Characterizing Logging Practices in Open-Source Software

Ding Yuan‡†, Soyeon Park†, and Yuanyuan Zhou†
†*University of California, San Diego,* ‡*University of Illinois at Urbana-Champaign*
{*diyuan,soyeon,yyzhou*}*@cs.ucsd.edu*

***Abstract*—Software logging is a conventional programming practice. While its efficacy is often important for users and developers to understand what have happened in the production run, yet software logging is often done in an arbitrary manner. So far, there have been little study for understanding logging practices in real world software. This paper makes the first attempt (to the best of our knowledge) to provide a quantitative characteristic study of the current log messages within four pieces of large open-source software. First, we quantitatively show that software logging is pervasive. By examining developers' own modifications to the logging code in the revision history, we find that they often do not make the log messages right in their first attempts, and thus need to spend a significant amount of efforts to modify the log messages as *after-thoughts*. Our study further provides several interesting findings on where developers spend most of their efforts in modifying the log messages, which can give insights for programmers, tool developers, and language and compiler designers to improve the current logging practice. To demonstrate the benefit of our study, we built a simple checker based on one of our findings and effectively detected 138 pieces of *new* problematic logging code from studied software (24 of them are already confirmed and fixed by developers).**

***Keywords*-log message, log quality, empirical study, failure diagnosis**

I. Introduction

Writing software log messages is a well established programming practice to record the dynamic information during a program's execution. It is often used in failure diagnosis, auditing, profiling, etc. Figure 1 shows how developers write log messages in real-world open-source software. In a log message, developers describe the logged event using *static text* and optionally record *variable values* related to the event. Each log message also has a verbosity level. For instance, Figure 1 shows four types of different verbosity levels: `fatal` (i.e., abort a process after logging), `error` (i.e., record error events), `info` (i.e., record important but normal events), `debug` (i.e., verbose logging only for debugging). The verbosity levels for less critical events (e.g., `debug`) naturally subsume those for more critical events (e.g. `error`), meaning that all the events logged under the latter are also recorded under the former. Using verbosity level, users or developers can trade the benefit of sufficient log messages with their cost (e.g., performance overhead). Under the default production setting, open-source software typically only log error events in addition to a few (less than 10% [35]) book-keeping messages (e.g., `info`). Other verbose mode messages are typically used only during in-house testing.

```
elog (FATAL, "out of memory");                                  /*PostgreSQL*/
ap_log_error (ERR, "could not open charset \
        conversion config file %s", confname);                  /*  Apache  */
logit (INFO, "Authentication refused: %s", line);               /* OpenSSH */
elog(DEBUG1, "archived transaction log file %s", xlog); /*PostgreSQL*/
```

Figure 1. Log printing code examples from open-source software.

Logs are particularly beneficial to diagnosing failures in the production environment, which often require immediate resolutions as they directly impact the end users. Log messages printed during the production run are often the only data source available for the developers to diagnose such failures. This is because it is often challenging to reproduce production failures. First, end-users are often reluctant to release their failure-triggering input due to privacy concerns. Second, it can be forbiddingly expensive for the vendors to recreate the exactly same execution environment (e.g., the same hardware, third party library, OS, etc.) as in the production setting. Consequently, software engineers mostly rely on log messages for trouble-shooting production failures. Besides developers, logs are also used by system administrators to resolve failures caused by security attacks, hardware errors, and misconfigurations.

The importance of logging has been widely identified [14]. Consequently, practically almost all open-source software print log messages. For example, the widely used OpenSSH server contains 3407 log printing statements in its 81K lines of code, and 2241 of them are printed under the default verbosity mode.

The importance of log message is also proved by its commercial acceptance. It is an industry common practice for vendors to request logs or even have their systems to automatically sending the logs back periodically (i.e., "call-home" [6]). As a result, most modern systems today (such as those from EMC [7], NetApp [21], Cisco [5], Dell [6], just to name a few) are able to collect logs from at least 50% of their customers [7], [21], [5], [6]. Once such logs are collected, they can be either analyzed by human or by commercial log analysis tools (e.g., Splunk [26] or ArcSight [1]).

A. State of the Art of Logging Design

Despite the importance of log messages, logging is often a subjective and arbitrary practice in reality. In general, there is no rigorous specification or systematic process on software logging, partially because logging is seldom a core feature provided by the vendors. Therefore, in many cases, log messages are written as "after-thoughts" after a failure happens and logs are needed. There are a number of logging libraries, such as syslog [28] and log4j [16], that provide better logging interface (e.g., multiple verbosity levels) than general-purpose output functions (e.g., printf). But even with them, it is still the developers' arbitrary decisions on when, what and where to log. Recently, a few research work proposed to systematically improve a few aspects of software logging. LogEnhancer [36] automatically suggests which variable values need to be recorded in each existing log message. However, even with LogEnhancer, developers still need to make majority of the logging decisions.

Improving the current logging practice will significantly benefit from a deep understanding of the real-world logging characteristics. Specifically, we first need to assess whether the current practice is good enough, and if not, what are the common issues and what are their consequences. Not only it could provide programmers with useful guidelines and motivations for better logging, but also shed lights to new tools and programming language support for systematic logging, better testing of logging, logging improvement, etc.

B. Our Contributions

This paper makes the first attempt (to the best of our knowledge) to study the practice of software logging. Specifically, we try to answer the following questions: (i) how pervasive is software logging? (ii) is the current logging practice good enough? (iii) if not, how are developers modifying logs?

To answer these questions, we study the logging practice in four pieces of large, widely used open-source software, including Apache httpd, OpenSSH, PostgreSQL, and Squid, each with at least over 10 years of developing history. To understand the pervasiveness of logging, we study the density of log messages in source code and the churn rate [19] from the revision history.

Answering question (ii) and (iii) is more challenging because judging the logging efficacy requires deep domain expertise. We address this challenge by studying *developers' own modifications* to their log messages. Specifically, we systematically and automatically analyze the revision history of each log message. We further separate those log modifications that are indeed modifying log messages as afterthoughts from those merely consistency updates together with other non-logging code changes. Then we analyze each category separately, with a focus on the former.

