applied sciences

MDPI

Article

CFTL: System Log Parsing Method Driven from Clustering According to First Token and Length for Anomaly Detection

Jie Hu [1,2], Lingchun Long [3], He Sui [3,*], Zhaojun Gu [4] and Guangming Zheng [5]

[1] The 28th Research Institute, China Electronics Technology Group Corporation, Nanjing 210007, China; hujie5@cetc.com.cn
[2] State Key Laboratory of Air Traffic Management System, Nanjing 210007, China
[3] College of Aeronautical Engineering, Civil Aviation University of China, Tianjin 300300, China; 18873261035@126.com
[4] Information Security Evaluation Centre, Civil Aviation University of China, Tianjin 300300, China; zjgu@cauc.edu.cn
[5] School of Finance, Nankai University, Tianjin 300000, China; kaworu922@126.com
* Correspondence: hsui@cauc.edu.cn

Abstract: Logs are extensively used to analyze the running states of systems in many fields, such as cyber security, performance monitoring, and fault diagnosis, by recording events representing real-time system states. Conventional system log parsing methods are time-consuming and are prone to overfitting or underfitting. Thus, automatic parsing methods, known as log parsers, have been proposed. However, most log parsers do not exhibit both high accuracy and low running time. In addition, they typically overfit during log template generation; thus, log parsers cannot effectively be deployed in large-scale distributed systems. To solve these problems, this study proposes an efficient heuristic log parsing method. The proposed method first clusters log messages by the first token and their length, then uses specific separation rules to divide them into refined groups, and finally matches the corresponding log templates. The performance of the proposed method was evaluated using reliable datasets and tools. The experimental results demonstrate that the proposed method not only exhibits high accuracy but also a low running time. The number of log events parsed by log parsers of log templates generated by the proposed method is close to the real count, and the proposed method also exhibits good accuracy for subsequent anomaly detection tasks. In addition, the proposed method is lightweight because it employs the most valuable parsing rules. Thus, the proposed method is suitable for deployment in large-scale distributed systems.

Keywords: system log; log parsing; clustering; first token

check for updates

Academic Editor: Andrea Prati

Received: 30 October 2024
Revised: 22 January 2025
Accepted: 30 January 2025
Published: 8 February 2025

Citation: Hu, J.; Long, L.; Sui, H.; Gu, Z.; Zheng, G. CFTL: System Log Parsing Method Driven from Clustering According to First Token and Length for Anomaly Detection. *Appl. Sci.* **2025**, *15*, 1740. https://doi.org/10.3390/app15041740

Copyright: © 2025 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

The scale of computer distributed systems is expanding, and these systems play a critical role in our daily lives. Whenever abnormal conditions occur in large-scale distributed systems, the corresponding company or enterprise suffers inestimable losses. Therefore, the stability of these systems is critical. System anomalies are typically stored in system logs, regardless of whether they are caused by hardware or software factors. To locate specific anomaly information, data mining techniques are applied to system logs for anomaly detection [1–5], fault diagnosis [6–8], performance monitoring [9,10], and other operations.

Logs are essential in the development and maintenance of systems because they record the operation and state of these systems. A large amount of log data exist on the server in large-scale distributed systems, including system, network traffic, application, and

audit logs. Among them, system logs are the most important; they record the events and behaviors of various system activities and are typically semi-structured text output by programmers using print statements such as when writing source code.

An unstructured log message typically includes a raw message comprising constant and variable components, with the timestamp and description information representing the event time and its emergency level, respectively. In a raw message, the constant part always consists of log print statements in the source code; in contrast, the variable part may depend on specific operations. A specific system operation is described by a raw message. Therefore, it can obtain the log events of each log message using some methods and then apply log events to represent each log message, thereby converting them into structured messages [11] (Figure 1).

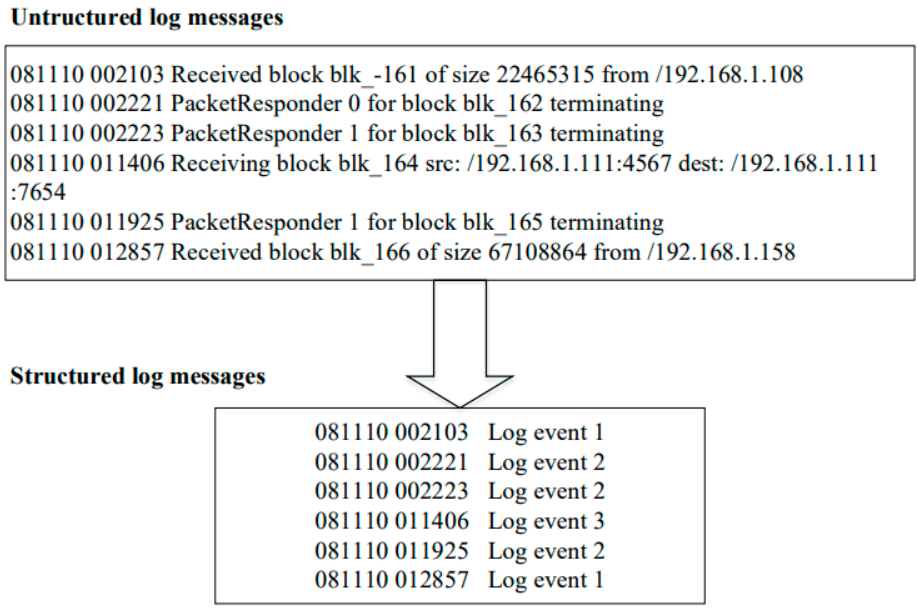

Figure 1. Conversion of unstructured log messages into structured log messages.

The conversion of unstructured log messages into structured log messages is important because most data mining models require structured data as input, and these data mining models are helpful for log analysis techniques. Therefore, for log analysis, it is important to first identify the log event of each log message. For example, two log messages are presented in Figure 2. In the first log message, "081109 203519" is the timestamp, and "PacketResponder 1 for block blk16089 terminating" is its raw message content part, where "PacketResponder", "for", "block", and "terminating" constitute the constant part, and "1" and "blk 168" constitute the variable part. "PacketResponder * for block * terminating" represents the log event, where "*" represents a variable. A log file contains multiple log events. Log messages belonging to the same event type always have the same constant message. However, the variable message differs as the event occurs. In general, log output statements always differ; thus, the format of log messages changes frequently. When the number of log messages is small, manual reading can be used to obtain log event templates. However, when the size of log files increases or there are many variations in log events, algorithms and tools are required to identify log event templates automatically from log messages. The extraction of log event templates is not only convenient for the corresponding analysis but also saves memory and queries and summarizes logs into a compact template.

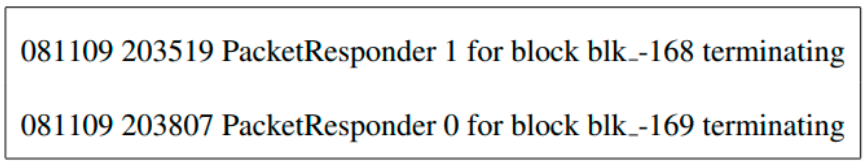

Figure 2. Example of log messages.

Before the emergence of large-scale distributed systems, the system log scale was small, and manual log parsing could be effectively performed using simple regularization expressions and other methods. Conventional log parsing methods generally require developers to manually create a regular expression list to match each log message, which depends on the output of the source code. Thus, if the output statement changes, the regular expression list needs to be updated. When there are many log output statements in a system and the update frequency is high, manually creating and updating the regular expression list become a laborious task. If the task is performed by experts or developers with domain knowledge, efficiency is improved. However, this approach is not promising, particularly in environments with different languages and frameworks. In addition, when large-scale distributed systems are widely used, the number of logs increases rapidly. As the number of event types increases, new services and functions are rapidly iterated. Manual log parsing cannot satisfy the requirements of large-scale distributed systems.

Previous studies have proposed various log parsing methods, including the log-based ELK algorithm [10], LenMa algorithm [12], and MoLFI algorithm [13]. These methods only require log files as input and employ clustering or evolutionary algorithms to automatically extract templates. Many techniques for automatically extracting templates have been proposed, with the goal of accurately and rapidly parsing log files.

These techniques can be divided into four main categories: frequent pattern mining-based methods, clustering-based methods, heuristic rules-based methods, and machine learning-based methods. Each category has its advantages and disadvantages. The system log messages of distributed servers differ from other log data. These messages exhibit unique characteristics, such as tokens and length, that can be used as enlightening prior knowledge to efficiently extract the common part of the log message as templates. AEL [14] extracts log templates based on the occurrence frequency of constant and variable tokens; IPLoM [15] extracts templates based on the log length and token mapping; Drain [16] constructs a parsing tree based on the log length and token prefix and takes leaf nodes as templates; Spell [17] uses the longest common subsequence (LCS) for log analysis based on prefix trees.

