Hindawi
Security and Communication Networks
Volume 2022, Article ID 1559270, 11 pages
https://doi.org/10.1155/2022/1559270

Research Article

Spray: Streaming Log Parser for Real-Time Analysis

Feng Zou,[1] **Xingshu Chen**,[2] **Yonggang Luo**,[2] **Tiemai Huang**,[3] **Zhihong Liao**,[3] **and Keer Song**[3]

[1]*Institute of Computer Application, China Academy of Engineering Physics, Mianyang 621000, Sichuan, China*
[2]*School of Cyber Science and Engineering, Sichuan University, Chengdu 610065, Sichuan, China*
[3]*China Mobile (Chengdu) Information & Telecommunication Technology Co., Ltd., Chengdu 610065, Sichuan, China*

Correspondence should be addressed to Feng Zou; 1019272555@qq.com

Received 20 March 2022; Revised 3 July 2022; Accepted 8 August 2022; Published 6 September 2022

Academic Editor: Biao Han

Logs is an important source of data in the field of security analysis. Log messages characterized by unstructured text, however, pose extreme challenges to security analysis. To this end, the first issue to be addressed is how to efficiently parse logs into structured data in real-time. The existing log parsers mostly parse raw log files by batch processing and are not applicable to real-time security analysis. It is also difficult to parse large historical log sets with such parsers. Some streaming log parsers also have some demerits in accuracy and parsing performance. To realize automatic, accurate, and efficient real-time log parsing, we propose Spray, a streaming log parser for real-time analysis. Spray can automatically identify the template of a real-time incoming log and accurately match the log and its template for parsing based on the law of contrapositive. We also improve Spray's parsing performance based on key partitioning and search tree strategies. We conducted extensive experiments from such aspects as accuracy and performance. Experimental results show that Spray is much more accurate in parsing a variety of public log sets and has higher performance for parsing large log sets.

1. Introduction

In today's Internet environment, applications, operating systems, and network devices will generate a variety of real-time logs, which play an important role in the field of cyber security. With the continuous evolution of means for network attacks, more and more attacks cannot be intercepted by firewalls. Legitimate users of an intranet may also operate corresponding internal systems illegally. As a result, many computer systems need to make security responses through attack identification, anomaly detection, and alarm generation by analyzing log data. Massive log-based research on security analysis has been conducted whose results have been used in log audit [1], intrusion detection [2], anomaly detection [3, 4], user behavior analysis [5], and network fault diagnosis [6], among others. [7] proposes a technique for extracting sensitive information from unstructured data. In addition, a large number of products for security analysis of logs have been put on the market, such as Splunk [8] and OCEANS [9]. They realize interactive analysis by loading IPS logs, application logs, and other heterogeneous data to help experts discover anomalies and security events rapidly.

System developers, however, usually write log print statements in the form of free text in the source codes. Therefore, raw log messages are essentially unstructured or semi-structured data. With these raw logs unprocessed, generally, we can only do simple keyword searches, but cannot effectively analyze the security issues hidden in the logs. Therefore, we need to parse the raw log data before analysis.

Log parsing is a process of discovering the log template corresponding to each log message (each template corresponds to a log print statement in the system), extracting variable parameters, and finally parsing unstructured or semi-structured log messages into structured log events.

Conventional rule-based [10, 11] log parsers require professionals to manually create massive complex regular expressions (each regular expression [12] corresponds to a log template) and add them to the parsing rule set. In the process of log parsing, log messages are matched with the

regular expressions in the rule set one by one. Such approaches have many demerits, including (1) users need to know all the template structures of a log; (2) creating massive regular expressions for complex systems is labor consuming and error-prone; and (3) when updating the system or application, it is necessary to update the parsing rule set at the same time to ensure the accuracy of log parsing.

Current automatic log parsers mostly work by batch processing log data, i.e., comprehensively computing all contents of raw log messages and exporting the parsing results in batches. All contents of the log data must be available before being parsed. As the data set needs to be fully loaded, this batch processing mode is restricted by computing resources. It may even fail to work if the historical log set is too large. For easy and unified management of logs from multiple sources, we usually employ some data acquisition tools (for example, Apache Flume [13] or Ismael [14]) to stream the real-time logs to the Kafka [15]. The log data will be cached in the form of message queues. It is also difficult to parse these real-time streaming logs by batch processing.

In this paper, we propose Spray, a streaming log parser for real-time analysis. At first, in our design, incoming log messages are tokenized to form a token list. Different from the tokenization by other log parsers, we save each separator used for tokenization as a separate token and classify these tokens after tokenization. Second, considering logs of the same template may have variable lengths, we filter the templates initially by computing the similarity between tokenized logs and their templates based on the longest common subsequences (LCS). Then, we accurately determine the relationship between logs and their templates based on the law of contrapositive in discrete mathematics, to extract log variables and update log templates. In addition, we use two strategies: key partitioning and search tree, to improve Spray's parsing performance. Finally, we conduct extensive experiments with a wide range of log data to evaluate Spray and compare it with other parsers, such as Drain [16], Spell [17], IPLoM [18], and MoLFI [19]. The experimental results show that, for 16 public log sets [20], Spray is more accurate, and for a greater number of log sets, Spray has higher parsing performance.

The rest of the paper is structured as below: Section 2 outlines the related studies on log parsers. Section 3 details the log parser proposed, including the parsing process and performance optimization strategies. Section 4 is the experiments and analyses, including multiple evaluates indicators, such as accuracy, performance, and effectiveness. Section 5 gives a summary of this paper.

2. Related Works

ELK [14], composed of Elasticsearch, Logstash, and Kibana, is the most active real-time log analysis platform in the open-source community. It parses unstructured or semi-structured log messages into structured data based on the user-defined regular expressions. Splunk [8], a kind of commercial software with a high market share in the field of log analysis, parses common types of log messages by virtue of prebuilt regular expressions. Both of them, as the mainstream in the industry, still employ the conventional rule-based log parsers and do not support the automatic parsing of unknown logs.

However, automatic log parsers have been extensively studied in the academic circle and can fall into two categories: batch processing and streaming.

