

Log-based Anomaly Detection Uses All Contextual Information with a Linear Self- Attention Encoder

Chi-Ming Chou

s1136107@mail.yzu.edu.tw

Yuan Ze University

Shang-Kuan Chen

Yuan Ze University

Research Article

Keywords: Anomaly detection, Deep learning, System log, Linear self-attention

Posted Date: July 3rd, 2026

DOI: <https://doi.org/10.21203/rs.3.rs-8217733/v1>

License:   This work is licensed under a Creative Commons Attribution 4.0 International License.

[Read Full License](#)

Additional Declarations: No competing interests reported.

Log-based Anomaly Detection Uses All Contextual Information with a Linear Self-Attention Encoder.

Chi-Ming Chou (corresponding author), Shang-Kuan Chen

Yuan Ze University, No. 135, Yuandong Rd., Zhongli Dist., Taoyuan City 320315,
Taiwan (R.O.C.)

E-mail addresses: s1136107@mail.yzu.edu.tw (Chi-Ming Chou),

cotachen@saturn.yzu.edu.tw (Shang-Kuan Chen)

Keywords:

Anomaly detection

Deep learning

System log

Linear self-attention

Abstract

Protecting systems from cyber threats is increasingly challenging as interconnected systems rapidly expand the attack surface. While log-based anomaly detection is widely adopted, most existing approaches rely on log parsing, and parsing errors inevitably weaken detection reliability. This paper introduces AllLinLog, a streamlined model that performs anomaly detection directly on system logs, removing the need for a log parser and preserving complete semantic information. However, using full contextual information increases log session length and poses a challenge to Transformers-based models. To address this, the proposed model employed a lightweight self-attention encoder with linear complexity, thereby reducing computational cost and enabling the scalable processing of long log sequences. Experiments show that the proposed model outperforms log-parsed models in detection accuracy and delivers faster inference, demonstrating its practical advantage for log-based anomaly detection.

1. INTRODUCTION

Interconnected systems have become essential to modern infrastructure, enabling smooth data sharing, remote access, and intelligent coordination across various environments. As these systems grow in size and complexity, the security attack surface also increases considerably. Failing to detect and address security threats and vulnerabilities can lead to operational disruptions, financial losses, data breaches, legal and regulatory penalties, damage to reputation, and loss of customers. For example, credential stuffing attacks on Roku [1] compromised over 576,000 accounts. Likewise, the Target data breach incident [2] affected over 40 million credit and debit card accounts, and 70 million personal records were stolen. System administrators must promptly detect and alert to anomalous events to prevent those incidents.

Software developers record and print system status in the system logs shown in Figure 1, which typically include a timestamp, logging level, and log messages. System logs are crucial for identifying abnormal events that may indicate security breaches, system failures, or performance issues. However, manually analyzing massive system logs collected from distributed devices will be time-consuming and prone to error. Log-based anomaly detection can automatically detect and timely alert to anomalies, reducing the burden on system administrators. Most devices incorporate built-in system log mechanisms, making log-based anomaly detection a highly suitable approach for detecting anomalous incidents.

Log Events

```
- 1117842440 2005.06.03 R23-M0-NE-C:J05-U01 2005-06-03-16.47.20.730545 R23-M0-NE-C:J05-  
U01 RAS KERNEL INFO 63543 double-hammer alignment exceptions  
- 1117842974 2005.06.03 R24-M0-N1-C:J13-U11 2005-06-03-16.56.14.254137 R24-M0-N1-C:J13-  
U11 RAS KERNEL INFO 162 double-hammer alignment exceptions  
- 1117843015 2005.06.03 R21-M1-N6-C:J08-U11 2005-06-03-16.56.55.309974 R21-M1-N6-C:J08-  
U11 RAS KERNEL INFO 141 double-hammer alignment exceptions  
- 1117848119 2005.06.03 R16-M1-N2-C:J17-U01 2005-06-03-18.21.59.871925 R16-M1-N2-C:J17-  
U01 RAS KERNEL INFO CE sym 2, at 0x0b85eee0, mask 0x05 APPREAD  
1117869872 2005.06.04 R04-M1-N4-I:J18-U11 2005-06-04-00.24.32.432192 R04-M1-N4-I:J18-U11  
RAS APP FATAL ciod: failed to read message prefix on control stream (CioStream socket to  
172.16.96.116:33569
```

Figure 1. System logs include the timestamp, logging level, and messages.

Log-based anomaly detection typically involves three steps. First, raw logs are processed, either with or without the use of a log parser, and grouped into log sessions. The second step converts these sessions into vectors or tokens to prepare them for anomaly detection. The final step classifies each log session as either regular or anomalous. In the system logs, developers use unrestricted text entries such as variable names, abbreviations, or technical terms to log status updates or errors. These unseen text combinations lead to out-of-vocabulary (OOV) issues. Additionally, system logs are unstructured text, which increases the difficulty of automatically analyzing log data. To address these issues in anomaly detection, as illustrated in Figure 2, prior works used log parsers [3] to extract timestamps, logging levels, components, events, and parameters from logs, transforming raw logs into structured event templates. Event templates are created by retaining the keywords that are constant parts of logs and removing parameters that carry dynamic runtime information. Existing log parsing methods include language modeling [4], heuristics [5–7], frequent pattern mining [8, 9], clustering [10], and longest common subsequence [11]. Among them, Drain [6] is recognized as the most accurate and efficient [3].