Accuracy and time efficiency are generally considered the most adequate indicators in this field. Although the aforementioned methods achieve good results, they have limitations, such as insufficient precision and low efficiency. The fundamental reason is that the system logs of large-scale distributed systems do not improve mining with specific heuristic rules in terms of token and length. These methods use only general rules, resulting in too many or insufficient template parameters: overfitting or underfitting.

To solve these problems, by analyzing the common characteristics of the system logs of large-scale distributed systems (Linux, Windows, Hadoop, Android, Apache, Spark), we find that the first token and log length are the two most enlightening factors. Therefore, we propose an efficient log parsing method with heuristic rules by clustering based on the first token and length (CFTL). The proposed method automatically and accurately identifies log event templates corresponding to raw messages. Compared with existing log parsers, CFTL is a relatively lightweight algorithm that exhibits higher accuracy and efficiency. The main contributions are summarized as follows:

(1) An efficient system log parsing method with heuristic rules was proposed to improve the accuracy and efficiency of large-scale edge system log parsing.
(2) Key local features and general global features were both used as heuristic rules, where the first token was the key local feature recording system behavior, and the length was selected as the general global feature. This approach not only ensured analytical accuracy but also prevented underfitting. It also overcame the excessive consideration of tokens and focused on the excessive overfitting of template parameters caused by non-key local features.
(3) The separation algorithm of constant and variable tokens and the LCS algorithm were both optimized. Based on the property of the optimal substructure, a recursive relationship of optimal values was established, which significantly reduced the complexity of rule separation and template matching, effectively improving efficiency.
(4) Experiments on several large-scale distributed server log datasets demonstrate that the proposed CFTL method exhibits obvious advantages in terms of accuracy, efficiency, and lightweightness.

The remainder of this paper is organized as follows: Section 2 introduces related studies on log parsing. Section 3 describes the proposed method in detail. Section 4 presents the experimental methodology. Section 5 reports and discusses all experimental ideas and results. Finally, Section 6 presents the conclusion and future works.

2. Related Works

Logs are ubiquitous in every system. As essential data, logs are typically used by developers to diagnose system abnormalities and maintain system stability and security. Log parsing is critical to effective log analysis. After parsing, the log information is converted into formatted log messages to promote the downstream data mining model. Anomaly detection is one of the most common downstream applications. It can be used for both unsupervised and supervised learning. In unsupervised methods, a time window is established according to the parsed log message template. In the time window, the occurrence times of log message events are encoded as vectors, and unsupervised methods, such as principal component analysis and clustering, are then used for anomaly detection. In supervised anomaly detection, the training data are first labeled, including abnormal and normal conditions. Representative supervised methods are logistic regression, decision trees (DTs), and support vector machines (SVMs).

Log parsing transforms unstructured log messages into structured log messages, and it is an important foundation for anomaly detection based on logs, which is used in the traditional manual inspection of system logs to detect exceptions. The simple logfile clustering tool (SLCT) automatically identifies log message templates [18].

Existing log parsers are primarily based on frequency pattern mining, clustering, heuristic rules, and machine learning.

2.1. Frequency Pattern Mining-Based Methods

Log parsing algorithms using frequency pattern mining include SLCT, LFA [19], LogCluster [20], and Logram [21]. Their strategy primarily relies on traversing log messages multiple times and counting frequently appearing words or a certain format in log messages to cluster log messages into groups to find log message templates corresponding to all log messages.

They first build a frequent item set by traversing the log multiple times, and based on tokens, token location pairs, and continuous N-byte combinations, they divide the log messages into clusters and extract templates from them. SLCT first applies frequency pattern mining to log parsing. Based on SLCT, LFA considers the distribution of token frequency,

and LogCluster, which is an extension of SLCT, further considers the unknown changes in tokens. Logram considers a sequence of n consecutive items that occur frequently.

2.2. Clustering-Based Methods

Clustering-based parsing algorithms include LKE [10], LenMa [12], LogSig [22], SHISO [23], and LogMine [24]. These methods employ various strategies to cluster log messages into groups, such as the weighted edit distance strategy, in which similar logs are clustered into groups according to the weighted distances among different log messages. After the log messages are grouped, the log message templates for each group are constructed and used to represent the logs within each group.

LKE and LogMine use hierarchical clustering to calculate the weighted distance and edit the distance to divide the initial group. These methods further split the initial group according to different numbers of values and entropy or gradually relax the threshold from the leaf node template in a bottom-up way to form a general log template. LogSig first converts each log message into a set of item pairs, including lexical items, retains the position order, partitions the items into a predefined number of clusters, and continuously optimizes the number of partitions. Finally, message signatures are constructed based on the common pairs identified by each message group. As an online method, SHISO processes log data in a streaming manner, dynamically clustering logs by building node data structures, and matching and updating existing new logs with existing log templates.

2.3. Heuristic Rule-Based Methods

Although logs are unstructured, a class of logs occasionally has certain composition rules that can be used for heuristic learning. This type of method is referred to as heuristic rule-based log parsing, e.g., MoLFI [13], AEL [14], IPLoM [15], Drain [16], and Spell [17].

MoLFI employs an evolutionary algorithm to select the most suitable template from the log messages. AEl divides log messages into groups by comparing the number of occurrences between constant and variable tokens. IPLoM iteratively partitions logs based on log length, unique token location, and token pair mapping. Drain groups logs by log length and token prefix to build a parse tree structure with a fixed depth and fixed number of branches, whose leaf nodes are log templates. Spell realizes streaming log analysis based on a prefix tree structure using LCS.

2.4. Machine Learning-Based Methods

In recent years, due to the gradual advancement of intelligent models with strong learning and autonomous decision-making abilities, machine learning methods have been applied to the field of log analysis, which primarily includes offline training and online analysis.

UniParser [25] trains an encoder by learning patterns from a log token and its adjacent context from a semantic perspective. Based on token saturation progressive clustering, SPINE [26] forms a binary tree structure and extracts templates from leaf nodes in combination with personnel feedback guidance. LogParse [27] trains a binary classifier to distinguish between template and variable words based on templates learned from historical logs. LogStamp [28] transforms the log parsing problem into a sequence annotation problem, learns embedded features from word and sentence levels, and predicts whether a word is a template or a variable during real-time parsing.

2.5. Brief Summary

Unlike conventional unstructured text data, log messages, particularly server system logs, typically exhibit unique characteristics despite structural differences. These features can be used as prior knowledge to construct log templates, and the proposed method adopts this perspective.

Existing log parsing methods can obtain the templates corresponding to each log message; however, there are defects, such as wrong or incorrect log templates and overfitting and underfitting problems.

Among them, Drain and IPLoM exhibit the best performance on average, and MoLFI is the most recent. In this study, the proposed parsing method is compared to them. Clustering is a common data mining technique that classifies data into clusters [14,16,29]. The log parsing step typically involves aggregating similar log messages into one category using various strategies and then performing subsequent processing. A unique characteristic of the proposed CFTL method is that its clustering strategy is based on heuristic rules, and each log message group is processed in detail. Thus, the proposed method obtains better results than existing methods. In addition, log parsing methods can improve existing log management software systems, such as Splunk 9.3.0 and ELK 7.7.0 [30–32]. These systems integrate the functions of log collection, storage, indexing, searching, filtering, analysis, and visualization. Combined with automatic log parsing methods, they can compensate for shortcomings and make their functions more complete.

3. CFTL Log Parsing Method

3.1. Motivation

The log parsing process learns the features of historical log data. Similar to the general machine learning process, the selection of features has a profound impact on learning outcomes. Too many selected features may lead to overfitting. Otherwise, underfitting may occur. Log data features can be divided into global and local features. Global features include log length, token spatial distribution, and token ratio of constants and variables. Local features refer to the specific token information, the specific token location, and the prefix and suffix information of the token. These features provide heuristic rule information for log parsing and provide an important basis for heuristic rule-based methods.

The strategies used in heuristic rules heavily depend on the experience of the researcher. Researchers establish rules by observing these log message characteristics, and analytic methods rely on these rules to perform clustering and finally obtain log templates. AEL analyzes logs using the ratio of constants to variables. It considers only the global features of logs, excluding local features, and the extracted heuristic rules are insufficient; thus, it easily underfits in actual applications. IPLoM considers global features such as the log length and local features such as unique token locations and token pair mappings. Drain and Spell further complement the local feature of the token prefix. These methods extract an excessive number of heuristic rules, which results in high training accuracy but can easily lead to overfitting and poor generalizability.