The log parsers based on batch processing include LFA [21], LogCluster [22], LogSig [23], LogMine [24], [25, 26], IPLoM [18], and MoLFI [19], among others. LFA and LogCluster believe that a log statement contains two types of characters: variables and constants. As constants are fixed and frequently occur, log parsing can be interpreted as the mining of frequent items. LogSig and LogMine follow the idea of clustering. Log templates form a natural pattern of a log message set, based on which log parsing can be modeled as the clustering of log messages. In [25, 26], static analysis techniques are employed to obtain log template information from program source codes. IPLoM uses an iterative partitioning strategy that partitions log messages into groups based on the message length, token location, and mapping relationship. MoLFI reveals that log parsing is to determine the trade-off between generality and specificity of log patterns and further interpret log parsing as multiobject optimization.

Since streaming or similar log parsers read and parse log messages one by one, such operations will not use too much CPU and memory as the number of logs increases, making it possible to process a nearly unlimited number of logs. Such log parsers mainly include Drain [16], Spell [17, 27], and Agrawal [28]. In the parsing process, Drain builds a parsing tree with a fixed depth and assigns the incoming log messages to the depth layer and token layer of the parsing tree in sequence before transferring these messages to the similarity computation layer for final parsing. This technique cannot accurately parse the log data with the same template but with different lengths. Spell is the first parser that proposes to match a log message with its template based on the LCS and optimizes the time complexity. Heavy reliance on the relationship between the LCS and log templates affects the accuracy of this technique. Through distributed processing, Logan partitions and assigns log data files to different template extraction tasks for parsing, realizing concurrent log parsing. This technique, however, has a drawback—the templates may be inconsistent in template extraction tasks. Although the solution is given, that is, hard merge and soft merge, it cannot guarantee real-time parsing. Besides, the parser needs to partition batches of log files for concurrent operations. Therefore, this is only a log parser similar to streaming.

3. Methodology

Spray is a streaming log parser for real-time analysis. It works in four main stages: tokenization, similarity computation, template filtering, and template updating and merging. In addition, we also design the key partitioning and search tree strategies to improve Spray's performance.

3.1. Tokenization. Tokenization consists of three steps, as shown in Figure 1. In the first step, each time when a raw log message is entered, we identify common variables from it and replace them with the wildcard character "*" through some simple regular expressions based on the domain knowledge (for example, IP, time, etc.).

In the second step, we segment the logs with common punctuation marks (such as space and comma) as the separators, to form a token list. Different from the tokenization by other log parsers, we save each separator used for tokenization as a separate token because separators often act as an important reference for determining a log template.

In the third step, based on their nature, Spray further classifies these split tokens into the following types: (a) space tokens; (b) known variable tokens with identified wildcard characters "*"; and (c) unknown tokens.

3.2. Similarity Computation. For a tokenized log, Spray will traverse the current log template set (the generation process of templates will be described in detail in Section 3.4) and compute the similarity between the log and the templates in turns.

A log message contains two types of tokens: constant and variable. We consider a log message or template as a sequence and each token it contains as an element of the sequence. When some log messages belong to the same template, the constant tokens in these log messages are in a fixed order of sequence. Moreover, the constant and variable tokens in log messages may be separated by each other, resulting in discontinuous constant tokens. Based on the above two characteristics of the constant tokens in log messages, i.e., orderliness and discontinuity, we choose to use LCS [17] for similarity computation. During this process, we skip space tokens because of their high proportion and low impact and compute only the remaining two types of tokens.

$$\text{Similarity} = \frac{\text{length}_{lcs}}{\text{length}_{t}}. \tag{1}$$

We compute the LCS (E, T) of the log E and template T and then obtain its similarity, as shown in (1), where length_{lcs} is the number of tokens of the LCS, and length_{t} is the number of tokens in the template T after all space tokens are excluded. If the similarity exceeds the threshold, it means the similarity between E and T meets the minimum requirements.

3.3. Template Filtering. Even after similarity computation, we still cannot guarantee that a log with a similarity greater than the threshold belongs to a certain log template. For example, the log "A B 1 C *D* 2" does not belong to the template "* A B C D" although their similarity exceeds the threshold. For this, we need to single out the right template from those meeting the similarity threshold requirements.

When the log message E belongs to the template T, after calculating the LCS of E and T, we cannot guarantee that all the tokens belonging to LCS belong to the constant part, but we can be sure that all the tokens not belonging to LCS belong to the variable part. Therefore, we can use the tokens in the LCS as the separators to divide E and T into the same number of variable subsequences.

Based on the characteristics of log messages, we can make the following hypothesis: if a log message E belongs to a template T, all the variable subsequences in E share the same structure with their counterparts in the template T.

According to the hypothesis, we can reason backward based on the law of contrapositive in discrete mathematics. The contrapositive law is described as follows: given that proposition P can deduce proposition Q, then the negation of proposition Q can deduce the negation of proposition P.

Thus, we have four propositions:

(1) Proposition 1: the log message E belongs to the log template T.

(2) Proposition 2: all the variable subsequences in the log message E share the same structure with their counterparts in the template T.

(3) Proposition 3: not all the variable subsequences in the log message E share the same structure with their counterparts in the template T.

(4) Proposition 4: the log message E does not belong to the log template T.

According to our hypothesis, we can infer that "Proposition 1 ⇒ Proposition 2." Given that Proposition 4 is the negation of Proposition 1 and Proposition 3 is the negation of Proposition 2, we can deduce that "Proposition 3 ⇒ Proposition 4," according to the law of contrapositive. Therefore, the proposed template filtering is based on Proposition 3. If Proposition 3 is true, Proposition 4 can be deduced. That is to say, when not all the variable subsequences in the log message E share the same structure with their counterparts in the template T, the log message E does not belong to the log template T.

In the tokenization process, we have classified the split tokens into (a) space tokens, (b) known variable tokens, and (c) unknown tokens. Based on the LCS, we break down the remaining tokens into multiple token subsequences and label their structures (as "a" or "aca," for example). Here we compare the labeled values of these variable subsequences one by one. In this process, as long as any one pair of structures are different, we consider that the log E does not belong to the template T. Figure 2 visually illustrates the comparison process.

We divide the comparison situation into two cases. For the first case, if the labeled values of two structures have the same character length, we compare whether each pair of characters are both space token or both not in turn, such as the example in Figure 2.

