The Journal of Systems & Software 200 (2023) 111646

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Self-supervised log parsing using semantic contribution difference ☆

Siyu Yu [a], Ningjiang Chen [a,b,c,*], Yifan Wu [d], Wensheng Dou [e]

[a] School of Computer and Electronic Information, Guangxi University, Nanning, 530004, China
[b] Guangxi Key Laboratory of Big Data in Finance and Economics, Nanning, 530003, China
[c] Guangxi Intelligent Digital Services Research Center of Engineering Technology, Nanning, 530004, China
[d] Peking University, Beijing, 100091, China
[e] Institute of Software Chinese Academy of Sciences, Beijing, 100190, China

ARTICLE INFO

Article history:
Received 8 August 2022
Received in revised form 28 October 2022
Accepted 8 February 2023
Available online 10 February 2023

Keywords:
Log parsing
Self-attention
Deep learning
Log semantics

ABSTRACT

Logs can help developers to promptly diagnose software system failures. Log parsers, which parse semi-structured logs into structured log templates, are the first component for automated log analysis. However, almost all existing log parsers have poor generalization ability and only work well for specific systems. In addition, some parsers cannot perform well based on partial data training and cannot support out-of-vocabulary (OOV) words. These limitations can cause erroneous log parsing results. We observe that logs are presented as semi-structured natural language, and we can treat log parsing as a natural language processing task. Thus, we propose Semlog, a novel log parser, requiring no domain knowledge about specific systems. For a log, constant and variable words contribute differently to the semantics of a log. We pretrain a self-attention based model to craft their semantic contribution difference, and then extract log templates based on the pretrained model. We have conducted extensive experiments on 16 benchmark datasets, and the results show that Semlog outperforms the state-of-the-art parsers in terms of average parsing accuracy, reaching 0.987.

1. Introduction

Nowadays, continuously running software systems (e.g., online service systems, cloud systems) are becoming more and more popular. Although dedicated efforts have been devoted to ensuring the reliability and availability of these systems, software systems are still suffering from failures in practice, which lead to user dissatisfaction and enormous economic loss. Once a failure occurs, engineers need to check the system operation logs to diagnose and mitigate the failure in time.

However, software systems often produce a massive volume of logs. For example, the cloud computing system of Alibaba Inc generates about 120 to 200 million lines of logs per hour (Mi et al., 2013). As a result, manually analyzing logs is time consuming and impractical. To tackle this problem, automated log analysis (e.g., log-based anomaly detection Zhang et al., 2019; Jia et al., 2021; Du et al., 2017, failure prediction Fronza et al., 2013, root cause localization Lu et al., 2017) has emerged in recent years. As a critical and first step of automated log analysis, log parsing parses semi-structured logs into structured log templates for further analysis (Le et al., 2022). For example, a log "`Jul 2 04:04:02 combo su(pam\_unix)[24117]: session closed for user cyrus`" consists of log header ("`Jul 2 04:04--:02 combo su(pam\_unix)[24117]:`") and log content ("`session closed for user cyrus`"). Log parsing extracts log content and parses it to log template "`session closed for user *`", that is, keeps constant words and replaces variable words with wildcards "`*`".

In the traditional log parsing approaches, developers manually set regular expressions based on the source code (Xu et al., 2009). But nowadays, the source code of most software systems is not available, and the huge amount of log templates makes it time-consuming to manually set regular expressions (e.g., over 76K templates in Android dataset in Zhu et al., 2019). To overcome the drawback of traditional approaches, tremendous efforts have been dedicated into automated log parsing, which can be divided into heuristics (He et al., 2017), clustering (Fu et al., 2009; Tang et al., 2011; Hamooni et al., 2016; Shima, 2016; Xiao et al., 2020), frequent item mining (Dai et al., 2020; Nagappan et al., 2010; Vaarandi, 2003), and neural network (Nedelkoski et al., 2020). For example, Drain He et al. (2017) uses the prefix tree to group the logs that may be generated by the same log template together. Spell (Du et al., 2016) uses the longest common substring algorithm to extract log templates. Nulog (Nedelkoski et al., 2020) uses Transformer encoder (Vaswani et al., 2017) to classify words one by one.

☆ Editor: Aldeida Aleti.

* Corresponding author at: School of Computer and Electronic Information, Guangxi University, Nanning, 530004, China.
E-mail addresses: gaiusyu6@gmail.com (S. Yu), chnj@gxu.edu.cn (N. Chen), yifanwu@pku.edu.cn (Y. Wu), wensheng@iscas.ac.cn (W. Dou).

https://doi.org/10.1016/j.jss.2023.111646
0164-1212/

However, existing parsers are still unsatisfactory in accuracy, which causes adverse effects on downstream tasks (Le et al., 2021). We summarize three major limitations of existing approaches: (1) Most parsers have poor generalization ability, that is, they only perform well with the appropriate hyperparameters and a lot of tokenization filters preset by domain knowledge, e.g., Drain needs to preset the depth of the prefix tree for different datasets. (2) Some log parsers use defective assumptions and make them perform well only on specific systems, e.g., existing open source parsers (Zhu et al., 2019; He et al., 2016) have an average accuracy of only 0.507 on Proxifier dataset in loghub (He et al., 2020). (3) Some parsers cannot perform well based on partial data training and require a vocabulary containing the entire corpus, so they cannot handle out-of-vocabulary (OOV) words.

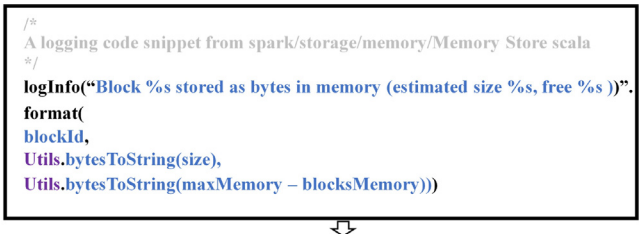
To solve the above problems, in this paper, we treat log parsing as a natural language processing task (i.e., language template extraction) to improve the generalization ability of the parser, because the logs generated by different systems can be regarded as a kind of semi-structured natural language. Further, we observe that the constant words in a log have determined the main semantics of the generated logs, while the variable words are supplementary to the main semantics of the log (e.g., file path, count, user name). This leads to the difference in the contribution of variable words and constant words to the semantics of a log. Based on the key observation, we propose Semlog to extract log templates using this semantic contribution difference between constant and variable words in a log. Semlog first preprocesses the logs into split sub tokens using WordPiece (Wu et al., 2016) to handle OOV words and pretrains a self-attention based model to widen the semantic contribution difference between constant and variable words. Then the pretrained model outputs the semantic contribution score, which is defined to quantify the semantic contribution of words. Finally, based on the assumption that the higher the semantic contribution score of a word, the more likely it is to be a constant word, Semlog extracts log templates by grouping logs with the same words that are most likely constant words and extracting common patterns.

We have evaluated Semlog on 16 benchmark datasets (Zhu et al., 2019; He et al., 2016) and compared it with 13 existing parsers, 12 of which are open source and reproduced by us (Zhu et al., 2019; He et al., 2016), and the comparison data with UniParser (Liu et al., 2022) is the same as the data claimed in their paper. Experimental results show that Semlog can achieve an average parsing accuracy of 0.987 and outperforms the state-of-the-art parsers in terms of average parsing accuracy, and the parsing accuracy standard deviation of Semlog reached 0.022, which is the best among the state-of-the-art parsers. More than that, Semlog trained on 5%, 20% and 40% data can achieve almost the same good results.

To summarize, the contributions of this paper are as follows:

- We propose Semlog to parse logs using the semantic contribution differences among words. To the best of our knowledge, Semlog is the first parser that does not use tokenization filters and hyperparameter tuning. Furthermore, it is also the first approach to use semantic features for log parsing.
- We evaluate Semlog on 16 benchmark datasets. The results show that Semlog performs better than all existing parsers. Its robustness and ability to handle OOV words are also verified through the experiments. We publish our code for better reproducibility.[1]

[1] https://github.com/gaiusyu/Semlog



TIMESTAMP	17/06/09 20:18:48
LEVEL	INFO
CLASS	Storage.MemoryStore
TEMPLATE	Block* stored as bytes in memory (estimated size * B, free * KB)
PARAMETERS	["broadcast_0_piece0", "93.0","317.1"]

Fig. 1. An illustrative example of log parsing.

The rest of the paper is organized as follows. Section 2 introduces the background and main idea. Section 3 describes the details of Semlog. Section 4 presents our experimental results. Finally, we discuss the future work in Section 5 and Section 6 concludes the paper.

2. Background and main idea

2.1. Log parsing

As shown in Fig. 1, logs are generated by logging statements (e.g., `logInfo()`, `print()`) in the source code, and consist of log header (e.g., time "`17/06/09 20:10:48`", PID, log level "`INFO`") and log content (e.g., "`Block broadcast_0_piece0 stored as bytes in memory (estimated size 93.0B, free 317.1 KB)`"). Log content is composed of constant words (e.g., "`Block`") and variable words (e.g., "`93.0, 317.1`"). Log parsing parses semi-structured logs into structured logs by keeping predicted constant words and replacing predicted variable words with wildcards: "`*`", and the parsing result is called log template, like log template: "`Block * stored as bytes in memory (estimated size *B, free * KB)`".