These findings demonstrate that both the global and local features of log messages should be considered; however, the extracted heuristic rules should be maintained at an appropriate granularity to balance training and practical application performance. The log length is widely recognized as the most representative global feature. To prevent overfitting of local features, the scale and complexity of selection for the number of tokens and related information should be controlled. This study focuses on the system logs of distributed servers. Different types of logs have different lengths, whereas logs of the same type have similar structures and token distributions. The most significant feature is the first token difference. Therefore, in the proposed CFTL method, the log length and first token are selected as global and local features, respectively, as a heuristic rule to train the analytic model to avoid underfitting. In addition, an excessive number of tokens or their auxiliary information is not selected as a rule to avoid overfitting. In summary, the combination of the log length and first token provides an appropriate heuristic rule granularity for the system logs of distributed servers, which prevents overfitting and underfitting.

3.2. CFTL Method Framework

To overcome these challenges, we design a CFTL log parsing algorithm and use it to achieve the following goals: (1) high resolution accuracy and good running time efficiency; (2) after CFTL processing, overfitting mitigation using the log message event template; (3) simple parsing rules.

The outline of the proposed CFTL method, which comprises four steps, is shown in Figure 3. To parse a log file, CFTL first preprocesses the file and then performs clustering, separation, and extraction.

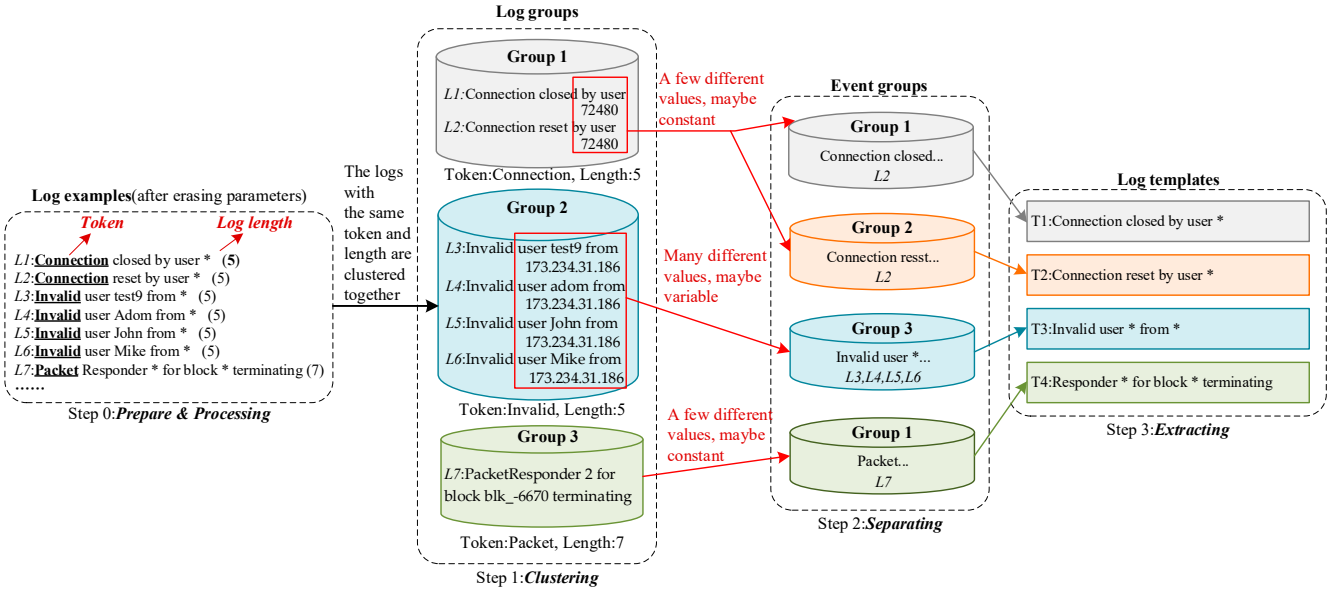

Figure 3. Outline of the proposed CFTL method, * is the simple parameters in the log event.

3.3. Algorithmic Details of CFTL

After introducing the overall processing flow of CFTL, we describe how CFTL parses raw log messages, as shown in the second row of Table 1.

Table 1. System logs and templates involved in CFTL.

CFTL Steps	Log No.	Log Attribution	Group No.	Log and Template Content
Raw log file	1	Raw log Messages	/	Connection closed by user 72480
	2		/	Connection reset by user 72480
	3		/	Invalid user test9 from 173.234.31.186
	4		/	Invalid user adom from 173.234.31.186
	5		/	Invalid user John from 173.234.31.186
	6		/	Invalid user Mike from 173.234.31.186
	7		/	PacketResponder 2 for block blk_-6670 terminating
Preprocessing (Step 0)	1	Logs after Parameter Erasing	/	Connection closed by user *
	2		/	Connection reset by user *
	3		/	Invalid user test9 from *
	4		/	Invalid user adom from *
	5		/	Invalid user John from *
	6		/	Invalid user Mike from *
	7		/	PacketResponder * for block * terminating

Table 1. *Cont.*

CFTL Steps	Log No.	Log Attribution	Group No.	Log and Template Content
Clustering (Step 1)	1	Processed Log Messages (Grouped or clustered by the first token and the length of log messages)	1	Connection closed by user *
	2			Connection reset by user *
	3			Invalid user test9 from *
	4		2	Invalid user adom from *
	5			Invalid user John from *
	6			Invalid user Mike from *
	7		3	PacketResponder * for block * terminating
Separating (Step 2)	1	Processed Log Messages (Making clustering results more accurate)	1	Connection closed by user *
	2		2	Connection reset by user *
	3			Invalid user test9 from *
	4		3	Invalid user adom from *
	5			Invalid user John from *
	6			Invalid user Mike from *
	7		4	PacketResponder * for block * terminating
Extracting (Step 3)	1	Generated Log Templates (Events)	1	Connection closed by user *
	2		2	Connection reset by user *
	3		3	Invalid user * from *
	4		4	PacketResponder * for block * terminating

Preprocessing. Log file preprocessing using the CFTL method is very simple with regular expressions. Similar to other log parsing algorithms, specific parameters, such as IP addresses, and special symbols (colons or equal signs) are erased by empirical rules. This simple preprocessing improves log parsing accuracy. The CFTL method preprocesses log files by defining regular expressions that match log parameters and then erasing them. To improve the accuracy of CFTL and its ability to handle the overfitting challenges associated with the final templates in the preprocessing step, simple parameters can be erased. The advantages of this approach are obvious. When there is an uncertain number of simple parameters in the same log event, replacing them with "*" results in a situation in which the log length for the same type of event may not be the same, thereby affecting the final result. However, if simple parameters are directly erased, after preprocessing log messages with different simple parameters belonging to the same type of event, their log lengths are the same.

Clustering. The proposed CFTL method employs a two-step clustering strategy. First, whether the first word is the same is considered for clusters; then, the length of log messages is also considered, as shown in Figure 4.

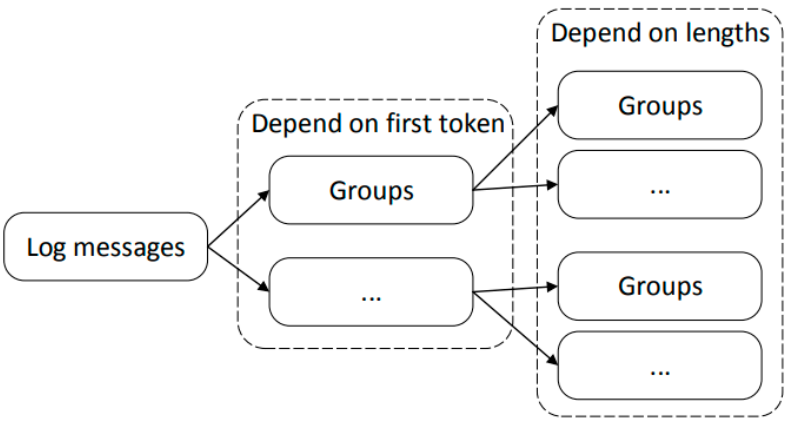

Figure 4. Clustering step.

Typically, the first word in each log message contains the most important information, because the process name is typically located in the first word. For example, the first word in an error message is typically "Error", and it may be "Warning" for a warning message. Therefore, log messages should be clustered primarily based on their first words. Messages with the same first words should be partitioned into the same cluster. For some software systems, the use of the first word token as a classification mark is highly accurate.

However, in other scenarios, using the first word alone for clustering is inaccurate. Different types of events may have the same first words in log records. In this case, additional information is required to improve clustering accuracy. The length of the log message is an appropriate token because the number of words in different print statements typically differs. With both the same first word and log message length, related log messages can be clustered more accurately. The proposed method employs a two-step clustering strategy: the first word is primarily considered for clustering, and the log message length is supplemented after the initial clustering process depending on the first word. In other words, clusters based on length are subclusters divided by the first word. Finally, log messages with the same first word and length are clustered into a group.

