Towards a Classification of Log Parsing Errors

Issam Sedki
ECE, Concordia University
Montreal, Canada
issam.sedki@concordia.ca

Abdelwahab Hamou-Lhadj
ECE, Concordia University
Montreal, Canada
wahab.hamou-lhadj@concordia.ca

Otmane Ait-Mohamed
ECE, Concordia University
Montreal, Canada
otmane.aitmohamed@concordia.ca

Naser Ezzati-Jivan
Brock University
St. Catharines, Canada
nezzati@brocku.ca

***Abstract*—Log parsing is used to extract structures from unstructured log data. It is a key enabler for many software engineering tasks including debugging, fault diagnosis, and anomaly detection. In recent years, we have seen an increase in the number of log parsing techniques and tools. The accuracy of these tools varies significantly. To improve log parsing tools, we need to understand the type of parsing errors they make, which is the purpose of this early research track paper. We achieve this by examining errors of four leading log parsing tools when applied to the parsing of four log datasets generated from various systems. Based on this analysis, we suggest a preliminary classification of log parsing errors, which contains nine categories of errors. We believe that this classification is a good starting point for improving the accuracy of log parsing tools, and also defining better logging practices.**

***Index Terms*—Log Parsing, Log analytics, Dynamic analysis, Software logging, AIOps.**

I. Introduction

The analysis of logs can help with many software engineering tasks including debugging [1] [2], detection of data leakage [3], and anomaly detection [4] [5]. Logs are also used for AI for IT Operations [6], a growing field of study related to the maintenance and operations of cloud-based systems. A log file contains a sequence of log events where each event is composed of a header and a message. The header is usually composed of the timestamp, process id, log level, and the logged function. Log headers follow the same format within the same file and are usually easy to parse. The log message is composed of static text and dynamic variables (also called static and dynamic tokens). The structure of log messages varies within the same log file and across logs of different systems, making it challenging to analyze logs.

To address this issue, several log parsing techniques have been proposed in the literature (e.g., Drain [7], Spell [8], ULP [9]). The aim is to extract log templates from log events by clearly distinguishing between the static and dynamic tokens. Figure 1 shows an example a log event (without the header) taken from the Hadoop Distributed File System (HDFS) log file and its corresponding log template where dynamic tokens (`blk_5792489080791696128`, `/10.251.30.6:33145`, and `/10.251.30.6:50010`) are replaced by $\star$. A good parser is the one that effectively

Log Event : `Receiving block blk_579248909 src:/10.251.30.6:335 dest:/10.251.30.6:500`

Template : `Receiving block * src:* dest:*`

Fig. 1. An example of a log event and its corresponding log template

recognizes all templates in a log file. These templates can later be used to parse incoming log events. A naive solution to this problem would be to use regular expressions. The problem is that typical log files may contain thousands of templates [10] [11]. In addition, as the system evolves, developers must constantly update the regular expressions, which is not practical. El-Masri et al. [12] showed that existing parsing tools rely on a panoply of methods including heuristics, pattern mining, natural language processing, clustering, and more. Recent studies [12] [13] compared the accuracy of these tools for parsing 16 log datasets of the LogHub benchmark[1]. The findings show that these tools do not perform consistently when applied to different log files. While their accuracy on some datasets is high, it is low on others, making it difficult for developers to choose which tools to use. To understand the causes behind these inconsistencies, we embarked on studying the type of errors these tools make with the long-term objective to improve existing tools, and to work towards better logging practices that would simplify the parsing process.

To achieve our goal, we selected four leading log parsing tools based on their overall accuracy and applied them to the parsing of four log datasets of the LogHub benchmark. We then analyzed the incorrectly-parsed log events to understand the root causes. We present the results of this analysis in the form of a classification of log parsing errors. To the best of our knowledge, this study is the first to build a classification of log parsing errors. Most related work focuses on improving log parsing algorithms (e.g., [15] [16] [17]).