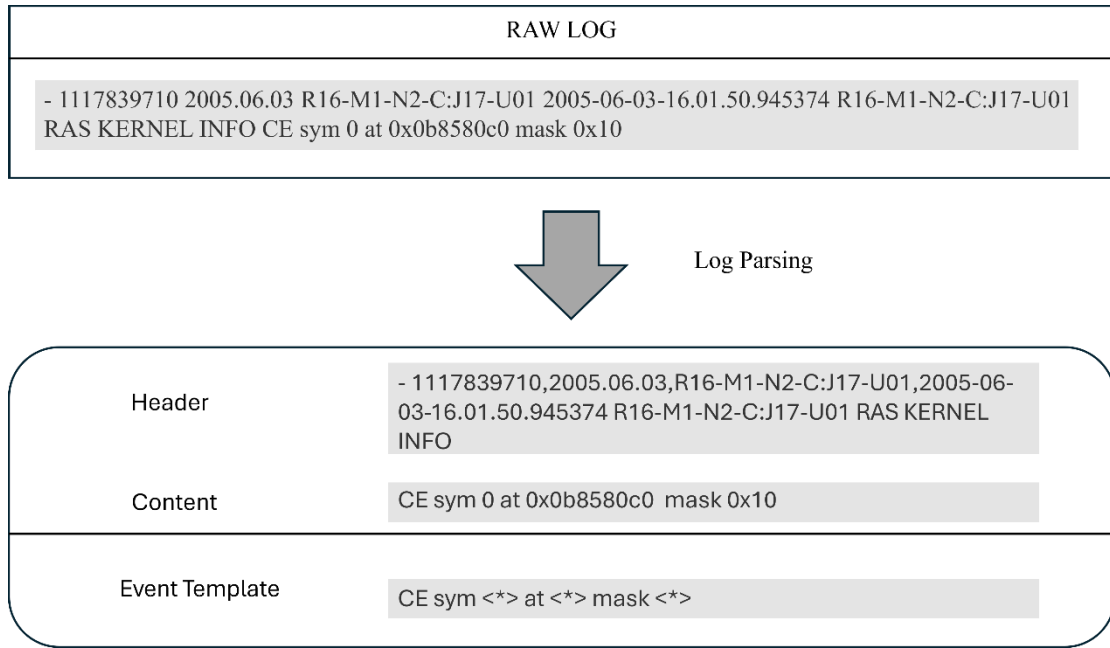


Figure 2. An example of the log parsing process

However, log parsers can cause semantic misunderstandings, such as misinterpreting parameters as keywords, which leads to extra log events. Conversely, some crucial keywords may be mistaken for parameters and removed, resulting in different log messages being merged into a single event. Furthermore, evolving software systems add new logging statements and system status parameters to the logs, leading to more unseen templates. As a result, log parsers must retrain to accommodate these changes. Studies [12, 13] have shown that computational overhead for log parsing, semantic misunderstanding errors, and loss of contextual details are challenges that can negatively impact the performance of log anomaly detection models, limiting the robustness and generality of these models in dynamic real-world logging systems.

In the field of anomaly detection, researchers have proposed data-driven machine learning techniques. Chen et al. [14] classified successful and failed requests using a decision tree. Breier et al. [15] utilized PCA to analyze and highlight anomalies. LogCluster [16] segmented logs through clustering and performed anomaly detection

using an unsupervised learning method. Machine learning methods struggle to generalize because they can't effectively capture sequential or contextual relationships in high-dimensional log spaces. As a result, researchers shifted their focus to deep learning techniques. Deeplog [17] utilized LSTM (Long Short-Term Memory) to capture the sequential relationship of log templates, achieving high accuracy in anomaly detection. Lu et al. [18] employed a CNN (Convolutional Neural Network), which can learn the hidden relationships between log events. Transformers [19], which combine efficient parallel processing with the Multi-headed Self-attention mechanism, outperform LSTMs, which are inherently sequential in capturing both local and long-range dependencies. Compared to CNNs, which focus on regional patterns and require depth to understand global context, Transformers analyze log sequences more effectively. LogBERT [20] leverages the Bidirectional Encoder Representations from Transformers (BERT) model to learn patterns in regular log sequences through unsupervised learning and then identifies outliers. LogGPT [21] employed the Generative Pre-trained Transformer (GPT) model for log anomaly detection, analyzing log sequences with the Chain of Thought (CoT) prompts. FSMFLog [22] utilized SBERT to get the semantic features as input, also projected the time and type features from parsed log event sequences, inserting them into the Transformer encoder as biases for anomaly detection. LogFormer [23] utilized a Transformer-based framework for log anomaly detection, thereby enhancing detection by incorporating overlooked temporal and other kinds of features during log parsing. However, the quadratic complexity of Multi-headed Self-attention makes the Transformer sensitive to long input sequences, which causes these models to discard features or truncate the input log sequence length. This might lead to the loss of valuable information in anomaly detection.

To mitigate the drawbacks of using a log parser, preserve all contextual information in logs, and efficiently detect anomalies from logs, AllLinLog (utilizing all contextual

information with the linear self-attention encoder in Log-based anomaly detection) is proposed for this purpose. This new framework uses the Linear Self-Attention (LSA) encoder to process all contextual information for log-based anomaly detection. It not only reduces the computational complexity from quadratic to linear but also maintains contextual dependencies. It exceeds baseline models in both anomaly detection and resource efficiency, resulting in faster inference.