For the second case, if they have different character lengths, we divide the structures of variable subsequences in the template into five types: "b," "ab," "aba," "ba," and others. The first four types correspond to the structures "[abc]+," "a[abc]+," "a[abc]+*a*," and "[abc]+*a*" of the variable subsequences in the log. The fifth type does not match any of the labeled structures. Structures are labeled in the

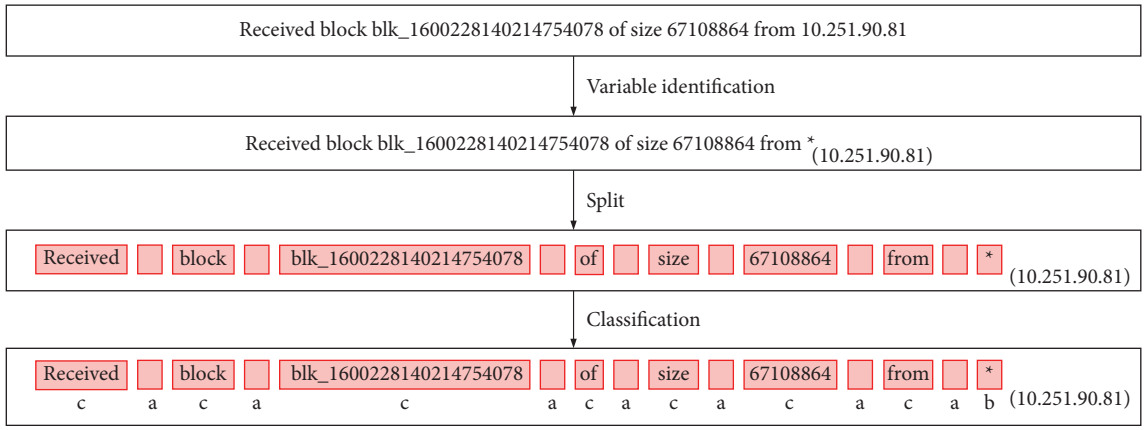


Figure 1: Tokenization process of Spray.

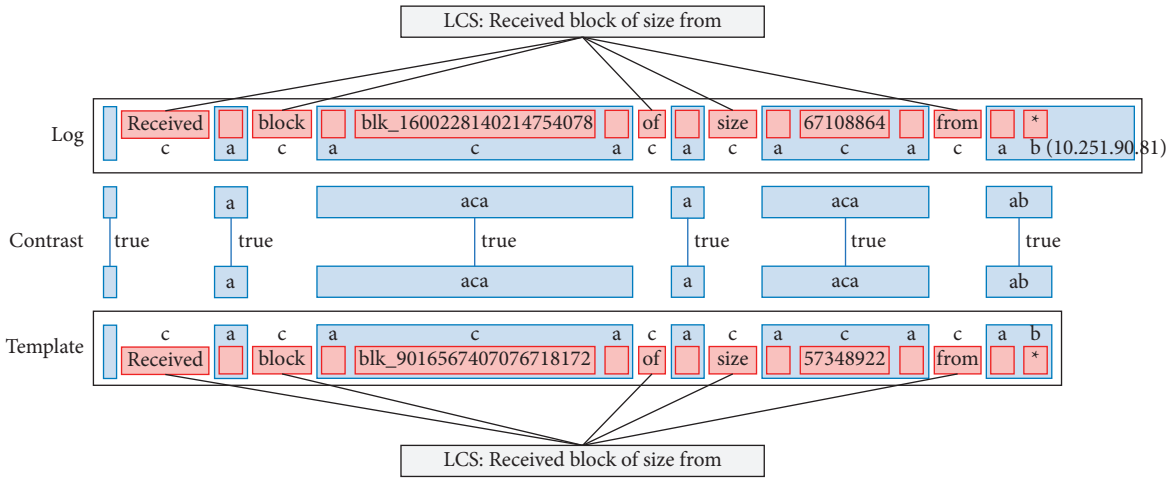


Figure 2: Comparison of labeled value of each variable subsequence one by one.

form of regular expressions, where "+" means the character before the mark occurs once or more times, and "[abc]" indicates "a" or "b" or "c," see Figure 3.

In the four types where the variable lengths are not equal, as shown in Figure 3, to determine whether a variable subsequence in the log has the same structure as its counterpart in the template, we check whether it starts or ends with the space character ("a"). In the log message output statement of the program, since the first and last characters of all variables are not spaces, space tokens immediately before and after the variables are extremely important and can be used as the basis to determine whether the variable structures are equal.

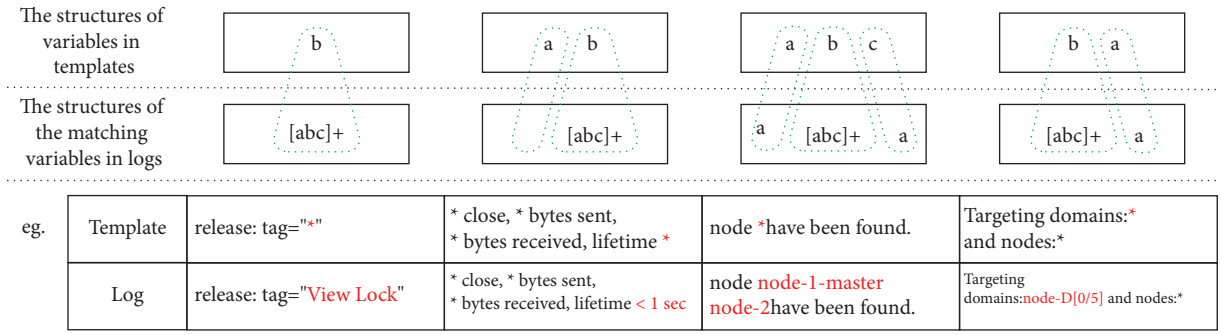
Variables can be extracted during the comparison process. When the comparison is over and the matching template is found, the parsing of this log is completed.

3.4. Template Updating and Merging. Through the previous process, if the matching template is found, the corresponding log can be parsed. However, in order to extract the log template, we also need to update and merge the templates because these templates are unknown and the template list is also empty at the beginning.

If a log does not match any template, we will save the incomplete parsing results of this log as a new template and include it in the template list. The new template may contain the variable locations found during tokenization, labeled as "*."

If a log matches a template, we may need to update the template. The template is updated only when the number of tokens between two adjacent LCS tokens of the template is the same as that of the log. In this case, we label non-space tokens as "*" (for example, if the log "A B 1 C *D* 2" matches the template "A B 3 C *D**," the template is updated to "A B * C *D**").

