LogTracker: Learning Log Revision Behaviors Proactively from Software Evolution History

Shanshan Li
National University of Defense Technology
Changsha, China
shanshanli@nudt.edu.cn

Xu Niu
National University of Defense Technology
Changsha, China
niuxu16@nudt.edu.cn

Zhouyang Jia
National University of Defense Technology
Changsha, China
jiazhouyang@nudt.edu.cn

Ji Wang
National University of Defense Technology
changsha, China
wj@nudt.edu.cn

Haochen He
National University of Defense Technology
Changsha, China
hehaochen13@nudt.edu.cn

Teng Wang
National University of Defense Technology
Changsha, China
wangteng13@nudt.edu.cn

ABSTRACT

Log statements are widely used for postmortem debugging. Despite the importance of log messages, it is difficult for developers to establish good logging practices. There are two main reasons for this. First, there are no rigorous specifications or systematic processes to guide the practices of software logging. Second, logging code co-evolves with bug fixes or feature updates. While previous works on log enhancement have successfully focused on the first problem, they are hard to solve the latter. For taking the first step towards solving the second problem, this paper is inspired by code clones and assumes that logging code with similar context is pervasive in software and deserves similar modifications. To verify our assumptions, we conduct an empirical study on eight open-source projects. Based on the observation, we design and implement LogTracker, an automatic tool that can predict log revisions by mining the correlation between logging context and modifications. With an enhanced modeling of logging context, LogTracker is able to guide more intricate log revisions that cannot be covered by existing tools. We evaluate the effectiveness of LogTracker by applying it to the latest version of subject projects. The results of our experiments show that LogTracker can detect 199 instances of log revisions. So far, we have reported 25 of them, and 6 have been accepted.

CCS CONCEPTS

• **Software and its engineering → Software reliability**; **Software evolution**; *Maintaining software*;

KEYWORDS

Log revision, software evolution, failure diagnose

ICPC '18, May 27–28, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5714-2/18/05…$15.00
https://doi.org/10.1145/3196321.3196328

ACM Reference Format:
Shanshan Li, Xu Niu, Zhouyang Jia, Ji Wang, Haochen He, and Teng Wang. 2018. LogTracker: Learning Log Revision Behaviors Proactively from Software Evolution History. In *ICPC '18: ICPC '18: 26th IEEE/ACM International Confernece on Program Comprehension , May 27–28, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3196321.3196328

1 INTRODUCTION

Log statements are inserted by developers to record the runtime status of software. Since log messages are both informative and convenient to collect, they have commonly been adopted to aid in postmortem failure diagnosis. Despite the importance of logging code, it is challenging for 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 [15, 29, 36]. Hence, the means by which developers make log placement decisions is both subjective and arbitrary. Second, logging code co-evolves with bug fixes or feature updates. This problem is illustrated in Figure 1. Figure 1a displays a patch that inserted preliminary validation of sensitive data. In addition to the validation code, the developers also inserted new log statements to make exception handling easier. In Figure 1b, the developers committed a patch to make the software work well on 64-bit platform. This patch also modified log variables of the original log statement. In both cases, log revisions are committed in response to bug fixes or feature updates.

There are already many works that focused on improving logging practices. Errlog [35] and LogAdvisor [39] help to insert missing log statements for given code snippets. LogEnhancer [37] appended informative variables to log messages in order to resolve the ambiguity in failure diagnosis. *Log*^{2} [8] and Log20 [38] decided what log messages to output by seeking a balance between informativeness and overhead. Although the abovementioned works are partly able to handle the first problem, it is difficult for them to predict log revisions that are related to bug fixes or feature updates (see Figure 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

Bug 2674 in Squid:

Crash occur if the safety limits on individual request/reply header length are changed upwards. MUST remain at or under 64KB.

Revision 9070 in Squid-3.0, client_side.cc:

```
+ /*NP: don't be tempted to move this down or remove again.
+  *It's the only DDoS protection old-String has against long URL*/
+ if (hp->bufsiz <= 0) {
+  debugs(33,5,"Incomplete request,waiting for end of request line");
+  return NULL; }
+ else if ((size_t)hp->bufsiz >= Config.maxRequestHeaderSize
+ && headersEnd(hp->buf, Config.maxRequestHeaderSize) == 0){
+  debugs(33, 5, "parseHttpRequest: Too large request");
+  return parseHttpRequestAbort(conn, "error:request-too-large");}
```

(a) Insert new log statement along with bug fixes

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);
```

(b) Update log statement along with feature updates

Figure 1: Log revision examples in software evolution

versions[1] of software. In Figure 1b, the original log statement was thought to be correct in the initial version. However, it required modifying in the new version, where new feature was introduced. Second, log statements cannot always provide sufficient clues when diagnosing unpredictable bugs. This point is supported by massive after-thought updates of log statements [2, 36, 37]. Figure 2a 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 Figure 2b, 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 [1, 8, 33]. 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 that share semantically similar context 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 second reference argument of apr_dir_open() and the

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

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)<<"}\nNULL\n{"<<
+   << getStringPrefix(nulpos+1, header_end)<<"}");
```

(a) Enhance existing log statement

Bug 2787 in Squid:

The message saying why is confusing and 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);
```

(b) Suppress misleading log statement

Figure 2: Examples of improving logging practices

Httpd-2.4.7 vs Httpd-2.4.9, file:htcacheclean.c, function:remove_directory, line:1090