2. RELATED WORK

Research [12, 24] has shown that the performance of anomaly detection is affected by the imperfection of using log parsers. Without using a log parser, both NeuralLog [12] and LogLLM [25] employ a BERT tokenizer, WordPiece, to process raw logs. NeuralLog embeds the alphabetic units of the content in raw log sequences, thereby learning anomalies from these embedded vectors. LogLLM embeds and then projects log sequences as vectors compatible with Llama (LLM), then packages these vectors as prompts to enable the LLM to identify anomalies. ContextLog [26] does not use a log parser either. Instead, ContextLog uses a GPT-supported tokenizer, BPE (Byte Pair Encoding), to embed both alphabetic and non-alphabetic units from the header and content parts of raw log sequences, thereby preserving all contextual information for anomaly detection.

The nature of the log dataset reveals a clear imbalance between regular and anomalous classes. Class resampling methods must be considered to achieve unbiased classification. Ma et al. [27] conclude that random oversampling of the minority class (anomalous) is the optimal class balancing strategy for log-based anomaly detection.

Due to its efficient parallel processing and strong ability to retain long-term information within input sequences, the multi-head self-attention mechanism serves as the current standard for anomaly detection. The computational complexity of self-attention is $O(n^2)$, where n is the sequence length of tokens in a log session. In log-based anomaly detection, to retain all contextual information, it must include both alphabetic and non-alphabetic tokens from embedded log messages. This causes the sequence length of a log session to grow to thousands of tokens, making the self-attention mechanism inefficient for such long sequences. Different self-attention variants have been developed to reduce the computational complexity from quadratic

to linear, thereby enhancing the efficiency of self-attention in handling long input sequences. Instead of computing attentions for all tokens, Sparse Transformer [28] leverages sparsity in the attention mechanism. Using sparse patterns to do factorized attention, such as striding, restricts each token’s attention to a subset of tokens, reducing computational complexity to $O(n\sqrt{n})$ but still capturing both local and long-range dependencies. Reformer [29] develops Locality-Sensitivity Hashing (LSH) attention to group similar tokens into buckets and limit attention computation within these buckets, reducing the computation complexity of self-attention to $O(n\log n)$. Wang et al. [30] demonstrate that a low-rank matrix can approximate the self-attention mechanism and propose a linear transformer, Linformer, which reduces the computational complexity of self-attention to $O(n)$ in both time and space.

In this paper, a streamlined log-based anomaly detection model is presented that is robust and adaptable to changing system logs, using the BPE tokenizer to avoid the need for a log parser. The proposed model performs unbiased anomaly detection by addressing class imbalance using the Random Oversampling technique. A linear self-attention encoder is also incorporated into the proposed model to efficiently and effectively process long log sequences, preserving all contextual information without discarding or truncating the input, thereby mitigating the quadratic complexity imposed by Transformers. The details of the proposed model are described in the Methodology section below.

3. Methodology

The proposed model, AllLinLog, is designed as a lightweight log-based anomaly detection framework. It is composed of three major components:

- 1) Data Processing: directly embedding raw log sessions using the GPT-4 BPE tokenizer to preserve all semantic and temporal information.
- 2) Linear Self-Attention (LSA) Encoder: reduces the computation complexity of self-attention to enable the proposed model to handle long log sequences efficiently.
- 3) Classifier: aggregating the encoder outputs and performing the overall anomaly detection process.

Figure 3 illustrates the AllLinLog methodology, including data processing, the encoder, and classification stages. The following subsections present the methods of the proposed model.