Then, how do we update the template if the number of tokens of the template is different from that of the log? We propose to merge the templates. If a log template is updated, we parse and compare the updated template with others in the template list (same as the process for parsing log messages as described above). If the updated template matches a template, we merge them into one. For example, after we input two logs "A B 1 2 C D" and "A B 3 C D" with the same



eg.					
	Template	release: tag="*"	* close, * bytes sent, * bytes received, lifetime *	node *have been found.	Targeting domains:* and nodes:*
	Log	release: tag="View Lock"	* close, * bytes sent, * bytes received, lifetime < 1 sec	node node-1-master node-2have been found.	Targeting domains:node-D[0/5] and nodes:*

Figure 3: Four types with unequal variable lengths.

template, Spray parses them and generates two new log templates "A B 1 2 C D" and "A B 3 C D." When we input another log "A B 4 C D," Spray matches it with the template "A B 3 C D," so the template is updated to "A B * C D." At this moment, Spray compares the updated "A B * C D" with the template "A B 1 2 C D." If they match, it will merge them into a template "A B * C D." After that, if we input the logs with potentially different numbers of tokens such as "A B 5 C D" and "A B 6 7 C D," Spray can parse them all based on the above template.

3.5. Key Partitioning. To reduce the times of matching between logs and templates, we employ the key partitioning strategy. This strategy splits the log messages that meet different key conditions into different partitions. Each partition has a template sublist (in which the number of templates is smaller than the total). Thus, we only need to match the log messages with the templates in the sublist. This largely reduces the number of computations and enhances the parsing performance.

In order to ensure the accuracy of parsing, we need to avoid including log messages that belong to the same template into different partitions. Therefore, we must find the appropriate key to guide the partitioning process.

In [16], three conclusions are made: (1) the log messages that belong to the same template have the same length; (2) the token at the beginning of the log message is more likely to be a constant; and (3) the tokens containing numbers should be excluded to determine whether a token is a constant. As Spray can parse logs that belong to the same template but have different lengths, we abandon the first conclusion. In addition, we strengthen the third conclusion by specifying that the tokens containing only upper- and lowercase letters should be considered when determining whether a token is a constant.

To sum up, Spray selects a token as the key based on the following rule: for each log message, Spray finds the first token containing only upper- and lowercase letters by checking from the beginning to the end. If the token meeting the above conditions cannot be found in some log messages, Spray assigns them to an additional partition.

3.6. Search Tree. As the parsing proceeds, the number and structure of log templates tend to be stable. Therefore, for most incoming log messages, their templates have been included in the template list and do not need to be updated. As the LCS-based similarity computation is characterized by high time complexity, parsing each incoming log following the above procedures will result in relatively poor parsing performance.

To further improve the parsing performance, we design a search tree as shown in Figure 4 to save the template list. The template list referred to is the list after key partitioning. Each node of the tree saves a constant, and the variable structure between this constant and its previous constant. As a result, template filtering can be executed in the search tree based on the law of contrapositive.

If the matching template is singled out through the search tree, we will skip similarity computation and template updating, and output the parsing results directly. Figure 5 shows the complete execution process of Spray after the search tree is incorporated.

4. Experiments and Analysis

To verify Spray's effectiveness, we conducted experiments to compare Spray with existing log parsers, including MoLFI [19], Drain [16], Spell [17], and IPLoM [18]. Spray, Drain, and Spell are streaming log parsers, while MoLFI and IPLoM are batch processing log parsers.

We first conducted accuracy experiments based on 16 public log sets [20] and then tested performance with larger log sets. We included all of the above five parsers for accuracy experiments and selected tree streaming log parsers, Spray, Drain, and Spell, for performance experiments. The experimental results show that Spray is better in terms of both accuracy and parsing performance. We also propose a new effectiveness evaluation method on the basis of [28], which also proves that Spray is better.

4.1. Accuracy Rate. The accuracy rate of log parsing is the ratio between the number of logs correctly parsed and the total number of logs in the log set. Theoretically, the accuracy rate should be calculated in such a manner that the parsing results are equal to those given in the ground truth. For example, however, the variable "blk_10737435122731" in the HDFS log set is expressed as "blk_*" in the ground truth, but most log parsers parse it as "*". Obviously, this cannot be

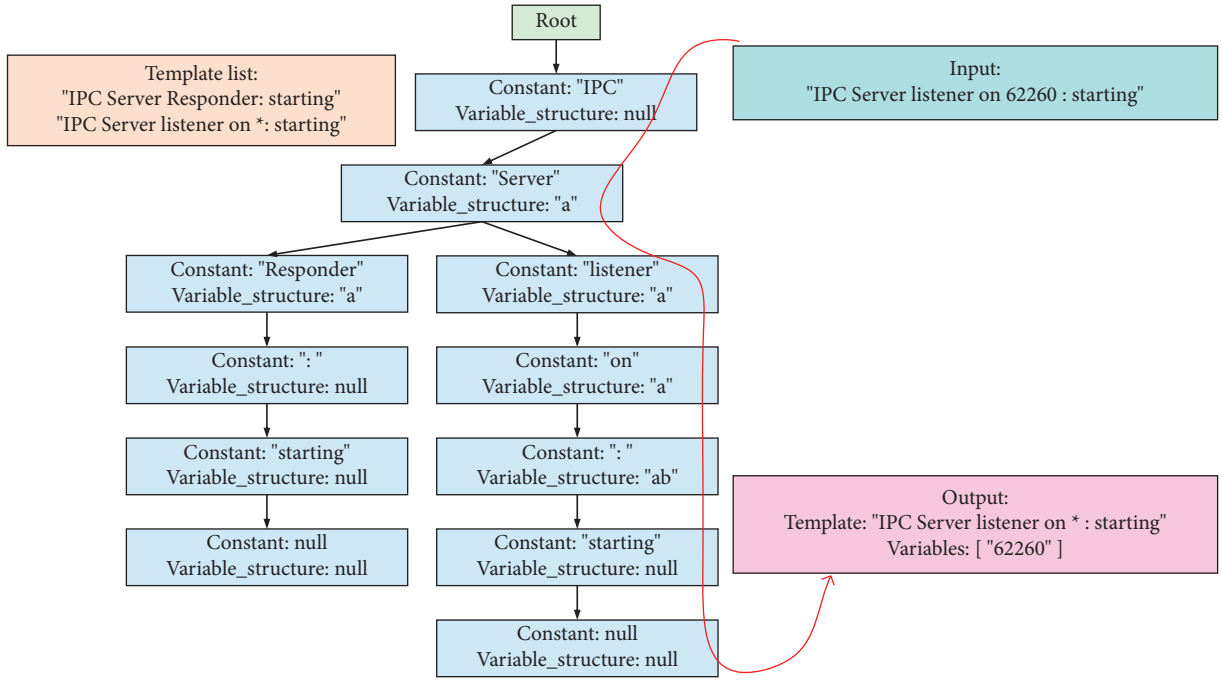


Figure 4: Search tree for saving log templates.