II. Study Setup

The goal of this preliminary study is to identify and classify the common errors that occur during log parsing. We believe that this classification can be a good starting point for the development of better and more consistent log parsing tools.

[1]https://zenodo.org/record/3227177#.YUqmXtNPFRE

A. Log Parsing Tools

Zhu et al. [13] compared the performance of 13 parsing tools and found that the best performing tools are Drain [7], AEL [14], and Spell [8]. We chose these three tools in this study. In addition, recently, Sedki et al. [9] presented ULP (Universal Log Parser) and showed that it performs better than Drain, AEL, and Spell. We, therefore, decided to include ULP as well. Drain [7] abstracts log messages into event types using a parse tree and uses a similarity metric that compares leaf nodes to event types to identify log groups. AEL (Abstracting Execution Logs) [14] relies on textual similarity. AEL tokenizes log events and assigns them to bins based on the number of terms they contain and then abstracts them into templates. Spell (Streaming Parser for Event Logs using an LCS) [8] converts log messages into event types. It relies on the idea that log messages that are produced by the same logging statement can be assigned a type, which represents their longest common sequence. ULP [9] combines string matching and local frequency analysis to parse large log files.

B. Datasets and Ground Truth

The datasets used in this study consist of log files of the LogHub benchmark [13][2]. The benchmark contains 16 log datasets that are generated from different types of systems, namely distributed systems, supercomputers, operating systems, mobile applications and server applications. Each log dataset of the LogHub benchmark comes with a subset of 2,000 log events that have been parsed manually. The log templates were identified, and each log event out of the 2,000 events was associated with a specific log template. These labelled log datasets are used as ground truth against which we can check the accuracy of log parsing tools. For the purpose of this study, we selected randomly one log dataset from each type of systems to have representative datasets. In total, we have four log datasets as shown in Table I. The table also shows the number of templates for each log dataset.

TABLE I
LOG DATASETS FROM THE LOGHUB BENCHMARK USED IN THIS STUDY

System	Type	Num of Templates
OpenStack	Distributed system	43
Linux	Operating system	118
HealthApp	Mobile application	75
Apache	Server application	6

C. Procedure of Identifying Parsing Errors

We consider a parsing error if a tool generates a template that is different from that of the ground truth. More precisely, we wrote a script to check if the static tokens are correctly identified and are in the same order as the ground truth. We also check that all the dynamic tokens are found and are in the same order as the ground truth. If the same error is repeated multiple times, we count it only once. Note that we ignore missing or excessive special characters to avoid being too restrictive in the comparison. We use each parsing tool to parse the four datasets included in this study. Then, using a script we compare the result of parsing each log event to the ground truth. The incorrectly-parsed log events (i.e., the ones for which the parser generates a template that is different from that of the ground truth) are checked manually to find the root causes. We group similar errors into categories, which form our classification. We also compute the distribution of each type of errors for each parser across the datasets.

III. RESULTS

After analyzing manually thousands of incorrectly-parsed log events, we found a total of 523 parsing errors (all the data used in this study is available on Zenodo[3]). We reviewed each parsing error carefully to understand the causes. We found that these errors can be classified into nine categories that are list in what follows and discuss in more details in the next subsections. Table II shows the distribution of parsing errors for each tool across the four datasets. Table III shows the breakdown of the number of errors found for each dataset. We found that the highest number of errors appears in Linux (231 log parsing errors across the four log parsing tools), followed by OpenStack (151 errors), HealthApp (130 errors). For Apache, we only have 11 errors, mainly because of the low number of templates Apache contains (6 templates).

