Noname manuscript No.
(will be inserted by the editor)

Extracting and Studying the Logging-Code-Issue-Introducing Changes in Java-based Large-Scale Open Source Software Systems

Boyuan Chen · Zhen Ming (Jack) Jiang

Received: date / Accepted: date

Abstract Execution logs, which are generated by logging code, are widely used in modern software projects for tasks like monitoring, debugging, and remote issue resolution. Ineffective logging would cause confusion, lack of information during problem diagnosis, or even system crash. However, it is challenging to develop and maintain logging code, as it inter-mixes with the feature code. Furthermore, unlike feature code, it is very challenging to verify the correctness of logging code. Currently developers usually rely on their intuition when performing their logging activities. There are no well established logging guidelines in research and practice.

In this paper, we intend to derive such guidelines through mining the historical logging code changes. In particular, we have extracted and studied the Logging-Code-Issue-Introducing (LCII) changes in six popular large-scale Java-based open source software systems. Preliminary studies on this dataset show that: (1) both co-changed and independently changed logging code changes can contain fixes to the LCII changes; (2) the complexity of fixes to LCII changes are similar to regular logging code updates; (3) it takes longer for developers to fix logging code issues than regular bugs; and (4) the state-of-the-art logging code issue detection tools can only detect a small fraction (3%) of the LCII changes. This highlights the urgent need for this area of research and the importance of such a dataset.

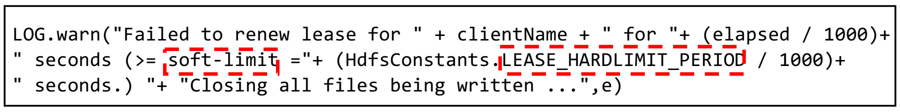
B. Chen
Software Construction, AnaLytics and Evaluation (SCALE) lab
York University, Toronto, ON, Canada
E-mail: chenfsd@cse.yorku.ca

Z.M. Jiang
Software Construction, AnaLytics and Evaluation (SCALE) lab
York University, Toronto, ON, Canada
E-mail: zmjiang@cse.yorku.ca

1 Introduction

Execution logs, which are usually readily available for large-scale software systems, have been widely used in practice for a variety of tasks (e.g., system monitoring [42], problem debugging [49], remote issue resolution [36], test analysis [28], and business decision making [17]). Execution logs are generated by executing the logging code (e.g., `Logger.info("User " + userName + " logged in")`) that developers have inserted into the source code. There are typically four types of components in a snippet of logging code: a logging object, a verbosity level, static texts, and dynamic contents. In the above example, the logging object is `Logger`, the verbosity level is `info`, the static texts are `User` and `logged in`, and the dynamic content is `userName`.

It is very challenging to develop and maintain high quality logging code for constantly evolving systems for the following two reasons: (1) **Management**: software logging is a cross-cutting concern, which tangles with the feature code [31]. Although there are language extensions (e.g., AspectJ [15]) to support better management of logging code, many industrial and open source systems still choose to inter-mix logging code with feature code [19,38,50]. (2) **Verification**: unlike feature code or other types of cross-cutting concerns (e.g., exception handling or configuration), whose correctness can be verified via software testing; it is very challenging to verify the correctness of the logging code. Figure 1 shows one such issue in the logging code. In the Hadoop DFSClient source code, the variable name was changed from `LEASE_SOFTLIMIT_PERIOD` to `LEASE_HARDLIMIT_PERIOD`. However, the developer forgot to update the static text from `soft-limit` to `hard-limit`. Such issues are very hard to be detected using existing software verification techniques, except conducting careful manual code reviews. Developers have to rely on their intuition to compose, review, and update logging code. Most of the existing issues in logging code (e.g., inconsistent or out-dated static texts, and wrong verbosity levels) are discovered and fixed manually [19,50].



```
LOG.warn("Failed to renew lease for " + clientName + " for "+ (elapsed / 1000)+
" seconds (>= soft-limit ="+ (HdfsConstants.LEASE_HARDLIMIT_PERIOD / 1000)+
" seconds.) "+ "Closing all files being written ...",e)
```

Fig. 1: An example of an issue in the logging code [11].

Most of the existing research on logging code focuses on "where-to-log" (a.k.a., suggesting where to add logging code) [23,26,53] and "what-to-log" (a.k.a., providing sufficient information in the logging code) [29,44,51]. There are very few works tackling the problem of "how-to-log" (a.k.a., developing and maintaining high quality logging code). Low quality logging code can hinder program understanding [44], cause performance slow-down [4], or even system crashes [6]. Unlike other software engineering processes (e.g., refactoring [25] and release management [27]), there are no well-established logging practices in industry [26,38]. Our previous work [20] is the first study, which characterizes and detects anti-patterns (common issues) in logging code by manually examining a sample of the logging code changes from three open source projects. Six anti-patterns in the logging code (ALC) were identified. The majority (72%) of the reported ALC instances, on the most recent releases of the ten open source systems, have been accepted or fixed by their developers. This clearly demonstrates the need for this area of research. However, one of the main obstacles facing researchers is the lack of the available dataset, which contains the Logging-Code-Issue-Introducing (LCII) changes, as many issues in the logging code are generally not documented in the commit logs or in the bug reports. An LCII change is analogous to a bug introducing change [32]. It is a type of code change, which will lead to future changes (e.g., changing the static texts for clarification or lowering the verbosity levels to reduce the runtime overhead) to the corresponding logging code. Logging code changes which are co-evolved with feature code are not included as LCII changes.

In this paper, we have developed a general approach to extracting the LCII changes by mining the projects' historical data. Our approach analyzes the development history (historical code changes and bug reports) of a software system and outputs a list of LCII changes for further analysis. We have performed a few preliminary studies on the resulting LCII change dataset and presented some open problems in this area. The contributions of this paper are as follows:

1. Compared to [20], using our new approach, the resulting LCII changes are more complete. In [20], the authors assumed that only the independently changed logging code changes (a.k.a., the logging code changes which are not co-changed with any feature code changes) may contain fixes to the LCII changes. In this paper, we have found that this assumption is invalid, as some fixes to the LCII changes may require feature code changes as well. Thus, rather than only focusing on the independently changed logging code, we extract the LCII changes from all the logging code changes.
2. Instead of manually identifying LCII changes as in [20], we have developed an adapted version of the SZZ algorithm [45], called LCC-SZZ (Logging Code Change-based SZZ), to automatically locate the LCII changes from their fixes. By leveraging this algorithm, we can extract the LCII changes among various code revisions.
3. To ease replication and encourage further research in the area of "how-to-log", we have provided a large-scale benchmarking dataset, which contains 8,748 LCII changes from six large-scale open source systems, each consisting of six to ten years of development history [16]. Such a dataset, which is the first of its kind to the authors' knowledge, can be very useful for interested researchers to develop new techniques to characterize and detect ALCs or to derive coding guidelines on effective software logging.
4. We have conducted some preliminary studies on the extracted LCII change dataset and reported four main findings: (1) both co-changed and inde-

pendently changed logging code can contain fixes to LCII changes; (2) the complexity of the LCII changes and other logging code changes are similar; (3) it takes significantly longer to fix a logging code issue than a regular bug, and (4) although none of the existing techniques on detecting logging code issues perform well (< 3% recall), their detection results complement each other. Our findings clearly indicate the need in this area of research and demonstrate the usefulness of our approach and the provided dataset.

Paper Organization

The rest of the paper is organized as follows. Section 2 gives an overview about the extraction process for the LCII changes. Section 3, 4, and 5 illustrate the three phases of our extraction approach. Section 6 describes our preliminary studies on the extracted dataset. Section 7 presents the related work. Section 8 discusses the threats to validity. Section 9 concludes the paper and describes some of the future work.

2 An Overview of Our Approach to Extracting the LCII Changes

In this section, we will explain our approach to extracting the LCII changes from the software development history. As shown in Figure 2, our approach consists of the following three phases:

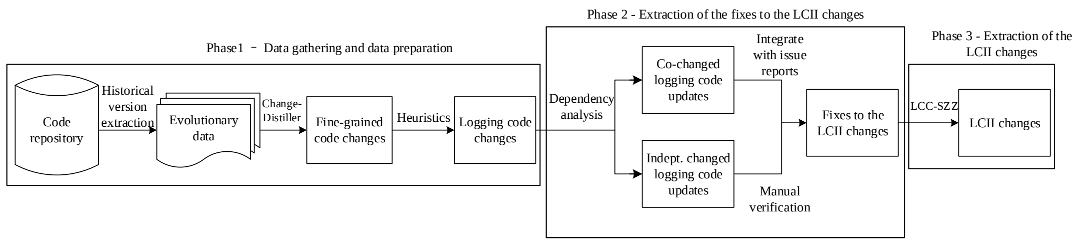

Phase1 - Data gathering and data preparation
Code repository
Historical version extraction
Evolutionary data
Change-Distiller
Fine-grained code changes
Heuristics
Logging code changes
Phase 2 - Extraction of the fixes to the LCII changes
Dependency analysis
Co-changed logging code updates
Indept. changed logging code updates
Integrate with issue reports
Manual verification
Fixes to the LCII changes
Phase 3 - Extraction of the LCII changes
LCC-SZZ
LCII changes

Fig. 2: Overall process.

1. **Phase 1 - Data gathering and data preparation** (Section 3): this is the data gathering and pre-processing phase. First, the versions of each source file are extracted from the source code repositories. Then, all the bug reports are downloaded for the studied project. Fine-grained code changes (e.g., function update, variable renaming, etc.) are identified between adjacent versions of the same files. Finally, heuristics are applied to automatically identify the changes which are related to the logging code.

2. **Phase 2 - Extraction of the fixes to the LCII changes** (Section 4): in order to identify the LCII changes, we first need to identify their fixes among all the logging code changes. In this phase, we extract the fixes to the LCII changes by carefully examining both the independently changed and co-changed logging code changes.
3. **Phase 3 - Extraction of the LCII changes** (Section 5): unlike the feature code changes in which each changed line can be traced back to a previous version, the process of identifying the LCII changes is different. As the logging code is tangled with the feature code, one line of logging code changes can be related to multiple feature code changes. For example, in one code commit, developers may update the static texts (due to method renaming) as well as the method invocations (due to changes in the method signatures) in one single line of the logging code. Hence, in this phase, we have developed an adapted version of the SZZ algorithm, called LCC-SZZ, to automatically identify LCII changes.

The next three sections (Section 3, 4, and 5) will explain the above three phases in details.

3 Phase 1 - Data Gathering and Data Preparation

We will first briefly describe the studied projects. Then we will explain our process to extract the logging code changes and bug reports.

3.1 Studied Projects

In this paper, we focus our study on six popular Java-based open source software projects: Elasticsearch, Hadoop, HBase, Hibernate, Geronimo, and Wildfly. Table 1 provides an overview of the studied projects. These projects are from different application domains. The selected projects are either server-side or supporting component-based projects, since the previous study [19] showed that software logging is more pervasive and more actively maintained in these two categories than in client-side projects. Each studied project has one or more bug repositories (GitHub or Jira) as shown in the third column of Table 1. As shown in the fourth column of the table, all the selected projects have a relatively long development history: ranging between six to ten years. The fifth column (LCC) shows the total number of logging code changes throughout the development history. LCC stands for the Logging Code Changes. It refers to the code changes which are directly related to software logging. Additional/replacement/removal of a variable inside a debug statement, changes to the verbosity levels, or adding and removing the static texts are three examples of LCCs. Here we only focus on the direct changes to the logging code. For example, if only the condition outside a logging statement is changed, such change is not considered as an LCC, as the changes may related to some

Table 1: Studied projects.

Project	Description	Bug Repository	Code History	LCC
Elasticsearch	Distributed analysis engine	GitHub	(2010-02-08, 2017-03-30)	2,781
Hadoop	Distributed compute platform	Jira	(2009-05-19, 2017-08-22)	2,652
HBase	Distributed database	Jira	(2007-04-03, 2017-05-05)	3,638
Hibernate	ORM framework	Jira	(2007-06-29, 2017-07-25)	2,619
Geronimo	Server runtime	Jira	(2003-08-07, 2013-03-21)	1,019
Wildfly	Application server	Jira, GitHub	(2010-05-28, 2017-03-30)	3,401

feature code changes. As we can see the logging code from all six projects are actively maintained: each containing thousands of logging code changes.

There are two different types of data that we need to extract from the historical repositories in order to identify the LCII changes: the logging code changes, and the issue reports.

3.2 Logging Code Changes

All six studied projects use GitHub as their source code repositories. We wrote a script to automatically extract the code version history from the master branch. We only focused on the code changes committed to the master branch, as the logging code changes committed there had been carefully reviewed. The extracted data includes every version of every source code file , along with the meta information of each code version (e.g., the commit hash, the commit date, the authors, etc.). For every file, we then sorted these code versions by their commit timestamps. For example, if the file `Foo.java` was changed in two commits (*hash:f4b214* and *hash:a7cc6b*), which correspond to the first and the second commits, it would result in two extracted versions named `Foo_v1.java` and `Foo_v2.java`.