2.2. Related work

Automated log parsing has gradually replaced manual log reading and the manual setting of regular expressions through source code (Xu et al., 2009). The main approaches of automated log parsing include frequent item mining (Dai et al., 2020; Nagappan et al., 2010; Vaarandi, 2003), clustering (Fu et al., 2009; Tang et al., 2011; Hamooni et al., 2016; Shima, 2016; Xiao et al., 2020), heuristics (He et al., 2017), and neural network (Liu et al., 2022; Nedelkoski et al., 2020).

Frequent Item Mining. The general idea of such approaches considers that items (e.g., tokens, n-gram) with more frequent occurrences are more likely to be constant patterns. SLCT (Vaarandi, 2003) is the first paper of automated log parsing, and it is also the first log parsing paper combined with frequent item mining. A similar improved algorithm is LFA (Nagappan et al., 2010). The latest frequent item mining is Logram (Dai et al., 2020) that combines n-gram dictionaries of natural language processing. Logram generates n-gram dictionaries for all logs, finds high frequency n-gram combinations, then determines log variables and constants.

Table 1
Hyperparameters that need to be tune in different parsers.

Approach	Drain	LKE	Spell	...
Hyperparameter	"Tree depth"	"split threshold"	"tau"	...

Table 2
Parsing accuracy of Drain with different tree depth on OpenStack and Proxifier.

"Tree depth"	2	3	4	5	...
OpenStack	0.239	0.239	0.310	**0.733**	...
Proxifier	0.945	0.945	**0.948**	0.935	...

Clustering. The clustering parser believes that logs belonging to the same log template can be clustered together by a certain feature. LKE (Fu et al., 2009) adopts a hierarchical clustering algorithm with a custom weighted edit distance metric. Instead of clustering logs directly, LogSig (Tang et al., 2011) converts each log into a set of word pairs and clusters the logs based on the corresponding word pairs. LogMine (Hamooni et al., 2016) adopts the agglomerative clustering algorithm. LenMa (Shima, 2016) focuses on the word length feature and converts log into a vector of the number of word letters, for example, "`token has expired on`" will be converted to a vector `[5, 3, 6, 2]`. LPV (Xiao et al., 2020) uses skip-grams to convert logs into vectors, which are then grouped according to the similarity of vectors.

Heuristics. Drain He et al. (2017) is the state-of-the-art parser in public benchmark datasets, which employs a fixed-depth tree structure to assist in dividing logs into different groups. Spell (Du et al., 2016) uses the longest common sub-sequence algorithm to extract log templates.

Neural Network. Nulog (Nedelkoski et al., 2020) is a novel kind of parser using neural network which requires less domain knowledge than traditional techniques. Transformer encoder is used by Nulog to categorize masked words one by one. UniParser (Liu et al., 2022) utilizes a Token Encoder module and a Context Encoder module (Bi-LSTM based) to learn the patterns from the log token and its neighboring context.

Others. FT-tree (Frequency template tree) (Zhang et al., 2017) determines the final template tree by cutting branches. MoLFI (Messaoudi et al., 2018) transforms log parsing into a multi-objective optimization problem.

2.3. Limitation of existing work

(1) Poor generalization ability: Most of the existing parsers need to tune some hyperparameters based on domain knowledge across different systems, and few parsers work well on all datasets without any domain knowledge. In Table 1, we list some hyperparameters that require tuning across different systems. As shown in Table 2, Drain needs to manually set the fixed depth of trees on different log datasets, and Drain is very sensitive to the values of these parameters. A log parser with good generalization ability should rely on as little domain knowledge as possible to achieve stable accuracy across various log datasets. When the parser is applied to a new system, tuning parameters to make the parser work well is not easy work because it is difficult to know whether the parser can continue to work well when dealing with the upcoming unseen logs.

(2) Defective assumptions: Some log parsers based on defective assumptions make them perform well only on special benchmark datasets. For example, in the Proxifier dataset, the parsers reproduced in Zhu et al. (2019) can only achieve an average accuracy of 0.507. Besides, the accuracy distribution of most parsers is quite large, e.g., LFA only has an accuracy of 0.026 in the Proxifier dataset, while it has an accuracy of 0.994 in the Spark dataset.

Table 3
Log samples with the same length vector in OpenStack.

0	`[instance: 70c1714bc11b-4c88-b300-239afe1f5ff8]-VM Started (Lifecycle Event)`
1	`[instance: 70c1714bc11b-4c88-b300-239afe1f5ff8]-VM Stopped (Lifecycle Event)`

Table 4
Log samples starting with the variable in Jenkins.

ID	Logs
0	`multi-web #45 main build action completed: SUCCES`
1	`multi-web #46 main build action completed: FAILURE`
2	`security #46 main build action completed: FAILURE`
3	`multi-back #18 main build action completed: SUCCESS`
4	`multi-back #19 main build action completed: SUCCESS`

We give some examples of defective assumptions: (1). LenMa believes that logs with similar word lengths should belong to the same log template. LenMa will convert all logs in Table 3 into `[9,37,2,5,7,10,5]`. Due to the vectors generated by these logs being identical, these logs will eventually be marked as the same log template: "`[instance: <*>] VM <*> (Lifecycle Event)`". However, they should belong to "`[instance: <*>] VM Started (Lifecycle Event)`" and '`[instance: <*>] VM Stopped (Lifecycle Event)`", respectively. (2). Nulog established a vocabulary including every unique word. Too many different logs in the system may lead to a vocabulary explosion. For example, if we use common delimiters (e.g., "`, = - :`"), HDFS will generate hundreds of millions of unique words. (3). Drain believes that logs generated by the same logging statement have the same first k words. As a consequence, Drain cannot apply to logs starting with variables. As shown in Table 4 the first word in Jenkins logs is the project name (i.e., variable word). The log template of these logs is "`* * main build action completed: *`", but Drain will parse them into 3 groups in the first round, which is completely wrong.

Therefore, we can get that if we want to accurately group logs belonging to the same log template, the only way is to find out the constant words in each log to group by, because logs generated by the same log template have the same constant words, not the first k position words.

(3) Partial data training and OOV words: In an industrial environment, we cannot access all logs of every software system, and new systems come online very frequently. In this case, the log parsers based on a neural network need to train a good model using partial datasets, and deal with OOV words, also called unseen words. For example, Nulog requires the words in the coming log are in the established vocabulary (i.e., OOV is not supported), and Uniparser is a supervised approach which requires a lot of labor to label so that this approach cannot be easily applied to a new system.

2.4. Main idea

The effectiveness of downstream tasks will be impacted by parsing errors, e.g., leading to a useless diagnosis and wasting time and manpower to deal with the non-existent failure. Therefore, it is urgent to design a more accurate and robust log parser to solve the above three limitations and reduce parsing errors. Logs are a semi-structured natural language, **can we transform log parsing into natural language processing tasks and extract common language templates?** For example, "`My name is **`" is a language template, the things in the placeholders vary from person to person, like "`joey`", "`jimmy`", "`bob`" etc. We observed that the main semantics of these sentences are concentrated in language templates, that is, about introducing one's own name.

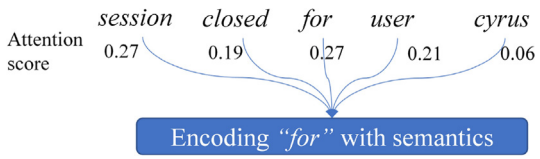


Fig. 2. Example of context feature integration in logs.

The specific name generated by the language template is like the information recorded in the log when the program is running.

New idea. For logs, the logging behavior of the developer inspires us that the constant words in the log have determined the main semantics of the generated logs, while the variable words are supplementary to the main semantics of the log (e.g., file path, count, user name). Based on this observation, we found that constant words contribute more features when encoding a log with semantics. As an example in log content "`session closed for user cyrus`", the variable word "`cryus`" indicates a symbol customized by users and just gives this log template a more detailed supplement, so that it does not affect the semantics (i.e., session closed for a user) of the entire log. In the same way, no matter what any file path is at the end of the log template "`File does not exist: <*>`", it will not change the intention of the developer to record that a certain file does not exist. As a result, we design a new log parser, which extracts log templates using the semantic contribution difference between constant and variable words.

Semantic contribution. In language representation, context features integration has achieved outstanding results in various semantic tasks such as reading comprehension, semantic similarity matching, and natural language inferencing. The most typical mechanism for considering context features is self-attention. Mature architectures based on self-attention have developed a lot, such as BERT (Devlin et al., 2018), Transformer, etc. The attention score in self-attention (the weight of context integrated information when encoding a word generated by Scaled Dot-Product Attention of Selfattention Vaswani et al., 2017) can use to quantify how many features other words can provide when encoding a word. As an example in Fig. 2, since constant words provide the main semantics to the log, when encoding word "`for`" with semantics in the log "`session closed for user cryus`", the attention score of context feature integration of variable word "`cryus`" is the lowest.

Since context features are the main part of semantics, we argue that a word contributes more to the semantics of the entire sentence when it contributes a larger weight (i.e., higher attention score) of context features to all other words. As a result, the semantic contribution of each word can be represented by context feature integration and generated from the attention score.

To the best of our knowledge, no existing approach considers the semantic contribution difference provided by variable words and constant words within a log (He et al., 2021). And if the multi-source log can be unified as a language system, the parser using the semantic contribution of words can be generalized to various log systems. Therefore, in order to solve the limitations of existing parsers, we take advantage of semantic contribution differences among words in a log to design a novel parser, called Semlog.