```
rv=apr_dir_open(&dirp, dir, pool);
if(APR_STATUS_IS_ENOENT(rv)){ return rv;}
if(rv!=APR_SUCCESS){
- char errmsg[120];
  apr_file_printf(errfile,"Could not
-  open directory %s: %s" APR_EOL_STR, dir,
-  apr_strerror(rv,errmsg, sizeof errmsg));
+  open directory %s: %pm" APR_EOL_STR, dir,&rv);
```

(a) Modify log statement similarly to Figure 3b

Httpd-2.4.7 vs Httpd-2.4.9, file:htcacheclean.c, function:find_directory, line:1154

```
rv=apr_dir_open(&dirp, base, pool);
if(rv!=APR_SUCCESS){
- char errmsg[120];
  apr_file_printf(errfile, "Could not
-  open directory %s: %s" APR_EOL_STR,base,
-  apr_strerror(rv, errmsg, sizeof errmsg));
+  open directory %s: %pm" APR_EOL_STR, base, &rv);
```

(b) Modify log statement similarly to Figure 3a

Figure 3: Example of similar modifications on log statements with similar logging context

return value of apr_strerror() when apr_dir_open() did not return APR_SUCCESS. And the log variable which expressed the return value of apr_strerror() was updated to the pointer of return value of apr_dir_open(). Based on this observation, we design and implement LogTracker 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 5) 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 80.3% of context-similar log revisions belong to similar modifications. This observation has validated the guiding significance of historical log revisions.
- A proactive log revision tool. Due to an enhanced modeling of logging context, LogTracker can predict intricate log revisions with the log revision rules [2] mined from software evolution.
- Validation of the effectiveness of LogTracker. By applying generated rules to the latest version of subject projects, LogTracker identifies 199 instances of log revisions. So far, we have reported 25 of them, and 6 have been accepted.

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, precision and recall of LogTracker. Section 6 presents the related work. Lastly, we conclude our work in section 7.

2 MOTIVATION

In this section, we conduct an empirical study on eight open-source projects. In order to verify our hypothesis, we answer the following three research questions (RQ).

RQ1. How pervasive are log revisions? It is acknowledged that software projects evolve with bug fixes and feature updates [28, 31]. Due to the high log density, it is unavoidable that logging code co-evolves with bug fixes or feature updates. This research question targets at quantitatively evaluating the proportion of log revisions in software evolution.

RQ2. What are the characteristics of log revisions? Log revisions can be classified into multiple sub-categories according to edit types (e.g., insert, delete, update) and edited components (e.g., function name, variable, static content). We propose this research question to characterize the distribution of different revision scenarios.

RQ3. How many context-similar log revisions are modified similarly? Previous study [23] 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.

2.1 Experimental Setup

This empirical study is conducted on eight open-source projects in C/C++ languages. They are Httpd [14], Git [6], Mutt [22], Rsync [7], Collectd [5], Postfix [34], Tar [11], Wget [12]. 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 code (LOC) is measured using SLOCCount [26] in order to eliminate comments and empty lines. The line of logging code (LLOC) is the total number of lines occupied by log statements[3]. The final metric evaluates the ratio of LOC to LLOC and is inversely proportioned to log density. On average, one line of logging code appears for every 34 lines of code. This result is consistent with the findings of previous study [2, 36] and indicates a high log density. In addition, the diversity of these indicators (especially log density) increases the universality of our research.

Table 1: Subject Software

Software	Description	LOC	LLOC	LOC/ 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
Total		1,136,455	33,454	**34**

In order to collect as many log revisions as possible, we crawl all available versions of subject projects and generate patches automatically by running Diffutils [10] on neighboring versions. For each patch, its containing hunks [4] 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 [35, 39]. All selected hunks are then passed to GumTree [9, 17] which generates the syntactical edit scripts. We use these edit scripts to identify hunks that modified log statements (i.e., log hunks) rather than empty lines or comments. The above processes produce the final syntactical revisions of the log statements, which serve as the input for this data research.

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

2.2 RQ1: How Pervasive are Log Revisions?

In order to capture the pervasiveness of log revisions in software evolution, we evaluate one indicator that have been commonly used by previous studies [2, 36].

We measure the relative churn rate of the logging code in comparison to the entire code. The formula for this is as follows.

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

As shown in Figure 4, the average churn rate of the logging code is 1.8 times over the churn rate of the entire code. This data is consistent with previous studies [2, 36] and indicates that logging code is modified at least as frequently as the entire code. This

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

[3] Logging statements are recognized by regex patterns which are explained in next paragraph

[4] 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.

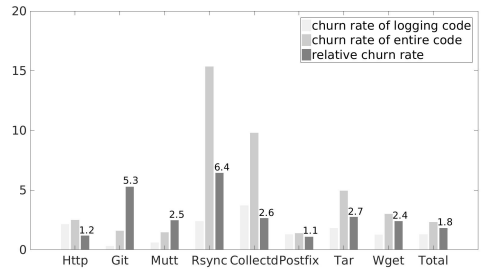

Figure 4: Churn rate of logging code and entire code

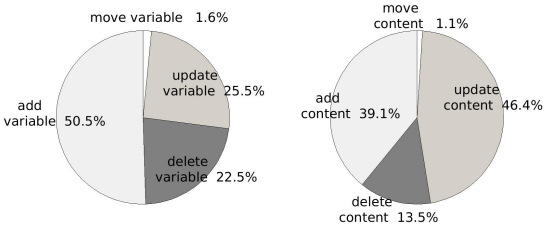

Figure 5: Distribution of modifications of log variables (left) and modifications of static content (right)

finding emphasizes the demand for improving logging practices and reveals the potential offered by learning log revision behaviors from historical modifications.

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), variables (e.g., variable_a) and static content (e.g., "value of variable a is %d"). In a similar fashion to previous works [2, 36], 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. This 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 2.

Generally speaking, the insertion, deletion, and update of log statements happen relatively frequently during software evolution. Among the five categories, the deletion of log statements occurs most infrequently, while the modifications of variables and static content happen most frequently. As such, we refine the modification of variables and static content into eight sub-categories, whose distribution characteristics are shown in Figure 5. Almost half of the modifications made to variables and static content are deletions and updates, which have not been covered by previous works [35, 37, 39]. This proves the necessity of mining rules from software evolution.