We ran ChangeDistiller (CD) [24] to get the source code level changes between two adjacent versions. CD parses two adjacent versions (e.g., `Foo_v1.java` and `Foo_v2.java`) of the same the file into Abstract Syntax Trees (ASTs) and compares the ASTs using a tree differencing algorithm. The output of CD contains a list of fine-grained code changes from four categories: insertion, deletion, update, and move. Since our focus is on the changes to the existing logging code, we only analyzed the updated code changes like update to a particular method invocation or update to a variable assignment.

Once we obtained the source code level changes, we applied keyword-based heuristics to filter out the non-logging code changes. We searched for commit messages which include the words like “log”, “trace”, and “debug”. We ruled out the changes which contained the mismatched words like “dialog”, “login”, etc. We also excluded the logging code which did not print any messages (e.g., logging verbosity level setting statement like `log.setLevel(org.apache.log-`

`4j.Level.toLevel(level))`). Such keyword-based heuristics have been used in many of the previous studies (e.g., [19,20,26,43,50]) to identify logging code changes with high precision.

3.3 Issue Reports

The studied projects use two kinds of bug tracking systems: Jira and GitHub. For Jira-based projects (Hadoop, HBase, Hibernate, Geronimo, and Wildfly), we followed a similar approach in [19] to crawl the Jira reports using the provided Jira APIs. For Elasticsearch and Wildfly, their issues are managed through GitHub in the form of pull requests and GitHub issues. We used the public APIs to crawl the related GitHub data.

4 Phase 2 - Extraction of the Fixes to the LCII Changes

We have categorized the logging code changes into the following two categories:

– *co-changed logging code*, in which the logging code is updated together with the corresponding feature code. In [20], there is an assumption that the fixes to the LCII changes only exist in the independently changed logging code. It is not clear whether such an assumption is valid or not.
– *independently changed logging code*, refers to the logging code changes which are not classified as co-changed logging code changes. Unlike the co-changed logging code changes, which are generally updated along with the feature code, independently changed logging code is usually about fixing LCII changes. However, it is not clear whether all the independently changed logging code is actually used to fix the LCII changes.

In the previous study [19], we have identified the following eight scenarios of the co-changed logging code changes: (1) co-change with condition expressions, (2) co-change with variable declarations, (3) co-change with feature methods, (4) co-change with class attributes, (5) co-change with variable assignment, (6) co-change with string method invocations, (7) co-change with method parameters, and (8) co-change with exception conditions. We encoded these rules to automatically identify the co-changed logging code changes. The remaining logging code changes belong to the independently changed logging code changes.

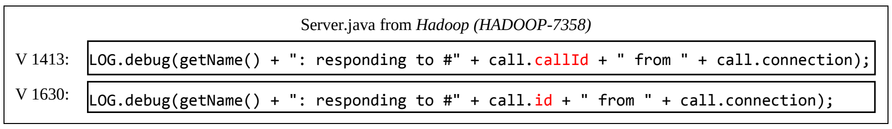
In the rest of this section, we will explain our process to extract fixes to the LCII changes by mining the independently changed and co-changed logging code changes. During this process, we will validate the above two assumptions.

4.1 Extracting Fixes to the LCII Changes from the Co-changed Logging Code Changes

We extract fixes to the LCII changes from the co-change logging code changes by leveraging the information contained in the filed bug reports. Usually, developers would include the bug report IDs in the commit logs for traceability [18] and code review [39]. For example, in Hadoop, there is a commit with the commit log stating the following: "*HADOOP-15198. Correct the spelling in CopyFilter.java. Contributed by Mukul Kumar Singh.*". This commit refers to the bug ID `HADOOP-15198`, whose details can be found by searching this Jira ID online (https://issues.apache.org/jira/browse/HADOOP-15198). Hence, we extracted the bug IDs mentioned in the commit logs to link to the corresponding Jira issues. Then we performed keyword-based filtering to only include the bug reports which addressed the logging issues by searching the subject of the issue reports with keywords like "logging", "logs", and "logging code".

We did not search the same keywords in the bug description and the comments sections. Many of the reported issues include logs or logging code in their bug descriptions or comments as an artifact to help facilitate developers to understand or reproduce the reported issues. Thus, searching through these two sections would cause too much noise in the resulting dataset. For example, Hadoop issue `HADOOP-12666` [1] is about "Support Microsoft Azure Data Lake" and in the discussion, someone pasted a code snippet which contained logging code. However, this issue report is obviously not related to the issues in logging code. Hence, to avoid too many false positives, we chose to only match those keywords in the title of the bug reports.

The six studied projects contain a total of 7, 092 co-changed logging code changes. There are 553 (8%) co-changed logging code changes which are linked to the log-related issue reports. For each of these changes, we manually categorized their change types based on their intentions and whether they belong to the fixes to the LCII changes. There are 130 logging code changes, which are not related to the LCII changes. Figure 3 shows one such example from the `Server.java` file in the Hadoop project. This code change is linked to the Hadoop issue `HADOOP-7358` [2]. The attribute of the `call` object was changed from `id` to `callId`. However, the bug report was about improving "log levels when exceptions caught in RPC handler", which is not related to this logging code changes. After filtering such irreverent changes, we ended up with 423 co-changed logging code changes, which are fixes to the LCII changes.


Server.java from *Hadoop (HADOOP-7358)*

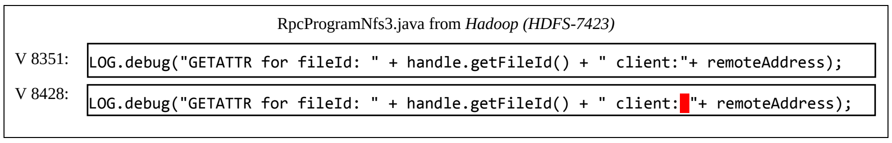
V 1413:
```
LOG.debug(getName() + ": responding to #" + call.callId + " from " + call.connection);
```
V 1630:
```
LOG.debug(getName() + ": responding to #" + call.id + " from " + call.connection);
```

Fig. 3: An example of the co-changed logging code, which is linked to a log-related bug report. However, it is not a fix to an LCII change.

4.2 Extracting Fixes to the LCII Changes from the Independently Changed Logging Code Changes

There are a total of 9, 018 independently changed logging code changes. Due to its sheer size, we can only manually examine a few sampled instances. We randomly selected 369 independently changed logging code changes for manual investigation. This corresponds to a confidence level of 95% with a confidence interval of 5%.

During our analysis, we found one scenario of the independently changed logging code changes, which are not fixes to the LCII changes. This type of logging code changes modifies the printing format (e.g., adding or removing spaces) without changing the actual contents. Figure 4 shows one such example. The only change for that logging statement in the new revision was adding a space after the word `client` in the static texts. There were no changes in any of the four logging components (logging library, verbosity level, static texts, or dynamic information). Hence, we automatically filtered out all the printing format changes from all the independently changed logging code changes using a script. We ended up with 8, 325 (92%) independently changed logging code changes, which are fixes to the LCII changes.



RpcProgramNfs3.java from *Hadoop (HDFS-7423)*

```
V 8351: LOG.debug("GETATTR for fileId: " + handle.getFileId() + " client:"+ remoteAddress);
V 8428: LOG.debug("GETATTR for fileId: " + handle.getFileId() + " client: "+ remoteAddress);
```

Fig. 4: An example of the printing format change.

5 Phase 3 - Extraction of the LCII Changes

In the previous phase, we have generated a dataset which contains the fixes to the LCII changes. In this phase, we will extract the code commits containing the LCII changes.

5.1 Our Approach to Extracting the LCII Changes

We used an adapted approach of the SZZ algorithm [45] to automatically identify the LCII changes from their fixes. The SZZ algorithm [45] is an approach which automatically identifies the bug introducing changes from their fixes. It first tags code changes to bug fixing changes by searching for bug reports. Then it traces through the code revision history to find when the changed code was introduced (bug introducing changes). There are many follow-up works to further enhance the effectiveness of this approach [22,32,46]. However, all these approaches are focused on identifying bugs in the feature code, directly applying these approaches on the fixes to LCII changes may lead to incomplete or incorrect results. Different from the feature code changes, one line of logging code changes can be related to multiple previous lines of the logging code changes.

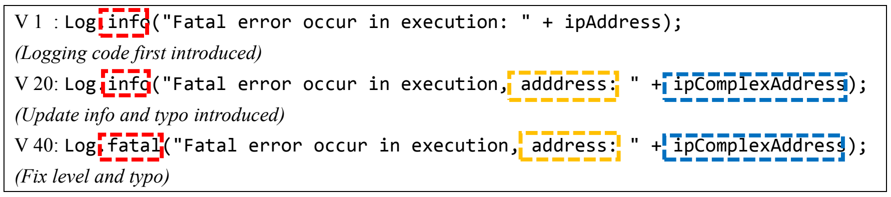
We will illustrate the problem of using the SZZ algorithm to locate the LCII changes using an example shown in Figure 5. V_n represents the version of the file. The file was changed through V_1 to V_{40} while the logging code was introduced at V_1 and subsequently was changed at V_{20} and V_{40}. The change at V_{40} is a fix to the LCII change since it corrected the verbosity level from `info` to `fatal` in order to be consistent with the text `Fatal error`. In addition, in the same change, the developer corrected a typo from `adddress` to `address` in the static texts. By using the SZZ algorithm, we find out that the problematic logging code (at V_{39}) was introduced at V_{20}. Therefore, we considered the LCII change to be at V_{20}. However, after examining the entire code revision history related to this logging code snippet, we noticed that the logging code at V_{20} is not the only LCII change. There are two problems associated with the logging code at V_{39}: the verbosity level and the static texts. The typo of the static texts was introduced in (V_{20}), and the verbosity level `info` was introduced when the logging code was initially added (V_1). Hence, for fix version V_{40}, there are two LCII change versions: V_1 and V_{20}.

The main problem with the original SZZ algorithm is that it would consider logging code as one entity instead of treating the various components (logging object, verbosity level, static texts, and dynamic contents) of the logging code separately. To cope with this problem, we have developed an adapted version of the SZZ algorithm, called LCC-SZZ. There are two inputs for the LCC-SZZ algorithm: (1) a list containing the historical revisions of a particular snippet of the logging code, and (2) the versions at which the logging code are fixed to resolve the LCII change. The output of the LCC-SZZ algorithm is the version(s) of LCII change(s). The pseudo code of this algorithm is shown in Algorithm 1.

The whole process contains two steps: (1) formulating a list of component chains; and (2) finding the version(s) of the LCII change(s) for each fixed component.

In step 1, LCC-SZZ breaks down each version of that logging code snippet into various components. Each component has a type and a string representation. The type could be logging library, verbosity level, static texts, or dynamic information (e.g. variables, method invocations, etc.). The string representation is the value of that component. For example, the string representation of a variable is the variable name. We then track the historical changes for each component and formalize them as component chains. Therefore, for a list of logging code, we have multiple component chains and grouop them together as the component chain list. This step is done by the procedure `CHAIN_FORMULATION`. In the beginning, the component chain list is created as an empty list on line 2. From line 3 to 6, the components for each version of that logging code snippet are extracted. Then this extracted data is transformed through the `CHAIN_FORMULATION` procedure. The inputs for `CHAIN_FORMULATION` are the extracted components from the logging code and the component chain list. The `for` loop on line 15 is used to iterate through all components to match with the existing component chains. The `for` loop on line 17 is used to iterate through all the component chains to see if the current component can be matched to any of them. As shown on line 18, the component needs to have the same type with the component chain. If the component type is logging library or verbosity level, it will be put into the corresponding component chain (shown from line 19 to 22); as each logging code snippet only contains one logging library and one verbosity level. For other types of components, they are analyzed using their string representations. The old and the new string representations of the component are checked to see if they are similar from line 24 to 29 based on the two following criteria: (1) if the string edit distance between the two component string representations is less than 3; or (2) if the length of the longest substring from the two component string representations are larger than 3. We choose threshold value to be 3 based on a trial and error process. If either one of the above criteria is true, they are considered to be similar, and the current component is inserted to this chain as a new node. If there is no match found, we will initialize a new component chain with this component to be the head of the chain. This process is implemented on line 33. This component chain is then added to the component chain list. In the end, for a list of logging code, we have formulated a list of component chains (i.e., the `component_chain_list`). In our example shown in Figure 5, the logging code list contains three lines of logging code: V_1, V_{20}, and V_{40}. There are two fix versions: V_{20} and V_{40}. There are four component chains in the component chain list for this example. The component chain for the verbosity level is: "info(V_1) $\leftarrow$ info(V_{20}) $\leftarrow$ fatal(V_{40})". The component chain for the variable is: "ipAddress(V_1) $\leftarrow$ ipComplexAddress(V_{20}) $\leftarrow$ ipComplexAddress(V_{40})". All four resulting component chains are shown in Figure 6.