3. Approach

3.1. Overview of semlog

The framework of Semlog is shown in Fig. 3, which consists of two phases, i.e., offline training phase and online parsing phase. In the offline training phase, we first preprocess the logs into split sub tokens. Then, we pretrain a self-attention based model and the training method adopts MLM (Masked language Model) task (Devlin et al., 2018), which let the model predict the masked words. In the online parsing phase, the real-time logs are tokenized as in the offline training phase without masking and fed into the pretrained model to get attention score. Then, the semantic contribution score is calculated based on the attention score. Finally, we use a template extraction algorithm to extract the log templates of each log based on the semantic contribution score. In summary, there are four components in Semlog, i.e., **Preprocessing**, **Model Pretraining**, **Semantic Contribution Score Calculation** and **Template Extraction**. We will delve into the technical details of all components in the following subsections.

3.2. Preprocessing

We first use delimiters to split the log into separate words and then split the word into sub tokens using WordPiece. Since most existing log parsers directly use some special symbols (e.g., commas, semicolons, dashes) as delimiters without further processing, vocabulary explosion may be easy to occur. In addition to different combinations of numbers, various abbreviations based on the different habits of each developer can also generate new words. For example, the benchmark HDFS dataset is not complicated, with only 30 log templates, but if special characters are used as delimiters, HDFS will generate hundreds of millions of unique words. However, if removing all these digital tokens (e.g., "`blk900189521102825241`"), HDFS only has 62 unique words left. This is exactly why tokenization filters set by domain knowledge are widely used in existing parsers.

To gain a better generalization ability, Semlog uses WordPiece (Wu et al., 2016) for tokenization, which can use about 30,000 sub tokens to represent almost all unique words, the vocabulary of Semlog follows WordPiece's. As an example in Fig. 4, we first use delimiter to obtain separate words, where "`JK2_init`" is not a common word and is split into four sub tokens (i.e., "`J`", "`##K`", "`##2`", "`_in`", "`##it`").

3.3. Model pretraining

Inspired by Bert (Devlin et al., 2018), a mature pretrained model based on self-attention, we designed a self-attention based model to extract semantic contribution and widen semantic contribution differences between constants and variables. Bert has 6 encoders and 12 attention heads. Multilayer self-attention with so many parameters and multi-head attention is used to ensure that Bert works properly when facing various downstream tasks or facing a huge corpus. However, the model in Semlog has only one downstream task (i.e., log parsing), and we only need to widen the semantic contribution difference between constants and variables. Therefore, compared with the structure of Bert, the model in Semlog is a lighter model for a smaller corpus (i.e., log corpus) without any fine-tuning. The detailed structure of the model in Semlog is shown in Fig. 5, which consists of four parts, i.e., **Embedding**, **Attention**, **Classifier**, and **MLM loss**.

(1) Embedding: Taking a split log as input, token embedding encodes each sub token into a vector token Embedding, and token embedding is a linear transform layer of dimension (`vocabu_size` * `d_model`), where `d_model` is the internal network dimension of model and `vocabu_size` is the number of words in vocabulary. Since the position information is important in the log, in order to make use of the order of the log, we add position Embedding to tokenEmbedding. The position Embedding is obtained by pos embedding layer, a linear transform layer of dimension (`max_len` * `d_model`), where `max_len` is the preset

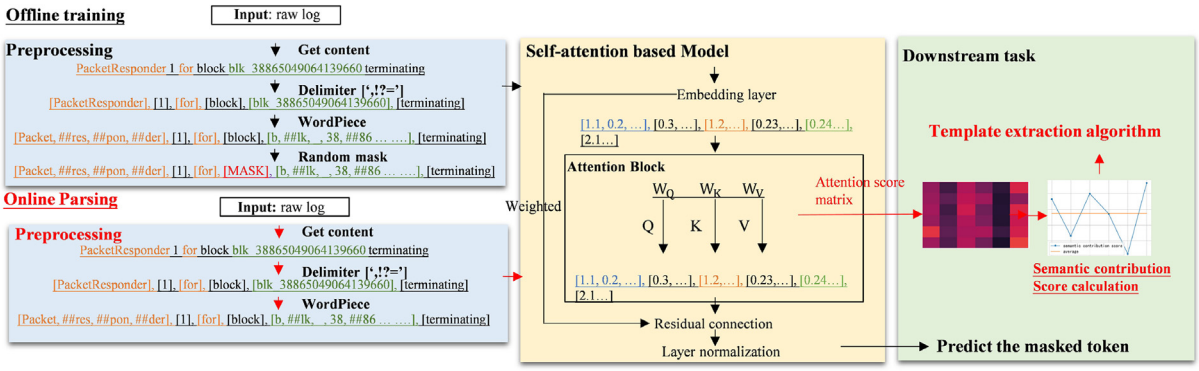


Fig. 3. Framework of Semlog.

Jk2_init	found	child	2330	in	score	board	slot	0
J ##k ##2 _ in ##it	found	child	233 #0	in	score	##board	slot	0

Fig. 4. Example of WordPiece dividing words into sub tokens.

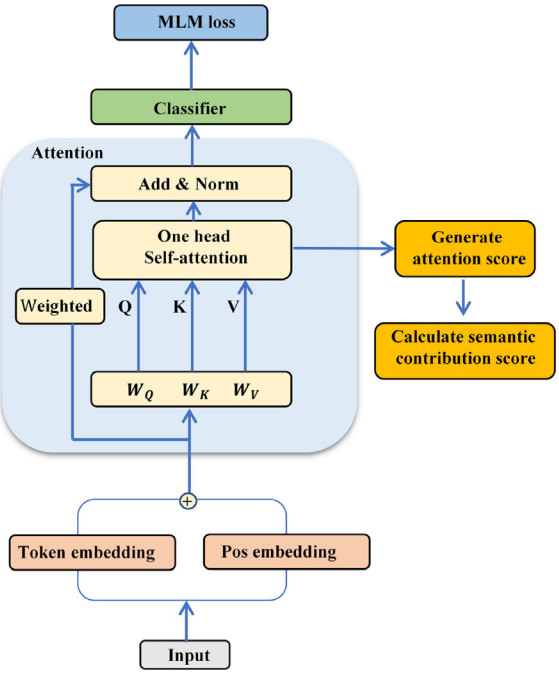


Fig. 5. The self-attention based model in Semlog.

maximum length of a log. In summary, the embedding of a sub token is calculated as:

$$Embedding = tokenEmebedding + posistionEmbedding. \longrightarrow \quad (1)$$

(2) Attention: As mentioned in Section 2.4 the semantic contribution is generated from context feature integration in language representation. To widen the difference between variables and constants, we should make the difference in feature integration of other words more apparent when encoding a word considering the context feature. Thus, compared with the structure of Bert and the model in Semlog, the following three changes are made: (1) Abandoning the multi-head attention mechanism in Bert, using only one attention head to widen differences. (2) The model in Semlog only has one encoder layer (self-attention), because it is only for log parsing and does not need to face huge corpus. The experimental results also verified that one encoder layer is effective for log parsing. (3) Different residual connection settings. The residual connection layer (Add & norm) is designed as:

$$y = Layernorm(weight * x + attention(x)), \quad (2)$$

where x is the original input embedding, and $attention(x)$ is the output of x as input of self-attention. $weight$ is used to enhance the influence of context features in pretraining. $weight$ is uniformly set to 0.01 in our experiments.

(3) Training method and classifier: We employ a self-supervised training method widely used in natural language processing, called MLM, to pretrain. Before pretraining the model, 20% of the tokens in each log will be randomly replaced. There are three replacement methods: these tokens have an 80% chance of being replaced with "`[MASK]`", and a 10% chance of being replaced with any other token, with a 10% chance of being intact. After that, the model is trained to predict and restore the masked or replaced part.

Due to the training object predicting the masked words, we use a classifier after residual connection to get the prediction result:

$$P = \delta(W * y + b), \quad (3)$$

where y is obtained by Eq. (2), both W and b are trainable parameters. δ is the activation function, which uses the same Gelu (Hendrycks and Gimpel, 2016) as Bert. The prediction result P is a probability distribution over the entire vocabulary.

(4) Loss function: When calculating the loss, only the randomly covered or replaced tokens are calculated, and the output of the rest is discarded. Specifically, we adopt the cross entropy loss as the objective function, which is defined as:

$$Loss_{mlm} = -\frac{1}{N}\sum_{j=1}^{N}\sum_{i=1}^{M} P^{j}_{masked_i} \log \hat{P}^{j}_{masked_i}, \quad (4)$$

where N is the total number of input logs in one batch, M is the total number of randomly masked tokens in the `jth` log, $P^{j}_{masked_i}$ is the real token for the `ith` masked token in `jth` log, and $\hat{P}^{j}_{masked_i}$ is the prediction result.