- C1 - Dynamic tokens with special characters
- C2 - Dynamic tokens with alpha-numerical characters
- C3 - Dynamic tokens with space
- C4 - Dynamic tokens expressed as text only
- C5 - Same type of dynamic token in different formats
- C6 - Dynamic tokens that represent constant values
- C7 - No demarcation between the tokens
- C8 - Static tokens with properties of dynamic tokens
- C9 - Log events differing slightly in their structure

A. Log Parsing Error Classification

1) C1-Dynamic tokens with special characters: Dynamic tokens that contain special characters such as 11.5 and 203.205.147.206:8080@0 are usually hard to parse. This is because most parsers use special characters as delimiters, resulting in splitting dynamic variables with special characters into several tokens. As an example, Drain, Spell, and ULP parse the Openstack log event `memory limit 14.5 MB` into the log template : `memory limit *.* MB` while the correct template is `memory limit * MB`. Based on Table II, this category of errors appears more frequently in Linux and Apache datasets with an average error ratio when using all four tools of 16.82% and 25% respectively. This is most likely due to the high number of dynamic tokens with special characters in these datasets. Note that, unlike other tools, ULP does not make C1 errors when applied to Apache. Understanding the reasons behind this requires future studies that investigate the relationship between the parsing algorithms and the characteristics of the datasets.

[2]https://zenodo.org/record/3227177#.YUqmXtNPFRE

[3]Link to the data used in this paper: https://doi.org/10.5281/zenodo.7623145

TABLE II
ERROR DISTRIBUTION PER DATASET

	Error distribution for Linux (%)									Error distribution for Openstack (%)								
	C1	C2	C3	C4	C5	C6	C7	C8	C9	C1	C2	C3	C4	C5	C6	C7	C8	C9
Drain	18.03	22.95	4.92	3.28	1.64	1.64	40.98	3.28	3.28	14.58	60.42	6.25	0.00	0.00	0.00	6.25	4.17	8.33
Spell	17.86	26.79	7.14	5.36	1.79	3.57	26.79	8.93	1.79	16.67	68.75	6.25	0.00	0.00	0.00	0.00	8.33	0.00
AEL	18.64	23.73	3.39	1.69	1.69	0.00	37.29	10.17	3.39	7.14	46.43	3.57	0.00	0.00	0.00	17.86	21.43	3.57
ULP	12.73	23.64	3.64	10.91	1.82	5.45	34.55	1.82	5.45	14.81	29.63	18.52	0.00	0.00	0.00	14.81	7.41	14.81
Average	16.82	24.28	4.77	5.31	1.73	2.67	**34.90**	6.05	3.48	13.30	**51.31**	8.65	0.00	0.00	0.00	9.73	10.33	6.68

	Error distribution for Apache (%)									Error distribution for HealthApp (%)								
	C1	C2	C3	C4	C5	C6	C7	C8	C9	C1	C2	C3	C4	C5	C6	C7	C8	C9
Drain	33.33	0.00	0.00	0.00	0.00	66.67	0.00	0.00	0.00	2.86	5.71	0.00	0.00	0.00	0.00	74.29	8.57	8.57
Spell	33.33	0.00	0.00	0.00	0.00	66.67	0.00	0.00	0.00	5.71	14.29	0.00	0.00	0.00	2.86	65.71	14.29	14.29
AEL	33.33	0.00	0.00	0.00	0.00	66.67	0.00	0.00	0.00	2.86	8.57	0.00	0.00	0.00	0.00	77.14	14.29	14.29
ULP	0.00	0.00	0.00	0.00	0.00	100.0	0.00	0.00	0.00	0.00	7.69	0.00	0.00	0.00	7.69	53.85	15.38	15.38
Average	25.00	0.00	0.00	0.00	0.00	**75.00**	0.00	0.00	0.00	2.86	9.07	0.00	0.00	0.00	2.64	**67.75**	13.13	13.13

TABLE III
ERRORS FOUND FOR EACH DATASET

	Linux	Openstack	Apache	HealthApp
Drain	61	48	3	35
Spell	56	48	3	41
AEL	59	28	3	41
ULP	55	27	2	13
Total	231	151	11	130