Separating. Although log messages in each cluster have the same first word and length, they may record different cases and are output by different log print statements. For example, almost all log messages record two connection events, with the keyword "successful" or "failed" in the log files. However, if the keyword "successful" or "failed" is not the first word, the clustering strategy in step 1 clusters them into the same group. In this step, this problem is solved using a separation mechanism.

As mentioned previously, a log message always comprises two types of components: constant and variable components. The former typically contains fixed information containing the keywords of log messages, whereas the latter typically contains changing information, along with the system running and event occurrence data. In each set of log messages, if the constant components are very similar, they can easily be treated as log events. Taking the first and second log messages in the fifth row of Table 1 as an example, they have the same first word and length. However, they represent two events: an event indicating that the connection is closed, and another event indicating that the connection is reset. These two log messages are grouped in the first step, and if the common component is extracted as the log message template, the two constants "closed" and "reset" are treated as variables and, thus, represented by "*." Thus, the next step is to accurately separate the variables and constants and each subgroup of log messages within an initial group.

Algorithm 1 provides a pseudocode for the second step, i.e., the separation stage. The separation function is recursively called. CFTL continuously separates clustered log messages into new groups until each group contains the same constant component. After the initial clustering of the raw messages, the messages are clustered into several groups. For each group, CFTL uses the same separation rules to classify the log messages accurately. Tokens in all positions are counted, and a threshold (s) for separation is established. For example, assuming that the number of different tokens in the first position is calculated as n, CFTL counts from the first token of each log. If n exceeds the preset threshold s, the token at this position is most likely a variable component. Otherwise, it is considered a constant component. For variable tokens, no changes to the log messages are made, and the tokens are changed directly to the "$*$" wildcard in the extracted template. However, for constant tokens, the log messages are regrouped according to the tokens. Log messages with the same token are clustered into the same group.

After the first token is counted by CFTL, the token at the second position is processed until the token at the last position is processed. In addition, to improve the processing

speed, CFTL first saves the common subsequence within the group, because the words in the common subsequence are constants and do not need to be rechecked.

Algorithm 1: Separating

Input: a group of clustered log messages: *logGroup*
Output: several separated groups
function SEPARATING (*logGroup*, *st*, *pos*, *groups*)
//st means separation threshold
//pos records separation position
//groups is the result
sDict ← Empty dictionary
logLen ← the length of log messages in logGroup
if *pos* ≥ *logLen* **then**
  groups ← add *logGroup*
  return
end if
for each log message *logMsg* in *logGroup* **do**
  tk ← the pos^{th} token in *logMsg*
  if *sDict* doesn't have the same key as *tk* **then**
    sDict[*tk*] ← 1
  else
    sDict[*tk*] ← *sDict*[*tk*] + 1
  end if
end for
sDlen ← the number of keys in *sDict*
pos ← *pos* + 1
if *sDlen* < *st* **then**
  Separate log messages into different groups according to different keys in *sDict*
  Calling function SEPARATING with separated groups
else
  SEPARATING (*logGroup*, *st*, *pos*, *groups*)
end if
end function

The main goal of the proposed CFTL method is to solve the problem of operation system log parsing in distributed systems, such as Linux, Windows, and Android. The training process of the log parser, including the separation mechanism, is performed on the server side, and the trained log template is then deployed on the terminal device to compensate for limited terminal equipment resources. In addition, there are other reasons for this design.

During the communication between the end devices and servers, the original token is generated on the server side, which allows it to provide richer and more original first token information.

The abnormal behaviors of the end devices are primarily reflected in the interaction with the servers and, thus, in the system logs of the servers because most intrusion behaviors targeting the end of distributed systems are aimed at the ultimate unauthorized access to the servers. The end device is only a springboard. Highly subtle and multistep abnormal behaviors may not be reflected in the end device system logs.

End devices of the same type may have different exception logs. Thus, the log templates of different terminals may differ. Abnormal behaviors known to other end devices of

the same type but unknown to the local end cannot be identified. All end devices interact with servers, and the servers are the convergence of all abnormal end device behaviors; thus, parsing on the server can extract richer abnormal information.

Simplifying these rules without compromising accuracy can be challenging given the limited resources on the edge server side. In addition, overfitting can occur if the rules are too complex. CFTL simplifies the separation rules by considering constant token differences and using a simple threshold design, which is a modest way to establish rules. If constant token difference is not considered, logs with similar initial tokens or lengths but completely different meanings may be mistaken for the same type of logs, resulting in insufficient precision. However, if an excessive number of rules are considered, such as the mapping relationship between token pairs and the information before and after tokens, the rules will be too complicated. The simplification of rules and the accuracy of the algorithm are typically contradictory; there are no perfect simplification schemes, and we can only balance the accuracy and usability of the algorithm as much as possible.

Extracting template. In this step, we use LCS to extract the templates of each log message group; that is, the LCS of each group represents the template of the respective group.

LCS refers to finding the LCS among all subsequences of two or more known sequences, which can be used to complement sequences of different lengths. A subsequence is a set of elements that are not necessarily continuous but remain in a consistent order. The LCS algorithm is expressed as follows:

$$C[i,j] = \begin{cases} 0, i = 0 \; or \; j = 0 \\ C[i-1, j-1] + 1, i, j > 0 \; and \; x_i = \; y_j \\ \max\{C[i, j-1], C[i-1.j]\}, i, j > 0 \; and \; x_i \neq \; y_j \end{cases}, \quad (1)$$

where $C[i, j]$ denotes a dynamic regular array. Considering each word in the log as an element, LCS calculates the similarity as follows:

$$similarity = \frac{|\mathrm{LCS}(log, \; Template)|}{|\; Template|}, \quad (2)$$

where LCS(log, $Template$) denotes the LCS length of the log and template; $|\; Template|$ denotes the length of the log template. If the similarity is greater than the threshold, the log belongs to a known template. Otherwise, the similarity between the log and the next template is calculated. If the log does not match any template, a new template is added.

After obtaining each log message template, each log message corresponds to the template in sequence. We label the template and list it as events 1, 2, 3, etc. Finally, log events are used to represent the log messages to obtain the final structured log. The extraction step (the sixth row of Table 1) yields the extracted log message templates. Four log message templates are listed as events 1, 2, 3, and 4. These events can be used to represent seven raw log messages as "Event 1", "Event 2", "Event 3", "Event 3", "Event 3", "Event 3", and "Event 4".

However, the disadvantage of LCS is that it consumes a significant amount of time. According to the optimal substructure property of the LCS algorithm, a recursive relationship of optimal values is established, and the spatiotemporal complexity of LCS can be reduced by introducing one-dimensional data and recursive scanning.

4. Experimental Methodology

Design. We designed four sets of experiments to evaluate (1) log parsing performance, (2) anomaly detection performance, (3) lightweight performance, and (4) the impact of parameters.

Evaluation. For log parsing, we first assessed the number of log events parsed, which is a direct indicator of model performance. Then, accuracy and F1-measure are selected as evaluation criteria, which are the most commonly used metrics. To calculate accuracy and F1-measure, true positive (TP), true negative (TN), false positive (FP), and false negative (FN) are obtained. TP means that log messages belonging to the same template are clustered in the same group; TN means that log messages belonging to different templates are clustered in different groups; FP means that log messages belonging to different templates are clustered in the same group; FN means that log messages belonging to the same template are clustered in different groups. For anomaly detection, we used precision, recall, and F1-measure as evaluation criteria, which were also calculated using TP, TN, FP, and TN. For the lightweight evaluation, central processing unit (CPU) usage, running time, and memory consumption were measured. The CPU is the core resource of a server system, and the running time and memory consumption represent the time and space complexity, respectively. To evaluate the impact of parameters, we mainly considered two parameters: separation and LCS thresholds. All experiments were conducted 20 times, and the average results were used for analysis.

Dataset. Nine datasets from real-world applications and related studies were used as benchmark datasets from LogPai [32]: Hadoop distributed file system (HDFS), Windows, Thunderbird Supercomputer Server (Thunderbird), Hadoop MapReduce server (Hadoop), OpenSSH Server (OpenSSH), Apache HTTP server (Apache), Andriod system log (Android), high-performance cluster (HPC), and large-scale data processing platform (Spark). In addition to these datasets, different HDFS dataset sizes were selected to evaluate the execution time. We selected the 2 k version of these datasets. In addition, six size levels of the selected HDFS were used: 1 kb, 10 kb, 100 kb, 1 mb, 10 mb, and 100 mb. In the experiment to determine the number of log events, we used HDFS datasets with log message sizes of 70 K, 100 K, 1 M, 5 M, and 11 M. The actual number of log events was artificially labeled.

Comparison. To demonstrate the effectiveness of the proposed method, we compared it with four state-of-the-art methods, including Drain, IPLoM, and MoLFI. In addition, to make the experimental results more comparable, we included the log parser Spell in the comparison.

