On Matching Log Analysis to Source Code: A Systematic Mapping Study

Vincent Bushong
Computer Science, Baylor University
Waco, Texas, USA
vincent_bushong1@baylor.edu

Russell Sanders
Computer Science, Baylor University
Waco, Texas, USA
Russell_Sanders1@baylor.edu

Jacob Curtis
Computer Science, Baylor University
Waco, Texas, USA
Jacob_Curtis1@baylor.edu

Mark Du
Computer Science, Baylor University
Waco, Texas, USA
Mark_Du1@baylor.edu

Tomas Cerny
tomas_cerny@baylor.edu
Computer Science, Baylor University
Waco, Texas, USA

Karel Frajtak
frajtak@fel.cvut.cz
CS, FEE, Czech Technical University
Prague, Czech Republic

Miroslav Bures
miroslav.bures@fel.cvut.cz
CS, FEE, Czech Technical University
Prague, Czech Republic

Pavel Tisnovsky
ptisnovs@redhat.com
Red Hat
Brno, Czech Republic

Dongwan Shin
Computer Science, New Mexico Tech
Socorro, New Mexico, United States
dongwan.shin@nmt.edu

ABSTRACT

Logging is a vital part of the software development process. Developers use program logging to monitor program execution and identify errors and anomalies. These errors may also cause uncaught exceptions and generate stack traces that help identify the point of error. Both of these sources contain information that can be matched to points in the source code, but manual log analysis is challenging for large systems that create large volumes of logs and have large codebases. In this paper, we contribute a systematic mapping study to determine the state-of-the-art tools and methods used to perform automatic log analysis and stack trace analysis and match the extracted information back to the program's source code. We analyzed 16 publications that address this issue, summarizing their strategies and goals, and we identified open research directions from this body of work.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**; *Maintaining software.*

KEYWORDS

log analysis, log mining, anomaly detection, program slicing, static code analysis

ACM Reference Format:
Vincent Bushong, Russell Sanders, Jacob Curtis, Mark Du, Tomas Cerny, Karel Frajtak, Miroslav Bures, Pavel Tisnovsky, and Dongwan Shin. 2020. On Matching Log Analysis to Source Code: A Systematic Mapping Study. In *International Conference on Research in Adaptive and Convergent Systems (RACS '20), October 13–16, 2020, Gwangju, Republic of Korea.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3400286.3418262

RACS '20, October 13–16, 2020, Gwangju, Republic of Korea

ACM ISBN 978-1-4503-8025-6/20/10…$15.00
https://doi.org/10.1145/3400286.3418262

1 INTRODUCTION

Program logging is one of the oldest methods for providing developers and users feedback on what is happening in an application. Logs range from simple progress reports in console applications to health checks, request status, and error reports in large distributed systems. These logs represent a rich source of information about the internal workings of an application. Developers usually place logging statements at critical junctures in their application, indicating the success or failure of certain operations and reporting abnormalities that occur.

For all the information contained within them, however, logs present challenges in extracting that information. Manual analysis of logs is possible, but the task quickly becomes unfeasible for large programs that produce many logs [7]. Making sense of distributed logs is even harder: if several modules each have their own output, the ordering of anomalous events may have to be reconstructed across several series of log messages matched with different code bases.

This is the problem space where automatic log analysis fits in. Logs give information programmatically, and automated log analysis seeks to recover that information. However, logs present several challenges to automated analysis. The biggest is that of formatting: while there are some common logging practices, logs are written in natural language by different people, almost guaranteeing there will not be a single, common format for all logs even within an application. As a result, creating generalized tools to parse logs from diverse applications is a non-trivial task that has generated much research interest in multiple directions.

A subset of log analysis work focuses on using a program's source code as an extra input into the analysis. Instead of using the logs solely, certain aspects of the log information is matched to locations in the source code. Including source code in the analysis

could improve the quality of log analysis itself since analysis tools do not need to make assumptions on how the logs were generated; they can instead use the actual logging statements in the code. In addition to making the analysis easier, using both logs and source code could also improve the final quality of the analysis: the use of source code provides a potentially more useful analysis than the logs alone could reveal. Using log analysis can also narrow down the areas of code that need to be analyzed to a more feasible number, allowing developers to identify bugs more quickly.