3.4. Semantic contribution score calculation

The semantic contribution score is defined based on the attention score in self-attention. Self-attention is widely used in natural language processing because it can consider the whole context compared to RNN and its variants. When encoding a word, the attention score generated from self-attention quantifies how many features of other words can be integrated. The self-attention mechanism is shown in Fig. 6. Taking a log containing

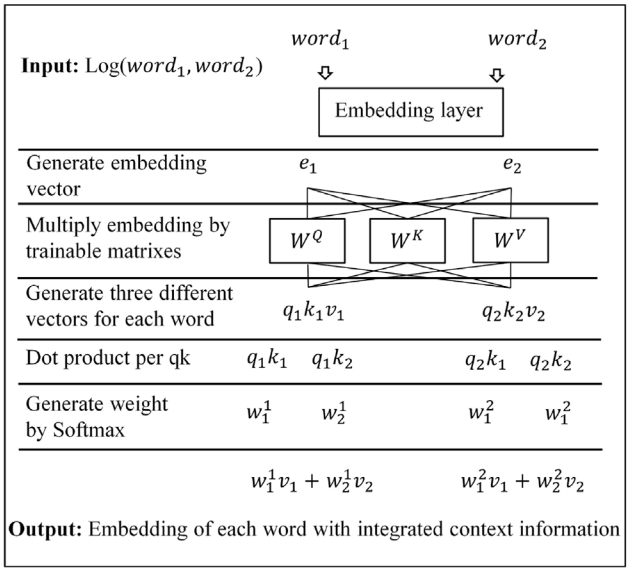


Fig. 6. Example of self-attention mechanism.

two words W_1, W_2 as input, each word will generate query vector q_1, q_1, key vector k_1, k_2 and value vector v_1, v_2. These vectors are calculated as:

$$\begin{aligned} q_i &= x_i \times W^Q \\ k_i &= x_i \times W^K \\ v_i &= x_i \times W^V, \end{aligned} \tag{5}$$

where W^Q, W^K and W^V are three trainable matrices. When encoding the first word W_1, we compute the dot products of the q_1 with all keys (i.e., k_1, k_2), divide each by $\sqrt{d_k}$, and apply a softmax function to obtain the weights on the values (i.e., v_1, v_2). The final output vector is the weighted sum of v_1 and v_2.

In practice, we compute the attention function on a set of queries simultaneously, queries will be packed together into a matrix Q, the keys and values will be packed together into matrices K and V, respectively, then we compute the matrix of output of self-attention as:

$$Attention(Q, K, V) = Softmax(\frac{Q \cdot K^T}{\sqrt{d_k}})V, \tag{6}$$

where $Softmax(\frac{Q \cdot K^T}{\sqrt{d_k}})$ represents the attention score matrix.

However, the attention score can only represent the feature contribution of words to each other. We believe that if a word can provide relatively more features when encoding all other words in a log, then this word will have a higher semantic contribution to the whole log. Thus, we define the semantic contribution score of a word as the sum of attention scores for all words. Specifically, the semantic contribution score of word W_i is calculated as follows:

$$semanticcontributionscore(W_i) = \sum_{m=0}^{len(WordPiece(l))} \sum_{n=\sum_{i=0}^{i-1} len(WordPiece(W_i))}^{\sum_{i=0}^{i} len(WordPiece(W_i))} score[m][n], \tag{7}$$

where l is the log that word W_i belongs to, W_i is the word with position index i in l, $WordPiece(l)$ splits l into sub tokens:

$$subtokens_q | q \in [0, len(WordPiece(l))] \cap Z, \tag{8}$$

and $len(WordPiece(l))$ calculates the number of sub tokens of l. $score[m][n]$ represents the attention score generated by $subtoken_m$ when $subtoken_n$ is encoded (i.e., dot product of q_m with k_n)

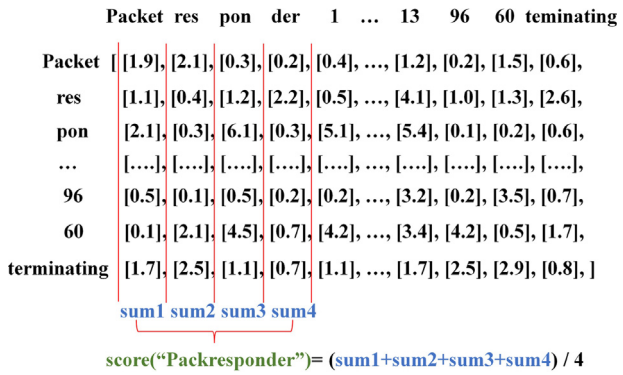


Fig. 7. Computing the Semantic Contribution Score via the Attention Matrix.

Further, Fig. 7 more clearly shows how to calculate the semantic contribution score of word "`PacketResponder`" in log "`PacketResponder 1 for block blk_38865049064139660 terminating`" through the attention matrix. As a result, WordPiece can help Semlog use a fixed vocabulary to calculate the semantic contribution score of almost all words and handle OOV words.

In summary, the semantic contribution score can quantify the semantic contribution of each word in a log, and constants and variables are significantly different in the semantic contribution score, which can be used to extract log templates.

3.5. Template extraction

After getting the semantic contribution score of each word, based on the assumption that the higher the semantic contribution score, the more likely the word is a constant word, Semlog groups logs by words with the top `k` semantic contribution score. Specifically, if a log contains three constant words, we extract the words with the top three semantic contribution scores, then the template extraction is completed. However, the number of constant words in each type of log templates is different and cannot be known in advance. In addition, based on our observation of the experimental results, although the semantic contribution score of constant words is higher than that of variable words, the order of the semantic contribution score of constant words may be uncertain in a log. For example, in the log template "`* open through proxy * https`", the word with the highest score in the logs belonging to this template may be either "`through`" or "`open`", so that we cannot simply group by the words with the top `k` high semantic contribution scores. In order to solve these difficulties, we design a template extraction algorithm called TESC. TESC uses the first `k` high semantic contribution score to replace the first `k` positions in the traditional approach Drain to gradually group the logs.

TESC algorithm is shown in Algorithm 1. The input is logs and corresponding semantic contribution scores of each word, and the output is the log template to which the input belongs. Firstly, the words in the log will be sorted from high to low according to their semantic contribution score, and the word with the highest score will be extracted, then compared with the existing log groups to see how many representatives of these log groups contain this word in the same column. If the number of matched groups is 0, the log will create a new log group with this log as the representative. If there is one and only one log group matches, the log will be assigned to that group and update the log template of that group. If the number of matched groups is greater than 1, the word with the highest score will be deleted from the sequence of words in descending order of semantic contribution score, and

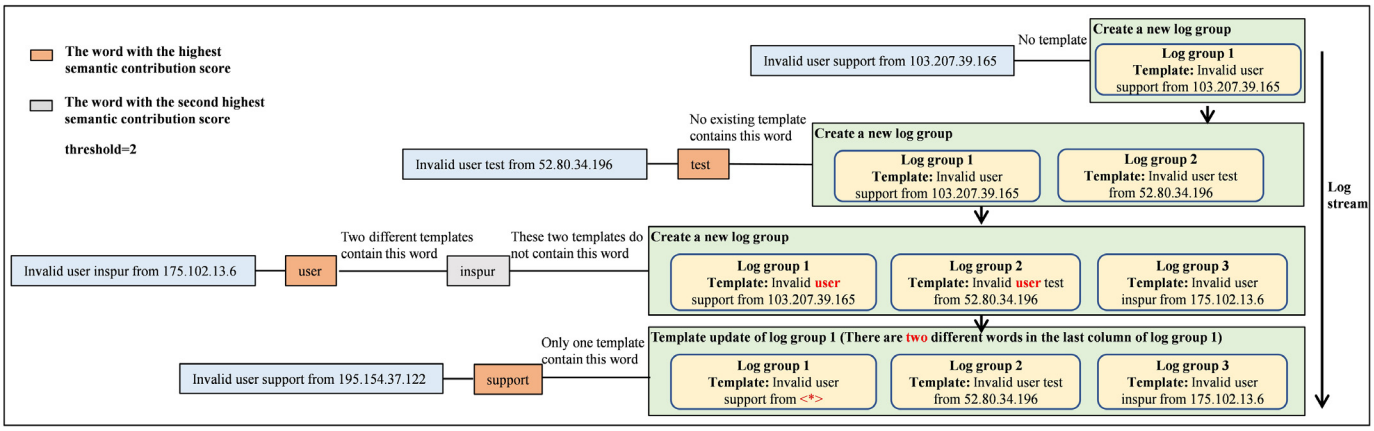


Fig. 8. Log parsing example of TESC algorithm.

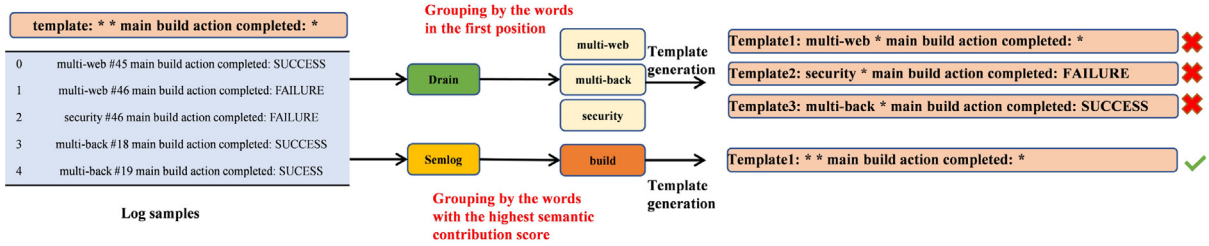


Fig. 9. The first round of grouping when Drain and Semlog parse logs.

