Empirical Software Engineering (2020) 25:2302–2340
https://doi.org/10.1007/s10664-019-09757-y

Guiding log revisions by learning from software evolution history

Shanshan Li[1] · Xu Niu[1] · Zhouyang Jia[1] · Xiangke Liao[1] · Ji Wang[1] · Tao Li[2]

Published online: 9 September 2019

Abstract
Despite the importance of log statements in postmortem debugging, developers are difficult to establish good logging practices. There are mainly two reasons. First, there are no rigorous specifications or systematic processes to instruct logging practices. Second, logging code evolves with bug fixes or feature updates. Without considering the impact of software evolution, previous works on log enhancement can partially release the first problem but are hard to solve the latter. To fill this gap, this paper proposes to guide log revisions by learning from evolution history. Motivated by code clones, we assume that logging code with similar context is pervasive and deserves similar modifications and conduct an empirical study on 12 open-source projects to validate our assumption. Upon this, we design and implement LogTracker, an automatic tool that learns log revision rules by mining the correlation between logging context and modifications and recommends candidate log revisions by applying these rules. With an enhanced modeling of logging context, LogTracker can instruct more intricate log revisions that cannot be covered by existing tools. Our experiments show that LogTracker can detect 369 instances of candidates when applied to the latest versions of software. So far, we have reported 79 of them, and 52 have been accepted.

Keywords Log revision · Software evolution · Failure diagnose · Empirical study

1 Introduction

Log statements are inserted by developers to record the runtime status of software. Being informative and convenient, log messages have commonly been adopted to aid in postmortem failure diagnosis. Despite the importance of logging code, it is challenging for

Communicated by: Chanchal Roy, Janet Siegmund, and David Lo

The work in this paper was supported by National Natural Science Foundation of China (Project No.61690203, U1711261, 61872373 and 61872375); National Key R&D Program of China (Project No.2017YFB1001802 and 2017YFB0202201). An earlier version (Li et al. 2018) was presented at the IEEE/ACM International Conference on Program Comprehension 2018.

✉ Shanshan Li
shanshanli@nudt.edu.cn

Extended author information available on the last page of the article.

developers to establish good logging practices as software evolves. There are two main reasons for this. First, there are no rigorous specifications and systematic processes to guide the practices of software logging (Fu et al. 2014; Yuan et al. 2012b; Pecchia et al. 2015). Hence, the means by which developers make log placement decisions is both subjective and arbitrary. Second, logging code evolves with bug fixes or feature updates. This problem is illustrated in Fig. 1. Figure 1a displays a commit that fixed bugs by inserting preliminary validation of sensitive data (i.e., return value of *refresh_cache_entry()*). In addition to the validation code, the developers also inserted one new log statement to support better exception handling. In Fig. 1b, the developers implemented a new feature which supports new parameter (i.e., "–*show-progress*"). This commit also updated the verbosity of original log statement to meet with new feature. In both cases, log revisions are committed in response to bug fixes or feature updates.

Considering the difficulty of reaching good logging practices, several log revisions may be missed by developers during software evolution. This meets with the finding of recent

Commit id: 55e9f0e5c9a918c246b7eae1fe2a2e... in Git
File: Git, merge-recursive.c
Messages: crash because developers forgot that refresh_cache() can return NULL

```
  nce = refresh_cache_entry(ce, CE_MATCH_REFRESH|CE_MATCH_IGNORE_MISSING);
+ if(!nce)
+  return error(_("addinfo_cache failed for patch '%s'"), path);
  if(nce != ce) ...
```

(a) Insert new log statement along with bug fixes

Commit id: 8c2fd06ba80b5312b4540859d45266... in Wget
File: log.h and progress.c
Messages: implement –show-progress switch to force the display of the progress bar

```
+ enum log_options{LOG_VERBOSE,..., LOG_PROGRESS};
  ...
  if(eta < INT_MAX - 1)
-  logprintf(LOG_VERBOSE, " %s",
+  logprintf(LOG_PROGRESS, " %s",
     eta_to_human_short((int) (eta+0.5), true));
```

(b) Update verbosity of log statement along with feature updates

Commit id: 1335d76e4569fa84e52dc24c88c04d... in Git
File: merge-recursive.c
Messages: calls refresh_cache_entry() to make a cache entry as up-to-date.

```
- return add_cache_entry(ce, options);
+ ret = add_cache_entry(ce, options);
+ if (refresh)
+  struct cache entry* nce;
+  nce = refresh_cache_entry(ce, CE_MATCH_REFRESH|CE_MATCH_IGNORE_MISSING);
+  if(nce != ce)
+   return add_cache_entry(nce, options);}
```

(c) Ancestor of the commit in Figure 1a which missed the log revision

Fig. 1 Logging code evolves with bug fixes or feature updates

Bug 35616 in MySQL: memory overrun on 64-bit linux on setting large values for keybuffer-size.
Revision 2715 in MySQL-5.6, mysys/safemalloc.cc:

```
if (file){
- fprintf(file,"Warning:Memory that was not free'ed
-  (%ld bytes):\n",sf_malloc_cur_memory);
+ fprintf(file,"Warning:Memory that was not free'ed
+  (%lu bytes):\n",(ulong)sf_malloc_cur_memory);
```

(a) Update variables of log statement to make it work on 64-bit machine

Bug 2830 in Squid: show the NULL byte more clearly in future releases.
Revision 9142 in Squid-3.0, HttpHeader.cc:

```
- if(memchr(header_start,'\0',header_end- header_start)) {
+ char *nulpos;
+ if((nulpos=(char*)memchr(header_start,'\0', header_end-header_start))){
   debugs(55,1,"WARNING: HTTP header
    contains NULL characters {"<<
-   getStringPrefix(header_start,header_end)<<"}");
+   getStringPrefix(header_start,nulpos)<<"}NULL{"
    <<getStringPrefix(nulpos+1,header_end)<<"}");
```

(b) Enhance existing log statement for ease of bug diagnosis

Bug 2787 in Squid: the message can quite probably be discarded completely. Adjust the message, bumping it down out of warnings completely.
Revision 9157 in Squid-3.0, http.cc:

```
 default:   /* Unknown status code */
- debugs(11,0,HERE<<"HttpStateData::cacheableReply:unexpected http status code"
+ debugs(11,DBG_IMPORTANT,"WARNING: Unexpected http status code"
   <<rep->sline.status);
```

(c) Suppress misleading log statement to avoid confusing users

Fig. 2 Examples of log revisions that improve logging practices

works (Yuan et al. 2012b, c; Chen and Jiang 2017a) which pointed out that around 33% of the log revisions are introduced as after-thoughts.[1] For instance, the log revision in Fig. 1a was actually missed in the first beginning. Figure 1c displayed its ancestor commit that inserted code which invokes the error-prone function (i.e., *refresh_cache_entry()*) and uses its return value without validation. Developers did not notice the necessity of logging until the failure happened. For making up, one commit (see Fig. 1a) was introduced to validate the return value as well as printing suitable log messages.

To avoid missing necessary log revisions, there are already many works that focused on improving logging practices. Errlog (Yuan et al. 2012a) and LogAdvisor (Zhu et al. 2015) help to insert missing log statements for given code snippets. LogEnhancer (Yuan et al. 2012c) appended informative variables to log messages in order to resolve the ambiguity in failure diagnosis. *Log*^{2} (Ding et al. 2015) and Log20 (Zhao et al. 2017) decided what log messages to output by seeking a balance between informativeness and overhead. Although

[1]Revisions are considered as after-thoughts if they are modified later than the modification of the surrounding code.

Git-2.4.2 vs Git-2.4.3, file:refs.c, function:rename_ref

```
  char *oldrefname ...
  lock=lock_ref_sha1_basic
-  (oldrefname,NULL,NULL,0,NULL);
+  (oldrefname,NULL,NULL,NULL,0,NULL,&err);
  if (!lock) {
-  error("unable to lock %s for rollback", oldrefname);
+  error("unable to lock %s for rollback: %s", oldrefname, err.buf);
```

(a) Modify log statement in a way similar to Figure 3b

Git-2.4.2 vs Git-2.4.3, file:refs.c, function:reflog_expire

```
  char *refname ...
  lock = lock_ref_sha1_basic
-  (refname, sha1, NULL, 0, &type);
+  (refname, sha1, NULL, NULL, 0, &type, &err);
  if (!lock)
-  return error("cannot lock ref '%s'",refname);
+  error("cannot lock ref '%s': %s",refname,err.buf);
```

(b) Modify log statement in a way similar to Figure 3a

Fig. 3 Example of similar modifications on log statements with similar logging context

the abovementioned works are partly able to provide guidances on software logging, it is difficult for them to predict log revisions that are related to bug fixes or feature updates (see Fig. 1). This is because they ignored the impact that software evolution has on logging code.

It is challenging to improve logging practices during software evolution. First, evaluation of the same log statements may vary in versions[2] of software. In Fig. 2a, the original log statement was thought to be correct in the initial version. However, it required modification in the new version which should run on 64-bit machines. Second, log statements cannot always provide sufficient clues when diagnosing unpredictable bugs. Figure 2b shows a real log enhancement patch in Squid. Here, the developers appended a new variable into an existing log statement in order to pinpoint the location of the null character. Third, verbose log messages may interfere with the understanding of failure causes, thus decreasing the efficiency of failure diagnosis. In Fig. 2c, users complained about the confusing log messages. This problem was discussed with the developers for over 150 days. Making reference to related documents, they finally established that this log statement was a verbose message. Its verbosity was bumped down from level-0 (*critical*) to level-1 (*important*) to release interference. In addition, excessive log messages also increase runtime overhead and detrimentally affect software performance (Arnold and Ryder 2001; Sigelman et al. 2010; Ding et al. 2015). Consequently, it is reasonable to remove or suppress verbose log messages in order to avoid unwanted interference and unnecessary overhead.

Facing the abovementioned challenges, this paper intends to learn log revision behaviors from software evolution. Motivated by code clones, we assume that log statements

[2]Here "version" means the internal version number (not the release version). This may be incremented many times in one day.

Table 1 Subject Software

Software	Description	SLOC	LLOC	SLOC/LLOC
Httpd 2.4.27	Web server	188,360	12,960	15
Git 2.9.5	Version control system	429,166	7,318	59
Mutt 1.9.1	E-mail client	93,527	1,430	65
Rsync 3.1.2	File synchronization	47,720	200	239
Collectd 5.8.0	Perfomance collector	97,475	817	119
Postfix 3.2.4	E-mail server	118,219	8,265	14
Tar 1.3	Archive management	77,310	822	94
Wget 1.19.2	File retriever	84,678	1,642	52
OpenDDS 3.13	Data distribution service	252,647	19,037	13
Ice 3.7.1	RPC framework	343,076	2,194	156
GIMP 2.10.10	Image manipulation program	800,691	8,258	97
Wireshark 3.0.2	Network protocol analyzer	2,308,080	9,736	237
Total		4,840,949	72,679	**67**

which share semantically similar context[3] are pervasive in software. As such, they ought to undergo similar modifications if they are revised. In order to verify our hypothesis, we conduct an empirical study using real-world log revisions. Figure 3 shows one pair of context-similar log statements which were modified similarly. They both printed the first reference argument of *lock_ref_sha1_basic()* when the return value of *lock_ref_sha1_basic()* is logically false. And the log variable which expressed the last reference argument of *lock_ref_sha1_basic()* was appended for better bug diagnosis.

Based on this observation, we design and implement LogTracker[4] to mine the correlation between logging context and modifications from historical log revisions. Since we propose logging context description model (LCDM) as an enhanced model of logging context, LogTracker can predict more intricate log revisions (see Table 6) that cannot be covered by existing tools (see Section 2.3). In summary, this paper makes the following contributions:

- Empirical study on log revisions. The results of this study show that around 74.5% of context-similar log revisions have undergone similar modifications. This shows the guiding significance of historical log revisions.
- A proactive log revision tool. With an enhanced modeling of logging context, LogTracker can recommend intricate log revisions by applying log revision rules[5] learned from software evolution history.
- Validation of the effectiveness of LogTracker. By applying generated rules to the latest versions of subject projects, LogTracker identifies 369 instances of log revisions. So far, we have reported 79 of them, and 52 have been accepted.
- Evaluation on the guidance of predicted log revisions. By recommending more instructive syntactical modifications, the average accept ratio of predicted log revisions reaches 86.4%.

This paper extends our ICPC 2018 paper "LogTracker: Learning Log Revision Behaviors Proactively from Software Evolution History" (Li et al. 2018). In this new version, we refine

[3]Log statements share semantically similar context if they print similar log variables under similar condition and are called as "context-similar log revisions" for simplification.

[4]Source code of our prototype is hosted in Github (2019).

[5]In the following sections, we will call these "rules" for simplicity.

our implementation of retrieving log modifications (see Section 3.3). Based on that, this paper recommends syntactical modifications for candidate log revisions (see Section 3.5.2) and evaluates quality of the recommendations (see Section 4.2.2). Besides, we evaluate our tool on four more projects and provide more discussion about the usefulness of log revision rules learned by LogTracker in Section 4.1.1, the precision of LogTracker in Section 4.2.1 and the future work in section 6.

