Log-based Anomaly Detection Without Log Parsing

Van-Hoang Le and Hongyu Zhang[†]

The University of Newcastle, NSW, Australia

vanhoang.le@uon.edu.au, hongyu.zhang@newcastle.edu.au

Abstract—**Software systems often record important runtime information in system logs for troubleshooting purposes. There have been many studies that use log data to construct machine learning models for detecting system anomalies. Through our empirical study, we find that existing log-based anomaly detection approaches are significantly affected by log parsing errors that are introduced by 1) OOV (out-of-vocabulary) words, and 2) semantic misunderstandings. The log parsing errors could cause the loss of important information for anomaly detection. To address the limitations of existing methods, we propose NeuralLog, a novel log-based anomaly detection approach that does not require log parsing. NeuralLog extracts the semantic meaning of raw log messages and represents them as semantic vectors. These representation vectors are then used to detect anomalies through a Transformer-based classification model, which can capture the contextual information from log sequences. Our experimental results show that the proposed approach can effectively understand the semantic meaning of log messages and achieve accurate anomaly detection results. Overall, NeuralLog achieves F1-scores greater than 0.95 on four public datasets, outperforming the existing approaches.**

Index Terms—**Anomaly Detection, Log Analysis, Log Parsing, Deep Learning**

I. INTRODUCTION

High availability and reliability are essential for large-scale software-intensive systems [1], [2]. With the increasing complexity and scale of systems, anomalies have become inevitable. A small problem in the system could lead to performance degradation, data corruption, and even a significant loss of customers and revenue. Anomaly detection is, therefore, necessary for the quality assurance of complex software-intensive systems.

Software-intensive systems often generate console logs to record system states and critical events at runtime. Engineers can utilize log data to understand the system status, detect the anomalies, and identify the root causes. As the amount of logs could be huge, anomaly detection based on manual analysis of logs is time-consuming and error-prone. Over the years, many data-driven methods have been proposed to automatically detect anomalies by analyzing log data [3], [4], [5], [6], [7], [8], [9]. Machine learning-based methods (such as Logistic Regression [10], Support Vector Machine [6], Invariant Mining [11]) extract log events and adopt supervised or unsupervised learning to detect the occurrences of system anomalies.

Recently, some deep learning-based approaches have been proposed. For example, LogRobust [8] and LogAnomaly [12] adopt Word2vec model [13] to obtain embedding vectors of log events, then applied an LSTM model to detect anomalies.

[†]Hongyu Zhang is the corresponding author.

However, the existing approaches rely on log parsing to preprocess semi-structured log data. Log parsers remove the variable part from log messages and retain the constant part to obtain log events. To investigate the inaccuracy of log parsing, we have performed an empirical study on real-world log data. We find that existing log parsers produce a noticeable number of parsing errors, which directly downgrade anomaly detection performance. The log parsing errors are mainly due to the following two reasons: 1) The logging statements could frequently change during software evolution, resulting in new log events that were not seen in training data; 2) Valuable information could be lost while parsing log messages into log events, which may lead to misunderstanding of the semantic meaning of log messages. Our empirical study also finds that the log parsing errors can affect the follow-up anomaly detection task and decrease the detection accuracy.

To overcome the above-mentioned limitations of existing approaches, we propose NeuralLog, a novel anomaly detection approach, which can achieve effective and efficient anomaly detection on real-world datasets. Unlike the existing approaches, NeuralLog does not rely on any log parsing, thus preventing the loss of information due to log parsing errors. Each log message is directly transformed into a semantic vector, which is capable of capturing both semantic information embedded in log messages and the relationship between log messages. Then, taking a sequence of semantic vectors as input, a Transformer-based classification model is applied to detect anomalies. The Transformer-based model with multi-head self-attention mechanism [14] can learn the contextual information from the log sequences in the form of vector representations. As a result, NeuralLog is effective for log-based anomaly detection.

We have evaluated the proposed approach using four public datasets. The experimental results show that NeuralLog can understand the semantic meaning of log data and adequately handle OOV words. It achieves high F1-scores (all greater than 0.95) for anomaly detection and outperforms the existing log-based anomaly detection approaches.

The main contributions of this paper are as follows:

1) We perform an empirical study of log parsing errors. We find that existing log-based anomaly detection approaches are adversely affected by the log parsing errors introduced by the OOV words and semantic misunderstanding.

2) We propose NeuralLog, a novel deep learning-based approach that can detect system anomalies without log parsing. NeuralLog utilizes BERT, a widely-used pre-trained language representation, to encode the semantic

meaning of log messages.

3) We have evaluated NeuralLog using public datasets. The results confirm the effectiveness of NeuralLog for representing log messages and detecting anomalies.

II. BACKGROUND

A. Log Data

Large and complex software-intensive systems often produce a large amount of log data for troubleshooting purposes during system operation. Log data records the system's events and internal states during runtime. By analyzing logs, operators can better understand systems' status and diagnose the system when a failure occurs.

1	081109 213908 2549 INFO dfs.DataNode$DataXceiver: 10.251.39.192:50010 Served block blk_-5341992729755584578 to /10.251.39.192		
2	081109 214009 2594 INFO dfs.DataNode$DataXceiver: 10.250.5.237:50010 Served block blk_3166960787499091856 to /10.251.43.147		
3	081109 214043 2561 WARN dfs.DataNode$DataXceiver: 10.251.30.85:50010 Got exception while serving blk_-2918118818249673980 to /10.251.90.64		

Parsing

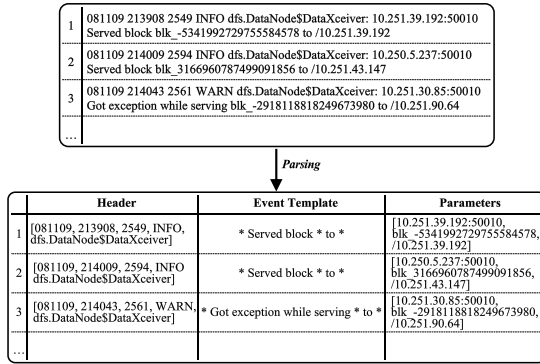|	Header	Event Template	Parameters
1	[081109, 213908, 2549, INFO, dfs.DataNode$DataXceiver]	* Served block * to *	[10.251.39.192:50010, blk_-5341992729755584578, /10.251.39.192]
2	[081109, 214009, 2594, INFO dfs.DataNode$DataXceiver]	* Served block * to *	[10.250.5.237:50010, blk_3166960787499091856, /10.251.43.147]
3	[081109, 214043, 2561, WARN, dfs.DataNode$DataXceiver]	* Got exception while serving * to *	[10.251.30.85:50010, blk_-2918118818249673980, /10.251.90.64]

Fig. 1. An Example of HDFS Logs and Parsed Results

Figure 1 shows a snippet of raw logs generated by HDFS (Hadoop Distributed File System). The raw log messages are semi-structured texts, which contain *header* and *content*. The header is determined by the logging framework and includes information such as timestamp, verbosity level (e.g., WARN/INFO), and component [15]. The log content consists of a constant part (keywords that reveal the event template) and a variable part (parameters that carry dynamic runtime information). Log parsing automatically converts each log message into a specific event template by removing parameters and keeping the keywords. The log events can be grouped into *log sequences* (i.e., series of log events that record specific execution flows) according to sessions or fixed/sliding time windows [16].

B. Log Parsing Methods

Log parsing automatically converts each log message into a specific event template by removing parameters and keeping the keywords. For example, the log template "* *Served block* * *to* *" can be extracted from the first log message in Figure 1. Here, "*" denotes the position of a parameter.

There are many log parsing techniques, including frequent pattern mining [17], [18], [19], clustering [20], [21], [22], language modeling [23], heuristics [24], [25], [26], etc. The heuristics-based approaches make use of the characteristics of logs and have been found to perform better than other techniques in terms of accuracy and time efficiency [15]. In this study, we evaluate four top-ranked parsers include

Drain [24], AEL [25], IPLoM [26] and Spell [27]. They utilize the characteristics of tokens (e.g., occurrences, position, relation, etc.) and special structures (e.g., a tree) to represent log messages and extract common templates. Drain applies a fixed-depth tree structure to represent log messages and extracts common templates effectively. Spell utilizes the longest common subsequence algorithm to parse logs in a stream manner. AEL separates log messages into multiple groups by comparing the occurrences between constants and variables. IPLoM employs an iterative partitioning strategy, which partitions log messages into groups by message length, token position, and mapping relation. These log parsers are widely used in existing studies and have proven their efficiency on real-world datasets [15], [16], [28], [29].