After the formulation of the component chain list, the next step (a.k.a., step 2) is to find the issue introducing version given a specified fix version. This process is explained in the procedure `EXTRACT_ISSUE_INTRODUCING_VERSION` starting from line 38. All the components which are changed in the fix version are retrieved. Then the components from the previous version of the fix version are picked, as they contain these issues. This step is implemented from line 40 to 43. For the components with issues in the previous version, the search continues until the first version when the issue appears is found. This process is implemented through the `while` loop on line 44. In our example, the fix versions are V_{40} and V_{20}. In fix version V_{40}, both the verbosity level, and the static texts are changed. Hence, the previous version (V_{20}) is examined. In V_{20}, the verbosity level is `info` whose first appearance is in the initial version of this logging code snippet, V_1. Similarly, at V_{40}, the spelling of `adddress` is changed to `address`, and the first appearance of this issue is at V_{20}. Therefore, for fix version V_{40}, there are two issue introducing versions: V_1 and V_{20}. In fix



```
V 1 : Log.info("Fatal error occur in execution: " + ipAddress);
```
(Logging code first introduced)
```
V 20: Log.info("Fatal error occur in execution, adddress: " + ipComplexAddress);
```
(Update info and typo introduced)
```
V 40: Log.fatal("Fatal error occur in execution, address: " + ipComplexAddress);
```
(Fix level and typo)

Fig. 5: An example of the logging code changes.

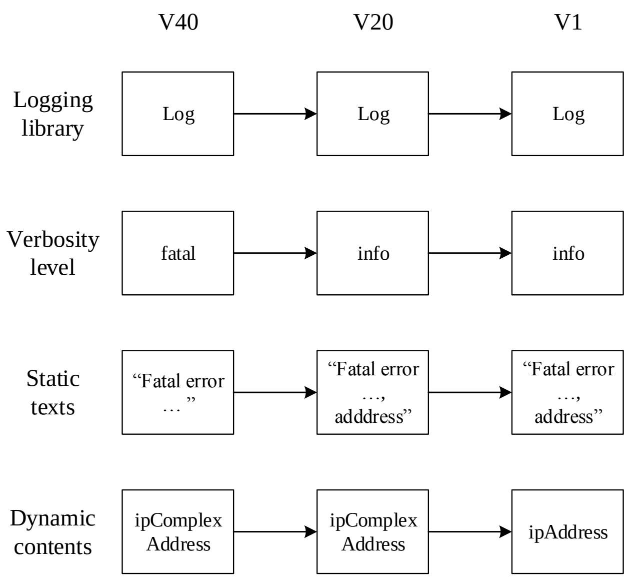


Fig. 6: The resulting component chains.

version V_{20}, the variable `ipAddress` is changed to `ipComplexAddress`. The issue introducing version is V_1. Hence, the LCII change versions for fix version V_{40} are V_1 and V_{20}. The LCII change version for fix version V_{20} is V_1.

5.2 Evaluation

To evaluate and compare the effectiveness of the LCC-SZZ and the SZZ algorithms, we used an approach similar to [21]. First, we applied both algorithms on the fixes to the LCII changes. Then we compared the results of the two algorithms from the following three dimensions: (1) disagreement ratio between the results from the two algorithms; (2) the earliest appearance of the LCII changes; and (3) manual verification. The first dimension is a general estimation of how these two algorithms differ. The second dimension is concerned

Algorithm 1 The pseudo code of the LCC-SZZ algorithm.

```
Input: logging_code_list, fix_version_list
procedure LCC_SZZ(logging_code_list, fix_version_list)
    component_chain_list ← [ ]
    for logging_code in logging_code_list do
        extracted_components ← extract_components(logging_code)
        CHAIN_FORMULATION(extracted_components, component_chain_list)
    end for
    for fix_version in fix_version_list do
        for component_chain in component_chain_list do
            EXTRACT_ISSUE_INTRODUCING_VERSION(fix_version, component_chain)
        end for
    end for
end procedure

procedure CHAIN_FORMULATION(extracted_components, component_chain_list)
    for component in extracted_components do
        find_chain ← false
        for component_chain in component_chain_list do
            if component.type=component_chain.type then
                if component.type = LEVEL or component.type = LIB then
                    component_chain.add(component)
                    find_chain ← true
                    break
                end if
                latest_component ← latest_component(component_chain)
                if similar_string(component,latest_component) = true then
                    component_chain.add(component)
                    find_chain ← true
                    break
                end if
            end if
        end for
        if find_chain = false then
            init_new_chain(component, component_chain_list)
        end if
    end for
end procedure

procedure EXTRACT_ISSUE_INTRODUCING_VERSION(fix_version, component_chain)
    for component in component_chain do
        if component.version = fix_version then
            issue_component ← component.previous
            issue_component_content ← component.previous.content
            if issue_component_content != component.content then
                while issue_component.content = issue_component_content do
                    if issue_component.previous = null then
                        break
                    end if
                    issue_component ← issue_component.previous
                end while
            end if
            print issue_component.version
        end if
    end for
end procedure
```

Table 2: Difference ratio of computed introducing versions by SZZ and LCC-SZZ.

Project	**Same**	**Different**	**Total**
Elasticsearch	758 (90.5%)	80 (9.5%)	838
Hadoop	905 (87.8%)	125 (12.2%)	1,030
HBase	1,276 (83.2%)	257 (16.8%)	1,533
Hibernate	1,657 (83.5%)	328 (16.5%)	1,985
Geronimo	489 (91.2%)	47 (8.8%)	536
Wildfly	2,553 (90.3%)	273 (9.7%)	2,826
Total	7,638 (87.3%)	1,110 (12.7%)	8,748

with the discrepancies between the two algorithms when compared to the estimates given by the development team. The third dimension is to estimate the accuracy by comparing against a human oracle.

5.2.1 The disagreement ratio between SZZ and LCC-SZZ

We calculated the disagreement ratio between the introducing versions generated by SZZ and LCC-SZZ. The results are shown in Table 2. In total, 12.7% of the results are different. Among the six studied projects, HBase has the largest difference ratio (16.8%) and Geronimo has the lowest difference ratio (8.8%).

5.2.2 The earliest appearance of the LCII changes

The evaluation on the earliest LCII changes is to compare the results from the SZZ and LCC-SZZ algorithms with the estimates provided by the development community. This evaluation dimension is not meant to tell whether SZZ/LCC-SZZ is absolutely correct. Rather, it aims to point out the obviously incorrectly outputted changes. Within each bug report, there is a field called "affected-version" showing the versions of the project that the logging issue impacted. For example, in HDFS, a component of the Hadoop system, the Jira issue `HDFS-4122` [10] shows the affected HDFS versions are 1.0.0 and 2.0.0-alpha. As the issue introducing date cannot be after the earliest affected version, we can use such a dataset to evaluate the SZZ/LCC-SZZ results. We consider the results from SZZ/LCC-SZZ to be incorrect if the resulting date reported by either algorithm is after the earliest affected version date. To notice, if the LCC-SZZ yields multiple version results, we use the date of the earliest version. For example, if the date of the earliest affected version is January 1, 2012 and the LCII changes outputted by the SZZ algorithm is October 1, 2012, we consider this output by the SZZ algorithm to be incorrect. However, if the resulting date is before the earliest affected version, we cannot judge the correctness of the outputted changes. Table 3 shows the details.

For Hadoop, HBase, Hibernate, and Geronimo, less than half ($\frac{535}{1168} \times 100\% = 46\%$) of the bug reports have valid data in the "affected-version"

Table 3: Comparing the time from the earliest bug appearance and the LCII code commit timestamp.

Project	**After affected version (SZZ)**	**After affected version (LCC-SZZ)**	**Bug reports with affected version**	**# of bug report**
Hadoop	41	29	286	407
HBase	27	25	87	397
Hibernate	2	2	141	332
Geronimo	0	0	21	32
Total	70 (13%)	56 (10%)	535	1,168

Table 4: Consistency compared to manual oracles.

Project	**SZZ**	**LCC-SZZ**	**Total**
Elasticsearch	25	29	35
Hadoop	37	41	43
HBase	53	60	64
Hibernate	75	76	84
Geronimo	20	22	23
Wildfly	105	115	119
Total	315	343	368

```
V 658:   logger.debug("Index [" + index + "]: Update mapping ["+ type+ "]
         (dynamic) with source ["+ updatedMappingSource+ "]")
V 715:   logger.debug("index [" + index + "]: Update mapping ["+ type+ "]
         (dynamic) with source ["+ updatedMappingSource+ "]")
V 736:   logger.debug("[{}] update mapping [{}] (dynamic) with source
         [{}]",index,type,updatedMappingSource)
```

MetaDataService.java in *ElasticSearch*

Fig. 7: Examples of both SZZ and LCC-SZZ failed.

field. Using the SZZ algorithm, 13% of the computed versions are after the earliest affected versions (a.k.a., incorrectly identified LCII changes). Using the LCC-SZZ algorithm, this number reduces to 10%, which is a 30% improvement compared to the original SZZ algorithm. The majority of these improvements come from Hadoop, which also happens to contain the largest number of bug reports with valid data in the “affected-version” field.

5.2.3 Manual verification

To further evaluate the results, we did a stratified sampling on all the fixes to the LCII changes, and went through the logging code change history to find the commits where the logging code snippets became issues. In this way, we generated a manually verified oracle dataset for the correctly identified LCII changes.

We calculated the difference ratio between the oracle and results from the SZZ/LCC-SZZ algorithms. The results are shown in Table 4. In total, we manually examined 368 fixes to the LCII changes and derived the oracle for code commits containing the LCII changes. 85% of the SZZ detecting results agree with the oracle while 93% of the LCC-SZZ results agree with the oracle. All of the LCII changes correctly identified by the SZZ algorithm are also correctly identified in the LCC-SZZ results.

For the case that both LCC-SZZ and SZZ failed, we manually examined a few of them. We found that the misclassified instances are mainly related to logging style changes. Figure 7 shows an example. The logging code style was changed to format printing from string concatenation. The logging code applied string concatenation style since introduction at version 658. Therefore, version 658 should be the LCII change version. However, there is a change from "Index " to "index" (fixing typos) from version 658 to 715. Both SZZ and LCC-SZZ mistakenly label 715 as the version containing the LCII changes.

Summary: our evaluation results show that: (1) 13.6% of the identified LCII changes obtained from the LCC-SZZ algorithm are different from the original SZZ algorithm; (2) when evaluating under the earliest appearance of the LCII changes, there are more incorrectly flagged results (3%) by the SZZ algorithm compared to the LCC-SZZ algorithm; (3) the LCC-SZZ algorithm achieves 93% accuracy when compared to the oracle, which is an 8% improvement compared to results from the SZZ algorithm.

6 Preliminary Studies

In the previous sections (Sections 3, 4, 5), we have explained our approach to extracting the LCII changes and evaluated the quality of the resulting dataset (93% accuracy based on manual verification). In this section, we will perform a few preliminary studies on this dataset to highlight the usefulness of such data and discuss some of the open problems. We want to study the characteristics of LCII changes by characterizing the intentions behind the fixes to the LCII changes (RQ1), studying the complexity of their fixes (RQ2), the duration to address these issues (RQ3), and the effectiveness of existing automated approaches to detecting logging code issues (RQ4). For each of the research question, we will describe our extraction process for the required dataset, explaining the data analysis process, and discuss its findings and implications.

6.1 RQ1: What are the intentions behind the fixes to the LCII changes?

In this RQ, we will characterize the intentions behind the fixes to the LCII changes. We intend to do this based on the two types of logging code changes

Project	TL	IND	IND_FIX	CO	CO_FIX	**T_FIX**
Elasticsearch	2,781	1,004	818	1,777	20	**838**
Hadoop	2,652	1,121	913	1,531	117	**1,030**
HBase	3,638	1,497	1,413	2,141	120	**1,533**
Hibernate	2,619	1,958	1,921	661	64	**1,985**
Geronimo	1,019	652	527	367	9	**536**
Wildfly	3,401	2,786	2,733	615	93	**2,826**
Total	16,110	9,018	8,325	7,092	423	**8,748**

Table 5: Summary of the fixes to LCII changes.

from two dimensions:(1) what-to-log vs. how-to-log, and (2) co-changed logging code changes vs. independently changed logging code changes.

6.1.1 Data Extraction

Table 5 shows the break down of the total number of logging code changes and the number of fixes to the LCII changes. TL shows the total number of logging code changes; IND shows the number of independently changed logging code changes; IND_FIX shows the number of IND which are fixes to LCII changes; CO shows the number of co-changed logging code changes; CO_FIX shows the number of CO which are fixes to the LCII changes; and T_FIX shows the total number of fixes to LCII changes. For example, in Hadoop, out of 2,652 logging code changes, 1,030 (38.8%) are fixes to the logging code changes. Out of these 1,030 fixes, 913 (88.6%) are independently changed logging code changes and 117 (11.4%) are co-changed logging code changes. Most of the independently changed logging code changes (92.3%) are fixes to the LCII changes. However, when we look at the co-changed logging code changes, majority of them (94.0%) are co-changes with the features code instead of fixes to the LCII changes. Among the total 8,748 fixes to LCII changes, majority of which (95.0%) are independently changed logging code changes.