Analyzing program logs is useful for different purposes. A common goal is that of anomaly detection [11]. If the program behaves differently than expected, the log output will often indicate this, either directly by logging a message describing the problem or indirectly by showing a different composition or ordering of messages than during successful program execution. Fault localization takes this a step further by searching for the root cause of the anomaly within the program's source code, for example, an erroneous value assignment [20]. Log analysis, combined with source code analysis, can be used to assist this process. In case of an unexpected program output or crash, logs can contain information that indicates the program's state, and they also implicitly show the program's execution path. If a message appears in the log, there had to be a logging statement in the code that generated it and, by extension, the execution path through the program's code had to contain that logging statement. Using this approach, the possible locations of the error can be narrowed down. Reducing the possible execution paths is known as program slicing, and it is a common technique to assist with fault localization [21].

Stack traces are even more conducive to program slicing since they provide execution path information explicitly by showing all functions on the stack at the time of the exception. These stack traces are generated and put into the programs log by the execution engines when runtime exceptions occur. While the error may have originated in a function that has since been popped off the stack (e.g., a problematic value was returned by a function), the stack trace provides a good starting place for fault localization through program slicing. Stack traces also adhere to a consistent format for a given programming language, making them easier to automatically parse than other forms of log output.

Our goal in this mapping study is to analyze the methods identified by the research literature for extracting information from program logs and stack traces and matching it back to information in the source code. We also identify the goals that such analysis and matching have been applied toward, as well as its current limitations and open research directions. We found that the current body of literature surrounding mapping information from logs to source code is quite small and open for expansion. The rest of the paper is organized as follows. In Section 2, we discuss our research method. Section 3 details our analysis of the literature and our answers to our research questions. We discuss threats to validity in Section 4, and section 5 concludes our work.

2 RESEARCH METHOD

In our mapping study, we followed simplified guidelines for mapping studies in software engineering, as suggested by Petersen et al. [13]. First, we defined research questions to guide the study. Next, we created a query to find existing works on the topic. We then filtered out results that were not relevant to our specific topic. Finally, we analyzed the remaining works to identify their methods and goals.

We defined the following research questions:

- RQ1 What methods have been used to extract and map information from logs to source code?
- RQ2 What other techniques can be improved by log-to-source matching?
- RQ3 What are the problems or opportunities that have been addressed by log-to-source mapping?
- RQ4 What other sources of information are combined with logs to give insight into the source?
- RQ5 What are the open problems and future research directions?

Next, we crafted our search query. To find those sources related to log analysis, we used the portion *"log mining" OR "log analysis" OR "log extraction" OR "log parsing" OR "log parser"*, and to narrow that down to sources that used the source code alongside the logs, we included *"program slicing" OR "code" OR "fault localization"*. To broaden our results to include more general works on program slicing (which implicitly involves the source code), we included *"program slicing" AND ("fault localization" OR "data flow analysis" OR "runtime exception" OR "stack trace")*. The complete query is as follows:

```
(("program slicing" OR "code" OR "fault localization")
AND  ("log mining" OR "log analysis" OR
  "log extraction" OR "log parsing" OR "log parser"))
OR
("program slicing" AND
  ("fault localization" OR "data flow analysis" OR
  "runtime exception" OR "stack trace"))
```

We used this query in searching four major research databases: the ACM Digital Library, IEEE Xplore, SpringerLink, and ScienceDirect. The query was adapted to local specifics of the particular database search engine.

Table 1: Search results on major indexing sites

Indexer	Found results	Used results
ACM DL	39	8
IEEE Xplore	39	5
SpringerLink	492	1
ScienceDirect	18	2
Total	588	16

The results of the search query are shown in Table 1. We then filtered these results, removing works that were not related to our topic. Specifically, we eliminated those results that did not use either program log output or a stack trace as a source of information. We also eliminated those that did not use the source code in their analysis. In filtering the results, we first discarded irrelevant papers based on their titles and abstracts, followed by a more thorough analysis of the full text.

After we filtered the results, only a small number of works remained. Out of the 588 works initially found, 16 were deemed relevant to the topic. The chosen works are listed in Table 2.

3 ANALYSIS OF RESULTS

In this section, we analyze the existing works, enumerating different methods of extracting information from logs and stack traces and matching it to source code, as well as the problems addressed in the current literature.

3.1 Log analysis and mapping methods

The first step many works take is creating log templates that match log messages to the line of source code that generated them. The most common approach is to parse the source code. For example, by creating and traversing an Abstract Syntax Tree (AST). It is common to find all logging function calls, and create a regular expression that matches the format of the logging function's parameters. This takes into consideration both the static message and the variables in it. That regular expression is then associated with its line of source code and stored. The regular expressions are then compared against future log messages to determine their origin point in the code. This approach is taken by [2, 5, 8, 9, 12, 14, 15, 22, 23, 25].