C. Log-based Anomaly Detection

Over the years, many log-based anomaly detection approaches have been proposed. Some of them are based on unsupervised learning methods, which require only unlabeled data to train. For example, Xu et al. [7] employed Principal Component Analysis (PCA) to generate two subspaces (normal space and anomaly space) of log count vectors. If a log sequence has its log count vector far from the normal space, it is considered an anomaly. IM [11] and ADR [4] discover the linear relationships among log events from log count vectors. Those log sequences that violate the relationship are considered anomalies. There are also many supervised anomaly detection approaches. For example, [30], [6], [10] represent log sequences as log count vectors, then applied Support Vector Machine (SVM), Logistic Regression (LR), and Decision Tree algorithm to detect anomalies, respectively. These approaches have many common characteristics. They all require a log parser to preprocess and extract log templates from log messages. Then, the occurrences of log templates are counted, resulting in log count vectors. Finally, a machine learning model is constructed to detect anomalies. Figure 2 shows an example of log sequence and log count vectors from log templatesproduced within Drain [24].
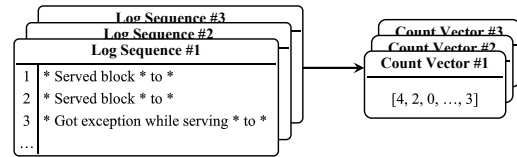

Fig. 2. An Example of Log Sequence, and Log Count Vector

In recent years, many deep learning-based models have been proposed to analyze log data and detect anomalies [5], [12], [8]. For example, DeepLog [5] first applies the Spell [27] parser to extract log templates. Then, it leverages the indexes of log templates and inputs them to an LSTM model to predict the next log templates. Finally, DeepLog detects anomalies by determining whether or not the incoming log templates are unexpected. LogAnomaly [12] uses log count vector to detect the anomalies reflected by anomalous log event numbers. It proposes a synonyms and antonyms based

method to represent the words in log templates. LogRobust [8] incorporates a pre-trained Word2vec model, namely FastText [31], and combines with TF-IDF weight [32] for learning the representation vectors of log templates, which are generated by Drain [24]. Then, these vectors input an Attention-based Bi-LSTM model to detect anomalies. Due to the imperfection of log parsing, the above methods tend to the lose semantic meaning of log messages, thus leading to inaccurate detection results.

III. AN EMPIRICAL STUDY OF LOG PARSING ERRORS

In this section, we describe an empirical study on the problem of existing log parsers and their impact on log-based anomaly detection. We use two public datasets in our study, namely Blue Gene/L (BGL) [33], [34] and Thunderbird [33], [34]. The datasets are collected between 2004 and 2006 from real-world supercomputing systems [33] and consist of 14,672,653 log messages in total, among which 353,394 log messages are manually labeled as anomalies.

A. Log Parsing Errors Introduced by OOV Words

During development and maintenance, developers can add new log statements to source code and modify the content of existing log statements. Besides, runtime information can be added to log messages as parameters to record system status. As a consequence, new words, i.e., out-of-vocabulary (OOV) words, frequently appear in log data. For example, for a log message *"memory manager address **parity** error"* in BGL, if the word "parity" did not appear in historical logs, it is an OOV word. To determine OOV words, we first sort log messages by the timestamps of logs and leverage the front $P\%$ (according to the timestamps of logs) as the training data and the rest as the testing data. Then, we split each training log message into a set of tokens by the whitespace character and build a vocabulary from these tokens. *OOV words* are those words in testing data that do not exist in the vocabulary. In this section, we increase the percentages of training data from 20% to 80%. Then we calculate the proportion of OOV words in all the splits.

To facilitate understanding, we use the BGL data at the 60/40 splitting (the first 60% of the BGL dataset is used for training, and the rest is for testing) to explain in detail the analysis of OOV words. The training set contains 153,786 unique words, and the testing set contains 384,730 unique words. Among the unique words in the testing set, 362,123 words (94.12%) are unseen in the training set. These OOV words only concentrate in a small subset of logs (i.e., 8.51% of the testing set, which is 160,403 out of 1,885,398 log messages). These OOV words result in 1,304 unseen log events (i.e., log templates with OOV words) in the testing set, accounting for 86.59% of the total number of log events in the testing set.

Figure 3 shows the percentages of OOV words in testing data when the percentages of training data increase from 20% to 80% on BGL and Thunderbird datasets. On the BGL dataset, there are always more than 80% of words in the testing

set that are unseen in the training set. On the Thunderbird dataset, with the growth of training data, the proportion of OOV words in the testing set is gradually decreasing. However, when we use 80% Thunderbird logs to train the model, we still find 30.4% of OOV words in the testing set.
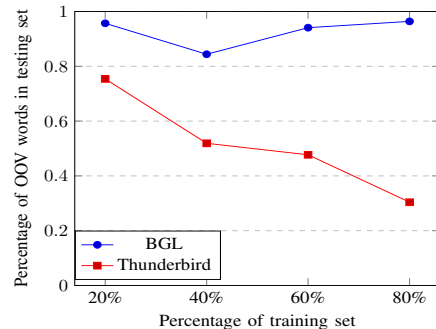

Fig. 3. Analysis of OOV words in log messages in public datasets

Next, we evaluate the number of log messages (log lines) and log templates that contain OOV words. The percentage of log messages containing OOV words is shown in Figure 4(a). It can be seen that the percentage of log messages containing OOV words in the testing set of both BGL and Thunderbird datasets is small. When we use 80% logs to train the model, we find only 6.7% and 1.7% log messages containing OOV words in the testing set of the BGL and Thunderbird dataset, respectively.
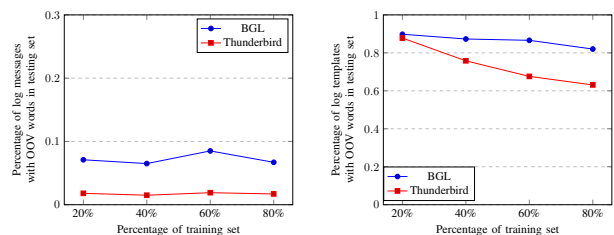
(a) Log messages with OOV words   (b) Log templates with OOV words

Fig. 4. Analysis of log with OOV words

Figure 4(b) shows the percentages of log templates (produced by Drain [24]) that contains OOV words on BGL and Thunderbird datasets, as the percentage of training data increases from 20% to 80%. We observe that all testing sets have log templates with OOV words, even when trained with 80% of the data. The proportion of log templates containing OOV words on the BGL dataset is always more than 80%, no matter how much data is used for training. The percentage of templates containing OOV words on the Thunderbird dataset decreases with the growth of training data, but still, more than 60% when 80% of data is trained.

The results show that a small number of log messages containing OOV words can produce many unseen log events in the testing set. There are three main reasons for this finding:

494

- Many log events only appear during a specific period [5]. For example, there are 842 events that only appear in the last 20% of logs, in the 80/20 splitting.
- The distribution of log events is imbalanced. For example, the event *"generating ∗"* appears in 1,706,751 log messages (35.95% of the dataset), while others such as *"memory manager ∗ buffer ∗"* only appear less than 100 times.
- OOV words can cause log parsing errors and lead to many extra log events. These extra log events usually appear a few times but still make up a majority of log templates. For example, 1,165 log events only appear once in the BGL dataset.

Our finding indicates that anomaly detection methods based only on log events could lead to many inaccurate detection results. For example, SVM [30] and LR [6], which transform log sequences into log count vectors, cannot take new log events as input because the dimension of log count vectors is fixed (i.e., the number of original log events). Moreover, DeepLog [5], using the indexes of log templates to predict the next log event, considers all new log events as anomalies because they cannot be predicted by the model.

B. Log Parsing Errors Introduced by Semantic Misunderstanding

We identify two main cases of parsing errors that are introduced by semantic misunderstanding:

- Case 1: Misidentifying parameters as keywords.
- Case 2: Misidentifying keywords as parameters.

