HitAnomaly: Hierarchical Transformers for Anomaly Detection in System Log

Shaohan Huang, Yi Liu, *Member, IEEE*, Carol Fung, Rong He, Yining Zhao,
Hailong Yang, *Member, IEEE*, and Zhongzhi Luan, *Member, IEEE*

***Abstract*—Enterprise systems often produce a large volume of logs to record runtime status and events. Anomaly detection from system logs is crucial for service management and system maintenance. Most existing log-based anomaly detection methods use log event *indexes* parsed from log data to detect anomalies. Those methods cannot handle unseen log templates and lead to inaccurate anomaly detection. Some recent studies focused on the *semantics* of log templates but ignored the information of *parameter values*. Therefore, their approaches failed to address the abnormal logs caused by parameter values. In this article, we propose HitAnomaly, a log-based anomaly detection model utilizing a hierarchical transformer structure to model both log template sequences and parameter values. We designed a log sequence encoder and a parameter value encoder to obtain their representations correspondingly. We then use an attention mechanism as our final classification model. In this way, HitAnomaly is able to capture the semantic information in both log template sequence and parameter values and handle various types of anomalies. We evaluated our proposed method on three log datasets. Our experimental results demonstrate that HitAnomaly has outperformed other existing log-based anomaly detection methods. We also assess the robustness of our proposed model on unstable log data.**

***Index Terms*—Log data analysis, anomaly detection, hierarchical transformers.**

I. Introduction

Modern computer systems have become increasingly complex as systems grow in both size and functionality [1]. Anomaly detection has become an essential task to build a trustworthy computer system. A single anomaly issue can impact millions of users' experience and service [2]. Accurate and effective anomaly detection model can reduce the damage from anomalies [3], which is crucial for service management and system maintenance. Logs are widely used to record significant events and system status in an operating system or other software systems. Since system logs contain noteworthy events and runtime status, they are one of the most important data sources for anomaly detection and system monitoring [4].

Existing log-based anomaly detection methods can be broadly classified into two categories: log event indexes based approaches (e.g., PCA [5], Invariant Mining [6], SVM [7], DeepLog [8]) and log template semantics-based approaches (e.g., LogAnomaly [9], LogRobust [10]). Log event indexes based approaches first extract log events from log messages and then convert log events into indexes feature spaces (e.g., using a one-hot vector to represent a log event). These methods do not attempt to utilize semantic information in log messages. Thus, they cannot handle unseen log templates and suffer from inaccurate log parsing. Log template semantics-based approaches model a log stream as a natural language sequence. They convert log templates into vectors using word vectors and then train their model based on those vectors. Existing approaches are limited to the semantics of log templates but may miss the key values that can be used for anomaly detection. We observed that some anomalies are not shown as a deviation from a normal log template sequence but as an abnormal parameter value. For example, "*Took 10 seconds to build instance*" and "*Took 600 seconds to build instance*" share the same template but with different parameter values. Taking too much time to build an instance may result in exceptions or anomalies. Therefore, the values of some specific parameters can be essential factors to be considered in log-based anomaly detection models.

DeepLog [8] is a stacked long short-term memory (LSTM) network [11] which is used to encode log templates and then use another LSTM to predict the next log. Recent studies [12] show that the transformer model performs better than LSTM in many tasks. Therefore, we adopt a hierarchical transformer [12] rather than a hierarchical LSTM. The Transformer is a deep machine learning model introduced in 2017, used primarily in the field of natural language processing [12]. Like the LSTM network, the transformer structure also handles ordered sequences of data. Transformers have become the basic building block of most state-of-the-art architectures in NLP, replacing gated recurrent neural network models such as LSTM in many cases.

Manuscript received April 28, 2020; revised September 19, 2020; accepted October 19, 2020. Date of publication October 29, 2020; date of current version December 9, 2020. This work is supported by National Key Research & Development Program of China (Grant No. 2018YFB0204003), which belongs to the High Performance Computing Key Project. This research is also supported by National Natural Science Foundation of China (Grant No. 62072018). The associate editor coordinating the review of this article and approving it for publication was T. Samak. *(Corresponding author: Shaohan Huang.)*

Shaohan Huang, Yi Liu, Hailong Yang, and Zhongzhi Luan are with the School of Computer Science and Engineering, Sino-German Joint Software Institute, Beihang University, Beijing 100191, China (e-mail: huangshaohan@buaa.edu.cn; yi.liu@buaa.edu.cn; luan.zhongzhi@buaa.edu.cn).

Carol Fung is with the Computer Science Department, Virginia Commonwealth University, Richmond, VA 23284 USA.

Rong He and Yining Zhao are with the Chinese Academy of Scientific Computer Network Information Center, Beijing 100864, China.

Digital Object Identifier 10.1109/TNSM.2020.3034647

In this article, we propose HitAnomaly, a log-based anomaly detection model utilizing a hierarchical transformer structure to model log template sequences and parameter values. In the HitAnomaly model, we design a log sequence encoder and a parameter value encoder to obtain their representations correspondingly. The log sequence encoder transforms a template sequence into a fixed-dimension vector as log sequence representation. The parameter value representation is generated by parameter value encoder, which can capture the semantic information in parameters and help the model to detect various types of performance anomalies. We then utilize an attention mechanism to build its classification. Since log template sequence and parameter values have different impacts on the classification result, our proposed model is able to learn different weights to these two representations by the attention mechanism.

We evaluated our proposed method on a total of three log datasets including HDFS [5], the BGL dataset [13], and the OpenStack dataset [8]. Experimental results show that HitAnomaly outperforms other existing log-based anomaly detection methods on stable log data sets. We also investigate the impacts of window size and number of layer size. In order to evaluate the robustness of our model, we also present the performance of HitAnomaly on unstable log data from two aspects: unseen log events and unstable log sequences.

The key contributions of this article can be summarized as follows:

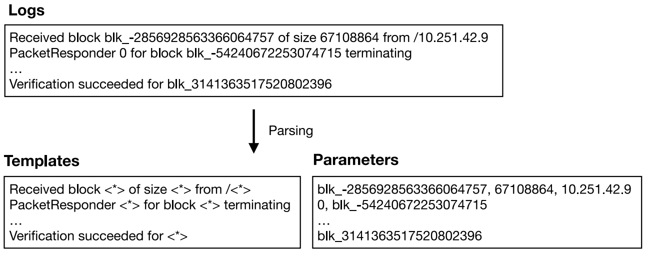
- We propose HitAnomaly, a hierarchical transformer model for log-based anomaly detection. To the best of our knowledge, our work is the first to utilize the transformer model into a log-based anomaly detection task.
- This study provides new insights to capture the semantic information on both log templates and parameter values for log-based anomaly detection.
- We design an attention mechanism to merge the semantic information log sequences and parameter values to detect log anomalies.

The rest of this article is organized as follows. We introduce the background of our work in Section II. The architecture design and our model are described in Section III. Section IV describes the experimental settings, and we evaluate the performance of HitAnomaly on stable log data and unstable log data. Related work is introduced in Section V. Finally, in Section VI, we conclude our work and state possible future work.

II. Background

A. Log Parser

Enterprise systems often produce a large volume of logs to record runtime status and events. Typically, unstructured and free-text logs have to be properly parsed before they are used in anomaly detection [7]. The goal of log parsers is to transfer raw messages content into structured log templates associated with key parameters. As shown in Figure 1, unstructured log messages come from the HDFS dataset. The first log message is "Received block blk_-2856928563366064757 of size 67108864 from /10.251.42.9". It is parsed into log template "Received block ⟨*⟩ of size ⟨*⟩ from /⟨*⟩" and parameter values ['blk_-2856928563366064757', '67108864', '10.251.42.9']. Here, '⟨*⟩' is a wildcard to match parameters. The log template is comprised of fixed text strings, and parameters record some system attributes and variables. Figure 1 shows that a log sequence is converted into a sequence of log templates and corresponding parameters. HitAnomaly model takes log template sequences and parameter sequences as input.



```
Received block blk_-2856928563366064757 of size 67108864 from /10.251.42.9
PacketResponder 0 for block blk_-54240672253074715 terminating
...
Verification succeeded for blk_3141363517520802396
```

```
Received block <*> of size <*> from /<*>
PacketResponder <*> for block <*> terminating
...
Verification succeeded for <*>
```

```
blk_-2856928563366064757, 67108864, 10.251.42.9
0, blk_-54240672253074715
...
blk_3141363517520802396
```

Fig. 1. Examples of log parsing.