There are a few variations on this basic approach. In [5], the authors augment the regular expression to detect logs that originated in looping or conditional code. In [15], the regular expressions for log templates are associated with the larger source code unit (class or method) they are contained in, instead of a source line. A more sophisticated template was generated in [25]; in addition to differentiating between the static message portion and passed-in variables, the variables themselves were classified by the probability their values would remain unchanged between requests, determined by data flow analysis. Variables unlikely to change were used as keys to differentiate individual requests.

Using the log messages usually consists only of matching the messages to their lines of code. Some works, however, applied further processing. In both [2] and [5], the logs were partitioned by the ID of the thread that generated them. The ordered, partitioned log sequence was then used in the next stages of their analysis. In [25], the logs were partitioned by individual requests using the variables that were determined to be invariant within a request. In [14], the log templates were also indexed in the database to reduce the number of regular expression comparisons that must be performed. Logs can be clustered by similarity, measured by weighted edit distance [8]. This can be used to match logs with parameters not detected in the creation of the original regular expression.

Another direction is to use statistical analysis on incoming logs to detect anomalous log sequences [9]. In this case, a series of log points represents a task, and a statistical analyzer detects anomalous tasks by detecting outliers among the log point series.

3.2 Execution traces

A common technique we found is using the log output to assist in building a graph representing the program's execution flow. In such a Control-Flow Graph (CFG) or function call graph, the number of paths can be reduced using information extracted from the log, a technique known as program slicing. This approach is seen in a number of studies [1, 2, 5, 10, 16, 17, 23–25].

A typical example of this approach is detailed by Chen et al. [5]. The authors narrow down the execution path by labeling statements with *"may"*, *"must-have"*, and *"must-not"* labels that show for any particular run of a function which statements could have been executed to get to that point in the code. Logging statements are used as checkpoints to indicate whether the program executed a certain line of code. The analysis depends on conditional branches in the code; a branch of a conditional that contains a log statement can be labeled *"must-have"* if that log is found, and *"must-not"* if the log was not found. Similar technique and labels are used in [24], this time calling the statements *"might-execute"*, *"partial-execute"*, and *"not-executed"*. The authors of [23] used combinatorics and a Boolean satisfiability (SAT) solver to rule out infeasible execution paths given conditional constraints encountered. By removing the unused functions, these techniques improve the accuracy of the program slice.

Once an execution trace has been created, it can be used as the basis for further analysis. In [10, 16, 24], stack traces used to slice programs. Since the exception gives a definite endpoint for the slice, backward data-flow analysis is used to find the erroneous statement. The authors of [17] generate a call graph from a stack trace, which they enhance by mining similar stack traces from external sources and including edges from the new traces in the call graph.

After building the execution trace, the authors of [1] utilize predicate switching to determine the location of variables critical to the execution at the time of the fault, then uses backward slicing in order to gather more relevant variables that may have affected the outcome of the predicate that led to the fault.

Additional analysis was applied in [2] to help with anomaly detection between separate executions. A reachability graph between log statements was generated from the program's CFG. Then, ordered log sequences partitioned by thread were used to generate different execution traces from the reachability graph. The execution traces were analyzed together and each was assigned a trace anomaly index, a measure of similarity between it and other traces. Assuming that anomalous executions are rarer than normal ones, and therefore more likely to have a different log sequence, execution traces with a drastically lower similarity value are more likely anomalous.

The approach in [25] is unique because it is designed to analyze distributed systems. Every top-level method of each node is analyzed to determine what logging methods can be reached from it, as well as what other nodes it calls. The resulting call graph is less granular than other examples, but it covers an entire distributed system and tracks the origin of the logs within the distributed nodes. Observed log messages are grouped by request, and the resulting sequence is applied to the call graph to determine which nodes the request traversed. The logs are then used to create a summary of the requests across nodes, for example, giving performance data using the log timestamps.