Case 1:

```
- Parsing results:
L3 ecc status register: 00200000      → L3 ecc status register: *
L3 global control register: 0x001249f0 → L3 * control register: *
- Ground truth Log Template:
L3 * * register: *
```

Case 2:

```
- Parsing results:
machine check enable      →     machine check *
machine check interrupt   →     machine check *
- Ground truth Log Templates:
machine check enable
machine check interrupt
```

Fig. 5. Examples of Log Parsing Error (Drain)

For Case 1, the parameters in log messages are misidentified as keywords and included in the log templates produced by the log parsers, thus leading to many extra log events. We compare the parsed template of each log message with the ground truth of the BGL dataset. If a template contains more keywords than the ground truth, it is considered wrongly parsed and an extra log event. The two log messages in Case 1 in Figure 5 only refer to one log template but are parsed into two different log templates. Figure 6 shows the percentages of extra log events produced by the four log parsers on two datasets. For example, there are about 80% extra log events on the BGL dataset and 72% extra log events on the Thunderbird dataset using Drain.
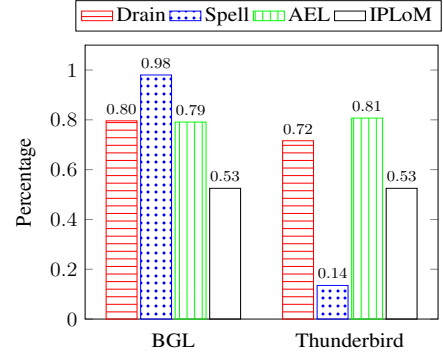

Fig. 6. Percentages of extra log events produced by four log parsers

For Case 2, some essential keywords in log messages could be removed after log parsing, resulting in different log messages being parsed into one log event. Figure 5 shows an example of this case. Two different log messages are parsed into the same log event *"machine check ∗"*. However, one indicates a normal behavior (i.e. *"machine check enable"*), while the other indicates a system anomaly (i.e., *"machine check interrupt"*). The errors of this type make the detection model difficult to distinguish between normal or abnormal logs based only on log events. Figure 7 shows examples of Case 2 parsing errors introduced by the four log parsers. Each example shows one normal log and one abnormal log which are parsed into the same log event. Valuable information such as the reason for login failure (i.e., Figure 7(a)) is missing from log events and leads to many wrong detection results.

```
- Anomaly: ciod: LOGIN chdir(/p/gb1/stel-
la/RAPTOR/2183) failed: Input/output error
- Normal: ciod: LOGIN chdir(/home/berts-
ch2/src/bgl_hello) failed: Permission denied
- Parsed event: ciod: LOGIN ∗ failed: ∗ ∗
```
(a) Errors introduced by Drain

```
- Anomaly: mptscsih: ioc0: attempting task
abort! (sc=00000101bfc7a480)
- Normal: mptscsih: ioc0: task abort: SUC-
CESS (sc=00000101bfc7a480)
- Parsed event: mptscsih ioc0 ∗ ∗ ∗ sc ∗
```
(b) Errors introduced by Spell

```
- Anomaly: floating point unavailable inter-
rupt
- Normal: floating point instr. enabled.....1

- Parsed event: floating point ∗ ∗
```
(c) Errors introduced by AEL

```
- Anomaly: ciod: Error creating node map
from file map.dat: No child processes
- Normal: ciod: Error creating node map
from file map.dat: Bad file descriptor
- Parsed event: ciod Error creating node map
from file ∗ ∗ ∗ ∗
```
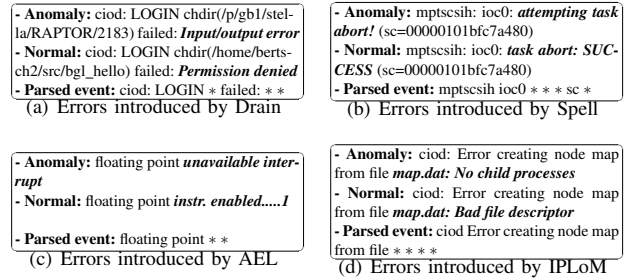(d) Errors introduced by IPLoM

Fig. 7. Examples of Valuable Information Removed by Log Parser

Table I shows examples of wrongly parsed log templates by the Drain parser [24]. For instance, on the BGL dataset, there are 6,541 log messages that have the template *"floating point ∗ ∗"*. However, only log messages in the form of *"floating point unavailable interrupt"* are labeled as anomalies, while others (such as *"floating point instr. enabled"* or *"floating point alignment exceptions"*) indicate normal system behavior. Similarly, Drain also produces 899 log messages that have the log templates indicating both normal and abnormal states on the Thunderbird dataset.

We also observe similar results for other log parsers. On BGL, the numbers of misidentified log messages produced by Spell, AEL, and IPLoM are 58,228, 20,154, and 31,298,

respectively. On Thunderbird, the numbers of misidentified log messages produced by Spell, AEL, and IPLoM are 3,851, 1,463, and 5,687, respectively.

TABLE I
EXAMPLES OF LOG PARSING ERRORS INTRODUCED BY DRAIN

	#Anom.	Log Template	Occu.
BGL	348,460	floating point * *	6,541
		machine check *	6,594
		ciod: Error creating node map from file * * * *	2,952
		ciod: LOGIN * failed: * *	1,289
Thunderbird	4,934	mptscsih: ioc0: attempting * * *	771
		EXT3-fs error (device * * * * * aborted)	121
		Out of Memory: Killed process * *	7

Note: $\#Anom.$ denotes the number of anomalies. $Occu.$ denotes the number of occurrences of the log template.

C. The Impact of Log Parsing Errors on Anomaly Detection

The existing approaches share a common process: they all utilize a log parser to parse the log messages into log events (i.e., the templates of log messages), construct log sequences, and then build unsupervised or supervised machine learning models to detect anomalies. The existing approaches can be adversely affected by the log parsing errors introduced by the OOV words and semantic misunderstanding.

In this section, we evaluate the impact of log parsing errors on two representative anomaly detection methods SVM-based method [30], and LogRobust [8]. SVM represents those ML-based approaches that use the log count vectors as input. LogRobust represents recent DL-based approaches that utilize the semantic vectors of log templates as input. They both use a log parser to generate a set of log events. SVM-based method [30] represents log sequences as log count vectors and then constructs a hyperplane to separate normal and abnormal samples in a high-dimension space. LogRobust [8] incorporates a pre-trained Word2vec model [31] to learn the semantic vectors of log templates instead of counting log events' occurrences. Using the Word2vec model allows LogRobust to discover the semantic relationship between log events and handle the instability of log data.

Figure 8 shows the results of the SVM-based method and LogRobust with four different log parsers (i.e., Drain [24], Spell [27], AEL [25], and IPLoM [26]). We observe that the performance of current anomaly detection methods is affected by the accuracy of log parsing, and different log parsers could lead to different results. SVM achieves better results when using Drain [24] and AEL [25] since these parsers produce a smaller amount of inaccurate log events, as discussed in Section III-B. Figure 8(a) and Figure 8(c) show that SVM with Drain achieves an F1-score of 0.46 and 0.50 on BGL and Thunderbird datasets, respectively. While SVM with Spell only produces F1-scores of 0.29 on the BGL dataset and 0.17 on the Thunderbird dataset. LogRobust can achieve better results than SVM since LogRobust can identify unstable log events with similar semantic meaning through semantic vectorization. Still, LogRobust suffers from the log parsing errors caused by semantic misunderstanding (see Section III-B) and achieves F1-scores of less than 0.8 on both datasets.
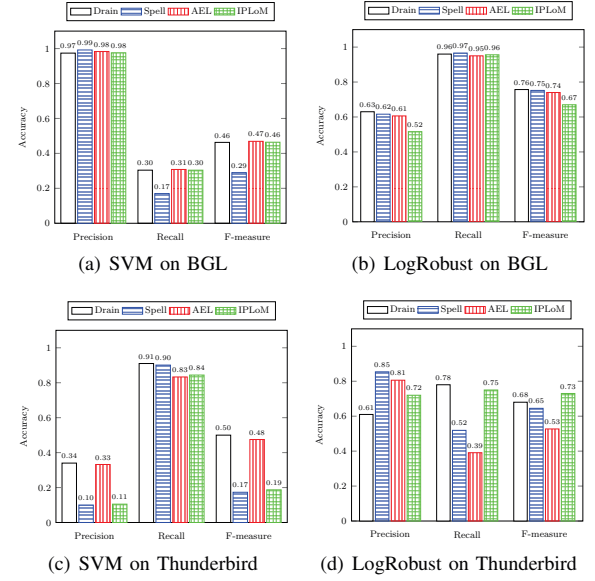