2) C2-Dynamic tokens with alpha-numerical characters: Dynamic tokens that contain a mix of numbers and text (e.g., a489c8f0cda93) are often detected only partially. The parsers usually recognize the numerical parts of the token, but fail to consider the textual part as part of the dynamic tokens. For example, the result of parsing the token `cp-1.slom1.tcloud-pg0.utah.cloudlab.us` is `cp-*.slom*.tcloud-pg*.utah.cloudlab.us` or `insta/_base/a*c*f*cda*` for the variable `a489c8f0cda93`, which is incorrect in both cases. Overall, C2 is the second most predominant error in average. As shown in Table II, C2 errors are considerable high when parsing OpenStack log dataset with an error ratio varying from 29.63% (ULP) to 68.75% (Spell), an average of 51.31%.

3) C3-Dynamic tokens with space: Dynamic tokens that contain spaces (e.g., "/v2/54/servers/ HTTP/1.1") are hard to detect and often times are interpreted as many variables by most log parsing techniques. These tools consider space is often considered as a default separator. The log event from OpenStack `3 DELETE "/v2/54/servers/ HTTP/1.1" status: 204 len: 203 time: 23` is interpreted by Drain as `* "DELETE * HTTP * status: * len: * time: *` while the correct template is `* "DELETE * status: * len: * time: *`. Based on Table II, this error appears mainly with Linux and OpenStack. For OpenStack, 8.65% of the logging errors could be avoided if this error was caught by the parser.

4) C4 - Dynamic variable expressed as text only : This category of errors refers to parsing situations where the dynamic variable is composed of text only, for example, to represent user names (a common case in Linux logs), and is hardly deciphered as dynamic. These variables are often misclassified as static text by a log parser. For example, Spell parses the Linux log event `session opened for user root` into `session opened for user root` while the expected is `session opened for user * .` As shown in Table II, C4 errors appear mainly when parsing the Linux dataset. This is more related to the presence of usernames such as root and admin, commonly found in operating systems.

5) C5 - Same type of a dynamic variable expressed in different formats: This happens when the dynamic token, which refers to the same variable type, is expressed in different formats. For example, for HDFS log, the block id is written in different format `blk_-5140072410813878235`, `blk_4292382298896622412`. Parsing the log events `BLOCK* ask 10.250.18.114:50010 to delete blk_-5140072410813878235`, from HDFS, the block id with "-" was not detected by ULP, whereas the log event `BLOCK* ask 10.251.122.79:50010 to delete blk_8048594464172649365` was detected correctly. The parser did not consider the two variables as identical because they don't have the same structure. As shown in Table II, we only noticed C5 errors in the Linux dataset with a relatively small average error ratio of 1.73%.

6) C6 - Dynamic tokens that represent constant values): Dynamic tokens that have the same value across the entire log file are difficult to parse because most parsers use some sort of frequency-based analysis to recognize dynamic tokens. This often happens when the dynamic token represents a constant in the code. For example, the Apache log event `workerEnv.init() ok /etc/httpd/conf/workers2.properties` refer to a file that contains system properties. The file name is constant. Most log parsers are unable to recognize this variable because its value never changes in the log. C6 errors are present in all the log datasets we analyzed. In average, C6 is the 3rd most common cause of parsing errors (with an average error ratio of 20.05%), slightly behind C2 (20.95%). The problem is worse for Apache with an average error ratio of 75%. Clearly, detecting constant dynamic tokens can significantly improve parsing Apache and other similar log files.

7) C7 - No demarcation between the tokens: We found many cases of static and dynamic tokens that are output into one single token without any clear demarcation. For example, ULP parses the Android log event `Process com.tencent.mobileqq:qzone (pid 12236) has died` as `Process * ( pid * ) has died` whereas the expected template is `Process *:qzone (pid *) has died`. Notice that the static token `qzone` was removed by ULP from the template because it was concatenated to the dynamic token `com.tencent.mobileqq` without having a delimiter. Overall, C7 is the number one cause of log parsing errors (see Table II). C7 causes 67.75 % of the errors resulting from parsing HealthApp dataset. After analyzing HealthApp log events, we noticed that they contain a large number of various delimiters such as comma, space, equality, colon, and so on that seem to mislead the parsers.