Methodology. For Darin, IPLoM, and MoLFI, we used the same parameter values as those reported in the original studies. For a fair comparison, the same preprocessing rules (for example, the same parameters, such as IP or numbers) to each log parser were used. The experiments were implemented using Python 2.7 on the same server running Windows 7 with 64 bits with an Intel(R) Core (TM) I7-6700 CPU and 16 GB RAM.

5. Results and Discussion

5.1. Log Parsing Performance

Numbers of log events. A good log parser should obtain the appropriate number of log events after parsing, which is close to the number of real log events in the log file. When a log parser overfits, it generates a larger number of log events because similar slightly different log messages are considered different events. Similarly, when a log parser underfits, it generates fewer log events because it considers that many similar log messages belong to the same event; thus, a log event matches many log messages. The publicly published HDFS dataset contains approximately 11 million log entries and only 29 events. We divided the dataset into sub-datasets of different sizes to evaluate the number of events generated by different log parsers. The results are presented in Figure 5. In particular, the ground truth was labeled manually.

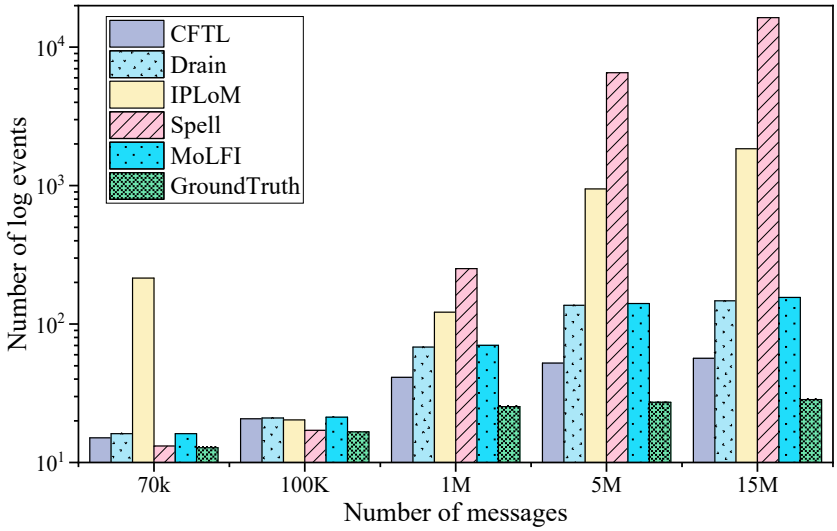

Figure 5. Numbers of log events parsed by log parsers for HDFS dataset.

The numbers of log events obtained by multiple log parsers were counted. As shown in Figure 5, the number of log events obtained by most log parsers exceeds the corresponding ground truth. The reason for this is that there are too many original logs, and the variables in the logs have too many variations. There may be multiple situations, e.g., the same event log has variables of different lengths (the number of words). Log parsers that only rely on the log length for parsing would be affected, resulting in different log events; thus, the number of log events is increased. Other log parsers treat variables as constants, and each different constant would have its event, thereby generating many different log events. Various situations affect the degree of fit of log parsers; thus, based on the number of log events, the pros and cons of log parsers can be assessed from another perspective.

As shown in the figure, the number of events generated by CFTL is closest to the real number. Many factors contributed to this result. First, the clustering strategy of CFTL can accurately cluster log messages into groups; second, to ensure that no additional log events are generated, CFTL performs special processing of variables and uses LCS to extract the corresponding events of each group. In contrast, Spell and IPLoM generate too many log events and, thus, overfit, which implies that they do not impose a constraint on the log message template, which results in unsatisfactory results. Both Drain and MoLFI impose a strong constraint on the number of log message templates. Drain uses a fixed tree depth. MoLFI constantly balances overfitting and underfitting during parsing.

Accuracy and F1-measure. Accuracy is the most important issue in log parsing. The accuracy metric indicates whether a log parser correctly matches messages with events. It can also prove the effectiveness of a log parsing method. A high-accuracy log parser not only generates accurate log message templates but also effectively performs subsequent log mining tasks.

The accuracy and F1-measure values of different log parsers are listed in Table 2. As shown in Table 2, the proposed CFTL method exhibits the best average accuracy and F1-measure results. In addition, the heuristic rule-based methods, such as Drain, exhibit higher accuracy than the other methods due to their specially designed rules. Among the various datasets, regardless of the log parser used, the HDFS and Apache datasets resulted in higher accuracy rates, reaching 99% in some cases. There are two main reasons for this. First, the log messages generated by the HDFS and Apache systems have a simple format and structure, which simplifies the parsing process. Second, the datasets contain log messages with small sizes, which reduces the difficulty of parsing.

Table 2. Accuracy and F1-measure of different approaches.

Dataset	MoLFI		Drain		IPLoM		Spell		CFTL	
	Acc	F1	Acc	F1	Acc	F1	Acc	F1	Acc	F1
HDFS	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99
Windows	0.40	0.91	0.99	0.99	0.56	0.99	0.98	0.99	0.99	0.99
Thunderbird	0.65	0.99	0.95	0.99	0.66	0.99	0.84	0.99	0.93	0.99
Hadoop	0.94	0.99	0.94	0.99	0.95	0.99	0.77	0.92	0.96	0.99
OpenSSH	0.48	0.99	0.78	0.99	0.80	0.99	0.55	0.91	0.80	0.99
Apache	1	1	1	1	1	1	1	1	1	1
Android	0.78	0.94	0.91	0.99	0.71	0.94	0.91	0.99	0.91	0.99
HPC	0.67	0.97	0.88	0.99	0.82	0.97	0.65	0.98	0.91	0.99
Spark	0.41	0.51	0.92	0.99	0.92	0.99	0.90	0.99	0.98	0.99

In the experiment, Drain used a tree structure with a fixed depth, which was more efficient for log message classification and resulted in a high accuracy rate. IPLoM adopted an iterative strategy; thus, it performed well on most datasets because of its effective heuristic rules. MoLFI, which used an evolutionary algorithm, requiring multiple iterations, did not have an advantage in terms of time. However, MoLFI used a multi-objective optimization strategy for the results; thus, it exhibited good accuracy on some datasets. Spell was the only log parser that used LCS; thus, its results were limited.

5.2. Anomaly Detection Performance

In the previous section, the log parsers were evaluated in terms of the number of generated templates, accuracy, and F1-measure. These aspects are critical for log parsers. If it is only to obtain the formatted results of log files, judging the advantages and disadvantages of log parsers from these aspects is sufficient. However, structured log messages parsed by log parsers are typically used for subsequent log mining models and are then used to analyze the system for anomaly detection, performance detection, etc. Therefore, the performance of log parsers should also be evaluated in subsequent anomaly detection tasks.

In this section, parsing performance is evaluated using a published HDFS dataset containing approximately 11 million log messages generated by approximately 570,000 HDFS blocks and 29 log event types. The HDFS blocks include normal and abnormal blocks, as labeled by the original author. We first used different log parsers to parse the HDFS dataset into structured log messages, and we then applied an open-source toolkit [33], taking structured log messages and ground truth as inputs, to obtain the required results. First, an event count matrix is generated from the structured log messages, where each row represents an HDFS block. Each column represents a log event type, and each cell counts the events on an HDFS block. Second, term frequency-inverse document frequency was used to preprocess the event count matrix. Finally, the event count matrix was input to the corresponding exception detection module for exception detection. The open-source toolkit provides six typical anomaly detection methods, including supervised and unsupervised methods. We primarily used the DT and SVM algorithms for evaluation, and the log parsing step used multiple parsers for comparison.

The HDFS dataset generates different numbers of log events after parsing by different log parsers. In addition, an HDFS block exhibits different log event sequences when different log parsers are applied, which results in different log event count matrices and anomaly detection results. We used five log parsers to perform the parsing task, and then, we input their parsing results into the exception detection module to obtain the results (Table 3).

Table 3. Performance of log parsing in anomaly detection.

		Train			Test		
		Precision	**Recall**	**F1**	**Precision**	**Recall**	**F1**
DT	Drain	0.99	0.99	0.99	1	0.98	0.99
	IPLoM	1	1	1	0.99	0.97	0.98
	Spell	1	1	1	1	0.99	0.99
	MoLFI	0.99	0.99	0.99	1	0.98	0.99
	CFTL	0.99	0.99	0.99	1	0.99	0.99
SVM	Drain	0.96	0.99	0.98	0.84	0.98	0.90
	IPLoM	0.97	0.99	0.98	0.84	0.98	0.90
	Spell	0.99	0.99	0.99	0.84	0.99	0.91
	MoLFI	0.96	0.99	0.97	0.83	0.98	0.90
	CFTL	0.96	0.99	0.97	0.83	0.99	0.91

The experimental results demonstrate that all log parsers perform well, demonstrating that after a log message passes the parsing process of a log parser, anomaly detection results with high accuracy can be obtained. For Spell and IPLoM, even though their log message templates were overfitted, better test results were obtained. Spell generated too many log templates, which performed particularly complex calculations during anomaly detection, resulting in excessive processing time. In general, generating an appropriate number of log message templates not only improves anomaly detection efficiency but also accuracy.

