2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)

Tracing Back Log Data to its Log Statement: From Research to Practice

Daan Schipper
Adyen N.V.
Amsterdam, The Netherlands
daan.schipper@adyen.com

Maurício Aniche
Software Engineering Research Group
Delft University of Technology
Delft, The Netherlands
m.f.aniche@tudelft.nl

Arie van Deursen
Software Engineering Research Group
Delft University of Technology
Delft, The Netherlands
arie.vandeursen@tudelft.nl

***Abstract*—Logs are widely used as a source of information to understand the activity of computer systems and to monitor their health and stability. However, most log analysis techniques require the link between the log messages in the raw log file and the log statements in the source code that produce them. Several solutions have been proposed to solve this non-trivial challenge, of which the approach based on static analysis reaches the highest accuracy. We, at Adyen, implemented the state-of-the-art research on log parsing in our logging environment and evaluated their accuracy and performance. Our results show that, with some adaptation, the current static analysis techniques are highly efficient and performant. In other words, ready for use.**

***Index Terms*—software engineering, runtime monitoring, log parsing.**

I. Introduction

Logs record runtime information of computer systems and produce timestamped documentation of events, states and interactions of components. The information gained from logging is used to perform root cause analysis on identified problems, which consists mostly of manual labour. Overall, a log entry is produced by a log printing statement in a system program's source code. Techniques have been developed to relieve this manual work and to take advantage of the rich information present in logs in an automated manner, such as process mining [8], [12], [25], anomaly detection [6], [10], [11], [28], [27], fault localisation [26], [32], invariant inference [7], performance diagnosis [15], [20], [22], [31], online trace checking [5], and behaviour analysis [21], [29].

In practice, as soon as developers learn something from these log analysis techniques, they often want to go back to the log statement in the source code that produced the log message they just analysed. However, tracing back log messages to their origin is a non-trivial challenge in large-scale systems. While frameworks like Log4j [3] enable developers to print the class name and line number of log statements together with the log message, collecting this information in a production environment at every log statement comes with a loss of performance. Behind the scenes, Log4j collects the log statement line by throwing an exception and capturing the generated stack trace.[1]

[1]Log4j's developers have experimented with other alternatives, but so far, this is the most efficient way. See https://issues.apache.org/jira/browse/LOG4J2-1029).

As there is no direct connection between log messages and source code in the produced (raw) log data, the link must be created afterwards. Previous works propose several approaches based on clustering [10], [23], heuristics [17], [24], longest common sequence method [9], textual similarities [14], evolutionary search [18], and static analysis [28] to solve this challenge.

We at Adyen, a payment service provider operating globally and providing over 250 different payment methods, decided to derive an approach based on the state-of-the-art research and apply it in our logging systems. We evaluate our implementation on a dataset consisting of 100,000 log lines, taken directly from our production servers. Our results show that 97.6% of the links were correctly determined (CI=5%, CL=95%). As a consequence, we believe that state-of-the-art research is ready for the real world.

This paper makes the following contributions:

1) The description of the architecture as well as the challenges we faced to implement state-of-the-art research on linking log data to their original log statement at Adyen, a large-scale software system.
2) Empirical evidence that Xu et al.'s [28] approach to link log lines to their original log statements works effectively in an industry setting.

II. Related work

Typically a log message consists of a constant part, which remains the same for every event occurrence, and a variable part containing dynamic information, which is determined during runtime. The goal of log parsing is to separate the constant and variable parts within a log message. As parsing is the basis for many log analysis techniques, log parsing is an active research area, as shown in the introduction.