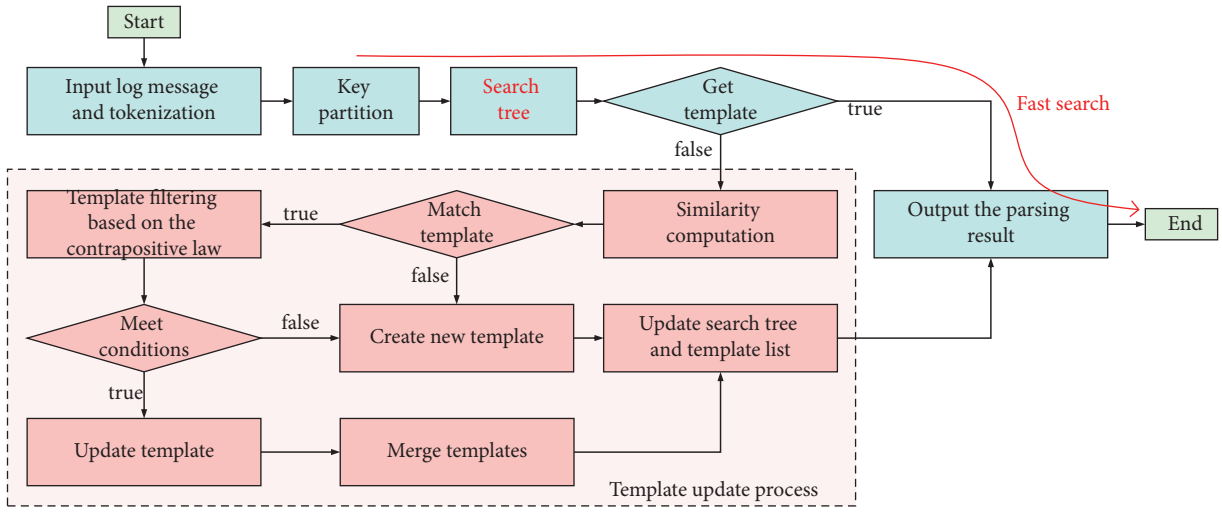


Figure 5: Complete execution process of Spray.

considered a parsing error. Therefore, we think this situation is also correct when we compute the accuracy rate.

First, we selected several thresholds to test their impact on the accuracy rate of Spray, with the results shown in Figure 6.

It can be found that the change in threshold values has little impact on Spray's parsing results. This is because Spray will run a template filtering process after computing the similarity. This process realizes more accurate matching between logs and templates. When the threshold is smaller than 0.6 or greater than 0.8, the accuracy rate reduces for some log sets. In existing log parsers, the threshold is taken as 0.5 in most cases, because the number of variable tokens in a log message can hardly reach half that of the log message. Considering the role of separators, Spray regards separators as tokens. As separators are more likely to be constants, we choose 0.7 as the threshold of Spray. The experiments proved that 0.7 is more rational for Spray.

Next, we compared Spray with several parsers in terms of the accuracy rate on 16 log sets, as shown in Table 1.

According to the results in Table 1, Spray has the highest accuracy rate on 14 log sets and the second-highest accuracy rate on the remaining 2 log sets. On 6 log sets, such as HDFS, Apache, and Windows, the accuracy of Spray exceeds 0.95.

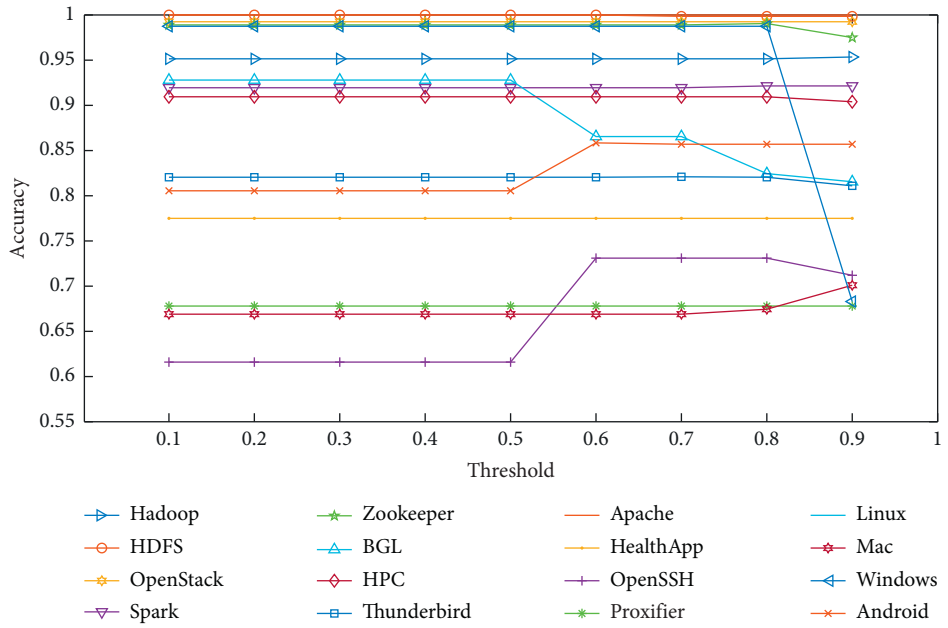


Figure 6: Impact of different thresholds on Spray's accuracy rate.

Table 1: Accuracy rates of several log parsers.