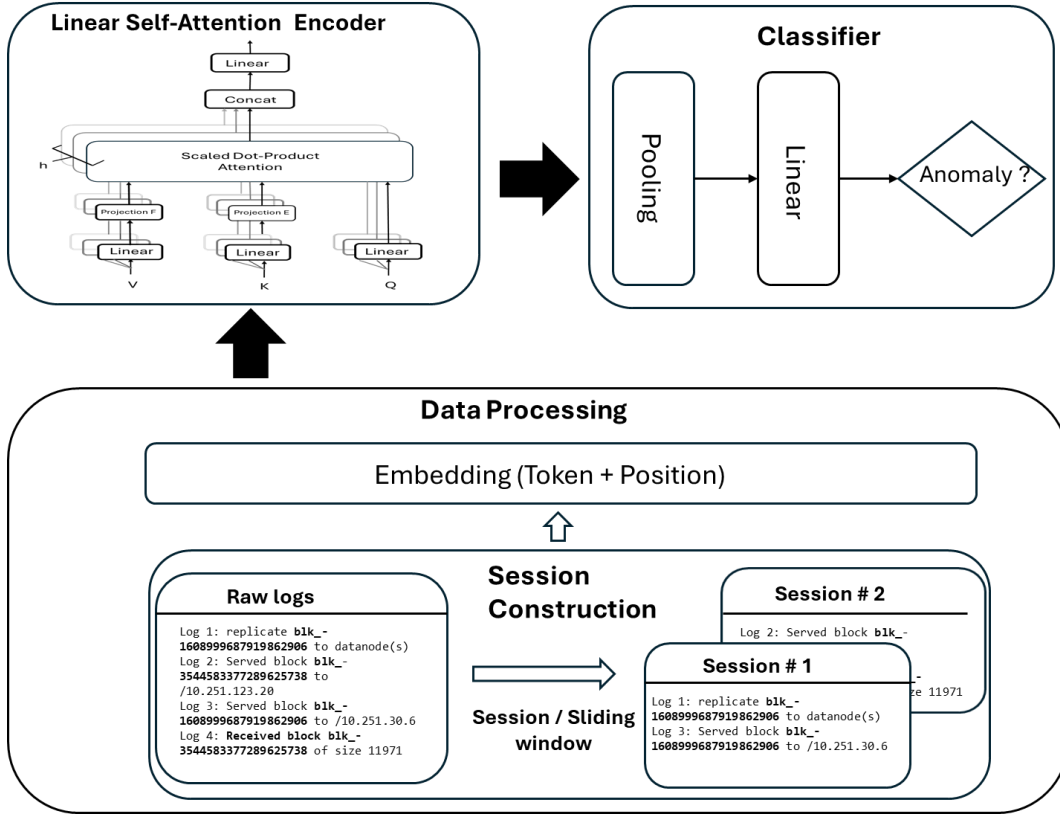


Figure 3. Workflow of the proposed AllLinLog model

3.1 Data Processing

Research typically utilizes open-source system log datasets, such as BGL, Thunderbird, and HDFS, to evaluate the performance of log-based anomaly detection models [31]. Without relying on a complex log parser, the proposed model directly uses raw logs as input, making it lightweight and robust in handling evolving software system logs.

3.1.1 Session Construction

System logs are typically a continuous, unstructured stream of textual entries that contain application sessions. A sequence of log messages forms a session either based on the same session identifier or a fixed-size window. As illustrated in Figures 4 and 5, the proposed model employs two different session construction methods depending

on the nature of the log dataset.

(a) Session ID Grouping (Figure 4)

For datasets such as HDFS, where logs can be linked by a unique identifier (e.g., block IDs), log entries are grouped into sessions based on the shared identifier. Logs referring to the same block ID are grouped into a session.

(b) Sliding Window Grouping (Figure 5)

For datasets such as BGL and Thunderbird, which lack clear identifiers, we use a sliding window approach to generate sessions. As shown in Figure 5, each session consists of a fixed number of consecutive log entries. For example, with a window size of three and a step size of three, non-overlapping sessions are created from the raw log stream.

Let $L = \{l_1, l_2, \dots, l_N\}$ be the log messages. Define $S = \{s_1, s_2, \dots, s_M\}$ is a collection of sessions created from set L . A session is segmented as:

$$S_m = \{l_{m1}, l_{m2}, \dots, l_{mw}\} \text{ where } w = \text{window size} \quad (1)$$

The proposed model is a supervised deep learning model that requires labels for classification. A session is the basic unit for processing. To assign the label (regular or anomalous) to a session, the rule is that if any log in a session contains an anomalous marker (e.g., does not start with ‘-’ in BGL), the entire session is labeled as anomalous. This session-based input construction enables the encoder to capture temporal dependencies and semantic relationships across logs within a session.

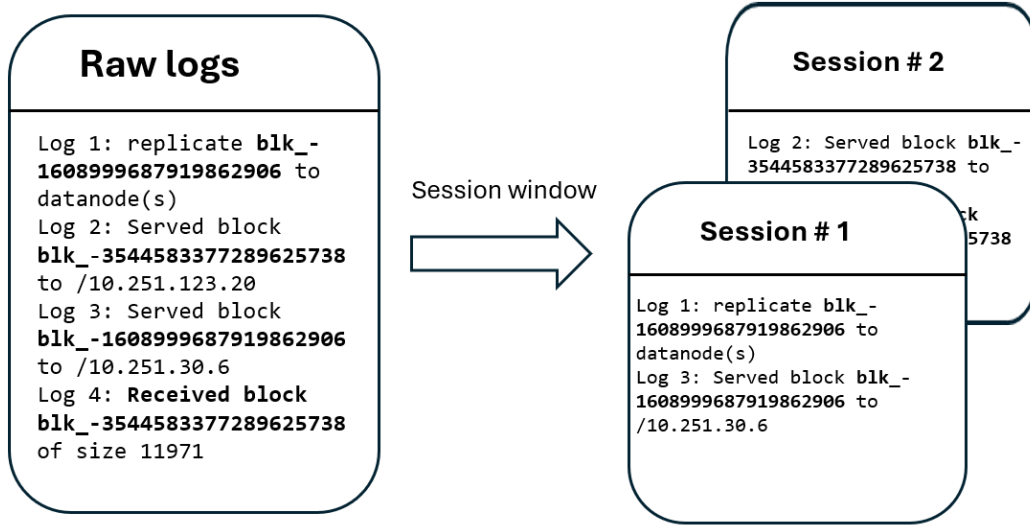


Figure 4: Grouping sessions by session identifiers, such as block ID.