Table I summarizes our major findings and their implications. Overall, our study makes the following contributions:

- Our work is the first (to the best of our knowledge) to provide quantitative evidences that logging is an important software development practice.
- We find that despite the importance of effective logging, unfortunately, developers often are not able to make log messages right at the first time. Therefore, many of the log messages need to be modified as *afterthoughts*. By examining developers' own modifications, we identify the particular aspects in logging choices where developers spend most efforts to get them right. Such findings can shed lights on various new tools and program language supports to improve log messages.
- To demonstrate the potential benefits of our findings, we developed a simple checker to detect problematic verbosity level assignment (inspired by our Finding 7). We detected 138 *new* problematic cases in the latest versions of the four pieces of the studied software. 24 of them have already been confirmed and fixed by the developers as a result of our bug reports. This result confirms that our findings are indeed beneficial to tool developers to systematically help programmers to improve their log messages.

While we believe that the open-source software we examined well represent the characteristics of current logging practice, we do not intend to draw any general conclusions. Our findings and implications should be taken with the examined open-source software and our methodology in mind (see our discussion about threats to validity in Section II).

II. Methodology

A. Software Used in Our study

We study four large, widely used software programs with over at least 10 years of developing history, namely Apache httpd, OpenSSH, PostgreSQL and Squid. Table II provides the descriptions. Each of them is popular as it is ranked either first or second in market share for its product's category. The lines of code (LOC) is measured using `sloccount` [25], which excludes non-functional code such as comments, white-spaces, etc.

All of the software we study include server component. We choose servers because, first, their availability and reliability are often important since they tend to be used as infrastructure software providing critical services (e.g., e-commerce services), and thus many users and other applications are depending on them.

Second, runtime logs are particularly important for diagnosing server failures, which are hard to be reproduced due to the privacy and environment setting issues. They typically run for a long period of time, handle large amounts of data from users, perform concurrent execution, and are sometimes deployed in a large distributed environment forming server farms, all of which make failure reproduction and diagnosis difficult.

Table I
OUR MAJOR FINDINGS ON REAL WORLD LOGGING CHARACTERISTICS AND THEIR IMPLICATIONS.

Density of software logging (Section III-A)	**Implications**
(1) On average, every 30 lines of code contains one line of logging code. Similar density is observed in all the software we studied.	Logging is pervasive during software development.
Benefit of log messages in failure diagnosis (Section III-B)	**Implications**
(2) Log messages can speed up the diagnosis time of production-run failures by 2.2 times [35].	Logging is beneficial for diagnosing production-run failures.
Churns of logging code (Section III-C)	**Implications**
(3) The average churn rate of logging code is almost two times (1.8), compared to the entire code.	Logging code is being actively maintained by developers.
(4) In contrast to its relatively small density, logging code is modified in a significant number (18%) of all the committed revisions.	Logging code takes a significant part of software evolution despite its relatively small presence.
(5) 33% of modifications on logging code are after-thoughts. The remaining ones are updates together with other non-logging code changes within the same patch to make them consistent. More than one third (36%) of our studied log messages have been modified at least once as after-thoughts.	The current logging practice is ad hoc, introducing problems to the log quality. Developers take their efforts to address them as after-thoughts.
Overview of log modifications (Section IV)	**Implications**
(6) Developers seldom delete or move logging code, accounting for only 2% of all the log modifications. Almost all (98%) of the log modifications are to the content of the log messages.	We surmise there is lack of documentations explaining the purpose of log messages so that developers are conservative in deleting/moving log messages.
Modification to logging content (Section V, VII, VI)	**Implications**
(7) Developers spend significant efforts on adjusting the *verbosity level* of log messages, accounting for 26% of all the log improvements. Majority (72%) of them reflect the changes in developers' judgement about the criticality of an error event.	Tools that systematically exposing error conditions would help testing the logging behavior. Testing and code analysis tools need to provide more information (e.g., error conditions) for developers to decide the proper verbosity.
(8) In 28% of verbosity level modifications, developers reconsider the trade-off between multiple verbosities. It might indicate that developers are often confused when estimating the cost (e.g., excessive messages, overhead) and benefit afforded by each verbosity level.	The scalar design of current verbosity level may not be a good way to help developers with such logging decision. More intelligent logging methods, such as adaptive logging, can help balancing the trade-offs.
(9) One fourth (27%) of the log modifications are to variable logging. Majority (62%) of them are adding new variables that are causally related to the log message. It indicates that developers often need additional runtime information to understand how a failure occurred.	Logging tools that automatically infer which variables to log (e.g., LogEnhancer [36]) can help informative logging. Given failing and passing test cases, Delta debugging [37] can also be used to infer the relevant variables to log.
(10) 45% of log modifications are modifying static content (text) in log messages. More than one third (39%) of them are fixing inconsistencies between logs and actual execution information intended to record. Software can leverage programming language support to eliminate certain inconsistency, as Squid does.	Developers should pay more attention to update the log messages as code changes. Tools combining natural language processing and static code analysis can be designed to detect such inconsistencies.

Table II
OPEN-SOURCE SOFTWARE WE STUDIED. THE THIRD COLUMN SHOWS THE POPULARITY OF THE SOFTWARE IN ITS OWN PRODUCT CATEGORY. *: AMONG ONLY OPEN-SOURCE DATABASE SERVERS, IT HAS THE SECOND LARGEST MARKET SHARE.

Software	Description	Market share	LOC	Verbose levels	Log messages		Lines of logging code	LOC per log	RCS
					total	default mode			
Apache httpd 2.2.16	Web server	top 1 [22]	249K	8	1838	1100 (`warn`)	6758	36	SVN
OpenSSH 5.8p2	Secure shell server	top 1 [23]	81K	8	3407	2241 (`info`)	4672	17	CVS
PostgreSQL 8.4.7	Database server	top 2* [30]	614K	13	6052	5818 (`warn`)	20733	30	GIT
Squid 3.1.16	Caching web proxy	top 1 [31]	155K	10	3474	1268 (`info`)	4103	38	Bazzar
Total			1.1M		14771	10427	36266	30	

B. Study Methodology

We study various aspects of the logging practice. To study the density of logging code, we measure both the lines of code (LOC) of the entire program and the logging code (shown in Table II). Note that the LOC of logging code is larger than the number of log printing statements since a logging statement might occupy more than one code lines.