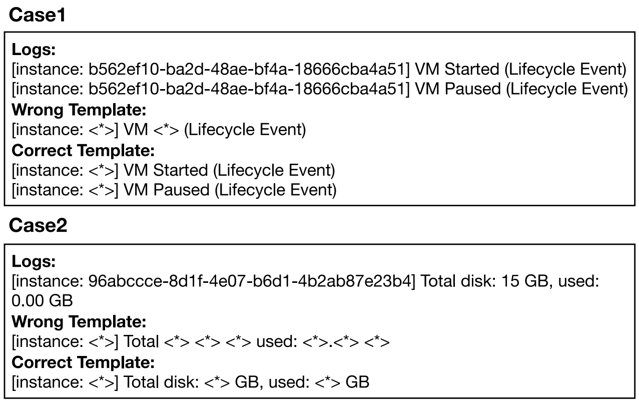
There have been many studies on log parsing [14]–[18]. Therefore, log parsing is not the focus of this article. Considering accuracy and speed, we adopt Drain [18] method to parse all log data. Drain applies a fixed-depth tree structure to parse log messages and extracts common templates. It achieves high performance compared to many other log parser methods. Thus, we choose this method to conduct log parsing for our project. However, Drain may introduce some noise because that method may occasionally produce inaccurate results. We will describe this issue in the following subsections.

B. Limitation of Previous Methods

In this section, we first describe the challenges in log-based anomaly detection and then introduce the limitation of the existing methods.

The first challenge is the noise from the pre-processing and log parsing. Inaccurate log parsing may cause false alarms, which is not avoidable even Drain provides high performance compared to other methods. For example, this method could achieve an accuracy of 96.3% in the BGL dataset. As claimed in [17], log mining can be sensitive to some critical events. 4% parsing errors on critical events can cause an order of magnitude performance degradation in log mining.

Figure 2 shows two examples of log parsing errors in the OpenStack data set, which are parsed by Drain. The first case is that log parser may mix up different log events. These two logs belong to two different log templates. However, Drain converts them into one template and misidentifies keywords "Started" and "Paused" as parameters. If only log templates are used to detect anomalies, the model cannot identify the status of the system in the first case. The second case shows that log parser omits some keywords, e.g., disk, used, and GB, compared to the ground truth log template. This parsing error may cause some potentially relevant information to be lost, which is not conducive to anomaly detection.



Case1

Logs:

```
[instance: b562ef10-ba2d-48ae-bf4a-18666cba4a51] VM Started (Lifecycle Event)
[instance: b562ef10-ba2d-48ae-bf4a-18666cba4a51] VM Paused (Lifecycle Event)
```

Wrong Template:

```
[instance: <*>] VM <*> (Lifecycle Event)
```

Correct Template:

```
[instance: <*>] VM Started (Lifecycle Event)
[instance: <*>] VM Paused (Lifecycle Event)
```

Case2

Logs:

```
[instance: 96abccce-8d1f-4e07-b6d1-4b2ab87e23b4] Total disk: 15 GB, used: 0.00 GB
```

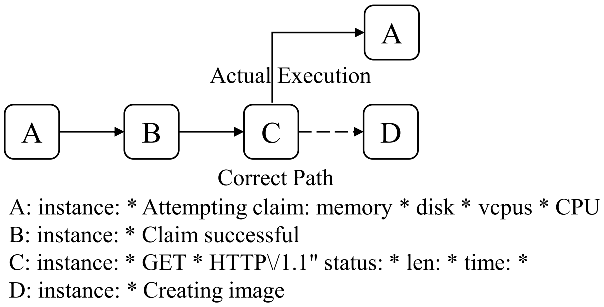
Wrong Template:

```
[instance: <*>] Total <*> <*> <*> used: <*>.<*> <*>
```

Correct Template:

```
[instance: <*>] Total disk: <*> GB, used: <*> GB
```

Fig. 2. Examples of log parsing errors.

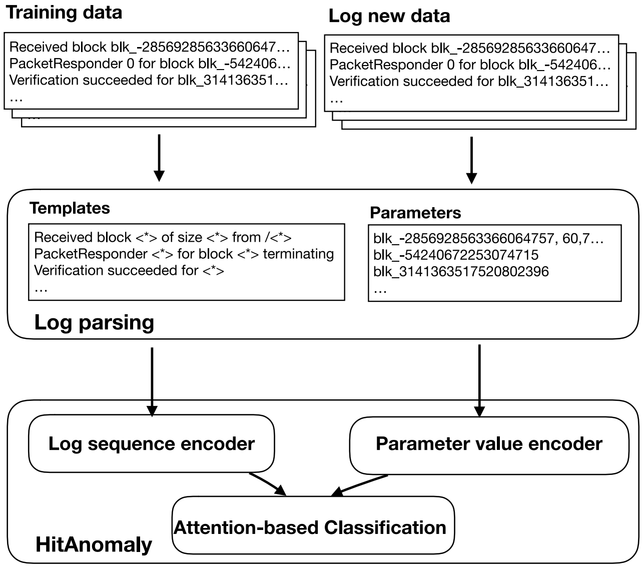


A: instance: * Attempting claim: memory * disk * vcpus * CPU
B: instance: * Claim successful
C: instance: * GET * HTTP\/1.1" status: * len: * time: *
D: instance: * Creating image

Fig. 3. Examples of template-related anomaly.

The second challenge is that log-based models need to analyze multiple factors to detect anomalies. For example, the order of log messages provides essential information for anomaly detection [8]. To some degree, the order of the log message represents the execution path of a program. A wrong execution order might mean that an anomaly has happened. As shown in Figure 3, the history template sequence is [A, B, C] and the expected event is D, which means the instance starts to create image. However, the actual log template is A, which indicates that the instance is trying to claim again. The model can capture the information of template sequence [A, B, C, A] to detect this anomaly. Secondly, the values of certain parameters are important factors to be considered in log-based anomaly detection models. As shown by the example in Section I, these metric values reflect the underlying system state and performance status.

Log event indexes based methods first extract log templates from log sequences and transform templates into log count vectors or template indexes. They build unsupervised (e.g., PCA [5], Invariant Mining [6], DeepLog [8]) or supervised (e.g., SVM [7]) machine learning models to detect anomalies. The limitation of these methods is obvious. First, template indexes ignore the context information in the log sequences and cannot provide any semantic information for the anomaly detection model. For example, they use two event IDs to represent "[instance: ⟨*⟩] VM Started (Lifecycle Event)" and "[instance: ⟨*⟩] VM Paused (Lifecycle Event)". Thus, the model cannot identify if a VM is started or paused. Secondly, the system can generate new log templates and these approaches cannot address this problem. The dimension of the log count vector must be changed and the model needs to be retrained accordingly.



```
Received block blk_-28569285633660647…
PacketResponder 0 for block blk_-542406…
Verification succeeded for blk_314136351…
…
```

```
Received block <*> of size <*> from /<*>
PacketResponder <*> for block <*> terminating
Verification succeeded for <*>
…
```

```
blk_-2856928563366064757, 60,7…
blk_-542406722530747l5
blk_3141363517520802396
…
```

Fig. 4. Overview of hitanomaly.

Log template semantics based approaches model a log sequence as a natural language sequence (e.g., LogAnomaly [9], LogRobust [10]). They utilize word embedding to transform log templates into vectors and then train their classification model based template vectors. Compared to log event indexes based methods, these methods are able to capture the semantic information in log templates and deal with new log templates, to some degree. However, such approaches only utilize log template sequences and ignore the semantic information of parameter values. As shown in log parsing error case 1, if only using log template to detect an anomaly, the model cannot identify the status of the system. We observe that some anomalies are not shown as a deviation from a normal log template sequence but as an irregular parameter value. Thus, parameter values are essential factors to be considered in a log-based anomaly detection model.

III. Design of HitAnomaly

A. Overview

To accurately detect anomalies, caused by log sequences or irregular parameter values, we propose HitAnomaly, a novel log-based anomaly detection model utilizing a hierarchical transformer structure. The overview of HitAnomaly is shown in Figure 4 with three main components: log sequence encoder (Section III-B), parameter values encoder (Section III-C), and attention-based classification (Section III-D).

The first step is log parsing (as introduced in Section II-A), which transfers raw messages into structured log templates associated with key parameters. Some of the existing methods [5], [6], [8] convert templates into indexes and use them as input. Some works [9], [10] only rely on log templates to detect anomalies. Instead, HitAnomaly takes both log templates and parameter values as input. Given a log template sequence, a log sequence encoder transforms it into a fixed-dimension vector (We call it *log sequence representation*). Parameter value encoder convert parameter values into *parameter value representation*. After log sequence encoding and parameter value encoding, HitAnomaly leverages the attention-based structure to classify anomalous logs, which can learn to assign various degrees of importance to log sequences or parameters.

A log sequence encoder has a hierarchical transformer structure: *single log encoder* to transform each log event into a vector using transformer blocks and *log sequence encoder* to leverage a transformer structure in order to learn a sequence of log templates representations. Log sequence encoder can capture the semantic information from log templates and also handle the contextual knowledge of the log sequence. Parameter value encoder takes parameter values as input and applies transformer block to encode parameters into vectors. Parameter value encoder can reduce noises caused by inaccurate log parser and also can help to recognize parameter-aware anomalies.

