdvisor		SmartLog	
		P	R	P	R	P	R
Httpd	21624	0.73	0.08	0.53	0.44	0.72	0.49
Subversion	24843	0.83	0.08	0.42	0.46	0.67	0.46
MySQL	48971	0.40	0.05	0.29	0.37	0.73	0.40
PostgreSQL	112736	0.83	0.12	0.65	0.48	0.81	0.60
GIMP	133127	0.64	0.23	0.51	0.51	0.81	0.54
Wireshark	250265	0.51	0.10	0.33	0.48	0.82	0.54
Average of F		0.18		0.45		0.61	

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on September 09,2023 at 01:42:18 UTC from IEEE Xplore. Restrictions apply.

TABLE V
RESULT OF SMARTLOG

Project	Number of Rules	Number of Logs
Httpd	109	294
MySQL	90	302
GIMP	76	242
Subversion	112	288
PostgreSQL	180	818
Wireshark	78	180

2) Result of SmartLog: SmartLog places new log statements for the violations of logging rules (mined in Section IV-E). When placing new logs, SmartLog first inserts check conditions (if necessary), and then adds log statements. Table V shows the results of SmartLog, including the number of log placement rules and additional logs. For Httpd, 109 rules are learned, and SmartLog finds and inserts log statements for 294 violations.

3) Evaluation of Utility: Due to the lack of ground truth, there is no guarantee about the quality of the additional log statements, even though we employ several mature software systems and achieve a relative high accuracy. The logging behaviors of developers may be ad-hoc. In some cases, developers perform logging as afterthoughts, for example, after a failure happens and logs are needed. Log statements may be added, modified, and even deleted with the evolution of systems [5]. To evaluate the *utility*, we select real-world patches of adding ELS during software evolution.

We apply *diff* utility [24] on two different versions to generate the patches, and each patch contains a function call which was not logged in old version but has been logged in new version. We totally filter 86 patches (this is so-far all the patches we can identify), and check the number that SmartLog, LogAdvisor and Errlog can cover. Figure 9 shows the versions and results. Taking PostgreSQL for example, 21 additional ELS were patched from version 9.3.5 to version 9.5.0, of which 12 are covered by SmartLog, while 6 and 4 are covered by LogAdvisor and Errlog, individually.

The overall *utility* of SmartLog, LogAdvisor and Errlog are 57% (49/86), 26% (22/86) and 12% (10/86), respectively. The results of both SmartLog and Errlog are consistent with the R in Table IV, while the *utility* of LogAdvisor is lower

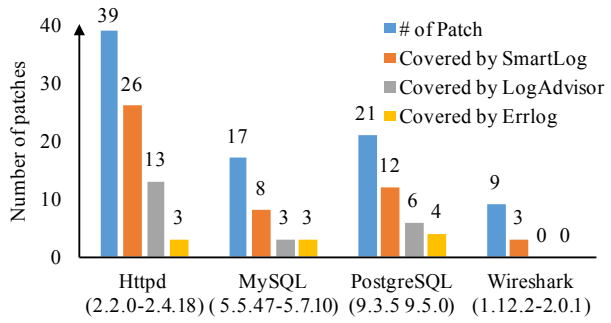


Figure 9. Utility evaluation.

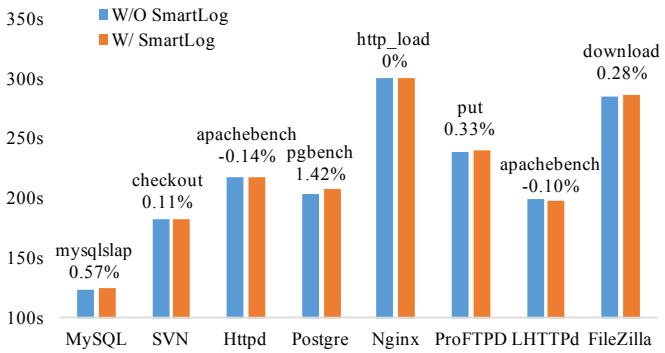


Figure 10. The overhead of SmartLog.

than expectation. Given a check condition, LogAdvisor can decide whether to log or not. Some patches, however, have no check condition in old version at all, and thus LogAdvisor may fail to log. By contrast, both SmartLog and Errlog can insert check conditions first, and then add error log statements.