The rest of paper is organized as follows. Section 2 summarizes the findings of the empirical study. Section 3 illustrates the design overview and implementation details of LogTracker. Section 4 evaluates the effectiveness and accuracy of LogTracker. Section 5 describes the limitations of this paper, Section 6 discusses the potential improvement directions of our work and Section 7 presents the related works. Lastly, we conclude our work in Section 8.

2 Motivation

In this section, we conduct an empirical study on 12 open-source projects. In order to verify our hypothesis, we answer the following three research questions (RQ). Among them, RQ1 and RQ2 aim to quantitatively evaluate the importance of providing more guidances on log revisions. RQ3 studies the feasibility of learning from software evolution history.

RQ1. How pervasive are log revisions? Considering the high log density (see Table 1), logging code may also be modified as the software evolves with bug fixes or feature updates (Meng et al. 2013; Rolim et al. 2017). Since the proportion of log revisions during software evolution is positively related to the importance of improving logging practices, this research question targets at quantitatively evaluating the pervasiveness of log revisions.

RQ2. What are the characteristics of log revisions? As pointed out by previous works (Chen and Jiang 2017a; Yuan et al. 2012b), log revisions may involve different sorts of log modifications. Existing works (Yuan et al. 2012a, c; Zhu et al. 2015; Ding et al. 2015; Zhao et al. 2017) mainly focused on certain type of log modifications (e.g., log insertion or insertion of log variables). This research question characterizes the distribution of different revision scenarios so as to evaluate the completeness of existing works.

RQ3. How many context-similar log revisions are modified similarly? Previous study (Kim et al. 2005) indicated that around 10% to 30% of the code in large projects belongs to clone code, and that 36% to 38% of clone genealogies consist of clone instances that have been systematically modified. Motivated by this observation, we assume that context-similar log statements are pervasive in software and deserve similar modifications if they are revised. In order to verify our hypothesis, this research question is proposed to measure the proportion of context-similar log revisions in evolution and how many context-similar log revisions are modified similarly (i.e., simplified as "similar log revisions").

2.1 Experimental Setup

This empirical study is conducted on 12 open-source projects in C/C++ languages. They are Httpd (Foundation 2017c), Git (Conservancy 2018), Mutt (kevin8t8 2018), Rsync (Davison 2018), Collectd (Collectd 2017), Postfix (Venema 2013), Tar (Foundation 2017a), Wget (Foundation 2017b), OpenDDS (OCI 2018), Ice (Ice 2018), GIMP (Team TG 2019) and

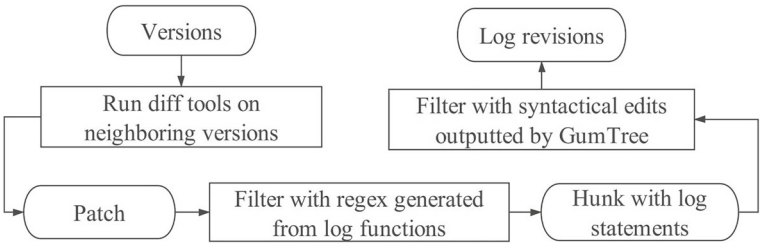


Fig. 4 The procedure of generating log revisions which serve as the input of this empirical study

Wireshark (Foundation 2019). Each of these has a development history of more than 13 years. This improves the reliability and validity of this research study.

Table 1 lists several metrics of these subject projects. Among these indicators, the line of source code (SLOC) is measured using SLOCCount (Media 2018) in order to eliminate comments and empty lines. The line of logging code (LLOC) is the total number of lines occupied by log statements.[6] The final metric evaluates the ratio of SLOC to LLOC and is inversely proportioned to log density. Despite the diversity among software, on average, one line of logging code appears for every 67 lines of code. This result is consistent with the findings of previous study (Yuan et al. 2012b; Chen and Jiang 2017a) and indicates a relatively high log density. In addition, the diversity of these indicators (especially log density) increases the generality of our research.

Figure 4 displays the process of generating the input for this empirical study. In order to collect as many log revisions as possible, we first crawl all available released versions of subject projects and then generate patches automatically by running Diffutils (Foundation 2016) on neighboring versions. For each patch, its containing hunks[7] are roughly filtered using regex to select hunks that contain log statements. The regex pattern used here is based on log functions that are recognized by traditional methods (Yuan et al. 2012a; Zhu et al. 2015). All selected hunks are then passed to GumTree (Falleri et al. 2014; Github 2018a) which generates the *syntactical edit scripts*.[8] We use these edit scripts to identify hunks that modified log statements (i.e., log hunks) rather than empty lines or comments (see Fig. 10). Meanwhile, log revisions with non-empty syntactical edit scripts on log statements, e.g., log revisions in Fig. 14, are retrieved and serve as the input of this data research.

The study methodology and main findings of our three RQs are explained in the following subsections.

2.2 RQ1: How Pervasive are Log Revisions?

In order to capture the pervasiveness of log revisions in software evolution, we evaluate two indicators that have been commonly used by previous studies (Yuan et al. 2012b; Chen and Jiang 2017a).

[6]Log statements are recognized with regex which is explained in next paragraph.

[7]A hunk is the basic unit in a patch. It begins with range information and is immediately followed with the line additions, line deletions, and any number of the contextual lines. Hunks used in this experiment contain six lines of contextual code before and after the edited code.

[8]With edit scripts as a sequences of edit actions (Falleri et al. 2014), syntactical edit scripts in this paper refers to sequences of edit actions made to syntactical structures.

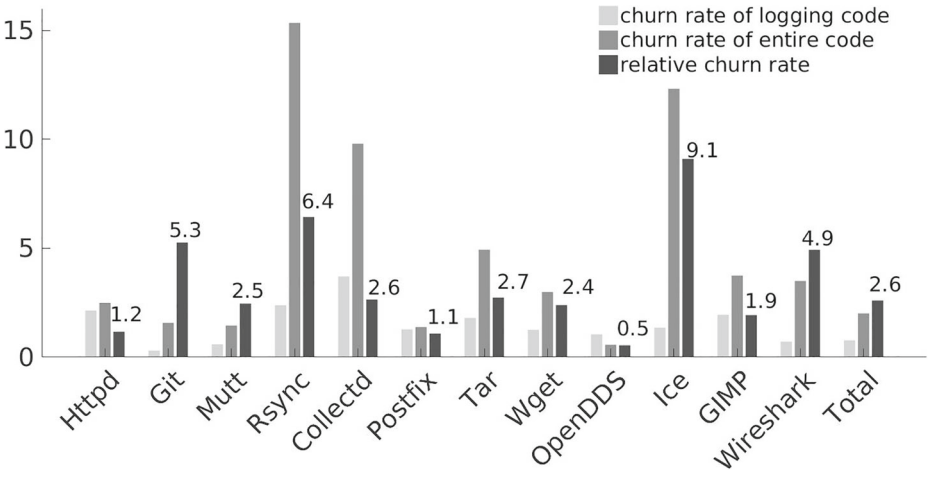


Fig. 5 Churn rate of logging code and entire code

First, we measure the relative churn rate of the logging code in comparison to the entire code. Its formula is listed as follows.

$$\begin{aligned} Relative\ churn\ rate &= \frac{Churn\ rate\ of\ the\ logging\ code}{Churn\ rate\ of\ the\ entire\ code} \\ Churn\ rate\ of\ the\ logging\ code &= \frac{Churned\ LLOC}{LLOC} \\ Churn\ rate\ of\ the\ entire\ code &= \frac{Churned\ SLOC}{SLOC} \end{aligned} \tag{1}$$

As shown in Fig. 5, the average churn rate of the logging code is 2.6 times over the churn rate of the entire code. This data is consistent with previous studies (Yuan et al. 2012b; Chen and Jiang 2017a) and indicates that logging code is modified at least as frequently as the entire code.

Second, we evaluate the density of log revisions in evolution. This metric is measured using the ratio of log hunks (i.e., hunks that contain log revisions) to all hunks, with following formula.

$$Density\ of\ log\ revisions = \frac{Log\ hunks\ in\ evolution}{All\ hunks\ in\ evolution} \tag{2}$$

This value is positively related to the pervasiveness of log revisions. The data displayed in Table 2 shows that, on average, 13.9% of revisions are about log statements. This result is also consistent with previous studies (Yuan et al. 2012b; Chen and Jiang 2017a) and indicates that log revisions are pervasive during software evolution, even when there is a comparatively low log density (1/67 in Table 1).

The pervasiveness of log revisions emphasizes the demand for improving logging practices and indicates the feasibility to learn log revision behaviors from evolution history.

2.3 RQ2: What are the Characteristics of Log Revisions?

One log statement (e.g., print("value of variable a is %d", variable_a)) consists of three components: log function (e.g., print), log variables (e.g., variable_a) and static content (e.g.,

Table 2 Revisions in software evolution

Software	Studied versions	#Log revisions	#Hunk	#Log hunk	Density of log revision
Httpd	2.0.40-2.4.27	5,347	33,422	7,435	22.3%
Git	2.0.0-2.9.5	2,002	14,432	4,380	30.4%
Mutt	1.4.2.3-1.9.1	322	3,848	686	17.8%
Rsync	0.1-3.1.2	557	8,251	1,026	12.4%
Collectd	1.7.0-5.8.0	635	18,750	2,048	10.9%
Postfix	1.1.13-3.2.4	2,222	18,909	4,877	25.8%
Tar	1.11.8-1.30	429	1,281	199	15.5%
Wget	1.5.3-1.19.2	730	9,276	1,433	15.5%
OpenDDS	1.0-3.13	1,981	21,722	3,550	16.3%
Ice	2.1.0-3.7.1	1,796	38,766	7,112	18.3%
GIMP	2.0.0-2.10.10	3,326	134,563	9,406	7.0%
Wireshark	2.0.0-3.0.2	3,090	106,981	7,551	7.1%
Total		22,437	303,220	42,152	**13.9%**

Table 3 Distribution of revisions among different categories of log modifications

Software	Total	Log Insertion	Log Deletion	Update of log function	Modification of variables	Modification of static content
Httpd	5,347	1,140 (21.3%)	553 (10.3%)	579 (10.8%)	4,252 (79.5%)	3,725 (69.7%)
Git	2,002	405 (20.2%)	224 (11.2%)	550 (27.5%)	1,768 (88.3%)	588 (29.4%)
Mutt	322	71(22.0%)	44 (13.7%)	91 (28.3%)	179 (55.6%)	128 (39.8%)
Rsync	557	78 (14.0%)	27 (4.8%)	192 (34.5%)	312 (56.0%)	73 (13.1%)
Collectd	635	133 (20.9%)	46 (7.2%)	310 (48.8%)	277 (43.6%)	158 (24.9%)
Postfix	2,222	913 (41.1%)	219 (9.9%)	269 (12.1%)	1,222 (55.0%)	909 (40.9%)
Tar	429	116 (27.0%)	58 (13.5%)	108 (25.1%)	270 (62.9%)	198 (46.1%)
Wget	730	216 (29.6%)	43 (5.9%)	102 (14.0%)	564 (77.3%)	300 (41.1%)
OpenDDS	1,981	366 (18.5%)	166 (8.4%)	102 (5.1%)	1,939 (97.9%)	1,442 (72.8%)
Ice	1,796	565 (31.5%)	342 (19.0%)	535 (29.8%)	1,133 (63.1%)	648 (36.1%)
GIMP	3,326	874 (26.3%)	263 (7.9%)	967 (29.1%)	2,398 (72.1%)	1,276 (38.4%)
Wireshark	3,090	604 (19.5%)	294 (9.5%)	995 (32.2%)	1,189 (38.5%)	1,312 (42.5%)
Total	22,437	5,481 (24.4%)	2,279 (10.2%)	4,800 (21.4%)	15,503 (69.1%)	10,757 (47.9%)

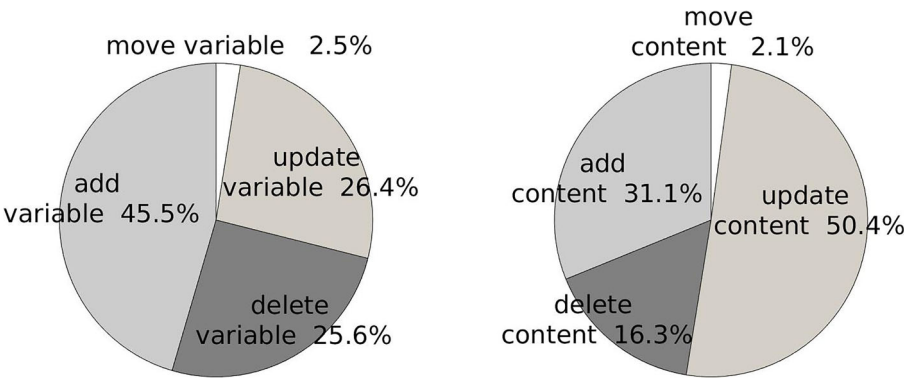


Fig. 6 Distribution of modifications of log variables (left) and modifications of static content (right)

"value of variable a is %d"). In a similar fashion to previous works (Yuan et al. 2012b; Chen and Jiang 2017a), we divide log revisions into five categories by combining edit types with edited components. These categories are log insertion, log deletion, update of log function, modification of log variables and modification of static content.