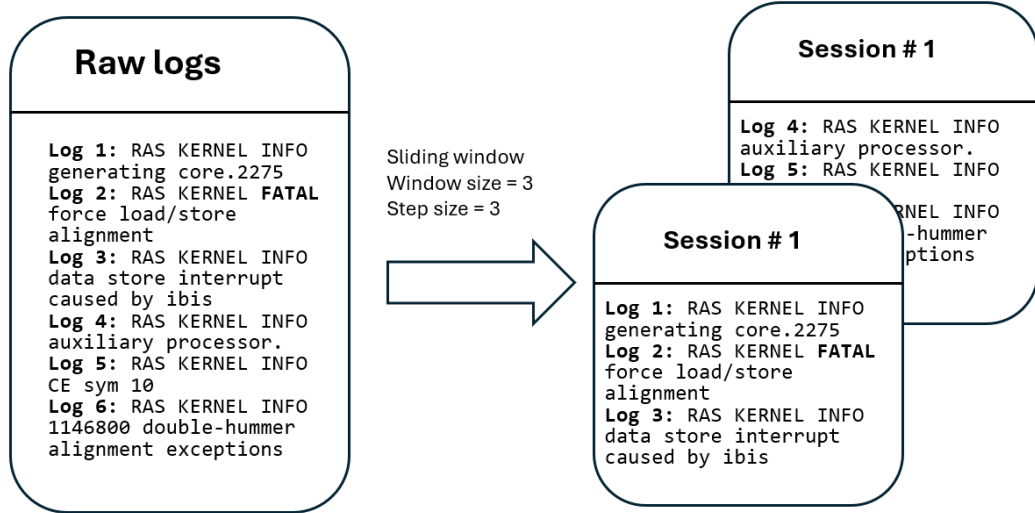


Figure 5. Constructing sessions uses a fixed-size sliding window over the raw system log.

3.1.2 Log Tokenization and Vectorization

After constructing logs for sessions, these unstructured log messages need to be converted to structured vectors for input to the encoder. The proposed model uses GPT-4 BPE, a byte-level subword tokenizer, to split raw log messages into tokens and then encode these tokens into IDs, as shown in Figure 6.

BPE [32], a method for tokenizing text, first converts the text into a byte array. Repeatedly merges the most frequent byte pairs to create new subword units until the

desired vocabulary size is reached. Because it tokenizes text at the byte level, each character, including special characters and emojis, is represented by a single byte. The proposed model utilizes GPT-4 BPE to address Out-Of-Vocabulary (OOV) issues by breaking unseen words into subword tokens, which are then embedded into vectors within the segmented sessions, making them compatible with the encoder's input. As in Figure 6, the unseen word “crond” is split into “cr” and “ond”. Let T be the pretrained GPT-4 tokenizer. The session S_m is tokenized by T to yield a sequence of token IDs.

$$x_m = T(S_m) = [t_1^{(m)}, t_2^{(m)}, \dots, t_{n_m}^{(m)}], t_i^{(m)} \in \mathbb{Z}_+ \quad (2)$$

```
Raw Log      : crond pam_unix 16448 : session closed for user root
Split to Tokens : ['cr', 'ond', ' pam', '_unix', '', '164', '48', ' ', 'session', ' closed', ' for', ' user', ' root']
Encode to IDs  : [5192, 2159, 42290, 81672, 220, 10513, 2166, 551, 3882, 8036, 369, 1217, 3789]
Decode to Log  : crond pam_unix 16448 : session closed for user root

Raw Log      : ERROR: Disk failure at sector 4096.
Split to Tokens : ['ERROR', ':', ' Disk', ' failure', ' at', ' sector', ' ', '409', '6', '.']
Encode to IDs  : [3767, 25, 39968, 8060, 520, 10706, 220, 12378, 21, 13]
Decode to Log  : ERROR: Disk failure at sector 4096.
```

Figure 6. System log tokenized by GPT-4 BPE

Table 1: Average and maximum token counts in dataset sessions

Dataset	# of log messages	Grouping method	# Avg of tokens of a session	# Max of tokens of a session
BGL	4,747,963	Sliding window=10	756	2756
		Sliding window=20	1453	4841
Thunderbird	10,000,000	Sliding window=10	757	4200
		Sliding window=20	1345	7185
HDFS	11,175,629	Session window (same blk-id)	1006	15983

The proposed model segments tokenized log messages into log sessions based on dataset characteristics. Table 1 shows the session lengths of BGL, Thunderbirds, and HDFS. In the BGL dataset, there are 4,747,963 log messages. When a window size of 20 is used to form a session, the average token length of a session is 1453, and the maximum token length is 4841. This long input sequence length will be a challenge for the traditional Transformer encoder. The proposed model addresses this issue with an efficient encoder, as stated in Section 3.2.

3.1.3 Position Embeddings

To incorporate temporal information when using the Transformer's self-attention mechanism to evaluate token importance in parallel, a position embedding must be added to the input sequence. Given the focus on fitting the proposed model to low-resource IoT or edge devices, the counting-based absolute position embedding is

chosen for its computational efficiency, and no external libraries are needed. The position IDs are created as $p_m = [0, 1, \dots, n_m - 1]$, then maps them into a learnable vector with the exact dimensions as the token vector. The final input matrix $E_m \in \mathbb{R}^{n_m \times d}$ for each session m is computed as follows:

$$E_m = Embed_{token}(x_m) + Embed_{position}(p_m) \quad (3)$$

where:

$$Embed_{token} : \mathbb{Z}_+^{n_m} \rightarrow \mathbb{R}^{n_m \times d}$$

$$Embed_{position} : \mathbb{Z}_+^{n_m} \rightarrow \mathbb{R}^{n_m \times d}$$

n_m : the sequence length of session m

d : the embedding dimension, we set $d = 128$ in the proposed model

3.1.4 Handling Data Imbalance with Oversampling

A significant class imbalance is present in the datasets, as shown in Table 2. Setting the window size (w) to 20 results in an anomalous session ratio of 9% for BGL and 25% for Thunderbird. Using the block ID, the anomalous session ratio for HDFS is 3%. Given the class imbalances in system logs, it is essential to consider class resampling to achieve unbiased performance in log anomaly detection.