B. Log Sequence Encoding

In this section, we first describe how to transform a single log into a fixed-dimension vector (we call it *log representation*) using transformer blocks in HitAnomaly. We then introduce the log sequence encoder, which leverages a transformer structure to obtain log sequence representation.

In recent years, the transformer structure has been proved to be highly effective across various NLP tasks [12], [19]. Compared to an LSTM recurrent neural network model, a transformer structure can efficiently handle large inputs and outputs, which is an advantage when dealing with complex log patterns and handling noisy and unstable log events. The original transformer model, proposed in [12], has an encoding component and a decoding component. The decoding component aims at generating an output sequence. In this work, HitAnomaly utilizes a transformer encoding structure to encode log templates and parameter values for anomaly detection.

We first introduce a single log encoder, which transforms a log template S into a fixed-dimension vector $R_l \in \mathbb{R}^d$ using transformer blocks. As shown in Figure 5, a single log encoder contains two transformer blocks (transformer layer size = 2) and a pooling layer. Each transformer block is broken down into two sub-layers: self-attention layer and feed-forward network layer. The log encoder involves four steps: (1) Word2Vector; (2) Self-Attention; (3) Feed Forward Neural Network(FFN); (4) Pooling Layer.

1) Word2Vector: The log encoder first maps each word in log template to a vector *x*. The words in log template are passed in as one-hot encoded vectors. Then we look up a representation vector in a weight matrix (We call it embedding matrix) using the one-hot vector. As shown in Figure 6, we use a vector [0, 0, 0, 1, 0] to represent the word ‘terminating’. Then we use the one-hot vector to look up its corresponding word vector in the embedding matrix. Finally, we obtain the word vector of ‘terminating’ as [16, 2, 11]. LogRobust [10] leverages off-the-shelf word vectors, which were pre-trained on Common Crawl Corpus dataset using the Fast-Text [20] algorithm. We initialize our word vectors by Bert [19], which is a pre-trained bidirectional transformer language model. We use the same WordPiece [21] tokenization method as Bert. WordPiece is a text tokenizer, which segments words into sub-words, e.g., IP address ‘10.251.122.79:50010’ is tokenized into [‘10’, ‘.’, ‘251’, ‘.’, ‘122’, ‘.’, ‘79’, ‘:’, ‘500’, ‘##10’].

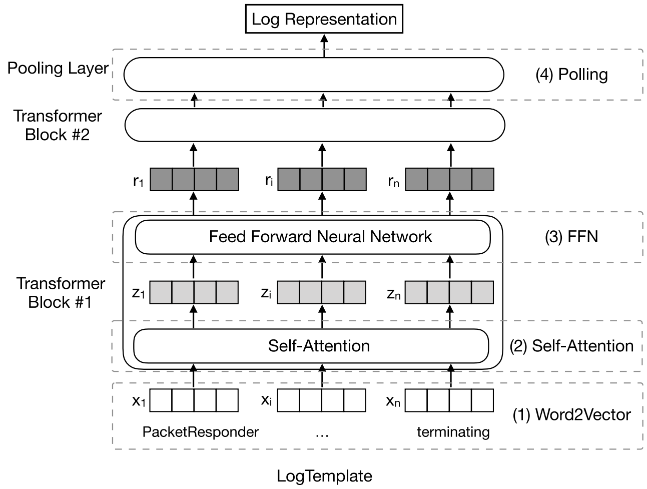


Fig. 5. A detailed view of single log encoder using transformer block.

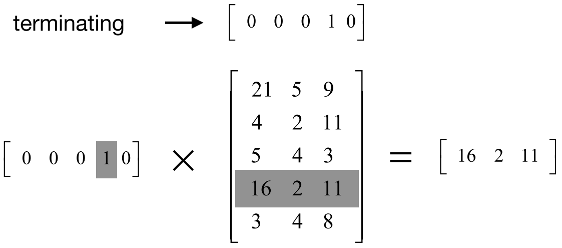


Fig. 6. An example of mapping a word to a vector.

After replacing words with corresponding vectors, a log template sentence S is transformed into a vector list $X = [x_1, x_2, \dots, x_n]$, where *n* is the length of log template S and $x_i \in \mathbb{R}^{1\times d}$, where *d* is the dimension of the word vector.

2) Self-Attention: The vector list *X* flows through the self-attention [12] layer. To calculate self-attention, we first create three vectors (query vector *Q*, key vector *K*, value vector *V*) for each vector *x*. These vectors are computed by multiplying *x* by three matrices as follows:

$$Q_i = x_i \mathbf{W}_Q \tag{1}$$

$$K_i = x_i \mathbf{W}_K \tag{2}$$

$$V_i = x_i \mathbf{W}_V \tag{3}$$

where $\mathbf{W}_Q \in \mathbb{R}^{d\times d_q}$, $\mathbf{W}_K \in \mathbb{R}^{d\times d_k}$, and $\mathbf{W}_V \in \mathbb{R}^{d\times d_v}$ are parameter matrices and d_q, d_k, d_v are the dimensions of query, key, value vectors.

We use z_i to represent the self-attention layer output x_i at position *i*. We compute the dot products of the query Q_i and all keys, and then divide the product by $\sqrt{d_k}$. After that we apply a softmax function to obtain the weights on the values as follows:

$$z_i = softmax\left(\frac{Q_i\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V} \tag{4}$$

Softmax function normalizes the scores ($\frac{Q_i\mathbf{K}^T}{\sqrt{d_k}}$) so they're all positive and add up to 1. It is defined as:

$$Softmax(x_i) = \frac{exp(x_i)}{\sum_j exp\left(x_j\right)} \tag{5}$$

In practice, a set of queries are computed by the attention function simultaneously, and then packed together into a matrix $\mathbf{Q}$. The keys and values are also packed together into matrices $\mathbf{K}$ and $\mathbf{V}$.

It is worth noting that the transformer model does not apply to a single attention function with keys, values, and queries. Instead, it uses a mechanism called "multi-headed" attention. We have not only one, but multiple sets of query/key/value weight matrices. We call the output computed by a set of query/key/value as an attention head. Our transformer uses twelve attention heads, so we end up with twelve attention heads. We concatenate those attention heads then multiply them by an additional weights matrix W_O. Multi-headed attention is computed as follows:

$$z_i = concat(head_1, \ldots, head_h)\mathbf{W}_O \tag{6}$$

$$head_h = softmax\left(\frac{Q_{i,h}\mathbf{K}_h^T}{\sqrt{d_k}}\right)\mathbf{V}_h \tag{7}$$

where $Q_{i,h} = x_i\mathbf{W}_{Q,h}$, $\mathbf{W}_{Q,h}$ is a parameter matrix at $head_h$ attention. $\mathbf{K}_h$ and $\mathbf{V}_h$ are key matrix and value matrix at $head_h$ attention. $\mathbf{W}_O \in \mathbb{R}^{hd_v \times d}$ is a trainable parameter for concatenation operation. Multi-headed attention expands the model's ability to focus on different positions and learn diverse representations.

3) Feed-Forward Network: Afterward, the outputs of the self-attention layer are fed to the feed-forward network. The feed-forward network (FFN) is a fully connected feed-forward network, which is applied to each position separately and identically. This consists of two linear transformation functions with a ReLU activation in between.

$$r_i = max(0, z_i\mathbf{W_1} + b_1)\mathbf{W_2} + b_2 \tag{8}$$

where $\mathbf{W}_1 \in \mathbb{R}^{d \times d_{ff}}$, $\mathbf{W}_2 \in \mathbb{R}^{d_{ff} \times d}$, $b_1 \in \mathbb{R}^{d_{ff}}$, and $b_2 \in \mathbb{R}^d$ are trainable parameters in feed-forward network layer and d_{ff} is the dimension of feed-forward network.

4) Pooling Layer: As shown in Figure 5, the second transformer block has the same structure as the first transformer block. The output of the first transformer block as input flows through the second transformer block. After the second transformer block, the outputs are fed into an average pooling layer, which reduces the number of weights and transforms a template into a fixed-dimension vector. R_l is a log representation generated by the single log encoder.

$$R_l = \frac{1}{n}\sum_{i=1}^{n} r_i \tag{9}$$

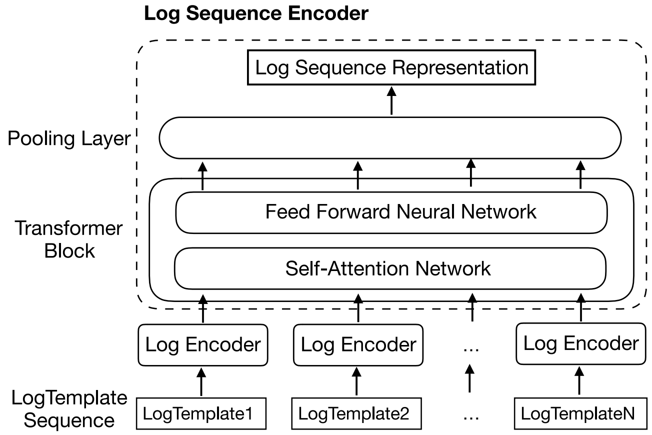


Fig. 7. Log sequence encoder using hierarchical transformers.

where $r_i \in \mathbb{R}^d$ represents the output vector of the second transformer block at position *i*.