5.3. Lightweight Performance

In addition to accuracy, the advantages and disadvantages of log parsers can be evaluated based on their efficiency. In this section, lightweight performance is evaluated in terms of CPU usage, running time, and memory consumption.

CPU usage. As the computing capabilities of distributed devices increase, the burden on servers increases, which should be reduced. Thus, the log parsing methods were evaluated in terms of CPU resource consumption during operation. A relatively lightweight log parser is more suitable for deployment in large-scale distributed systems than a heavy log parser. We used the existing packages in the Python 3.9 environment to calculate the CPU usage at each moment and adopted the HDFS dataset containing 10 K log messages. The results obtained by the five methods are presented in Figure 6.

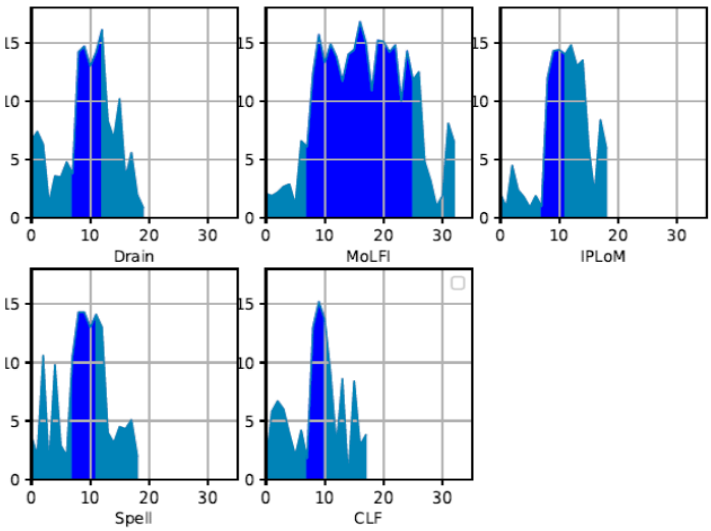

Figure 6. CPU usage of five log parsers.

The blue part indicates the CPU usage while the program is running, and the light-blue part indicates the CPU usage before and after the program runs. The results show that the resource consumption of the log parser at runtime is not gradual and has a peak. This is because all log data have been read and calculations have started. After the log parser finishes running, it releases the occupied resources. The experimental results indicate that CFTL is a lightweight log parser suitable for deployment on large-scale distributed servers.

Running time. To more intuitively analyze the time efficiency of the different methods, six HDFS dataset sizes, namely, 1 kb, 10 kb, 100 kb, 1 mb, 10 mb, and 100 mb, were used to evaluate their running times. The times required by the methods to complete the entire parsing process were recorded and compared on a logarithmic scale. The results are presented in Figure 7.

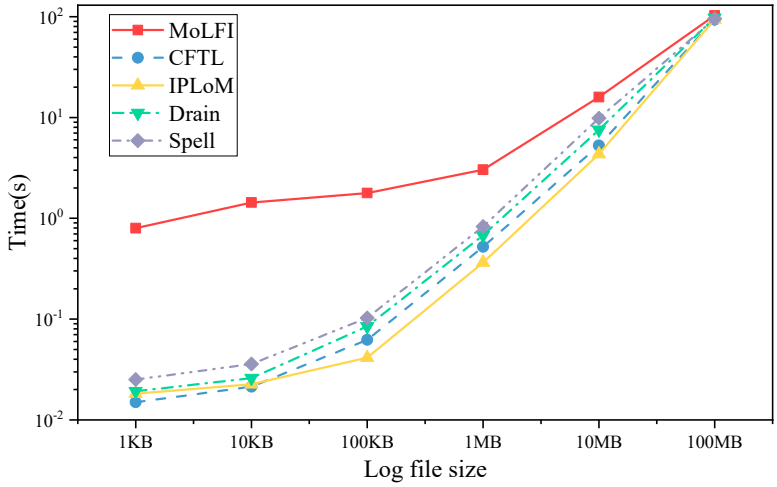

Figure 7. Running times of different log parsing methods.

In general, the proposed CFTL algorithm requires less time than the comparison log parsers. For the other methods, MoLFI was slower than Drain, IPLoM, and CFTL regardless of the dataset size. This is because MoLFI applies an iterative evolutionary strategy, which is time-consuming. IPLoM was the most efficient log parser in this comparative experiment. The running times of CFTL, IPLoM, and Drain were almost identical when the dataset size was less than 1 MB. However, as the size increased, the runtime of CFTL increased and was significantly slower than that of Drain but faster than that of IPLoM.

Memory consumption. When handling large log files, the memory consumption of the log parser should be considered an important aspect of performance in addition to accuracy. If the log parser uses excessive memory, other important programs of the server will be affected in terms of timeliness. This will also lead to a series of problems that reduce the communication efficiency between the server and edge device. The same six HDFS dataset sizes were used to evaluate the memory consumption of the log parsing methods. The results are presented in Figure 8.

In general, the proposed CFTL algorithm requires less memory than the comparison log parsers. For the other methods, MoLFI requires more memory space than Drain, IPLoM, and CFTL regardless of the dataset size. The reason for this is that MoLFI applies an iterative evolutionary strategy, which is memory-consuming. The memory consumption of the different methods was almost the same when the dataset size was less than 100 KB. However, as the size increased, the memory consumption of CFTL increased significantly more slowly than that of the other methods.

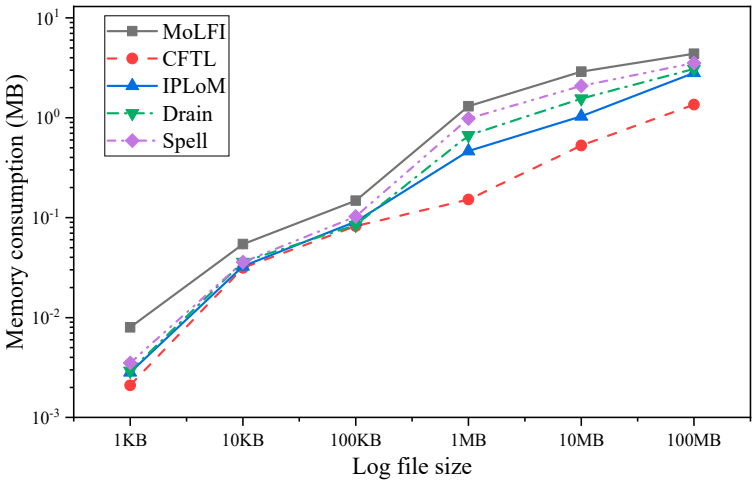

Figure 8. Memory consumption (MB) of different log parsing methods.

CFTL significantly reduces the runtime and memory consumption of log parsing compared with the other methods, particularly for large-scale log files. CFTL runs faster for the following reasons. On the one hand, its clustering strategy has fewer traversal runs, which reduces processing time. On the other hand, it is an offline log parser with linear time complexity and requires less processing time than online methods. Consequently, as the dataset size increases, the advantages of CFTL become more obvious.

5.4. Impact of Parameters

Understanding the sensitivity of the proposed CFTL method to parameter changes is necessary for practical deployment. Without this, CFTL will have very limited application and usability. The performance of the CFTL method is primarily affected by two parameters: separation and LCS thresholds. The former primarily affects the number of log templates, whereas the latter primarily affects the matching between new logs and templates.

Separation threshold. The separation threshold is typically set to a certain proportion based on the total number of token types in the log dataset. In this section, seven separation thresholds ranging from 0.65 to 0.95 are used in the experiment, and their impacts on the number of log templates parsed are evaluated. The results are presented in Table 4.

Table 4. Impact of different separation thresholds on number of log templates.

	0.65	**0.70**	**0.75**	**0.80**	**0.85**	**0.90**	**0.95**	**Actual**
Apache	4	6	6	6	6	6	6	6
HDFS	11	12	12	14	14	14	14	14
OpenSSH	19	21	25	26	27	27	27	27
Spark	22	23	28	31	32	36	36	36
HPC	21	27	32	35	40	44	46	46
Windows	31	37	42	47	48	49	49	50
Hadoop	83	96	104	107	109	110	112	114
Thunderbird	65	83	97	108	126	137	144	149
Android	101	109	114	123	139	150	157	166

Table 4 shows that the number of log templates increases as the separation threshold increases, indicating an improvement in log parsing accuracy. In addition, the number of actual log events may significantly affect the result. If the number of actual log events is smaller, a lower separation threshold level is sufficient to obtain better results. For example, when the separation threshold was 0.8 for HDFS, all events were parsed into log templates. However, at the same threshold for Android, only 123 templates were parsed, which was

only approximately 75% of the actual events. In other words, a smaller separation threshold may be suitable for datasets with fewer actual log events, and a larger threshold may lead to overfitting. However, a larger threshold appears better for datasets with more actual log events, and a smaller threshold may lead to underfitting.