8) C8 - Static tokens that have similar properties as dynamic tokens: This error occurs in the presence of static tokens that contain digits and special characters, resembling at a certain degree dynamic tokens. For example, ULP parses the event `For more details see: http://wiki.apache.org/hadoop/NoRouteToHost` into `For more details see: http://*/hadoop/NoRouteToHost` while the correct template is `For more details see: http://wiki.apache.org/hadoop/NoRouteToHost`. Here the URL does not refer to the value of a variable, but it is part of the static text of the log message. C8 errors affect mainly HealthApp and OpenStack log datasets.

9) C9 - Log events differing slightly in their structure: Logs events that use a slightly different structure fall into this category. For example, Spell considers the events `pam unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=` and `pam unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost= user=` as similar because they start with the same tokens. These cases seem to occur when the parser gives more weight to the static tokens at the beginning of the event, assuming that log events that start with the same tokens are identical. C9 errors represent 13.13% of the total number of errors in HealthApp logs.

B. Recommendations

Our analysis shows that most parsing errors can be addressed using the following recommendations.

1) R1: Improving the pre-processing step of a log parser: Log parsers rely on a pre-processing step to identify trivial dynamic tokens (e.g., IP addresses, URLs) using regular expressions. This reliance on regular expressions during pre-processing can have detrimental effects which make up 28.5% of total captured errors in our experiments. For instance, in the case of C2 errors, if the numbers are captured first during pre-processing, it results in complex outcomes like `/var/lib/nova/insta/_base/a*c*f*c*da*` which greatly complicate the log parsing logic. Similarly, when extracting a variable based solely on its format in the case of C5 errors or when substituting a static token like the wiki link `http://wiki.apache.org` with a dynamic variable in the case of C8 errors, similar complications have been noticed.

It is recommended to minimize the use of regular expressions during pre-processing in order to prevent these types of errors. This can be achieved by restricting regular expressions to only the most challenging cases that cannot be detected by the parsers including cases of less common dynamic tokens such as system parameters, configuration numbers, or specific system method names that are mistaken for dynamic variables.

2) R2: Improving tokenization using prior knowledge of the log events: In the log parsing process, the first step is to extract a list of tokens from each log message through tokenization. The study found that 65.51% of the total errors, including C1, C2, C3 and C7 errors, are related to how the tokenization is done. One solution would be to create a list of frequent dividers for each dataset using a prior analysis of a sample of log events. For example, Apache logs make a great use of the special characters "[|=]" as word delimiters. Knowing this in advance can help better tokenize Apache log events. The unique case of the dividers "." and ":" commonly found in IP addresses, decimals, etc. can be handled in a more targeted manner to determine whether or not these characters should be used to separate tokens during the tokenization stage.

3) R3: Improving the formatting of logging statements: To address C3, C7, C8, and C9 errors, it is recommended to improve the formatting of logging statements using tags describing the data being logged. For example, we can add a tag to a file name indicating that it should be recognized by a parser as a file name. Following a standardized way for writing logging statements ensures consistency in interpreting and analyzing the generated log events. Good logging practices should also enhance the quality and readability of log events, making it easier to understand and analyze log files effectively.

IV. Conclusion and Future work

We presented a preliminary classification of log parsing errors. The resulting classification contains nine categories of errors, which range from errors related to the formatting of static and dynamic tokens to errors caused by complex structures of log events. Future work includes extending the current study to larger datasets to verify the generalizability of the results. We should also examine the interplay between parsing errors and the type of systems. By exploring the commonalities between datasets of different systems, such as operating systems, mobile devices, and distributed systems, we could further improve the accuracy of log parsing tools. In addition, we need to conduct further studies to understand how existing parsing algorithms can be improved to reduce these errors. The recommendations presented in this paper are a good starting point. Finally, parsing can be further improved if consistency in writing logging statements is enforced. We can use parsing errors to guide the development of better and more consistent logging practices.