6.1.2 An Overview of the Data Analysis Process

Since there are only 423 co-changed logging code changes, which are fixes to the LCII changes, we manually studied all of them. On the other hand, there are much more independently changed logging code changes than the co-changed logging code changes ($8,325$ vs. 423). It would take too much time for us to examine all the instances manually. Hence, we only examined 367 randomly sampled instances. This sample size corresponds to a confidence level of 95% with a confidence interval of 5%. When selecting the sample instances for manual inspection, we applied the stratified sampling technique [19] to ensure the representative samples are selected and studied from each project. Using the stratified sampling approach, the portion of sampled logging code changes from each project is equal to the relative weight of the total number of independently changed logging code changes. For example, among the total $8,325$

Category	Rationale	Co-changed	Independently
What-to-log	Adding More Information (AMI)	64	24
	CLArification (CLA)	84	25
	Fixing Language Problems (FLP)	14	38
	Avoid Excessive Logging (AEL)	38	15
How-to-log	Checking Nullable Variables(CNV)	0	1
	Removing Object Casting (ROC)	0	1
	Refactoring Logging Code (RLC)	0	1
	Changing Output Format (COF)	0	1
	Updating Logging Style (ULS)	82	255
	Adding Logging Guards (ALG)	114	0
	Mess-up Commits in VCS (MCV)	0	6
Others	-	27	0
Total	-	423	367

Table 6: Our manual characterization results on intentions behind the fixes to LCII changes

independently changed logging code changes from the six studied projects, 913 of them are from Hadoop. Thus, 40 (a.k.a., $367 \times \frac{913}{8325}$) of the manually studied independently changed code changes were selected from Hadoop. The detailed breakdown of our characterization results is shown in Table 6.

In total, we have characterized 4 intentions and 7 intentions behind the what-to-log vs. how-to-log dimension. Among co-changed logging code changes, 27% ($\frac{114}{423} \times 100\%$) are "Add Logging Guards (ALG)", which is the most common intention. Among independently changed logging code changes, the majority of the fixes ($\frac{255}{367} \times 100\% = 70\%$) are for "Updating Logging Style (ULS)". There are also other intentions. However, since the number of these intentions is small, we just grouped them into a category called "Others". We will explain below using real-world code examples for each characterized intention.

6.1.3 Detailed Characterization of the Intentions behind the LCII changes

Here we will explain our detailed characterization of the intentions behind the LCII changes from the following three categories: what-to-log, how-to-log, and others.

Figure 8 shows the real world code examples for each intention for the fixes of the LCII changes behind the what-to-log category:

– *Adding More Information (AMI)*: extra information is added to the logging code for the purpose of providing additional contextual information. Both co-changed and independently changed logging code changes can have this intention. Figure 8(a) shows an example for co-changed logging code changes. A method `getClientIdAuditPrefix` is added in the newer version in order to provide additional runtime context [7]: "this (client IP/-port of the balancer invoker) is a critical piece of admin functionality, we should log the IP for it at least ...". An independently changed logging code

Intention	Change Type	Example
AMI	Co	`LOG.debug("Submitting snapshot request for:" + ClientSnapshotDescriptionUtils.toString(request.getSnapshot()));` `String getClientIdAuditPrefix() { return "Client=" + RequestContext.getRequestUserName() + "/" + RequestContext.get().getRemoteAddress(); } LOG.info(getClientIdAuditPrefix() + " snapshot request for:" + ClientSnapshotDescriptionUtils.toString(request.getSnapshot()));}` (a) HMaster.java *in HBase*
	Ind	`LOG.info("SplitLogWorker starting");` `LOG.info("SplitLogWorker " + this.serverName + " starting");` (b) SplitLogWorker.java *in HBase*
CLA	Co	`int numEdits = new FSEditLogLoader(namesystem).loadFSEdits(new EditLogFileInputStream(editFile)); System.out.println("Number of edits: " + numEdits);` `int numEditsThisLog = loader.loadFSEdits(new EditLogFileInputStream(editFile), startTxId); System.out.println("Number of edits: " + numEditsThisLog);` (c) TestEditLogRace.java *in Hadoop*
	Ind	`LOG.error("Can't make a speculator -- check " + AMConstants.SPECULATOR_CLASS + " " + ex);` `LOG.error("Can't make a speculator -- check " + MRJobConfig.MR_AM_JOB_SPECULATOR, ex);` (d) MRAppMaster.java *in Hadoop*
FLP	Co	`private static final String AUTHZ_SUCCESSFULL_FOR = "Authorization successfull for "; AUDITLOG.info(AUTHZ_SUCCESSFULL_FOR + user + " for protocol="+protocol);` `private static final String AUTHZ_SUCCESSFUL_FOR = "Authorization successful for "; AUDITLOG.info(AUTHZ_SUCCESSFUL_FOR + user + " for protocol="+protocol);` (e) ServiceAuthorizationManager.java *in Hadoop*
	Ind	`MetricsUtil.LOG.error("Unexpected attrubute suffix");` `MetricsUtil.LOG.error("Unexpected attribute suffix");` (f) MetricsDynamicMBeanBase.java *in Hadoop*
AEL	Co	`LOG.error("Couldn't verify if the referenced file still exists, keep it just in case")` `if (LOG.isDebugEnabled()) { LOG.debug("Couldn't verify if the referenced file still exists, keep it just in case: " + hfilePath)` (g) HFileLinkCleaner.java *in HBase*
	Ind	`LOG.info("Attempt #" + numAttempt + "/" + numTries + " failed all ops, trying resubmit," + " tableName=" + tableName + ", location=" + location);` `LOG.info("Attempt #" + numAttempt + "/" + numTries + " failed all ops, trying resubmit," + location);` (h) AsyncProcess.java *in HBase*

Fig. 8: The intentions behind various fixes to the LCII changes, which are related to **what-to-log**. For each intention, we have included real-world code examples.

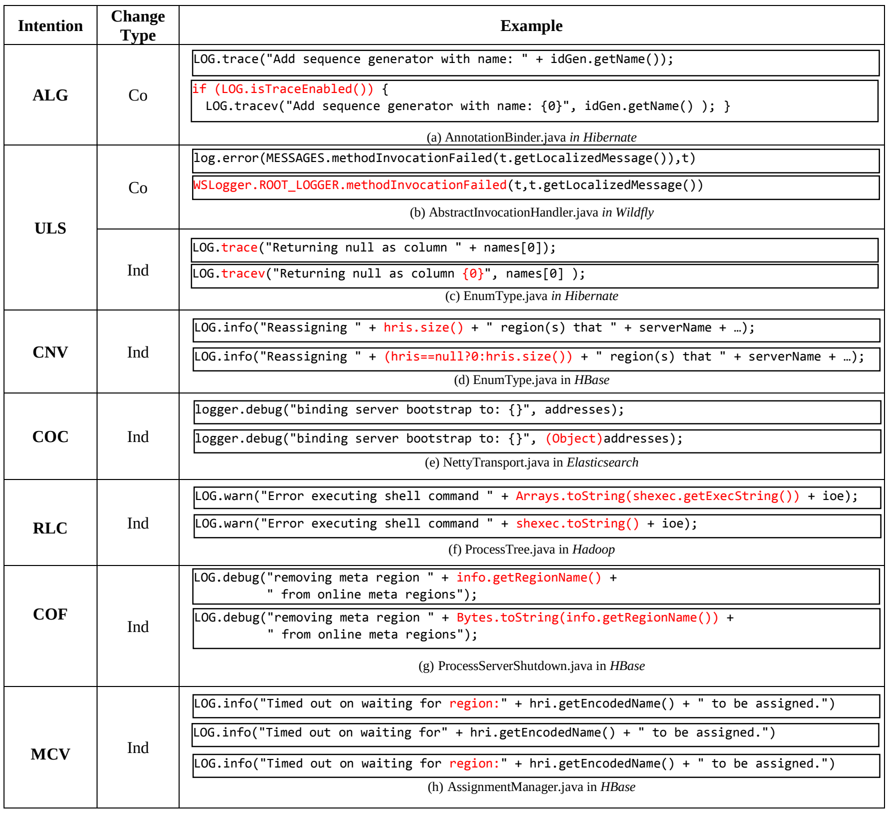
changes, as shown in Figure 8(b), `this.serverName` was added to provide more information about the SplitLogWorker.

- *CLArification (CLA)*: to clarify the runtime context, some of the outdated or imprecise dynamic information (e.g., local variables or class attributes) is fixed. Both co-changed and independently changed logging code changes can have this intention. Figure 8(c) shows an example of the co-changed logging code changes. The variable `numEdits` is updated to `numEditsThisLog` to output a more accurate number of edits required for this logging context [8]. Figure 8(d) shows an example for the independently changed logging code change. The variable `AMConstants.SPECULATOR_CLASS` was replaced by `MRJobConfig.MR_AM_JOB_SPECULATOR` to better reflect the reported error.

– *Fixing Language Problems (FLP)*: the logging code is updated to fix typos or grammar mistakes in texts or dynamic information (e.g., class attributes, local variables, method names, and class names). Both co-changed and independently changed logging code changes can have this intention. Figure 8(e) shows an example for the co-changed logging code changes. The class attribute `AUTHZ_SUCCESSFULL_FOR` was misspelled, as reported in [3], and was corrected in the next revision. Figure 8(f) shows an example for the independently changed logging code changes. The word `attrubute` in the static texts was misspelled and corrected to `attribute` in the next version.
– *Avoid Excessive Logging (AEL)*: the logging code is updated to avoid excessive logging to reduce the runtime overhead. Both co-changed and independently changed logging code changes can have this intention. Figure 8(f) shows an example for the co-changed logging code changes. The verbosity level of the logging code was changed from `error` to `debug` and the logging guard was also added. This was to reduce the amount of logging to lessen the runtime overhead and the effort to analyze the log files [5]. Figure 8(h) shows an example for the independently changed logging code changes. The variable and the text `tableName` were deleted since the variable `location` already contained this information.

Figure 9 shows the real world code examples for each intention for the fixes of the LCII changes behind the how-to-log category:

– *Adding Log Guards (ALG)*: the condition statements are added to ensure that the logs are only generated for some scenarios. Only co-changed logging code changes have this intention. Figure 9(a) shows one such example. The log guard `LOG.isTraceEnabled()` is added to ensure the corresponding generated logs got printed when the trace level logging is enabled in the configuration settings [12]. Such logging code changes are considered as fixes to the LCII changes, since these changes can improve software performance by preventing unnecessary creation of strings.
– *Updating Logging Style (ULS)*: the logging code is updated due to the changes in the logging library/method/API. Both co-changed and independently changed logging code changes can have this intention. Figure 9(b) shows an example for the co-changed logging code changes. The new version of the logging code uses log-specific API (`methodInvocationFailed`) to generate logs as explained in the pull request [13] of the Wildfly project. This style is commonly used in some of the third-party logging libraries like AspectJ [15]. Figure 9(c) shows an example for the independently changed logging code changes. The invoked method was changed from `trace` to `tracev`. The latter method requires the parameter to be formatted as `{0}` instead of string concatenation.



Intention	Change Type	Example

ALG	Co	`LOG.trace("Add sequence generator with name: " + idGen.getName());`
		`if (LOG.isTraceEnabled()) {`
		`  LOG.tracev("Add sequence generator with name: {0}", idGen.getName() ); }`
		(a) AnnotationBinder.java *in Hibernate*

ULS	Co	`log.error(MESSAGES.methodInvocationFailed(t.getLocalizedMessage()),t)`
		`WSLogger.ROOT_LOGGER.methodInvocationFailed(t,t.getLocalizedMessage())`
		(b) AbstractInvocationHandler.java *in Wildfly*

	Ind	`LOG.trace("Returning null as column " + names[0]);`
		`LOG.tracev("Returning null as column {0}", names[0] );`
		(c) EnumType.java *in Hibernate*

CNV	Ind	`LOG.info("Reassigning " + hris.size() + " region(s) that " + serverName + …);`
		`LOG.info("Reassigning " + (hris==null?0:hris.size()) + " region(s) that " + serverName + …);`
		(d) EnumType.java in *HBase*

COC	Ind	`logger.debug("binding server bootstrap to: {}", addresses);`
		`logger.debug("binding server bootstrap to: {}", (Object)addresses);`
		(e) NettyTransport.java in *Elasticsearch*

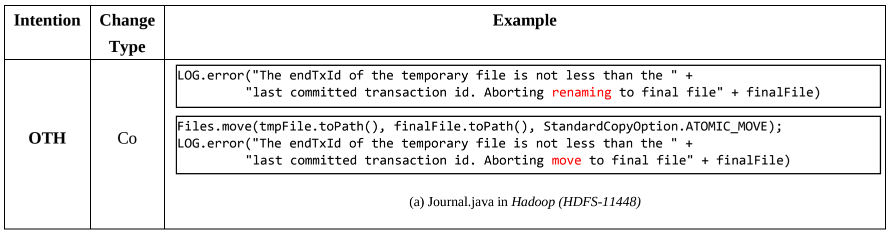
RLC	Ind	`LOG.warn("Error executing shell command " + Arrays.toString(shexec.getExecString()) + ioe);`
		`LOG.warn("Error executing shell command " + shexec.toString() + ioe);`
		(f) ProcessTree.java in *Hadoop*

COF	Ind	`LOG.debug("removing meta region " + info.getRegionName() +`
		`        " from online meta regions");`
		`LOG.debug("removing meta region " + Bytes.toString(info.getRegionName()) +`
		`        " from online meta regions");`
		(g) ProcessServerShutdown.java in *HBase*

MCV	Ind	`LOG.info("Timed out on waiting for region:" + hri.getEncodedName() + " to be assigned.")`
		`LOG.info("Timed out on waiting for" + hri.getEncodedName() + " to be assigned.")`
		`LOG.info("Timed out on waiting for region:" + hri.getEncodedName() + " to be assigned.")`
		(h) AssignmentManager.java in *HBase*

Fig. 9: The intentions behind various fixes to the LCII changes, which are related to **how-to-log**. For each intention, we have included real-world code examples.