LCS threshold. Five different LCS thresholds ranging from 0.80 to 0.99 were selected, and their impacts on accuracy and F1-measure were evaluated. The results are presented in Figure 9.

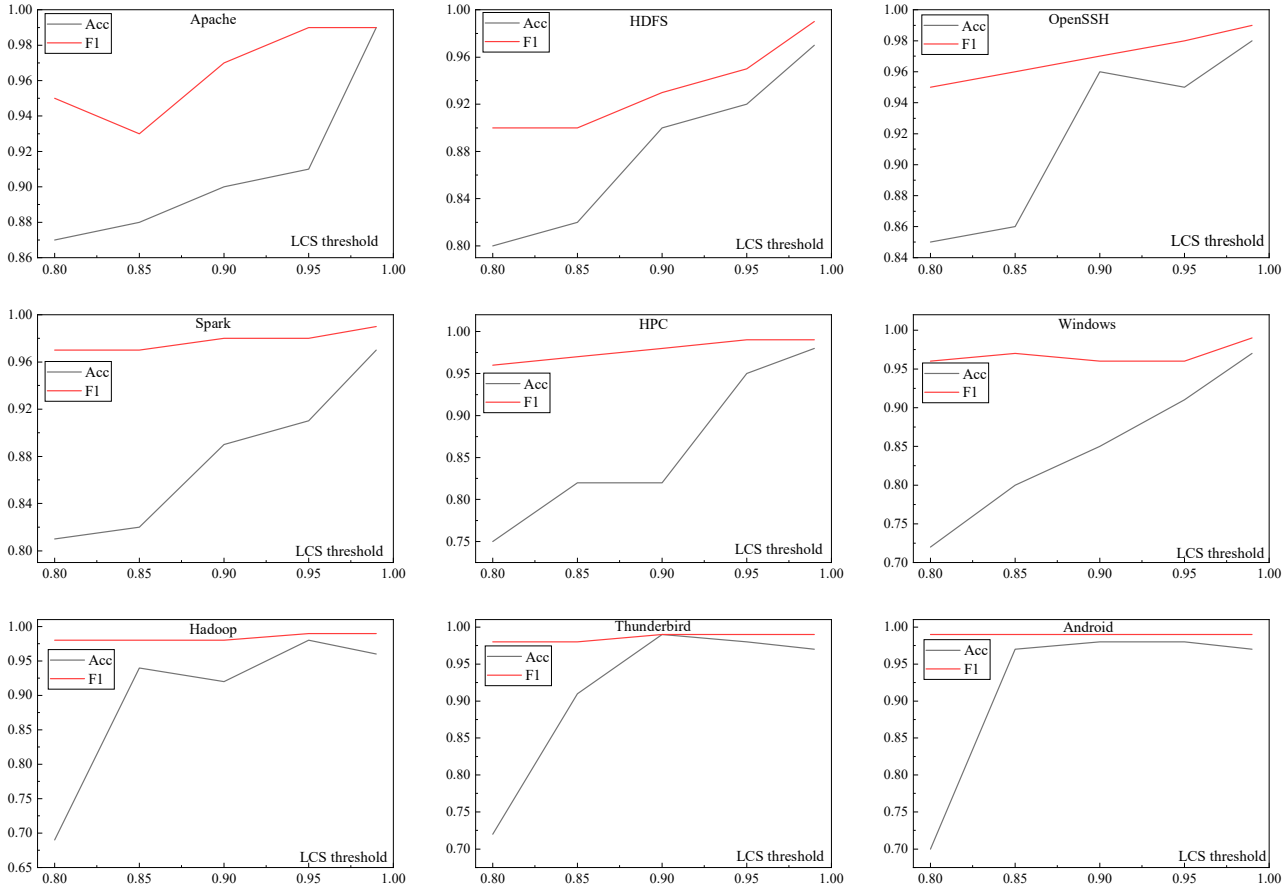

Figure 9. Impacts of different LCS thresholds on accuracy and F1-measure.

Figure 9 shows that the LCS threshold affects accuracy more than F1-measure. In particular, for log data with many actual events such as Hadoop, Thunderbird, and Android, the LCS threshold nearly exerts no effect on F1-measure. However, as the threshold increased, the accuracy first increased and then decreased. This result can be attributed to overfitting with larger LCS thresholds for these datasets. For log data with fewer actual events such as Apache, HDFS, and OpenSSH, a larger LCS threshold resulted in better results in terms of both accuracy and F1-measure. Smaller LCS thresholds may lead to underfitting. For log data with a moderate number of actual events such as Spark, HPC, and Windows, the LCS threshold primarily influences accuracy. In general, a smaller LCS threshold is more suitable for log data with fewer actual events, and vice versa.

6. Conclusions and Future Work

System logs are widely used to analyze system running states in many fields and record events representing real-time system states. However, most log parsers do not exhibit both high accuracy and low running time. In this study, we proposed an efficient log parsing method with heuristic rules by clustering based on its length and first token (CFTL). CFTL first clusters log messages by the first token and the log message length, then uses specific

separation rules to divide the messages into groups, and finally obtains the corresponding log template of each log message. The proposed CFTL method was evaluated on seven datasets and compared with four state-of-the-art log parsing methods in terms of accuracy, F1-measure, and efficiency. The experimental results show that CFTL not only exhibits higher accuracy but also a lower running time. The number of log events parsed by log parsers of log templates generated by CFTL is also closer to the real count, and CFTL also exhibits a good accuracy rate for subsequent anomaly detection modules. In addition, CFTL is more lightweight with the most valuable parsing rules; thus, it is suitable for anomaly detection tasks. In future work, we will further investigate the proposed method from the perspective of multimodel integration and large-scale practical deployment to promote its deployment in practical application scenarios as soon as possible. In addition, the structure of application logs differs, and the applicability of CFTL in these application unstructured logs is not discussed in this study. Other logs can only provide very few heuristic rules. How to further optimize the proposed CFTL method to be suitable for application logs, audit logs, and other unstructured logs is also a future research direction. In addition, first determining which token is most relevant and then determining the cluster should be adopted as a leading policy, which will further extend the scope of application of the proposed approach.

Author Contributions: Conceptualization, J.H.; methodology, H.S.; software, L.L.; validation, J.H., and G.Z.; formal analysis, H.S.; investigation, G.Z.; writing—original draft preparation, J.H.; writing—review and editing, H.S.; visualization, L.L.; supervision, Z.G.; project administration, Z.G. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by The States Key Laboratory of Air Traffic Management System, grant number SKLATM202306, and The Fundamental Research Funds for the Central one Universities of Civil Aviation University of China, grant number 3122023033.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: All data generated or analyzed during this study are included in this published article, and further requests can be sent to the corresponding author.

Conflicts of Interest: Author Jie Hu was employed by the company China Electronics Technology Group Corporation. The remaining authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Abbreviations

ELK	Elasticsearch, Logstash, Kibana
MoLFI	Multi-objective Log Format Identification
CFTL	Clustering based on First Token and Length
IPLoM	Iterative Partition Log Mining
SPELL	Sequence-based Pattern Extraction for Log Line
LFA	Log Feature Analysis
LKE	Log Key Extraction
LogSig	Log Signature
Uniparser	Unified Log parser
HDFS	Hadoop Distributed File System
LenMa	Length Matters
LCS	Longest Common Subsequence
AELDTSVM	Automated Event LogDecision TreeSupport Vector Machine
Drain	Depth Tree-based online log parsing
SLCT	Simple Logfile Clustering Tool

Logram	Log-based Anomaly Detection
LenMa	Length-based Matching
SHISO	Scalable Hierarchical Log Parsing
SPINE	Scalable Log Parser with Feedback Guidance