In analogy to the single log encoder, as shown in Figure 7, the log sequence encoder is yet another transformer structure but applies on the sequence level. The log sequence encoder consists of a transformer layer and a pooling layer.

Assume that a log template sequence contains N log templates. After running log encoding, we obtain a log representations list $R_L = [R_{L1}, R_{L2}, \ldots, R_{LN}]$, where R_{LN} is the N-th log representation. After running the transformer layer on a sequence of log template representations R_L, we obtain the context sensitive log template representations $Z_L = [z_{L1}, z_{L2}, \ldots, z_{LN}]$.

$$Z_L = \texttt{TransformerBlock}\ (R_L) \tag{10}$$

The log sequence encoder also applies an average layer to transform log template representations Z_L into final log sequence representation R_s. Averaging the output layer is one of the most commonly used approach to encode a sequence [22].

$$R_s = \frac{1}{N}\sum_{i=1}^{N} Z_i \tag{11}$$

The hierarchical nature of our model reflects the intuition that a log template consists of log messages, and a log message is generated from words. We, therefore, employ a representation framework that reflects the same architecture, with global information being discovered and local information being preserved.

C. Parameter Value Encoding

In this section, we describe how to obtain the parameter value representation in HitAnomaly. The log template sequence is essential for detecting execution path anomalies. However, some anomalies are not shown as a deviation from a normal template sequence, but as an irregular parameter value [8]. DeepLog proposed a parameter value detection model, which views each parameter value vector sequence (for the same log key) as a separate time series. DeepLog only handles the numeral value and uses the average and

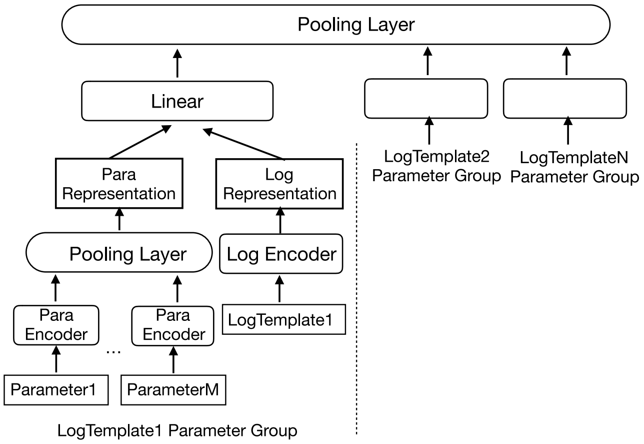


Fig. 8. Parameter value encoder using hierarchical transformers.

the standard deviation of all values as parameter value vectors. HitAnomaly utilizes a transformer network to capture the semantic information in parameter values.

Similar to log sequences, parameter value sequences also have a hierarchical structure. A log sequence consists of a series of log messages, and each log message has a corresponding parameter value. Parameter value encoder also leverages a hierarchical transformer structure to convert parameter values into a fixed-dimension vector (we call it parameter value representation).

If the parameter value encoder only takes parameter value sequences but ignores the interactions between templates and parameter values, it will not be able to model correlation that exists between the templates and parameter values. For example, in the log *"Took 600 seconds to build instance"*, the parameter value *'600'* means taking too much time to build an instance. If the log changes to *"Took 600 ms to build instance"*, it is then a legal event. To tackle this problem, we integrate the information of the log template into parameter value encoding.

As shown in Figure 8, we use a hierarchical transformer structure to model parameter values. We first merge the parameter values with the same log template. Assume that we have N log template parameter groups. Each group shares a log template and contains a series of parameters. For example in Figure 8, the log template parameter group 1 contains a log template and M parameter values. We use a similar transformer encoder as shown in Figure 5 as *Para Encoder* to model each parameter. After parameter encoding, we obtain a parameter value representation list $[r_{v1}, r_{v2}, \ldots, r_{vM}]$. Then we employ an average pooling layer to transform the representation list into a vector $R_{v1} \in \mathbb{R}^{1\times d}$.

$$R_{v1} = \frac{1}{M}\sum_{i=1}^{M} r_{vi} \tag{12}$$

We use R_{L1} to represent the template vector, which is generated by a log encoder. We design a linear layer to model the interactions between templates and parameter values. The final representation of group one is computed as follows:

$$R_{V1} = R_{L1}\mathbf{W_1} + R_{v1}\mathbf{W_2} + b \tag{13}$$

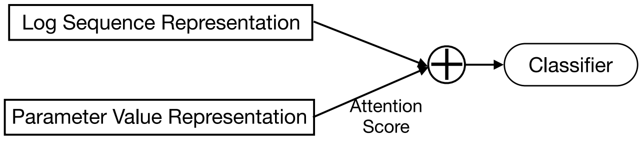


Fig. 9. Attention-based classification.

$\mathbf{W_1} \in \mathbb{R}^{d\times d}$, $\mathbf{W_2} \in \mathbb{R}^{d\times d}$, and $b \in \mathbb{R}^{1\times d}$ are trainable parameters and we update them during training.

After transformer block, parameter values encoder also applies an average pooling layer to transform parameter representations list $[R_{V1}, R_{V2}, \ldots, R_{VN}]$ into final parameter sequence representation R_p.

Since parameter values in a log message often record important system state metrics, parameter values encoder is able to capture the semantic information in parameters and help the model to detect various types of performance anomalies.

D. Attention-Based Classification

Through log sequence encoding and parameter value encoding, a log sequence is transformed into two vectors log sequence representation R_s and parameter sequence representation R_p. Considering R_s and R_p have different impacts on the classification result, we leverage the attention mechanism to build the classification model, which pre-computes different weights (denoted as α) to these two representations.

As shown in Figure 9, the final vector is computed by the weighted sum of log sequence representation R_s and parameter sequence representation R_p. For example, if the anomaly happens in parameter values, parameter representation R_p has larger weights than R_s. The attention scores α are computed as follows:

$$\alpha_s = \frac{exp(\mathtt{f}(R_s))}{exp(\mathtt{f}(R_s)) + exp\big(\mathtt{f}\big(R_p\big)\big)} \tag{14}$$

$$\alpha_p = \frac{exp\big(\mathtt{f}\big(R_p\big)\big)}{exp(\mathtt{f}(R_s)) + exp\big(\mathtt{f}\big(R_p\big)\big)} \tag{15}$$

The computation of $\mathtt{f}(R_s)$ and $\mathtt{f}(R_p)$ are shown as follows:

$$\mathtt{f}(R_s) = v^{\mathsf{T}}\mathtt{tanh}(R_s\mathbf{W}_\alpha) \tag{16}$$

$$\mathtt{f}\big(R_p\big) = v^{\mathsf{T}}\mathtt{tanh}\big(R_p\mathbf{W}_\alpha\big) \tag{17}$$

where $\mathbf{W}_\alpha$ and *v* are both parameters of the attention layer. A softmax layer is introduced to compute probability distribution of anomaly:

$$y = \mathtt{softmax}\big(\alpha_s R_s + \alpha_p R_p\big) \tag{18}$$

The model is trained using the Stochastic Gradient Descent (SGD) [23] algorithm and we use the cross-entropy as the loss function.

IV. Experiment

In this section, we first describe the experimental dataset and evaluation metrics. We then show the performance of HitAnomaly on stable log data. Lastly, we present the evaluation of our proposed model on unstable log data.

TABLE I
STATISTICS OF LOG DATA

Datasets	Duration	# of logs	# of anomalies
HDFS	38.7 hours	11,175,629	16,838(blocks)
BGL	214.7 days	4,747,963	348,469(logs)
Openstack	-	70,746	503(instances)

TABLE II
STATISTICS OF LOG TEMPLATES

Datasets	Accuracy	# templates	# templates in training data
HDFS	0.998	48	46
BGL	0.963	1,848	971
Openstack	0.733	4,248	3,400

A. Experiment Setting

1) Datasets: We conduct our experiments on three datasets: the HDFS dataset [5], the BGL dataset [13] and the OpenStack dataset [8]. We list the summary statistics for the log datasets in Table I. The detailed information on the three datasets are described as follows:

(1) HDFS: The HDFS dataset consists of 11,175,629 logs, which is generated through running Hadoop-based jobs on more than 200 Amazon's EC2 nodes. HDFS dataset is manually labeled through handcrafted rules to identify the anomalies. HDFS logs record a unique block ID for each block operation. There are 575,061 blocks of logs in the dataset, among which 16,838 blocks were labeled as anomalous.