	Spray	MoLFI [19]	Drain [16]	Spell [17]	IPLoM [18]
Hadoop	**0.9515**	0.1530	0.6125	0.1205	0.1305
HDFS	**0.9985**	0.4835	0.9965	0.2780	—
OpenStack	**0.9925**	—	0.2070	0.0110	—
Spark	**0.9195**	0.2080	0.9025	0.6370	0.6380
Zookeeper	**0.9890**	0.7480	0.9615	0.7185	0.4720
BGL	**0.8655**	0.6850	0.8055	0.1420	0.3510
HPC	**0.9095**	0.6320	0.8125	0.5075	0.6375
Thunderbird	0.8210	0.0325	**0.8815**	0.7380	0.4690
Apache	**1.0000**	0.2695	0.6935	0.2695	0.6935
HealthApp	**0.7750**	0.1665	0.6080	0.1520	0.1550
OpenSSH	**0.7310**	0.0615	0.5065	0.1210	—
Proxifier	**0.6780**	—	0.5035	—	—
Linux	**0.2695**	0.0295	0.2665	0.1095	0.1700
Mac	**0.6690**	0.0440	0.3935	—	—
Windows	**0.9875**	0.0030	0.7460	0.0010	0.0010
Android	0.8570	0.0790	**0.9065**	0.1925	\

In contrast, for other parsers, only Drain achieves an accuracy of 0.95 on HDFS and Zookeeper log sets. On MoLFI, Spell, and IPLoM, no log set achieves an accuracy rate higher than 0.90.

Spray has higher accuracy than other parsers mainly because (1) Spray considers the role of separators in log message tokenization, especially the impact of space characters on log parsing; (2) Spray can parse the log messages belong to the same log template but with varying lengths; and (3) Spray realizes the accurate matching between logs and their templates based on the law of contrapositive.

4.2. Performance and Effectiveness. Parsing performance is another important indicator to measure the quality of log parsers. Without efficient log parsing, if logs are generated faster than they are parsed, the real-time incoming logs will pile up. Therefore, we evaluated the parsing performance of Spray by comparing it with the two other streaming log parsers, Drain and Spell.

4.2.1. Time Complexity Analysis. Suppose the average length of log messages and templates of a log set is L, and the number of templates is N.

For Spray, the average depth of the search tree is half the template length, i.e., $L/2$. With key partitioning, the average number of templates in each partition is log$N$, so the number of paths in the search tree is log$N$. Based on the depth $L/2$ and number of paths log$N$ of the search tree, the log messages with an average length L can successfully match the templates in the search tree, and the time complexity is O (Llog (Llog$N$)). Therefore, for parsing a log, the time complexity of Spray is approximately O (Llog (Llog$N$)).

Similarly, the time complexity of Drain and Spell can be obtained as shown in Table 2.

For Spray and Drain, their time complexity cannot be determined if the magnitude of Llog$N$ and N cannot be determined. However, it is known that their time complexity is lower than that of Spell. It can be inferred from time complexity alone that, compared with Drain and Spell, the parsing performance of Spray is less affected by the number of templates N in the log set.

Table 2: Time complexity of several log parsers.

Method	Time complexity
Spray	O (Llog (Llog$N$))
Drain [16]	O (Llog$N$)
Spell [17]	O (Llog (LN))

Table 3: Log sets used for performance test.

Log sets	Number of entries
Linux	25567
Apache	56482
Zookeeper	74380
HealthApp	253395
SSH	655147
Android	1555005
BGL	4747963

Table 4: The information of software and hardware.

OS	CentOS Linux release 7.7.1908
Core	3.10.0–1062.18.1.el7.x86_64
CPU	Intel (R) Xeon (R) CPU E5-2680 v4 @ 2.40 GHz
Memory	256 GB
Disk	Mechanical hard disk

4.2.2. Performance Comparison. To compare the performance of these parsers more accurately, we conducted experiments using the same log sets. The log sets used are shown in Table 3.

During the experiments, these parsers run in single-threaded mode, and the software and hardware used are shown in Table 4.

The performance test results of these parsers on several log sets are shown in Figure 7, where (a) shows the parsing time used and (b) reflects the throughput (number of logs parsed per second).

From Figure 7(a), it can be seen that the more the log sets, the longer the time consumed by all three parsers. Specifically, Spray consumes less time than Drain and Spell for handling all log sets. According to Figure 7(b), Spray has the best performance, and its average throughput can be 10,000 entries per second, compared to only 4,000 and 2,000 entries per second, respectively, for Drain and Spell.

According to the evaluation results stated in [20], Drain and Spell are high-performance techniques among the existing 13 log parsers. In general, the performance of Spray is better than that of Drain and Spell, so it can be considered that Spray has high parsing performance.

4.2.3. Effectiveness. To evaluate the effectiveness of parsers on large data sets, we have to define the regular expressions for log sets with conventional rule-based techniques to obtain the ground truth. This is highly labor consuming and error prone. To avoid this, [28] proposes a new effectiveness indicator, as shown in.

$$\text{Loss}(T) = (T \cdot \text{length})^{\theta} + \frac{1}{T \cdot \text{length}} \times \sum_{i=0}^{T \cdot \text{length}} \frac{\text{avgTokensLost}(t_i)}{t_i \cdot \text{length}}. \tag{2}$$

This indicator defines two calculable values: T.length and avgTokensLost. If T.length (number of templates) is too large, it means massive templates in the log are not properly identified. The larger the value of avgTokensLost (the difference between the average length of each template and that of its matching log), the greater the possibility that the constant tokens in the log are parsed to variables. Therefore, the lower Loss is, the more effective log parsing is.