To identify the category for given log revisions, we design and implement a simple classifier. It utilizes syntactical edit scripts that are generated by GumTree to decide what components of the log statements have been modified. We sample 100 log revisions and manually justify the correctness of automatic classification. As it turns out, the accuracy of this classifier is 94.0%. With the help of this classifier, we characterize the distribution of log revisions among the five categories and display it in Table 3.

As shown in the table, 10.2% of the log revisions delete log statements, 24.4% insert new log log statements while over 69.1% of the log revisions modify log variables. In this way, the insertion, deletion, and update of log statements all happen relatively frequently during software evolution. Among the five categories, the modifications of variables and static content take larger proportion. As such, we refine these two categories into eight sub-categories, whose distribution characteristics are shown in Fig. 6. Almost half of the modifications made to variables and static content are deletions and updates, which have not been covered by existing works (Yuan et al. 2012a, c; Zhu et al. 2015). This indicates the limitation of related works and the necessity of mining rules from software evolution.

2.4 RQ3: How Many Context-Similar Log Revisions are Modified Similarly?

To capture the pervasiveness of context-similar log revisions, we evaluate the proportion of context-similar log revisions to all log revisions. As a prerequisite, we run hierarchical clustering algorithm (Defays 1977) on historical log revisions with the semantics of logging context[9] as the feature vector.[10] This generates groups whose members all share semantically similar context (i.e., *context-similar revision groups*). Then, we compute the number of context-similar log revisions by summarizing all instances in *context-similar revision groups*.

[9]Logging context model used to describe the semantics context of log revisions is explained detailedly in Section 3.2.

[10]For consideration of accuracy, clustering algorithm used in this paper takes the threshold of similarity as one.

Table 4 The ratio of context-similar log revisions (Context-similar groups: context-similar revision groups, Average Group Size: average group size of context-similar revision groups, Similar groups: similar revision groups)

Software	Total Revisions	Context-similar Revisions	Ratio	Context- similar Groups	Average Group Size	Similar Groups
Httpd	5,347	2,733	51.1%	894	3.1	587
Git	2,002	868	43.4%	272	3.2	188
Mutt	322	103	32.0%	37	2.8	25
Rsync	557	393	70.6%	124	3.2	91
Collectd	635	424	66.8%	139	3.1	155
Postfix	2,222	1,200	54.0%	380	3.2	286
Tar	429	184	42.9%	57	3.2	36
Wget	730	338	46.3%	119	2.8	60
OpenDDS	1,981	1291	65.2%	350	3.7	295
Ice	1,796	1,075	59.9%	321	3.3	241
GIMP	3,326	1,889	56.8%	627	3.0	412
Wireshark	3,090	2,294	74.2%	486	4.7	459
Total	22,437	12,792	**57.0%**	3,806	3.4	3,031

Table 4 displays the detailed data. On average, 57.0% of historical log revisions share semantically similar logging context. That is to say, a comprehension of log revision behaviors may allow us to predict half of the revisions on log statements. This result indicates the potential effectiveness of rules that are mined from software evolution history.

Additionally, for judging whether context-similar log revisions deserve similar modifications, we measure the proportion of similar modifications in context-similar log revisions. We first classify log revisions that not only share similar logging context, but also undergo similar modifications[11], thus generating *similar revision groups*. From this, it is obvious that the *similar revision groups* are a subset of *context-similar revision groups*, and that the ratio of the former to the latter is positively related to the proportion of similar modifications in context-similar log revisions.

As shown in Fig. 7, on average, 74.5% of the *context-similar revision groups* show similar modifications. This result validates our assumption that log statements with semantically similar context deserve similar modifications. Hence, it is reasonable to apply modifications that are learned from evolution history to log statements that share semantically similar context with historical revision behaviors.

3 Design and Implementation

3.1 Overview

In order to provide guidances on logging practices, this paper designs and implements LogTracker, which can predict intricate log revisions by mining the correlation between logging context and log modifications. In this section, we detail the implementation of LogTracker.

As shown in Fig. 8, LogTracker consists of two main phases. The first phase involves mining rules from software evolution. When given one log revision, LogTracker should

[11] This paper models log modifications based on syntactical edit scripts, see Section 3.3 for more details

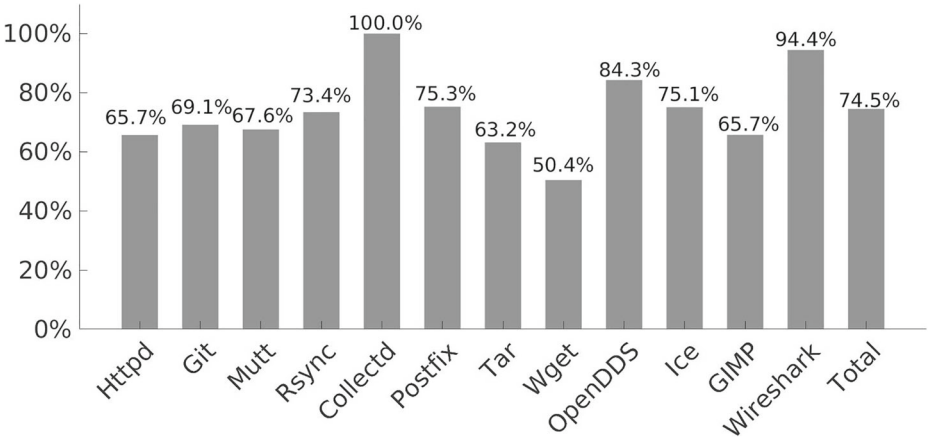


Fig. 7 Pervasiveness of similar modifications among context-similar log revisions

first analyze the semantics of the logging context and retrieve log modifications. These then serve as the input of generating rules. In the second phase, LogTracker suggests log modifications for candidate code snippets by applying rules. In summary, there are four modules in LogTracker, detailed below.

Extracting the semantics of logging context. This module analyzes the semantics of logging context for log statements. Since LogTracker aims to suggest modifications for log statements that share similar logging context, the precision of this module seriously affects the effectiveness of the whole tool. In Section 3.2, we illustrate the design and implementation details of this module.

Retrieving log modifications. This module utilizes syntactical edit scripts to represent log modifications. Section 3.3 explains how we handle these edit scripts.

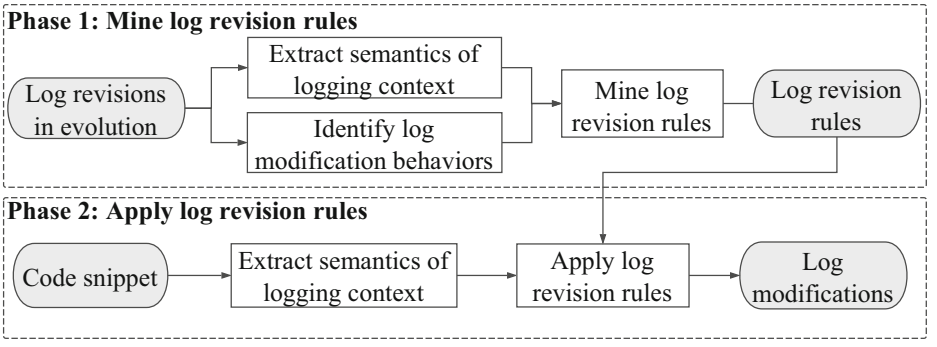


Fig. 8 Architecture of LogTracker

Mining log revision rules. In this module, historical log revisions are classified into multiple groups according to their logging context and log modifications. One group is treated as one rule. One rule consists of two elements: logging context and modifications. This indicates that those log statements that fall under the logging context deserve such modifications. The details of how the rules are produced are illustrated in Section 3.4.

Applying log revision rules. This module applies the learned rules to code snippets which share semantically similar logging context. For each candidate code snippet, syntactical log modifications will be suggested. More details about locating candidates and generating recommendations are provided in Section 3.5.

3.2 Extracting the Semantics of Logging Context

As mentioned above, the understanding of semantics of logging context seriously affects the effectiveness of any recommendations. This subsection illustrates how LogTracker extracts logging context.

Software: Postfix, file: util/poll_fd.c

```
switch (select(fd + 1, read_fds, write_fds, &except_fds, tp)) {
 case -1:
  if (errno != EINTR)
   msg_fatal("select: %m");
```

(a) One log statement that shares similar logging context with Figure 9b

Software: Postfix, file: util/events.c

```
event_count = select(event_max_fd + 1, &rmask, &wmask, &xmask, tvp);
if (event_count < 0) {
 if (errno != EINTR)
  msg_fatal("event_loop: select: %m");
```

(b) One log statement that shares similar logging context with Figure 9a

Software: Git, file: fast-import.c

```
buf = gfi_unpack_entry(myoe, &size);
if(!buf)
  die("Can't load tree %s", sha1_to_hex(sha1))
```

(c) One log statement that shares different logging context with Figure 9d

Software: Git, file: builtin/unpack-objects.c

```
obj_buf = lookup_object_buffer(obj);
if(!obj_buf)
  die("Whoops! Cannot find object '%s'", sha1_to_hex(obj->sha1));
```

(d) One log statement that shares different logging context with Figure 9c

Fig. 9 Real-world log statements that are with similar and different logging context

3.2.1 Semantics of Logging Context

It is worth noting that understanding the semantics of logging context is challenging. First, Logging context with similar semantics may correspond to several syntactical representations. In Fig. 9a and b, the two log statements both print messages when the return value of *select()* is negative and *errno* is not *EINTR*. That is to say, despite the differences in syntactical structures, the semantics of these two logging context are similar. In this case, traditional algorithms (Jiang et al. 2007; Gabel et al. 2008; Juergens et al. 2009) that approximate semantics using syntactics may fail to recognize some semantically similar logging context. Second, the length of logging context is usually so short that traditional algorithms (Jiang et al. 2007; Gabel et al. 2008; Juergens et al. 2009) which calculate code similarity based on syntactical structures may raise false alarms. While the two log statements in Fig. 9c and d share similar syntactical structures, their semantics vary a great deal. Specifically, the log statement in Fig. 9c prints log messages when the return value of *gfi_unpack_entry()* is logically false while the log statement in Fig. 9d prints messages when the return value of *loopup_object_buffer()* is logically false.

In summary, traditional algorithms (Jiang et al. 2007; Gabel et al. 2008; Juergens et al. 2009) that describe the semantics of context of functional code fail when used on logging context. To overcome this challenge, we design LCDM to accurately describe the semantics of logging context.

3.2.2 Logging Context Description Model

Logging context is generally made up of two components: where to log and what to log. Where to log indicates that under what conditions log messages should be printed so that it can be modeled by check conditions of log statements. And what to log describes what variables to output and depends on the semantics of log variables. In this case, the task of understanding the semantics of logging context can be split into a comprehension of the check conditions and log variables. Consequently, this paper defines the Logging Context Description Model as a 2-tuple $m=<c,v>$, where c means the check conditions and v means the log variables.

$$\begin{aligned} c &= lists\ of\ related\ functions \\ related\ function &= < relation\ type,\ function\ name\ (, index) > \\ relation\ type &\in \{RET\_VAL, REF\_ARG, ARG\} \end{aligned} \tag{3}$$

Log statements are usually controlled by branch statements that check one or multiple variables which are related to error-prone functions. For example, log statement in Fig. 9c depends on a branch statement that validates the return value of an error-prone function named *gfi_unpack_entry*. Thus, LCDM models the check conditions of one log statement with the semantics of its validated variables. Specifically, c is formulated as lists of error-prone functions that are related to the validated variables (see Formular (3)). Each related function is made up of the function name and type of relation between this function and the validated variable.

There are mainly three sorts of relations: Return_VALUE (i.e., RET_VAL in formulas), REFERENCE_ARGUMENT (i.e., REF_ARG in formulas) and ARGUMENT (i.e., ARG in formulas). Relation of type Return_VALUE indicates write dependences through the return

value of functions (e.g., *a=fun()*). REFERENCE_ARGUMENT means write dependences on functions by reference arguments (e.g., *fun(&a)*). Relation of type ARGUMENT represents read dependences on functions by arguments (e.g., *fun(a)*). For relations of type REFERENCE_ARGUMENT and ARGUMENT, another index value is used to point out the position of given argument in the argument list (e.g., index of a in *fun(&a)* is 1). For instance, the variable *buf* in Fig. 9c related to *gfi_unpack_entry()* through return value. Thus, the type of relation between *buf* and *gfi_unpack_entry()* is Return_VALUE.

Similar to validated variables, log variables may also relate to error-prone functions. In addition to the related functions, variable types (especially self-defined variable types) also imply the semantics of log variables. As such, LCDM formulates v as lists of semantics expressions which represent the semantics of log variables by either the related functions or the variable types (see Formular (4)). To do this, LCDM implements three levels of semantics: RELATED_FUNCTION (i.e., RLAT_FUN in following formula), VARIABLE_TYPE (i.e., VAR_TYPE) and NO_INFORMATION (i.e., NO_INFO).

$$\begin{aligned} v &= lists\ of\ semantics\ expressions \\ semantics\ expression &= < semantics\ level, value > \\ semantics\ level &\in \{RLAT\_FUN, VAR\_TYPE, NO\_INFO\} \end{aligned} \tag{4}$$

In detail, each semantic expression consists of semantics level and value. Actually, different levels of semantics correspond to different sorts of value. For level of RELATED_FUNCTION, value is the related function (shown in formula (3)) of one variable. For level of VARIABLE_TYPE, value is the data type of one variable (e.g., value of *a* in "*int a;*" is *int*). If neither the related function nor the variable type can be inferred in limited context (e.g., function that contains this log statement), value is NULL and the semantics level is marked as NO_INFORMATION.