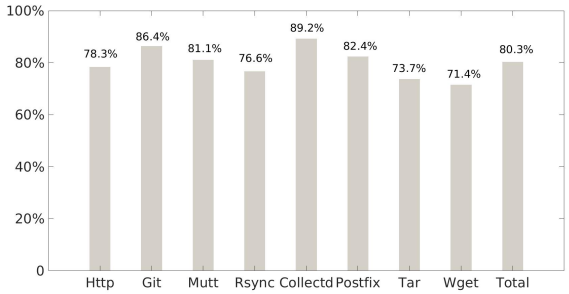

Figure 6: Pervasiveness of similar modifications among context-similar log revisions

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 automatically classify log revisions that share semantically similar context to produce *context-similar log revision groups*. Following this, we compute the number of context-similar log revisions by summarizing all instances in *context-similar log revision groups*. Table 3 displays the detailed data. On average, 51.0% 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 to log statements. This result indicates the potential effectiveness of rules that are mined from software evolution history.

Additionally, 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 belong to similar modifications, thus generating *modifications-similar log revision groups*. From this, it is obvious that the *modifications-similar log revision groups* are a subset of *context-similar log 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 Figure 6, on average, 80.3% of the *context-similar log 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 historical log revisions to log statements that share semantically similar context with historical revision behaviors.

3 DESIGN AND IMPLEMENTATION

3.1 Overview

In order to guide log revisions, this paper designs and implements LogTracker, which can mine the correlation between logging context and log modifications. In this section, we detail the implementation of LogTracker.

As shown in Figure 7, LogTracker consists of two main phases. The first phase involves mining rules from software evolution. When given one log revision, LogTracker should first analyze the semantics of the logging context and retrieves log modifications. These then serve as input when generating rules. In the second phase, LogTracker suggests log modifications for code by applying

Table 2: Distribution of log insertion, log deletion, update log function, variables and static content

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%)
Total	12,244	3,072 (25.1%)	1,214 (9.9%)	2,201 (18.0%)	8,844 (72.2%)	6,079 (49.7%)

Table 3: The ratio of context-similar log revisions (T: Total log revisions, C: Context-similar log revisions)

Software	T	C	Ratio	Software	T	C	Ratio
Httpd	5,347	2,733	51.1%	Collectd	635	424	66.8%
Git	2,002	868	43.4%	Postfix	2,222	1,200	54.0%
Mutt	322	103	32.0%	Tar	429	184	42.9%
Rsync	557	393	70.6%	Wget	730	338	46.3%
Total	12,244	6,243	**51.0**%				

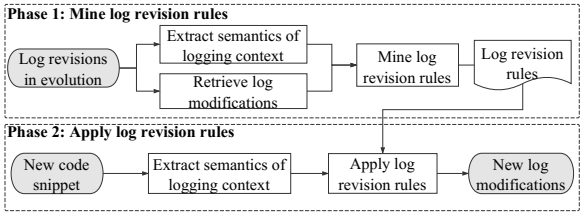

Figure 7: Architecture of LogTracker

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 accuracy of this module seriously affects the accuracy of the whole tool. In section 3.2, we illustrate the design and implementation details of this module.

Retrieving log modifications. This module generates both syntactical and textual edit scripts in order to represent log modifications. Section 3.3 explains how we generate these edit scripts.

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 by detecting semantically similar logging context. More details are provided in section 3.5.

3.2 Extracting the Semantics of Logging Context

As mentioned above, the understanding of the semantics of logging context seriously affects the accuracy of any recommendations. This subsection illustrates how LogTracker extracts 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 Figure 8a and 8b, the two log statements both print messages when the return value of select() is negative and errno is not EINTR. That is to say, the semantics of these two logging context are similar, but differ in syntactical structures. In this case, traditional algorithms [16, 19, 20] 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 [16, 19, 20] may raise false alarms. While the two log statements in Figure 8c and 8d share similar syntactical structures, their semantics vary a great deal. Specifically, the log statement in Figure 8c prints log messages when the return value of gfi_unpack_entry() is false while the log statement in Figure 8d prints messages when the return value of loopup_object_buffer() is false.

In summary, traditional algorithms 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 main components: branch statements that indicate under what conditions the log message should be printed, and log statements that describe what variables to output. In this case, the task of understanding the semantics of logging context can be split into a comprehension of the branch statements and the log statements. Consequently, we define the Logging Context Description Model as a 2-tuple $m=<c,v>$, where c models the check conditions and v models the log variables.

We notice that both branch statements and log statements consist of variables. These variables usually relate to functions according to data dependence. There are three types of data dependence in total. We call them type 1, type 2 and type 3 for simiplicity, and assign corresponding postfixes. Type 1 dependence expresses write dependence through the return value of functions (e.g., *rv=fun()*) and its postfix is "ret". Type 2 indicates write dependence on function by

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 8b

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 8a

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 8d

Spftware: 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 8c

Figure 8: Real-world log statements used to explain challenges in extracting semantics of logging context

reference parameter (e.g., *fun(&a)*) and its postfix is "arg_index_ret" [5]. Type 3 represents read dependence on function by parameter (e.g., *fun(a)*) with the postfix as "arg_index". For example, the variable buf in Figure 8c has a type 1 dependence on gfi_unpack_entry().

Since the semantics of these related functions (i.e., f) indicates the semantics of variables contained by branch statements or log statements, LCDM utilizes these related functions to express the semantics of the check conditions and the log variables. Each variable of c and v is represented as the concatenation of related function name and a postfix that indicates the dependence type.[6]

In order to aid understanding, we illustrate LCDM with the initial code snippet of the patch in Figure 3b as an example. Its initial code snippet is:

```
rv = apr_dir_open(&dirp, base, pool);
if(rv!=APR_SUCCESS){ ...
 apr_file_printf(errfile,"Could not open directory %s: %s"
   APR_EOL_STR, base, apr_strerror(rv, errmsg, sizeof errmsg));
```

The LCDM of this logging context equals to *<[apr_dir_open_ret], [apr_strerror_ret, null, apr_dir_open_arg_2_ret]>*. It indicates that this log statement will output a return value of apr_strerror() and the second reference argument of apr_dir_open() if the return value of apr_dir_open() is not APR_SUCCESS.

3.2.3 Extracting LCDM. The primary task when extracting LCDM is to identify *c* and *v*. We extract the related functions of branch statements and log statements to express the semantics of the check condition (i.e., *c*) and log variables (i.e., *v*). Algorithm 1 explains how we extract related functions for one statement (i.e., branch statement or log statement).

[5] "Index" means the index of this parameter in function. For example, in fun(a, b), the index of a is 1 and the index of b is 2.

[6] The variable is represented as null if there is no related functions.

Algorithm 1 Extracting related functions for one statement

Require: AST of this statement, PDG of function body
Ensure: list of related functions F={$f | f is related function$}

```
1:  F = []
2:  for each node n in statement s do
3:      f = NULL
4:      if n invokes a function f' then
5:          f=concat(name of f', ret); goto line14;
6:      from s traverse upward PDG
7:      if n has type 1 dependence on function f' then
8:          f=concat(name of f', ret); goto line 14
9:      if n has type 2 dependence on function f' then
10:         f=concat(name of f', arg_index_ret); goto line 14
11:     from s traverse downward PDG
12:     if n has type 3 dependence on function f' then
13:         f = concat(name of f', arg_index); goto line 14
14:     F.add(f)
15: return
```

We break down the extraction of related functions for one statement into the extraction of related functions for its descendant nodes. Given the syntactical structure of one statement (AST[7] in this paper), this algorithm processes each node in breadth-first order. For nodes that indicate an invocation of functions, invoked functions are identified as type 1 related to the statement. For other nodes, the program dependence graph (PDG) of the function body is traversed upward from the input statement to search for type 1 or type 2 dependence. If it succeeds, program exits with related functions; otherwise, it traverses downward the PDG to search for type 3 dependence. If type 3 dependence is not found, the related function of this node is marked as null.

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

```
rv = apr_dir_open(&dirp, base, pool);
if(rv != APR_SUCCESS)
```

This branch statement only contains one variable node. It has a type 1 dependence on function apr_dir_open(). Thus, the value of *c* is *[apr_dir_open_ret]*. The log statement is:

```
apr_file_printf(errfile, "Could not open directory %s: %s"
  APR_EOL_STR, base, apr_strerror(rv, errmsg, sizeof errmsg));
```

This log statement consists of two variable nodes and one node that invokes function. The node that invokes function indicates a type 1 dependence between the log statement and apr_strerror(). The variable node with text base corresponds to type 2 dependence on the second reference parameter of apr_dir_open(). The related function of the variable node with text errfile is regarded as null since we cannot find any related functions. Thus, the value of *v* is *[apr_strerror_ret, null, apr_dir_open_arg_2_ret]*.

The extraction of LCDM demands two inputs: the syntactical structure of the statement and PDG of the function body. The first of these is generated by srcML [4] which translates incomplete code into a syntactical unit. In order to retrieve PDG of the function body, we should first trace back to the function body of modified log

[7] Abstract Syntax Tree

Httpd-2.0.64 vs Httpd-2.0.65, file: modules/arch/netware/mod_nw_ssl.c, line: 668

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

Figure 9: Patch that inserted empty line in log statement

statement. To do this, we identify the location of the log statement in the source file by analyzing the range information in the hunk. Then, we extract the function body by traversing the ancestor nodes of log statement. We then process the syntactical structure of the function body to build a partial PDG, which illustrates function-related dependencies.

3.3 Retrieving Log Modifications

Retrieving log modifications is another prerequisite when learning log revision behaviors. This paper represents log modifications using syntactical and textual edit scripts. The first of these is used to select log revisions that syntactically modify log statements, while the second represents log modifications using differential token lists to make digitalization easier.

3.3.1 Syntactical Edit Scripts. Syntactical edit scripts express the differences between the syntactical structures of two code snippets. They are used to eliminate the interference from comments or empty lines. For example, the patch in Figure 9 inserted an empty line among the static content and log variables, but did not modify the syntactics of the log statement.

In this paper, we generate these differences using GumTree that implement a state of art tree differentiating algorithm. The syntactical edit scripts for the log statement in Figure 3b are shown as follows.

Delete a node that invokes apr_strerror().
Insert a variable node whose text is &rv.
Update text of one literal node from "...%s" to "...%pm".

3.3.2 Textual Edit Scripts. To aid in numeralization, this paper approximates syntactical edit scripts using textual edit scripts. During implementation, old and new log statements are first split into lists of string tokens through a widely used method [21, 25]. Then, common tokens are eliminated to generate differential lists from two token lists. Here, we name these differential lists as textual edit scripts. It is then easy to digitalize these tokens with hash algorithms [13].

The textual edit scripts for the patch shown in Figure 3b are *[s, apr_strerror, errmsg, sizeof, errmsg]* and *[pm]*. For this example, the textual edit scripts approximate syntactical edit scripts well. We have manually verified the accuracy of this approximation through sampling. The result indicates that textual edit scripts work well in most of the time, but are inaccurate when the modifications contain the movement of syntactical nodes. Fortunately, movement of log variables or static content seldom happens according to the distribution characteristics shown in Figure 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, v composes LCDM, which describes the semantics of logging context, and e is edit scripts (including syntactical edit scripts and textual edit scripts). One rule indicates that, when given a log statement, if its context is similar to $<c,v>$, it deserves modifications represented by e.

In order to mine rules, we run a cluster algorithm [32] 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, we select groups that have at least two voters as effective rules (In the following sections, we take effective rules as "rules" for simplicity).

3.5 Applying Log Revision Rules

As mentioned in section 2.3, log revisions during software evolution fall under a number of categories. As such, generated rules also involve various types of log modifications, such as log insertion, log deletion, and update of log function. Specifically, rules that modify existing log statements (i.e., "M" in Tables) and rules that insert new log statements (i.e., "I" in Tables) vary in application procedures.

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). When given code, to apply this sort of rules, we first recognize the inner log statements, and extract their logging context. Then, when deciding 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, we further manually validate the feasibility of these modifications on the log statement before recommending it to the developers.

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 applying these rules to code, we judge whether all related functions in the rule context are contained in the *call set*, which consists of all functions invoked in code. If true, we further validate the necessity of inserting new log statements before informing the developers.

4 EVALUATION

This section evaluates the performance of LogTracker from three aspects. Section 4.1 measures its effectiveness on suggesting missed log revisions. Section4.2 evaluates its precision and recall when predicting log revisions. In section 4.3, we measure the accuracy of LCDM by locating context-similar log revisions with LogTracker. In addition, we compare LCDM with DECKARD+ [16, 18].

4.1 Effectiveness of LogTracker

As mentioned by previous work [3, 28, 31], 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.

First, we train LogTracker with historical log revisions to generate rules (i.e., 871 rules shown in Table 4 and 5) Then, those rules are applied to the latest version of the eight subject projects to

Table 4: Rules learned by LogTracker

Software	M	I	Total	Software	M	I	Total
Httpd	234	66	300	Collectd	78	23	101
Git	20	123	143	Postfix	128	84	212
Mutt	17	2	19	Tar	17	5	22