Algorithm 1 Template extraction algorithm based on semantic contribution score

Input: log $T = t_1, t_2, ..., t_n$; semantic contribution score corresponding to $TC = c_1, c_2, ..., c_n$;
Output: Template of log T.

```
count ← 0,, i ← C.index(max(C)),, G, G_m ← ∅
T ← T.sortby(C), # t_1, t_2, ..., t_n sorted by their corresponding c_1, c_2, ..., c_n
for group in G do
    if t_1 == group.template[i] then
        count ← count + 1
        G_m ← G_m ∪ group #G_m, the matched groups
    end if
end for
if count > 1 then
    T.pop(0)
    G_m.template ← TESC(T, C, G_m)
end if
if count == 0 then
    G_m ← T
    G ← G.add(G_m) # a new group is added to G
    G_m.template ← T
end if
if count == 1 then
    G_m ← G_m ∪ T
    G_m.template ← templateupdate(G_m, 3)
end if
return G_m.template
```

then TESC will be called recursively, input is the new sequence of words, and the matched groups in first round.

For the updates to log templates representing each group, we use the number of different words in the same column to determine whether the word in this position is a variable, the threshold is uniformly set to 3 in all benchmark datasets. If the word at a certain position is recognized as a variable word, the log template will be replaced by "`*`" at that position. For example, in Proxifier, "`* open through proxy * https`" and "`* open through proxy * scoks5`" are labeled as two different templates in LogPai (Zhu et al., 2019; He et al., 2016), but they are grouped together by TESC. In fact, it is hard to judge whether the last position is a variable or a constant without source code. The setting of the threshold solves this problem well and tolerates some grouping errors. This setup also works well with synonyms and antonyms. Further, a log parsing example of TESC is shown in Fig. 8, where the threshold value is set to 2. The four logs were finally divided into three groups. In log group 1, word `103.207.39.165` is replaced with wildcard `*` because the addition of the fourth log makes the number of different words at this position greater than or equal to the threshold (2).

Since TESC is inspired by Drain, which is the state-of-the art parser, we give an example in Fig. 9 to show that the logs generated by the same template should not be grouped by the words in the previous `k` positions, but by the words with the top `k` high semantic contribution score. In the log samples in Fig. 9, the words in the first position have three different words (i.e., variable words), so Drain will divide the logs belonging to one template into three groups, and generate template from each group, which is completely wrong. For Semlog, pretrained model will output the semantic contribution score of each word, which quantifies the possibility of whether it is a constant word. And then TESC groups the log based on the output of the pretrained model. Since the highest semantic contribution score of each log in the log sample is "`build`", the grouping is completed in the first round, and all logs are correctly divided into one group.

4. Evaluation

In this section, we evaluate our approach by answering the following research questions:

Table 5
Data statistics and configuration.

Dataset	Data Size	Messages	Description	Delimiter	Epoch
HDFS	1.47 GB	11,175,629	Hadoop distributed file system log	,!?=	1
Apache	4.96 MB	56,481	Apache server log	,!?=	10
Zookeeper	9.95 MB	74,380	Zookeeper service log	.,!?	10
Mac	16.09 MB	117,283	Mac OS log	,\[\]	10
HealthApp	22.44 MB	253,395	Health app log	,!?=:	10
Android	183 MB	1,546,686	Android framework log	,?_:	10
OpenStack	60.01 MB	207,820	OpenStack software log	,.	15
BGL	708.76 MB	4,747,963	Blue Gene/L supercomputer log	,!?=	1
Proxifier	2.42 MB	21,329	Proxifier software log	,!?=	10
Linux	2.25 MB	25,567	Linux system log	,!?=	10
HPC	32.00 MB	433,489	High performance cluster log	,=	10
Hadoop	48.61 MB	394,308	Hadoop mapreduce job log	,=	10
Windows	26.09 GB	114,608,388	Windows event log	,!?=	1
Thunderbird	29.60 GB	211,212,192	Thunderbird supercomputer log	,!?=	1
Spark	2.75 GB	33,236,604	Spark job log	,!?=	1
OpenSSH	70.02 MB	655,146	OpenSSH server log	,_	10

- RQ1: Is the parsing accuracy of Semlog better than existing parsers, and can it perform stably across various datasets?
- RQ2: When Semlog is trained using a partial dataset and encounters OOV words, can it maintain accuracy?
- RQ3: How effective is the pretrained model in Semlog for widening semantic contribution difference?

The accuracy and robustness of Semlog are evaluated in 16 benchmark datasets and compared with 12 classic log parsers in RQ1. Then, the ability to handle OOV words of Semlog is evaluated in RQ2. Finally, because the model in Semlog is inspired by Bert, and Bert has achieved attractive success in language representation, we conduct a comparative experiment with the model in Semlog and Bert on the ability to widen semantic contribution difference in RQ3.

4.1. Experimental setting

(1) Dataset: We conduct experiments based on 16 benchmark datasets published in LogPai, which include distributed system logs (HDFS, Zookeeper, Spark, Hadoop, OpenStack), supercomputer logs (HPC, BGL, Thunderbird), operating system logs (Mac, Linux,Windows), mobile system logs (Android, HealthApp), server application logs (Apache, OpenSSH) and standalone software logs (Proxifier). Each log message is labeled which a log template as ground truth. Table 5 summarizes the statistics of the origin datasets, all the delimiters used by each dataset to obtain separate words and how many epochs they trained. In order to make the results easier to reproduce, we follow the guidelines in Zhu et al. (2019), the evaluation results are generated from 2000 sample logs randomly selected from LogPai, which have been labeled with the templates they belong to, so that the results can be easily reproduced by using our opensource code.

(2) Evaluation metric: We use Parsing Accuracy (abbreviated as PA in the following) from the guidelines in Zhu et al. (2019) for evaluation, which measures the ratio of correctly parsed logs over the total number of logs. A log is considered correctly parsed if its log template corresponds to the same group of logs as the ground truth does. For example, if we parse the log sequence [e1, e2, e2] as [e1, e4, e5], we get PA = 1/3 because the second and third messages are not grouped together. PA can evaluate whether a parser can facilitate most downstream tasks (e.g., log sequence anomaly detection) because only logs belonging to the same template are considered to be the same log key, can we get the real log key sequence.

(3) Implementation and Configuration: All experiments are conducted on a GPU server with NVIDIA GeForce RTX 3090 GPU and CUDA 11.3. We implement Semlog based on Python 3.8 and PyTorch 1.10. All hyperparameters are shown in Table 6. We set the learning rate to 0.01 and the batch size to 2048. The epoch and delimiter we used for each dataset is shown in Table 5. We set weight to 0.01, threshold to 3, d_model to 64 and max_len to 512 in the model of Semlog. And in order to reduce the "same length" limit, we use some regular expressions to preprocess some datasets, so that * can represent continuous variables, which can be viewed in detail in the Table 7. And Table 8 shows us that the Proxifier dataset has a lot of brackets for explanation, even if these brackets are not always present (variables do not necessarily appear at positions "`... close, bytes * ...`" and "`... bytes * sent ....`"), such logs are also marked with the same template: "`* close, * bytes * sent, * bytes * receive- -d, lifetime *`". In addition, when we calculate the semantic contribution score, we will remove the semantic contribution score of some special and meaningless symbols, including ['.', '_', '-', ':', '/', '\"', ';', '\'', '[', ']', '(', ')', '', '']. For example, when we calculate the semantic contribution score of "`used_disk=0GB`", after dividing "`used_disk=0GB`" into subtokens "`used`", "`_`", "`disk`", "`=`", "`0`", "`GB`", we do not need to consider "`_`", "`=`", but only need to consider the semantic contribution score of "`used`", "`disk`", "`0`", "`GB`".

Table 6
Semlog hyperparameter setting.

Hyperparameter	Value
learning rate	0.01
batch size	2048
weight	0.01
threshold	3
d_model	64
max_len	512

Table 7
Preset regular expressions.

Dataset	regular expression
Windows	'\(.*?\)'
Android	'\(.*?\)'
Proxifier	'<1 s', '\(.*?\)'
Thunderbird	'name .+\d'
OpenStack	'10 [\d]+ 10 [\d]+', ' json', '8 5', '11 5 .+?us'

4.2. RQ1: Parsing accuracy and robustness evaluation

Parsing accuracy. We reproduced 11 log parsers code provided in LogPai, and obtained the PA of each parser on 16 benchmark datasets. As a comparison, we got the PA of Semlog on 16 benchmark datasets in the same way, Table 9 shows the PA of 13 log parsers in 16 benchmark datasets, which the PA of

Table 8
Proxifier log samples with brackets.

Template	"* close, * bytes * sent, * bytes * received, lifetime *"
log0	proxy.cse.cuhk.edu.hk:5070 close, 1292 bytes (1.26 KB) sent, 1500 bytes (1.46 KB) received, lifetime <1 s
log1	proxy.cse.cuhk.edu.hk:5070 close, 2455 bytes (2.39 KB) sent, 727 bytes received, lifetime 00:43 <1 s

Table 9
Parsing accuracy comparison on 16 benchmark datasets.

