Automatically pinpointing original logging functions from log messages for network troubleshooting

Gaspard Damoiseau-Malraux
Sorbonne Université, CNRS, LIP6
Paris, France
gaspard.dama@gmail.com

Satoru Kobayashi
Okayama University
Okayama, Japan
sat@okayama-u.ac.jp

Kensuke Fukuda
NII / Sokendai
Tokyo, Japan
kensuke@nii.ac.jp

***Abstract*—Modern large-scale computer networks generate massive amounts of log data due to their increasing size, usage, and complexity. At the same time, as cloud-based businesses continue to grow, the need for services and software dedicated to log analysis is more important than ever. Although very useful, log messages often lack the necessary details for efficient troubleshooting, requiring extensive human analysis of the source code. In this paper, we present a new architecture designed with performance in mind, capable of identifying links between software-generated logs and their logging function calls in the source code (referred to as "origins" of the logs). The system we propose uses static code analysis to generate exact log templates, which are used to match log messages efficiently using a combination of a prefix tree and regular expressions. Our implementation SCOLM can pinpoint the origin of log messages with excellent performance and success rate. SCOLM can parse nearly 1 million log lines per minute on a single thread, with a match rate of 90 to 100% on our datasets. It outperforms the speed of traditional regex-based approaches, reducing the speed by about 98.7% in our experiments. The applications of this system are numerous, including live troubleshooting and statistical event analysis.**

***Index Terms*—log analysis, regular expression, source code analysis, parsing, static code analysis**

I. Introduction

Log messages are ubiquitous in modern computing, filling the gap between software developers and system operators. They are crucial to diagnosing or identifying problems and are often read along with the source code for debugging. In large-scale networks, operators can expect very high throughput of log messages, up to petabytes of logs daily [1], [2]. This volume is impossible for humans to analyze effectively. In addition, although they are extremely useful for diagnosing issues, logs can be vague and may not directly help address or even identify the problem (as observed by Xu et al. [3]). Problems can include configuration problems, software bugs, or hardware failures, for example, each of which has radically different causes and consequences.

To effectively troubleshoot a system, it is essential to understand its behavior during failure. Reading log messages is a useful approach; unfortunately, they often lack enough detail to explain failures. Operators sometimes need to read the source code to understand the reason behind a problem, but it is a time-consuming and difficult task without a guide. If additional information about the source code related to the log can be obtained, the investigation can be greatly simplified, saving time and effort for the operator by eliminating the need for extensive code reading.

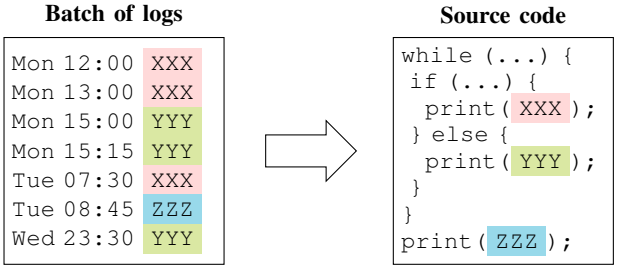

Figure 1. Simplified diagram of the process of pinpointing batches of logs to the source code. Understanding the execution flow of a program is crucial for troubleshooting.

In this paper, we propose a new architecture designed for speed, able to pinpoint the original logging calls from log messages (as outlined with Figure 1). To that end, we extract regular expressions from the source code and use them as templates for matching log messages — examples of regular expressions are shown later in Figure 3 and Figure 5.

However, exclusively relying on regular expressions causes processing time to reach unacceptable durations on modern amounts of data. The solution we propose is to accelerate log matching, by narrowing the search space: we combine the use of regular expressions with a tree-like structure (similar to Kobayashi et al.' word-segmented prefix tree (WSPT) [4]), allowing us to get the best out of both.

We evaluate the performance of this system using our implementation SCOLM[1], measuring both the processing time and conciseness of the outputs. The target software projects in our evaluation are *FRRouting* (FRR) [5], *dhcpcd* [6], and *Avahi* [7], and we use dataset sizes ranging from 2,000 to 89,000 log lines. We find that SCOLM achieves a 98.7% processing time reduction compared to the regular expression-only method.

The contributions of this paper are as follows.

1) A novel architecture combining prefix trees for search-space reduction, using regular expressions and which does not compromise match rate.

[1]SCOLM stands for Source Code Origins of Log Messages.

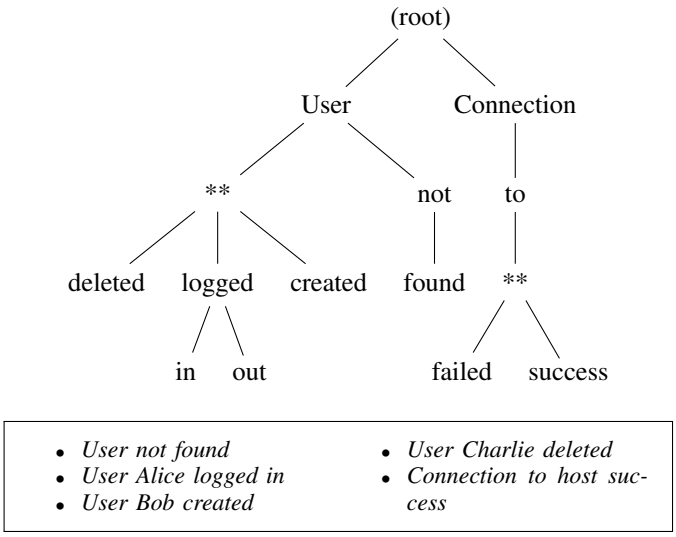

Figure 2. Simplified example of a fictional word-segmented prefix tree, with some corresponding log messages.