He et al. [13] performed an evaluation study on the most popular clustering-based methods and found, despite achieving high accuracy, that SLCT [24] and IPLoM [17] do not scale well with the volume of logs, since the clusters are constructed according to the difference in the messages. Furthermore, offline log parsing methods are limited by the memory of a single computer. Therefore, He et al. [14] propose an online method that parses raw log messages in a streaming manner, outperforming previous methods [9], [10], [19], [17]. Another finding of clustering based methods is that the overall accuracy is improved when log messages are preprocessed with some domain knowledge-based rules to remove obvious numerical parameters, such as numbers, memory and IP addresses [13]. Although beneficial to the effectiveness of the log parsing, this is a manual process. Messaoudi et al. [18] capture the template of a message by applying an evolutionary algorithm. This first of a kind approach significantly outperforming other approaches ([14], [17]).

However, approaches based on static analysis have an additional source of information available, the source code itself. Templates are extracted from the logging statements which are then used to match log messages with. This extra knowledge additionally allows the techniques to be completely automated, thus eliminating the need for manual work. Examples of such an approach can be found in Xu et al. [28] and Zhao et al. [30], where authors parse the source code, extract regular expression templates out of the log statements, and match them to the log messages they observe in their log systems.

III. From research to practice: our approach

The overview of the approach is shown in Figure 1, where a square represents a process, and a hexagon represents input or output. All these steps are done automatically by our tool (that we will make available at https://github.com/SERG-Delft/msr19-logs). We start by identifying log statements in the source code, for which we traverse the abstract syntax tree (AST), and analyze nodes related to log statements. Next, we extract a template from the statement along with its severity level and class name. We construct a template in the form of a regular expression that matches all possible log messages produced by it. We then enrich the templates with type analysis information such as the textual representation of objects and type hierarchies to make the templates more precise. To make the templates easily searchable, we conclude this phase by creating an index of the templates. With this template database at hand we then find, for each log message that comes to our production systems, the regular expressions that match, and select the one that has the highest similarity to the constant part of the log message. In the following, we describe each part of our approach in detail.

a) Identify Log Statements: We parse the source code to an AST to programmatically search the source code for statements corresponding to log statements. Our implementation uses JavaParser [2], a simple and lightweight AST library.

In order to analyze the log statements, we iterate over the individual nodes to find those that represent log statements. Previous work by Zhao et al. [30] identifies log statements by searching for method calls corresponding to the standard logging methods, such as those defined by Log4j [3] (e.g., `log.info()` and `log.warn()`). In practice we had to extend this; companies like Adyen create their own logging libraries suited for their needs (e.g., to automatically log transaction IDs).

b) Create Template: We construct templates based on the arguments of the log statement; a template that would match any message generated by it. These templates are then used to match the log messages with, providing the trace back to the log statement in the source code. Note that a more precise template will more accurately match the log messages. However, in practice, the arguments of a log statement have restrictions: developers can construct the argument in every way imaginable, as long as it follows the language specification (e.g., messages that contain integers, doubles, Strings, etc). Even non-primitive objects with a custom textual representation can be included. In other words, arguments can vary from a simple literal expression to an interpolation of primitive types together with objects, which all are converted to a single line of text during runtime. We apply static analysis to create templates based on the arguments of the logging statement. Any literal expression is directly copied to the regular expression, while runtime variables are replaced by wildcards (often enhanced by the type of the variable).

Suppose we have the following log statement with mixed expressions: `log.info("average = " + avg)`, where `"average = "` is a String, `avg` is of type double, and everything is concatenated together, forming a single String.

The AST, in a simplified view, contains three types of nodes: one representing the String, one representing the concatenation (+), and one representing the double value. Our approach recognizes the first literal string and copy it directly to the template; then it recognizes the double variable, and it generates a wildcard for double numbers. The final regular expression for that log statement is then `average = .*[double]`.

c) Enrich templates: While capturing the type of primitive variables and generating proper regular expressions for them is trivial, finding a precise regular expression for an object requires more work. In Java, objects are transformed to a String through the `toString()` method, which exists in any Java object. The `toString()` method is often overridden by developers, so that objects print useful information.