Rsync	35	1	36	Wget	25	13	38
Total	214	657	**871**				

Table 5: Distribution of rules among five categories

Category	Rule	Ratio	Category	Rule	Ratio
Log insertion	250	28.7%	Log deletion	111	12.7%
Update of log function	176	20.2%	Modification of variables	471	54.1%
Modification of static content	258	29.6%			

Table 6: Recommendations proposed by LogTracker

Software	M	I	Total	Software	M	I	Total
Httpd	25	74	99	Collectd	8	22	30
Git	32	9	41	Postfix	14	15	29
Total	79	120	**199**				

detect missed log revisions. At last, by manually validating the correctness of recommended results, we summarize 199 true positives (see Table 6) from four software.

We are in the process of reporting those recommendations to developers for feedback. Up to now, we have reported 25 instances. 6 (24.0%) instances have been accepted by developers, 5 (20.0%) instances are under discussion, 1 (4.0%) instance has been rejected for too many dependent modifications, 13 (52.0%) instances (10 instances are detected by the same rule) has been rejected due to the out-date rules.

Here, we illustrate an 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.

There are two main reasons for the rejected instances. 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, rules learned from reverted log revisions are out of date. In a similar fashion to other commitments, log revisions may be reverted during software evolution. As such, the reverted log revisions are not suitable to latest software. For example, two revisions in Postfix-2.3.18 updated log function from msg_panic() to msg_warn() when return value of dict_handle() is 0. These revision was reverted in Postfix-2.4.0. LogTracker learns one rule from the former revisions and detects two missed revisions in the latest version of Postfix. Although the learning and applying phases work well, these revisions are rejected for the out-date rule [8](*"It would be a mistake to replace panic calls with warning calls"*).

Table 7: Precision and recall of LogTracker

Software	train:test = 5:5			train:test = 8:2		
	P	R	F	P	R	F
Httpd	96.1%	17.4%	29.5%	88.2%	33.6%	48.7%
Git	92.2%	19.1%	31.7%	84.9%	37.2%	51.8%
Collectd	95.0%	25.8%	40.6%	64.4%	55.3%	59.5%
Postfix	92.7%	25.1%	39.6%	83.5%	47.2%	60.3%
Total	93.3%	19.7%	32.5%	84.9%	38.1%	52.6%

4.2 Precision and Recall of LogTracker

The aim of LogTracker is to guide intricate log revisions by learning from software evolution. This section evaluates the precision (P) and recall (R) of LogTracker when predicting log revisions. For ease of comparison, we calculate the value of F_{score} (F).

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 predicts log revisions by applying these rules. We compare the predicted log revisions with train and test data, thus calculating the value of precision, recall and F_{score} with following formulas.

$$Precision = \frac{Predicted\ log\ revisions\ in\ test\ data}{Predicted\ log\ revisions} \tag{2}$$

$$Recall = \frac{Predicted\ log\ revisions\ in\ test\ data}{Log\ revisions\ in\ test\ data} \tag{3}$$

$$F_{score} = \frac{2 * Precision * Recall}{Precision\ +\ Recall} \tag{4}$$

Above process is repeated five times to get the average value.

Table 7 displays the precision and the recall of four subject projects[9] in response to two ratios of train data. Generally speaking, the precision is high in both cases and indicates that LogTracker is a reliable tool when guiding log revisions. Comparatively, the recall of LogTracker is lower. LogTracker predicts 19.7% log revisions with half historical log revisions. In addition, as the ratio of train data increases, the recall rises from 19.7% to 38.1%. 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 51.0% of historical log revisions share similar logging context. That is to say, for our methods, the theoretical value of recall is 51.0%. Second, in this experiment, LogTracker is trained with partial historical data, it is unavoidable to miss some rules and generate a lower recall.

[8]Here "out-date rule" indicates one rule that is out of date.
[9]As shown in Table 3 and 4, 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: Groups of similar log revisions

Software	M	I	Total	Software	M	I	Total
Httpd	38	15	53	Collectd	20	5	25
Git	24	14	38	Postfix	34	11	45
Mutt	7	0	7	Tar	5	1	6
Rsync	7	0	7	Wget	8	4	12
Total	143	50	**193**				

Table 9: Accuracy of locating context-similar log revisions

Software	M	I	Software	M	I
Httpd	97.4%	86.7%	Collectd	90.0%	80.0%
Git	91.7%	92.9%	Postfix	94.1%	72.7%
Mutt	100.0%	-	Tar	100.0%	100.0%
Rsync	100.0%	-	Wget	100.0%	75.0%
Total	**95.1%**	**84.0%**			

4.3 Accuracy of LCDM

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

4.3.1 Accuracy of Locating Context-Similar Log Revisions. We build an oracle test suit from eight subject projects. One author of this paper manually selects groups of log revisions that were similarly modified from all historical log revisions. It takes the author almost 600 hours to select these groups out. Then another expert who does not participate in the design and implementation of LogTracker validates the similarity of log revisions that belong to the same group. We finally identify 193 groups of similar log revisions (see Table 8). 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. We then apply these rules to historical versions of subject software, and collect candidates detected by LogTracker. By comparing candidates with test oracles and manual validation, we calculate the accuracy [10] when locating context-similar log revisions.

Table 9 displays the accuracy for two sorts of rules in eight subject projects. For rules that modify existing log statements, the accuracy is 95.1%. This is consistent with the high precision in section 4.2, and indicates that LCDM is accurate when describing the semantics of logging context.

These false positives are caused 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 raises further false alarms when locating context-similar log revisions.

In addition, the accuracy for rules that insert new log statements is evidently lower than the accuracy for rules that modify existing log statements. We manually check all the false positives and work out that there are two main cases.

[10] As mentioned in section 4.1, developers may miss log revisions. As such, recall of this experiment is not reliable and we do not mention it here.

Httpd-2.2.34, file: worker.c