In order to aid understanding, we illustrate LCDM with the initial code snippet of the patch in Fig. 3a as an example:

```
char* newrefname ...
lock=lock_ref_sha1_basic(oldrefname,NULL,NULL,0,NULL);
if (!lock) {
 error("unable to lock %s for rollback",oldrefname);
```

The LCDM of this logging context equals to *<[<Return_VALUE, lock_ref_sha1_basic>], [<RELATED_FUNCTION, <REFERENCE_ARGUMENT, lock_ref_sha1_basic, 1>>]>*. It indicates that this log statement will output the first argument of *lock_ref_sha1_basic()* if the return value of *lock_ref_sha1_basic()* is logically false.

3.2.3 Extracting LCDM

Before illustrating the implementation details of extracting LCDM, we should first identify which log statement to deal with. In reference to Fig. 8, LogTracker should analyze LCDM for historical log revisions when mining rules, and for candidate code snippets (i.e., the code snippet where rules apply) when applying rules. Specially, when applying rules to one candidate code snippet, LogTracker needs to extract LCDM for its inner log statements (see Section 3.5).

Algorithm 1 Extracting semantics expression for one log variable

Require: syntactical structure and PDG of function body
Ensure: variable node n, level of semantics: $level$ and value of semantics: $value$
1: $level$ = NO_INFORMATION; $value$ = NULL
2: **if** n invokes a function $f^{'}$ **then**
3: $level$= RELATED_FUNCTION;
$value$= <RETURN_VALUE, name of $f^{'}$>; goto line 20;
4: **end if**
5: from n traverse upward PDG
6: **if** n has write dependence on function $f^{'}$ through return value **then**
7: goto line 3;
8: **end if**
9: **if** n has write dependence on function $f^{'}$ by the no. i reference argument **then**
10: $level$ = RELATED_FUNCTION;
$value$= <REFERENCE_ARGUMENT, name of $f^{'}$, i>; goto line 20
11: **end if**
12: from n traverse downward PDG
13: **if** n has read dependence on function $f^{'}$ by the no. i argument **then**
14: $level$ = RELATED_FUNCTION;
$value$= <ARGUMENT, name of $f^{'}$, i>; goto line 20
15: **end if**
16: from n traverse upward PDG
17: **if** n declared to be variable type of $type$ **then**
18: $level$ = VARIABLE_TYPE;
19: $value$= $type$; goto line 20
20: **end ifreturn** <$level$, $value$>

When mining rules from historical log revisions, the input log statement is related to the category of log modifications. In detail, if the historical log revision modifies existing log statement, LogTracker extracts LCDM of the modified log statement (i.e., old log statement). While if the historical log revision inserts new log statements, the inserted log statement (i.e., new log statement) will be analyzed.

Given the input log statement, the primary task when extracting LCDM is to identify c and v. As mentioned in Section 3.2.2, c consists of the related functions of validated variables and v consists of either the related functions or the variable types of log variables. Technologically speaking, extraction of c is just a special case of identifying v. Thus, this section focuses on explaining how to extract v for one log statement.

As the basic unit of v, extraction of semantics expression for one log variable is shown in Algorithm 1. This procedure requires two inputs: the syntactical structure and the program dependence graph (PDG) of the function body. As a prerequisite, we should first trace back to the function body of the input log statement. To do this, we identify the location of input log statement in the source file by analyzing the range information in the hunk. Then, by traversing the ancestor nodes of input log statement in source file, we extract the function body. With the function body, the syntactical structure is generated by srcML (Collard et al. 2013) which translates incomplete code into a syntactical unit. In order to retrieve PDG of the function body, We then process the syntactical structure of the function body to build a partial PDG, which illustrates data dependencies.

Given the syntactical structure and PDG of the function body, we first identify the syntactical node of one log variable before extracting its semantical expression. Specifically, if this variable node indicates an invocation of function, the invoked function will be identified as relating to this variable in type of Return_VALUE. Meanwhile, this semantics expression belongs to level of RELATED_FUNCTION. Otherwise, the PDG of function body is traversed upward from this variable node to search for functions related to it by either reference arguments (i.e., relation of type REFRENCE_ARGUMENT) or return values (i.e., relation of type Return_VALUE). If fails, the program traverses downward the PDG to search for read dependences by arguments (i.e., relation of type ARGUMENT). If either of above processes succeeds, program will exit with level as RELATED_FUNCTION and value as corresponding related function; otherwise, the program continues to traverse upward the PDG to search for the declaration of this variable node. If succeeds, the program exits with level as VARIABLE_TYPE and value as declared type. If not, it exits with level as NO_INFORMATION and value as NULL.

We explain the extraction of LCDM with the same instances in Section 3.2.2. The branch statement and its dependent statement are:

```
lock=lock_ref_sha1_basic(oldrefname,NULL,NULL,0,NULL);
if (!lock) {
```

This branch statement only validates one variable which corresponds to the invocation of *lock_ref_sha1_basic()*. Thus, the value of *c* is *[<Return_VALUE, lock_ref_sha1_basic>]*. The log statement is:

```
error("unable to lock %s for rollback",oldrefname);
```

This log statement prints one log variable which is the first reference argument of *lock_ref_sha1_basic()*. Thus, there is a write dependence between this log variable and *lock_ref_sha1_basic()*. And the value of *v* is *[<RELATED_FUNCTION, <REFERENCE_ARGUMENT, lock_ref_sha1_basic, 1>>]*.

3.3 Retrieving Log Modifications

Retrieving log modifications is another prerequisite when learning log revision behaviors. This paper represents log modifications based on syntactical edit scripts.

Syntactical edit scripts of two log statements express the differences between their syntactical structures and can eliminate the interference from comments or empty lines. For example, the patch in Fig. 10 inserted an empty line among the static content and log variables, but did not modify the syntactics of the log statement.

Httpd-2.0.64 vs Httpd-2.0.65, file: mod_nw_ssl.c, line: 668

```
 if (!found) {
  ap_log_perror(APLOG_MARK, APLOG_WARNING, 0, plog,
-  "No Listen ... %s:%d",slu->addr,slu->port);
+  "No Listen ... %s:%d",
+     slu->addr, slu->port);}
```

Fig. 10 Patch that inserted empty line in log statement

In this paper, we generate syntactical edit scripts using GumTree that implement a state of art tree differentiating algorithm. The syntactical edit scripts for the old and new log statements[12] in Fig. 3a are shown as follows.

Insert a variable whose text is *err.buf* after a variable whose text is *oldrefname*.
Update text of one literal node from "..." to "...: %s".

Considering the limited information of variable names, we enhance above syntactics edit scripts with semantics of log variables to reach a better understanding of log modifications. In detail, we replace variables with their corresponding semantics expressions.

For example, the *err.buf* in above syntactical edit scripts can be enhanced with *<REFERENCE_ARGUMENT, lock_ref_sha1_basic, 7>* since it is the last reference argument of *lock_ref_sha1_basic()*. As such, syntactical edit scripts turn to be:

Insert a variable whose semantics expression is *<REFERENCE_ARGUMENT, lock_ref_sha1_basic, 7>* after a variable whose text is *oldrefname*.
Update text of one literal node from "..." to "...: %s".

Meanwhile, the *oldrefname* in this syntactical edit scripts can be enhanced with *<REFERENCE_ARGUMENT, lock_ref_sha1_basic, 1>* since it is the first reference argument of *lock_ref_sha1_basic()*. This turns the syntactical edit to be as follows:

Insert a variable whose semantics expression is *<REFERENCE_ARGUMENT, lock_ref_sha1_basic, 7>* after a variable whose semantics expression is *<REFERENCE_ARGUMENT, lock_ref_sha1_basic, 1>*.
Update text of one literal node from "..." to "...: %s".

The final syntactical edit scripts are then digitalized with traditional hash algorithm (Foundation 2018) for better performance during rule mining. Comparing with textual edit scripts in our previous work (Li et al. 2018), this enhanced syntactical edit scripts eliminate the limitation of variable names and turn out to find more log revision rules (see Table 5).

3.4 Mining Log Revision Rules

One rule consists of two parts: logging context and log modifications. By combining LCDM and edit scripts, we can define the rule as 3-tuple $r = <c,v,e>$, where c and v compose LCDM, which describes the semantics of logging context, and e is syntactical edit scripts. One rule indicates that if one code snippet shares similar context with $<c,v>$, it deserves modifications represented by e.

Similar to log revisions during software evolution, generated rules also involve various categories of log modifications, such as log insertion, log deletion, and update of log function (i.e., Tables 3 and 6). Specifically, given one rule, if it modifies existing log statement, then log statements whose context is similar to $<c,v>$ deserve e. If it inserts new log statement, then code snippets that meet with similar $<c,v>$ may also need new log statements.

In order to mine rules, we run the agglomerative hierarchical clustering algorithm (Defays 1977) on historical log revisions with a feature vector consisting of LCDM and edit scripts. Each generated group is recognized as one rule. Every instance in a group can be treated as supporters of this rule, which is positively related to its reliability. Conservatively,

[12]Given one revision, if its category is "log deletion", the new log statement is marked as empty string. Similarly, if its category is "log insertion", the old log statement is empty string.

Table 5 Rules learned from historical log revisions and candidates detected in the latest releases (L-Rules: learned rules, W-Rules: learned rules that recommend candidates)

Software	Studied versions	Modify existing log statements			Insert new log statements		
		L-Rules	W-Rules	Candidate	L-Rules	W-Rules	Candidate
Httpd	2.0.40-2.4.27	465	18	52	122	11	61
Git	2.0.0-2.9.5	146	9	28	42	7	29
Mutt	1.4.2.3-1.9.1	20	1	3	5	0	0
Rsync	0.1-3.1.2	79	1	1	12	1	1
Collectd	1.7.0-5.8.0	106	7	16	49	9	27
Postfix	1.1.13-3.2.4	161	6	21	125	14	41
Tar	1.11.8-1.30	27	2	3	9	2	3
Wget	1.5.3-1.19.2	37	1	6	23	3	3
OpenDDS	1.0-3.13	248	7	24	47	0	0
Ice	2.1.0-3.7.1	181	4	15	60	1	4
GIMP	2.0.0-2.10.10	298	7	12	114	3	6
Wireshark	2.0.0-3.0.2	362	5	8	4	5	
Total		**2,130**	**68**	**189**	**705**	55	**180**

we select groups that have at least two voters as effective rules (In the following sections, we take effective rules as "rules" for simplicity).

In addition, as pointed out by our previous work (Li et al. 2018), some log revisions may be reverted during evolution, which causes "out-of-date" rules. For ease of this problem, we first identify rules that work on the same log statement by comparing their edit scripts and source file information. Then, we erase the older one by comparing their version information. This promises a smaller rejection ratio in the experiment Section 4.1 and a higher precision in Section 4.2.1.

3.5 Applying Log Revision Rules

This module applies learned rules by first locating candidate code snippets (see Section 3.5.1) and then recommending corresponding syntactical log modifications (see Section 3.5.2) for them.

3.5.1 Locating Candidate Code Snippets

Rules that modify existing log statements indicates that log statements whose logging context is similar to their rule context (i.e., $<c,v>$) deserve the specified modifications (i.e., e). Given code snippet, we first recognize all the inner log statements and extract their logging context. Then, to decide which rule should be applied to which log statement, we pairwise calculate the similarity between the candidate logging context and the rule context. For each candidate pair with a high similarity,[13] we further validate the feasibility of these modifications on the log statement before recommending it to the developers by simply checking the existence of modified components.

[13]For reducing false alarms, we only recommend revisions if the similarity of candidate pair is 100%.

In contrast, rules that insert new log statements only indicate that there should be one log statement under the rule context (i.e.,$<c,v>$). When locating candidates for these rules, we judge whether all related functions in the rule context are contained in the functions invoked in given code snippets.[14] If true, we further validate the necessity of inserting new log statements before informing the developers. For example, if there already exists log statement under that rule context, we do not need to log. To do this, we compare the rule context with the logging context of existing log statements and only make suggestion when there is no matching.

3.5.2 Generating Candidate Log Modifications

In order to recommend instructive log modifications when applying rules, this paper aims to generate candidate syntactical edit scripts for candidate code snippets based on historical syntactical edit scripts mentioned in Section 3.3.