Following the approach of Xu et al. [28], whenever we notice an object as an argument in a log statement, we try to infer its regular expression based on its `toString()` implementation. We apply it recursively, as an object's `toString()` method can also print another object. If no `toString()` implementation is found, we replace the object with a generic wildcard "`.*`". Given that the real type of the object is only known at runtime (i.e., polymorphism), we create one template for each sub-class of the argument's type in the log statement. Each template contains a regular expression extracted from a sub-class implementation of the `toString()` method.

d) Create Index: Scanning the entire (extensive) template database to find a match for each log message is unfeasible. To solve the problem, we compile the constant part of all templates into a reverse index [4] to make them searchable. Then we query this index to retrieve a set of similar templates based on TF-IDF [16] of the constant part, which has a higher possibility of matching. Implementation-wise, we use Apache Lucene [1] to index the templates, following the approach by Xu et al. [28].

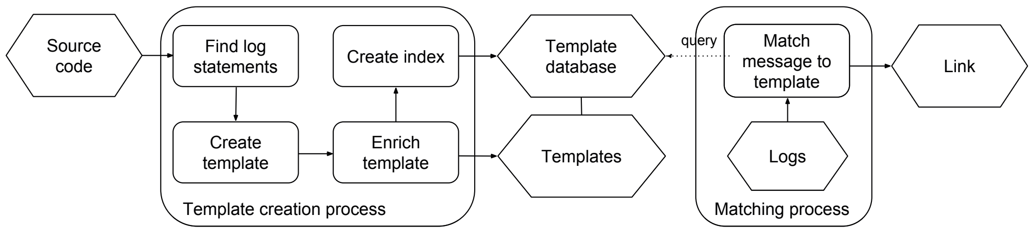

Fig. 1. Overview of the approach.

e) Match messages to relevant templates: With the indexed database at hand, our system is ready to provide the log statement that originated any given log message we receive in our production environment. This happens in four steps: 1) we first filter out templates that do not belong to the class that originated the log statement (logging the class name is a cheap operation in Log4j), 2) we query the index for the templates that are more likely to match based on the content of the message and receive a relevance ranked list, 3) we then evaluate whether the corresponding regular expression matches the message, and 4) we return the highest ranking template where the log message matches the regular expression.

IV. Empirical Study

In this section, we evaluate the effectiveness of linking log data to its log statement in the source code with static analysis regarding accuracy and performance. To that aim, we propose the following research questions:

RQ_1 What is the accuracy of the approach when dealing with extensive log data?

RQ_2 What is the performance of our approach?

A. Studied Sample

We evaluate the effectiveness of the approach in real life conditions and will use log data taken directly from the production servers of Adyen. The logs are produced by a software system which has the purpose of processing payments from all over the world. At the moment of writing the codebase consists of millions of lines of code[2], written by over 150 developers. Of those lines, approximately tens of thousands of log statements generate log messages. The percentage of lines of log statements is on the lower end compared to that of other systems, normally about 1%-5% [28], as developers try to be as efficient as possible in their logging.

The dataset used to evaluate the approach consists of logs produced during normal operations, and no filtering was applied to the messages. We obtained 100,000 messages from a normal (i.e. non-holiday) weekday.

B. Methodology

To answer RQ_1, we match the log messages from the dataset to the source code and evaluate whether the link provided by the approach is indeed correct. In the 100,000 messages in our sample, our approach generates 676 links (i.e., connected the log messages to 676 different log statements in the source code). To identify whether the link was correctly made, we manually analyze a statistically significant sample of 245 links, with a confidence level of 95% and confidence interval of 5%. For each link, we select one random matched message to evaluate the link. More specifically, we check whether the statement could have produced the message by taking into account the structure of the message, the severity, and the accompanying class name. Furthermore, we evaluate the log messages of which the approach provides a link to an incorrect log statement. We manually inspect these log messages to inspect why the approach was unable to provide a correct link. We explain and show the underlying cause for the misidentified messages.