```
static apr_status_t create_wakeup_pipe(apr_pollset_t *pollset){
 ...
 return apr_pollset_add(pollset,&pollset->wakeup_pfd);}
```

(a) False positive caused by return statement

Httpd-2.2.34, file: pollset.c

```
static void *listener_thread(apr_thread_t *thd, void * dummy){
 ...
 for(lr=ap_listeners;lr!=NULL;lr=lr->next){ ...
  (void)apr_pollset_add(pollset, &pfd);} ...}
```

(b) Flase positive caused by loop

Figure 10: False positives in application of rules that insert new log statements

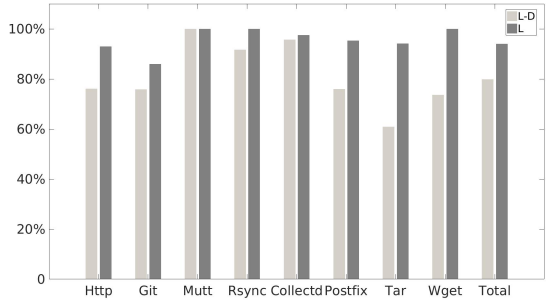

Figure 11: Accuracy of LogTracker and LogTracker-DECKARD

First, the related functions whose return value should be checked and logged, are invoked in the return statement. In Figure 10a, 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.

$$Log\,rate = \frac{Times\,of\,being\,logged}{Times\,of\,being\,invoked} \tag{5}$$

Second, the related function is invoked inside the loop structure. In Figure 10b, 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.

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 accuracy 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 accuracy of LogTracker-DECKARD using the oracle test suit and methods in section 4.3.1, and compare this accuracy with that of LogTracker. Figure 11 indicates that the accuracy of LogTracker is higher. This result is consistent with the discussion in section 3.2.3 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 LogTracker recommends proactive log revisions by applying rules that are learned from historical log revisions. Consequently, the effectiveness of LogTracker depends on the quality of log revisions in software evolution. In fact, it is unavoidable that developers will commit false log revisions that may adversely impact the effectiveness of LogTracker. Thus, to assure the quality of historical log revisions, we have selected eight open-source projects that are popular in their respective fields and have a long development history.

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 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 accuracy of LogTracker is closely related to that of fuzzy parsing techniques. Section 4.2 has evaluated the precision of LogTracker and indicates that the deviation is acceptable.

Accuracy of Oracle Construction For lack of benchmark, the oracle data set used in section 4.3 is built manually. In this way, the accuracy of that experiment is seriously affected by the correctness of manual analysis. To be conservative, we choose one expert who has not participated in the design and implementation of LogTracker to validate the correctness of oracle.

6 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 [15, 29, 37]. As a prerequisite, many researchers have devoted to summarizing the characteristics of existing logging practices. Yuan et al. [36] 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 et al. [2] 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 at. [15] conducted an empirical study on two industrial software and a questionnaire survey on 54 experienced developers. 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 et al. [3] 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. [24] identified whether new commitments require log modifications to reduce after-thought 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.

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 [21] and CPMiner [25] detected code clones with token vectors that are generated by lexical parser. DECKARD [19], DECKARD+ and CloneDetective [20] detected code clones with features that are generated by syntactical structures. Among clone management tools, SysEdit [27] 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 [28] 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 Figure 8a and 8b). Thus, it is difficult to predict systematic log revisions with LASE. REFAZER [31] utilized program synthesis [30] 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. Above researches on clone detection and management have motivated our design of LogTracker which detects and manages logging code under similar context.

7 CONCLUSIONS

Previous works ignored that logging code co-evolves with bug fixes or feature updates. To release this problem, we propose to learn log revision proactively from software evolution. The empirical study figure 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.

ACKNOWLEDGEMENT

The work described in this paper was supported by National Natural Science Foundation of China (Project No.61690203 and U1711261); National Key R&D Program of China (Project No.2017YFB1001802 and 2017YFB0202201).

REFERENCES

[1] Matthew Arnold and Barbara G. Ryder. 2001. A framework for reducing the cost of instrumented code. *ACM SIGPLAN Notices* 36, 5 (2001), 168–179. https://doi.org/10.1145/381694.378832

[2] Boyuan Chen and Zhen Ming (Jack) Jiang. 2017. Characterizing logging practices in Java-based open source software projects - a replication study in Apache Software Foundation. *Empirical Software Engineering* 22, 1 (2017), 330–374. https://doi.org/10.1007/s10664-016-9429-5

[3] Boyuan Chen and Zhen Ming Jiang. 2017. Characterizing and Detecting Anti-Patterns in the Logging Code. *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering, ICSE 2017* (2017), 71–81. https://doi.org/10.1109/ICSE.2017.15

[4] Michael L. Collard, Michael John Decker, and Jonathan I. Maletic. 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, 516–519. https://doi.org/10.1109/ICSM.2013.85

[5] Collectd. 2017. Start page - collectd - The system statistics collection daemon. (2017). http://collectd.org/

[6] Software Freedom Conservancy. 2018. Git. (2018). https://git-scm.com/

[7] Wayne Davison. 2018. rsync. (2018). https://rsync.samba.org/

[8] Rui Ding, Hucheng Zhou, Jian-Guang Lou, Hongyu Zhang, Qingwei Lin, Qiang Fu, Dongmei Zhang, and Tao Xie. 2015. Log 2: a cost-aware logging mechanism for performance diagnosis. *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference - USENIX ATC '15* (2015), 139–150.

[9] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Montperrus. 2014. Fine-grained and accurate source code differencing. *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering - ASE '14* (2014), 313–324. https://doi.org/10.1145/2642937.2642982