For rules that modify existing log statements, we first identify which syntactical nodes to be modified by mapping the modified nodes to candidate log statements that are located in Section 3.5.1. Then, modified nodes in the syntactical edit scripts are substituted with corresponding ones in candidate log statements to generate candidate syntactical log modifications. For example, when applying the rule learned from log revision in Fig. 3a, LogTracker will locate following candidate log statement (i.e., the one in Fig. 3b) since it shares similar logging context with the rule.

```
char* refname ...
lock=lock_ref_sha1_basic(refname,sha1,NULL,0,&type);
if (!lock)
  return error("cannot lock ref '%s'", refname);
```

Then, by analyzing the mapping relationship generated by GumTree, LogTracker realized that historically modified variable *oldrefname* corresponds to the *refname* in the candidate log statement. After substituting the *oldrefname* with *refname*, the candidate syntactical log modifications turn to be:

Insert a name node whose semantics expression is *<REFERENCE_ARGUMENT, lock_ref_sha1_basic, 7>* after a variable whose text is ***refname***.
Update text of one literal node from "..." to "...: %s".

As for rules that insert new log statements, they just indicate there should be one log statement under given logging context. Thus, we inform developers that log statements are needed in candidate code snippets (i.e., the one located in the Section 3.5.1) and directly recommend historical syntactical edit scripts as references for what kinds of log statements should be inserted.

4 Evaluation

This section evaluates the performance of LogTracker from three aspects. Section 4.1 measures its effectiveness on learning log revision rules and suggesting missed log revisions.

[14]For rules that insert new log statements, we split code snippets on basis of function.

Section 4.2 evaluates its precision and recall when locating log revisions as well as the quality of recommended syntactical log modifications. In Section 4.3, we measure the precision of LCDM by locating context-similar log revisions with LogTracker. Besides, we compare LCDM with DECKARD+ (Gabel et al. 2008; GitHub 2018b) to prove the advantages of LCDM over traditional context modeling algorithms.

4.1 Effectiveness of LogTracker

The aim of LogTracker is to guide intricate log revisions by learning from software evolution. Thus, this section first studies the usefulness of rules learned by LogTracker (see Section 4.1.1), including the quantity of rules learned in subject projects, the distributions of these rules among different categories of log revisions, the trivialness of these rules, and their dispersion among files. After that, we further study whether LogTracker is able to detect missed log revisions in the latest releases (see Section 4.1.2).

4.1.1 Usefulness of Log Revision Rules Learned by LogTracker

To evaluate the ability of LogTracker on mining log revision behaviors, this subsection characterizes log revisions rules which are learned by LogTracker. Specifically, we train

Table 6 Distribution of learned rules among different categories of log modifications

Software	Total	Log Insertion	Log Deletion	Update of log function	Modification of variables	Modification of static content
Httpd	587	122 (20.8%)	70 (11.9%)	64 (10.9%)	386 (65.8%)	350 (59.6%)
Git	188	42 (22.3%)	30 (16.0%)	36 (19.1%)	121 (64.4%)	38 (20.2%)
Mutt	25	5 (20.0%)	4 (16.0%)	9 (36.0%)	9 (36.0%)	7 (28.0%)
Rsync	91	12 (13.2%)	2 (2.2%)	8 (8.8%)	69 (75.8%)	4 (4.4%)
Collectd	155	49 (31.6%)	13 (8.4%)	54 (34.8%)	43 (27.7%)	44 (28.4%)
Postfix	286	125 (43.7%)	33 (11.5%)	27 (9.4%)	103 (36.0%)	100 (35.0%)
Tar	36	9 (25.0%)	5 (13.9%)	12 (33.3%)	17 (47.2%)	19 (52.8%)
Wget	60	26 (43.3%)	2 (3.3%)	4 (6.7%)	34 (56.7%)	23 (38.3%)
OpenDDS	295	47 (15.9%)	22 (7.5%)	8 (2.7%)	290 (98.3%)	203 (68.8%)
Ice	241	60 (24.9%)	49 (20.3%)	85 (35.3%)	112 (46.5%)	72 (29.9%)
GIMP	412	114 (27.7%)	32 (7.8%)	119 (28.9%)	264 (64.1%)	137 (33.3%)
Wireshark	459	97 (21.1%)	51 (11.1%)	145 (31.6%)	146 (31.8%)	156 (34.0%)
Total	2,835	708 (25.0%)	313 (11.0%)	571 (20.1%)	1,594 (56.2%)	1,153 (40.7%)

LogTracker with all historical log revisions of the 12 subject systems to generate log revision rules. Table 5 displays the detailed number of log revision rules. In general, LogTracker mines 2130 pieces of rules that modify existing log statements and 705 rules that insert new log statements. This improves our previous data (Li et al. 2018) since the syntactical edit scripts model log modifications in a more general manner.

With a summary of 2835 pieces of rules mined, this paper concludes that LogTracker is able to automatically mine log revision behaviors from software evolution history. Despite the quantity of learned rules, in order to discuss the usefulness of these rules, we further study whether learned rules can cover different categories of log revisions, how many of them are non-trivial ones which have impacts on software behaviors, and how many remain in various files which are difficult for manual location.

What are the characteristics of the learned rules? As shown in Table 3, log revisions may involve multiple categories of log modifications, such as log deletion, update of log functions, and modification of log variables and so on. In order to tell whether LogTracker is able to understand diverse types of log revisions in evolution history, we calculate the distribution of learned rules among the five categories of log modifications.

Similar to Section 2.3, we classify log revision rules into five categories on basis of syntactical edit scripts and display the result in Table 6. Generally speaking, LogTracker successfully cover five categories of log revisions.

Furthermore, we would like to know whether revision rules and historical log revisions share similar distribution characteristics. To do this, we show their distribution vividly in Fig. 11. In general, except Rsync and Wget, other ten subject systems share nearly the similar distributions.

Are learned rules trivial or not? Previous works (Mondai et al. 2018; Kawrykow and Robillard 2011) pointed out that trivial modifications are less meaningful, including rename-inducing modifications, local variable extractions, and whitespace updates. As

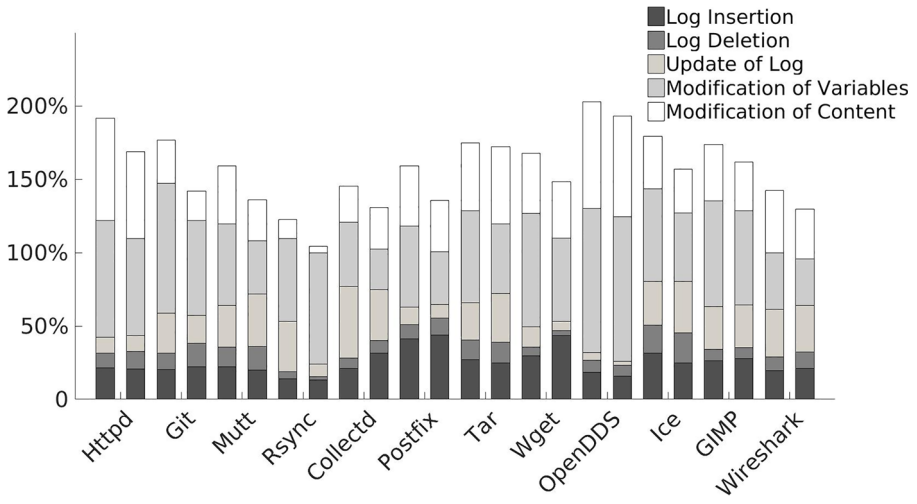


Fig. 11 Distribution characteristics of historical revisions and learned log revision rules (For each software, left column is distribution of log revisions and right column is distribution of rules.)

OpenDDS-1.1 vs OpenDDS-1.2, file: examples/DCPS/Messenger_Imr/subscriber.cpp, line: 62

```
 while ((c = get_opts ()) != -1)
 {
   switch (c)
   ...
   default:
    ACE_ERROR_RETURN ((LM_ERROR,
-    "usage:  %s "
-    "-t <tcp/udp/default> "
-    "\n",
+   ACE_TEXT("usage:  %s -t <tcp/udp/default> \n"),
    argv [0]),
    -1);
```

(a) Trivial log revisions which only update the empty line of log message

OpenDDS-3.13, file: tests/DCPS/ManualAssertLiveliness/subscriber.cpp, line: 58

```
 while ((c = get_opts ()) != -1)
 {
   switch (c)
   ...
   default:
    ACE_ERROR_RETURN ((LM_ERROR,
    "usage:  %s "
    "\n",
    argv [0]),
    -1);
```

(b) Candidates detected with rules learned from the trivial log revision in Figure 12a

Fig. 12 Example of trivial log revisions and candidate detected by the rule learned from trivial log revisions

shown in Fig. 12a, some log revisions may be trivial ones and have no impacts on software behaviors. In this case, rules mined from trivial log revisions are less valuable. Thus, we measure the proportion of *trivial log revision rules* which only involve with trivial log revisions to evaluate the usefulness of learned rules.

In detail, we randomly sample 510 pieces[15] of log revision rules from the 12 projects and identify whether one rule is a trivial one by manually checking whether the involved log revisions are trivial modifications on log statements.

As displayed in Table 7, on average, only 13.3% of learned rules are trivial ones and 86.7% of them are meaningful and have influences on software behaviors. This indicates the potential usefulness of LogTracker to mine non-trivial revision behaviors from software evolution history.

Are the rules learned from different files? Comparing with revisions in the same file, revisions which reside in different files are much more difficult to track. Thus, this paper evaluates how many rules are learned from revisions which reside in different files to judge the usefulness of learned rules.

[15]Confidence interval is 3.93 with a confidence level as 95%. This is calculated with Sample Size Calculator (Systems CR 2019).

Table 7 Proportion of rules that are learned from different files and proportion of trivial revision rules (D-Rules: rules that are learned from different files)

Software	Groups	D-Rules	Ratio of D-rules	Sample rules	Trivial rules	Ratio of Trivial rules
Httpd	587	257	43.8%	50	10	20.0%
Git	188	85	45.2%	50	11	22.0%
Mutt	25	2	8.0%	10	2	20.0%
Rsync	91	66	72.5%	50	3	6.0%
Collectd	155	101	65.2%	50	7	14.0%
Postfix	286	172	60.1%	50	6	12.0%
Tar	36	14	38.9%	20	2	10.0%
Wget	60	20	33.3%	30	4	13.3%
OpenDDS	295	117	39.7%	50	8	16.0%
Ice	241	112	46.5%	50	6	12.0%
GIMP	412	144	35.0%	50	5	10.0%
Wireshark	459	212	46.2%	50	4	8.0%
Total	2,835	1,302	**45.9%**	510	68	**13.3%**

As for implementation, given one rule, we collect all of its corresponding historical log revisions. With location information of each historical log revision, it is easy to judge whether the rule is learned from revisions that spreads in different files or not.

As displayed in Table 7, despite the diversity among software, averagely speaking, 45.9% of rules are learned from log revisions which reside in different files. With near half of rules involved with revisions that spread in various files, it turns more important to automatically mine rules with LogTracker which can provide guidances on locating and modifying these log revisions.

In summary, LogTracker is able to learn 2,835 pieces of revision rules from the 12 subject systems. Meanwhile, rules learned by LogTracker is proved to be useful. First, they cover a variety categories of log revisions. Second, average speaking, 86.7% of the learned rules involve non-trivial modifications on log messages. Third, 45.9% of them are learned from revisions in different files which are difficult for manually tracking.

4.1.2 Ability to Detect Missed Log Revisions

As mentioned by previous works (Meng et al. 2013; Rolim et al. 2017; Chen and Jiang 2017b), developers may miss some systematic edits. By learning log revision behaviors from historical log revisions, LogTracker can detect missed log revisions that share similar logging context with rules. This section evaluates how effective LogTracker is at detecting log revisions that are missed by developers.

To do this, rules listed in Table 5 are applied to the latest versions of the 12 subject projects to detect missed log revisions. Then, we manually validate the feasibility and reasonableness of recommended revisions[16] and summarize 369 true positives (see Table 5) from 12 software.

[16]As mentioned in Section 3.5.1, LogTracker automatically filters infeasible log revisions, while for considering of accuracy, we also manually verify the correctness of automatic filtering.

We are in the process of reporting those recommendations to developers for feedback. Up to now, we have reported 79 instances. 52 (65.8%) instances have been accepted by developers, 8 (10.1%) instances are under discussion,[17] 19 (24.1%) instances has been rejected. As we have discussed in Section 3.4, erasing of rules that are learned from the reverted log revisions decreases the rejection ratio of our work.

Here, we illustrate one accepted instance. This instance is detected by the rule generated from four log revisions in *Git-2.3.10*. One is from file *builtin/merge-tree.c* with code as follows.

```
- xdi_diff(&src, &dst, &xpp, &xecfg, &ecb);
+ if (xdi_diff(&src, &dst, &xpp, &xecfg, &ecb))
+  die("unable to generate diff");
  free(src.ptr);
```

This revision inserted check of the return value of *xdi_diff()* and one log statement. Hence, LogTracker learns a rule that *xdi_diff()* should be checked and logged. By applying this rule, LogTracker detects the missed log revision in *Git-2.14.2 builtin/rerere.c*. The initial code is as follows.

```
ret = xdi_diff(&minus, &plus, &xpp, &xecfg, &ecb);
free(minus.ptr);
```

xdi_diff() is invoked without validating the return value. We report this to the mailing list of Git and the developer accepted this instance.

As for the rejected instances, there are mainly three reasons. First, log revisions are related to other code and may cause too many dependent modification (*"That's much bigger than a single-line change, since groups of dependent functions need to be converted."*).