To answer RQ_2, we apply the approach ten times on the dataset and measure the performance to eliminate any bias of external programs influencing the execution time. The machine used has two cores @ 3.1GHz from a Intel Core i7 CPU, and 16GB RAM. We analyze the execution time per individual step of the creation process and report the mean execution time of the ten runs. Finally, we link the log messages from the dataset to the templates and analyze the execution time needed per individual log message. We evaluate the distribution of the execution time according to the mean, quartiles, and quantiles.

C. Results

1) RQ_1. What is the accuracy of the approach when dealing with extensive log data?: Overall, the approach achieves 97.6% accuracy (239 out of 245 analyzed log statements) on tracing back the origin of log data to its log statement in the source code. All but two log messages have been linked to log statements in the source code. These two failures can be explained by the fact that they have been both produced by the same log statement, which logs a message that is too large (approximately 17k characters) to be handled by our logging facilities. This log statement has already been modified in a future release.

Out of the 676 log statement identified as the source of all log messages, six of those have proven to be incorrect. The misidentifications occur due to the following underlying causes:

JSON-based logs: Before querying the index, we strip the JSON data out of the message since we consider this to be variable information. Therefore, when the message consists of JSON data only, the resulting query is an empty string. We

[2]Exact number is omitted due to confidentiality reasons.

Step	Time (minutes)	Percentage
Finding log statements	37:25	92.1%
Creating template	00:32	1.3%
Enriching template	02:27	5.9%
Creating index	00:16	0.7%
Total	40:42	100%

TABLE I
MEAN EXECUTION TIME PER STEP OF THE APPROACH (AVERAGE OF TEN RUNS)

Statistic	Time (ms)	Cumulative Time (%)
Minimum (0%)	3	0,00%
1st quartile (25%)	4	1.97%
Median (50%)	4	4.27%
3rd quartile (75%)	6	6.98%
90th quantile	6	9.61%
95th quantile	13	12.39%
98th quantile	59	15.22%
99th quantile	132	17.43%
Maximum (100%)	139922	100.00%

TABLE II
EXECUTION TIME STATISTICS, PER LOG MESSAGE

also observed our tool failing due to bad JSON stripping. In future versions, we should propose better ways to deal with JSON-only log messages.

Unknown logging method: Adyen also uses custom-made logging methods to construct logs in specific formats. Since our implementation was unaware of them, no templates were created for log statements using these methods. However, since the implementation is easily configurable, these logging methods can be added in future versions.

Inaccuracies in the creation process of templates: The approach uses static analysis to create templates of log statements that predict what the messages will look like at runtime. Unfortunately, not all predictions are completely accurate. The inaccuracy often happens when the message is constructed outside the log statement itself, e.g., `String logMsg = "..." + variable; log.info(logMsg);`.

2) RQ_2. What is the performance of our approach?: Table I shows the execution time per individual process of the template creation process at Adyen's codebase. The total execution time took on average approximately 40 minutes across all ten runs. Most of this time is spent searching the codebase for method calls that correspond to log statements, on average 92,1%. The actual execution time of creating and indexing templates only takes 3 minutes and 16 seconds, around 7,9% of the total execution time. The results are in line with a similar approach by Zhao et al. [30], whose static analysis takes less than two minutes to run for systems ranging from 100.000 to 300.000 lines of code.

In the matching process, the median execution time to process a single log message is only 4 milliseconds. Table II details the statistics, which shows that for 99% of the log messages are processed in under 132ms.

However, the total execution time to process all log messages is 01h13m19.969, which is much higher than expected. We observed that the maximum time to process a single log message is exceptionally high: 02m19.922, or 35,000 times the median. The approach derived an incorrect template from the statement in question. Two wildcards are missing from the template, which results in a template not able to match the log messages produced by the statement. This results that *all* templates are evaluated to try to find a matching log statement for this specific message, thus causing such a long execution time. Therefore, it is most important to create as precise as possible templates which significantly reduces the time needed to match the log messages. Given this finding, we implemented a fail-safe mechanism in our approach so that it gives up if the search for a matching template takes too long.