3.3 Stack trace analysis

Stack traces are unique from other logs since they specify the exact location of an exception as well as a chain of functions that led to

Table 2: Selected papers

Reference	Title	Year
[5]	An Automated Approach to Estimating Code Coverage Measures via Execution Logs	2018
[15]	Bridging the divide between software developers and operators using logs	2012
[22]	Detecting Large-Scale System Problems by Mining Console Logs	2009
[8]	Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis	2009
[2]	Execution anomaly detection in large-scale systems through console log analysis	2018
[12]	Industry Practices and Event Logging: Assessment of a Critical Software Development Process	2015
[25]	Lprof: A Non-Intrusive Request Flow Profiler for Distributed Systems	2015
[23]	SherLog: Error Diagnosis by Connecting Clues from Run-Time Logs	2010
[9]	Stage-aware anomaly detection through tracking log points	2014
[14]	Tracing Back Log Data to Its Log Statement: From Research to Practice	2019
[1]	Fault localisation for WS-BPEL programs based on predicate switching and program slicing	2018
[17]	On the Use of Mined Stack Traces to Improve the Soundness of Statically Constructed Call Graphs	2017
[16]	Fault Localization and Repair for Java Runtime Exceptions	2009
[19]	Boosting Bug-Report-Oriented Fault Localization with Segmentation and Stack-Trace Analysis	2014
[24]	An improved static program slicing algorithm using stack trace	2011
[10]	A Debugging Approach for Java Runtime Exceptions Based on Program Slicing and Stack Traces	2010

it. They are also more consistently structured than human-defined logs. While they are not available in all circumstances, stack traces contain valuable information about a program's execution in the situations where they are generated (e.g., during a runtime exception). The information contained in a stack trace can be used to recreate a representation of a program's execution [10, 16, 17, 24].

In [10, 16, 24], logged stack traces are used to create a program slice that can be analyzed for fault localization, as described above. In a different approach, the authors of [17] create the initial call graph from a stack trace, and then mine similar stack traces from internet sources, including Stack Overflow and GitHub issues. They use these similar stack traces to create more call graphs, which they combine with the original to create a more comprehensive view of the program execution.

3.4 Other sources of information

While most of the works we found solely used the log and source code, three notable exceptions used information from external sources to enhance analysis. For example, in [19], stack traces were used to augment a traditional information retrieval approach to bug report-based fault localization. Additional weight was given to files mentioned in stack traces, improving the accuracy over textual analysis alone. In [17], the authors mined stack traces from GitHub issues and Stack Overflow that involved the class of interest from the stack trace generated from local program execution. These stack traces were used to enhance the analysis done from the original stack trace. In [15], log templates are linked to bug reports that contain messages fitting those templates. Log templates changes are tracked, and when a template changes, system operators can be alerted that a bug report is no longer up to date.

3.5 Problems addressed

Regardless of the methods used, we found several overarching goals the current research seeks to address. The main goals we found were that of fault localization using program slicing [1, 10, 16, 19, 23, 24] and performance analysis using timing information from logs [8, 25].

Other goals were found, as well. Sometimes, anomaly detection (as opposed to localizing a known fault) is the aim. The authors of [2] compare similarity between executions to determine which ones are likely anomalous. In [22], the goal is to detect anomalies in executions of distributed systems, as well as demonstrate sufficient performance of log analysis on such systems. The results show they have the ability to handle large scale systems involving millions of console log lines. In [9], the goal is to show that anomaly detection in real-time is possible by tracking points in logs.

The main goal of [5] is to achieve high-accuracy code coverage estimates through the use of log statements. The labeling of statements based on the existence of logs for a particular run helps the developers determine over multiple runs the average code coverage estimates for a particular test case.

The authors in [15] aim to inform system operators of code changes that are relevant to them since they are not as familiar with the code base as the developers. They enhance bug reports by detecting when they become outdated by changes to the logs contained in them.

Some works aim to demonstrate improvements in existing techniques. The goal of [17] is to expand the call graph for a given stack trace using additional information about similar stack traces mined from the internet. The goal of [14] is to demonstrate that an analysis algorithm for log-to-log statement matching is efficient and ready for use. The goal of [24] is to improve the accuracy of creating program slices using stack trace information and filtering possible execution paths.

3.6 Future research directions

The relative lack of research in this area indicates that the field is open for innovation. One gap we noticed within the existing work is that utilizing stack traces and using other logging output have been analyzed separately. General logs and stack traces have both

been shown to improve program analysis, especially in program slicing and fault localization. Using both sources of information could create an analyzer that is more general than one that solely uses stack traces, and more accurate than one that only uses general logging output when stack traces are available.