Second, log revisions are trivial ones so that developers do not want to modify code for something have no impacts on software behaviors. For example, candidate shown in Fig. 12b is rejected for this reason (*"These other changes that just combine multiple lines aren't necessary."*).

Third, recommended log revisions are not suitable for some special cases despite the correctness of learned rules. For example, LogTracker learns one rule that the return value of *setsockopt()* should be checked and logged from two revisions in *Wget-1.18 src/connect.c*. One of them is listed as below.

```
-  setsockopt (sock,SOL_SOCKET,SO_REUSEADDR,setopt_ptr,
   setopt_size);
+  if(setsockopt (sock,SOL_SOCKET,SO_REUSEADDR,setopt_ptr,
   setopt_size))
+    logprintf (LOG_NOTQUIET, _("setsockopt SO_REUSEADDR
     failed: %s\n"),
+               strerror (errno));
```

[17] We found that candidates posted in Github are more possible to be replied. In fact, 29 candidates detected in OpenDDS, Ice and GIMP are both replied in time since their issues are managed with Github.

With this rule, we find one missed spot in *Wget-1.19.2 lib/setsockopt.c*.

```
int rpl_setsockopt
(int fd, int level, int optname, const void *optval,
socklen_t optlen)
{  ...
   r = setsockopt (sock, level, optname, optval, sizeof (int));
    ...
```

Developer agreed with the reasonableness of our finding but pointed out this invocation should not be checked or logged because *"rpl_setsockopt is a replacement function for systems where setsockopt doesn't behave in a sane manner"*.

4.2 Accuracy of LogTracker

As a tool for guiding log revisions, the accuracy of LogTracker matters when putting it into practice. In this section, we evaluate the accuracy of LogTracker from two aspects. First, the precision and recall is calculated to test how accurate LogTracker can be when locating candidate code snippets. Second, we manually compare recommended edit scripts generated by LogTracker against historical log revisions to evaluate how many candidate log modifications are acceptable.

4.2.1 Precision and Recall when Locating Candidate Code Snippets

This section evaluates the precision and recall of LogTracker when locating candidate code snippets. For ease of comparison, we calculate the value of F_{score}.

We randomly split historical log revisions into train and test data with a ratio of 5:5 and 8:2. Then LogTracker learns revision rules from the train data and locates candidate code snippets by applying these rules. We compare the located code snippets with log revisions in train and test data, thus counting how many log revisions in test data are successfully located (i.e., $Located\ code\ snippets\ in\ test\ data$ in the formulas). Based on this, we further calculate the value of precision, recall and F_{score} with following formulas. Above process is repeated five times to get the average value.

$$Precision = \frac{Located\ code\ snippets\ in\ test\ data}{Located\ code\ snippets} \tag{5}$$

$$Recall = \frac{Located\ code\ snippets\ in\ test\ data}{Log\ revisions\ in\ test\ data} \tag{6}$$

$$F_{score} = \frac{2 * Precision * Recall}{Precision\ +\ Recall} \tag{7}$$

Table 8 displays the precision and the recall of the eight subject projects[18] in response to two ratios of train data. Generally speaking, the precision is high (i.e., averagely 83.5% or 95.1%) in both cases and indicates that LogTracker is a reliable tool when guiding log revisions. Comparatively, the recall of LogTracker is lower. LogTracker locates 17.3% log revisions with half historical log revisions. In addition, as the ratio of train data increases,

[18] As shown in Tables 4 and 5, log revisions and rules of other four projects are so few that we do not show data of the four software in this experiment.

Table 8 Precision and recall of LogTracker when locating candidate code snippets

Software	Train:test = 5:5			Train:test = 8:2		
	Precision	Recall	F_{score}	Precision	Recall	F_{score}
Httpd	98.4%	15.0%	26.1%	90.4%	31.0%	46.2%
Git	97.2%	12.4%	22.0%	87.5%	29.3%	43.9%
Collectd	93.4%	24.8%	39.2%	69.7%	52.0%	59.6%
Postfix	100.0%	13.2%	23.3%	88.9%	33.1%	48.2%
OpenDDS	91.1%	17.1%	28.8%	85.1%	37.1%	51.7%
Ice	89.0%	25.1%	39.2%	73.7%	43.4%	54.7%
GIMP	95.1%	15.5%	26.7%	85.1%	30.3%	44.6%
Wireshark	96.7%	15.2%	26.3%	87.9%	31.8%	46.7%
Average	**95.1%**	**17.3%**	29.3%	**83.5%**	**36.0%**	50.3%

the recall rises from 17.3% to 36.0%. Thus, this result is acceptable as a first step towards guiding intricate log revisions.

There are two main reasons for the low recall. First, as we mentioned in Section 2.4, only 57.0% of historical log revisions share similar logging context. That is to say, for our methods, the theoretical value of recall is 57.0%. Second, in this experiment, LogTracker is trained with partial historical data, it is unavoidable to miss some rules and generate a lower recall. Besides, comparing with the previous work (Li et al. 2018), this paper has a lower recall and a higher precision mainly because it erases rules that are reverted in evolution history.

Considering 45.9% of rules are learned from revisions that spreads in various files, we further evaluate whether LogTracker can locate candidate code snippets accurately for rules involved with different files. As for implementation, we preprocess the input data of aforementioned experiment and only keep historical log revisions which are members of rules that are learned from different files. Then, with a ratio of 5:5, we split train and test data and calculate the precision and recall with the same method as aforementioned.

Figure 13 display the precision and recall when locating candidate code snippets for rules that are involved with different files. On average, the recall increases to 34.8% while the precision decreases to 81.9%. This indicates that LogTracker works well when locating candidates for rules that are learned from various files.

4.2.2 Quality of Candidate Log Modifications

Despite the ability of locating candidate code snippets, LogTracker also recommends log modifications which is presented in a manner of syntactical edit scripts (explained in Section 3.5.2). In this section, we evaluate the quality of these syntactical log modifications generated by LogTracker.

To do this, we randomly sample 800 (100 from each software) instances from the true positives (i.e., $located\ code\ snippet\ in\ test\ data$). Since these log revisions belong to test data which have already taken place in evolution history (see the definition of test data in Section 4.2.1), it is easy to judge the correctness of recommended log modifications by manually comparing them with the historical log revisions.

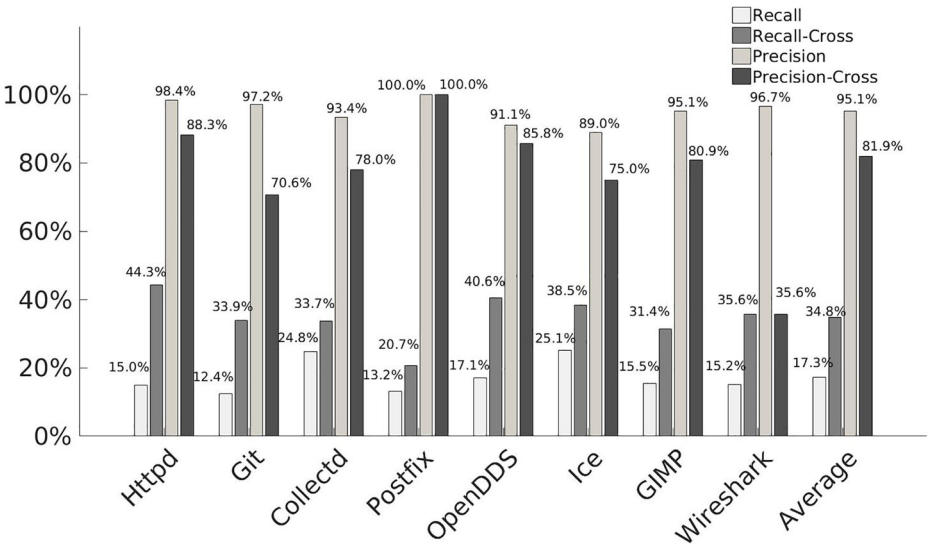


Fig. 13 Comparison of the precision and recall when LogTracker locates candidate code snippets for two types of rules (Recall-Cross: recall of LogTracker when locating candidate code snippets for rules learned from different files, Precision-Cross: precision when locating candidate code snippets for rules learned from different files.)

Specifically, according to the helpfulness of recommended log modifications, we classify them into three categories: $accept$, $change\_need$, $reject$. For ease of understanding, Fig. 14 shows examples of candidate log modifications that belong to the three categories. Similar to the one in Fig. 14a, candidate log modifications are accepted unless LogTracker has successfully recommend the demanded edit scripts which transform the candidate code snippet in the same way as the historical patch. Figure 14b displays a representative instance of candidate log modifications that need manual changes. In this case, LogTracker fails to recognize the differences of static content between the historical patch and candidate log statement and suggests the wrong static content (i.e., "*cannot open %s*" instead of "*can't write crash report %s*"). Fortunately, it is feasible for user to correct this mistake by checking historical patches. Candidate log modifications are rejected when LogTracker misunderstands the historical log revisions, and hence suggests unreasonable log revisions. In Fig. 14c, LogTracker recommends to update *msg_fatal* to *msg_warn* without figuring out the need of adding one more log statement.

As the results listed in Table 9, the average ratio for accepted edit scripts is 86.4%. Meanwhile, only 5.3% of recommended log modifications are totally wrong and unacceptable. This points out the usefulness of LogTracker on guiding logging revisions.

4.3 Precision of LCDM

As the key technology, the precision of LCDM seriously affects the precision of LogTracker. In order to evaluate the precision of LCDM, we measure how accurate LogTracker is at locating the context-similar log revisions, and take a comparison experiment between LCDM and DECKARD+.

Patch: Postfix-1.1.13, File: src/sendmail/sendmail.c, line: 613

```
 case SM_MODE_MAILQ:
 if (argv[OPTIND])
-  msg_fatal("display queue mode requires no recipient");
+  msg_fatal_status(EX_USAGE,
+    "display queue mode requires no recipient");
```

Candidate: Postfix-1.1.13, File: src/sendmail/sendmail.c, line: 843

```
case SM_MODE_USER:
if (argv[OPTIND])
  msg_fatal("stand-alone mode requires no recipient");
```

Recommended edit scripts that are accepted:

```
update name:msg_fatal to name:msg_fatal_status
add name:EX_USAGE in argument_list:("stand-alone mode requires no recipient")
```

(a) Example of candidate log modification that belongs to *accept* category

Patch: Git-2.8.6, file: builtin/fetch.c, line: 609

```
 fp = fopen(filename, "a");
 if (!fp)
-  return error(_("cannot open %s: %s\n"), filename, strerror(errno));
+  return error_errno(_("cannot open %s"), filename);
```

Candidate: Git-2.8.6, file: fast-import.c, line: 417

```
FILE *rpt = fopen(loc, "w"); ...
if (!rpt) {
  error("can't write crash report %s: %s", loc, strerror(errno));
```

Recommended edit scripts that need to change:

```
update name:error to name:error_errno
update literal:"can't write crash report %s: %s" to literal:"cannot open %s"
remove call:strerror(errno) from argument_list:
   ("can't write crash report %s: %s", loc, strerror(errno))
```

(b) Example of candidate log modification that belongs to *change − need* category

Patch: Postfix-2.4.16, file: src/smtpstone/smtp-source.c, line: 761

```
- if ((resp = response(session->stream, buffer))->code / 100 != 2)
-   msg_fatal("data %d %s", resp->code, resp->str);
+ if ((resp = response(session->stream, buffer))->code / 100 == 2) {
+   /* void */ ;
+ } else if (allow_reject) {
+   msg_warn("end of data rejected: %d %s", resp->code, resp->str);
+ } else {
+   msg_fatal("end of data rejected: %d %s", resp->code, resp->str);
+ }
```

Candidate: Postfix-2.4.16, file: src/smtpstone/smtp-source.c, line: 641

```
if ((resp = response(session->stream, buffer))->code / 100 != 2)
  msg_fatal("recipient rejected: %d %s", resp->code, resp->str);
```

Recommended edit scripts that are rejected:

```
update name:msg_fatal to name:msg_warn
update literal:"recipient rejected:%d %s" to literal:"end of data rejected:%d %s"
```

(c) Example of candidate log modification that belongs to *reject* category

Fig. 14 Examples of candidate log modifications that belong to *accept*, *change_need* and *reject* categories

Table 9 Quality of candidate log modifications recommended by LogTracker (A: ratio of accepted modifications, C: ratio of change_needed modifications, R: ratio of rejected modifications)

Software	A	C	R
Httpd	87.0%	11.0%	2.0%
Git	83.0%	10.0%	7.0%
Collectd	88.0%	6.0%	6.0%
Postfix	84.0%	8.0%	8.0%
OpenDDS	89.0%	6.0%	5.0%
Ice	84.0%	10.0%	6.0%
GIMP	86.0%	12.0%	2.0%
Wireshark	90.0%	4.0%	6.0%
Total	**86.4%**	**8.4%**	**5.3%**

4.3.1 Precision when Locating Context-Similar Log Revisions

We build an oracle test suit from 12 subject projects. One author of this paper manually selects groups of log revisions that have similar logging context from all historical log revisions.[19] It takes the author almost 800 hours to select these groups out. Then another expert who does not participate in the design and implementation of LogTracker validates the similarity of logging context among log revisions that belong to the same group. We finally identify 316 groups of similar log revisions, including 226 groups of similar log revisions that modify existing log statements and 90 groups that insert new log statements (see Table 10). Each group of similar log revisions corresponds to a test case. The input is one instance of this group, while the test oracles are the other instances.