2) The ability of the system to adapt in order to match its data more closely, by learning new templates on the fly and inserting them into its database.
3) Improving the processing time required to parse large amounts of log messages, with a single-threaded speed close to 1 million lines per minute on current hardware.

II. Background

A. Log matching

Using *print* statements is a common and effective way to debug and understand the execution of a software. It is often the first choice for beginners because of its simplicity, providing powerful insights into the execution flow, and making it easier to identify issues. As projects grow in size and complexity, actual logging systems become a necessity, but the underlying point remains similar. For system operators, using templates to identify log messages can be an easier and more efficient solution to troubleshooting. The process of identifying a log message by comparing it to a template is called *log matching* in the following.

There are two main approaches to using templates for log matching. The first one uses regular expressions, typically where a template is generated from the source code in order to match specific log messages (e.g. one regular expression per logging function call). Another approach uses word-segmented templates, which consist of lists of words and wildcards. These wildcards act as placeholders for the parts that the operator expects to be variables. We will briefly present each approach and then the idea of our proposed method.

1) Regex templates: Generating regular expressions from source code analysis was originally proposed by Xu et al. [3]. The source code is scanned and used to automatically generate regular expressions by detecting logging occurrences (print statements, for example). These regular expressions are meant to match the log messages that will be produced by the software during runtime. All these regular expressions (called *regex templates*) are stored in a list or a table, along with contextual information associated with the log, including the line number, source file, belonging function, etc. During matching, every log message can be identified by testing against the entire regex table that was built from the analysis. For each log, all matching templates are collected and shown as potential candidates. This method presents the benefit of being exhaustive and accurate, provided that the regex templates are carefully generated by inferring the types of the arguments. It will be referred to as the *regex table* or *regex table-only* approach.

However, the inherent exhaustiveness of the regex table causes a performance issue. For a single log, the lookup time in a regex table is linear with the number of templates. Therefore, as more templates are extracted from the source code, each query will take longer to be fulfilled. Although some projects could be satisfied with the sole regex table solution, other projects such as FRR (approximately 6,500 templates) are too extensive for this method. We find experimentally that the regex solution using precompiled expressions takes 401.30 seconds for 11.8 MB of log messages, with an average of 4.47 ms per log (close to Schipper et al.'s 4 ms [8]). Although this seems to be a good average-case speed, it is still not acceptable for large batches of logs. In comparison, Rabkin and Katz [9] considered back in 2010 that 1.8 GB/min (30 MB/s) is an expected log message throughput for large industry servers such as *Yahoo!*'s. With such high volumes of log messages, we can linearly extrapolate that we will exceed one hour of processing before even reaching 100 MB, and that 1.8 GB would require over 17 hours to process. This shows that for modern standards, we need algorithms capable of handling much more data in shorter amounts of time.

2) WS-templates: Another approach is to use word-segmented templates (WS-templates), which consist of sequences of words and wildcard symbols. *Words* are essentially tokens, chains of characters that do not contain whitespace-like characters (whitespaces, line returns, or tabulations). Wildcard symbols, denoted $\star\star$ in this paper, are used as placeholders for variable parts when searching, and perform just the same as words (one wildcard per word ; consequently, a wildcard does not contain whitespace-like characters either). Past research has shown good performance with prefix tree-based approaches that use **word-segmented prefix trees** (WSPT) [4]. A WSPT is a prefix tree that uses word nodes and wildcard nodes. As such, we can use it to store sentences, or conveniently, log templates (see Figure 2).

The key difference between regex templates and WS-templates is how they match text. Regular expressions use finite automata to parse text, by applying rules defined by patterns. These patterns can be designed in such a way that they offer very little freedom to the text, to increase precision and reduce overmatching (additional false positives). On the other hand, WS-templates traverse the WSPT, following wildcard nodes when encountering expected variables in the

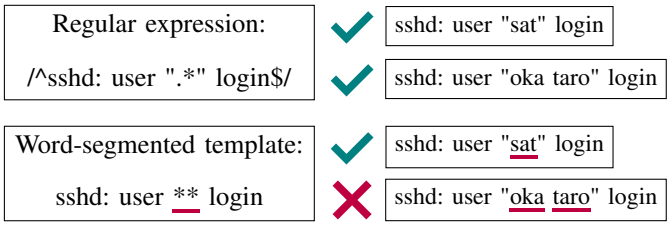

Figure 3. Difference between matching with regex templates v.s. WS-templates.

string. Each wildcard corresponds to a single word, meaning that two WS-templates with similar static parts but different numbers of wildcards cannot match any common log message. We illustrate the nuance in Figure 3, showing how a difference between the number of words and the number of wildcards will cause a failed test.

A major benefit of using WS-templates and WSPT is speed. The search process is fast, because it requires only a few comparisons per word of the log message. For each word of the input, the algorithm checks the current node's children for a match and moves downward in the tree. If no children match the current word but a wildcard is present, it is used as the matching node, and we move to the next word in the sequence. Backtracking also exists in WSPT as it does in regex, but the maximum number of paths on each node is two: a matching static node and a possible wildcard, used if the static node's path turns out to fail.

Unfortunately, the reason why WSPT are fast is also a weakness: the lack of restrictiveness in wildcards can make a WSPT return false positives — a behavior called *overmatching*, because invalid templates are marked as matches. For a log message, *overmatching* occurs when more than one template is returned. That can happen because WS-templates do not allow for fine-grained matching, as any data type can fit in a wildcard. In contrast, for example, a regex template could only allow IP addresses in a specific variable part — any string different from an IP address would be rejected. On the contrary, wildcards do not offer the same restrictiveness: all wildcards are equivalent, and any word can fit any wildcard, as long as the non-variable words match.