Table 2: The ratio of anomalous sessions of BGL, Thunderbird, and HDFS datasets

Dataset	# of Log messages	# of Log sessions	# Anomalous sessions	Anomaly ratio
BGL (w size=20)	4,747,963	237,389	20,711	0.09
Thunderbird (w size=20)	10,000,000	499,999	122,721	0.25
HDFS	11,175,629	575,061	16,838	0.03

The class resampling techniques can be grouped into three categories: oversampling, undersampling, and hybrid [27]. From a data perspective, they can also be divided into resampling from raw data or resampling from feature space (where raw data is embedded). When resampling is done directly on raw data, the data distribution should be adjusted before feature embedding. Specifically, this involves either duplicating samples of the minority class or selectively removing samples from the majority class in the original log sequences.

The proposed model employs Random Oversampling (ROS) on the raw data. If β is the target proportion of the anomalous class for oversampling in a dataset, and α is the current proportion of the anomalous class with $\alpha < \beta$, the total number of log sessions to classify is N . To reach the proportion of β , the training data will need to be randomly sampled from the training set as an additional anomalous session quantity X , which satisfies $\alpha \times N + X = \beta(N + X)$. The following formula can be used to calculate X .

$$X = \frac{N(\beta - \alpha)}{1 - \beta} \quad (4)$$

For the BGL dataset, the current percentage of the anomalous class is 0.09 (9%) for α . The target percentage of the anomalous class β is set to 0.4 (40%) in the proposed model implementation.

The random oversampling function O applies to session set S to rebalance the minority class S' . Let α be the original minority class ratio, β the expected minority ratio, and S_1 be the set of anomalous sessions.

$$S' = O(S; \alpha, \beta) \text{ where } \alpha = \frac{|S_1|}{|S|} \quad (5)$$

After incorporating the oversampled anomalous samples, the number of anomalous sessions can be calculated as follows:

$$|S_1^{new}| = \left\lceil \frac{\beta(1-\alpha)}{1-\beta} \cdot |S| \right\rceil \quad (6)$$

3.2 Linear Self-Attention Encoder

In log-based anomaly detection, token length increases substantially if the raw log is not parsed to preserve all contextual information, as is done in the proposed model. As shown in Table 1, session token lengths can reach thousands, which significantly intensifies the quadratic computational complexity of traditional Transformers.

3.2.1 Reducing Self-Attention Complexity

The self-attention function maps a query and a collection of key-value pairs to produce the output in the attention mechanism. The queries, keys, values, and outputs are represented as vectors. The output is determined by calculating a weighted sum of the values, where the weights are derived from a compatibility function that evaluates the query against its corresponding key. Scaled dot-product self-attention is illustrated in Figure 7, where the input originates from the same source and is duplicated as Q (Query), K (Key), and V (Value) with the same dimension d . The dot products between

the query and each key are calculated, then each is scaled by the square root of the dimension. Then, the softmax function is applied to produce weights for the values. The output matrix is determined as follows:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V \quad (7)$$

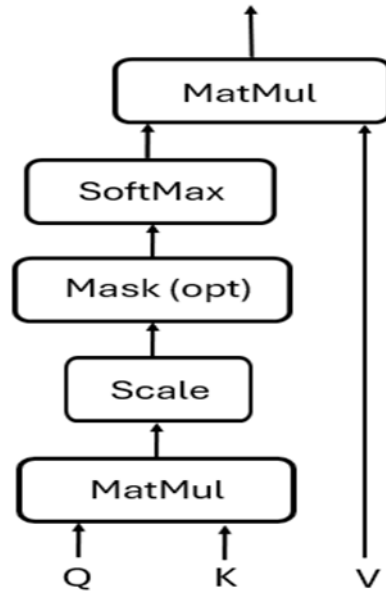


Figure 7. Scaled dot-product self-attention

Instead of applying a single attention function directly to the keys, values, and queries of dimension d_{model} , it is more effective to project the queries, keys, and values separately through h different learned linear projections, mapping them to d_k , d_k , and d_v dimensions, respectively. After these projections, the attention function is applied in parallel to transform queries, keys, and values, generating output values with dimension d_v . These output values are then concatenated and projected once more to produce the result, as illustrated in Figure 8.