– *Checking Null Variable (CNV)*: the logging code is updated to check if the variable is null to prevent runtime failure. Only independently changed logging code changes have this intention. Figure 9(d) shows one such example. The null check of the variable `hris` was added to avoid the null pointer exception.
– *Changing Object Cast (COC)*:
the logging code is updated to change the object cast of a variable. Only independently changed logging code changes have this intention. Figure 9(e) shows one such example. An explicit cast was added to the variable `addresses` to improve the formatting of the output string.
– *Refactoring Logging Code (RLC)*: the logging code is updated for refactoring purposes. Only independently changed logging code changes have this intention. Figure 9(f) shows one such example. For the variable `shexec`, both `Arrays.toString(shexec.getExecString())` and `shexec.toString()`



Intention	Change Type	Example

OTH	Co	`LOG.error("The endTxId of the temporary file is not less than the " +`
		`"last committed transaction id. Aborting renaming to final file" + finalFile)`
		
		`Files.move(tmpFile.toPath(), finalFile.toPath(), StandardCopyOption.ATOMIC_MOVE);`
		`LOG.error("The endTxId of the temporary file is not less than the " +`
		`"last committed transaction id. Aborting move to final file" + finalFile)`
		
		(a) Journal.java in *Hadoop (HDFS-11448)*

Fig. 10: An example of fixes to LCII changes due to other reasons.

would output the same string. It is better to pick the latter one to improve readability and maintainability.

– *Changing Output Format (COF)*: the logging code is updated to correct the malformed output. Only independently changed logging code changes have this intention. Figure 9(g) shows one such example. The type of return value of `info.getRegionName()` is `Byte`. If it is outputted directly, it will not be human readable. Hence, it was wrapped by the method `Bytes.toString()` to improve the readability.
– *Mess-up Commits in VCS (MCV)*: the logging code is updated due to mess-up commits in version control systems (VCS). Only independently changed logging code changes have this intention. Figure 9(h) shows one such example. The static text `region` was deleted at first and then added back. This is because developers first merged commits from another branch and later reverted the change.

Some fixes to the LCII changes are due to other reasons. Figure 10 shows one such example. The system functionality has been changed from renaming to moving as stated in the bug report [9]. Thus, the logging texts are updated as well. However, the numbers for these intentions are very small. Hence, we grouped these together into one category (“Others”).

6.1.4 Breakdown for fixes to LCII changes per project

We further break down the characterization of the fixes to the LCII changes for each of the studied project. The results are shown in Table 7.

In general, ULS is the biggest group in three out of six projects (Hibernate, Geronimo, Wildfly). CLA is the biggest group in Elasticsearch (27%) and HBase (24%). In Hadoop, the biggest group is ALG. When we look among the intentions in the what-to-log category, CLA is the biggest category in five out of the six studies projects and AMI is the second biggest. This shows that developers tend to clarify or add additional information when they fix logging code issues related to what-to-log. In how-to-log, ULS is the biggest category in four out of six projects. The second biggest intention is ALG. There are less than 1% MCV instances due to the merging issue in the version control systems.

Category	Rationale	Elasticsearch	Hadoop	HBase	Hibernate	Geronimo	Wildfly
What-to-log	AMI	8	29	42	0	8	1
	CLA	15	30	44	9	1	10
	FLP	10	14	19	2	1	6
	AEL	12	15	24	0	1	1
How-to-log	CNV	0	0	1	0	0	0
	ROC	1	0	0	0	0	0
	RLC	0	1	0	0	0	0
	COF	0	0	1	0	0	0
	ULS	10	23	13	77	19	195
	ALG	0	38	17	57	2	0
	MCV	0	2	2	1	0	1
Others	-	0	5	22	0	0	0
Total	-	56	157	185	146	32	214

Table 7: Our manual characterization results on the intentions behind the fixes to the LCII changes

6.1.5 Summary

Findings: Contrary to our previous study [20], both co-changed and independently changed logging code changes can contain fixes to the LCII changes. The majority of the fixes to the LCII changes are from the independently changed logging code changes. In total, there are ten different intentions behind the fixes to the LCII changes: four are related to what-to-log, and seven are related to how-to-log. Among them, "Clarification" and "Updating Logging Style" are the top two intentions.
Implications: Our current approach to characterizing the fixes to the LCII changes is done manually. This is very time consuming and prevents us from studying larger datasets. Advanced text analysis approaches like topic modeling or natural language processing can be helpful to automatically cluster related code fixes and provide summarizations.

6.2 RQ2: Are the fixes to the LCII changes more complex than other logging code changes?

In the previous RQ, we have characterized the intentions behind different fixes to the LCII changes. In this RQ, we intend to compare the complexity between the fixes to the LCII changes and other logging code changes. Since there are no existing metrics available to quantify the logging code change complexity, we have defined a complexity metric (the number of changed components of the logging code) to characterize how complex a logging code change is. Each logging code snippet contains four components: the logging library, the verbosity level, statics texts, and dynamic invocations. A logging code change is more complex, if there are more types of components changed together. The most complex logging code change is that all the components of a logging code snippet have been changed. On the other hand, some logging code changes only involve component position changes without actual content changes. For example, the following logging code snippet `log.info("text" + var);` could be changed to `log.info("text", var);`, since none of the components' contents have been changed. For such changes, the value of the complexity metric is "0".

6.2.1 Data Extraction

For each logging code change, we tracked whether any logging components have been changed using the heuristics similar to our previous works [19,20] and calculated their change complexity metrics defined above.

6.2.2 Data Analysis

Table 8 shows the breakdown of different components for the fixes to the LCII changes and other logging code changes. Generally, there are large differences in three out of four components. 27.5% of the fixes to LCII changes contain logging library changes, while only 5.1% of the other logging code changes contain logging library changes. The portion of verbosity level changes in the fixes to the LCII changes and other logging code changes are 46.4% versus 8.2%. While the fixes to LCII changes contain more logging library changes and verbosity level changes, only 14.4% of them contain dynamic content changes. On the other hand, 78.6% of other logging code changes contain dynamic content changes. For static text changes, fixes to the LCII changes and other logging code changes are similar (42.9% versus 47.1%). For fixes to the LCII changes, the verbosity level changes rank first while dynamic content changes rank first in other logging code changes. Static text changes are ranked second in both types of logging code changes.

Table 9 shows the complexity of the logging code changes for each project. The majority of logging code changes are "1 type" changes (72.9% in the fixes to the LCII changes and 66.8% in other logging code changes). The second largest category is "2 types" logging code changes, 21.9% and 27.9% for the fixes to the LCII changes and other logging code changes, respectively. For "3 types" and "4 types" logging code changes, they only make up 4.8% and 5.3% of the fixes to the LCII changes and other logging code changes. There are 0.5% fixes to the LCII changes categorized as "0 type" changes. Hence, in terms of change complexity, fixes to the LCII changes and other logging code changes are similar.

Here we further examined the top two most types of most frequently occurred complexity changes: "1 type" and "2 types" changes. The results for the "1 type change" are shown in Table 10. For each component, there are two columns. The column on the left shows the counts of co-changed logging code changes which change that component. The column on the right shows the

Table 8: Number of individual logging component changes.

Project	Lib	Level	Static	Dynamic	Total
Elasticsearch_LCII	35 (4.2%)	132 (15.8%)	603 (72.0%)	236 (28.2%)	838
Elasticsearch_Other	18 (0.9%)	47 (2.4%)	933 (48.0%)	1,633 (84.0%)	1,943
Hadoop_LCII	182 (17.7%)	360 (35.0%)	479 (46.5%)	240 (23.3%)	1,030
Hadoop_Other	28 (1.7%)	53 (3.3%)	738 (45.5%)	1,253 (77.3%)	1,622
HBase_LCII	122 (8.0%)	407 (26.5%)	875 (57.1%)	461 (30.1%)	1,533
HBase_Other	11 (0.5%)	48 (2.3%)	847 (40.2%)	1,858 (88.3%)	2,105
Hibernate_LCII	742 (37.4%)	1,292 (65.1%)	1,089 (54.9%)	84 (4.2%)	1,985
Hibernate_Other	251 (39.6%)	346 (54.6%)	443 (69.9%)	296 (46.7%)	634
Geronimo_LCII	123 (22.9%)	291 (54.3%)	141 (26.3%)	63 (11.8%)	536
Geronimo_Other	18 (3.7%)	24 (5.0%)	269 (55.7%)	293 (60.7%)	483
Wildfly_LCII	1,206 (42.7%)	1,575 (55.7%)	565 (20.0%)	177 (6.3%)	2,826
Wildfly_Other	48 (8.3%)	87 (15.1%)	236 (41.0%)	453 (78.8%)	575
Total_LCII	2,410 (27.5%)	4,057 (46.4%)	3,752 (42.9%)	1,261 (14.4%)	8,748
Total_Other	374 (5.1%)	605 (8.2%)	3,466 (47.1%)	5,786 (78.6%)	7,362

Table 9: The complexity of logging code changes grouped by each project.

Project	0 type	1 type	2 types	3 types	4 types	Total
Elasticsearch_LCII	1 (0.1%)	668 (79.7%)	169 (20.2%)	0 (0.0%)	0 (0.0%)	838
Elasticsearch_Other	0 (0.0%)	1,287 (66.2%)	625 (32.2%)	30 (1.5%)	1 (0.1%)	1,943
Hadoop_LCII	9 (0.9%)	799 (77.6%)	205 (19.9%)	16 (1.6%)	1 (0.1%)	1,030
Hadoop_Other	0 (0.0%)	1,209 (74.5%)	381 (23.5%)	27 (1.7%)	5 (0.3%)	1,622
HBase_LCII	1 (0.1%)	1,217 (79.4%)	298 (19.4%)	16 (1.0%)	1 (0.1%)	1,533
HBase_Other	0 (0.0%)	1,477 (70.2%)	598 (28.4%)	29 (1.4%)	1 (0.0%)	2,105
Hibernate_LCII	3 (0.2%)	952 (48.0%)	838 (42.2%)	189 (9.5%)	3 (0.2%)	1,985
Hibernate_Other	0 (0.0%)	181 (28.5%)	219 (34.5%)	219 (34.5%)	15 (2.4%)	634
Geronimo_LCII	15 (2.8%)	431 (80.4%)	84 (15.7%)	5 (0.9%)	1 (0.2%)	536
Geronimo_Other	0 (0.0%)	372 (77.0%)	104 (21.5%)	4 (0.8%)	3 (0.6%)	483
Wildfly_LCII	13 (0.5%)	2,307 (81.6%)	318 (11.3%)	172 (6.1%)	16 (0.6%)	2,826
Wildfly_Other	0 (0.0%)	393 (68.3%)	125 (21.7%)	47 (8.2%)	10 (1.7%)	575
Total_LCII	42 (0.5%)	6,374 (72.9%)	1,912 (21.9%)	398 (4.5%)	22 (0.3%)	8,748
Total_Other	0 (0.0%)	4,919 (66.8%)	2,052 (27.9%)	356 (4.8%)	35 (0.5%)	7,362

counts of independently changed logging code changes which change that component. For example, there are 58 "1 type"co-changed logging code changes which changes static texts and 1661 for independently changed logging code changes. Among all components, changes related to verbosity levels are the most, accounting for 36.8%. None of the co-changed logging code changes are related to logging library and verbosity level, which is a big difference from indpendently changed logging code changes. The detail results for the "2 types" changes are shown in Table 11. Changes related to static texts and verbosity levels are the two most common changes. 84.5% of the changes modify texts and 68.1% of the changes modify verbosity levels. The independently changed logging code and co-changed logging code changes have the similar distribution.