REFERENCES

[1] Barik T., DeLine R., Drucker S., Fisher D., "The bones of the system: A case study of logging and telemetry at Microsoft," in Proc. of the 38th International Conference on Software Engineering Companion (ICSE-C), 2016, pp. 92-101.

[2] Huang P., Guo C., Lorch J. R., Zhou L., Dang Y., "Capturing and enhancing in situ system observability for failure detection," in Proc. of 13th USENIX Symposium on Operating Systems Design and Implementation, 2018, pp. 1-16.

[3] Zhou R., Hamdaqa M., Cai H., Hamou-Lhadj A., "MobiLogLeak: A preliminary study on data leakage caused by poor logging practices," in Proc. of the 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2020, pp. 577-581.

[4] Xu W., Huang L., Fox A., Patterson D., Jordan M. I., "Detecting large-scale system problems by mining console logs," in Proc. of the ACM SIGOPS 22nd symposium on Operating systems principles, 2009, pp. 117-132.

[5] He S., Zhu J., He P., Lyu M. R., "Experience report: System log analysis for anomaly detection," in Proc. of the 27th international symposium on software reliability engineering (ISSRE), 2016, pp. 207–218.

[6] Dang Y., Lin Q., Huang P., "AIOps: Real-world challenges and research innovations," in Proc. of the 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), 2019.

[7] He P., Zhu J., Zheng Z., Lyu M. R., "Drain: An Online Log Parsing Approach with Fixed Depth Tree," in Proc. the 2017 IEEE International Conference on Web Services (ICWS), 2017, pp. 33-40.

[8] M. Du and F. Li, "Spell: Online Streaming Parsing of Large Unstructured System Logs," in IEEE Transactions on Knowledge and Data Engineering, vol. 31, no. 11, pp. 2213-2227.

[9] Sedki I., Hamou-Lhadj A., Ait-Mohamed O., Shehab M.A., "An Effective Approach for Parsing Large Log Files," in Proc. of the 38th IEEE International Conference on Software Maintenance and Evolution (ICSME), 2022, pp. 1-12.

[10] Lemoudden M., El Ouahidi B., "Managing cloud-generated logs using big data technologies," in Proc. of the International Conference on Wireless Networks and Mobile Communications (WINCOM), 2015.

[11] Miranskyy A., Hamou-Lhadj A., Cialini E., Larsson, A., "Operational-log analysis for big data systems: Challenges and solutions." IEEE Software, vol. 33, num. 2, 2016, pp. 52-59.

[12] El-Masri D., Petrillo F., Guéhéneuc Y. G., Hamou-Lhadj A., Bouziane A., "A systematic literature review on automated log abstraction techniques," Information and Software Technology, vol. 122, 2020.

[13] Zhu J., He S., Liu J., He P., Xie Q., Zheng Z., Lyu M. R., "Tools and benchmarks for automated log parsing," in Proc. of the 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), 2019, pp. 121–130.

[14] Jiang Z.M., Hassan A. E., Flora P., Hamann G., "Abstracting execution logs to execution events for enterprise applications," in Proc. of The 8th International Conference on Quality Software, 2008, pp. 181–186.

[15] Fu Y., Yan M., Xu J., Li J., Liu Z., Zhang X., Yang D., "Investigating and improving log parsing in practice," in Proc. of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 2022, pp. 1566–1577.

[16] Wang X., Zhang X., Li L., He S., Zhang H., et al., "SPINE: a scalable log parser with feedback guidance," in Proc. of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), pp. 1198–1208.

[17] Khan Z. A., Shin D., Bianculli D., and Briand L., "Guidelines for assessing the accuracy of log message template identification techniques," in Proc. of the 44th International Conference on Software Engineering (ICSE), 2022, pp. 1095–1106.