The code churn rate is measured in $Churned\ LOC/Total\ LOC$ [20]. The churn rate of logging code is thus measured in $churned\ LOC\ of\ logging\ code/LOC\ of\ logging\ code$. We analyze each revision in the recent five year's history of software to measure the churned code. We first measure the churn rate for each of the years, then take the average of these five one-year churn rates.

For studying the modifications on logging code, we only focus on the modifications to the log printing behavior, including verbosity levels, static content, variable values and

log message location. None-behavioral modifications, such as renaming log printing functions or verbosity levels (e.g., from `warn` to `warning`), indent changes, etc., are excluded from our analysis.

With the collected modifications, we study the types of modifications. In the revision history, there could be two types of log modifications: some are merely for consistent update with other non-logging code changes *within the same revision*, and others are modifying the logging behavior as after-thoughts. To separate the modification types, one possible policy is to check whether the revisions only include changes solely to logging code but not to other code. However, this is too conservative in that developers tend to batch multiple patches into one revision.

Instead, we use a few simple heuristics following our observations on the common logging practice: developers often log right after checking a certain condition (e.g., an error condition), which is usually implemented with a branch statement (e.g., if, while, etc.). In a revision, if such a branch statement is modified together with following logging code, it may not be introduced for logging adjustment, but for changing program semantic together with logging. Similarly, if a variable or a function is renamed consistently both in the logging code and non-logging code within the same revision, it is also not modifying the logging behavior.

Since our automatic analysis may not be accurate, we further manually verify our analysis result on 400 randomly sampled modifications (100 from each program). Our manual verification suggests the accuracy of our analysis is 94%.

We further study the details of those log modifications. For some types of such modifications, to reason about why the previous logging was problematic or insufficient, we randomly sample the same modification cases and carefully examine the relevant source code, comments, commit log, bug reports, and discussions in mailing list (if any). If we cannot clearly understand the reason, we always conservatively classify them as the "other" category when presenting our results. The confidence interval of our sampling is reported together with our result whenever sampling is used.

C. Threats to Validity

As with all the previous characteristics study, our work is also subject to a validity problem. Potential threats to the validity of our characteristic study are representativeness of the software and examination methodology.

To address the former, we selected diverse open-source software in terms of functionality, including Web server, database server, caching proxy, shell server, and together with their client utilities, all of which are widely used in their product category as shown in Table II. They have at least 10 years of history in their code repositories and more than 14771 static log points in source code. Overall, we believe that they well represent large software which embed the current logging practices. However, our study may not reflect the characteristics of logging practices in other types of non-server/client software, such as scientific applications, operating systems, commercial software, or software written in other programming languages.

As for our examination methodology, we try to minimize our own subjective judgement on the quality of log messages by systematically analyzing *developers' own modifications* to the log messages. Also, we examine developers' commit logs, comments, related bug reports, etc., together with the source code to reason about the modifications. Furthermore, our study includes all of the aspects typically considered by developers for logging, including verbosity level, static content and variables to record, and log placement in source code. As a limitation, for some logging problems unknown even to the developers, our methodology may also miss them, since the modification is not in revision history. However, if the problem is general enough, it should have been fixed in at least one of the program we studied.

We do not study the *additions* of new logging code. This is because our goal is to reveal issues with the current logging practices, therefore we only focus on the *modifications* (including deletions) to the previously existing logging code. However, adding new logging code might also reflect issues with existing logs where it is a revival of the existing logging code that has been deleted once. While we study the deletions in such addition/deletion chains, we will miss the additions where they might have important meanings as well. However, our results in Table V and Table XII suggest that the deletions of log messages rarely occur (less than 2% among all of the modifications). Therefore, we expect that such deletion/addition chains are also rare.

Overall, while we cannot draw any general conclusions that can be applied to all software logging, we believe that our study provides insights about efficacy and pitfalls of software logging, particularly in open source server applications written in C/C++.

III. Importance of Log Messages

In this section we study the pervasiveness of software logging in reality and the benefit of software logging to production-run failure diagnosis.

A. Code Density of Sofware Logging

> **Finding 1:** On average, every 30 lines of code contains one line of logging code. Similar density is observed in all the software we studied.
>
> **Implications:** Logging is a pervasive practice during software development.

The code density of software logging is shown in the "LOC per log" column in Table II. It is calculated using the LOCs of logging code and the entire code. Even in the software with least log density (Squid), there is still one line of logging code per 38 lines of code.

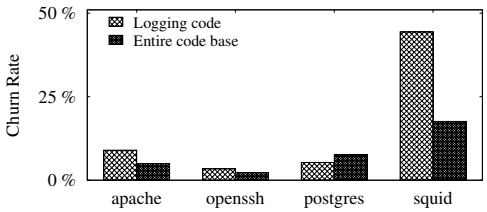

Figure 2. Churn rates comparison between logging and entire code.

B. Benefit of log messages in failure diagnosis

Finding 2 (Benefit of log messages): Log messages reduces median diagnosis time of production-run failures between 1.4 and 3 times (on average 2.2X speedup).

Implications: Logging is beneficial for failure diagnosis.

We randomly sampled 250 user reported failures from Apache, Squid, and PostgreSQL, and compared the failure resolution time between the set of failures where user provided *any* log messages with the ones without *any* log messages. The details are discussed in our previous work [35]. Jiang et. al. [13] revealed a similar finding by studying production failures in commercial systems.

C. Churns of Logging Code

Finding 3: As shown in Figure 2, the average churn rate of logging code is almost two times (1.8) compared to the entire code. Interestingly, except for PostgreSQL, all the software show that logging code have higher churn rates than the entire code base.

Implications: Developers are actively maintaining logging code like other non-logging code for software functionality. Logging is at least as important as other part of code in the maintenance perspective.

Such churns on logging code are also scattered across many revisions, indicating the logging code is *continuously* maintained as a significant part of software evolution:

Finding 4: In contrast to the relatively small density of logging code (Finding(1)), a significant number (18%) of all the committed revisions modify logging code.