We also believe external sources could be utilized to a greater degree. For example, in one instance, logging statements were linked to bug report issues [15]. The proposed analysis was to track which logs (and bug reports) were impacted by changes in the source code. We believe that if logs from widely-used third party libraries could be collected, they could be linked to publicly known issues with that library to automatically find solutions to common problems. For example, if a GitHub issue or Stack Overflow question is opened for an open source project describing an error associated with a series of log lines, these logs could be compared for similarity with user-generated logs, automatically recommending solutions to encountered problems

Improving the quality of log templates is another area of interest. Xu et al. mention the problem of logging objects using a `toString` method in object-oriented languages [22]. Their solution is to generate a log template that matches the `toString` method of the object passed into the log, as well as up to 100 descendant subclasses of that class. Not only does this cause an exponential increase in the number of templates, the authors mention that in certain cases, more subclasses than this may need to be considered. An alternative solution may be needed.

3.7 Discussion

To answer RQ1 (methods of extracting and mapping log information to source), we found the primary method of extracting and mapping information from logs to source code is regular expression templates. These templates contain the static portion of the log message as well as the matching interpolated variable values [2, 5, 8, 9, 12, 14, 15, 22, 23, 25]. To aid in extraction beyond simply matching to a single location in source code, we found log clustering methods. Clustering can be done by edit distance between logs [8], a thread or request identifier [2, 5], and by finding request-specific values in logs using data flow analysis [25].

Table 3: Extraction/mapping methods^{RQ1}

Method	References	Total
Regular expression templates	[2, 5, 8, 9, 12, 14, 15, 22, 23, 25]	10
Edit distance clustering	[8]	1
Request clustering	[2, 5, 25]	3

For RQ2 (techniques improved by log-to-source mapping), we found program slicing was frequently addressed. Matching a stack trace to locations in the code can be used to generate a program slice [10, 16, 17, 24], and existing program slices can be further narrowed down using the locations of logging statements [2, 5, 23, 24]. For certain kinds of logging runtimes, an entire program slice can be rebuilt from the logs alone [1]. Another technique enhanced by log matching was data flow analysis. In the presence of log-to-source matching, data flow analysis can be used to recreate a sequence of events [25] or in backtracking from an exception's throw point to its root cause [10, 16].

Table 4: Techniques improved by log-source mapping^{RQ2}

Technique	References	Total
Program slicing		**9**
Program slicing (by stack trace)	[10, 16, 17, 24]	4
Program slicing (by logs)	[1, 2, 5, 23, 24]	5
Data flow analysis		**3**
Data flow analysis (backtracking)	[10, 16]	2
Data flow analysis (sequence reconstruction)	[25]	1

In addressing RQ3 (problems addressed by log-to-source mapping), the main goal we found was that of fault localization, using stack traces or general logs to narrow down a program slice to the location of the fault [1, 10, 16, 19, 23, 24]. Analyzing log sequences was also used for anomaly detection [2, 9, 22]. Performance analysis can be performed using the timestamps inherent in many logs [8, 25]. Other goals include estimating code coverage [5], bug report change detection [15], and improving another technique, such as improving program slice accuracy [24], call graph expansion [17], or simply efficient log-to-source matching itself [14].

Table 5: Problems addressed by log-source mapping^{RQ3}

Problem	References	Total
Fault localization	[1, 10, 16, 19, 23, 24]	6
Anomaly detection	[2, 9, 22]	3
Performance analysis	[8, 25]	2
Code coverage	[5]	1
Bug report change detection	[15]	1
Program slice accuracy	[24]	1
Call graph expansion	[17]	1
Improved log-to-source matching	[14]	1

For RQ4 (external sources used), three external sources were found to be combined along with logs and source code. Stack Overflow and GitHub issues were mined for stack traces similar to the one encountered locally [17], and bug reports were combined with changes to logging statements to detect when they may be affected by code changes [15].

Table 6: External sources used^{RQ3}

Source	References	Total
Stack Overflow	[17]	1
GitHub	[17]	1
Bug reports	[15]	1

For RQ5 (future research directinos), we found that an approach that combined analyzing stack traces and general logs together to improve program slicing has not yet been addressed. There are also open issues in more fully utilizing external sources in the code analysis, as well as in generating log templates that are more comprehensive in the presence of complex logging parameters.