Threats to Validity. Our data sample is collected from a single random day. This sample exercises 676 different log statements of our system. After manually analyzing these log statements, we observe that log statements were quite diverse in terms of the number of literals, variables, variable types, and importance to our system. Therefore, while a replication would strengthen the validity of our findings, our results already give us a high degree of certainty and interesting insights.

V. CONCLUSION

This paper presented our implementation of the state-of-the-art research on log parsing and its evaluation for Adyen's large-scale software system. Our results show that our approach (and consequently, the state-of-the-art research) is highly effective in tracing back the origin of log data to its log statement in the source code, on average in 4 milliseconds per log message and with an accuracy of 97.6% (CL=95%, CI=5%). We thus believe that state-of-the-art research is ready for the real world.

We also learned that, for such state-of-the-art research to be applied in companies, implementations have to be:

- **Non-intrusive:** Companies will not afford changes in their existing source code for any technique to work. It also can not have any negative impact on the performance of the software. In our implementation, our parser makes sure it understands the specificities of the source code, and all the parsing, template enriching, and matching process happen on separate machines.
- **Adaptable:** Companies have their logging utilities, and tooling needs to adapt to it. Our implementation provides an easy way for developers to add support for different log libraries.
- **Extendable:** Complex software makes use of operations other than simple concatenation to build a log statement. Our tool currently supports log messages consisting of any combination of primitive objects, non-primitive objects (including class polymorphism), formatted strings and referenced strings for Java. In the future, we plan to provide better JSON support as well as a more complex log building code.

ACKNOWLEDGMENTS

This work is partially funded by the research projects STAMP ICT-16-10 No. 731529 (EC H2020) and NWO EW MIPL No. 628.008.003.

References

[1] Apache lucene - apache lucene core. http://lucene.apache.org/core/. Accessed: 2018-08-31.

[2] Javaparser - for processing java code. http://javaparser.org/. Accessed: 2018-08-27.

[3] Log4j - apache log4j 2. https://logging.apache.org/log4j/2.x/. Accessed: 2018-08-20.

[4] R. Baeza-Yates, B. Ribeiro-Neto, et al. *Modern information retrieval*, volume 463. ACM press New York, 1999.

[5] D. Basin, G. Caronni, S. Ereth, M. Harvan, F. Klaedtke, and H. Mantel. Scalable offline monitoring. In *International Conference on Runtime Verification*, pages 31–47. Springer, 2014.

[6] C. Bertero, M. Roy, C. Sauvanaud, and G. Trédan. Experience report: Log mining using natural language processing and application to anomaly detection. In *Software Reliability Engineering (ISSRE), 2017 IEEE 28th International Symposium on*, pages 351–360. IEEE, 2017.

[7] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 267–277. ACM, 2011.

[8] H.-J. Cheng and A. Kumar. Process mining on noisy logs—can log sanitization help to improve performance? *Decision Support Systems*, 79:138–149, 2015.

[9] M. Du and F. Li. Spell: Streaming parsing of system event logs. In *Data Mining (ICDM), 2016 IEEE 16th International Conference on*, pages 859–864. IEEE, 2016.

[10] Q. Fu, J.-G. Lou, Y. Wang, and J. Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*, pages 149–158. IEEE, 2009.

[11] M. Goldstein, D. Raz, and I. Segall. Experience report: Log-based behavioral differencing. In *Software Reliability Engineering (ISSRE), 2017 IEEE 28th International Symposium on*, pages 282–293. IEEE, 2017.

[12] C. Günther and W. Aalst, van der. Fuzzy mining - adaptive process simplification based on multi-perspective metrics. In G. Alonso, P. Dadam, and M. Rosemann, editors, *Proceedings of the 5th International Conference on Business Process Management (BPM 2007) 24-28 September 2007, Brisbane, Australia*, Lecture Notes in Computer Science, pages 328–343, Germany, 2007. Springer.