(2) BGL: BGL is an open dataset of logs collected from a BlueGene/L supercomputer system at Lawrence Livermore National Labs. The BGL dataset contains 4,747,963 logs and 348,460 logs were labeled as anomalous. The system administrators determined the subset of log entries that they would tag as being alerts. Unlike HDFS data, BGL logs have no identifier recorded for each job execution. Thus, we have to use fixed windows to slice logs as log sequences.

(3) OpenStack: OpenStack is a cloud operating system that controls large pools of computer, storage, and networking resources throughout a datacenter. This data set was generated on CloudLab, which contains 1,335,318 INFO level logs. Three types of anomalies were injected at different execution points. OpenStack data is grouped into different sessions by instance ID. There are 503 instances labeled as anomalous.

In the following experiments, we leverage the front 80% (according to the timestamps of logs) as the training data, and the remaining 20% as the testing data. Moreover, because the above datasets were manually labeled, we take these labels as the ground truth for evaluation.

2) Baselines: We compare HitAnomaly with three unsupervised baseline methods and three supervised methods. These methods are briefly described as follows:

- PCA [5]: This work first converts logs into count vectors and then uses Principal Component Analysis (PCA) algorithm to divide them into normal space and anomalous space.
- Invariant Mining (IM) [6]: This article proposed an Invariant Mining (IM) method to mine invariants from log count vectors, then invariants were labeled as anomaly logs.
- LogCluster [4]: LogCluster proposed an approach that clusters the logs for a easier log-based problem identification and utilizes a knowledge base to reduce redundancy by previously examined log sequences.
- SVM [7]: This work represents log sequences as count vectors and uses a Support Vector Machine (SVM) as its classification algorithm.
- LR [24]: It's similar to the SVM [7] method. It converts log sequences to vectors and then applies Logistic Regression (LR) as its supervised learning algorithm.
- DeepLog [8]: DeepLog is a deep neural network model using LSTM to model a system log as a natural language sequence.
- LogRobust [10]: LogRobust extracts semantic information of log events and represents them as semantic vectors. It utilizes an attention-based Bi-LSTM model to detect anomalous log sequences.

For these baseline methods we explore their parameter space and report their best results.

3) Implementation: We implement LogRobust and HitAnomaly based on Pytorch [25] on the Linux server with NVIDIA Tesla P40 GPU. We set the window size of the log sequence as 15. We use the following parameters to train our HitAnomaly model. We set two transformer layers, hidden dimension size as 128, and the initial learning rate as $5e^{-5}$. For window size and number of layers, we investigate their impacts in our experiment.

Table II presents the accuracy of log parser Drain and the number of log templates in our data sets. The accuracy of Drain is reported in [26], which is evaluated in 2000 log messages. We can observe that the HDFS and BGL datasets are accurately (over 95%) parsed by Drain. However, OpenStack still could not be parsed accurately. Zhu *et al.* [26] explained that the complex structure and abundant event templates caused the inaccuracy of log parsing. As shown in Table II, we list the statistics of log templates in our data set and in training data. The HDFS dataset includes 48 templates and of these 46 templates are used in training data. The BGL dataset has 1,848 templates, and its training data takes about 50% of these templates. The OpenStack dataset has quite a large number of event templates where 3,400 templates are used in training data.

LogRobust [10] has found that real-world log data is *unstable*. For example, developers may frequently modify source code including logging statements, which leads to changes to log data and means that unseen log events or new but similar log sequences often appear. Through Table II, we observe that most of templates can be found the training dataset of HDFS and OpenStack datasets. However, only half of templates appear in the training data for BGL. In this work, HDFS and OpenStack original data are regarded as *stable* log datasets. The BGL data is used as *unstable* log dataset.

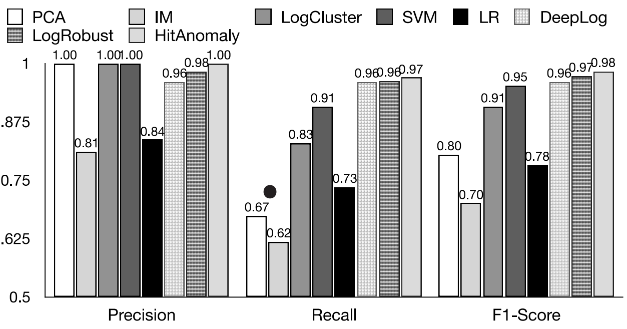


Fig. 10. Evaluation on HDFS dataset.

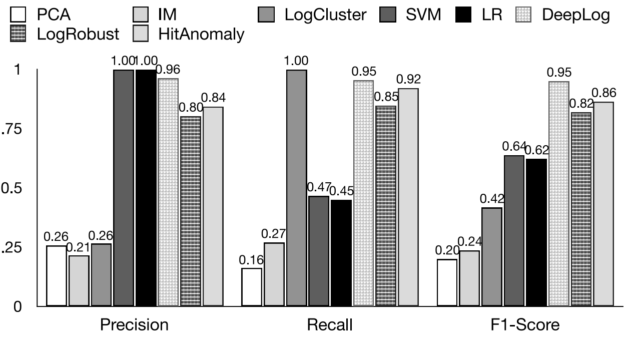


Fig. 11. Evaluation on OpenStack dataset.

4) Evaluation Metrics: Anomaly detection is a binary classification task. A classification method is usually evaluated by Precision, Recall, and F1-Score. We compute these metrics as follows:

$$\texttt{Precision} = \frac{TP}{TP + FP} \tag{19}$$

Precision shows the percentage of true anomalies among all anomalies detected by the model.

$$\texttt{Recall} = \frac{TP}{TP + FN} \tag{20}$$

Recall means the percentage of anomalies in the data being detected.

$$\texttt{F1-Score} = \frac{2 \cdot \texttt{Precision} \cdot \texttt{Recall}}{\texttt{Precision} + \texttt{Recall}} \tag{21}$$

F1-Score is the harmonic average of Precision and Recall. TP (true positive) is the number of anomalies that are correctly detected by the model. FP (false positive) is the number of normal log sequences that are labeled as anomaly by the model. FN (false negative) stands for the number of abnormal log sequences that are not detected by the model.

B. Experiment on the Stable Log Data

In this part, we first study the performance of HitAnomaly on the three log data sets and then investigate the impacts of window size and number of layer size.

1) Evaluation on HDFS Dataset: Figure 10 shows the performance of HitAnomaly compared to six baseline methods over the HDFS data set. HitAnomaly achieves the highest accuracy among the eight methods, having an F1 score of 0.983. LogRobust and HitAnomaly can detect anomalies with a more than 97% F1-score, which demonstrates that anomaly detection models benefit from the semantic information of log templates. LogRobust generates more false alarms than HitAnomaly because they did not utilize the parameter value information contained in the dataset. PCA and LogCluster methods show high precision on this data set but the recall rate is low (less than 70% for PCA and less than 85% for LogCluster). This is because these two methods miss a large percentage of actual anomalies which may result in severe consequences for service management.

We also observe that 5 out of 8 detection methods have achieved good accuracy (over 90%) on the HDFS data set, which means anomaly detection is a relatively easy task over the HDFS dataset. It is because the log parser on the HDFS dataset is accurate and most templates have been used in training data. The results demonstrate that log parser plays an essential role in log-based anomaly detection. DeepRobust randomly collects 6,000 normal log sequences and 6,000 anomalous log sequences as training set. The recall, precision and F1-score achieved by LogRobust are 1.00, 0.98 and 0.99, respectively. We follow this preprocess method to build balanced traning data. Our model achieved precision of 99.1%, recall of 98.5%, and F1 of 98.7%, which is comparable to LogRobust model and better than other baselines. This experiment means that splitting based on timestamp is more challenging than balanced training data.

2) Evaluation on OpenStack Dataset: Figure 11 shows the performance of HitAnomaly over OpenStack data set. DeepLog achieves the best accuracy among those methods and HitAnomaly has an F1 score of 0.862 as second. The OpenStack data set has quite a large number of log templates, which makes the counting matrix very sparse. Thus, log event indexes based approaches have poor performance (F1-score less than 65%) over OpenStack data. PCA and IM approaches have both poor precision and recall. In contrast, even though LogCluster has achieved a perfect recall in this case, it has an inferior precision rate of only 26% (many VM instances are detected as abnormal executions). It is because the distance between cluster pairs is too large in a sparse matrix, and many subsets cannot be merged into normal clusters [27]. This leads the remaining cluster pair to be detected as abnormal. LogRobust and HitAnomaly methods utilize word vectors to convert log templates into fixed-dimension vectors where semantic information between observed templates and unseen templates can be shared.