Fig. 8. Results of anomaly detection with different log parsers on BGL and Thunderbird datasets

Next, we manually fix the errors produced by log parsing methods on the BGL dataset, then apply SVM and LogRobust to confirm whether or not the accuracy of anomaly detection methods is improved if log parsing performs more accurately. We leverage the ground truth log templates for the BGL dataset from [35]. For each wrongly parsed log message, we match it with the most similar log template in the ground truth. After the fixing process, the outputs produced by the log parser are actually the ground truth. Also, the outputs of different log parsers are the same after fixing (as they are all the same as the ground truth). Figure 9 shows the accuracy (measured in terms of F-measure) of each individual parser before and after fixing the log parsing errors. We can see that both SVM and LogRobust perform better when the log parsing errors are fixed (on average, 25% improvement for LogRobust and 29% improvement for SVM).
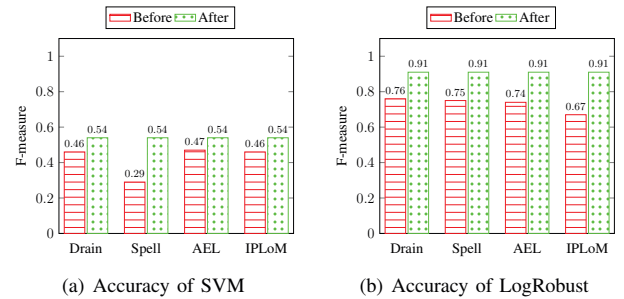

Fig. 9. The accuracy of anomaly detection before and after fixing the log parsing errors on the BGL dataset

Overall, the results show that log parsing accuracy affects the performance of anomaly detection. Existing log parsing methods cannot handle well the OOV words in new logs,

thus losing semantic information while detecting anomalies. Furthermore, current log parsing methods could produce errors due to semantic misunderstanding. Therefore, existing anomaly detection methods that leverage log events are unable to achieve satisfying results due to the imperfections of log parsing methods.

IV. NEURALLOG: LOG-BASED ANOMALY DETECTION WITHOUT LOG PARSING

To overcome the limitation of existing approaches, we propose NeuralLog, a new log-based anomaly detection approach that directly uses raw log messages to detect anomalies. The overview of the proposed approach is shown in Figure 10. Overall, NeuralLog consists of three steps: preprocessing (Section IV-A), neural representation (Section IV-B), and transformer-based classification (Section IV-C). The first step is log preprocessing. After that, each log message is encoded into a semantic vector by using BERT. In this way, our approach can prevent the loss of valuable information from log messages. Finally, we leverage the Transformer [14] model to detect the anomalies.
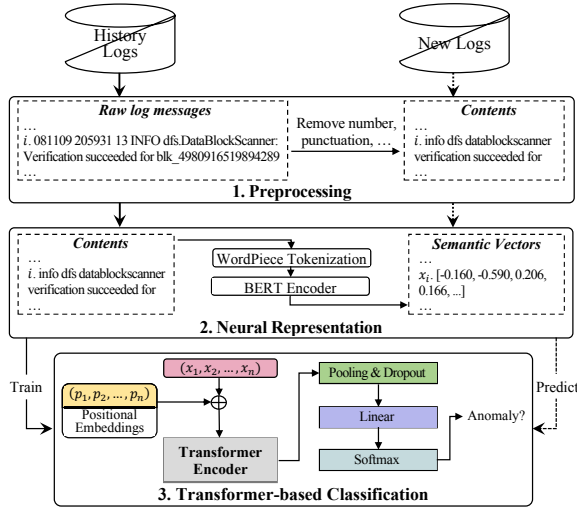

Fig. 10. An Overview of NeuralLog

A. Preprocessing

Preprocessing log data is the first step for building our model. In this step, we first tokenize a log message into a set of word tokens. We use common delimiters in the logging system (i.e., white space, colon, comma, etc.) to split a log message. Then, every capital letter is converted to a lower letter, and we remove all non-character tokens from the word set. These non-characters contain operators, punctuation marks, and number digits. This type of tokens is removed since it usually represents variables in the log message and is not informative. As an example, the raw log message *"081109 205931 13 INFO dfs.DataBlockScanner: Verification succeeded for blk_-4980916519894289629"* is first split into a set of words based on common delimiters. Then non-character

tokens are excluded from the set. Finally, a set of words {*info, dfs, datablockscanner, verification, succeeded*} is obtained.

B. Neural Representation

Each log message records a system event with its header and message content. The message header contains fields determined by the logging framework, such as component and verbosity level. The message content written by developers reflects a specific state of the system. Existing methods usually analyze only message content and remove other information. In this paper, NeuralLog uses all textual information such as verbosity, component, and content to extract the semantic meaning of log messages. In order to reserve semantic information and capture relationships among existing and new log messages, the representation phase tries to represent log messages in the vector format.

1) Subword Tokenization: Tokenization can be considered as the first step to handle OOV words. In our work, we adopt the WordPiece tokenization [36], [37], which is widely used in many recent language modeling studies [38], [39], [40].

WordPiece includes all the characters and symbols into its base vocabulary first. Instead of relying on the frequency of the pairs, WordPiece chooses the one that maximizes the training data's likelihood. It trains a language model starting from the base vocabulary and picks the pair with the highest likelihood. This pair is added to the vocabulary, and the language model is again trained on the new vocabulary. These steps are repeated until the desired vocabulary is reached. For example, the rare word *"datablockscanner"* is split into more frequent subwords: {*"data", "block", "scan", "ner"*}. In this way, the number of OOV words is reduced and their meanings are captured.

The reason we choose WordPiece is that it can effectively handle the OOV words and reduce the vocabulary size. Compared with other tokenization (chunking) approaches, WordPiece is more effective. For example, space/stemming/camel case based tokenization strategies can lead to many OOV words and a big vocabulary [41].

2) Log Message Representation: After preprocessing and tokenization, NeuralLog transforms each log message into a set of words and subwords. Conventionally, words of log content are further transformed into vectors by using Word2Vec [42], then the representation vector of each sentence would be calculated based on the word vectors. However, Word2Vec produces the same embedding for the same word. In many cases, a word can have different meanings based on its position and context. BERT [38] is a recent deep learning representation model that has been pre-trained on a huge natural language corpus. In our work, we employ the feature extraction function of pre-trained BERT to obtain the semantic meaning of log messages.

More specifically, after tokenizing, the set of words and subwords is passed to the BERT model and encoded into a vector representation with a fixed dimension. NeuralLog utilizes the BERT base model [43] that contains 12-layers of transformer encoder and 768-hidden units of each transformer.

497

Each layer generates embeddings for each subword in a log message. We use the word embeddings generated by the last encoder layer of BERT in our work. Then, the embedding of a log message is calculated as the average of its corresponding word embeddings. As any word that does not occur in the vocabulary (i.e., OOV words) is broken down into subwords, BERT can learn the representation vector of those OOV words based on the meaning of subword collections. Besides, the positional embedding layer allows BERT to capture the representation of a word based on its context in a log message. BERT also contains self-attention mechanisms that can effectively measure the importance of each word in a sentence.

C. Transformer-based Classification

To better understand the semantics of logs, we adopt the transformer model [14], which has been introduced to overcome the limitations of RNN-based models. Taking the semantic vectors of log messages as input (i.e. $\mathbf{X} = \{x_1, x_2, \ldots, x_n\}$), we use a transformer encoder-based model for anomaly detection. In this section, we briefly describe the proposed transformer-based classification model, which contains Positional Encoding and Transformer Encoder.

a) Positional Encoding: The order of a log sequence conveys important information for the anomaly detection task. BERT encoder represents a log message into a fixed-dimensional vector where log messages with similar meanings are closer to each other. However, those vectors do not contain the relative position information of log messages in a log sequence. Therefore, a sinusoidal encoder is applied to generate an embedding p_i using sin and cos functions for each position i in the log sequence $\mathbf{X}$ [14]. Then, p_i is added to the semantic vector x_i at position i, and $x_i + p_i$ will be used to feed the transformer-based model (see Figure 10 (Step 3)). In this way, the model can learn the relative position information of each log message in the sequence and can distinguish log messages at different positions.

b) Transformer Encoder: This model is based on the transformer architecture [14], which contains self-attention layers followed by position-wise feed-forward layers. Given an input $\mathbf{X} = \{x_1, x_2, \ldots, x_n\}$, the positional embeddings are added before it enters into the transformer. In the transformer module, multi-head attention layers calculate the attention score matrices for each log message with different attention patterns. The attention score is calculated by training the query and key matrices of the attention layers. Different attention patterns are obtained with multi-head self-attention layers, which enable the model to consider which attention score is significant. The inter-layer features are connected into a feed-forward network, which contains two fully connected layers in order to reach the combination of different attention scores. Then, the output of the transformer model is fed into the pooling, dropout, and a fully connected layer. The class probabilities, which identify normal/abnormal log sequences, are calculated using the softmax classifier. The architecture of the classification model is shown in Figure 10 (Step 3).