Table 10: Component changes for the “1 type” changes

Project	Lib		Level		Text		Dynamic		Total
	Co	Ind	Co	Ind	Co	Ind	Co	Ind	
Elasticsearch	0	30	0	103	2	441	8	84	668
Hadoop	0	120	0	261	42	273	23	80	799
HBase	0	69	0	312	10	601	50	175	1,217
Hibernate	0	530	0	297	1	70	4	50	952
Geronimo	0	71	0	231	2	97	5	25	431
Wildfly	0	899	0	1,144	1	179	43	41	2,307
Total	0	1719(27.0%)	0	2,348(38.0%)	58(0.9%)	1,661(26.1%)	133(2.1%)	454(7.1%)	6,374

Table 11: Component changes for the “2 types” changes

Project	Lib		Level		Static		Dynamic		Total
	Co	Ind	Co	Ind	Co	Ind	Co	Ind	
Elasticsearch	0	5	0	29	9	151	9	135	169
Hadoop	0	55	18	65	38	109	28	97	205
HBase	0	51	14	65	44	203	30	189	298
Hibernate	0	33	55	748	54	772	1	13	838
Geronimo	0	49	0	54	0	36	0	29	84
Wildfly	11	124	2	251	14	185	15	34	318
Total	11(0.6%)	317(16.6%)	89(4.7%)	1,212(63.4%)	159(8.3%)	1,456(76.2%)	83(4.3%)	497(26.0%)	1,912

Findings: the majority of the fixes to the LCII changes are changing the verbosity level, while the majority of the other logging code changes are changing dynamic contents. The two types of logging code changes (LCII changes and co-evolving logging code changes) are similar in terms of change complexity. The majority of both types of logging code changes only have one or two component changes (a.k.a., “1 type” and “2 types” logging code changes). Among the “1 type” logging code changes, the most common changes are about verbosity level changes, followed by the logging library and static text changes tied at the second place. Among the “2 types” logging code changes, the top two most common changes are changes in the verbosity level and static texts.
Implications: correcting logging code issues just needs to change one or two components in most of the cases with particular focuses on the verbosity levels and the static texts. Although there are existing automated approaches (e.g., [20,51]) to helping developers determine the appropriate verbosity levels for each logging code snippet, there are no techniques focusing on detecting and improving the static texts component.

Project	**# of LCII fixes**	**# of bug reports**
Elasticsearch	838	N/A
Hadoop	1,030	31,420
HBase	1,533	15,486
Hibernate	1,985	9,009
Geronimo	536	5,594
Wildfly	2,826	14,469

Table 12: The number of fixes to the LCII changes and the number of bug reports in each project.

6.3 RQ3: How long does it take to fix an LCII change?

In this RQ, we will measure how long it takes to resolve a LCII change. We are going to compare the resolution time between LCII changes and reported bugs.

6.3.1 Data Extraction

We first extracted the bug creation date and the bug resolution date from the bug reports in order to compute the bug resolution time. To compute the resolution time for each LCII change, we computed the time differences between the LCII changes and their fixes.

6.3.2 Data Analysis

Table 12 shows the numbers of fixes to LCII changes and regular bug reports for each studied project. There are much more reported bugs than fixes to LCII changes. For example, in Hadoop, there are $1,030$ fixes to LCII changes while there are $31,420$ bug reports (almost 30 times more). Elasticsearch does not have an issue tracking system, so we only show the number of fixes to LCII changes.

Table 13 shows the median resolution time for both LCII and regular bugs in each project. For example, in Hadoop, the median resolution time for LCII changes is 210.9 days and it is 15.5 days for regular bugs. For all five studied projects, the median resolution time of the LCII changes is longer than that of regular bugs. In particular, the resolution time can be as long as 379 days for the fixes to the LCII changes (Wildfly) while the longest median resolution time for regular bugs is only 30.9 days (Hibernate).

Figure 11 visually compares the distribution of the resolution time for the fixes to the LCII changes, and the duration for fixing regular bugs. Each plot is a beanplot [30]. The left part of the plot shows the distributions of the durations for fixing the LCII changes, while the right part of the plot shows the distribution of the bug resolution time. The vertical scale is shown under the unit of the natural logarithm of days. Among the five studied projects, the left part of the plots are always higher (a.k.a., containing higher extreme points) than the right part.

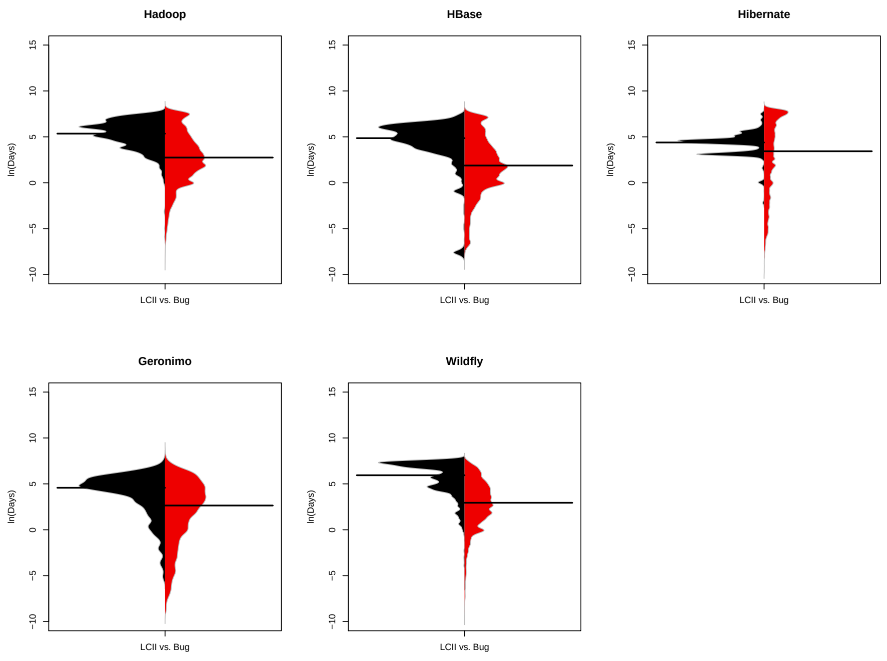


Fig. 11: Comparing the resolution time of LCII and regular bugs.

To statistically compare the resolution time for LCII changes and regular bugs across all five projects, we performed the non-parametric Wilcoxon rank-sum test (WRS). Table 13 shows our results. The two-sided WRS test shows that the resolution time of LCII changes is significantly different from that of regular bugs ($p < 0.001$) in all five projects. To assess the magnitude of the differences between the resolution time for LCIIs and regular bugs, we have also calculated the effect sizes using Cliff's Delta for all five projects in Table 13. The strength of the effects and the corresponding range of Cliff's Delta (d) values [40] are defined as follows:

$$\text{effect size} = \begin{cases} \text{negligible} & \text{if } |d| \leq 0.147 \\ \text{small} & \text{if } 0.147 < |d| \leq 0.33 \\ \text{medium} & \text{if } 0.33 < |d| \leq 0.474 \\ \text{large} & \text{if } 0.474 < |d| \end{cases}$$

Our results show that the effect sizes for three out of five projects (Hadoop, HBase, and Wildfly) are large, whereas the other projects have negligible (Hibernate) or small (Geronimo) effect sizes. However, the actual rationales be-

Table 13: Comparing the resolution time between the LCII changes and regular bugs. The unit of resolution time for both the LCII changes and regular bugs are in days.

Project	**LCII**	**Bugs**	**p-values for WRS**	**Cliff's Delta(d)**
Hadoop	210.9	15.5	<0.001	-0.54 (large)
HBase	128.6	6.5	<0.001	-0.5 (large)
Hibernate	80.4	30.9	<0.001	-0.13 (negligible)
Geronimo	96.9	14.0	<0.001	-0.33 (small)
Wildfly	379.2	18.9	<0.001	-0.63 (large)

Table 14: Comparing the resolution time between the co-changed LCII changes and independently changed LCII changes. The unit of resolution time for both the co-changed and independently changed changes are in days.

Project	**Co-changed**	**Independently**	**p-values for WRS**	**Cliff's Delta(d)**
Elasticserach	103.9	84.1	0.04	-0.3 (small)
Hadoop	210.9	216.8	0.73	0.02 (negligible)
HBase	183.5	126.1	0.14	-0.08 (negligible)
Hibernate	163.2	80.3	<0.001	-0.72 (large)
Geronimo	21.3	96.9	0.27	0.22(small)
Wildfly	219.3	397.3	<0.001	0.20 (small)

hind the longer resolution time for logging code issues are not clear, as there are both high priority and low priority issues in regular bugs and logging code issues. In the future, we plan to investigate this further by surveying the developers or the domain experts of these projects.

We further compared the resolution time between the LCII changes from co-changed logging code changes and the independently changed logging code changes using the similar method as described above. The results are shown in Table 14. Each row corresponds to one projet. For example, in Hadoop, the median resolution time of fixes by co-changed LCII changes is 210.9 days and it is 216.8 days for independently changed LCII changes. For Elasticsearch, HBase and Hibernate, the median resolution time of the co-changed LCII changes is longer. The differences between the two types of LCII changes are only statistically significant in Hibernate and Wildfly ($p < 0.001$). The magnitude of differences is either small or negligible for five out of six projects.

We also studied the distribution of the resolution time with respect to the complexity of the LCII changes. The results are visualized in Figure 12. Each plot contains 2 to 5 violin plots for each project. Some plots are missing because there are not enough instances. Take Elasticsearch for example, there are no enough instances for "0 type", "3 types" and "4 types" changes. Hence, there are no corresponding plots for this project. The vertical scale for all the plots is in the unit of the natural logarithm of days. As we can see across the violin plots, the distribution differs from project to project. It demonstrates that the complexity of the fixes to the LCII changes do not necessarily impact the resolution time.

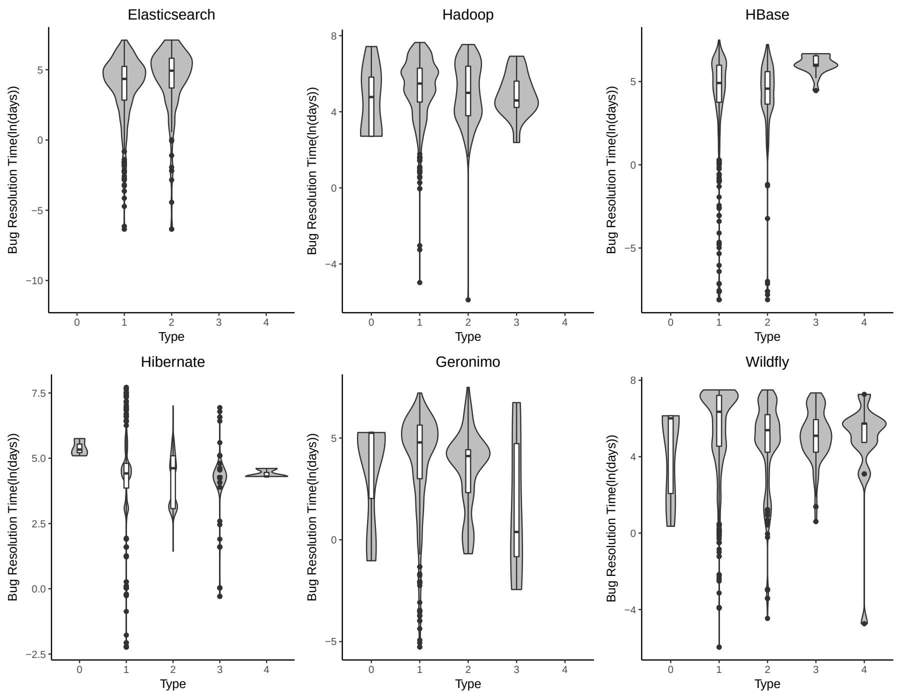


Fig. 12: Comparing the resolution time of different types of LCII changes.

6.3.3 Summary

Findings: the resolution time of the LCII changes and regular bugs are statistically different in all studied projects. The median resolution time of the LCII changes is much longer than that of regular bugs and the magnitude of differences is large in three out of the five studied projects. Within the LCII changes, there is no clear statistical difference between the types of changes (co-changed vs. independently changed) or the complexity (1 type vs. 2 types vs. 3 types vs. 4 types) of the changes.
Implications: although software logging is used widely in many contexts (e.g., debugging, runtime monitoring, business decision making, and security), the correctness of the logging code cannot be easily verified using conventional software verification techniques. The issues in the logging code changes can take much longer to be detected and fixed than regular bugs, as most of the existing issues in the logging code can only be detected and fixed manually. Simple logging code changes may take as much time as complex logging code changes. Automated approaches to validating and improving the quality of the logging code is becoming an increasingly important research area.