A more fundamental flaw of WSPT is that they rely on the assumption that all WS-templates can be determined in advance from static analysis. This is not true because it is often impossible to know the necessary number of wildcards for any WS-template in advance. This implies that a WSPT may not find a matching template for a log message, even though the corresponding template exists in the tree. For example, if a variable part in the source code includes unexpected white spaces (e.g. a multiple-words username), then the WS-template will reject the log message if the number of wildcards does not match the number of words to be skipped. This causes a lower match rate than the regular expression approach.

3) The idea of our approach: Our approach involves the combination of both methods, to take advantage of the accuracy of regular expressions and of the speed of the WSPT.

Table I
OVERVIEW OF THE CHARACTERISTICS OF EACH APPROACH: REGEX TABLE, WSPT, AND OUR PROPOSED METHOD.

Method	Processing time	External templates	Overmatching
Regex table	Large	Available	Small
WSPT	Small	Not available	Medium
Proposed method	Small	Available	Small

The key idea in our approach is to use the WSPT to return a small subset of the regex table, which is then tested by regex matching. This search space reduction is effective because only a few regex templates convert to the same WS-template, resulting in a small subset. If the WSPT rejects the log message (no matching WS-template was found), we proceed to search in the full regex table. Assuming that the regex templates were correctly extracted, we should always find at least one matching regular expression. We then use these matching regex templates to build new WS-templates with the appropriate number of wildcards and update the tree. This ensures that future queries for similar log messages will not require a full search through the regex table again. A visual summary comparing the benefits of our approach with both the regex table and the WSPT-only method is presented above in Table I.

B. Existing work about log matching

Deducing regular expressions from source code to match log messages has been done previously by [3], [10]–[12]. Schipper et al. [8] also explored the process of using regular expressions to associate log messages with their source. Most of the work can be classified into two categories by purpose.

The first purpose is to generate log templates to classify logs [4], [13]–[15]. The format of the log messages is included in the source code, and used to generate templates. For classification, log templates are often created using clustering of output logs. However, these techniques involve estimation and can be prone to failure, in particular considering uncommon log messages. In contrast, the advantage of using source code to create log templates is that the log templates no longer depend on estimations. This means that these templates provide more reliable results for all logs, regardless of their observed frequencies.

Another purpose is to map the execution flow to the source code [8], [10]–[12]. It is generally nontrivial to figure out which branching was taken and which process the system has executed in the source code. If it were possible to map exactly the output logs to their respective origins in the source code, then it would provide partial information on the execution flow of the program. This information would be useful for understanding the behavior of the system.

These existing papers focus on the applications of template generation and log matching, rather than on the process itself. Our work aims to improve the efficiency and accelerate the

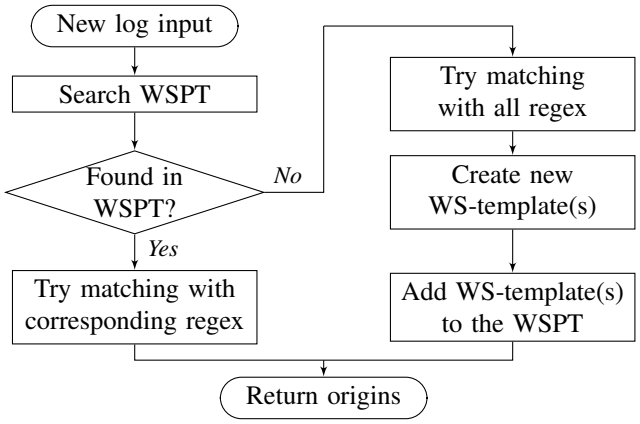

Figure 4. Simplified diagram of the matcher algorithm. We can see how the WSPT narrows the set of possible regex templates. With the datasets we used, the vast majority of WS-templates only have 1 or 2 regex templates associated.

current technologies based on log matching, by proposing a new architecture adapted for speed.

III. Proposed design

A. Overview

The pipeline for pinpointing logging function from log messages can be divided into two main steps:

(1) **Templates generation:** the source code of the target software is parsed and we extract logging calls. Regex templates are generated using the format strings from logging calls. WS-templates are then created from these regex templates. This is performed as a preprocessing step.

(2) **Log matching:** during the matching step (online processing), the system takes a log message as input and returns its origin(s)[2] in the source code.

What sets our system apart is the fact that it uses a combination of two algorithms during online processing. We show the overall process in Figure 5. In the diagram, the process is split into two phases: the offline preprocessing step and the online matching step. The preprocessing step is dedicated to parsing the source code and generating templates, and is performed once beforehand. The matching step is performed on every query to the algorithm, *i.e.* every time the operator wishes to find the origin of a log message. The header part is first stripped from the raw log message, and then the message part is sent to the WSPT algorithm, which uses WS-templates to return a subset of candidate regex that will need to be tested. This first search in the tree saves a lot of computation: even on large software like FRR, the average WS-template only has one or two regex templates associated.

B. Template generation

The main purpose of the template generation step is to prepare datasets for the matching algorithm. This process is decomposed into three steps: code parsing, regex template generation, and WS-template generation.

1) Code parsing: Code parsing relies on the assumption that logging functions include a basic blueprint of the log messages. SCOLM requires that the logging function arguments include a format string similar to *printf*'s — after the extraction, this format string is going to be the base material for template generation.

We parse the source code by identifying files and lines that contain calls to logging functions (a list is provided as user input). The arguments of logging function calls are extracted, the most important being the format string. Optionally, the other arguments can be parsed in order to generate even more precise templates, that depend on the actual values passed to the function.

In the example of FRR, the list of logging functions that we choose to provide to the parser is the following:

- `flog_err_sys`
- `flog_err`
- `flog_warn`
- `zlog_debug`
- `zlog_err`
- `zlog_notice`
- `zlog_info`
- `zlog_warn`

Every line that invokes any of these functions will be detected by the system and used accordingly.