D. Anomaly Detection

Following the above steps, we can train a transformer-based model for log-based anomaly detection. When a set of new log messages arrives, NeuralLog firstly conducts preprocessing. Then it transforms the new log messages into semantic vectors. The log sequence, represented as a list of semantic vectors, is fed into the trained model. Finally, the transformer-based model can predict whether this log sequence is anomalous or not.

V. EVALUATION

A. Experimental Design

1) Research Questions: In this section, we evaluate our approach by answering the following research questions (RQs):

RQ1: How effective is NeuralLog in log-based anomaly detection?

RQ2: How effective is NeuralLog in understanding the semantic meaning of log data?

RQ3: How effective is NeuralLog under different settings?

2) Datasets: In this paper, we evaluate NeuralLog on four public datasets [44], namely HDFS, Blue Gene/L, Thunderbird, and Spirit. HDFS dataset [34], [7] contains 11,175,629 log messages collected from a Hadoop Distributed File System on the Amazon EC2 platform. Each session identified by block ID in the HDFS dataset is labeled as normal or abnormal. BGL dataset [33], [34] contains 4,747,963 log messages collected from the Blue Gene/L supercomputer at Lawrence Livermore National Labs. Thunderbird and Spirit datasets [33] were collected from two real-world supercomputers at Sandia National Labs. Each log message in these datasets was manually labeled as anomalous or not. In this experiment, we leverage 10 million continuous log messages from the Thunderbird dataset, and 1GB log messages from the Spirit dataset, which were also used in prior work [45]. The details of the datasets are shown in Table II.

TABLE II
THE DETAILS OF LOG DATASETS

	Category	Size	#Messages	#Anomalies
HDFS	Distributed system	1.5 G	11,175,629	16,838
Blue Gene /L	Supercomputer	743 M	4,747,963	348,460
Thunderbird	Supercomputer	1.4 G	10,000,000	4,934
Spirit	Supercomputer	1.0 G	7,983,345	768,142

3) Implementation and Environment: In our experiments, NeuralLog has one layer of the transformer encoder. The number of attention heads is 12, and the size of the feed-forward network that takes the output of the multi-head self-attention mechanism is 2048. The Transformer-based model of NeuralLog is trained using AdamW optimizer [46] with the initial learning rate of $3e - 4$. We set the mini-batch size and the dropout rate to 64 and 0.1, respectively. We use the cross-entropy as the loss function. We train the Transformer-based model for a maximum of 20 epochs and perform early stopping for five consecutive iterations.

498

We implement NeuralLog with Python 3.6 and Keras toolbox and conduct experiments on a server with Windows Server 2012 R2, Intel Xeon E5-2609 CPU, 128GB RAM, and an NVIDIA Tesla K40c.

4) Evaluation Metrics: To measure the effectiveness of NeuralLog in anomaly detection, we use the Precision, Recall, and F1-Score metrics. We calculate these metrics as follows:

- *Precision:* the percentage of correctly detected abnormal log sequences amongst all detected abnormal log sequences by the model. $Precision = \frac{TP}{TP+FP}$.
- *Recall:* the percentage of log sequences that are correctly identified as anomalies over all real anomalies. $Recall = \frac{TP}{TP+FN}$.
- *F1-Score:* the harmonic mean of *Precision* and *Recall*. $F1 - score = \frac{2*Precision*Recall}{Precision+Recall}$

TP (True Positive) is the number of abnormal log sequences the are correctly detected by the model. FP (False Positive) is the number of normal log sequences that are wrongly identified as anomalies. FN (False Negative) is the number of abnormal log sequences that are not detected by the model.

B. RQ1: How effective is NeuralLog?

This RQ evaluates whether or not NeuralLog can work effectively on public log datasets. For the HDFS dataset, we construct log sequences by correlating log messages with the same block ID, as the data is labeled by blocks. Then, we randomly select 80% of log sequences for training, and the rest of the dataset is used for testing. For BGL, Thunderbird, and Spirit datasets, we first sort the log messages by time. Then, we leverage the first 80% (according to the timestamps of logs) log messages as the training set and the rest 20% as the testing set. This design ensures that the testing data contains new log messages previously unseen in the training set. Following the previous work [47], [12], we apply a sliding window with a length of 20 messages and a step size of 1 message to construct log sequences.

TABLE III
RESULTS OF DIFFERENT METHODS ON PUBLIC DATASETS

Dataset		LR	SVM	IM	LogRobust	Log2Vec	NeuralLog
HDFS	P	0.99	0.99	1.00	0.98	0.94	0.96
	R	0.92	0.94	0.88	1.00	0.94	1.00
	F1	0.96	0.96	0.94	**0.99**	0.94	0.98
BGL	P	0.13	0.97	0.13	0.62	0.80	0.98
	R	0.93	0.30	0.30	0.96	0.98	0.98
	F1	0.23	0.46	0.18	0.75	0.88	**0.98**
Thunder-bird	P	0.46	0.34	-	0.61	0.74	0.93
	R	0.91	0.91	-	0.78	0.94	1.00
	F1	0.61	0.50	-	0.68	0.84	**0.96**
Spirit	P	0.89	0.88	-	0.97	0.91	0.98
	R	0.96	1.00	-	0.94	0.96	0.96
	F1	0.92	0.93	-	0.95	0.95	**0.97**

'-' denotes timeout (30 hours), P denotes Precision, R denotes Recall, and F1 is the F1-score.

We compare the results of NeuralLog and five existing approaches, including Support Vector Machine-based approach **(SVM)** [6], Logistic Regression-based approach **(LR)** [10], Invariant Mining **(IM)** [11], **LogRobust** [8], and **Log2Vec**

[48]. Traditional approaches, such as SVM, LR, and IM, transform the log sequences into *log count vectors*, then build unsupervised or supervised machine learning models to detect anomalies. In our work, we utilize Drain [24] to generate the log events for SVM, LR, and IM. LogRobust incorporates a pre-trained Word2vec model [31] to learn the representations vector of log templates instead of counting the occurrences of log events. LogRobust then leverages an Attention-based Bi-LSTM to learn and detect anomalies. Log2Vec [48] accurately extracts the semantic and syntax information from log messages and leverages the Deeplog [5] model to improve the accuracy of anomaly detection. We do not compare with DeepLog [5] because previous studies already showed that Log2Vec outperforms DeepLog [48]. Note that there are some other recent state-of-the-art methods such as LogAnomaly [12]. However, LogAnomaly [12] has no publicly available implementation and requires operators' domain knowledge (to manually add domain-specific synonyms and antonyms). Therefore, it is not experimentally compared in this paper.

The comparison results are shown in Table III. Overall, NeuralLog achieves the best results on BGL, Thunderbird, and Spirit datasets and comparable results on the HDFS dataset. It is worth noting that the recall value achieved by NeuralLog on the HDFS dataset is 1.00, which means that NeuralLog can identify all anomalies captured by the dataset with high precision. NeuralLog achieves the best F1-score of 0.98 on the BGL dataset, 0.96 on the Thunderbird dataset, and 0.97 on the Spirit data.

As discussed in Section II-C, existing approaches (including SVM, Decision Tree, and LR) are heavily affected by the accuracy of log parsing. Besides, these approaches cannot capture the semantic information of log messages. Therefore, these approaches perform poorly on BGL and Thunderbird datasets when the log parsing is inaccurate. They can achieve a high F1-Score on the Spirit dataset since the parsing error rate is only 0.1% for this dataset.