Dataset	SLCT	AEL	LKE	LFA	LogSig	SHISHO	LenMa	LogMine	Spell	Drain	MoLFI	UniParser	Semlog
HDFS	0.545	0.998	**1.000**	0.885	0.850	0.998	0.998	0.851	**1.000**	0.998	0.998	**1.000**	0.998
BGL	0.573	0.758	0.128	0.854	0.227	0.711	0.69	0.723	0.787	0.963	0.960	**0.997**	0.986
HPC	0.839	0.903	0.574	0.817	0.354	0.325	0.830	0.784	0.654	0.887	0.824	0.966	**0.996**
Apache	0.731	**1.000**	**1.000**	**1.000**	0.582	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**
HealthApp	0.331	0.568	0.592	0.549	0.235	0.397	0.174	0.684	0.639	0.780	0.440	**1.000**	**1.000**
Mac	0.558	0.764	0.369	0.599	0.478	0.595	0.698	0.872	0.757	0.787	0.636	**0.997**	0.923
Proxifier	0.518	0.518	0.515	0.026	0.967	0.517	0.508	0.517	0.527	0.527	0.013	0.976	**1.000**
Zookeeper	0.726	0.921	0.438	0.839	0.738	0.660	0.841	0.688	0.964	0.967	0.839	**0.995**	0.992
Thunderbird	0.882	0.941	0.813	0.649	0.694	0.576	0.943	0.919	0.844	0.955	0.646	**0.990**	**0.990**
Spark	0.685	0.905	0.634	0.994	0.544	0.906	0.884	0.576	0.905	0.920	0.418	**1.000**	**1.000**
Android	0.882	0.682	0.909	0.616	0.548	0.585	0.880	0.504	0.919	0.911	0.788	**0.973**	0.948
Linux	0.291	0.673	0.519	0.279	0.169	0.701	0.701	0.612	0.605	0.690	0.284	0.878	**0.999**
Hadoop	0.423	0.538	0.670	0.900	0.633	0.867	0.885	0.870	0.778	0.948	0.957	**1.000**	0.993
OpenStack	0.867	0.758	0.787	0.200	0.200	0.722	0.743	0.743	0.764	0.733	0.213	**1.000**	0.967
Windows	0.697	0.690	0.990	0.588	0.689	0.701	0.566	0.993	0.989	0.997	0.406	**1.000**	**1.000**
OpenSSH	0.521	0.538	0.426	0.501	0.373	0.619	0.925	0.431	0.554	0.788	0.500	**1.000**	0.997
Average	0.637	0.754	0.563	0.652	0.482	0.669	0.721	0.694	0.751	0.865	0.605	0.986	**0.987**
STD	0.183	0.163	0.245	0.277	0.234	0.186	0.207	0.169	0.156	0.131	0.297	0.030	**0.022**

Table 10
Similar templates of Mac.

"CCFile::captureLog Received Capture notice id: *, reason = RoamFail:sts:⟨NUM⟩_rsn:⟨NUM⟩"
"CCFile::captureLog Received Capture notice id: *, reason = AssocFail:sts:⟨NUM⟩_rsn:⟨NUM⟩"
"CCFile::captureLog Received Capture notice id: *, reason = DeauthInd:sts:⟨NUM⟩_rsn:⟨NUM⟩"
"CCFile::captureLog Received Capture notice id: *, reason =

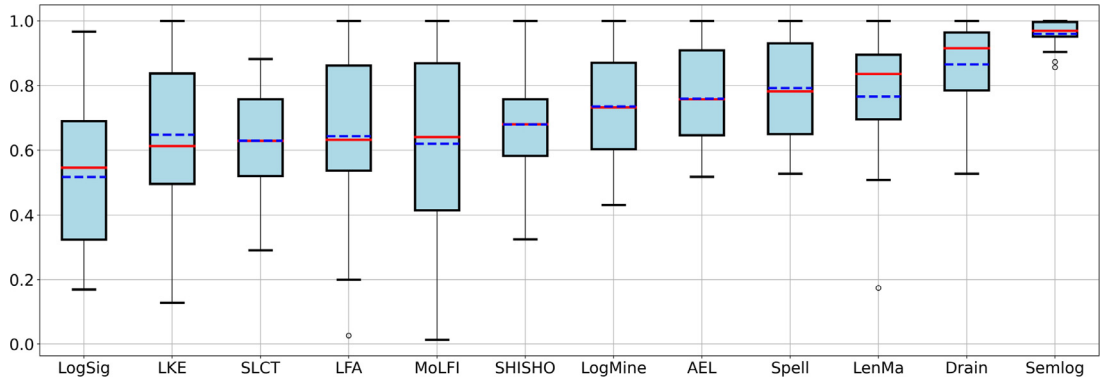


Fig. 10. Parsing accuracy distribution of log parsers across different types of logs.

UniParser is the same claimed in their paper. Semlog can perform better in most datasets, and its average PA reaches 0.987, which is 12.2% higher than the state-of-the-art open source parser. Semlog achieved a PA of 1.000 in 5 datasets, and there are 12 benchmark datasets with a PA of over 0.990. And regarding the lowest performance of Semlog in Mac, with a PA of only 0.923, Mac log templates in Table 10 are labeled by LogPai. Without source code, these templates are difficult to label accurately even by manual labeling, Semlog did not focus semantics on "`RoamFail:sts:<NUM>_rsn:<NUM>`", but on "`Received`" and other. So Semlog will group them together. This kind of log is very deceptive and need further study.

Robustness. In industry, maintaining high accuracy across different datasets is very important for log parsers. Both the poor generalization ability and the defective assumptions of parsers can cause poor robustness. As shown in Table 9, the PA standard deviation (STD) of Semlog is 0.022. And Fig. 10 shows a boxplot that indicates the PA distribution of each log parser across the 16 benchmark log datasets. For each box, the horizontal solid lines from bottom to top correspond to the minimum, 25-percentile, median, 75-percentile and maximum PA values, and the blue dotted line represents the average PA of each log parser. The boxes are arranged from left to right according to the median of PA. From the results, we can see that the PA of many log parsers has a large distribution range. For example, LFA only achieved a Parsing Accuracy of 0.026 in Proxifier, but got a PA of 1.0 in Apache. Logsig may be the best in Proxifier except Semlog, PA of 0.967, but it does not perform well in other datasets, only got PA of 0.354 in HPC, and PA of 0.235 in HealthApp. In summary, Semlog is a robust parser that maintains high accuracy across different datasets, with the highest median PA, highest mean PA and lowest PA STD.

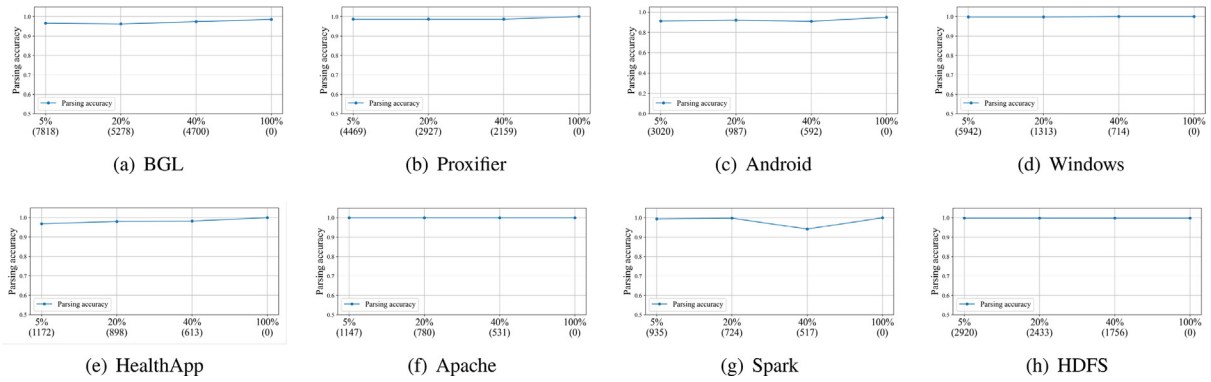


Fig. 11. OOV words impact evaluation results (The number in brackets below the *x*-axis is the number of OOV words encountered in the test set).

Table 11
Test data of model evaluation.

Sample dataset	Messages	Test log template	Template words
HDFS	2000	packetresponder * for block * terminating	4
Apache	2000	[client *] file does not exist: *	5
HealthApp	2000	New date * type * * old *	4

4.3. RQ2: impact of OOV words

In our experiments, OOV indicates that words that have never appeared in the training dataset are encountered during online parsing. To evaluate the performance of Semlog facing with OOV words, we selected 2K logs for at least one system from each type of system (i.e., HDFS, Spark, Android, BGL, Proxifier, Apache, Windows and HealthApp) and sampled 5%, 20%, and 40% of the original datasets to pretrain model. The number of OOV words encountered in the training of each scale and PA are shown in Fig. 11. The results show that the performance of Semlog trained by different ratios of the 8 datasets are stable, and the PA of Semlog trained by different ratios of HDFS and Apache is the constant. After further analysis, we found the templates of HDFS and Apache are relatively simple, and variable words basically are digital tokens, so that Semlog can easily filter out variables without focusing semantics on important words. For the Spark datasets, we can see that there is fluctuation in "40%". The effect depends on how well Semlog learns the most semantically important words. Therefore, as long as the corresponding template patterns (logs belonging to the same log template) are learned well, Semlog using WordPiece can handle OOV words well and maintain the same good accuracy, regardless of any new variables.

4.4. RQ3: model evaluation

We firstly take the log ("`packetresponder 0 for block blk -9001895211102825241 terminating`") as input to compare the attention score generated by Bert and the pretrained model in Semlog. Fig. 12 shows the heatmap of the attention score matrix of the first encoder layer generated by Bert. 0 to 5 is used to replace '`packetresponder`', '0', '`for`', '`block`', '`blk9001895211102825241`', '`terminating`' respectively in the matrix coordinate axis. Bert was originally designed as a pretrained model with various downstream tasks, and layers with large volume of parameters are generally designed for large corpus. While applying on log systems, the heatmaps of the attention score of `2th` to `12th` encoder layers are basically the same (the attention scores of 6 words are about `0.13-0.14` (`1/6`), which is close to the average score, no significant difference), therefore, the semantic contribution score of Bert is calculated based on the sum of the 12 attention score matrices generated by the first encoder layer.