Implications: Logging code takes a significant part of software evolution despite its relatively small presence.

Overall, Finding 3 and 4 together implicate that logging code is being continuously and actively modified. To understand what these modifications are, we studied modifications from the revisions of our studied software.

Table III shows the detailed classification of modifications to logging code, which is from our automatic analysis tool on committed revisions (with 94% accuracy) as described in Section II-B. Our tool identifies a modification as a *consistent update* with the other changes on non-logging code if the *same patch* contains one of the following three types of changes: (i) modification on the conditions that the logging code is dependent on; (ii) re-declaration of the logged variable that is also changed in logging code; (iii) modification on a function name that is also referred in the logging code as static text. Otherwise, our tool classify the modification on logging code as an after-thought. Table IV shows the number of log messages that have been modified at least once as after-thoughts by these 9076 modifications.

Table III
MODIFICATIONS IN EACH REVISION TO THE LOG MESSAGES.

Software	total	after-thoughts	following other code change		
			condition	variable	function
apache	3035	810 (27%)	1941	64	220
openssh	3459	1446 (42%)	1703	284	26
postgres	15455	5389 (35%)	9153	746	167
squid	5536	1431 (26%)	2951	930	224
Total	27485	9076 (33%)	15748	2024	637

Table IV
LOG MESSAGES(%) THAT HAVE BEEN MODIFIED.

log msgs	apache	openssh	postgres	squid	total
modified	605	628	3128	1106	5367
total	1838	3407	6052	3474	14771
percentage	40%	18%	52%	30%	36%

Table V
TYPE OF LOG MODIFICATIONS AS AFTER THOUGHTS.

Software	Log Modifications				
	total	location	verbosity	text	variables
apache	810	35 (4%)	118 (15%)	429 (52%)	228 (28%)
openssh	1446	33 (2%)	550 (38%)	264 (18%)	599 (41%)
postgres	5389	17 (1%)	1148 (21%)	3000 (56%)	1224 (23%)
squid	1431	65 (5%)	573 (40%)	364 (25%)	429 (30%)
Total	9076	150 (2%)	2389 (26%)	4057 (45%)	2480 (27%)

Finding 5: 33% modifications on logging code are *after-thoughts*. The remaining ones are consistent updates with the other changes on non-logging code *in the same patch*. As a result, 36% of the total 14771 log messages have been modified at least once as after-thoughts.

Implication: In current practice, logging is conducted in a subjective and arbitrary way, introducing problems to the log quality. Developers take efforts to improve them as after-thoughts.

In remainder of the paper, we will use *modifications* to only refer to these modifications that are *not* consistent updates with other non-logging code changes, unless otherwise specified. We focus on studying these modifications as they are likely to reflect more directly developers' concerns over the previously problematic log messages.

IV. OVERVIEW OF LOG MODIFICATIONS

In Table V, we further break the 9076 modifications based on what developers modified: the location of logging code within the source code, verbosity level, static content of a log message, and variables to log. For location change, we consider the logging code's relative location within a basic block, including both move and deletion.

Table VI
VERBOSITY-LEVEL MODIFICATION WITH ERROR EVENT LOGGING AND WITH NON-ERROR EVENT LOGGING

software	total	error	non-error
apache	118	84 (71%)	34 (29%)
openssh	550	398 (72%)	152 (28%)
postgres	1148	831 (72%)	317 (28%)
squid	573	399 (70%)	174 (30%)
Total	2389	1712 (72%)	677 (28%)

Finding 6: Developers seldom delete or move the logging code, accounting for only 2% of all the log modifications. Almost all (98%) modifications are to the content of the log messages, namely verbosity level, static text and variables.

Implications: Given the lack of specifications on logging behaviors, developers would not delete/move log messages unless they cause serious problems (Section VIII).

V. VERBOSITY LEVELS MODIFICATION

This section analyzes the 2389 modifications to verbosity levels (Table V), which indicate developers often do not assign the right verbosity level at the first time.

In Table VI, we further break down the verbosity level modification into those for error event logging and non-error event logging. In the former (72%), at least one verbosity level before or after the modification is an error-level (e.g., `error`, `fatal`, etc.). These indicate that developers might have misjudged *how critical the event to log is* at the first place. Please recall that in these modifications developer *did not change the conditions* (typically the error condition) leading to the log messages, but only the verbosity level themselves, indicating they are likely *after-thoughts*. In the other 28% verbosity level modifications, developers change between non-error (also non-fatal) levels, such as `info` and `debug`, which are supposedly to record non-error events.

Finding 7: Majority (72%) of the verbosity-level modifications reflect the changes in developers' judgement about the criticality of an error event (Table VI).

Implications: Tools that systematically exposing error conditions would help test the logging behaviors. For example, fault injection tools [12] can be used to inject faults to trigger an error and consequently its error logging. Similarly, software model checking [2] can be extended to explore the execution paths reaching the logging code.

A. Reconsidering Error Criticality

Table VII breaks down the verbosity-level modifications for error event logging. More than half (56%) of the cases are changing levels between non-fatal and fatal. This class is different from others in that they are introduced to change the system's execution behavior as well as logging behavior. Specifically, with the modifications, developers

Table VII
RECONSIDERATIONS OF ERROR CRITICALITY AND VERBOSITY-LEVEL MODIFICATION.

Software	non-fatal to fatal	fatal to non-fatal	non-error to error	error to non-error	others
apache	18	12	12	37	5
openssh	80	169	75	71	3
postgres	236	294	148	67	86
squid	29	127	42	201	0
Total	363 (21%)	602 (35%)	277 (16%)	376 (22%)	94 (6%)

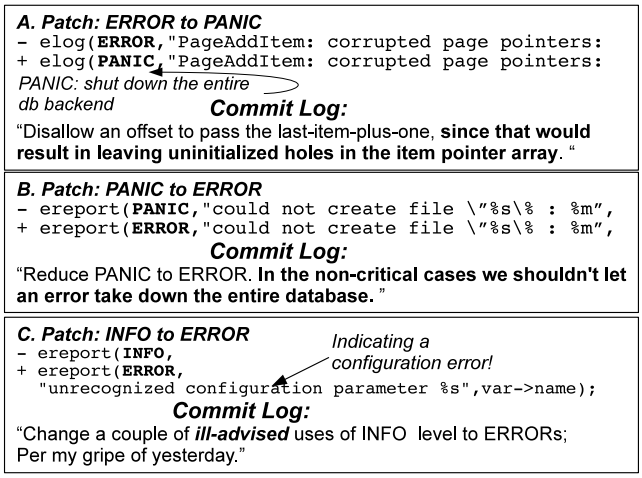

Figure 3. Error verbosity level modifications from PostgreSQL

changed their decision either to enforce a system to abort after logging, or allow it to continue to run.