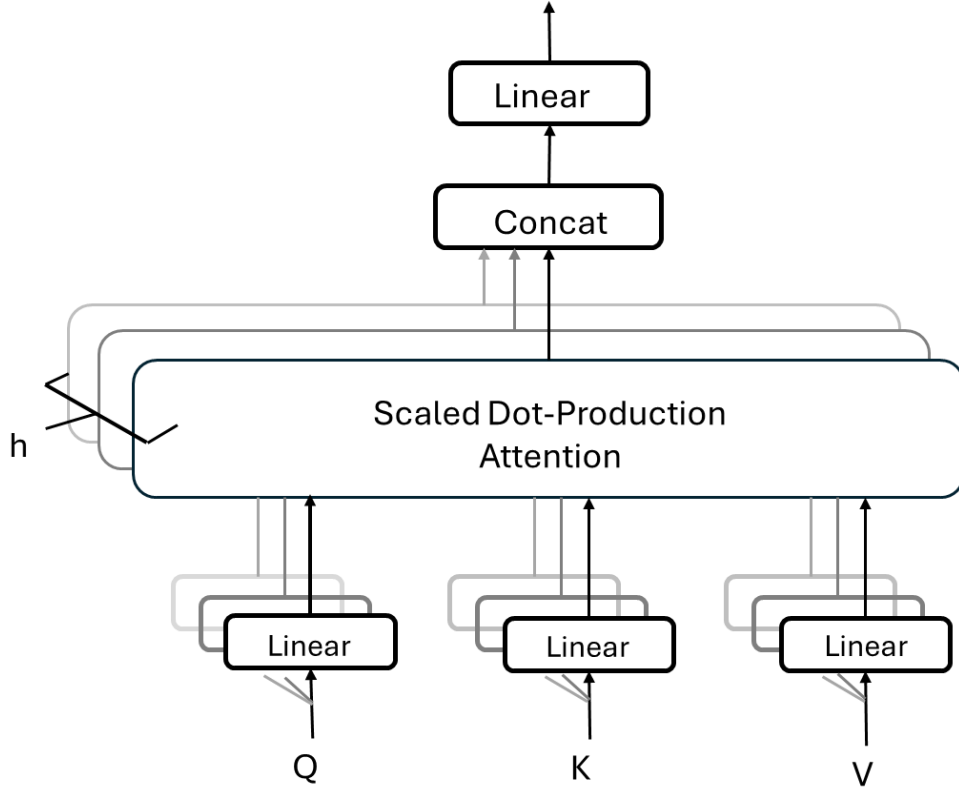


Figure 8. Multi-head attention consists of several attention layers running in parallel.

A multi-head self-attention encoder can simultaneously check different aspects of the input by attending multiple representation subspaces at various positions. In contrast, using a single attention head limits the encoder's expressiveness, as it tends to average out diverse relationships, reducing its ability to capture complex dependencies effectively. This expression describes the multi-head attention:

$$Multihead(Q, K, V) = Concat(head_1, \dots, head_h) W^O \quad (8)$$

$$\text{where } head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$$

The projections are parameter matrices $W_i^Q \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{model} \times d_v}$, and $W^O \in \mathbb{R}^{hd_v \times d_{model}}$

However, the self-attention mechanism requires each token to attend to every other token in the sequence, resulting in $O(n^2)$ complexity in both time and memory space. This becomes a performance bottleneck for long log sequences, as observed in the log

session preparation.

The linear self-attention [30] approach addresses quadratic complexity by leveraging the low-rank nature of the attention matrix. Full-rank self-attention can be accurately approximated using substantially fewer dimensions. Instead of computing full attention over the $n \times n$ space, the linear self-attention introduces two learnable projection matrices, as shown in Figure 9.

$E \in \mathbb{R}^{k \times n}$ used to compress the key sequence

$F \in \mathbb{R}^{k \times n}$ used to compress the value sequence

These matrices reduce the quadratic attention computation to the following linear form:

$$Linear - Attn(Q, K, V) = softmax\left(\frac{Q(EK)^T}{\sqrt{d}}\right)(FV) \quad (9)$$

This reduces the complexity from $O(n^2)$ to $O(nk)$, where $k \ll n$.

The details of getting computational complexity for the standard Transformer and the linear self-attention are illustrated in Table 3, where n is the input sequence length, d is the model embedding dimension, and k is a constant with $k \ll n$.

Table 3: The computational complexity of the transformer and linear self-attention

operation	Transformer complexity	Linear self-attention complexity
Computing QK^T	$O(n^2d)$	$O(nkd)$
Softmax	$O(n^2)$	$O(nk)$
Multiplying by V	$O(n^2d)$	$O(nkd)$
Total Complexity	$O(n^2)$	$O(n)$