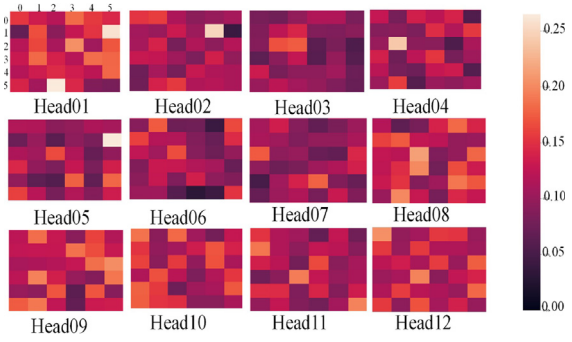
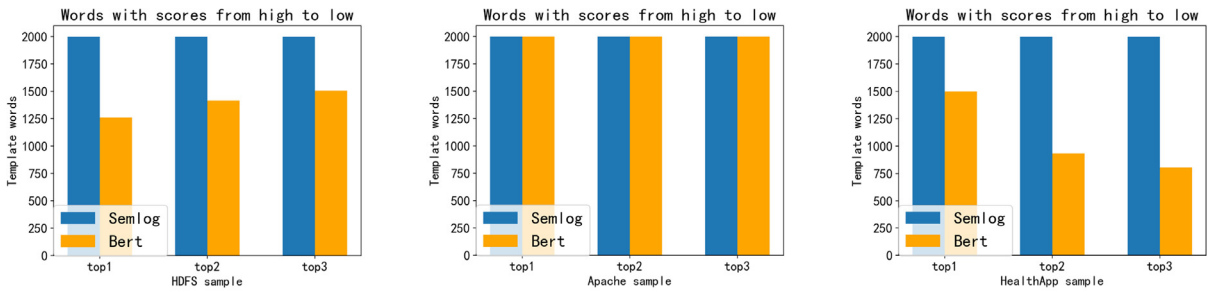


Fig. 12. Heatmaps of the attention score matrix of the first encoder generated by Bert.

For Bert, it is difficult to judge which word is the variable (i.e., words with significantly low attention score) by visually observing the heatmap of the attention score. In contrast to the heatmap of Bert, the heatmap of the attention score generated by the pretrained model in Semlog is shown in Fig. 14. We can see that the attention scores in columns "0" and "blk9001895211102825241" are relatively low, which means that the semantic contribution score of these two words will be lower.

In order to further compare the ability of the two models to distinguish constants from variables, we first manually mark how many constant words each log contains, and extract less than or equal to this number of words in order of semantic contribution scores from high to low. we use three sample datasets as training datasets, each of which is generated by sampling 10,000 logs randomly from benchmark datasets, to pretrain Bert and the model in Semlog. And then, we take 2K logs belonging to the same template of three sample datasets as input to output words with top `k` high semantic contribution scores in Bert and the model in Semlog. Table 11 describes which template the test logs belong to. We checked whether these outputs are constant words to demonstrate which model is more suitable for log parsing. The results are shown in Fig. 13. From the results, we can see that the model in Semlog outputs 100% of the constant words in the top three high semantic contribution scores among three sample



(a) HDFS (b) Apache (c) Health

Fig. 13. The number of template words for the words with top 3 semantic contribution scores.

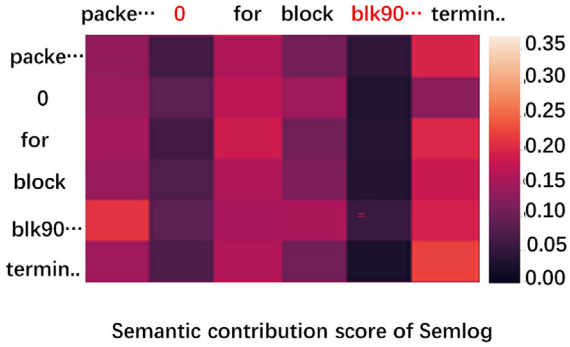


Fig. 14. Heatmaps of the attention score matrix generated by Semlog.

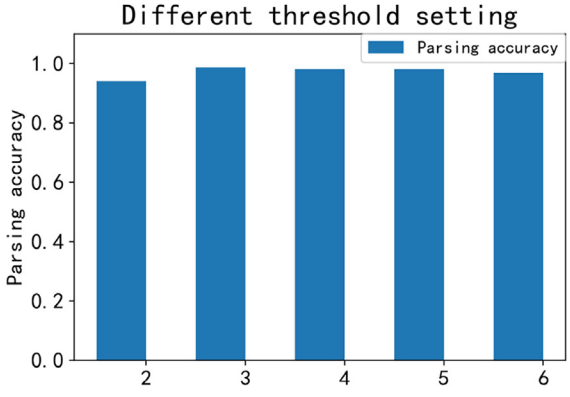


Fig. 15. Sensitivity analysis on "threshold" of Semlog.

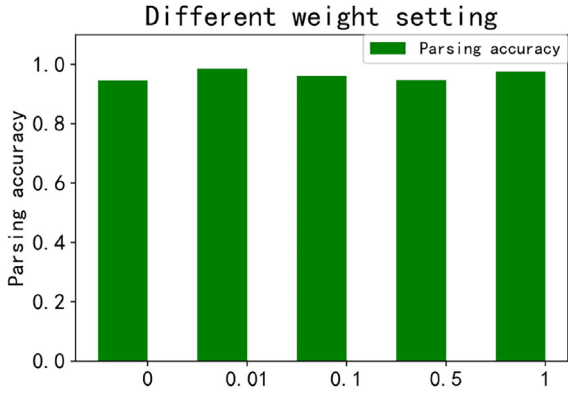


Fig. 16. Sensitivity analysis on "weight" of Semlog.

datasets, but the ratio of constant words extracted by Bert at each round cannot achieve 100% (e.g., The number of constant words output by the Bert in the HDFS: top1: 1262, top 2: 1416, top3: 1508).

As mentioned in Section 2.3, the premise of correctly classifying logs is to use constant words. As a consequence, Bert cannot be well used for log parsing, because its output does not conform to the higher semantic contribution score, the more likely it is to be a constant word.

4.5. Sensitivity analysis on parameters

We further performed a sensitivity analysis on the parameters in Semlog. The parameters in Semlog do not need to tune across different systems. There are two main parameters in Semlog, (1) threshold, in the log group generated by Semlog, the number of different words in each column will be used to compare with threshold to classify themselves (Section 3.5), (2) residual connection weight, which is used to educe the importance of the original vector and enhance the influence of the context, thus widen the semantic difference between constant and variable words (Section 3.3. Fig. 15 shows the parsing accuracy achieved by pre-trained Semlog on BGL datasets with different weight settings. And Fig. 16 shows the average parsing accuracy achieved by pre-trained Semlog on 16 benchmark datasets based on different threshold settings.

From the figures we can see that Semlog is not sensitive to these two parameters. For the threshold, this is because the position of the variable word in the log in the real world is used to record different information when the software is running, so that the variable position will generate many different words whose frequency is usually far beyond the threshold we set. For the weights, the reason why we adjust them to be relatively low is that we want the model to learn the contextual features faster.

In summary, the experimental results show that the pre-trained model in Semlog is an effective model for log parsing, which can effectively widen the semantic contribution difference between variable words and constant words. And, it is verified that Semlog has good accuracy (the average PA reaches 0.987, which is 0.12 higher than state-of-the-art open source parsers and 0.01 higher than UniParser) in 16 public benchmark datasets. In addition, the small distribution of PA shows that Semlog has good robustness, and the Semlog is proved to be almost immune to OOV words and achieves the same good PA using partial training data.

5. Discussion

Our original idea is to directly classify variables and constants by threshold based on semantic contribution score, but how to set a universal threshold got us into trouble. Fig. 17 shows the distribution of the semantic contribution score when parsing a log. We can observe that there are two words with abnormally

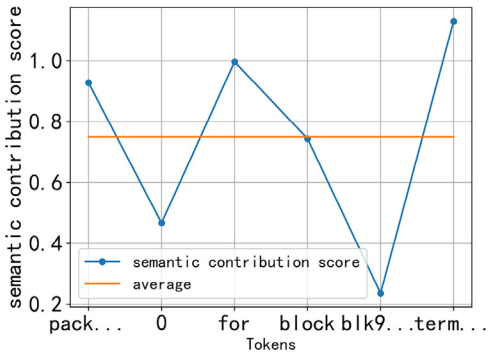


Fig. 17. Semantic contribution score of each word and average score in Semlog.

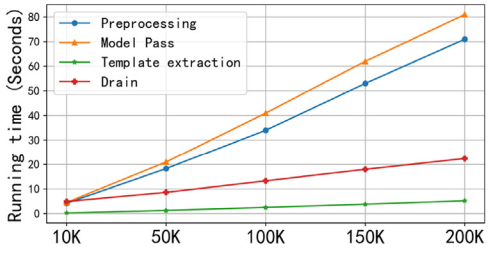


Fig. 18. Running time of Semlog.

low values, which are exactly the variable words in this log. But if we set the average semantic contribution score as the threshold, the score value of "block" is a bit below the average semantic contribution score, so that it will be regarded as a variable word. We also tried to shift the threshold downward by weight*std (std represents the standard deviation of the semantic contribution score of each word), but it is difficult to achieve uniform weight across all datasets. Therefore, in order to make Semlog more robust without tuning the hyperparameters, in this paper, we designed TESC algorithm. Setting thresholds to parse logs can reduce time consumption. And if the threshold is set to:

$$threshold = average - 0.2std, \tag{9}$$

word-level parsing accuracy (i.e., the ratio of each word is correctly classified as the same constant and variable as the ground truth) in Proxifier can exceed 85%. Therefore, we will find a better way to set uniform thresholds across different datasets in the future.