To further evaluate the impact of log parser on log anomaly detection tasks, we use LFA [28] to preprocess the OpenStack dataset. The accuracy of LFA over the OpenStack dataset is only 0.2. Figure 12 shows the F1 score in comparison with Drain and LFA log parser. We found that log parsers have a significant impact on the log anomaly detection, especially for log event indexes based approaches. The performance of log event indexes based approach degrades significantly when log parsers are incorrect. LogRobust and HitAnomaly perform better than other baselines due to using the semantic information of logs.

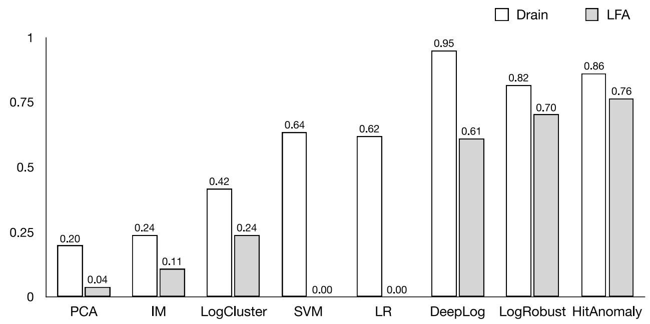
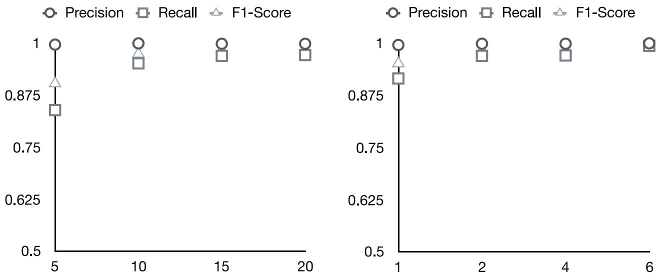


Fig. 12. F1 score comparison with different log parsers.



(a) Window size

(b) Number of layer

Fig. 13. Analysis of HitAnomaly with different parameters.

3) Analysis of HitAnomaly: We first investigate the performance impact of window size and the number of layers in HitAnomaly. As shown in Figure 13, we varied the values of one parameter while using the default values for others and report its results over the HDFS data set.

In general, the precision of HitAnomaly is fairly stable with respect to different window sizes or the number of layers. Figure 13(a) shows that the length of window size has an impact on the recall of anomaly detection and a smaller window size leads to lower recall. Under a short window size setting, the HitAnomaly model captures fewer failure symptoms, causing our model to be unable to detect some types of anomalies and resulting in poor performance. Figure 13(b) presents the performance impact of number of layer in HitAnomaly. We observe that the HitAnomaly model is not very sensitive to different number of layer settings and is able to achieve good performance even though it only has one layer. The HitAnomaly model, with a greater number of layers, can achieve a better performance. However, a greater number of parameters lead to a longer training time and prediction time.

In this work, we design both the log sequence encoding and the parameter values encoding for anomaly detection. To demonstrate the benefits of the combination of these two encoders, we calculate the accuracy of HitAnomaly without (w/o) log sequence encoding. This captures the semantic information of log template sequences as well as the accuracy of HitAnomaly without (w/o) the parameter value representation vector.

As shown in Table III, we conducted our ablation experiments over the BGL dataset. Compared to the original HitAnomaly, HitAnomaly without log sequence encoding has a much lower precision. The information on the log template sequence is essential for the anomaly detection model. HitAnomaly without parameter values encoding has a lower recall over the BGL dataset, which demonstrates that this setting ignores some anomalies caused by an irregular parameter value. HitAnomaly without parameter values encoding and the LogRobust model both ignore the information of parameter values. However, HitAnomaly outperforms LogRobust on the BGL dataset, which means that the transformer structure can efficiently handle the semantic information of log templates and is highly effective in handling complex log patterns.

TABLE III
ABLATION EXPERIMENTS OF HITANOMALY

	Precision	Recall	F1-Score
HitAnomaly	0.948	0.897	0.921
w/o log sequence encoding	0.756	0.834	0.793
w/o parameter value encoding	0.933	0.847	0.890

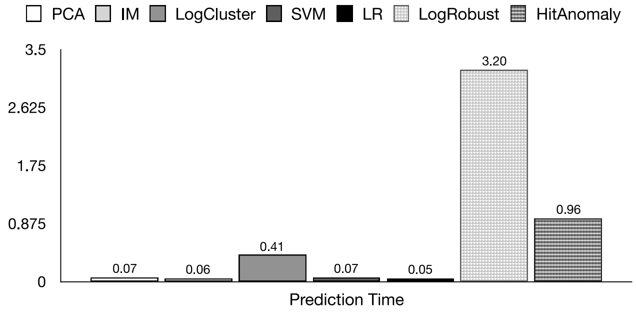


Fig. 14. Execution time comparison result.

Some enterprise systems often produce a large volume of event logs, anomaly detection model shall be time efficient. We evaluated the overhead of our anomaly detection model in the HDFS dataset. These results are the prediction time based on over 2000 experiment runs. In this article, these experiments were conducted on a 2.60GHz Intel Xeon CPU PC machine with 16-gigabyte main memory.

As shown in Figure 14, we observe that our proposed HitAnomaly model requires longer prediction time than classic machine learning models e.g., PCA, IM and LR. However, compared with neural network model LogRobust, our model typically runs three times faster. We hypothesize that the reason is that in terms of computational complexity, self-attention layers are faster than recurrent layers [12].

C. Experiment on the Unstable Log Data

In this section, we evaluate our method on unstable log data. In practice, log data often contains previously unseen log events or log sequences. Following the analysis of LogRobust, we assess the effects on the unstable log data to evaluate the robustness of our model. We conduct our experiments from two aspects: unseen log events and new but similar log sequences.

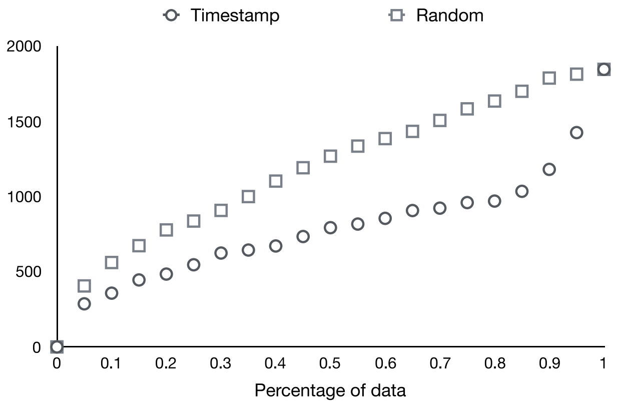


Fig. 15. Log templates distribution on BGL dataset.

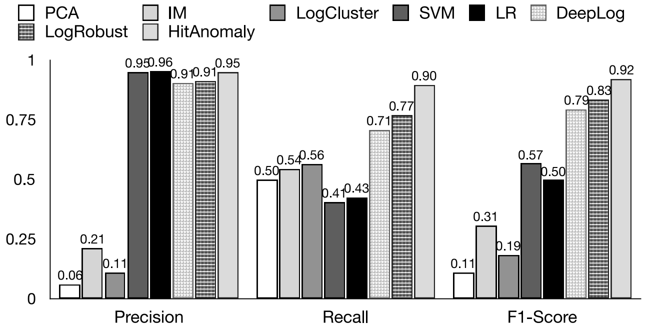


Fig. 16. Evaluation on BGL dataset.

1) Evaluation on Unseen Log Events: We first introduce the evaluation results on the BGL original log data. As describe in Table II, about 50% of log templates in the BGL dataset are used as the training data, and half of the log templates only happen in the testing set. The log templates distribution statistics for the BGL data are presented in Figure 15. We observe that the number of log templates increases sharply in the remaining 20% data ordered by the log timestamp. After random shuffling, the log templates is close to linear distribution. Figure 15 means that log data of BGL have an obvious update after 80% data. The BGL data is suitable for evaluating robustness of model.

The evaluation results of BGL data are shown in Figure 16. The results demonstrate that our proposed model outperforms other baseline models on the BGL data set, with an F1-score of 92.1%.

One interesting finding is that log event indexes based approaches (except LogRobust and HitAnomaly) have poor performance over BGL data. PCA, IM, and LogCluster methods generate a large number of false alarms since they have low precision. Generally, large service systems often produce a large volume of event logs every day. If a log-based anomaly detection method generates a large number of false alarms every day, it will become unmanageable by system admins. On the other hand, SVM and LR methods miss a large volume of actual anomalies with low recall rate. This result may be explained by the fact that the BGL dataset has a large number of log templates, which leads to a sparse counting matrix, rendering these techniques ineffective on the BGL dataset. Another reason is that a large portion (about 50%) templates are in test data, which makes it hard for the event indexes based methods to learn features from unseen templates. On the contrary, LogRobust and HitAnomaly methods can tackle this problem by learning the semantic information of the log templates.