T.length and avgTokensLost represent different dimensions (generally, T.length is high while avgTokensLost is low). If these two values are to be added up, at least one of them must be subject to exponentiation (for example, θ in (2)). This indicator, however, needs to be adjusted for different log sets, which means (2) is not universal. Therefore, it would be better if we do the computation by multiplying these two values. However, an adjustment needs to be made considering that this technique may not be applicable in some cases. For example, for Drain, a log is always assumed to have the same length as its template (i.e., avgTokensLost is always 0). Therefore, the product will always be 0 if these two values are multiplied. For that reason, we replace avgTokensLost with avgVarTokens, i.e., the average number of variable tokens identified in each log, as shown in.

$$\text{avgVarTokens} = \frac{1}{n} \times \sum_{i=0}^{n} \text{varTokens}(i). \tag{3}$$

Therefore, we introduce a new method for calculating the effectiveness evaluation indicator as shown in.

$$\text{Loss}(T) = (T \cdot \text{length})^{(1/2)} \times \text{avgVarTokens}^2. \tag{4}$$

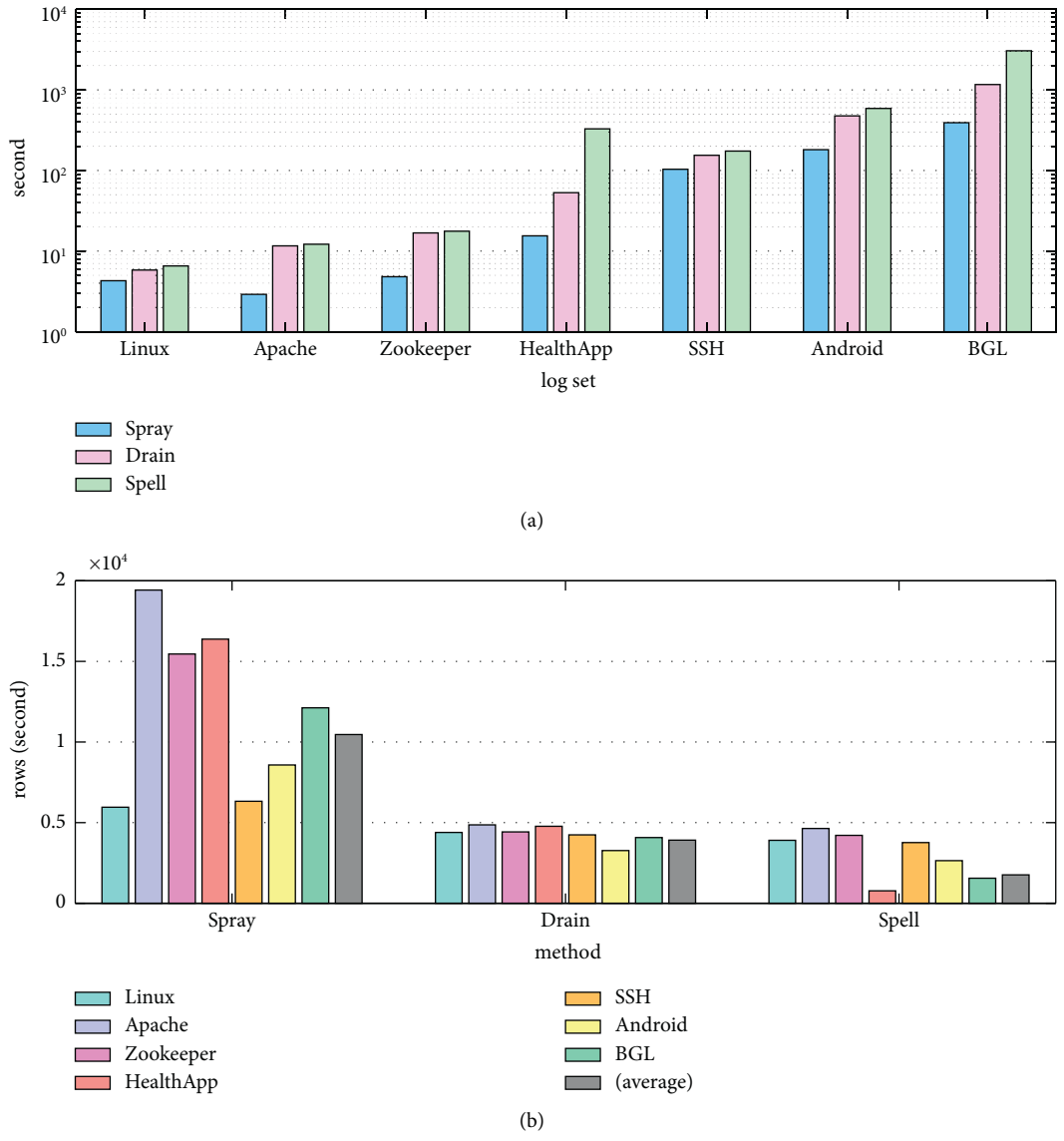


Figure 7: Parsing performance of several parsers. (a) Take up time. (b) Throughput.

Table 5: Comparison of loss.

Log sets	Spray	Drain [16]	Spell [17]
Linux	**139.90**	175.48	166.14
Apache	**4.76**	11.67	16.28
Zookeeper	9.55	14.02	**9.25**
HealthApp	**7.38**	710.58	250.42
SSH	77.69	**58.16**	104.64
Android	**864.54**	1059.30	1302.23
BGL	**43.67**	155.23	814.77

We continue to calculate Loss of Spray, Drain, and Spell based on the log sets listed in Table 4. The results are shown in Table 5.

Based on the results in Table 5, Spray does better in parsing more log sets, with the lowest Loss for 5 of 7 log sets. For Zookeeper log set, Spray achieves a slightly higher Loss than Spell while for SSH log set, its Loss is only higher than that of Drain. For HealthApp log set, Drain generates too many templates, making its Loss much greater than those of the other two parsers. As excessive parsing leads to excessive extraction of variables, Spell has a higher Loss than those of the other two parsers for Apache, SSH, Android, and BGL log sets.

5. Conclusion and Future Works

To realize automatic, accurate, and efficient real-time log parsing of unstructured log text, we propose Spray, a streaming log parser for real-time analysis in this paper. This parser innovatively realizes accurate matching between logs and their templates based on the law of contrapositive after tokenizing the incoming log messages and computing the similarity, thus obtaining accurate parsing results. In addition, we use two strategies: key partitioning and search tree for high parsing throughput. We conducted extensive

experiments from such aspects as accuracy, time complexity, performance, and effectiveness. The experimental results show that Spray has the highest accuracy rate on 14 log sets and the second-highest accuracy rate on the remaining 2 log sets. In terms of parsing performance, Spray realizes an average throughput of 10,000 entries per second, higher than those of Drain and Spell. From the aspect of effectiveness, Spray has the lowest Loss for most log sets. Therefore, we believe Spray has better accuracy and parsing performance and can parse large real-time logs effectively.

In the future, we plan to automatically tag the semantics of log variables and automatically assign field names to the extracted variables in log messages. This will not only help us understand the semantics represented by log variables but also facilitate the direct use of the analysis platform for structured data analysis.