2) Regular expression table: The regex table is created by transforming all format strings into their regex equivalent, such that any log generated from the format string must match the regular expression. Placeholder parts in the regex template should be accurate "translations" of their C-style format specifiers in the string (*%d*, *%s*, *%.2f*, etc.). As pointed out by Schipper et al. [8], it is important to construct the most precise expression possible, to minimize the tolerance of the expression. In addition, precise regex will decrease lookup speed by avoiding costly backtracking. For example, specifiers like *%d* will only produce (signed) integers matching `/-?\d+/`. In contrast, other specifiers like *%s* can hardly be translated to anything more specific than `/.*/` without further knowledge of the corresponding argument.

Returning to the FRRouting example, the EIGRP component of FRR 8.5 includes the following line:

```
zlog_debug("ip_sum 0x%x", iph->ip_sum);    (1)
```

It is extracted and divided into a list of arguments.

```
['"ip_sum 0x%x"', 'iph->ip_sum']    (2)
```

The format string is then converted to the following regex template:

```
/^ip_sum 0x([0-9a-f]+)$/    (3)
```

If implemented correctly, regular expressions created by this process necessarily match any log generated by its corresponding format string.

[2]In case of uncertainty, all matching origins are returned as candidates.

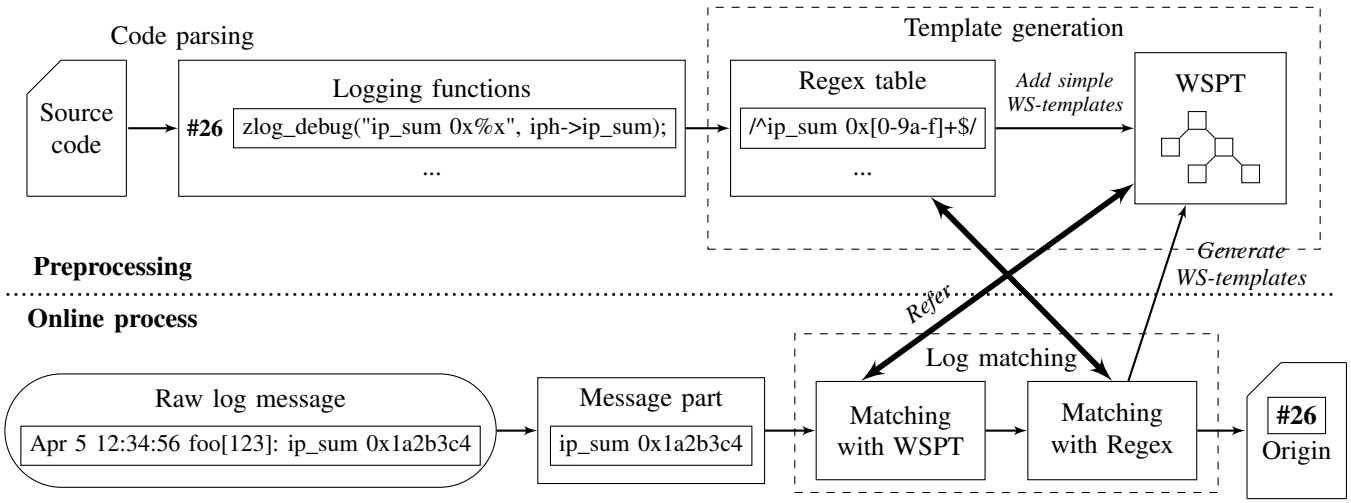

Figure 5. Diagram of the pipeline of our proposed method. The system is separated is two main steps. The first is preprocessing (or *offline processing*), to parse source code and generate templates. The second is the online process, dedicated to match log messages using the templates generated before.

3) WSPT: If we were to start matching logs with a WSPT initialized empty, the system would have to go through the regex table for every type of log message it has not seen before. Of course, as the matching process goes on, the WSPT would eventually fill with all appropriate WS-templates, but at the cost of a high initial processing time.

By default, we assume that format specifiers in the format string will require only one wildcard in the WS-template. Although this is not true for *all* log messages, the assumption should cover a majority of them.

This heuristic helps us to pre-fill the WSPT in advance before starting the online process. The WSPT is filled by generating basic WS-templates from the source code: in the extracted format string, every word that contains a format specifier would be replaced by a wildcard during template generation (e.g. `msg size=%zu` becomes `msg` $\star\star$). When multiple places in the source code create the same WS-template, we store all possible origins, and regex matching will later further reduce the number of origins. Afterwards, during the online processing step, additional WS-templates may be generated when needed, to better fit the dataset.

C. Log matching

During log matching, we will first look into the WSPT matching algorithm for a matching WS-template. If found, we look at the associated subset of regex templates and use the second matching algorithm to keep only regex-matching ones. This narrowing helps the system save a lot of unnecessary computations. If no matching WS-template is found (for example because our heuristic failed), the system uses the second algorithm on all regex templates in the table, and collects the matching ones. New matching WS-templates are created from these results and saved to the WSPT for future lookups. In other words, the WSPT "learns" the proper WS-templates, so the expensive regex table search is not performed more than once for each kind of log message.

The general matching algorithm illustrated in Figure 4 works as follows: we first match with the WSPT, and then match with the regex templates. If the WSPT finds a working WS-template, we only use the regular expressions associated with it. Otherwise, if no matching WS-template is found, the second matching step is performed using all regular expressions available. The output of the general algorithm is the subset of matching templates after the regex matcher. We need a second regex step, because the WSPT does not consider the types of variables between words, meaning that it is too "permissive". Regular expressions are used to enforce stricter rules to reduce the number of candidates returned. In this way, the WSPT has the main function of narrowing down the candidates that need to be tested with regex.