The modification from a non-fatal to a fatal level is to prevent a non-survivable error from propagating, which can lead to serious system malfunctions or security issues. On the other hand, the modification from a fatal to a non-fatal level is to avoid an unnecessary system termination on a survivable error for better system availability.

For example, in Figure 3 (A), PostgreSQL developers originally record an error event (i.e., an access to an uninitialized pointer) at `ERROR` level, which could potentially introduce security holes if not aborted right away. Later they provide this patch only to promote the level to a `PANIC` (fatal in this software) that will abort the entire database backend. As an opposite example, in Figure 3 (B), PostgreSQL developers prevent non-critical cases from taking down the entire database by demoting the original `PANIC` to `ERROR`.

In Table VII, some others (38%) are changing verbosity levels between an error level and non-error levels. In those cases, developers may reconsider their original judgements about whether the event to record is an error or not, because recording a real error with non-default verbosity level such as `debug` would cause missing important error messages for failure diagnosis, and recording a non-error event with `error` level might either confuse the users and developers or cause unnecessary production run overhead. For example, Figure 3 (C) shows that PostgreSQL developers originally

missed to report a configuration error by logging it with `info` which is not a default verbosity level for production run in PostgreSQL. After suffering from diagnosing it without logs, they committed a patch only to change it to `error`.

B. Reconsidering Logging Trade-offs

As shown in Table VI, 28% of the verbosity-level modifications come from non-error event logging. In general, non-error events are logged with one of multiple verbose levels such as `debug1, debug2`,..., or sometimes even with a default levels, e.g. `info` in Squid. Table VIII decomposes the verbosity modifications for non-error events.

Table VIII
RECONSIDERATION OF LOGGING TRADE-OFF AND VERBOSITY-LEVEL MODIFICATION

Software	between verbose	verbose to default	default to verbose	between default
apache	23 (67%)	3 (9%)	8 (24%)	0 (0%)
openssh	116 (76%)	11 (7%)	25 (16%)	0 (0%)
postgres	132 (42%)	108 (34%)	59 (19%)	18 (5%)
squid	115 (66%)	38 (22%)	21 (12%)	0 (0%)
Total	386 (57%)	160 (24%)	113 (17%)	18 (3%)

Finding 8: For non-error event logging, developers reconsider the trade-off between multiple verbosity levels. It might indicate that developers are often confused when estimating the cost (e.g., excessive messages, logging overhead) and benefit afforded by each verbosity level.

Implications: The scalar design of current verbosity level may not be a good way to help developers with such logging decision. Adaptive logging in runtime, similar to adaptive sampling [11], can help balancing the trade-off by dynamically backing-off the logging rate.

In Table VIII, more than half (57%) of the non-error verbosity level modifications are changing between two *verbose* levels. In all the studied software, verbose levels are not enabled by default, meaning that they are mostly used during in-house testing. Therefore, the logging overhead may be less of concern when developers make such adjustments. Instead, developers probably are more concerned about balancing the amount of logs: too excessive logging would rather make noises for failure diagnosis, but insufficient logging would miss important runtime information.

One of the possible causes for such many adjustments within verbose levels might be because no clear division among multiple verbose levels is given in terms of purpose of use, benefit, and cost, resulting in confusing developers when deciding among the verbose levels. For example, in Squid, there are 7 `debug` levels out of total 10 verbosity levels, but no guidance for which cases they should be used. Indeed, in their header file, developers wrote a comment saying *"level 2-8 are still being discussed amongst the developers"*. We surmise that developers would decide which level to use arbitrarily at the first place and often revisit the decision later.

Table IX
MODIFICATIONS TO IMPROVE VARIABLE LOGGING (*: E.G. FROM INTEGER FORMAT TO FLOAT FORMAT)

software.	total	add var.	replace var.	delete var.	change format(*)
apache	228	81	68	15	64
openssh	599	348	106	24	121
postgres	1224	839	184	102	99
squid	429	278	45	26	80
Total	2480	1546 (62%)	403 (16%)	167 (7%)	364 (15%)

Bug Report from user: Downgrade log messages

User: Recommending downgrade of these annoying messages from **LOG** to **DEBUG1**:
LOG: transaction ID wrap limit is 214170, limited by database "postgres"
LOG: transaction ID wrap limit is 214170, limited by database "postgres"

Developer: *"*Seems reasonable -— we put that in at LOG level for purposes of testing the 8.1 XID wraparound prevention logic, but by now I think we can trust that code a bit more."

Figure 4. Example from PostgreSQL of a verbosity level demotion from default level (non-erroneous) to verbose level, together with developers' commit log.

For non-error logging with default level (e.g., bookkeeping with `info`), developers may need to carefully consider more factors since it would directly affect production run. For example, Figure 4 shows that PostgreSQL users complained about the excessive log messages, and thus developers demote the previous default level (`LOG`) to verbose level (`DEBUG`). Interestingly the developers originally assigned a default level because the event was in some new code that potentially is buggy, but it resulted in excessive logging at a user site. In addition, of course developers may need to consider logging overhead in production run.

Overall, setting the verbosity level by considering all those factors may not be easy at the first place, or need further adjustment as software and environment changes. Unfortunately, the current scalar design of verbosity level does not provide enough information for developers. To help developers, systematic and dynamic logging tools or assists, such as adaptive logging [11], are needed. Instead of using a statically assigned verbosity, adaptive logging exponentially decreases logging rate when a certain logging statement is executed many times, only recording its 2^n dynamic occurrences. Such strategy will reduce both the amount of logs and performance overhead, while preserving the first several occurrences of each log message.