[10] Free Software Foundation. 2016. Diffutils - GNU Project - Free Software Foundation. (2016). https://www.gnu.org/software/diffutils/

[11] Free Software Foundation. 2017. Tar - GNU Project - Free Software Foundation. (2017). https://www.gnu.org/software/tar/

[12] Free Software Foundation. 2017. Wget - GNU Project - Free Software Foundation. (2017). https://www.gnu.org/software/wget/

[13] Python Software Foundation. 2018. Built-in Functions-Python 2.7.14 documentation. (2018). https://docs.python.org/2/library/functions.html

[14] The Apache Software Foundation. 2017. httpd - Apache Hypertext Transfer Protocol Server - Apache HTTP Server Version 2.4. (2017). http://httpd.apache.org/docs/2.4/programs/httpd.html

[15] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. 2014. Where do developers log? an empirical study on logging practices in industry. *Proceedings of the 36th International Conference on Software Engineering - ICSE '14* (2014), 24–33. https://doi.org/10.1145/2591062.2591175

[16] Mark Gabel, Lingxiao Jiang, and Zhendong Su. 2008. Scalable detection of semantic clones. *Proceedings of the 30th international conference on Software engineering - ICSE '08* (2008), 321. https://doi.org/10.1145/1368088.1368132

[17] Github. 2018. GitHub - GumTreeDiff/gumtree: A neat code differencing tool. (2018). https://github.com/GumTreeDiff/gumtree

[18] GitHub. 2018. skyhover/Deckard: Code clone detection; clone-related bug detection; semantic clone analysis. (2018). https://github.com/skyhover/Deckard

[19] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stéphane Glondu. 2007. DECKARD: Scalable and accurate tree-based detection of code clones. *Proceedings of the 29th International Conference on on Software Engineering - ICSE '07* (2007), 96–105. https://doi.org/10.1109/ICSE.2007.30

[20] Elmar Juergens, Florian Deissenboeck, and Benjamin Hummel. 2009. CloneDetective - A workbench for clone detection research. *Proceedings of the 31th International Conference on Software Engineering - ICSE '09* (2009), 603–606. https://doi.org/10.1109/ICSE.2009.5070566

[21] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 28, 7 (2002), 654–670. https://doi.org/10.1109/TSE.2002.1019480

[22] kevin8t8. 2018. The Mutt E-Mail Client. (2018). http://www.mutt.org/

[23] Miryung Kim, Vibha Sazawal, and David Notkin. 2005. An empirical study of code clone genealogies. *ACM SIGSOFT Software Engineering Notes* 30, 5 (2005), 187. https://doi.org/10.1145/1095430.1081737

[24] Heng Li, Weiyi Shang, Ying Zou, and Ahmed E. Hassan. 2017. Towards just-in-time suggestions for log changes. *Empirical Software Engineering* 22, 4 (2017), 1831–1865. https://doi.org/10.1007/s10664-016-9467-z

[25] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. 2004. CP-Miner : A Tool for Finding Copy-paste and Related Bugs in Operating System Code. *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - OSDI '04* (2004), 20. https://doi.org/10.1109/TSE.2006.28

[26] Slashdot Media. 2018. SLOCCount download | SourceForge.net. (2018). https://sourceforge.net/projects/sloccount/

[27] Na Meng, Miryung Kim, and Kathryn S. McKinley. 2011. Systematic editing. *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation - PLDI '11* (2011), 329. https://doi.org/10.1145/1993498.1993537

[28] Na Meng, Miryung Kim, and Kathryn S. McKinley. 2013. LASE : Locating and Applying Systematic Edits by Learning from Examples. *Proceedings of the 35th International Conference on Software Engineering - ICSE '13* (2013), 502–511.

[29] Antonio Pecchia, Marcello Cinque, Gabriella Carrozza, and Domenico Cotroneo. 2015. Industry Practices and Event Logging: Assessment of a Critical Software Development Process. *Proceedings of the 37th IEEE International Conference on Software Engineering - ICSE '15* (2015), 169–178. https://doi.org/10.1109/ICSE.2015.145

[30] Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: a framework for inductive program synthesis. *ACM SIGPLAN Notices* 50, 10 (2015), 107–126. https://doi.org/10.1145/2858965.2814310

[31] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Bjorn Hartmann. 2017. Learning syntactic program transformations from examples. *Proceedings of the 39th International Conference on Software Engineering - ICSE '17* (2017), 404–415. https://doi.org/10.1109/ICSE.2017.44

[32] Warren S. Sarle, Anil K. Jain, and Richard C. Dubes. 1990. Algorithms for Clustering Data. *Technometrics* 32, 2 (1990), 227. https://doi.org/10.2307/1268876 arXiv:tesxx

[33] Benjamin H Sigelman, Luiz Andr, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper , a Large-Scale Distributed Systems Tracing Infrastructure.* Technical Report. California, USA. https://doi.org/dapper-2010-1

[34] Wietse Venema. 2013. The Postfix Home Page. (2013). http://www.postfix.org/

[35] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, and Mm Lee. 2012. Be conservative: enhancing failure diagnosis with proactive logging. *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation - OSDI '12* 41, 6 (2012), 293–306.

[36] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. 2012. Characterizing logging practices in open-source software. In *Proceedings of the 34th International Conference on Software Engineering - ICSE '12*. 102–112. https://doi.org/10.1109/ICSE.2012.6227202

[37] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. 2012. Improving Software Diagnosability via Log Enhancement. *ACM Transactions on Computer Systems* 30, 1 (2012), 1–28. https://doi.org/10.1145/2110356.2110360

[38] Xu Zhao, Kirk Rodrigues, and Michael Stumm. 2017. Log20 : Fully Automated Optimal Placement of Log Printing Statements under Specified Overhead Threshold. *Proceedings of the 26th Symposium on Operating Systems Principles - SOSP '17* (2017), 565–581. https://doi.org/10.1145/3132747.3132778

[39] Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R. Lyu, and Dongmei Zhang. 2015. Learning to log: Helping developers make informed logging decisions. *Proceedings of the 37th International Conference on Software Engineering - ICSE '15* (2015), 415–425. https://doi.org/10.1109/ICSE.2015.60