During matching, when the algorithm needs to convert regular expressions to WS-templates on the fly, the process for each of them is carried out in three steps: (1) the input log is divided into words on whitespace-like characters (as previously described), multiple contiguous whitespace-like characters treated as one. (2) In the input log, the positions corresponding to the variable parts in the regular expression are extracted. (3) In the split log message, all words that correspond to variable parts are replaced with wildcards ($\star\star$). The list of words and wildcards is assembled back to form a WS-template, which should now match the log messages. The new WS-template is then added to the WSPT.

IV. Evaluation setup

A. Implementation

We implemented the algorithms in Python 3.10 using the built-in *re* library for regular expressions. Our implementation also used the following frameworks:

- **Amulog** [4] is a tool to manage system logs that allows for automatic classification of log messages using an incremental learning approach, and it provides an API

Table II
INITIAL NUMBER OF LOGS AND TEMPLATES FOR EACH SOFTWARE DATASET USED IN THE EVALUATION PART.

Software	# of regex tpl.	# of WS-tpl.	# of logs available
FRRouting 8.5	6,481	6,346	89,856
dhcpcd 10.1.0	586	580	5,509
Avahi 0.8	367	363	1,971

for searching its database. We use it for all WSPT-related operations, including search and add methods (see *LTSearchTreeNew*). Amulog uses WSPT to store the templates learned for its classification.

- **Log2seq** [16] a rule-based header parser to extract information and remove log headers, also used by Amulog. It is useful for cleaning incoming log messages just before matching, and keeping the message part that is going to be used in the query.
- **Ctags** [17] used to parse and analyze source code (specifically, *u-ctags*). In SCOLM, it is used to extract the line spans of functions, which allows us to determine which functions invoke specialized logging functions.

The source code is available online for reproducibility, along with the necessary configuration for FRR [18].

B. Datasets

The target softwares for our evaluations were *FRRouting*, *Avahi* and *dhcpcd*. *FRRouting* is an open-source routing software suite for Unix-based platforms. It is widely used by companies and data centers to handle dynamic routing and networking [19]. It is a very large software: version 8.5 exceeds a million lines of code and over 3,000 files, mostly written in C language. The scale of FRR is what makes it an interesting test subject: the large number and wide variety of log templates lets us effectively test the performance of our proposed system. *Avahi* is a Zeroconf protocol and DNS service networking software, and is used in major Linux distributions such as GNOME and KDE. *dhcpcd* (Roy Marples' *dhcpcd* [6]) is a software dedicated to automatic and zeroconf DHCP protocol used in many client systems, including Android and Linux.

The log messages for FRR were generated by virtual instances of FRRouting 8.5 running on Docker. Docker containers were managed by *netroub* [20] which relies on *containerlab* [21] for administering the containers, and *Pumba* [22] to introduce hardware and software failures. Netroub was configured on a wide variety of scenarios in order to have diversity in log messages: high communication delays, duplication or loss of devices, misconfigurations, etc. Many of these scenarios can be found on netroub's repository [23]. The scenarios use the network configuration based on *bgp_features*, one of the test networks in FRR topotests [24] (test configurations of FRRouting). It consists of five routers controlled by BGP and OSPF and ten L2 switches. Our data collection uses 25 example scenarios provided in the netroub repository, and each scenario is reproduced from 15 times on average. The final dataset is composed of log messages obtained from all trials and all scenarios.

Log datasets for the Avahi and dhcpcd daemons were collected from real-life operating under normal conditions, from laboratory and private contexts respectively. The settings for the collection of data were three months of operation for the dhcpcd dataset using the default lease time of 43200 seconds (12 hours), and a little over 330 IP address attributions for the Avahi dataset.

Table II provides a summary of the statistics for our datasets. It shows the number of logs available for each target software, along with the initial number of regex templates and WS-templates generated during preprocessing. The number of regex templates represents the number of templates in the regex table and potential candidates for log messages. Similarly, the number of WS-templates corresponds to the number of templates that exist in the WSPT in its initial state.

V. EVALUATION

A. Compared methods

The evaluation was performed comparing four approaches:

(a) The regex table-only approach (baseline for benchmarking the other methods).
(b) SCOLM (proposed method), i.e. the combination of WSPT and regex table. Comparing it with the regex table-only approach will demonstrate its effectiveness.
(c) SCOLM with the WSPT initialized empty, that is, without inserting WS-templates during preprocessing. We still create and insert new WS-templates during the matching step, in order to adapt to the data, but we start "from scratch." Comparing this approach with normal SCOLM will show how useful inserting WS-templates during preprocessing is, to the matching effectiveness.
(d) SCOLM with a pre-trained WSPT. This approach is the opposite of the previous one. We do a first pass in the test data in order to "warm up" the tree and make it learn new templates. Then we do a second pass on the data and measure the time taken. This approach is useful because it shows how fast we can expect the system to perform, once the WSPT has trained. In general, training is performed early during the matching step (see Figure 6). As the WSPT is supplemented with new templates built from the full regex table search, looking into that table happens less frequently. Because it takes a lot of time to perform a full search in regex templates, once the system has warmed up, it almost never goes through the regex table again.

B. Processing time

For simplicity, we will not proceed with a detailed performance analysis of the preprocessing step. Briefly, processing time for parsing the source code and generating templates for FRR, dhcpcd, and Avahi are respectively 29.4 seconds, 3.0 seconds, and 2.4 seconds. In the following, we will admit that the operator has performed preprocessing in advance.