6.4 RQ4: Are state-of-the-art code detectors able to detect logging code with issues?

In this RQ, we want to study the effectiveness of the state-of-the-art techniques on detecting issues in the logging code. There are two techniques in the existing research literature attempting to flag issues in the logging code:

– **Rule-based Static Analysis (LCAnalyzer)**: this technique is proposed by Chen et al [20]. It scans through source code searching for six anti-pattern instances using a set of rules.
– **Code Clones (Cloning)**: this technique is proposed by Yuan et al [50]. It uses a code clone detection tool to find all the clone groups and checks if the logging code in the clone groups have the same verbosity levels. If the levels are inconsistent, at least one of them have incorrect verbosity levels.

6.4.1 Data Extraction

Our extracted dataset contains all the LCII changes and their fixes throughout the entire development history. However, both techniques listed above need to be applied on the entire source code from one release of each project, since LCAnalyzer needs to extract code dependency information and Cloning needs to scan all the source code to find code clones. Therefore, we selected bi-monthly snapshots of the studied projects and ran both techniques on them.

We extracted bi-monthly snapshots for all the studied projects. For each snapshot, we computed the number of existing logging code issues using our dataset. For each logging code issue, we kept track of the issue introduction and fixed time/version, so that we can track the number of logging code issues in each snapshot by checking its timestamp against various commits: a logging code issue exists in a snapshot if it is introduced before the release time of the snapshot and fixed after the release time of the snapshot.

To avoid repeated counts, we counted each unique logging code issue once. For example, if a piece of logging code with issue exists in two snapshots, and is detected by the above techniques, we only count it once. Across all the snapshots, we computed the total number of unique logging code issues and aggregated the total number of detected ones from the two approaches.

In order to further characterize the capability of these two detecting techniques, we classified the logging code with issues into four categories: (1) issues in the logging library, (2) issues in the verbosity level, (3) issues in the static text, and (4) issues in the dynamic contents. Note that Cloning can only detect issues in the verbosity level. We obtained the LCAnalyzer from [14] and implemented the Cloning technique by ourselves.

6.4.2 Data Analysis

The detected results of LCAnalyzer and Cloning are shown in Figure 13. We calculated the recall of the detection results as $\frac{Number\ of\ unique\ LCII\ detected}{Number\ of\ all\ unique\ LCII} \times 100\%$. In total, only 2.1% of the logging code issues have been detected by LCAnalyzer, and only 0.1% of them have been detected by Cloning technique. We then split the issues based on its problematic components. For example, 0.3% of verbosity level related issues are detected by Cloning technique while 1.5% of them are detected by LCAnalyzer.

In general, LCAnalyzer performs the best on issues related to dynamic contents, while it is only able to detect 5.1% of all issues in the total 60 snapshots. The worst performance of LCAnalyzer is 1.0% recall, when the issues are related to the logging library. On the other hand, Cloning technique can only detect verbosity level related issues due to its design, yet it is still outperformed by LCAnalyzer (0.3% vs. 1.5%).

When we compared the correctly detected instances from the two studied tools, we found that the two techniques complement each other. We noticed that all the logging code with issues found by the Cloning techniques are not detected by the LCAnalyzer, and vice versa. To demonstrate the differences of these two approaches, we show two real world examples in Figure 14. LCAnalyzer can only detect the issue in the top, whereas Cloning can only detect the issue in the bottom. The reason that LCAnalyzer can detect the logging code issue in the top is because the verbosity level and the static content are inconsistent. In the static text, it shows that the logging code is for debugging purpose while the verbosity level is `info`. In the fix version, the verbosity level is changed to `debug` and the text `DEBUG` is deleted. For the issue in the bottom, two pieces of logging code snippets (highlighted in red) are considered as clones. The verbosity level of one logging code is `debug` whereas the other one is `info`. Therefore, the `info` level should be changed to `debug`. However, even

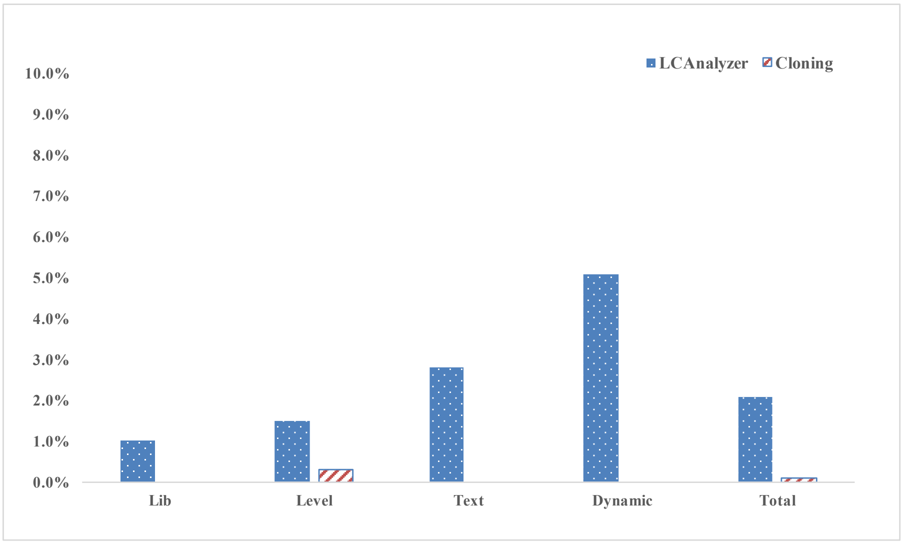
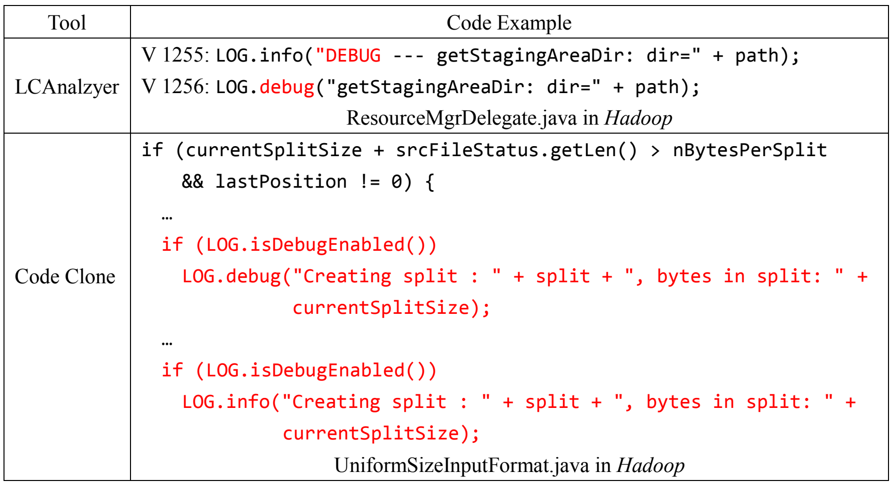


Fig. 13: Comparing the recall of the detection results for the two studies techniques.

combining the power of these two techniques, most of the issues in logging code are still undetected.



Tool	Code Example
LCAnalzyer	V 1255: `LOG.info("DEBUG --- getStagingAreaDir: dir=" + path);` V 1256: `LOG.debug("getStagingAreaDir: dir=" + path);` ResourceMgrDelegate.java in *Hadoop*

Code Clone	`if (currentSplitSize + srcFileStatus.getLen() > nBytesPerSplit`
	`    && lastPosition != 0) {`
	`  …`
	`  if (LOG.isDebugEnabled())`
	`    LOG.debug("Creating split : " + split + ", bytes in split: " +`
	`              currentSplitSize);`
	`  …`
	`  if (LOG.isDebugEnabled())`
	`    LOG.info("Creating split : " + split + ", bytes in split: " +`
	`             currentSplitSize);`
	UniformSizeInputFormat.java in *Hadoop*

Fig. 14: Examples of the detected issues in the logging code from the two studied techniques.

6.4.3 Summary

Findings: both the LCAnalyzer and the Cloning technique can only detect a small fraction ($< 3\%$) of the issues in logging code. The results outputted by the two techniques complement each other.
Implications: there are still many logging code issues, which cannot be detected by existing automated techniques. Leveraging our provided dataset, researchers can develop and benchmark their new techniques on automatically detecting issues in the logging code and deriving effective logging guidelines. As shown in RQ2, the majority of the LCII changes are related to verbosity level changes and static texts, researchers are recommended to prioritize their effort on detecting and improving the issues in these two categories.

7 Related Works

In this section, we will discuss three areas of related research: (1) empirical studies on software logging, and (2) research on automated suggestions on the logging code, and (3) research on identifying the bug introducing changes.

7.1 Empirical Studies on Software Logging

Empirical studies show that there are no well-established logging practices used in industry [26,38] as well as open source projects [19,50]. Although most of the logging code has been actively maintained, some of the existing logging code can be confusing and difficult to understand [44]. Various studies have been conducted to study the relationship between software logging and code quality. Shang et al. [43] that the amount of logging code is correlated with the amount of post-release bugs. Kabinna et al. [29] studied the migration of logging libraries and its rationales. Over 70% of the migrated systems have been found buggy afterwards. In [52], the authors analyzed the logging code changes from three open source systems (Hadoop, HBase, and ZooKeeper). They found that the majority of the logging code changes are due to adding logging statements. Many modifications to the exsting logging code are related to the verbosity level changes.

In this paper, we studied the historical logging code changes from six open source systems. We have performed three empirical studies on the commits with a focus on the issues in the logging code and their fixes. First, we compared the change characteristics between the fixes to the logging code and regular logging code changes. Second, we studied the resolution time for fixing logging code issues and compared the duration against the resolution time for bugs. Finally, we assessed the effectiveness of the state-of-the-art techniques on detecting issues in the logging code. All the above three research questions, which had never been studied before, leveraged our extracted dataset presented in this paper.

7.2 Research on Automated Suggestions on the Logging Code

Orthogonal with the feature code, the logging code is considered as a cross-cutting concern. Adding and maintaining high quality logging code is very challenging and usually requires a large amount of manual effort. Various research has been conducted in the area of automated suggestions of the logging code:

– **where-to-log** focuses on the problem of providing automated suggestions on adding logging code. Yuan et al. [49] leveraged program-analysis techniques to suggest code locations to add logging code for debugging purposes. Zhu et al. [53] learned common logging patterns from existing code and made automated suggestions based on the resulting learned models. Zhao et al. [52] introduced Log20, which can automate the placement of logging code under certain overhead threshold.
– **what-to-log** is related to the problem of adding sufficient information into the logging code. Yuan et al. [51] proposed a program analysis-based approach to suggesting adding additional variables into the existing logging code to improve the diagnosability of software systems. Li et al. [33] learned from the development history to automatically suggest the most appropriate level for each newly-added logging code.
– **how-to-log** is about maintaining high quality logging code. Li et al. [34] learned from the code level changes to predict whether the logging code is a particular code commit requiring updates (a.k.a., just-in-time logging code changes). However, their technique did not pin-point the exact logging code snippets to be updated within each code commit. Yuan et al. [50] leveraged code cloning techniques to detect inconsistent verbosity levels among similar feature code segments. Chen et al. [20] inferred six anti-patterns in the logging code by manually sampling a few hundred logging code changes.

This paper fits in the area of “how-to-log” and is close to [20]. However, the technique to extract issues in the logging code and their fixes have been improved in this paper so that the resulting dataset is more complete and more accurate. Furthermore, instead of only focusing on a small set of sampled instances, this paper provided a dataset containing all the historical issues in the logging code for six popular open source projects. In addition, we have also conducted three empirical studies on the resulting dataset to demonstrate the usefulness of this dataset and presented some open research problems in the area of “how-to-log”.

7.3 Research on Identifying the Bug Introducing Changes

It is important to identify bug introducing changes so that developers and researchers can learn from them and develop tools to detect and fix them automatically. There are generally two steps in identifying bug introducing changes:

- **Step 1 - Identifying bug fixing commits**: keyword heuristics based approaches have been used to identify bug fixing commits by searching through relevant keywords (e.g., "Fixed 42233" or "Bug 23444", etc.) in the code commit logs [18,54].
- **Step 2 - Identifying bug introducing commits based on their fixes**: Śliwerski et al. [45] were the first to develop an automatic algorithm to identify bug introducing changes based on their fixes. Their algorithm, called the SZZ algorithm, initially started at the code commits which fixes these issues, then tracks backward to the previous commits which touched those source code lines. Afterwards, there are various modifications to the SZZ algorithms [32,47,48] to improve its precision and to evaluate its effectiveness [21].

This paper is different from the above, as it aims to extract and study the LCII changes instead of the software bugs. Software bugs are usually reported to the bug tracking systems and their fixes are logged in the code commit messages. However, issues in the logging code are usually undocumented. Thus, in this paper, we have proposed an approach to first identifying fixes to the LCII changes and then automatically identifying the LCII changes based on their fixes. We have analyzed both of the co-changed and independently changed logging code changes to flag fixes to the LCII changes. Since there are multiple components (e.g., logging library, verbosity level, static texts, and dynamic contents) in each logging code snippets related to more than one line of feature code, there can be multiple issues from different code commits in one line of the logging code. Hence, we have developed an adapted SZZ algorithm (LCC-SZZ) to identify issues in the logging code changes.