Running time is an important indicator for evaluating parsers. Fig. 18 shows the running time of different modules of Semlog and Drain on different volumes of BGL datasets. The Approach based on deep learning is difficult to surpass the traditional algorithm in running time. In our experiment, we send the data into the model in batches to reduce the running time, and Semlog takes about 70 s to process every 100,000 logs. In the future, we will further optimize the running time of Semlog.

The state-of-the-art parser. To the best of our knowledge, Uniparser achieves the best parsing accuracy on 16 benchmark datasets. However, it is not open sourced, and the technical detail of Uniparser is not described well in their paper so it is hard for us to reproduce their experimental result (e.g., the 96 characters they used to cover the most of tokens formed by their combinations). Compared with Uniparser, Semlog is a self-supervised approach requiring no labels so we suppose that Semlog is more convenient to apply to a new system. The approach based on deep learning seems to refresh its best performance in log parsing. Since log is a semi-structured natural language, we believe that the advanced deep learning model in the field of NLP is promising in the field of log analysis.

Anomaly Detection. Most sequence-based log anomaly detection approaches use a log parser to obtain log templates, encode the log templates to embedding vectors, and then take the vectors within a time window as input to the anomaly detector. The methods for encoding log templates include word2vec, one-hot, etc. However, Semlog can directly encode the original logs instead of parsing them to templates and then encoding templates. Because variable words in the pretrained model of Semlog have significantly lower attention scores than constant words, so features provided by variables will have low weights in the embedding of the entire log, making the log embedding quite close to the log template embedding. As a result, this will mitigate the impact of parsing errors on downstream tasks. This is why the model in Semlog adds the "`[CLS]`" at the beginning of each log. In language representation, one of the ways BERT does embedding is to directly use the vector generated by "`[CLS]`". In the future, we plan to design a log anomaly detection approach based on the pretrained model in Semlog.

6. Conclusion

Log parsing, which parses semi-structured logs into structured logs, is a critical step of automated log analysis. In this paper, we treat log parsing as natural language processing task and propose Semlog to extract log templates using the semantic contribution differences among words in a log. After preprocessing the logs into split sub tokens, Semlog pretrains a self-attention based model and generates the semantic contribution score for each word. Then, a template extraction algorithm is designed to extract log templates based on the semantic contribution score. We have evaluated Semlog on 16 benchmark datasets, and the results show that Semlog outperforms the state-of-the-art open source log parsers by 12.2% on Parsing Accuracy. Besides, the experiments also verified its robustness and ability to handle OOV words.

CRediT authorship contribution statement

Siyu Yu: Investigation, Methodology, Software, Writing – original draft, Validation, Conceptualization. **Ningjiang Chen:** Writing – review & editing, Supervision, Funding acquisition. **Yifan Wu:** Writing – review & editing. **Wensheng Dou:** Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

I have shared my data/code repository link at my manuscript

Acknowledgments

This work is supported by the Natural Science Foundation of China (No. 62162003), the Guangxi Key Laboratory of Big Data in Finance and Economics (No. FEDOP2022A02), the National Key Research and Development Project of China (No. 2018YFB 1404404), and the Nanning Science and Technology project (No. 20221031)

References

Dai, H., et al., 2020. Logram: Efficient log parsing using n-gram dictionaries. IEEE Trans. Softw. Eng.

Devlin, J., et al., 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. ArXiv preprint arXiv:1810.04805.

Du, M., et al., 2016. Spell: Streaming parsing of system event logs. In: 2016 IEEE 16th International Conference on Data Mining. ICDM, IEEE, pp. 859–864.

Du, M., et al., 2017. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 1285–1298.

Fronza, I., et al., 2013. Failure prediction based on log files using random indexing and support vector machines. J. Syst. Softw. 86 (1), 2–11.

Fu, Q., et al., 2009. Execution anomaly detection in distributed systems through unstructured log analysis. In: 2009 Ninth IEEE International Conference on Data Mining. IEEE, pp. 149–158.

Hamooni, H., et al., 2016. Logmine: Fast pattern recognition for log analytics. In: Proceedings of the 25th ACM International on Conference on Information and Knowledge Management. pp. 1573–1582.

He, P., et al., 2016. An evaluation study on log parsing and its use in log mining. In: 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. DSN, IEEE, pp. 654–661.

He, P., et al., 2017. Drain: An online log parsing approach with fixed depth tree. In: 2017 IEEE International Conference on Web Services. ICWS, IEEE, pp. 33–40.

He, S., et al., 2020. Loghub: a large collection of system log datasets towards automated log analytics. ArXiv preprint arXiv:2008.06448.

He, S., et al., 2021. A survey on automated log analysis for reliability engineering. ACM Comput. Surv. 54 (6), 1–37.

Hendrycks, D., Gimpel, K., 2016. Bridging nonlinearities and others.

Jia, T., et al., 2021. LogFlash: Real-time streaming anomaly detection and diagnosis from system logs for large-scale software systems. In: 2021 IEEE 32nd International Symposium on Software Reliability Engineering. ISSRE, IEEE, pp. 80–90.

Le, V.-H., et al., 2021. Log-based anomaly detection without log parsing. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 492–504.

Le, V.-H., et al., 2022. Log-based anomaly detection with deep learning: How far are we? In: Proceedings of the 44th International Conference on Software Engineering. pp. 1356–1367.

Liu, Y., et al., 2022. UniParser: A unified log parser for heterogeneous log data. In: Proceedings of the ACM Web Conference 2022. pp. 1893–1901.

Lu, S., et al., 2017. Log-based abnormal task detection and root cause analysis for spark. In: 2017 IEEE International Conference on Web Services. ICWS, IEEE, pp. 389–396.

Messaoudi, S., et al., 2018. A search-based approach for accurate identification of log message formats. In: 2018 IEEE/ACM 26th International Conference on Program Comprehension. ICPC, IEEE, pp. 167–16710.

Mi, H., et al., 2013. Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems. IEEE Trans. Parallel Distrib. Syst. 24 (6), 1245–1255.

Nagappan, M., et al., 2010. Abstracting log lines to log event types for mining software system logs. In: 2010 7th IEEE Working Conference on Mining Software Repositories. MSR 2010, IEEE, pp. 114–117.

Nedelkoski, S., et al., 2020. Self-supervised log parsing. In: Joint European Conference on Machine Learning and Knowledge Discovery in Databases. Springer, pp. 122–138.

Shima, K., 2016. Length matters: Clustering system log messages using length of words. ArXiv preprint arXiv:1611.03213.

Tang, L., et al., 2011. LogSig: Generating system events from raw textual logs. In: Proceedings of the 20th ACM International Conference on Information and Knowledge Management. pp. 785–794.

Vaarandi, R., 2003. A data clustering algorithm for mining patterns from event logs. In: Proceedings of the 3rd IEEE Workshop on IP Operations & Management. IPOM 2003 (IEEE Cat. No. 03EX764), IEEE, pp. 119–126.

Vaswani, A., et al., 2017. Attention is all you need. Adv. Neural Inf. Process. Syst. 30.

Wu, Y., et al., 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. ArXiv preprint arXiv:1609.08144.

Xiao, T., et al., 2020. Lpv: A log parser based on vectorization for offline and online log parsing. In: 2020 IEEE International Conference on Data Mining. ICDM, IEEE, pp. 1346–1351.

Xu, W., et al., 2009. Detecting large-scale system problems by mining console logs. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles. pp. 117–132.

Zhang, S., et al., 2017. Syslog processing for switch failure diagnosis and prediction in datacenter networks. In: 2017 IEEE/ACM 25th International Symposium on Quality of Service. IWQoS, IEEE, pp. 1–10.

Zhang, X., et al., 2019. Robust log-based anomaly detection on unstable log data. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 807–817.

Zhu, J., et al., 2019. Tools and benchmarks for automated log parsing. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice. ICSE-SEIP, IEEE, pp. 121–130.

Siyu Yu is a Master student in the School of Computer and Electronic Information at Guangxi University, Guangxi, China, supervised by Professor Ningjiang Chen. His research lies within Software Engineering, with special interests in software log analysis. He obtained his BS from Guangxi University. Contact him at gaiusyu6@gmail.com, his github repository for research is https://github.com/gaiusyu.

Ningjiang Chen (Member, IEEE), received the Ph.D. degree from the Institute of Software, Chinese Academy of Sciences, in 2006. He is currently a Professor at Guangxi University. His research interests include intelligent software engineering, big data, and cloud computing, etc.

Yifan Wu is a Ph.D. student in the School of Software & Microelectronics at Peking University in China. He received his B.S. degree from University of Electronic Science and Technology of China in 2015. His current research interests include AIOps, software engineering.

Wensheng Dou received the Ph.D. degree from Institute of Software Chinese Academy of Sciences (ISCAS) in 2015. He is currently a professor at ISCAS. His research interests are program analysis and software testing.