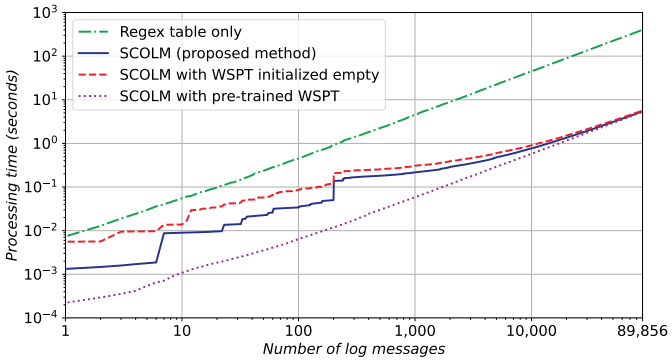

Figure 6. Cumulative sum of the processing time on the FRR dataset (89,856 log lines, 11.8 MB), comparing the regex table and variations of SCOLM.

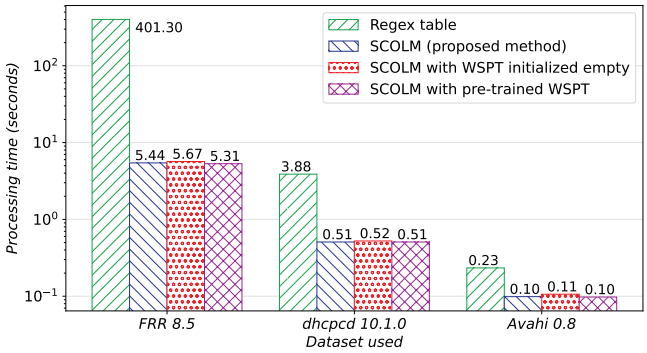

Figure 7. Total processing time with each method on the three datasets. Logarithmic scale for legibility.

The hardware dedicated to performance analysis uses a commodity computer with an Intel Core i5-13600KF and 32 GB of memory. We evaluated SCOLM's speed performance by measuring the time taken on every single query. Figure 7 presents an overview of the benchmarks, comparing the regex table method with our approach SCOLM (with varying settings) across three datasets. Our benchmarks consistently show that SCOLM outperforms the regex table approach. Notably, on our largest dataset, SCOLM has significantly faster performance (reducing the processing time by about 98.7%), with an average processing time of 0.06 ms per log compared to 4.47 ms per log for the regex table method. Variations of SCOLM show similar performance to the original, with the pre-trained WSPT version (*d*) taking the least amount of time, and the version (*c*) being the slowest as expected — although the difference is small. This proves the effectiveness of our heuristic, which is creating WS-templates during preprocessing.

Figure 6 shows the evolution of processing time as a function of the dataset size, by plotting the cumulative sums of all measures for each method. We see that version (*c*) initially performs similarly to the regex table but quickly outperforms it, benefiting from the effectiveness of the WSPT. All SCOLM-based versions converge towards the same processing time, showing the learning capability of the system. Large variations in the left-hand side of the figure are caused by the double logarithmic scale of the figure.

Comparing the performance of SCOLM with the regex table, we recall that the regex table method parses 11.8 MB of log data in 401.30 seconds, and that we predict 17 hours of CPU time to process 1.8 GB of log data. In contrast, SCOLM can process 11.8 MB of log data in 5.44 seconds, hence requiring just over 13 minutes single-threaded to parse a 1.8 GB log dump.

C. Number of candidates

We measure the effectiveness of matching with the *match rate* metric. The match rate is defined as the ratio of log messages finding at least one match, out of the total number of logs. Therefore, an example match rate of 90% would mean that SCOLM found one or more matching templates for 90% of the log messages in the dataset. In addition to the match rate, we also count the number of candidates returned on each query, to estimate how prone our system is to overmatching (returning multiple candidates).

Our hypothesis is that the match rate for all methods will be strictly equal. When the system cannot find a match in the WSPT, it looks into the regex table (hence the match rate of the WSPT cannot be lower than the regex table-only approach). On the contrary, if the system finds a matching WS-template, it returns a subset of the regex table. No WS-template can be returned without its associated regex templates, because all WS-templates were created from at least one regex template. For this reason, the match rate of the WSPT cannot be higher than the regex table-only approach either.

However, we should see a difference in the overmatching metric: because the WSPT returns a small subset of the regex table for testing, the system skips many regex templates that could technically match because they are too permissive. For example, `/(.*) from (.*) to (.*)/`[3] tends to cause overmatching. We expect SCOLM and its variations to return smaller sets of candidates than the regex table approach, and overmatching should be minimal when the tree has fully trained.

Table III displays the number of candidate regex templates returned for each dataset, comparing approaches (a) to (d). All methods find at least one match for all log messages on FRR and Avahi. Since SCOLM falls back to the full regex table when no match is initially found, we indeed observe the same total match rate. As expected, SCOLM shows less overmatching than the regex table method, with 98.28% of its queries finding a single result on the FRR dataset. In contrast, the regex table finds a single candidate for 90.04% of its queries, and even 2.20% of the queries returning 3 candidates. This shows that the regex table is not selective enough and that about 9.96% of the queries contain false positives (compared to 1.72% for SCOLM).

For dhcpcd, the listed methods have a match rate of 90.96%

[3]In actual implementations, `/.*?/` is preferred over `/.*/` when lazy quantifiers are supported. This is an easy trick to reduce the number of steps of the corresponding regular expressions, thereby improving processing time.

Table III
NUMBER OF CANDIDATES RETURNED BY THE REGEX TABLE METHOD (A), SCOLM (PROPOSED METHOD) (B), AND SCOLM VARIATIONS (C) AND (D), ON EACH LOG DATASET.