LogRobust [8], which encodes log templates into semantic vectors using the FastText pre-trained model [31], cannot work well on 2 out of 4 datasets. LogRobust shows a lower F1-Score of 0.75 and 0.68 on BGL and Thunderbird datasets, respectively. The main reason is that LogRobust utilizes the Drain log parser [24] to obtain log templates. As aforementioned in Section III-C, the Drain parser could inaccurately parse a noticeable number of log messages on BGL and Thunderbird datasets. Log2Vec [48] transforms raw log messages into semantic vectors, thus can avoid errors from log parsers. Besides, Log2Vec also adopts MIMICK [49], an approach of inducing word embedding from character-level features to handle OOV words to improve anomaly detection performance. However, it is hard to extract contextual information from characters to form meaningful words [50], [51], [52]. Therefore, Log2Vec could not effectively handle some domain-specific words, such as technical terms or entity names [50], [51], [52]. Consequently, compared to NeuralLog that uses subword-level feature to handle OOV words, Log2Vec achieves lower F1-scores on BGL and Thunderbird datasets

(0.88 and 0.84, respectively).

We also evaluate the time efficiency of NeuralLog on the four datasets. On average, it takes NeuralLog 14.3 minutes per dataset to encode all log messages and 5.2 minutes to train a detection model (20 epochs). The average time of the anomaly detection phase is 3.1 milliseconds per log sequence. Baseline methods that require log parsing take 102 minutes on average for preprocessing. Log2Vec spends an average of 314 minutes in the preprocessing phase. SVM and LR models can finish training within 0.5~0.7 minutes. LogRobust and Log2Vec can train a detection model in an average of 2.2 and 13.1 minutes, respectively. In the detection phase, it takes LogRobust 0.2 milliseconds per log sequence, and it is 26.3 milliseconds for Log2Vec. NeuralLog can scale to large datasets. For example, NeuralLog is able to handle the HDFS dataset, which contains 11,175,629 log messages. It took NeuraLog 19.7 minutes for preprocessing, 7.2 minutes for training, and 0.6 minutes for testing to perform the experiment for this RQ on the HDFS dataset.

In summary, the experimental results confirm that Neural-Log can work effectively and efficiently for log-based anomaly detection.

C. RQ2: How effective is NeuralLog in understanding the semantic meaning of log data?

In this section, we evaluate the ability of NeuralLog to capture the semantic meaning of log messages. To this end, we examine the effectiveness of the encoding component that represents log messages as semantic vectors and the subword tokenization component that handles OOV words.

In NeuralLog, we preprocess the raw log messages and directly encode the preprocessed log messages into semantic vectors. We compare NeuralLog with two variants:

- NeuralLog-Index: the indexes of log templates, obtained by Drain [24], are simply encoded into numeric vectors and passed to the Transformer model for anomaly detection. The rest of NeuralLog is kept the same.
- NeuralLog-Template: we utilize BERT to encode the log templates produced by the Drain [24] into semantic vectors. We then feed these semantic vectors to the Transformer model for anomaly detection. The rest of NeuralLog is kept the same.

Table IV shows the results of two variants of NeuralLog. We can see that, on HDFS and Spirit dataset, these two variants can achieve high F1-scores. The reason is that log parsers perform well on these datasets. We find that the parsing error rate on the Spirit dataset is only 0.1%. Besides, the HDFS system records relatively simple operations with only 29 event types, making log parsers easy to analyze. In contrast, the results of the variants on BGL and Thunderbird datasets are greatly affected by log parsing methods because they cannot precisely represent the meaning of log messages, especially when using the indexes of log templates. For example, the model using log templates' indexes and template embeddings only achieve F1-scores of 0.46 and 0.90 on the BGL dataset,

which are much lower than the 0.98 F1-score achieved by NeuralLog (which uses raw log messages).

TABLE IV
RESULTS OF DIFFERENT REPRESENTATION METHODS

Dataset	Metric	NeuralLog-Index	NeuralLog-Template	NeuralLog
HDFS	Precision	0.93	0.93	0.96
	Recall	1.00	1.00	1.00
	F1-Score	0.96	0.96	**0.98**
BGL	Precision	0.98	0.92	0.98
	Recall	0.30	0.88	0.98
	F1-Score	0.46	0.90	**0.98**
Thunderbird	Precision	0.58	0.89	0.93
	Recall	0.98	0.91	1.00
	F1-Score	0.73	0.90	**0.96**
Spirit	Precision	0.96	0.93	0.98
	Recall	0.95	0.95	0.96
	F1-Score	0.95	0.94	**0.97**

We next evaluate whether or not NeuralLog can effectively handle OOV words. NeuralLog utilizes WordPiece [36], [37] to split an OOV word into a set of subwords and then extracts the embedding of the OOV words based on its subwords. We compare NeuralLog with two variants:

- NeuralLog-Word2Vec: We use a pre-trained Word2vec model [31] to generate the embeddings of log messages. Those words that do not exist in the vocabulary are removed from log messages. Then, embedding vectors are passed to the Transformer model to detect anomalies.
- NeuralLog-NoWordPiece: We exclude WordPiece tokenizer from the model (see Figure 10). Log messages, after preprocessing, are directly input to the BERT model to obtain the semantic vectors. These vectors are then input to the Transformer model for anomaly detection. In this way, OOV words that do not exist in the vocabulary will be removed instead of broken down into subwords.

The experimental results are shown in Table V. NeuralLog achieves the best performance since it utilizes WordPiece [36], [37] to tokenize an OOV word into a set of subwords. Therefore, the meaning of an unseen word is kept by its subwords. Both variants achieve lower F1-scores than NeuralLog since they rely on a fixed-size vocabulary and cannot handle the OOV words. For example, on the Thunderbird dataset, F1-scores achieved by NeuralLog-Word2Vec and NeuralLog-NoWordPiece are 0.80 and 0.90, respectively, which are much lower than the F1-score of 0.96 achieved by NeuralLog.

In summary, our results show that NeuralLog can effectively represent the semantic meaning of log messages. Since NeuralLog uses raw log messages (after preprocessing) for anomaly detection, the problem of inaccurate log parsing can be avoided. The results also show that NeuralLog can effectively learn the meaning of OOV words.

D. RQ3: Effectiveness of NeuralLog under different settings

NeuralLog utilizes BERT [38] as a pre-trained language representation to understand the semantic meaning of log messages. In this RQ, we would like to evaluate the performance

TABLE V
RESULTS OF HANDLING OOV WORDS

Dataset	Metric	NeuralLog-Word2Vec	NeuralLog-NoWordPiece	NeuralLog
HDFS	Precision	0.94	0.94	0.96
	Recall	0.93	1.00	1.00
	F1-Score	0.94	0.97	**0.98**
BGL	Precision	0.94	0.93	0.98
	Recall	0.88	0.96	0.98
	F1-Score	0.91	0.96	**0.98**
Thunderbird	Precision	0.80	0.90	0.93
	Recall	0.80	0.89	1.00
	F1-Score	0.80	0.90	**0.96**
Spirit	Precision	0.94	0.93	0.98
	Recall	0.92	0.80	0.96
	F1-Score	0.93	0.86	**0.97**

of NeuralLog with different pre-trained language models. We replace the BERT model in NeuralLog with GPT2 [53] and Roberta [54], and then perform experiments to evaluate the performance of log-based anomaly detection. For GPT2 and Roberta encoders, we use their base model with 12 layers, 12 attention heads, and 768 hidden units. Table VI shows the results. We observe that these three pre-trained models can all understand the semantic meaning of log messages and achieve promising results. Overall, the performance of BERT is higher than that of GPT2 and Roberta.