VI. MODIFICATIONS TO VARIABLE LOGGING

Table IX shows how developers improved variable logging. Majority of them are adding new variables to original logging code, which could provide more dynamic information for failure diagnosis. For example, in Figure 5, a user of

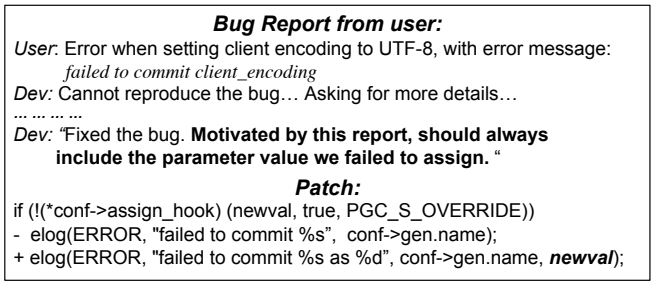

Figure 5. Example of adding variable values to log message.

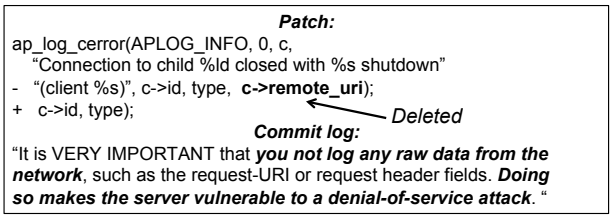

Figure 6. Logging a wrong variable causing Apache vulnerable to denial of service attack.

PostgreSQL reported a production-run failure with an error message printed by the software. Unfortunately, developers could not diagnose the failure due to the lack of runtime information. Only after a couple of rounds of back-and-forth discussion with users they resolved this. From the lessons, later they committed a patch only to a new variable causally-related to the logging point.

Finding 9: One fourth (27%) of the log modifications are to variable logging. Majority (62%) of them are adding new variables that are causally related to the log message. It indicates that developers often need additional run-time information to understand how a failure occurred.

Implication: Logging tools that automatically infer which variables to log (e.g., LogEnhancer [36]) can help informative logging. Given failing and passing test cases, Delta debugging [37] can be used to log those variable values that are specific to a failing run.

Interestingly, once variables are introduced into logging code, they are seldom (7%) deleted, as shown in Table IX. Probably it is because recording unnecessary variables often would not introduce serious concerns besides one or two useless variable values in the log.

However, there can also be certain variables that should not be logged, considering security and privacy concerns, and developers should be careful to avoid them. Figure 6 shows that Apache developers deleted a variable including a client's URI from the logging code, since recording it could "make the server vulnerable to denial-of-service attack".

To further understand why variables are deleted or replaced in the logging code, we manually study 154 such modifications that are randomly sampled. As a result, Table X shows that (i) as the most dominant case, the original logging code records wrong variables at the first place, either only by mistakes or by not being aware of security or privacy concerns; (ii) other non-logging code was evolved but the variable logging was not updated together, becoming inconsistent; (iii) an error number such as `errono` was printed without interpretation, requiring replacement to readable description; (iv) a log message includes redundant variables, preferred to be deleted. The remaining cases, where we cannot understand the modifications from their source code, commit logs, or comments, belong to the "other" category.

Table X
VARIABLE REPLACEMENT AND DELETION. THE MARGIN OF ERRORS ARE SHOWN AT 95% CONFIDENCE LEVEL.

wrong var.	incon-sistency	read-ability	redun-dancy	other
46% (±6%)	11 (%±4%)	23 (%±5%)	2 (%±2%)	18 (%±5%)

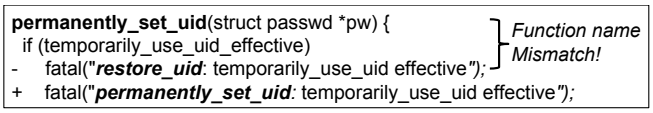

Figure 7. Example of inconsistency between log messages and the code in OpenSSH. This patch is just to fix this inconsistency.

VII. MODIFICATIONS TO STATIC CONTENT

45% of the log modifications are modifying the static content (text) in log messages. Since it is challenging to automatically analyze the text written in natural language, we randomly sampled 200 modifications and studied them, which are shown in Table XI.

Table XI
IMPROVING STATIC CONTENT OF LOG MESSAGES. THE MARGIN OF ERRORS ARE SHOWN AT 95% CONFIDENCE LEVEL.

incon-sistency	clarifi-cation	spell/ grammar	incorrect content	others
39%(±6.6%)	36%(±6.5%)	18%(±5.2%)	5%(±2.9%)	2%(±1.9%)

In some cases (39%), developers modified the out-of-date log messages that are inconsistent with the actual execution information, which could mislead and confuse the developers or users (please note that those consistent updates of both log and code in the *same* patch are excluded from our analysis by our analysis tool). Majority (76%) of them are related to function name changes. For example, in Figure 7, OpenSSH developers changed a function name from "restore_uid" to "permanently_set_uid" but forgot to update the logging code to record this name. Later, they were confused with the out-of-date log message while trying to resolve a failure. Finally, they committed this patch just to fix the inconsistent logging code.

Such inconsistency can be partly avoided by using programming language support. For example, C programming language provides a macro "__FUNCTION__" as part of the ANSI-C99 standard, which holds the function name within which the code is currently executing. As a good logging practice, as shown in Figure 8, Squid started to use this in its logging code to automatically recognize a function

```
/* HERE is a macro that you can use like this:
 * debugs(1, HERE << "some message");  */
#define HERE __FILE__<<"("<<__LINE__<<") "<<__FUNCTION__<<": "

                                  Patch:        Fixing inconsistency
 if (fd < 0) {                                  using 'HERE'
-  debugs(3, "BlockingFile::open: got failure (" << errno <<  ")");
+  debugs(3, HERE << ": got failure (" << errno << ")");
```

Figure 8. The use of the programming language support to hold location information for log messages in Squid.

name and log it. This eliminates the need for developers to manually record or update a location information, avoiding the inconsistent update problem at the first place.