Beyond 57% patches covered by SmartLog, the rest 43% is caused by two reasons, (i) the function has no log statement in old version, so SmartLog can not learn (33%); (ii) the function has log statements in old version, but the semantics of their log contexts are non-equivalent (10%).

4) Evaluation of Overhead: Since all the additional log statements are associated with a check condition, the new log statements only print message when an error happens, and thus the *overhead* is limited. We evaluate the *overhead* by some widely-used workloads listed in Figure 10. T_{wo} and T_w are running times without/with applying SmartLog. The *overhead* is calculated by $(T_w - T_{wo})/T_{wo}$. We independently run the experiments 5 times. For MySQL, T_{wo} is 123.3s and T_w is 124s when using workload *mysqlslap*. Thus it has an *overhead* of 0.57%. The average *overhead* is less than 1%, with a maximum of 1.4% for PostgreSQL. The negative percentage is probably caused by run-time environment such as network delay. By comparison, the *overhead* of Errlog is 1.1% to 1.4%, and there is no significant difference between these two tools. For LogAdvisor, it is designed as an online logging suggestion tool, and does not actually insert any log statements, thus there is no *overhead*.

VI. THREATS TO VALIDITY

Quality of developer-written logs: SmartLog learns from the real-world projects representing how the experienced developers write log statements. The learning process works under the premise that, in a large-scale software project, there are always part of developers who write high-quality logs. However, there is no ground truth on what is high-quality logging. We believe the logs written by experienced developers are closer to the ground truth, since all the studied software projects have high code quality, active maintenance and long history of evolution. In such a setting, SmartLog can learn the good logging practices and improve the bad ones.

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on September 09,2023 at 01:42:18 UTC from IEEE Xplore. Restrictions apply.

Completeness of IDM and GIDM models: We propose IDM to describe the log intention, or the semantics of log context, and the IDM is further generalized to GIDM, which is able to evaluate the semantic equivalence between different log contexts. The GIDM model is not supposed to evaluate equivalence between any two log contexts. In large-scale software, we need to find a practical tradeoff between scalability and completeness. In SmartLog design, we prefer a light-weight algorithm, which can evaluate the equivalence of overwhelming majority cases instead of all cases.

Effectiveness of overhead evaluation: In evaluation of overhead, we only use some widely-used workload for each software project to compare the extra running time caused by additional log statements. Using this workload, however, can not rest assured that all the additional log statements will be executed. It is non-trivial to trigger a certain line in large-scale software project. We evaluate the normal execution of the studied software projects. To prevent significant performance degradation in extreme situation, all the new logs are controlled by a check condition that determines whether the log will be printed.

VII. Related Work

Existing log related works can be roughly classified as log characterization, postmortem analysis and log automation.

Log characterization and enhancement. Many works study existing log statements, including characterization and enhancement. Salfner *et al.* [3] introduce a set of recommendations to improve the expressiveness of the logs. Yuan *et al.* [5] systematically study log modifications to assess the existing log quality and the demand of developers. Fu *et al.* [4] conduct a questionnaire survey and find that most participants think logs are a primary source for problem diagnosis. Rabkin *et al.* [25] propose an approach to visualize console logs. They describe how to obtain a graph that can be used to improve the logging mechanisms. LogEnhancer [2] can automatically identify important variables and insert them into existing log statements. Cinque *et al.* [26] analyze the limitations of current logging mechanisms and propose a rule-based approach to make logs effective to analyze software failures. Kabinna *et al.* [27] empirically study the stability of logging statements and mitigate negative effects that are caused by unstable logging statements. Chen *et al.* [28] characterize and detecte anti-patterns in the logging code helping developing and maintaining high-quality logging code. These works aim to characterize the existing log statements, while SmartLog is designed to insert new log statements.