TABLE VI
RESULTS OF DIFFERENT PRE-TRAINED MODELS

Dataset	Metric	BERT [38]	GPT2 [53]	RoBERTa [54]
HDFS	Precision	0.96	0.95	0.85
	Recall	1.00	1.00	1.00
	F1-Score	**0.98**	0.97	0.92
BGL	Precision	0.98	0.95	0.95
	Recall	0.98	0.99	0.90
	F1-Score	**0.98**	0.97	0.93
Thunderbird	Precision	0.93	0.85	0.78
	Recall	1.00	0.91	1.00
	F1-Score	**0.96**	0.88	0.88
Spirit	Precision	0.98	0.88	0.84
	Recall	0.96	0.95	0.90
	F1-Score	**0.97**	0.91	0.87

The number of attention heads and the feed-forward network size are two major hyperparameters of the Transformer model used in NeuralLog. To evaluate the impact of these parameters on detection accuracy, we vary their values and perform experiments on the four datasets. The resulting F1-scores are shown in Figure 11. We observe that reducing the number of attention heads and feed-forward network size can slightly hurt the performance of NeuralLog. For example, NeuralLog achieves F1-scores ranging from 0.96 to 0.98 when using 12 attention heads. These results are higher than those obtained by using only one attention head (0.88 - 0.93). Similarly, F1-scores achieved by a larger feed-forward network are usually better. Overall, the Transformer model achieves promising results with different hyperparameter values (most F1-scores are above 0.90). We observe that the performance is best when the number of attention heads is between 4 and 12, and the feed-forward network size is from 512 to 2048.
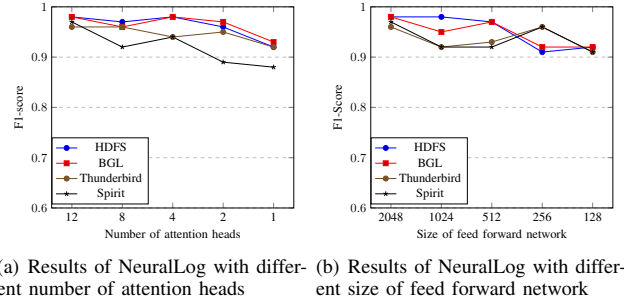

(a) Results of NeuralLog with different number of attention heads

(b) Results of NeuralLog with different size of feed forward network

Fig. 11. Results of different hyperparameter settings

VI. DISCUSSION

A. Why does NeuralLog Work?

There are three main reasons that make NeuralLog perform better than the related approaches. First, NeuralLog directly uses raw log messages instead of using a log parser in preprocessing. Since there is no loss of information from log messages, NeuralLog can precisely learn the semantic representation of log messages, compared to other approaches that depend on log parsing. Second, NeuralLog leverages BERT [38] and WordPiece [36], [37] to capture the meaning of OOV words at the subword level. Moreover, the transformer-based classification model can also improve the performance of anomaly detection. The transformer utilized by NeuralLog can learn different sequence patterns in log messages and determine which patterns are more relevant to anomalies.

Our study has demonstrated the effectiveness of Neural-Log for anomaly detection. However, NeuralLog still has limitations. Our approach is based on the learning of the semantic meanings of log messages. Given a log message, we first remove those words that contain numbers and special characters. However, in some cases, the removed words may carry important information, such as node ID, task ID, IP address, or exit code. These information could be useful for anomaly detection in certain scenarios. In our future work, we will encode more log-related information and investigate their impact on log-based anomaly detection.

B. Threats to Validity

We have identified the following major threats to validity.

Subject datasets. In this work, we use datasets collected from the distributed system (i.e., HDFS) and supercomputer (including BGL, Thunderbird, and Spirit). Although these datasets all come from real-world systems and contain millions of logs, the number of subject systems is still limited and do not cover all the domains. In the future, we will evaluate the proposed approach on more datasets collected from a wide variety of systems.

Tool comparison. In our evaluation, we compared our results with those of related approaches (i.e., SVM, LR, IM, LogRobust, and Log2Vec). We adopt the implementation of SVM, LR, and IM-based methods provided by Loglizer [55]. We adopt the implementation of LogRobust and Log2Vec

provided by their authors. We apply the default parameters and settings (e.g., sliding window size, step size, etc.) used in the previous work [28], [48], [8]. Still, the correctness of these implementations could be a threat. To reduce this threat, we make sure that the implementation of related work can produce similar results as those reported in the original papers.

Noises in labeling. Our experiments are based on four public datasets that are widely used by related work [47], [4], [12], [56], [28]. These datasets are manually inspected and labeled by engineers. Therefore, data noise (false positive/negatives) may be introduced during the manual labeling process. Although we believe the amount of noise is small (if it exists), we will investigate the data quality issue in our future work.

VII. RELATED WORK

A. Log Parsing Errors

The log parsing accuracy highly influences the performance of log mining [16]. Log parsers could produce inconsistent results depend on the preprocessing step and the set of parameters [16], [15]. The preprocessing step can further improve log parsing accuracy [16] and despite the simplicity, it still requires some additional manual work [15]. Zhu et al. [15] benchmarked 13 automated log parsers on a total of 16 datasets. They found that Drain [24] is the most accurate log parser, which attains high accuracy on 9 out of 16 datasets. The other top-ranked log parsers include IPLoM [26], AEL [25] and Spell [27]. They also found that some model parameters need to be tuned manually, and some models did not scale well with the volume of logs. He et al [16] evaluated four widely used log parsers, including SLCT [57], IPLoM [26], LKE [58] and LogSig [20].

In practice, new types of logs always appear [12], as OOV words can be added to log templates and lead to many extra log events, which will confuse the downstream tasks. Zhang et al. [8] indicated that log data is unstable, meaning that new log events often appear due to software evolution at its lifetime. Their empirical study on a Microsoft online service system shows that up to 30.3% logs are changed in the latest version. In our work, we perform an empirical study of the log parsing errors caused by the OOV problem and semantic misunderstanding, and investigate their impact on the performance of anomaly detection.

B. Log Representations

As described in Section II, most of the existing log-based anomaly detection approaches use log parsers to obtain log events and represent log messages as log events. Therefore, the existing approaches suffer from the OOV problem and the inaccurate log parsing. Recently, deep learning-based models have been adopted into log-based anomaly detection. DeepLog [5] applies Spell [27] to extract log events, then each log event is assigned with an index. Since DeepLog represents log messages as the indexes of log templates, it cannot prevent semantic information loss and could produce many wrong detection results [47], [48]. LogRobust [8] leverages Drain [24] to obtain log templates, then encodes these templates using the FastText [31] framework combined with TF-IDF [32] weight. LogAnomaly [12] applies FT-Tree [59] to parse log messages to templates, then proposes template2Vec to encode these templates based on Word2Vec [42]. SwissLog [47] obtains the semantic information of log messages after parsing log messages using a dictionary-based approach. Due to imperfect log parsing, these methods could fail to capture the semantic meaning of log messages and produce incorrect results. Log2Vec [48] transforms raw log messages into semantic vectors. As it utilizes character-level features, it could not effectively handle some domain-specific words [50], [51], [52]. Besides, Log2Vec adopts word2vec-based model that ignores the contextual information in sentences [47], thus it cannot fully understand the semantic meaning of log messages. Nedelkoski et al. [56] proposed Logsy, which is a classification-based method to learn log representations in a way to distinguish between normal data from the target system and anomaly samples from auxiliary log datasets. It does not provide mechanism for handling OOV words in log messages either.

To overcome the limitations of existing methods, we propose NeuralLog, a deep learning-based anomaly detection approach using raw log data. NeuralLog utilizes WordPiece tokenization to effectively handle OOV words that constantly appear in log messages. It also leverages BERT, a widely used pre-trained language representation, to understand the semantic meaning and capture the contextual information of raw log messages. Combined with a Transformer-based classification model, NeuralLog achieves high accuracy on anomaly detection. Furthermore, we only use log data from the target systems and do not require any auxiliary data.

VIII. CONCLUSION

Log-based anomaly detection is important for improving the availability and reliability of large-scale software systems. Our empirical study shows that existing approaches suffer from inaccurate log parsing and cannot handle OOV words well. To overcome the limitations introduced by log parsing, in this paper, we propose NeuralLog, a log-based anomaly detection approach that does not require log parsing. Our approach employs BERT encoder to capture the semantic meaning of raw log messages. To better capture contextual information from log sequences, we construct a Transformer-based classification model. We have evaluated the proposed approach using four public datasets. The experimental results show that NeuralLog is effective and efficient for log-based anomaly detection.

Our source code and experimental data are publicly available at https://github.com/vanhoanglepsa/NeuralLog.

REFERENCES

[1] E. Bauer and R. Adams, *Reliability and availability of cloud computing.* John Wiley & Sons, 2012.

[2] R. S. Kazemzadeh and H.-A. Jacobsen, "Reliable and highly available distributed publish/subscribe service," in *2009 28th IEEE International Symposium on Reliable Distributed Systems.* IEEE, 2009, pp. 41–50.

[3] J. Breier and J. Branišová, "Anomaly detection from log files using data mining techniques," in *Information Science and Applications.* Springer, 2015, pp. 449–457.

[4] B. Zhang, H. Zhang, P. Moscato, and A. Zhang, "Anomaly detection via mining numerical workflow relations from logs," in *2020 International Symposium on Reliable Distributed Systems (SRDS).* IEEE, 2020, pp. 195–204.

[5] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security,* 2017, pp. 1285–1298.