To detect other inconsistent updates (e.g., an event to log and its description), it would be beneficial to use natural language processing together with static source code analysis, similar to iComment [29] which uses natural language processing to automatically analyze comment and source code in order to detect inconsistency.

Finding 10: More than one third (39%) of modifications to static content are fixing inconsistency between logs and actual execution information intended to record. Software can leverage programming language support to eliminate some of the inconsistency, as Squid does.

Implication: Tools combining natural language processing and static code analysis can be designed to detect such inconsistency.

Bug Report from user: Confusing message in log file

"I changed the postgresql.conf file, and see the following messages:

> *parameter "shared_buffers" cannot be changed after server start;*
> ***configuration file change ignored***

so I expect both newly enabled "*archive_command" and "shared_buffers"* not to take effect…. But in fact, "*archive_command*" does take effect."

Patch:

```
ereport (ERROR,
-   "parameter \"%s\" cannot be changed after server start;", gconf->name
-   "configuration file change ignored"
+   "attempted change of parameter \"%s\" ignored,", gconf->name
+   "This parameter cannot be changed after server start"
```

Figure 9. Example of a log message clarification from PostgreSQL.

In some other cases (36%), developers modified static content of log messages to clarify the event description in it. As an example, Figure 9 shows that a log message in PostgreSQL was unclear and thus it misled a user to believe that all his configuration changes would lose effect, which was not true. At the end, the modification was made only to clarify the content of the log message.

VIII. Location Change

As we discussed in Finding 6, developers seldom delete or move logging code once it is written. To understand under which cases developers delete/move logging code, we randomly sampled 57 such cases from the 150 location modifications and manually examined them. The Table XII summarizes the results with the sampling errors at the 95% confidence level.

Table XII
Reasons for moving or deleting log messages

software failure	misleading log msg.	reduce noises	others
26%(±9%)	21% (±8%)	40%(±10%)	12% (±7%)

```
 sigusr2_handle(int sig) {
-  debug(1, "sigusr2_handle: SIGUSR2 received.\n");   I/O in signal
+  /* no debug() here; bad things happen if           handler can corrupt
      * the signal is delivered during debug()*/      the system state.
```

Figure 10. Deleting Logging from a signal handler in Squid.

Interestingly, 26% of the location changes were required because the original logging code was misplaced and resulted in software failures. For example, logging in signal/interrupt handlers is dangerous since the non-reentrant I/O operations during logging might corrupt system states and open up vulnerabilities. Figure 3 shows a patch to delete such a problematic logging code from a signal handler in Squid. In addition, logging variables before their initialization would result in system crash or misbehavior; logging non-error events with `fatal` verbosity level will unnecessarily terminate the software execution. To identify these problems above, in-house testing tools and static analysis tools [8] can be extended to explore logging place. For example, we can use static analysis to detect logging statements within interrupt handlers and the use of uninitialized variables.

In some cases, developers delete some misleading log messages (e.g, an error message printed under a non-error situation). From several commit logs, we find that some developers tend to actively log certain events simply with the `error` verbosity level for the purpose of in-house testing, then forgot to completely delete them before the production release. In other cases, log messages are moved out of a loop body or combined into one that can summarize them, probably in order to avoid overhead and noises from excessive logging. Finally, the "others" category includes the cases that we cannot clearly understand.

IX. Verbosity Level Checker

To show the feasibility of automatic logging assistance from our findings, we designed a simple verbosity-level checker which helps identify certain type of problematic verbosity-level assignment. This is motivated by the significant number of verbosity-level adjustments (Finding 7).

Our checker is based on the observation that if the logging code within the similar code snippets have inconsistent verbosity levels, at least one of them is likely to be incorrect [9], [15], [10], [33]. First, the tool identifies all the code clones in the source code (we used CP-Miner [15] to detect code clones). Then, it further checks each pair of clones to see whether they contain logging code and their verbosity levels are consistent.

Table XIII
VERBOSITY-LEVEL INCONSISTENCY DETECTED.

	apache	openssh	postgres	squid
Inconsistency	12	4	89	33

As a result, our checker detected 138 inconsistent pairs of logging code, as shown in Table XIII. We reported 45 cases to the developers. 24 of them are confirmed and fixed [27], 10 are confirmed as false positives where the cloned code snippets are in different contexts so they should have different verbosity levels [27], and the remaining ones are not being responded.

This result shows that based on our finding, even a simple checker can effectively help for better logging. It confirms that the first important step towards systematic and automatic supports for better software logging is to *understand* the current manual efforts for logging, which is exactly the goal of this work.

X. RELATED WORK

Logging effectiveness study: Two pieces of recent work [35], [13] studied the effectiveness of logging in failure diagnosis as one part of their work. LogErr [35] characterized the problem of lacking of log messages for diagnosis, and suggest to check generic error conditions and log all detected errors (i.e., where to log). They further proposed a tool to automatically insert error messages for those generic error conditions. In contrast, we focus on many other aspects of logging decisions, such as verbosity levels, static content, variable values, etc. In particular, LogErr studied failures, but not programmer's modifications on existing log messages. Also it only studied the default-mode log messages, where as we study *all* the messages in this work. Therefore, all of our findings except finding 2 are unique to this work.

Jiang et al. [13] mainly studied the correlations among root causes of storage system failures, impact and diagnosis time. One of their finding confirms the benefit of logging, but does not provide detailed efficacy of logging practices.

Logging improvement: There have been some work to help systematically improve logging. Some can help developers in inserting new log messages, but only for error conditions [35], [4], [24], [3]. SMELL [4] detects some exception handling code which is not logged or inappropriately logged. Some other work [24], [3] propose to generate error checking code (e.g., assertions) from a program specifications, which provide natural logging places for error messages. LogEnhancer [36] can suggest new variable values to be recorded in each existing log message by analyzing the source code.

Our work is complementary to the tools for log improvement as the characteristics of logging practice we reveal can be used to either support the usefulness of these tools and inspire future tools.