References

1. Bertero, C.; Roy, M.; Sauvanaud, C.; Tredan, G. Experience report: Log mining using natural language processing and application to anomaly detection. In Proceedings of the IEEE 28th International Symposium on Software Reliability Engineering (ISSRE), Toulouse, France, 23–26 October 2017; pp. 351–360.
2. Goldstein, M.; Raz, D.; Segall, I. Experience report: Log-based behavioral differencing. In Proceedings of the IEEE 28th International Symposium on Software Reliability Engineering (ISSRE), Toulouse, France, 23–26 October 2017; pp. 282–293.
3. Kc, K.; Gu, X. Elt: Efficient log-based troubleshooting system for cloud computing infrastructures. In Proceedings of the IEEE 30th International Symposium on Reliable Distributed Systems, Madrid, Spain, 4–7 October 2011; pp. 112–120.
4. Fu, X.; Ren, R.; Zhan, J.; Zhou, W.; Jia, Z.; Lu, G. Logmaster: Mining event correlations in logs of large-scale cluster systems. In Proceedings of the IEEE 31st Sympo-sium on Reliable Distributed Systems, Irvine, CA, USA, 8–11 October 2012; pp. 71–80.
5. Xu, W.; Huang, L.; Fox, A.; Patterson, D.; Jordan, M.I. Detecting largescale system problems by mining console logs. In Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles—SOSP, Big Sky, MT, USA, 11–14 October 2009; pp. 117–126.
6. Zou, D.Q.; Qin, H.; Jin, H. Uilog: Improving log-based fault diagnosis by log analysis. *J. Comput. Sci. Technol.* **2016**, *31*, 1038–1052. [CrossRef]
7. Wong, W.E.; Debroy, V.; Golden, R.; Xu, X.; Thuraisingham, B. Effective software fault localization using a rbf neural network. *IEEE Trans. Reliab.* **2012**, *61*, 149–169. [CrossRef]
8. Reidemeister, T.; Jiang, M.; Ward, P.A. Mining unstructured log files for recurrent fault diagnosis. In Proceedings of the 12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011) and Workshops, Dublin, Ireland, 23–27 May 2011; pp. 377–384.
9. Karthik, N.; Charles, K.; Jennifer, N. Structured comparative analysis of systems logs to diagnose performance problems. In Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12), San Jose, CA, USA, 25–27 April 2012; pp. 353–366.
10. Fu, Q.; Lou, J.G.; Wang, Y.; Li, J. Execution anomaly detection in distributed systems through unstructured log analysis. In Proceedings of the 9th IEEE International Conference on Data Mining, Miami, FL, USA, 6–9 December 2009; pp. 149–158.
11. JLou, G.; Fu, Q.; Yang, S.Q.; Xu, Y.; Li, J. Mining invariants from console logs for system problem detection. In Proceedings of the 2010 USENIX Annual Technical Conference (USENIX ATC 10), Boston, MA, USA, 23–25 June 2010.
12. Shima, K. Length matters: Clustering system log messages using length of words. *arXiv* **2016**, arXiv:1611.03213.
13. Messaoudi, S.; Panichella, A.; Bianculli, D.; Briand, L.; Sasnauskas, R. A search-based approach for accurate identification of log message formats. In Proceedings of the 26th Conference on Program Comprehension, Gothenburg, Sweden, 28–29 May 2018; pp. 167–177.
14. AMakanju, A.; Zincir-Keywood, A.N.; Milios, E.E. Clustering event logs using interative portioning. In Proceedings of the 15th ACM SIGKDD International Conference on Knowledge and Data Mining, Paris, France, 28 June–1 July 2009; ACM Press: New York, NY, USA, 2009; pp. 1255–1264.
15. Makanju, A.A.; Zincir-Keywood, A.N.; Milios, E.E. A lightweight algorithm for message type extraction in system application logs. *Trans. Knowl. Data Eng.* **2012**, *24*, 1924–1936. [CrossRef]
16. He, P.J.; Zhu, J.M.; Zheng, Z.B.; Lyu, M.R. Drain: An online log parsing approach with fixed depth tree. In Proceedings of the 2017 IEEE International Conference on Web Services (ICWS), Honolulu, HI, USA, 25–30 June 2017; IEEE Press: Piscataway, NJ, USA, 2017; pp. 33–40.
17. Du, M.; Li, F.F. SPELL: Streaming parsing of system event logs. In Proceedings of the 2016 IEEE 16th International Conference on Data Mining (ICDM), Barcelona, Spain, 12–15 December 2016; IEEE Press: Piscataway, NJ, USA, 2016; pp. 859–864.
18. Vaarandi, R. A data clustering algorithm for mining patterns from event logs. In Proceedings of the 3rd IEEE Workshop on IP Operations and Management, Kansas City, MO, USA, 1–3 October 2003; IEEE Press: Piscataway, NJ, USA, 2003; pp. 119–126.
19. Nagappan, M.; Vouk, M.A. Abstracting log lines to log event types for mining software system logs. In Proceedings of the 7th IEEE Conference on Mining Software Repositories (MSR 2010), Cape Town, South Africa, 2–3 May 2010; IEEE Press: Piscataway, NJ, USA, 2010; pp. 114–117.
20. Vaarandi, R.; Pihelgas, M. LogCluster—A data clustering and pattern mining algorithm for event logs. In Proceedings of the 11th International Conference on Network and Service Management (CNSW), Barcelona, Spain, 9–13 November 2015; IEEE Press: Piscataway, NJ, USA, 2015; pp. 1–7.

21. Dai, H.; Li, H.; Shang, W.; Chen, T.H.; Chen, C.S. Logram: Efficient log parsing using n-Gram dictionaries. *IEEE Trans. Softw. Eng.* **2020**, *47*, 2803–2822. [CrossRef]
22. Tang, L.; Li, T.; Perng, C.S. LogSig: Generating system events from raw textual logs. In Proceedings of the 20th ACM International Conference on Information and Knowledge Management, Glasgow, UK, 24–28 October 2011; ACM Press: New York, NY, USA, 2011; pp. 785–794.
23. Hamooni, H.; Debnath, B.; Xu, J.W.; Zhang, H.; Jiang, G.; Mueen, A. LogMine: Fast pattern recognition for log analytics. In Proceedings of the 25th ACM International Conference on Information and Knowledge Management, Indianapolis, IN, USA, 24–28 October 2016; ACM Press: New York, NY, USA, 2016; pp. 1573–1582.
24. Mizutani, M. Incremental mining of system log format. In Proceedings of the 2013 IEEE International Conference on Services Computing, Santa Clara, CA, USA, 28 June–3 July 2013; IEEE Press: Piscataway, NJ, USA, 2013; pp. 595–602.
25. Liu, Y.D.; Zhang, X.; He, S.L.; Zhang, H.; Li, L.; Kang, Y.; Xu, Y.; Ma, M.; Lin, Q.; Dang, Y.; et al. Uniparser: A unified log parser for heterogeneous log data. In Proceedings of the ACM Web Conference 2022, Lyon, France, 25–29 April 2022; ACM Press: New York, NY, USA, 2022; pp. 1893–1901.
26. Wang, X.H.; Zhang, X.; Li, L.Q.; He, S.; Zhang, H.; Liu, Y.; Zheng, L.; Kang, Y.; Lin, Q.; Dang, Y.; et al. SPINE: A scalable log parser with feedback guidance. In Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE), New York, NY, USA, 14–16 November 2022; ACM Press: New York, NY, USA, 2022; pp. 1198–1208.
27. Meng, W.B.; Liu, Y.; Federico, Z.; Zhang, S.; Chen, Y.; Zhang, Y.; Zhu, Y.; Wang, E.; Zhang, R.; Tao, S.; et al. Logparse: Making log parsing adaptive through word classification. In Proceedings of the 29th International Conference on Computer Communications and Networks (ICCCN), Honolulu, HI, USA, 3–6 August 2020; IEEE Press: New York, NY, USA, 2020; pp. 1–9.
28. Tao, S.M.; Meng, W.B.; Cheng, Y.M.; Zhu, Y.; Liu, Y.; Du, C.; Han, T.; Zhao, Y.; Wang, X.; Yang, H.; et al. Logstamp: Automatic online log parsing based on sequence labelling. *ACM Sigmetrics Perform. Eval. Rev.* **2022**, *49*, 93–98. [CrossRef]
29. Manning, C.D.; Raghavan, P.; Schutze, H. *Introduction to Information Retrieval*; Cambridge University Press: New York, NY, USA, 2008.
30. Salton, G.; Buckley, C. Term-weighting approaches in automatic text retrieval. *Inf. Process. Manag.* **1998**, *24*, 513–523. [CrossRef]
31. Tang, L.; Li, T. Logtree: A framework for generating system events from raw textual logs. In Proceedings of the IEEE International Conference on Data Mining, Sydney, Australia, 13–17 December 2010; pp. 491–500.
32. Zhu, J.; He, S.; Liu, J.; He, P.; Xie, Q.; Zheng, Z.; Lyu, M.R. Tools and benchmarks for automated log parsing. In Proceedings of the International Conference on Software Engineering: Software Engineering in Practice, Las Vegas, NV, USA, 29 July–1 August 2019; pp. 121–130.
33. Hochreiter, S.; Schmidhuber, J. Long short-term memory. *Neural Comput.* **1997**, *9*, 1735–1780. [CrossRef] [PubMed]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.