[6] M. Chen, A. X. Zheng, J. Lloyd, M. I. Jordan, and E. Brewer, "Failure diagnosis using decision trees," in *International Conference on Autonomic Computing, 2004. Proceedings.* IEEE, 2004, pp. 36–43.

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

[8] X. Zhang, Y. Xu, Q. Lin, B. Qiao, H. Zhang, Y. Dang, C. Xie, X. Yang, Q. Cheng, Z. Li *et al.*, "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,* 2019, pp. 807–817.

[9] H. Guo, S. Yuan, and X. Wu, "Logbert: Log anomaly detection via bert," *arXiv preprint arXiv:2103.04475,* 2021.

[10] P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen, "Fingerprinting the datacenter: automated classification of performance crises," in *Proceedings of the 5th European conference on Computer systems,* 2010, pp. 111–124.

[11] J.-G. Lou, Q. Fu, S. Yang, Y. Xu, and J. Li, "Mining invariants from console logs for system problem detection." in *USENIX Annual Technical Conference,* 2010, pp. 1–14.

[12] W. Meng, Y. Liu, Y. Zhu, S. Zhang, D. Pei, Y. Liu, Y. Chen, R. Zhang, S. Tao, P. Sun *et al.*, "Loganomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs." in *IJCAI,* vol. 7, 2019, pp. 4739–4745.

[13] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781,* 2013.

[14] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *arXiv preprint arXiv:1706.03762,* 2017.

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

[16] P. He, J. Zhu, S. He, J. Li, and M. R. Lyu, "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, 2016, pp. 654–661.

[17] M. Nagappan and M. A. Vouk, "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, 2010, pp. 114–117.

[18] R. Vaarandi and M. Pihelgas, "Logcluster-a data clustering and pattern mining algorithm for event logs," in *2015 11th International conference on network and service management (CNSM).* IEEE, 2015, pp. 1–7.

[19] H. Dai, H. Li, C. S. Chen, W. Shang, and T.-H. Chen, "Logram: Efficient log parsing using n-gram dictionaries," *IEEE Transactions on Software Engineering,* 2020.

[20] L. Tang, T. Li, and C.-S. Perng, "Logsig: Generating system events from raw textual logs," in *Proceedings of the 20th ACM international conference on Information and knowledge management,* 2011, pp. 785–794.

[21] H. Hamooni, B. Debnath, J. Xu, H. Zhang, G. Jiang, and A. Mueen, "Logmine: Fast pattern recognition for log analytics," in *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management,* 2016, pp. 1573–1582.

[22] K. Shima, "Length matters: Clustering system log messages using length of words," *arXiv preprint arXiv:1611.03213,* 2016.

[23] S. Thaler, V. Menkonvski, and M. Petkovic, "Towards a neural language model for signature extraction from forensic logs," in *2017 5th International Symposium on Digital Forensic and Security (ISDFS).* IEEE, 2017, pp. 1–6.

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

[25] Z. M. Jiang, A. E. Hassan, P. Flora, and G. Hamann, "Abstracting execution logs to execution events for enterprise applications (short paper)," in *2008 The Eighth International Conference on Quality Software.* IEEE, 2008, pp. 181–186.

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

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

[28] S. He, J. Zhu, P. He, and M. R. Lyu, "Experience report: System log analysis for anomaly detection," in *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE).* IEEE, 2016, pp. 207–218.

[29] D. El-Masri, F. Petrillo, Y.-G. Guéhéneuc, A. Hamou-Lhadj, and A. Bouziane, "A systematic literature review on automated log abstraction techniques," *Information and Software Technology,* vol. 122, p. 106276, 2020.

[30] Y. Liang, Y. Zhang, H. Xiong, and R. Sahoo, "Failure prediction in ibm bluegene/l event logs," in *Seventh IEEE International Conference on Data Mining (ICDM 2007).* IEEE, 2007, pp. 583–588.

[31] A. Joulin, E. Grave, P. Bojanowski, M. Douze, H. Jégou, and T. Mikolov, "Fasttext. zip: Compressing text classification models," *arXiv preprint arXiv:1612.03651,* 2016.

[32] G. Salton and C. Buckley, "Term-weighting approaches in automatic text retrieval," *Information processing & management,* vol. 24, no. 5, pp. 513–523, 1988.

[33] A. Oliner and J. Stearley, "What supercomputers say: A study of five system logs," in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07).* IEEE, 2007, pp. 575–584.

[34] S. He, J. Zhu, P. He, and M. R. Lyu, "Loghub: a large collection of system log datasets towards automated log analytics," *arXiv preprint arXiv:2008.06448,* 2020.

[35] (2021) Logpai. [Online]. Available: https://github.com/logpai/logparser

[36] M. Schuster and K. Nakajima, "Japanese and korean voice search," in *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP).* IEEE, 2012, pp. 5149–5152.

[37] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," *arXiv preprint arXiv:1609.08144,* 2016.

[38] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805,* 2018.

[39] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter," *arXiv preprint arXiv:1910.01108,* 2019.

[40] K. Clark, M.-T. Luong, Q. V. Le, and C. D. Manning, "Electra: Pre-training text encoders as discriminators rather than generators," *arXiv preprint arXiv:2003.10555,* 2020.

[41] R.-M. Karampatsis, H. Babii, R. Robbes, C. Sutton, and A. Janes, "Big code!= big vocabulary: Open-vocabulary models for source code," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE).* IEEE, 2020, pp. 1073–1085.

[42] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *International conference on machine learning.* PMLR, 2014, pp. 1188–1196.

[43] (2021) Bert pretrained models. [Online]. Available: https://github.com/google-research/bert

[44] (2021) Loghub. [Online]. Available: https://github.com/logpai/loghub

[45] K. Yao, H. Li, W. Shang, and A. E. Hassan, "A study of the performance of general compressors on log files," *Empirical Software Engineering*, vol. 25, no. 5, pp. 3043–3085, 2020.

[46] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," *arXiv preprint arXiv:1711.05101*, 2017.

[47] X. Li, P. Chen, L. Jing, Z. He, and G. Yu, "Swisslog: Robust and unified deep learning based log anomaly detection for diverse faults," in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2020, pp. 92–103.

[48] W. Meng, Y. Liu, Y. Huang, S. Zhang, F. Zaiter, B. Chen, and D. Pei, "A semantic-aware representation framework for online log analysis," in *2020 29th International Conference on Computer Communications and Networks (ICCCN)*. IEEE, 2020, pp. 1–7.

[49] Y. Pinter, R. Guthrie, and J. Eisenstein, "Mimicking word embeddings using subword rnns," *arXiv preprint arXiv:1707.06961*, 2017.

[50] S. Sasaki, J. Suzuki, and K. Inui, "Subword-based compact reconstruction of word embeddings," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, 2019, pp. 3498–3508.

[51] J. Zhao, S. Mudgal, and Y. Liang, "Generalizing word embeddings using bag of subwords," *arXiv preprint arXiv:1809.04259*, 2018.

[52] Z. Hu, T. Chen, K.-W. Chang, and Y. Sun, "Few-shot representation learning for out-of-vocabulary words," *arXiv preprint arXiv:1907.00505*, 2019.

[53] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[54] A. Conneau, K. Khandelwal, N. Goyal, V. Chaudhary, G. Wenzek, F. Guzmán, E. Grave, M. Ott, L. Zettlemoyer, and V. Stoyanov, "Unsupervised cross-lingual representation learning at scale," *arXiv preprint arXiv:1911.02116*, 2019.

[55] (2021) Loglizer. [Online]. Available: https://github.com/logpai/loglizer

[56] S. Nedelkoski, J. Bogatinovski, A. Acker, J. Cardoso, and O. Kao, "Self-attentive classification-based anomaly detection in unstructured logs," *arXiv preprint arXiv:2008.09340*, 2020.

[57] R. Vaarandi, "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, 2003, pp. 119–126.

[58] Q. Fu, J.-G. Lou, Y. Wang, and J. Li, "Execution anomaly detection in distributed systems through unstructured log analysis," in *2009 ninth IEEE international conference on data mining*. IEEE, 2009, pp. 149–158.

[59] S. Zhang, W. Meng, J. Bu, S. Yang, Y. Liu, D. Pei, J. Xu, Y. Chen, H. Dong, X. Qu *et al.*, "Syslog processing for switch failure diagnosis and prediction in datacenter networks," in *2017 IEEE/ACM 25th International Symposium on Quality of Service (IWQoS)*. IEEE, 2017, pp. 1–10.