For generating rules, we randomly select one instance from each group to train LogTracker.[20] We then apply these rules to historical versions of subject software, and collect candidates detected by LogTracker. By comparing candidates with test oracles, we calculate the precision[21] when locating context-similar log revisions as the ratio of candidates supported by the test oracles to all candidates.

Table 10 displays the precision for two sorts of rules in 12 subject projects. For rules that modify existing log statements, the precision is 93.4%. This is consistent with the high precision in Section 4.2, and indicates that LCDM is accurate when describing the semantics of logging context.

For rules that modify existing log statements, false positives are mainly caused by logging context described by related functions that are widely used. That is because related functions (e.g., *strcmp*) that are widely used cannot express the semantics of logging context effectively. The inaccurate comprehension of logging context further raises false alarms when locating context-similar log revisions.

As shown in Table 10, the precision for rules that insert new log statements is even lower. We manually check all the false positives and work out that there are two main cases.

[19] This process is done by searching historical log revisions that share the same keywords in contextual lines.

[20] In this case, each of the generated similar revision group consists of only one train instance. Considering the limited input, they are taken as effective rules.

[21] As mentioned in Section 4.1, developers may miss log revisions. Besides, the process of manually building oracle test suit may also miss some context-similar log revisions. As such, recall of this experiment is not reliable and we do not mention it here.

Table 10 Precision of LCDM (Modify: rules that modify existing log statements; Insert: rules that insert new log statements; G: groups of similar log revisions; P: precision)

Software	Modify		Insert	
	G	P	G	P
Httpd	38	97.4%	15	86.7%
Git	24	91.7%	14	92.9%
Mutt	7	100.0%	0	–
Rsync	7	100.0%	0	–
Collectd	20	90.0%	5	80.0%
Postfix	34	94.1%	11	72.7%
Tar	5	100.0%	6	100.0%
Wget	8	100.0%	12	75.0%
OpenDDS	21	90.5%	8	87.5%
Ice	18	88.9%	10	90.0%
GIMP	30	90.0%	13	76.9%
Wireshark	14	92.9%	9	88.9%
Total	**226**	**93.4%**	**90**	84.4%

First, the related functions whose return value should be checked and logged, are invoked in the return statement. In Fig. 15a, the related function, *apr_pollset_add()*, is called in the return statement. Hence, the necessity of checking and logging the return value is switched to the caller (i.e., *create_wakeup_pipe()*). Thus, this candidate is recognized as one false positive, if the caller is more possible to be checked and logged. The possibility of being logged for one function is measured using log rate. Its formula is as follow.

$$Log\ rate = \frac{Times\ of\ being\ logged}{Times\ of\ being\ invoked} \tag{8}$$

Second, the related function is invoked inside the loop structure. In Fig. 15b, the related function, *apr_pollset_add()*, is called inner the "*for*" structure. It is too time-consuming to check and log one non-fatal exception in every iteration. Hence, the candidate is recognized as one false positive if the exception is not fatal (e.g., The verbosity is information or warning). The severity of one exception depends on the verbosity of historically inserted log statements.

Httpd-2.2.34, file: worker.c

```
static apr_status_t create_wakeup_pipe(...){
 ...
 return apr_pollset_add(...);}
```

(a) False positive caused by return statement

Httpd-2.2.34, file: pollset.c

```
static void *listener_thread(...){
 ...
 for(lr=ap_listeners;lr!=NULL;lr=lr->next){ ...
  (void)apr_pollset_add(...);}
```

(b) Flase positive caused by loop

Fig. 15 False positives in application of rules that insert new log statements

4.3.2 Comparison Experiment

As the algorithm in DECKARD+ is widely used to depict the semantics of code context. This section compares LCDM with the algorithm in DECKARD+ to evaluate the precision of LCDM when describing logging context. To do this, we implement LogTracker-DECKARD by extracting the semantics of logging context with the algorithm in DECKARD+.

We calculate the precision of LogTracker-DECKARD using the same oracle test suit and method in Section 4.3.1, and compare this precision with that of LogTracker. Figure 16 indicates that the precision of LogTracker is higher by 12.6%. This result is consistent with the discussion in Section 3.2.1 and validates the statement that LCDM is more suitable to describe the semantics of logging context in comparison to traditional algorithms.

5 Threats to Validity

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

Quality of Log Revisions in Software Evolution Since LogTracker recommends proactive log revisions by applying rules that are learned from historical log revisions, its effectiveness is closely related to the quality of log revisions in software evolution. For example, false log revisions committed by developers will also be learned by LogTracker. Besides, as proved in Section 4.2, LogTracker can only provide guidances on log revisions which share similar logging context. That is to say, the ratio of context-similar log revisions limit the effectiveness of our tool. Despite the dependence on historical log revisions, LogTracker turns out to be useful by recommending 17.3% to 36.0% of historical log revisions when applied to the eight widely-used open source software.

Accuracy of Fuzzy Parsing Techniques Patches (incomplete code snippets) are the main inputs for learning log revision behaviors. Thus, traditional static analysis techniques, which are based on compilers, fail to effectively analyze patches. To solve this

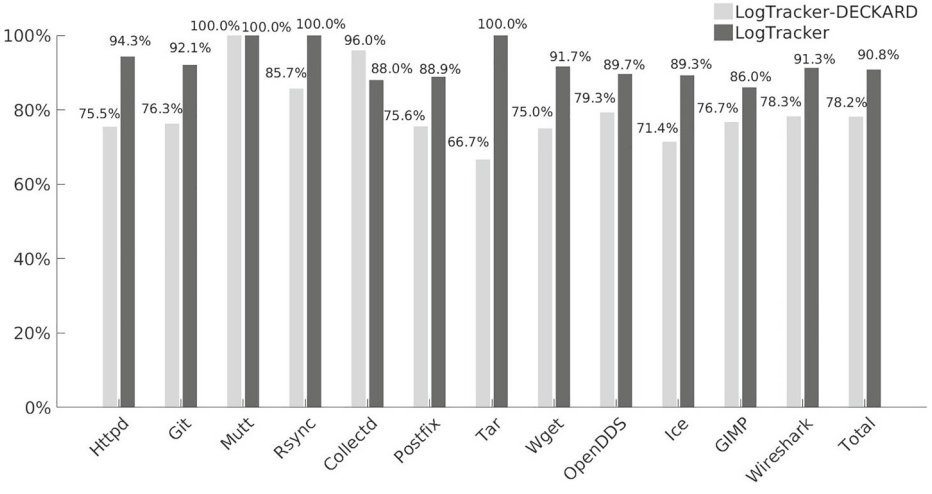


Fig. 16 Precision of LogTracker and LogTracker-DECKARD

problem, we employ fuzzy parsing techniques. Specifically, we use srcML to generate a syntactical structure for the incomplete code and GumTree to produce edit scripts for the two syntactical structures. Consequently, the effectiveness of LogTracker is closely related to that of fuzzy parsing techniques. Section 4.2.1 has evaluated the precision of LogTracker and indicates that the deviation is acceptable.

Accuracy of Manually Evaluation and Oracle Construction For lack of benchmark, the validation of recommended log revisions in Section 4.1.2 and the building of oracle data set in Section 4.3 both depend on manual analysis. In this case, the accuracy of manual analysis affects the accuracy of our experiments result. To release this problem, we have asked one expert developer who does not participate in the design and implementation of LogTracker to evaluate and correct our analysis results. This decreases the subjectivity of manual analysis in a way.

6 Future Work

This section discusses our future work on improving the performance and usability of LogTracker from three aspects.

How to improve the precision of LogTracker when predicting candidate log revisions? As a proactive log revision tool, the precision of LogTracker seriously affect its usability. As discussed in Section 4.1.2, candidate log revisions would be rejected for too trivial modifications on log statements. Thus, it is important to study how to automatically pre-check the trivialness of log revision rules. As far as we are concerned now, rename-inducing trivial log modifications can be identified by checking the semantics of renamed variables since renaming of variables does not affect their semantics.

How to improve the quality of candidate log modifications? Instructive candidate log modifications can guide developers on log revisions, while incorrect ones may mislead developers. Pointed out in Section 4.2.2, candidate log modifications need to be changed because it can not handle the modifications of static content well and are wrong if it can not recognize the correlation between log revisions. For the first problem, we think it can be released a bit by treating static content as groups of words and refining the modifications of static content as the addition or deletion of these groups. For example, the one in Fig. 14b should be modeled as "delete ':%s\n' from the end of static content". As for the second problem, we wonder it is due to the inaccurate separation of log revisions. Currently, LogTracker split log revisions in unit of log statements without considering the inner correlations. Thus, separation of log revisions with a larger unit (e.g., basic block) may partially release this problem.

How to use LogTracker in practice? At present, LogTracker is developed as one independent command line tool. To use it, developers should first run '–generate' to mine log revision rules from input versions of source code and then run '–apply' command to generate candidate log revisions by applying previous rules into input version of source code. Obviously, the working scenarios of LogTracker is independent from daily software development and should be further improved. We plan to integrate LogTracker with existing IDE (e.g., Eclipse or Visual Studio Code) as one code-intelligence plugin which aims to improve logging practices. In this case, once one stable version is released, LogTracker can incrementally learn from it, and once developers write one new log statement, LogTracker can check it against existing rules to provide real-time suggestions.

7 Related Work

There are three areas of research that are closely related to our work: empirical studies on logging practices, improving logging practices, and detecting and managing code clones.

Empirical Studies on Logging Practices Despite the importance of log statements, there are no rigorous specifications and systematic processes to guide practices of software logging (Fu et al. 2014; Yuan et al. 2012c; Pecchia et al. 2015). As a prerequisite, many researchers have devoted to summarizing the characteristics of existing logging practices. Yuan et al. (2012b) quantitatively studied the logging practices of four open-source subjects in C/C++ languages. They concluded ten impressive findings and built a verbosity checker to validate the effectiveness of their findings. Additionally, Chen and Jiang (2017a) performed a replication study on 21 Java-based open source software and concluded several unique characteristics of logging practices in Java-based systems. To characterize log placements in industry, Fu et al. (2014) conducted an empirical study on two industrial software and a questionnaire survey on 54 experienced developers. Hassani et al. (2018) carefully studied the characteristic of issues related to logging practices (i.e., log-related issues) and manually summarized seven root causes which imply the opportunity of detecting log-related issues automatically. In this paper, we also perform an empirical study on logging practices, but our focus is on the characteristics of context-similar log revisions. In this case, our work is supplementary of the above works.

Improving Logging Practice When it comes to improving logging practices, previous works mainly have addressed three main problems, as follows. 1) Where to log. This problem concerns where to place log statements. Errlog and LogAdvisor suggested whether to place log statements in one code by summarizing or learning log patterns. Log^2 and Log20 quantitatively represented the informativeness and overhead of logging practices. They recommended runtime log placements by seeking a balance between informativeness and overhead. 2) What to log. This problem concerns what variables should be output in one log statement. LogEnhancer detected uncertainty variables through back-slicing and constraint solving and appended them to log statements . 3) How to log. This problem is about improving quality of logging code. Chen and Jiang (2017b) summarized six anti-patterns from historical log revisions, and detected logging code that belongs to anti-patterns. We diverge from this work as we automatically mine rules from evolution history instead of manually summarizing anti-patterns. Li et al. (2017) identified whether new commitments require log modifications to reduce afterthought updates. They mined the correlation between log revisions and other revisions, while we aim to mine the correlation between logging context and modifications from log revisions. Hassani et al. (2018) designed and implemented four checkers to detect log-related issues automatically, including typo in static content, incorrect log level guard, missing log statements. Essentially speaking, this work depended on statistical information summarized from single version of source code, while LogTracker aims to guide log revisions with knowledge implicated in software evolution.

Detecting and Managing Code Clones In order to resolve code smell and improve code practices, researchers have proposed many clone detection and management techniques. Among clone detection tools, CCFinder (Kamiya et al. 2002) and CPMiner (Li et al. 2004) detected code clones with token vectors that are generated by lexical parser. DECKARD (Jiang et al. 2007), DECKARD+ and CloneDetective (Juergens et al. 2009) detected code clones with features that are generated by syntactical structures. Among

clone management tools, SysEdit (Meng et al. 2011) generated systematic edits by learning from historical modifications on clone code. It recommended how to modify code, but cannot locate where to be modified. LASE (Meng et al. 2013) automatically located and applied systematic edits by learning from at least two historical modifications on clone code. The algorithm of describing code context in LASE depended on syntactical structures, and could not accurately describe the semantics of logging context (see examples in Fig. 9a and b). Thus, it is difficult to predict systematic log revisions with LASE. REFAZER (Rolim et al. 2017) utilized program synthesis (Polozov and Gulwani 2015) to automatically locate and generate systematic edits by learning from historical modifications. Without consideration of code context, REFAZER is hard to recommend systematic log revisions. Although above researches on clone detection and management could not successfully handle context-similar logging code, they have motivated our design of LogTracker which makes a step toward solving this problem.