Postmortem analysis and failure diagnosis. Logs act as an important role to help developers locate failures. Xu *et al.* [29], [6] propose a general approach on problem detection via the analysis of console logs, which is the built-in monitoring information in most software systems. Kadav *et al.* [7] insert logging code so that system administrators can proactively repair or replace hardware that fails. SherLog [8] analyzes logs from a failed production run to help engineers diagnose the error without reproducing it. Pecchia *et al.* [9] analyze the factors that determine accurate detection of software failures through event logs. Ding *et al.* [30] propose Log2, a cost-aware logging mechanism for performance diagnosis. Failure diagnosis sometimes requires a mass of manual assignment and domain professional knowledge to locate failures, like cooperative debugging [31] and deterministic replay [32], [33], [34], [35]. There are also many works focus on fault localization [36] and program repair [37]. Barik *et al.* [38] catalog a diverse set of activities that leverage log messages, and identify challenges in conducting these activities. This type of works leverages log messages to diagnose failures. When lacking enough log messages, SmartLog can add new log statements and improve the capability of these works.

Log Placement. Yuan *et al.* [1] and Zhu *et al.* [10] try to predict potential error log places. Yuan *et al.* propose Errlog, it provides a set of error patterns and applies the patterns to the source code. A log will be placed when patterns are matched, e.g., function return errors, exception signals and unexpected cases. Zhu *et al.* propose LogAdvisor, it identifies and selects a number of syntax features from the software code, and then applies machine learning and noise handling techniques to make the log decisions. The syntax features include the structural features, textual features and syntactic features. In contrast, SmartLog is able to learn beyond code patterns, and mine log placement rules on semantic level. Zhao *et al.* propose Log20 [39], which focuses on placing non-error log statements for disambiguating code paths, while SmartLog is designed to place error ones.

VIII. Conclusion

When software systems fail, logging is a principal measure for engineers in the field to troubleshoot and diagnose in the very first time. Current logging practices frequently are not strong enough to provide efficient guidance. Log automation tools have focused on this gap. We comprehensively investigated the state-of-the-art tools, and studied their limitations. One limitation is learning on syntax level, another is mixed use of error log statements and non-error ones. We propose IDM to describe the log intention, which is further generalized to GIDM. As a proof of concept, we implement an automatic log placement tool, SmartLog, which can insert error log statements in proper places. Assessment on six large-scale software systems demonstrates that SmartLog has a better accuracy than state-of-the-art tools. Evaluation of effectiveness shows that for 86 real-world patches aimed to add logs, 57% of them can be covered by SmartLog, while the overhead is less than 1%.

Acknowledgment

The authors would like to thank the anonymous reviewers for their insightful comments. The work described in this paper was substantially supported by National Key R&D Program of China (Project No. 2017YFB1001802); National Natural Science Foundation of China (Project No. 61690203, 61379146, 61332018 and 61772046).

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on September 09,2023 at 01:42:18 UTC from IEEE Xplore. Restrictions apply.

REFERENCES

[1] 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 *OSDI*, vol. 12, 2012, pp. 293–306.

[2] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving software diagnosability via log enhancement," *ACM Transactions on Computer Systems (TOCS)*, vol. 30, p. 4, 2012.

[3] F. Salfner, S. Tschirpke, and M. Malek, "Comprehensive logfiles for autonomic systems," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*. IEEE, 2004, p. 211.

[4] Q. Fu, J. Zhu, W. Hu, J.-G. Lou, R. Ding, Q. Lin, D. Zhang, and T. Xie, "Where do developers log? an empirical study on logging practices in industry," in *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 24–33.

[5] D. Yuan, S. Park, and Y. Zhou, "Characterizing logging practices in open-source software," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 102–112.

[6] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 117–132.

[7] A. Kadav, M. J. Renzelmann, and M. M. Swift, "Tolerating hardware device failures in software," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 59–72.

[8] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, "Sherlog: error diagnosis by connecting clues from run-time logs," in *ACM SIGARCH Computer Architecture News*, vol. 38. ACM, 2010, pp. 143–154.

[9] A. Pecchia and S. Russo, "Detection of software failures through event logs: An experimental study," in *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*. IEEE, 2012, pp. 31–40.