Log analysis work: Many systems analyze the production-run logs for post-mortem diagnosis [32], [34], [18], [17]. Some [32], [18] of them learn statistical signatures to detect and diagnose anomalies. SherLog [34] infers the partial execution paths by mapping log messages to source code. Mariani and Pastore [17] analyze logs to learn the correct dependency between log messages from the normal executions, and use the information to identify anomalies in failed executions. The effectiveness of these work depends on the quality of log messages, thus can potentially benefit from our study.

XI. CONCLUSIONS

This paper presents the first (to the best of our knowledge) attempt to study the practice of software log messages using four pieces of large open-source software. We first quantify the pervasiveness and the benefit of software logging. By further studying developers' own modifications on their log messages, we found they often cannot get the log messages right after the first attempts. In particular, developers spend significant efforts in modifying the verbosity level, static text, and variable values of log messages in various ways, but rarely change the message locations. By identifying these common log-modification efforts that are done manually, we reveal many opportunities for tools, compiler and programming language support to improve the current logging practices. Such benefit of our findings is confirmed by a simple checker we built, which is motivated by identifying developers' large amount of manual efforts in modifying the verbosity level, that can effectively detect 138 new pieces of problematic logging code .

ACKNOWLEDGEMENT

We thank the anonymous reviewers for their insightful feedback. This research is supported by NSF CNS-0720743 grant, NSF CCF-0325603 grant, NSF CNS-0615372 grant, NSF CNS-0347854 (career award), NSF CSR Small 1017784 grant and NetApp Gift grant.

REFERENCES

[1] ArcSight Log analysis. http://www.arcsight.com/Logger.

[2] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 1–3, 2002.

[3] L. Baresi and M. Young. Toward translating design constraints to run-time assertions. *Electron. Notes Theor. Comput. Sci.*, 116:73–84, January 2005.

[4] M. Bruntink, A. v. Deursen, and T. Tourwé. Discovering faults in idiom-based exception handling. In *Proceedings of the 28th international conference on Software engineering (ICSE)*, pages 242–251, 2006.

[5] Cisco system log management. http://www.cisco.com/en/US/docs/voice_ip_comm/cucm/service/3_3_2/ccmsrvs/sssyslog.html.

[6] Dell. Streamlined Troubleshooting with the Dell system E-Support tool. *Dell Power Solutions*, 2008.

[7] EMC seen collecting and managing log as key driver for 94 percent of customers. http://www.rsa.com/press_release.aspx?id=7596.

[8] D. Engler, B. Chelf, and A. Chou. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation (OSDI)*, pages 1–16, 2000.

[9] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the eighteenth ACM symposium on Operating systems principles (SOSP)*, pages 57–72, 2001.

[10] M. Gabel, J. Yang, Y. Yu, M. Goldszmidt, and Z. Su. Scalable and systematic detection of buggy inconsistencies in source code. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications (OOPSLA)*, pages 175–190, 2010.

[11] M. Hauswirth and T. M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XI, pages 156–164, 2004.

[12] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer. Fault injection techniques and tools. *Computer*, 30:75–82, 1997.

[13] W. Jiang, C. Hu, S. Pasupathy, A. Kanevsky, Z. Li, and Y. Zhou. Understanding customer problem troubleshooting from storage system logs. In *Proccedings of the 7th conference on File and storage technologies (FAST)*, pages 43–56, 2009.

[14] B. W. Kernighan and R. Pike. *The Practice of Programming*. Addison-Wesley, 1999.

[15] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation (OSDI)*, pages 176–192, 2004.

[16] Apache Logging Services - Log4j. http://logging.apache.org/log4j.

[17] L. Mariani and F. Pastore. Automated identification of failure causes in system logs. In *Proceedings of the 2008 19th International Symposium on Software Reliability Engineering (ISSRE)*, pages 117–126, 2008.

[18] L. Mariani, F. Pastore, and M. Pezzè. A toolset for automated failure analysis. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, pages 563–566, 2009.

[19] J. C. Munson and S. G. Elbaum. Code churn: A measure for estimating the impact of code change. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 24–31, 1998.

[20] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th international conference on Software engineering (ICSE)*, pages 284–292, 2005.

[21] NetApp. Proactive health management with auto-support. *NetApp White Paper*, 2007.

[22] Netcraft report: Apache httpd is the most popular webserver. http://news.netcraft.com/archives/category/web-server-survey/.

[23] Lessons from the success of ssh. http://www.cs.virginia.edu/~drl7x/sshVsTelnetWeb3.pdf.

[24] M. Pezzè and J. Wuttke. Automatic generation of runtime failure detectors from property templates. In B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, and J. Magee, editors, *Software Engineering for Self-Adaptive Systems*, pages 223–240. Springer-Verlag, Berlin, Heidelberg, 2009.

[25] SLOCCount: Source Lines Of Code Count. http://www.dwheeler.com/sloccount/.

[26] Splunk Log management. http://www.splunk.com/view/log-management/SP-CAAAC6F.

[27] Squid bug report 3319. http://bugs.squid-cache.org/show_bug.cgi?id=3319.

[28] RFC3164 – the BSD Syslog protocol. http://tools.ietf.org/html/rfc3164.

[29] L. Tan, D. Yuan, G. Krishna, and Y. Zhou. /* iComment: Bugs or bad comments? */. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles (SOSP)*, pages 145–158, 2007.

[30] Top 10 Enterprise database systems to consider. http://www.serverwatch.com/trends/article.php/3883441/Top-10-Enterprise-Database-Systems-to-Consider.htm.

[31] D. Wessels. *Squid: The Definitive Guide*. O'Reilly, 2004.

[32] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, pages 117–132, 2009.

[33] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Trans. Softw. Eng.*, 30:574–586, September 2004.

[34] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. SherLog: Error diagnosis by connecting clues from run-time logs. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems (ASPLOS)*, pages 143–154, 2010.

[35] D. Yuan, S. Park, P. Huang, Y. Liu, M. Lee, Y. Zhou, and S. Savage. Did you log the error? Characterizing and improving software error reporting. Technical Report.

[36] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage. Improving software diagnosability via log enhancement. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems (ASPLOS)*, pages 3–14, 2011.

[37] A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, SIGSOFT'02/FSE-10, pages 1–10, 2002.