# of candidates	Regex table only	SCOLM (proposed method)	SCOLM (WSPT initially empty)	SCOLM (pre-trained WSPT)
		FRR 8.5		
0 candidates	0 (00.00%)	0 (00.00%)	0 (00.00%)	0 (00.00%)
1 candidate	80904 (90.04%)	88310 (98.28%)	88296 (98.26%)	88314 (98.28%)
2 candidates	6972 (07.76%)	1546 (01.72%)	1557 (01.73%)	1542 (01.72%)
3 candidates	1980 (02.20%)	0 (00.00%)	3 (0.003%)	0 (00.00%)
Total match rate	89856 (100.0%)	89856 (100.0%)	89856 (100.0%)	89856 (100.0%)
		dhcpcd 10.1.0		
0 candidates	498 (09.04%)	498 (09.04%)	498 (09.04%)	498 (09.04%)
1 candidate	4882 (88.62%)	5011 (90.96%)	5009 (90.92%)	5011 (90.96%)
2 candidates	129 (02.34%)	0 (00.00%)	2 (00.04%)	0 (00.00%)
Total match rate	5011 (90.96%)	5011 (90.96%)	5011 (90.96%)	5011 (90.96%)
		Avahi 0.8		
1 candidate	1971 (100.0%)	1971 (100.0%)	1971 (100.0%)	1971 (100.0%)
Total match rate	1971 (100.0%)	1971 (100.0%)	1971 (100.0%)	1971 (100.0%)

with 5,011 identified log messages out of 5,509. Similarly to the FRR dataset, we observe for dhcpcd that the regex table method has a lower accuracy than the SCOLM method and its variations: 2.34% of the log messages find two candidate templates. We also see that method (c) suffers from the drawbacks of regular expressions, with a lower speed (Figure 6) and two templates (0.04%) finding 2 candidates. In contrast, SCOLM and pre-trained SCOLM always return a single candidate when a match is found. This also shows that our heuristic effectively increases the accuracy of our model.

Lastly, the Avahi dataset shows a 100% match rate for single candidates. A possible explanation is the size difference between this project and FRR; fewer templates are extracted from its source code, meaning fewer possible candidates during identification and less "uncertainty." In addition, the number of log messages is lower than that for FRR, and the variety of log messages in our dataset may not cover a sufficient part of the actual set of logs that can be produced.

VI. LIMITATIONS

The proposed method presents three main limitations. The first two limitations are not caused by the design of our architecture, but occur due to the nature of regular expressions, and often the information present in the source code.

A. Limitation of our general assumption

The largest limitation of SCOLM is that it is largely dependent on the style of the logging functions. Our system assumes that the logging function takes a format string as a blueprint for the output, that we can extract and turn into regex. Unfortunately, it may happen that the log message is dynamically built from a buffer for example (as opposed to a string literal in plain code), or even that no format string is used at all. This is not an issue specific to our proposed method, but rather a general limitation in the way templates can be extracted from source code.

Some way to compensate for the lack of information in format strings would be to use more complex code parsing techniques. In the example of a dynamic buffer, if the system were able to track the modifications made to the buffer, it may be able to create a minimal working template.

This is what prevented SCOLM from obtaining a 100% match date on dhcpcd. Some investigation reveals that most, if not all failed searches were caused by log messages similar to:

ps_root_dispatch: No such file or directory

Reading the source code manually, we find that this message was produced by the following function call:

```
logerr(__func__);
```
(4)

This function does not contain any format string, which prevented SCOLM from building a template — let alone a nontrivial template (e.g., not `/(.*)/`). This means that no corresponding template could be added to the database during preprocessing. In this case, some special rules, defined case by case, could help improve the match rate.

Such logging functions represent a minority in our datasets and in the source code. Besides, in contrast with the two other softwares, dhcpcd seems particularly prone to this kind of problem. This shows that this limitation is very dependent on the target software.

B. Ambiguous format strings

Another limitation (direct consequence of the first and also encountered with existing methods) is caused by the format of some logs. Format strings in source code are sometimes exclusively composed of format specifiers. This causes the corresponding regex and WS-templates to match almost all log messages that exist. Take the following log line:

```
zlog_debug("%s: %s", __func__, buf);
```
(5)

This example is very typical and originates from *lib/event.c* on line 2167. Its corresponding regular expression would be:

```
/^(.*): (.*)$/                    (6)
```

It is easy to see how this regex template will interfere with other regex templates during the full table search. A radical solution is to simply discard this kind of format strings when found in the source code, and choose not to process them. The benefits of avoiding this kind of templates usually outweigh the possibly lower match rate metric.

In the case of FRR, 75 format strings were skipped due to being too ambiguous, and they are not included in the numbers of Table II. Fortunately, no effect was observed from deleting these templates in the range of data we used for evaluation.

C. Full-star templates

Similarly, the WSPT sometimes returns templates composed only of wildcards by mistake (*full-star templates*, e.g. "`** ** ** **`"). It may happen that the correct and more specific template exists in the database, where the number of wildcards does not match. For example, suppose that the following log comes in:

User Alice Smith logged in

and the WSPT contains exactly these two WS-templates:

```
User ** logged in                 (7)
** ** ** ** **                    (8)
```

We, as humans, understand that the corresponding template should be (7). However, because this template does not have the correct number of wildcards, (8) is actually returned because it has the right number of wildcards. We can detect when the WSPT returns a full-star template, and in that situation, choose to still look into the regex table for better matches. From the matching regex templates found, we create more specific WS-templates and add them to the tree.

Fortunately, we observe experimentally that these limitations have little impact on the performance of our system, with the exception of Limitation A on dhcpcd which affects the extraction of templates from the source code.