4 THREATS TO VALIDITY

The primary threat to the validity of our work is omitting relevant research from our review. To ensure we found all related work in our initial search, we designed our query to be as broad as possible. To reduce the threat of dismissing a relevant work, we had multiple researchers review each search result, beginning with a permissive analysis of the works' title, abstract, and keywords, followed by a more scrutinous examination of the full text.

Another concern is that in our study, we employed four major research databases, namely ACM Digital Library, IEEE Xplore, SpringerLink, and ScienceDirect. Potentially, more papers can be published by other publishers, which we did not include.

5 CONCLUSION

Program logs contain a wealth of information about how a program is behaving. Harnessing this information through automated analysis and connecting it to source code locations has the potential to improve developers' abilities to find bugs, detect anomalies, and analyze the performance of their code. In this study, we selected 16 works out of an initial search of 588 results in an effort to assess the current literature on the techniques and uses of mapping information from logs to source code.

We examined the works to determine the methods they used in extracting and mapping information from logs, as well as the techniques this mapping was meant to improve. Next, we identified the problems these works addressed using these techniques. We discussed open research areas including an approach that combines logs and stack traces, the need for improved log templates, and the potential for inclusion of data from external sources.

Our long term goal is an automated method for software architecture reconstruction of distributed systems. Towards this end, we have performed other studies laying the foundation for this goal [3, 4, 6, 18]. The future work for this study will contribute by using logs to gain insight into a system's architecture.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1854049 and Red Hat, Inc.

REFERENCES

[1] Chang ai Sun, Yufeng Ran, Caiyun Zheng, Huai Liu, Dave Towey, and Xiangyu Zhang. 2018. Fault localisation for WS-BPEL programs based on predicate switching and program slicing. *Journal of Systems and Software* 135 (2018), 191 – 204. https://doi.org/10.1016/j.jss.2017.10.030

[2] Liang Bao, Qian Li, Peiyao Lu, Jie Lu, Tongxiao Ruan, and Ke Zhang. 2018. Execution anomaly detection in large-scale systems through console log analysis. *Journal of Systems and Software* 143 (2018), 172 – 186. https://doi.org/10.1016/j.jss.2018.05.016

[3] Tomas Cerny, Jan Svacina, Dipta Das, Vincent Bushong, Miroslav Bures, Pavel Tisnovsky, Karel Frajtak, Dongwan Shin, and Jun Huang. 2020. On Code Analysis Opportunities and Challenges for Enterprise Systems and Microservices. *IEEE Access* (2020), 1–22. https://doi.org/10.1109/ACCESS.2020.3019985

[4] Tomas Cerny, Andrew Walker, Vincent Bushong, Dipta Das, Karel Frajtak, Miroslav Bures, and Pavel Tisnovsky. 2020. Mapping Study on Constraint Consistency Checking in Distributed Enterprise Systems. In *International Conference on Research in Adaptive and Convergent Systems(RACS '20) (RACS '20)*. ACM, New York, NY, USA, 1–8. https://doi.org/10.1145/3400286.34182571

[5] B. Chen, J. Song, P. Xu, X. Hu, and Z. M. J. Jiang. 2018. An Automated Approach to Estimating Code Coverage Measures via Execution Logs. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 305–316. https://doi.org/10.1145/3238147.3238214

[6] Dipta Das, Micah Schiewe, Elizabeth Brighton, Mark Fuller, Tomas Cerny, Miroslav Bures, Karel Frajtak, Dongwan Shin, and Pavel Tisnovsky. 2020. Failure Prediction by Utilizing Log Analysis: A Systematic Mapping Study. In *International Conference on Research in Adaptive and Convergent Systems(RACS '20) (RACS '20)*. ACM, New York, NY, USA, 1–7. https://doi.org/10.1145/3400286.3418263

[7] Diana El-Masri, Fabio Petrillo, Yann-Gaël Guéhéneuc, Abdelwahab Hamou-Lhadj, and Anas Bouziane. 2020. A systematic literature review on automated log abstraction techniques. *Information and Software Technology* 122 (2020), 106276. https://doi.org/10.1016/j.infsof.2020.106276

[8] Q. Fu, J. Lou, Y. Wang, and J. Li. 2009. Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis. In *2009 Ninth IEEE International Conference on Data Mining*. 149–158. https://doi.org/10.1109/ICDM.2009.60 ISSN: 2374-8486.