8 Conclusions

Despite the importance of log messages, it is difficult to reach good logging practices. This is mainly caused by two reasons. First, there are no well-established specifications on reaching good logging practices. Second, logging code evolves with bug fixes or feature updates during software evolution. Due to the bad logging practices, several log revisions may be missed by developers. Related works mainly targeted at the first reason and ignored the impacts of software evolution on logging code.

To fill this gap, we propose to learn log revision proactively from software evolution. We first conduct an empirical study on 12 open-source projects which figures out that logging code with similar logging context deserves similar modifications. This finding motivates the design and implementation of LogTracker. With an enhanced modeling of logging context, LogTracker is able to guide intricate log revisions that cannot be handled by existing tools. In addition, for better instruction on log revisions, we provide syntactical modifications for each candidate code snippet. The experiments turn out that LogTracker is able to predict 17.3% to 36.0% historical log revisions and correctly generates 86.4% of candidate log modifications. When applying LogTracker to the latest version, it successfully detects 369 instances of log revisions. Up to now, 79 log revisions were posted and 52 have already been accepted.

References

Ice (2018) Ice - comprehensive rpc framework. https://zeroc.com/products/ice

Arnold M, Ryder BG (2001) A framework for reducing the cost of instrumented code. ACM SIGPLAN Not 36(5):168–179. https://doi.org/10.1145/381694.378832. http://portal.acm.org/citation.cfm?doid=381694.378832

Chen BJ, Jiang ZM (2017) Characterizing logging practices in Java-based open source software projects - a replication study in apache software foundation. Empir Softw Eng 22(1):330–374. https://doi.org/10.1007/s10664-016-9429-5

Chen B, Jiang ZM (2017) Characterizing and detecting anti-patterns in the logging code. Proceedings - 2017 IEEE/ACM 39th international conference on software engineering, ICSE 2017, pp 71–81. https://doi.org/10.1109/ICSE.2017.15

Collard ML, Decker MJ, Maletic JI (2013) SrcML: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration. In: IEEE international conference on software maintenance, ICSM, IEEE, pp 516-519. https://doi.org/10.1109/ICSM.2013.85

Collectd (2017) Start page - collectd - The system statistics collection daemon. http://collectd.org/
Conservancy SF (2018) Git. https://git-scm.com/
Davison W (2018) rsync. https://rsync.samba.org/
Defays D (1977) An efficient algorithm for a complete link method. Comput J 20(4):364–366. https://doi.org/10.1093/comjnl/20.4.364. http://oup.prod.sis.lan/comjnl/article-pdf/20/4/364/1108735/200364.pdf
Ding R, Zhou H, Lou JG, Zhang H, Lin Q, Fu Q, Zhang D, Xie T (2015) Log 2: a cost-aware logging mechanism for performance diagnosis
Falleri JR, Morandat F, Blanc X, Martinez M, Montperrus M (2014) Fine-grained and accurate source code differencing. Proceedings of the 29th ACM/IEEE international conference on automated software engineering - ASE '14 pp 313–324. http://dl.acm.org/citation.cfm?doid=2642937.2642982
Foundation FS (2016) Diffutils - gnu project - free software foundation. https://www.gnu.org/software/diffutils/
Foundation FS (2017a) Tar - gnu project - free software foundation. https://www.gnu.org/software/tar/
Foundation FS (2017b) Wget - gnu project - free software foundation. https://www.gnu.org/software/wget/
Foundation PS (2018) Built-in functions-python 2.7.14 documentation. https://docs.python.org/2/library/functions.html
Foundation TAS (2017c) httpd - apache hypertext transfer protocol server - apache http server version 2.4. http://httpd.apache.org/docs/2.4/programs/httpd.html
Foundation W (2019) Wireshark - go deep. https://www.wireshark.org/
Fu Q, Zhu J, Hu W, Lou JG, Ding R, Lin Q, Zhang D, Xie T (2014) Where do developers log? an empirical study on logging practices in industry. Proceedings of the 36th international conference on software engineering - ICSE '14 pp 24–33. http://dl.acm.org/citation.cfm?doid=2591062.2591175
Gabel M, Jiang L, Su Z (2008) Scalable detection of semantic clones. Proceedings of the 30th international conference on Software engineering - ICSE '08 p 321. http://portal.acm.org/citation.cfm?doid=1368088.1368132
Github (2018a) Github - gumtreediff/gumtree: A neat code differencing tool. https://github.com/GumTreeDiff/gumtree
GitHub (2018b) skyhover/deckard: Code clone detection; clone-related bug detection; semantic clone analysis. https://github.com/skyhover/Deckard
Github (2019) niuxu18/logtracker: Automatic tool which tries to guide log revisions by mining software evolution
Hassani M, Shang W, Shihab E, Tsantalis N (2018) Studying and detecting log-related issues. Empir Softw Eng 11:1–33
Jiang L, Misherghi G, Su Z, Glondu S (2007) DECKARD: Scalable and accurate tree-based detection of code clones. In: Proceedings of the 29th International Conference on on Software Engineering - ICSE '07, pp 96–105. https://doi.org/10.1109/ICSE.2007.30
Juergens E, Deissenboeck F, Hummel B (2009) CloneDetective - A workbench for clone detection research. In: Proceedings of the 31th International Conference on Software Engineering - ICSE '09, pp 603–606. https://doi.org/10.1109/ICSE.2009.5070566
Kamiya T, Kusumoto S, Inoue K (2002) CCFinder: A multilinguistic token-based code clone detection system for large scale source code. IEEE Trans Softw Eng 28(7):654–670. https://doi.org/10.1109/TSE.2002.1019480
Kawrykow D, Robillard MP (2011) Non-essential changes in version histories. In: Proceedings of the 33th international conference on Software engineering - ICSE '11, pp 351–360. https://doi.org/10.1145/1985793.1985842
kevin8t8 (2018) The mutt e-mail client. http://www.mutt.org/
Kim M, Sazawal V, Notkin D (2005) An empirical study of code clone genealogies. ACM SIGSOFT Software Engineering Notes 30(5):187. https://doi.org/10.1145/1095430.1081737. http://portal.acm.org/citation.cfm?doid=1095430.1081737
Li H, Shang W, Zou Y, E Hassan A (2017) Towards just-in-time suggestions for log changes. Empir Softw Eng 22(4):1831–1865. https://doi.org/10.1007/s10664-016-9467-z
Li S, Niu X, Jia Z, Wang J, He H, Wang T (2018) Logtracker: Learning log revision behaviors proactively from software evolution history. In: Proceedings of IEEE/ACM international conference on program comprehension 2018 - ICPC, 2018
Li Z, Lu S, Myagmar S, Zhou Y (2004) CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - OSDI '04, pp 20. https://doi.org/10.1109/TSE.2006.28
Media S (2018) Sloccount download — sourceforge.net. https://sourceforge.net/projects/sloccount/

Meng N, Kim M, McKinley KS (2011) Systematic editing. Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation - PLDI '11 p 329. http://portal.acm.org/citation.cfm?doid=1993498.1993537

Meng N, Kim M, McKinley KS (2013) LASE : Locating and Applying Systematic Edits by Learning from Examples

Mondai M, Roy CK, Schneider KA (2018) Micro-clones in evolving software. Proceedings of 25th IEEE international conference on software analysis, evolution and reengineering - SANER'18, pp 50–60. https://doi.org/10.1145/381694.378832

OCI (2018) Opendds. http://opendds.org/

Pecchia A, Cinque M, Carrozza G, Cotroneo D (2015) Industry practices and event logging: Assessment of a critical software development process. In: Proceedings of the 37th IEEE international conference on software engineering - ICSE '15, pp 169–178. https://doi.org/10.1109/ICSE.2015.145

Polozov O, Gulwani S (2015) FlashMeta: A framework for inductive program synthesis. ACM SIGPLAN Not 50(10):107–126. https://doi.org/10.1145/2858965.2814310. http://dl.acm.org/citation.cfm?doid=2858965.2814310

Rolim R, Soares G, D'Antoni L, Polozov O, Gulwani S, Gheyi R, Suzuki R, Hartmann B (2017) Learning syntactic program transformations from examples. In: Proceedings of the 39th international conference on software engineering - ICSE '17, pp 404–415. https://doi.org/10.1109/ICSE.2017.44

Sigelman BH, Andr L, Burrows M, Stephenson P, Plakal M, Beaver D, Jaspan S, Shanbhag C (2010) Dapper, A large-scale distributed systems tracing infrastructure. Tech rep., California, USA. https://ai.google/research/pubs/pub36356

Systems CR (2019) Sample size calculator. https://www.surveysystem.com/sscalc.htm

Team TG (2019) Gimp - gnu image manipulation program. https://www.gimp.org/

Venema W (2013) The postfix home page. http://www.postfix.org/

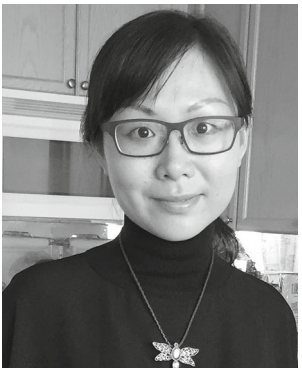
Yuan D, Park S, Huang P, Liu Y, Lee M (2012a) Be conservative: enhancing failure diagnosis with proactive logging. In: Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation - OSDI '12, 41(6):293–306

Yuan D, Park S, Zhou Y (2012b) Characterizing logging practices in open-source software. In: Proceedings of the 34th international conference on software engineering - ICSE '12, pp 102–112. https://doi.org/10.1109/ICSE.2012.6227202

Yuan D, Zheng J, Park S, Zhou Y, Savage S (2012c) Improving software diagnosability via log enhancement. ACM Trans Comput Syst 30(1):1–28. https://doi.org/10.1145/2110356.2110360. http://dl.acm.org/citation.cfm?doid=2110356.2110360

Zhao X, Rodrigues K, Stumm M (2017) Log20: Fully automated optimal placement of log printing statements under specified overhead threshold. In: Proceedings of the 26th symposium on operating systems principles - SOSP '17, pp 565–581. https://doi.org/10.1145/3132747.3132778

Zhu J, He P, Fu Q, Zhang H, Lyu MR, Zhang D (2015) Learning to log: Helping developers make informed logging decisions. In: Proceedings of the 37th international conference on software engineering - ICSE '15, pp 415–425. https://doi.org/10.1109/ICSE.2015.60



Shanshan Li is an associate professor in the Department of Computer Science at National University of Defense Technology. Her main research interests include empirical software engineering, with a particular interest in software quality enhancement, defect prediction, and misconfiguration diagnosis. She has published more than 50 papers and won several awards including the Distinguished Paper Award in Saner 2018, Spotlight paper in TPDS, et al. She is a member of the IEEE and ACM.

Xu Niu received the BS degree in the Department of Software Engineering from Chongqing University, Chongqing, China, in 2016. She is currently a master student in the School of Computer from National University of Defense Technology, Changsha, China. Her main research interests includes software reliability and software quality enhancement.

Zhouyang Jia received the BS and MS degrees from the School of Computer Science, National University of Defense Technology, Changsha, China, in 2013 and 2015, respectively. He is currently working toward the PhD degree at National University of Defense Technology. His main research interests include software engineering, software reliability, operating system and so on. He is a student member of the ACM.

Xiangke Liao is a fellow of the Chinese Academy of Engineering and the chief designer of Tianhe Supercomputer and Kylin Operating System. He is currently a full professor and the dean of School of Computer, National University of Defense Technology. He has published more than 100 papers in top conferences and journals including International Conference on Computer Communications (INFOCOM), Transactions on Parallel and Distributed System(TPDS) and International Conference on Program Comprehension(ICPC), et. al. His research interests include parallel and distributed computing, high-performance computer systems, operating systems, cloud computing and networked embedded systems. He is a member of the IEEE and ACM.

Ji Wang is a professor in College of Computer at National University of Defense Technology, China. His research interests include programming methodology, program analysis and verification, formal methods and software engineering. He received his Ph.D. degree in Computer Science from National University of Defense Technology in 1995. Wang is serving as the vice director of Academic Board of State Key Laboratory of High Performance Computing, China. He is a member of the IEEE and the ACM.

Tao Li received the BS and PhD in the Department of Computer Science and Technology from Nankai University, Tianjin, China, in 1999 and 2007 respectively. He is currently a full professor at Nankai University. His research interests include parallel and distributed computing, machine learning, and network science. He is a member of the IEEE CS and ACM, and a senior member of the CCF.

Affiliations

Shanshan Li[1] · Xu Niu[1] · Zhouyang Jia[1] · Xiangke Liao[1] · Ji Wang[1] · Tao Li[2]

Xu Niu
niuxu16@nudt.edu.cn

Zhouyang Jia
jiazhouyang@nudt.edu.cn

Xiangke Liao
xkliao@nudt.edu.cn

Ji Wang
wj@nudt.edu.cn

Tao Li
litao@nankai.edu.cn

[1] The School of Computer, National University of Defense Technology, Changsha, China

[2] The School of Computer Science and Technology, Nankai University, Tianjin, China