VII. Conclusion

Identifying log messages can be an invaluable technique for statistical analysis, troubleshooting, and system monitoring. By analyzing the source code, it is possible to create specific regular expressions that can match software-generated log messages with a good success rate. However, regular expressions are slow, and trying to parse large batches of log messages will cause a performance overhead on datasets of thousands of templates. In effect, it is not possible to parse large dumps of logs without a way to accelerate processing. Our approach leverages a special type of tree (word-segmented prefix trees), thereby reducing the number of templates that need to be tested. We implemented this system in SCOLM and evaluated its performance on the datasets of three software: FRRouting 8.5, dhcpcd 10.1.0, and Avahi 0.8. We find that our system achieves an average processing time of 0.06 ms per log message on our largest target (FRR), reducing the processing time by 98.7% compared to traditional regex matching (4.47 ms per log). With the same match rate as the regex table-only method, SCOLM could match 100% (FRR), 90.96% (dhcpcd), and 100% (Avahi) of the datasets we submitted. SCOLM also has a better conciseness than regex-only, with 98% to 100% of its successful matches being a single candidate template on our target projects. This architecture allows us to analyze significantly larger datasets while still maintaining the matching capabilities of traditional regular expression methods.

Acknowledgements

This work was supported by ROIS NII Open Collaborative Research 2025 (251S1-22594) and JSPS KAKENHI Grant Number JP25K15079.

It was carried out in the context of an academic agreement between Okayama University and Sorbonne Université. In particular, the authors thank the Master's Program of Computer Science at Sorbonne Université for its support.

References

[1] Q. Lin, H. Zhang, J.-G. Lou, Y. Zhang, and X. Chen, "Log clustering based problem identification for online service systems," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, 2016, pp. 102–111.

[2] S. Ghanbari, A. B. Hashemi, and C. Amza, "Stage-aware anomaly detection through tracking log points," in *Proceedings of the 15th International Middleware Conference*, 2014, pp. 253–264.

[3] 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*, 2009, p. 117–132.

[4] S. Kobayashi, Y. Yamashiro, K. Otomo, and K. Fukuda, "amulog: A general log analysis framework for comparison and combination of diverse template generation methods," *International Journal of Network Management*, vol. 32, no. 4, p. e2195, 2022.

[5] FRRouting , " frr: The FRRouting Protocol suite ," https://github.com/FRRouting/frr/, accessed 2024-12-28.

[6] R. Marples, "dhcpcd," https://roy.marples.name/projects/dhcpcd, accessed 2025-02-03.

[7] Avahi , " Avahi: Service Discovery for Linux using mDNS/DNS-SD ," https://github.com/alexei-led, accessed 2024-12-28.

[8] D. Schipper, M. Aniche, and A. van Deursen, "Tracing back log data to its log statement: From research to practice," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 2019, pp. 545–549.

[9] A. Rabkin and R. Katz, "Chukwa: A system for reliable large-scale log collection," in *Proceedings of the 24th International Conference on Large Installation System Administration*, 2010, pp. 1–15.

[10] L. Bao, Q. Li, P. Lu, J. Lu, T. Ruan, and K. Zhang, "Execution anomaly detection in large-scale systems through console log analysis," *Journal of Systems and Software*, vol. 143, pp. 172–186, 2018.

[11] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, "SherLog: error diagnosis by connecting clues from run-time logs," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 1, pp. 143–154, 2010.

[12] 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 *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 629–644.

[13] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, "Drain: An online log parsing approach with fixed depth tree," in *Proceedings of the 2017 IEEE International Conference on Web Services (ICWS)*, 2017, pp. 33–40.

[14] A. Makanju, A. N. Zincir-Heywood, and E. E. Milios, "A Lightweight Algorithm for Message Type Extraction in System Application Logs," *IEEE Transactions on Knowledge and Data Engineering*, vol. 24, no. 11, pp. 1921–1936, 2012.

[15] R. Vaarandi and M. Pihelgas, "LogCluster - A Data Clustering and Pattern Mining Algorithm for Event Logs," in *Proceedings of the 11th International Conference on Network and Service Management (CNSM)*, 2015, pp. 1–8.
[16] S. Kobayashi, "log2seq: parse syslog-like messages into word sequences," https://github.com/amulog/log2seq, accessed: 2024-01-24.
[17] Universal Ctags, "ctags," https://github.com/universal-ctags/ctags, accessed: 2024-01-24.
[18] G. Damoiseau-Malraux, "SCOLM: Source code origins of log messages," https://github.com/3atlab/scolm.
[19] NVIDIA, "What is frrouting? | nvidia developer," https://developer.nvidia.com/networking/ethernet-switches/frrouting, accessed 2025-02-03.
[20] C. Regal-Mezin, S. Kobayashi, and T. Yamauchi, "netroub: Towards an emulation platform for network trouble scenarios," in *Proceedings of the CoNEXT Student Workshop 2023 (CoNEXT-SW)*, 2023, pp. 17–18.
[21] Nokia, "containerlab: orchestrating and managing container-based networking labs," https://github.com/srl-labs/containerlab, accessed: 2024-09-30.
[22] A. Ledenev, "Pumba: chaos testing tool for docker," https://github.com/alexei-led, accessed 2024-12-28.
[23] C. Regal-Mezin, "netroub: Platform to emulate network trouble scenarios to collect synthetic operational data," https://github.com/3atlab/netroub, accessed: 2024-09-30.
[24] FRRouting , " Topotests of the FRRouting suite v8.5 ," https://github.com/FRRouting/frr/tree/0546211faab810b46b509afc8ffd42942f8a031d/tests/topotests, accessed 2025-05-16.
[25] J. R., "Using static code analysis to improve log parsing," Sep 2019. [Online]. Available: https://blog.palantir.com/using-static-code-analysis-to-improve-log-parsing-18f0d1843965