[10] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang, "Learning to log: Helping developers make informed logging decisions," in *Proc. of ACM/IEEE ICSE*, 2015.

[11] "The art of logging," http://www.codeproject.com/Articles/42354/The-Art-of-Logging, 2016.

[12] "The problem with logging," http://blog.codinghorror.com/the-problem-with-logging, 2016.

[13] R. Agrawal, R. Srikant *et al.*, "Fast algorithms for mining association rules," in *Proc. 20th int. conf. very large data bases, VLDB*, vol. 1215, 1994, pp. 487–499.

[14] Httpd, https://httpd.apache.org, 2016.

[15] Subversion, https://subversion.apache.org, 2016.

[16] MySQL, https://www.mysql.com, 2016.

[17] PostgreSQL, https://www.postgresql.org, 2016.

[18] GIMP, https://www.gimp.org, 2016.

[19] Wireshark, https://www.wireshark.org, 2016.

[20] WordNet, http://wordnet.princeton.edu, 2016.

[21] W. Segmentation, https://jeremykun.com/2012/01/15/word-segmentation/, 2016.

[22] R. Allen and K. Kennedy, *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann San Francisco, 2002, vol. 1.

[29] W. Xu, L. Huang, A. Fox, D. A. Patterson, and M. I. Jordan, "Mining console logs for large-scale system problem detection," *SysML*, vol. 8, pp. 4–4, 2008.

[23] "Reaching definition," https://en.wikipedia.org/wiki/Reaching_definition, 2016.

[24] "Gnu diffutils," https://www.gnu.org/software/diffutils/, 2016.

[25] A. Rabkin, W. Xu, A. Wildani, A. Fox, D. Patterson, and R. Katz, "A graphical representation for identifier structure in logs," in *Proc. Workshop Managing Systems via Log Analysis and Machine Learning Techniques*, 2010.

[26] M. Cinque, D. Cotroneo, and A. Pecchia, "Event logs for the analysis of software failures: A rule-based approach," *Software Engineering, IEEE Transactions on*, vol. 39, no. 6, pp. 806–821, 2013.

[27] S. Kabinna, W. Shang, C.-P. Bezemer, and A. E. Hassan, "Examining the Stability of Logging Statements," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2016, pp. 326–337.

[28] B. Chen and Z. M. J. Jiang, "Characterizing and detecting anti-patterns in the logging code." *ICSE*, pp. 71–81, 2017.

[30] R. Ding, H. Zhou, J.-G. Lou, H. Zhang, Q. Lin, Q. Fu, D. Zhang, and T. Xie, "Log2: A cost-aware logging mechanism for performance diagnosis," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, 2015, pp. 139–150.

[31] 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 *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 103–116.

[32] G. Altekar and I. Stoica, "Odr: output-deterministic replay for multicore debugging," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 193–206.

[33] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen, "Revirt: Enabling intrusion analysis through virtual-machine logging and replay," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 211–224, 2002.

[34] D. Subhraveti and J. Nieh, "Record and transplay: partial checkpointing for replay debugging across heterogeneous systems," in *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*. ACM, 2011, pp. 109–120.

[35] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy, "Doubleplay: parallelizing sequential logging and replay," *ACM Transactions on Computer Systems (TOCS)*, vol. 30, no. 1, p. 3, 2012.

[36] X. Mao, Y. Lei, Z. Dai, Y. Qi, and C. Wang, "Slice-based statistical fault localization," *Journal of Systems and Software*, vol. 89, pp. 51–62, 2014.

[37] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 254–265.

[38] T. Barik, R. DeLine, S. M. Drucker, and D. Fisher, "The bones of the system - a case study of logging and telemetry at Microsoft." *ICSE*, 2016.

[39] X. Zhao, K. Rodrigues, Y. Luo, M. Stumm, D. Yuan, and Y. Zhou, "Log20: Fully Automated Optimal Placement of Log Printing Statements under Specified Overhead Threshold," in *the 26th Symposium*. New York, New York, USA: ACM Press, 2017, pp. 565–581.

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on September 09,2023 at 01:42:18 UTC from IEEE Xplore. Restrictions apply.