TABLE IV
STATISTICS OF UNSEEN TEMPLATES ON BGL DATA SET (THE FIRST ROW IS TRAINING RATIO)

	1%	10%	20%	50%
# in training	156	358	452	672
% unseen in testing	91.6%	83.1%	81.80%	80.61%

We also observe that HitAnomaly gives a 13% performance improvement for recall compared to LogRobust. LogRobust leverages the log template sequence(without parameter values) to learn the anomalous and normal patterns. However, some anomalies are caused by irregular parameter values from a normal log template sequence. HitAnomaly models parameter value, which can capture the semantic information in parameter values and improve the recall rate of the model.

To further evaluate the effectiveness of HitAnomaly on unseen log events, we conduct four experiments, respectively using the first 1% log data, 10%, 20% and 50% as training data. We show the statistics of unseen templates on BGL data set in Table IV. We can find there are only 156 templates in training data. More than 90% log templates of testing data are unseen when using the first 1% log data for training. Even if using the first 50% as training data, there are still more than 80% of the log templates to be new for testing. Because the log indexes based methods under-perform in the BGL dataset, we only compare the performance of SVM and LogRobust methods in this experiment.

The evaluation results on unseen log events are shown in Table V. The results demonstrate that our proposed model outperforms other baseline models on unseen log events. We can observe that with the decreasing training ratio, the performance of the SVM approach has declined in different degrees. However, HitAnomaly still achieves an F1-score more than 90%, only using 1% log data for training. It confirms that our approach is robust against the unstable log events. The reason for this robustness is that the transformer structure can effectively capture the semantic information from the log sequences and the log indexes based method cannot handle unseen log templates. Another important finding is that the results of the HitAnomaly model using 10% for training is comparable to using 80% log data as training. This demonstrates that our proposed model can utilize fewer log data for training.

2) Evaluation on Unstable Log Sequences: In order to show the effectiveness of our approach in tackling the instability of log data, we follow LogRobust to create unstable testing datasets based on the original HDFS dataset. We synthesize the unstable log sequences data, as illustrated in Table VI.

TABLE V
EVALUATION ON BGL UNSTABLE DATA SET

Training Ratio	Model	Precision	Recall	F1-Score
1%	SVM	0.981	0.341	0.506
	LogRobust	0.868	0.752	0.805
	HitAnomaly	0.968	0.845	0.902
10%	SVM	0.973	0.397	0.564
	LogRobust	0.873	0.812	0.841
	HitAnomaly	0.983	0.877	0.922
20%	SVM	0.982	0.383	0.551
	LogRobust	0.867	0.821	0.843
	HitAnomaly	0.977	0.887	0.929
50%	SVM	0.970	0.405	0.571
	LogRobust	0.897	0.792	0.841
	HitAnomaly	0.964	0.886	0.923

TABLE VI
SYNTHETIC LOG SEQUENCES ON HDFS DATASET

Original log sequence:
Event 1 → Event 2 → Event 3 → Event 4 → Event 5
Deletion in sequence:
Event 1 → Event 2 → ~~**Event 3**~~ → Event 4 → Event 5
Shuffle in sequence:
Event 1 → **Event 4** → **Event 2** → **Event 3** → Event 5
Insertion in sequence:
Event 1 → Event 2 → **Event 3** → **Event 3** → Event 4 → Event 5

As introduced in LogRobust, log sequences are likely to be changed during the process of log evolution or collection. We adopt three methods to simulate unstable log sequences. First, we randomly remove a few unimportant log templates (which do not affect the corresponding anomaly status labels) from the original log sequences. We also shuffle a sub-sequence in log templates or randomly select an unimportant log event and repeat it several times in a log sequence. We randomly inject these synthetic logs into the original log data.

As shown in Table VII, we evaluate the performance of our proposed method with different injection ratios. The injection ratio means the percentage of synthetic log sequences in log data. We can observe that with the increasing injection ratio, the performance of the SVM approach has declined a lot (F1-score from 0.959 to 0.552), which demonstrates the sensitivity of the log event indexed based methods to changes in sequences. HitAnomaly performs best under 5% and 10% injection ratios and achieves a comparable result under a 20% injection ratio. The evaluation results on unstable log sequences confirm that the hierarchical transformer model is robust to small variations in the sequences.

TABLE VII
EVALUATION ON HDFS SYNTHESIZED DATA SET

Injection Ratio	Model	Precision	Recall	F1-Score
5%	SVM	0.973	0.946	0.959
	LogRobust	0.998	0.932	0.956
	HitAnomaly	0.968	0.968	0.982
10%	SVM	0.941	0.508	0.660
	LogRobust	0.945	0.922	0.933
	HitAnomaly	0.962	0.952	0.956
20%	SVM	0.437	0.752	0.552
	LogRobust	0.983	0.929	0.955
	HitAnomaly	0.945	0.948	0.946

V. RELATED WORK

Recording runtime status or events via log is very common for almost all computer systems. System logs are abundantly informative and valuable resources for anomaly detection and diagnosis. However, since logs are composed mainly of unstructured, free-form text, log-based analysis is challenging.

There have been many studies on log-based anomaly detection. Most early research on log-based anomaly detection uses rule-based approaches [29]–[32]. These methods use keywords (e.g., "fail") or regular expression to match anomaly logs. These are limited by specific scenarios and also require domain knowledge. For example, Cinque *et al.* [30] propose a rule-based approach to analyze software failures. It manually extracts some entries about service error or service complaints to build its rules. Oprea *et al.* [31] uses belief propagation to detect early-stage enterprise infection from DNS logs. Logsurfer [32] dynamically changes its rules based on events or time. This provides much of the flexibility needed to relate events. HitAnomaly is a general approach that does not rely on any domain-specific knowledge.

Some works employed usage analysis to extract parameter values from logs and then used classical anomaly detection techniques [33]–[35]. Barham *et al.* [33] obtained canonical request descriptions to construct concise workload models for performance prediction and anomaly detection. We can also regard parameter values from log sequences as sequential data. Graph-based model is another method widely used for anomaly detection. Noble and Cook [36] introduced a graph-based method to detect anomaly by calculating the regularity of a graph. Hilmi *et al.* proposed a spectral anomaly detection model using graph-based filtering, which can project the graph signals on normal and anomaly subspaces [37]. These approaches heavily depend on the quality of log parser and also ignore the semantic information in log messages.

Recently existing automatic log anomaly detection approaches apply a two-step procedure. First, a log parser [14]–[18] is used to parse log messages to log templates and parameters. Parameter values and timestamps are discarded. Then these methods use log event indexes or count vectors to represent log sequences. After constructing such a matrix, they apply Principal Component Analysis (PCA) [5] or invariant mining (IM) [6] to build unsupervised anomaly detection models. Supervised methods [7], [24] use normal and abnormal vectors to train a binary classifier that detects anomalies in new log data. Deeplog [8] uses a small portion of normal data to train its model. It utilizes LSTM to predict the next logs likely to appear after current log sequences. An anomaly is detected if the actual log is unexpected according to the prediction. These approaches all ignore the semantic information in log messages. We designed a log sequence encoder to capture the context of semantic information, which can improve the performance of anomaly detection.

Log template semantics-based approaches first extract log templates from log messages and then model templates as a natural language sequence. They utilize word embedding to transform log templates into vectors and train their classification model based on these vectors. LogAnomaly [9] proposes a template2vec method to extract the semantic information, including synonyms and antonyms hidden in log templates. It is the first anomaly detection framework considering semantic information of logs. LogRobust [10] also extracts semantic information of log events and represents them as semantic vectors. It then detects anomalies by utilizing an attention-based Bi-LSTM model. However, such approaches only utilize log template sequences and have not dealt with parameter values. HitAnomaly designs a parameter value encoder to utilize parameter values, which is missing in these two works.

VI. Conclusion

This article proposed HitAnomaly, a log-based anomaly detection model utilizing a hierarchical transformer structure to model both log template sequences and parameter values. We designed a log sequence encoder and a parameter value encoder to obtain their representations respectively and built its classification model based on an attention mechanism. The research has shown that the transformer model outperforms LSTM and the hierarchical structure can effectively model log sequences. We evaluated our proposed method on three log datasets. Our experimental results demonstrated that HitAnomaly has outperformed other existing log-based anomaly detection methods. We also evaluated the performance and robustness of our model on unstable log data.

One of the future directions of our work is to incorporate the transformer structure into a log-based anomaly prediction task. This, in turn, will be able to predict anomalies before they occur and allows actions to be taken to prevent anomalies from happening and reduce the damage from anomalies.

References

[1] Y. Tan and X. Gu, "On predictability of system anomalies in real world," in *Proc. IEEE Int. Symp. Model. Anal. Simulat. Comput. Telecommun. Syst.*. 2010, pp. 133–140.