[9] Saeed Ghanbari, Ali B. Hashemi, and Cristiana Amza. 2014. Stage-aware anomaly detection through tracking log points. In *Proceedings of the 15th International Middleware Conference on - Middleware '14*. ACM Press. https://doi.org/10.1145/2663165.2663319

[10] S. Jiang, H. Zhang, Q. Wang, and Y. Zhang. 2010. A Debugging Approach for Java Runtime Exceptions Based on Program Slicing and Stack Traces. In *2010 10th International Conference on Quality Software*. 393–398. https://doi.org/10.1109/QSIC.2010.23 ISSN: 2332-662X.

[11] Meenakshi, A. C. Ramachandra, and Subhrajit Bhattacharya. 2020. Literature Survey on Log-Based Anomaly Detection Framework in Cloud. In *Computational Intelligence in Pattern Recognition*, Asit Kumar Das, Janmenjoy Nayak, Bighnaraj Naik, Soumi Dutta, and Danilo Pelusi (Eds.). Springer Singapore, Singapore, 143–153.

[12] Antonio Pecchia, Marcello Cinque, Gabriella Carrozza, and Domenico Cotroneo. 2015. Industry Practices and Event Logging: Assessment of a Critical Software Development Process. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2 (ICSE '15)*. IEEE Press, 169–178. https://dl.acm.org/doi/abs/10.5555/2819009.2819035

[13] Kai Petersen, Sairam Vakkalanka, and Ludwik Kuzniarz. 2015. Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and Software Technology* 64, Supplement C (2015), 1 – 18. https://doi.org/10.1016/j.infsof.2015.03.007

[14] Daan Schipper, Maurício Aniche, and Arie van Deursen. 2019. Tracing Back Log Data to Its Log Statement: From Research to Practice. In *Proceedings of the 16th International Conference on Mining Software Repositories (MSR '19)*. IEEE Press, 545–549. https://doi.org/10.1109/MSR.2019.00081

[15] W. Shang. 2012. Bridging the divide between software developers and operators using logs. In *2012 34th International Conference on Software Engineering (ICSE)*. 1583–1586. https://dl.acm.org/doi/10.5555/2337223.2337490

[16] Saurabh Sinha, Hina Shah, Carsten Görg, Shujuan Jiang, Mijung Kim, and Mary Jean Harrold. 2009. Fault Localization and Repair for Java Runtime Exceptions. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis (ISSTA '09)*. Association for Computing Machinery, New York, NY, USA, 153–164. https://doi.org/10.1145/1572272.1572291

[17] L. Sui, J. Dietrich, and A. Tahir. 2017. On the Use of Mined Stack Traces to Improve the Soundness of Statically Constructed Call Graphs. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. 672–676. https://ieeexplore.ieee.org/document/8306000

[18] Jan Svacina, Jackson Raffety, Connor Woodahl, Stone Brooklynn, Tomas Cerny, Miroslav Bures, Karel Frajtak, Dongwan Shin, and Pavel Tisnovsky. 2020. On Vulnerability and Security Log analysis: A Systematic Literature Review on Recent Trends. In *International Conference on Research in Adaptive and Convergent Systems(RACS '20) (RACS '20)*. ACM, New York, NY, USA, 1–6. https://doi.org/10.1145/3400286.3418261

[19] C. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, and H. Mei. 2014. Boosting Bug-Report-Oriented Fault Localization with Segmentation and Stack-Trace Analysis. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 181–190. https://doi.org/10.1109/ICSME.2014.40

[20] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740.

[21] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. 2005. A Brief Survey of Program Slicing. *SIGSOFT Softw. Eng. Notes* 30, 2 (March 2005), 1–36. https://doi.org/10.1145/1050849.1050865

[22] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. 2009. Detecting Large-Scale System Problems by Mining Console Logs. In *Proceedings*

of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09). Association for Computing Machinery, New York, NY, USA, 117–132. https://doi.org/10.1145/1629575.1629587

[23] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. 2010. SherLog: Error Diagnosis by Connecting Clues from Run-Time Logs. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1736020.1736038

[24] H. Zhang, S. Jiang, and Rong Jin. 2011. An improved static program slicing algorithm using stack trace. In *2011 IEEE 2nd International Conference on Software Engineering and Service Science*. 563–567. https://doi.org/10.1109/ICSESS.2011.5982378 ISSN: 2327-0594.

[25] Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan Ullah, Yu Luo, Ding Yuan, and Michael Stumm. 2014. Lprof: A Non-Intrusive Request Flow Profiler for Distributed Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, USA, 629–644. https://dl.acm.org/doi/10.5555/2685048.2685099