8 Threats to Validity

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

8.1 Internal Validity

There are two general types of logging code changes: independently and co-changed logging code changes. We have identified the fixes to the LCII changes by carefully analyzing the contents of both types of changes. For analysis of co-changed logging code changes, we manually examined 533 instances. We inspected the source code files which contain those co-changed logging code changes, and examined the context to verify whether these are fixes to LCII changes. This manual investigation was repeated by both authors in this paper. The resulting outputs from both authors were compared and discussed to generate a reconciled dataset in the end. The resulting LCII changes identified using the LCC-SZZ algorithm have also been verified to be highly accurate (93% accuracy).

8.2 External Validity

In this paper, we extracted the LCII changes from six Java-based systems. We do feel our approach is generic and can be applied to identify LCII changes for systems implemented in other programming languages (e.g., C, C#, and Python).

We have conducted three empirical studies on the resulting dataset extracted from the development history of six Java-based projects (Elasticsearch, Hadoop, HBase, Hibernate, Geronimo, and Wildfly), which come from different domains (big data platform, web server, database, middleware, etc.). All of these projects have relatively long development history and are actively maintained. However, our findings in the research questions may not be generalizable to other programming languages.

8.3 Construct Validity

We have used ChangeDistiller [24]to extract the fine-grained code changes across different code versions. ChangeDistiller has been used in many of the previous works [19,20] and is proven to be highly accurate and very robust.

We have demonstrated that LCC-SZZ is more effective than SZZ in terms of identifying LCII changes across the following three dimensions: (1) the disagreement ratio between the extraction results from the two algorithms; (2) comparing the earliest appearance of the LCII changes against the bug reporting time; and (3) manual verification. Our evaluation approaches are similar to many of the existing studies in this area (e.g. [21,35,37,41]).

9 Conclusions and Future Work

Software logging has been used widely in large-scale software systems for a variety of purposes. It is hard to develop and maintain high quality logging code, as it is very challenging to verify the correctness of a particular logging code snippet. To aid effective maintenance of logging code, in this paper we have extracted and studied the historical issues in logging code and their fixes from six popular Java-based open source projects. To demonstrate the effectiveness of our dataset, we have conducted four preliminary case studies. We have found that both the co-changed and the independently changed logging code changes can contain fixes to the LCII changes. The change complexity metrics between the fixes to the LCII changes and other logging code changes are similar. It usually takes much longer to address an LCII change than a regular bug. Existing state-of-the-art techniques on detecting logging code issues cannot detect a majority of the issues in logging code. In the future, we plan to further leverage our derived dataset to: (1) develop better techniques to automatically detect issues in logging code, and (2) derive best practices in terms of developing and maintaining high quality logging code.

References

1. HADOOP-12666: Support Microsoft Azure Data Lake - as a file system in Hadoop. https://issues.apache.org/jira/browse/HADOOP-12666. Last accessed: 02/06/2018
2. HADOOP-7358: Improve log levels when exceptions caught in RPC handler. https://issues.apache.org/jira/browse/HADOOP-7358. Last accessed: 02/14/2018
3. HADOOP-8347: Hadoop Common logs misspell 'successful'. https://issues.apache.org/jira/browse/HADOOP-8347. Last accessed: 02/14/2018
4. HBASE-10470: Import generates huge log file while importing large amounts of data. https://issues.apache.org/jira/browse/HBASE-10470. Last accessed: 01/24/2018
5. HBASE-12539: HFileLinkCleaner logs are uselessly noisy. https://issues.apache.org/jira/browse/HBASE-12539. Last accessed: 02/14/2018
6. HBASE-750: NPE caused by StoreFileScanner.updateReaders. https://issues.apache.org/jira/browse/HBASE-750/. Last accessed: 08/26/2016
7. HBASE-8754: Log the client IP/port of the balancer invoker. https://issues.apache.org/jira/browse/HBASE-8754. Last accessed: 02/14/2018
8. HDFS-1073: Simpler model for Namenode's fs Image and edit Logs. https://issues.apache.org/jira/browse/HDFS-1073. Last accessed: 02/14/2018
9. HDFS-11448: JN log segment syncing should support HA upgrade. https://issues.apache.org/jira/browse/HDFS-11448. Last accessed: 02/14/2018
10. HDFS-4122: Cleanup HDFS logs and reduce the size of logged messages. https://issues.apache.org/jira/browse/HDFS-4122. Last accessed: 02/07/2018
11. HDFS-5800: Typo: soft-limit for hard-limit in DFSClient. https://issues.apache.org/jira/browse/HDFS-5800. Last accessed: 02/14/2018
12. HHH-6732: Some logging trace statements are missing guards against unneeded string creation. https://hibernate.atlassian.net/browse/HHH-6732. Last accessed: 02/14/2018
13. PR5906: Split all log messages into separate module project codes. https://github.com/wildfly/wildfly/pull/5906. Last accessed: 02/14/2018
14. Replication Package for the LCAnalyzer work. http://www.cse.yorku.ca/~zmjiang/share/replication_package/icse2017_chen/LCAnalyzer.zip. Last accessed: 02/14/2018
15. The AspectJ Project. https://eclipse.org/aspectj/. Last accessed: 08/26/2016
16. The replication package. http://www.cse.yorku.ca/~zmjiang/share/replication_package/emse2018_chen/replication_package.zip. Last accessed: 04/09/2018
17. Barik, T., DeLine, R., Drucker, S., Fisher, D.: The Bones of the System: A Case Study of Logging and Telemetry at Microsoft. In: Companion Proceedings of the 38th International Conference on Software Engineering) (2016)
18. Bird, C., Bachmann, A., Aune, E., Duffy, J., Bernstein, A., Filkov, V., Devanbu, P.: Fair and balanced?: Bias in bug-fix datasets. In: Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE) (2009)
19. Chen, B., Jiang, Z.M.: Characterizing logging practices in Java-based open source software projects – a replication study in Apache Software Foundation. Empirical Software Engineering (2016)
20. Chen, B., Jiang, Z.M.: Characterizing and detecting anti-patterns in the logging code. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pp. 71–81 (2017)
21. da Costa, D.A., McIntosh, S., Shang, W., Kulesza, U., Coelho, R., Hassan, A.: A framework for evaluating the results of the szz approach for identifying bug-introducing changes. IEEE Transactions on Software Engineering **43**(7), 641–657 (2017). DOI 10.1109/TSE.2016.2616306
22. Davies, S., Roper, M., Wood, M.: Comparing text-based and dependence-based approaches for determining the origins of bugs. Journal of Software: Evolution and Process **26**(1), 107–139 (2014)
23. Ding, R., Zhou, H., Lou, J.G., Zhang, H., Lin, Q., Fu, Q., Zhang, D., Xie, T.: Log2: A Cost-aware Logging Mechanism for Performance Diagnosis. In: Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (ATC) (2015)

24. Fluri, B., Wursch, M., Pinzger, M., Gall, H.: Change distilling:tree differencing for fine-grained source code change extraction. Software Engineering, IEEE Transactions on **33**(11), 725–743 (2007)
25. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Longman Publishing Co., Inc. (1999)
26. Fu, Q., Zhu, J., Hu, W., Lou, J.G., Ding, R., Lin, Q., Zhang, D., Xie, T.: Where Do Developers Log? An Empirical Study on Logging Practices in Industry. In: Companion Proceedings of the 36th International Conference on Software Engineering (2014)
27. Humble, J., Farley, D.: Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation. Addison-Wesley Professional (2010)
28. Jiang, Z.M., Hassan, A.E., Hamann, G., Flora, P.: Automated performance analysis of load tests. In: Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM) (2009)
29. Kabinna, S., Bezemer, C.P., Shang, W., Hassan, A.E.: Logging Library Migrations: A Case Study for the Apache Software Foundation Projects. In: Proceedings of the 13th International Conference on Mining Software Repositories (MSR) (2016)
30. Kampstra, P.: Beanplot: A boxplot alternative for visual comparison of distributions. Journal of Statistical Software, Code Snippets **28**(1) (2008)
31. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming (1997)
32. Kim, S., Zimmermann, T., Pan, K., Whitehead, E.J.J.: Automatic identification of bug-introducing changes. In: 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06) (2006)
33. Li, H., Shang, W., Hassan, A.E.: Which log level should developers choose for a new logging statement? Empirical Software Engineering **22**(4) (2017)
34. Li, H., Shang, W., Zou, Y., Hassan, A.E.: Towards just-in-time suggestions for log changes. Empirical Software Engineering **22**(4), 1831–1865 (2017)
35. Moha, N., Gueheneuc, Y.G., Duchien, L., Meur, A.F.L.: DECOR: A Method for the Specification and Detection of Code and Design Smells. IEEE Transactions on Software Engineering (TSE) (2010)
36. Oliner, A., Ganapathi, A., Xu, W.: Advances and challenges in log analysis. Commun. ACM **55**(2), 55–61 (2012)
37. Palomba, F., Bavota, G., Penta, M.D., Oliveto, R., Poshyvanyk, D., Lucia, A.D.: Mining Version Histories for Detecting Code Smells. IEEE Transactions on Software Engineering (TSE) (2015)
38. Pecchia, A., Cinque, M., Carrozza, G., Cotroneo, D.: Industry Practices and Event Logging: Assessment of a Critical Software Development Process. In: Companion Proceedings of the 37th International Conference on Software Engineering (2015)
39. Rigby, P.C., German, D.M., Storey, M.A.: Open source software peer review practices: A case study of the apache server. In: Proceedings of the 30th International Conference on Software Engineering (ICSE) (2008)
40. Romano, J., Kromrey, J.D., Coraggio, J., Skowronek, J.: Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys? In: Annual meeting of the Florida Association of Institutional Research (2006)
41. Sajnani, H., Saini, V., Svajlenko, J., Roy, C.K., Lopes, C.V.: SourcererCC: Scaling Code Clone Detection to Big-code. In: Proceedings of the 38th International Conference on Software Engineering (ICSE) (2016)
42. Shang, W., Jiang, Z.M., Adams, B., Hassan, A.E., Godfrey, M.W., Nasser, M., Flora, P.: An exploratory study of the evolution of communicated information about the execution of large software systems. Journal of Software: Evolution and Process **26**(1), 3–26 (2014)
43. Shang, W., Nagappan, M., Hassan, A.E.: Studying the relationship between logging characteristics and the code quality of platform software. Empirical Software Engineering **20**(1) (2015)
44. Shang, W., Nagappan, M., Hassan, A.E., Jiang, Z.M.: Understanding Log Lines Using Development Knowledge. In: Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME) (2014)

45. Śliwerski, J., Zimmermann, T., Zeller, A.: When do changes induce fixes? In: Proceedings of the 2005 International Workshop on Mining Software Repositories (MSR) (2005)
46. Śliwerski, J., Zimmermann, T., Zeller, A.: When do changes induce fixes? In: Proceedings of the 2005 International Workshop on Mining Software Repositories (2005)
47. Williams, C., Spacco, J.: Szz revisited: Verifying when changes induce fixes. In: Proceedings of the 2008 Workshop on Defects in Large Software Systems, DEFECTS '08, pp. 32–36. New York, NY, USA (2008)
48. Williams, C.C., Spacco, J.W.: Branching and merging in the repository. In: Proceedings of the 2008 International Working Conference on Mining Software Repositories, MSR '08, pp. 19–22. New York, NY, USA (2008)
49. Yuan, D., Park, S., Huang, P., Liu, Y., Lee, M.M., Tang, X., Zhou, Y., Savage, S.: Be Conservative: Enhancing Failure Diagnosis with Proactive Logging. In: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI) (2012)
50. Yuan, D., Park, S., Zhou, Y.: Characterizing logging practices in open-source software. In: Proceedings of the 34th International Conference on Software Engineering, ICSE '12, pp. 102–112. IEEE Press, Piscataway, NJ, USA (2012)
51. Yuan, D., Zheng, J., Park, S., Zhou, Y., Savage, S.: Improving software diagnosability via log enhancement. In: Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) (2011)
52. Zhao, X., Rodrigues, K., Luo, Y., Stumm, M., Yuan, D., Zhou, Y.: Log20: Fully Automated Optimal Placement of Log Printing Statements Under Specified Overhead Threshold. In: Proceedings of the 26th Symposium on Operating Systems Principles (SOSP) (2017)
53. Zhu, J., He, P., Fu, Q., Zhang, H., Lyu, M.R., Zhang, D.: Learning to log: Helping developers make informed logging decisions. In: Proceedings of the 37th International Conference on Software Engineering (2015)
54. Zimmermann, T., Premraj, R., Zeller, A.: Predicting defects for eclipse. In: Proceedings of the Third International Workshop on Predictor Models in Software Engineering (PROMISE) (2007)