[2] S. Huang, C. Fung, K. Wang, P. Pei, Z. Luan, and D. Qian, "Using recurrent neural networks toward black-box system anomaly prediction," in *Proc. IEEE/ACM 24th Int. Symp. Qual. Service (IWQoS)*. 2016, pp. 1–10.

[3] S. Huang *et al.*, "Arena: Adaptive real-time update anomaly prediction in cloud systems," in *Proc. 13th Int. Conf. Netw. Service Manag. (CNSM)*. 2017, pp. 1–9.

[4] Q. Lin, H. Zhang, J.-G. Lou, Y. Zhang, and X. Chen, "Log clustering based problem identification for online service systems," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng. Companion (ICSE-C)*. 2016, pp. 102–111.

[5] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proc. ACM SIGOPS 22nd Symp. Oper. Syst. Principles*, 2009, pp. 117–132.

[6] J.-G. Lou, Q. Fu, S. Yang, Y. Xu, and J. Li, "Mining invariants from console logs for system problem detection," in *Proc. USENIX Annu. Tech. Conf.*, 2010, pp. 1–14.

[7] P. He, J. Zhu, S. He, J. Li, and M. R. Lyu, "Towards automated log parsing for large-scale log data analysis," *IEEE Trans. Dependable Secure Comput.*, vol. 15, no. 6, pp. 931–944, Nov./Dec. 2018.

[8] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *Proc. ACM SIGSAC Conf. Comput. Commun. Security*, 2017, pp. 1285–1298.

[9] W. Meng *et al.*, "LogAnomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs," in *Proc. 28th Int. Joint Conf. Artif. Intell. (IJCAI)*, vol. 7, 2019, pp. 4739–4745.

[10] X. Zhang *et al.*, "Robust log-based anomaly detection on unstable log data," in *Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Foundations Softw. Eng.*, 2019, pp. 807–817.

[11] F. A. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: Continual prediction with LSTM," in *Proc. 9th Int. Conf. Artif. Neural Netw. (ICANN)*, 1999, pp. 2451–2471.

[12] A. Vaswani *et al.*, "Attention is all you need," in *Proc. 31st Conf. Neural Inf. Process. Syst.*, 2017, pp. 5998–6008.

[13] A. Oliner and J. Stearley, "What supercomputers say: A study of five system logs," in *Proc. 37th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*. 2007, pp. 575–584.

[14] K. Yamanishi and Y. Maruyama, "Dynamic syslog mining for network failure monitoring," in *Proc. 11th ACM SIGKDD Int. Conf. Knowl. Discovery Data Min.*. 2005, pp. 499–508.

[15] A. Oliner, A. Ganapathi, and W. Xu, "Advances and challenges in log analysis," *Commun. ACM*, vol. 55, no. 2, pp. 55–61, 2012.

[16] S. Niwattanakul, J. Singthongchai, E. Naenudorn, and S. Wanapu, "Using of Jaccard coefficient for keywords similarity," in *Proc. Int. MultiConf. Eng. Comput. Scientists*, vol. 1, 2013, pp. 380–384.

[17] 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 *Proc. 46th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*. 2016, pp. 654–661.

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

[19] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," 2018. [Online]. Available: arXiv:1810.04805.

[20] A. Joulin, E. Grave, P. Bojanowski, and T. Mikolov, "Bag of tricks for efficient text classification," 2016. [Online]. Available: arXiv:1607.01759.

[21] Y. Wu *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," 2016. [Online]. Available: arXiv:1609.08144.

[22] N. Reimers and I. Gurevych, "Sentence-BERT: Sentence embeddings using siamese BERT-networks," in *Proc. Conf. Empir. Methods Nat. Lang. Process. 9th Int. Joint Conf. Nat. Lang. Process. (EMNLP-IJCNLP)*, 2019, pp. 3980–3990.

[23] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proc. COMPSTAT*, 2010, pp. 177–186.

[24] Y. Liang, Y. Zhang, H. Xiong, and R. Sahoo, "Failure prediction in IBM BlueGene/L event logs," in *Proc. 7th IEEE Int. Conf. Data Mining (ICDM)*, 2007, pp. 583–588.

[25] A. Paszke *et al.*, "Automatic differentiation in PyTorch," in *Proc. 31st Conf. Neural Inf. Process. Syst. Workshop*, 2017.

[26] J. Zhu *et al.*, "Tools and benchmarks for automated log parsing," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng. Softw. Eng. Pract. (ICSE-SEIP)*, 2019, pp. 121–130.

[27] T.-D. Nguyen, B. Schmidt, and C.-K. Kwoh, "SparseHC: A memory-efficient online hierarchical clustering algorithm," *Procedia Comput. Sci.*, vol. 29, no. 29, pp. 8–19, 2014.

[28] M. Nagappan and M. A. Vouk, "Abstracting log lines to log event types for mining software system logs," in *Proc. 7th IEEE Working Conf. Min. Softw. Repositories (MSR)*. 2010, pp. 114–117.

[29] J. P. Rouillard, “Real-time log file analysis using the simple event correlator (SEC),” in *Proc. LISA*, vol. 4, 2004, pp. 133–150.

[30] M. Cinque, D. Cotroneo, and A. Pecchia, “Event logs for the analysis of software failures: A rule-based approach,” *IEEE Trans. Softw. Eng.*, vol. 39, no. 6, pp. 806–821, Jun. 2013.

[31] A. Oprea, Z. Li, T.-F. Yen, S. H. Chin, and S. Alrwais, “Detection of early-stage enterprise infection by mining large-scale log data,” in *Proc. 45th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2015, pp. 45–56.

[32] J. E. Prewett, “Analyzing cluster log files using logsurfer,” in *Proc. 4th Annu. Conf. Linux Clusters*, 2003, pp. 83–95.

[33] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, “Using magpie for request extraction and workload modelling,” in *Proc. OSDI*, vol. 6, 2004, p. 18.

[34] L. Cherkasova, K. Ozonat, N. Mi, J. Symons, and E. Smirni, “Automated anomaly detection and performance modeling of enterprise applications,” *ACM Trans. Comput. Syst. (TOCS)*, vol. 27, no. 3, pp. 1–32, 2009.

[35] T. Jia, P. Chen, L. Yang, Y. Li, F. Meng, and J. Xu, “An approach for anomaly diagnosis based on hybrid graph model with logs for distributed services,” in *Proc. IEEE Int. Conf. Web Services (ICWS)*, 2017, pp. 25–32.

[36] C. C. Noble and D. J. Cook, “Graph-based anomaly detection,” in *Proc. 9th ACM SIGKDD Int. Conf. Knowl. Discovery Data Min.*, 2003, pp. 631–636.

[37] H. E. Egilmez and A. Ortega, “Spectral anomaly detection using graph-based filtering for wireless sensor networks,” in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process. (ICASSP)*, 2014, pp. 1085–1089.

Shaohan Huang received the B.S. and M.S. degrees from Beihang University, Beijing, China, where he is currently pursuing the Ph.D degree. His current research interests include anomaly detection, text analytic, and natural language processing.

Yi Liu (Member, IEEE) received the Ph.D. degree from the Department of Computer Science, Xi’an Jiaotong University in 2000. He is currently a Professor with the School of Computer Science and Engineering and the Director of the Sino-German Joint Software Institute, Beihang University, China. His research interests include computer architecture, high-performance computing, and network technology.

Carol Fung is an Associate Professor with Virginia Commonwealth University. Her research interests include collaborative intrusion detection networks, social networks, security issues in mobile networks and medical systems, security issues in next generation networking, and machine learning in intrusion detection. She serves as an Associate Editor in multiple journals, including IEEE TRANSACTIONS ON NETWORK AND SERVICE MANAGEMENT and *Computer Networks* (Elsevier).

Rong He received the master’s degree in computer science. She is currently working with the Computer Network Information Center, Chinese Academy of Sciences and studying with the University of Chinese Academy of Sciences. Her research interests include grid computing and distributed systems.

Yining Zhao received the Ph.D. degree in computer science. He is currently working with the Computer Network Information Center, Chinese Academy of Sciences. His research interest includes distributed computing and data analyses.

Hailong Yang (Member, IEEE) received the Ph.D. degree from the School of Computer Science and Engineering, Beihang University, in 2014, where he is currently working as an Associate Professor. He has been involved in several scientific projects, such as performance analysis for big data systems and performance optimization for large scale applications. His research interests include parallel and distributed computing, HPC, performance optimization, and energy efficiency. He is a Member of China Computer Federation.

Zhongzhi Luan (Member, IEEE) is an Associate Professor with the School of Computer Science and Engineering, Beihang University. He has been engaged in teaching and research of distributed computing, high performance computing, and computer architecture for many years. His main research interests include resource management in distributed environment, supporting methods and technologies for application development on heterogeneous computing systems, performance analysis, and optimization of high performance computing applications.