[13] P. He, J. Zhu, S. He, J. Li, and M. R. Lyu. An evaluation study on log parsing and its use in log mining. In *Dependable Systems and Networks (DSN), 2016 46th Annual IEEE/IFIP International Conference on*, pages 654–661. IEEE, 2016.

[14] P. He, J. Zhu, Z. Zheng, and M. R. Lyu. Drain: An online log parsing approach with fixed depth tree. In *Web Services (ICWS), 2017 IEEE International Conference on*, pages 33–40. IEEE, 2017.

[15] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora. Automated performance analysis of load tests. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 125–134. IEEE, 2009.

[16] J. Leskovec, A. Rajaraman, and J. D. Ullman. *Mining of massive datasets*. Cambridge university press, 2014.

[17] A. Makanju, A. N. Zincir-Heywood, and E. E. Milios. A light-weight algorithm for message type extraction in system application logs. *IEEE Transactions on Knowledge and Data Engineering*, 24(11):1921–1936, 2012.

[18] S. Messaoudi, A. Panichella, D. Bianculli, L. Briand, and R. Sasnauskas. A search-based approach for accurate identification of log message formats. In *International Conference on Program Comprehension (ICPC), 2018 IEEE/ACM International Conference on*. IEEE/ACM, 2018.

[19] M. Mizutani. Incremental mining of system log format. In *Services Computing (SCC), 2013 IEEE International Conference on*, pages 595–602. IEEE, 2013.

[20] K. Nagaraj, C. Killian, and J. Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 26–26. USENIX Association, 2012.

[21] N. Poggi, V. Muthusamy, D. Carrera, and R. Khalaf. Business process mining from e-commerce web logs. In *Business process management*, pages 65–80. Springer, 2013.

[22] M. D. Syer, Z. M. Jiang, M. Nagappan, A. E. Hassan, M. Nasser, and P. Flora. Leveraging performance counters and execution logs to diagnose memory-related performance issues. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 110–119. IEEE, 2013.

[23] L. Tang, T. Li, and C.-S. Perng. Logsig: Generating system events from raw textual logs. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 785–794. ACM, 2011.

[24] R. Vaarandi. A data clustering algorithm for mining patterns from event logs. In *IP Operations & Management, 2003.(IPOM 2003). 3rd IEEE Workshop on*, pages 119–126. IEEE, 2003.

[25] J. M. E. Van der Werf, B. F. van Dongen, C. A. Hurkens, and A. Serebrenik. Process discovery using integer linear programming. In *International conference on applications and theory of petri nets*, pages 368–387. Springer, 2008.

[26] W. E. Wong, V. Debroy, R. Golden, X. Xu, and B. Thuraisingham. Effective software fault localization using an rbf neural network. *IEEE Transactions on Reliability*, 61(1):149–169, 2012.

[27] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan. Online system problem detection by mining patterns of console logs. In *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*, pages 588–597. IEEE, 2009.

[28] 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*, pages 117–132. ACM, 2009.

[29] X. Yu, M. Li, I. Paik, and K. H. Ryu. Prediction of web user behavior by discovering temporal relational rules from web log data. In *International Conference on Database and Expert Systems Applications*, pages 31–38. Springer, 2012.

[30] X. Zhao, Y. Zhang, D. Lion, M. F. Ullah, Y. Luo, D. Yuan, and M. Stumm. lprof: A non-intrusive request flow profiler for distributed systems. In *OSDI*, volume 14, pages 629–644, 2014.

[31] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang. Learning to log: Helping developers make informed logging decisions. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 1, pages 415–425. IEEE, 2015.

[32] D.-Q. Zou, H. Qin, and H. Jin. Uilog: Improving log-based fault diagnosis by log analysis. *Journal of Computer Science and Technology*, 31(5):1038–1052, 2016.