Data Availability

The log data supporting this log parser are from previously reported studies and data sets, which have been cited.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported by Defense Industrial Technology Development Program (No. JCKY2019602B013).

References

[1] M. Macak, I. Vanát, M. Merjavý, T. Jevočin, and B. Buhnova, "Towards process mining utilization in insider threat detection from audit logs," in *Proceedings of the 2020 7th International Conference on Social Networks Analysis, Management and Security (SNAMS)*, pp. 1–6, Paris, France, December 2020.

[2] S. Honda, Y. Unno, K. Maruhashi, M. Takenaka, and S. Torii, "Topase: Detection of brute force attacks used disciplined IPs from IDS log," in *Proceedings of the 2015 IFIP/IEEE 1st International Workshop on Security for Emerging Distributed Network Technologies (DISSECT)*, pp. 1361–1364, Ottawa, Canada, May 2015.

[3] Q. Fu, J. Lou, Y. Wang, and J. Li, "Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis," in *Proceedings of the 2009 9th IEEE International Conference on Data Mining (ICDM)*, pp. 149–158, Miami, Florida, USA, December 2009.

[4] C. Bertero, M. Roy, C. Sauvanaud, and G. Tredan, "Experience Report: Log Mining Using Natural Language Processing and Application to Anomaly Detection," in *Proceedings of the 2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 351–360, Toulouse, France, October 2017.

[5] 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 *Proceedings of the 23rd International Conference on Database and Expert Systems Applications (DEXA)*, pp. 31–38, Vienna, Austria, September 2012.

[6] D. Q. Zou, H. Qin, and H. Jin, "Uilog: improving log-based fault diagnosis by log analysis," *Journal of Computer Science and Technology*, vol. 31, no. 5, pp. 1038–1052, 2016.

[7] Y. Guo, J. Liu, W. Tang, and C. Huang, "Exsense: extract sensitive information from unstructured data," *Computers & Security*, vol. 102, Article ID 102156, 2021.

[8] J. Aayushi, "Splunk Tutorial for Beginners: Explore Machine Data with Splunk," 2021, https://www.edureka.co/blog/splunk-tutorial.

[9] S. Chen, C. Guo, X. Yuan, F. Merkle, H. Schaefer, and T. Ertl, "OCEANS: online collaborative explorative analysis on network security," in *Proceedings of the 11th Workshop on Visualization for Cyber Security*, pp. 1–8, Paris, France, November 2014.

[10] H. Susan, "Logz.io Adds a Cast of Thousands to Help with Log Analysis," 2017, https://thenewstack.io/logz-adds-cast-thousands-help-log-analysis/.

[11] Loggly, "Automated Parsing Log Types," 2014, https://www.loggly.com/docs/automated-parsing/.

[12] L. Zheng, S. Ma, Y. Wang, and G. Lin, "String generation for testing regular expressions," *The Computer Journal*, vol. 63, pp. 41–65, 2020.

[13] "Welcome to Apache Flume," 2009, http://flume.apache.org/.

[14] H. Ismael, "A Practical Introduction to Elasticsearch," 2021, https://www.elastic.co/cn/blog/a-practical-introduction-to-elasticsearch.

[15] A. Kafka, 2017.

[16] P. He, J. Zhu, Z. Zheng, and R. Michael, "Lyu. Drain: an online log parsing approach with fixed depth tree," in *Proceedings of the 2017 IEEE 24th International Conference on Web Services (ICWS)*, pp. 33–40, Honolulu, HI, USA, June 2017.

[17] M. Du and F. Li, "Spell: online streaming parsing of large unstructured system logs," *IEEE Transactions on Knowledge and Data Engineering*, vol. 31, no. 11, pp. 2213–2227, 2019.

[18] A. A. O. Makanju, A. N. Zincir-Heywood, E. Evangelos, and Milios, "Clustering event logs using iterative partitioning," in *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD)*, pp. 1255–1264, Paris, France, July 2009.

[19] S. Messaoudi, A. Panichella, D. Bianculli, L. Briand, and R. Sasnauskas, "A search-based approach for accurate identification of log message formats," in *Proceedings of the 26th Conference on Program Comprehension (ICPC)*, pp. 167–177, Gothenburg, Sweden, May 2018.

[20] J. Zhu, S. He, J. Liu et al., "Tools and benchmarks for automated log parsing," in *Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 121–130, Montreal, Quebec, Canada, May 2019.

[21] M. Nagappan and A. Mladen, "Vouk. Abstracting log lines to log event types for mining software system logs," in *Proceedings of the 2010 7th IEEE Working Conference on Mining Software Repositories (MSR)*, pp. 114–117, Cape Town, South Africa, May 2010.

[22] R. Vaarandi and M. Pihelgas, "LogCluster - a data clustering and pattern mining algorithm for event logs," in *Proceedings of the 2015 11th International Conference on Network and Service Management (CNSM)*, pp. 1–7, Barcelona, Spain, November 2015.

[23] L. Tang, T. Li, and Chang-Shing Perng, "LogSig: generating system events from raw textual logs," in *Proceedings of the 20th ACM international conference on Information and knowledge management (CIKM)*, pp. 785–794, Glasgow, Scotland, October 2011.

[24] H. Hamooni, B. Debnath, J. Xu, H. Zhang, G. Jiang, and M. Abdullah, "LogMine: fast pattern recognition for log analytics," in *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management (CIKM)*, pp. 1573–1582, Indianapolis, USA, October 2016.

[25] W. Xu, L. Huang, A. Fox, D. Patterson, I. Michael, and Jordan, "Detecting large-scale system problems by mining console logs," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP)*, pp. 117–132, Big Sky, Montana, USA, October 2009.

[26] M. Nagappan, K. Wu, and A. Mladen, "Vouk. Efficiently extracting operational profiles from execution logs using suffix arrays," in *Proceedings of the 2009 20th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 41–50, Bengaluru-Mysuru, India, November 2009.

[27] M. Du and F. Li, "Spell: streaming parsing of system event logs," in *Proceedings of the 2016 IEEE 16th International Conference on Data Mining (ICDM)*, pp. 859–864, Barcelona, Spain, December 2016.

[28] A. Agrawal, R. Karlupia, and R. G. Logan, "A distributed online log parser," in *Proceedings of the 2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pp. 1946–1951, Macau, China, April 2019.