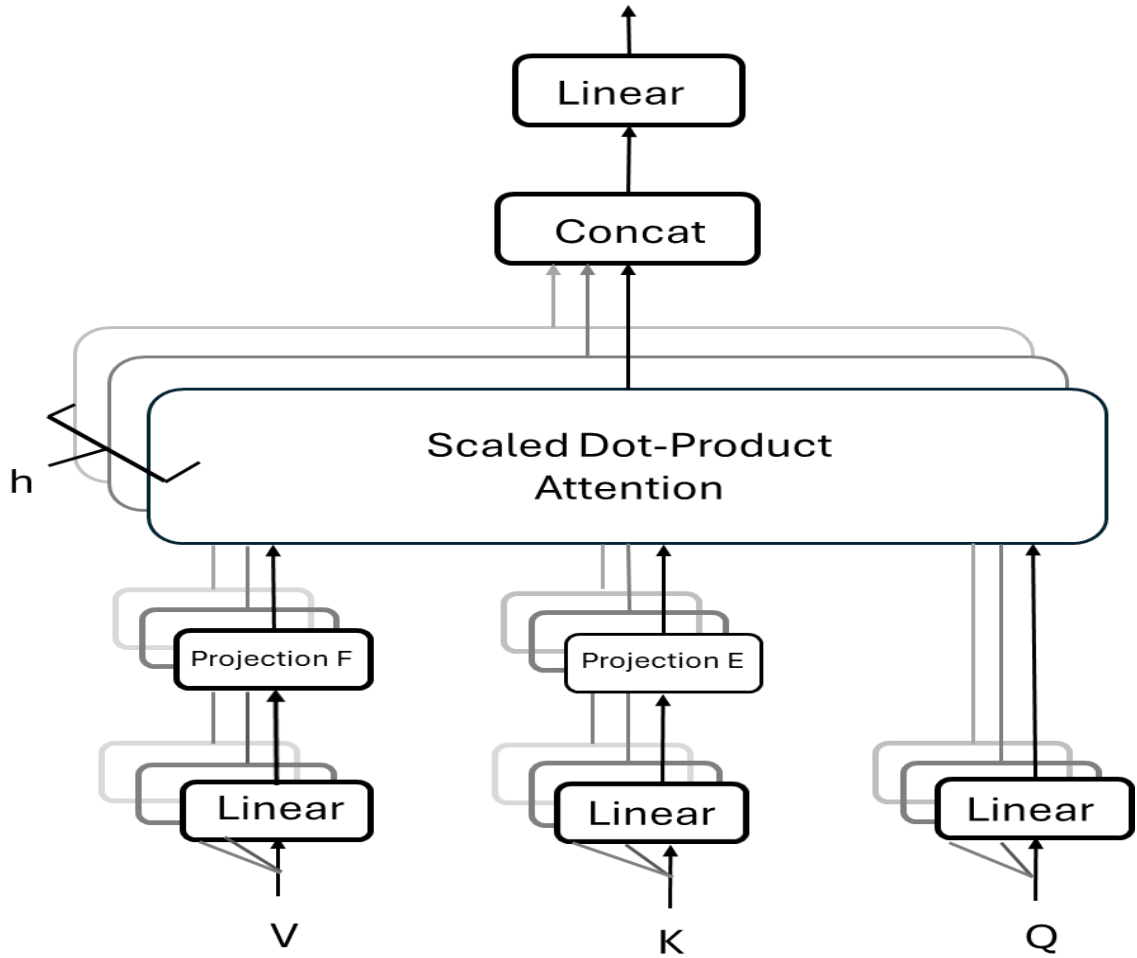


Figure 9. Multi-head self-attention with projection E and F

3.2.2 Efficient Encoder

The proposed model reduces the quadratic computation complexity by using a linear self-attention encoder. Illustrated in Figure 9, it allows the encoder to compute a

contextual attention matrix with linear time and memory complexity related to the sequence length. The encoder projects the original key (KW_i^K) and value (VW_i^V) layers, which are $n \times d$ in dimension, into $k \times d$ dimensional projected key and value layers using learnable matrices E and F . Then, the encoder applies the scaled dot-product and the softmax function to Q ($n \times d$) and K ($k \times d$) to compute the attention score. With this attention score and V ($k \times d$), it obtains the attention matrix for each head as follows:

$$\begin{aligned} head_i &= \text{Attention}(QW_i^Q, E_iKW_i^K, F_iVW_i^V) \\ &= \text{softmax}\left(\frac{QW_i^Q(E_iKW_i^K)^T}{\sqrt{d_k}}\right) \cdot F_iVW_i^V \end{aligned} \quad (10)$$

To make the encoder lighter, the proposed model adopts a layer-wise parameter sharing technique utilizing a single projection matrix E , where $E_i = F_i = E$ across all layers for all heads. This reduces the number of additional parameters in the proposed model, consequently decreasing its memory footprint.

3.3 Classification Layer

The encoder processes the input sequence and produces the output $H \in \mathbb{R}^{n \times d}$, which is then passed to the classification layer to perform anomaly detection. The classifier uses average pooling across the sequence length dimension to get the session-level representation as follows:

$$h_{avg} = \frac{1}{n} \sum_{i=1}^n H_i \quad (11)$$

where $H_i \in \mathbb{R}^d$ is the encoded representation of the i -th token in the sequence. To perform the classification, the pooled representation $h_{avg} \in \mathbb{R}^d$ is then passed through a linear layer:

$$z = W \cdot h_{avg} + b \quad (12)$$

Where $W \in \mathbb{R}^{2 \times d}$ and $b \in \mathbb{R}^2$ are all learnable parameters, and $z \in \mathbb{R}^2$ represents the unnormalized logits for the two classes: regular and anomalous.

4. EXPERIMENTS

A series of experiments is conducted to evaluate the performance of the proposed model. First, the experimental setup will be discussed, and the proposed model will be compared with other baseline models.

4.1 Experimental Setup

The experiments are conducted for the proposed model with an Intel Core i9-13900HX 2.20 GHz CPU with 32 GB of RAM and an NVIDIA GeForce RTX 4080 12 GB GPU. The operating system is Ubuntu 24.04 LTS, and the PyTorch version is 2.5.1.

For the proposed model's configuration, the minority class oversampling ratio is set to 0.4. The k value for linear self-attention is set to 32. The embedding dimension is set to 128. To retain all feature information, the input sequence is not truncated. The maximum sequence length of the proposed model is set to the maximum length encountered during the construction of input sessions.

4.1.1 Loss Function & Class Weighting

The proposed model adopts a weighted cross-entropy loss function to address the class imbalance commonly observed in log datasets. During training, the weighted loss function assigns a higher penalty to misclassified anomalous instances, helping to prevent bias toward the regular class. Let $\mathbf{z} = [z_0, z_1] \in \mathbb{R}^2$ represent the logit vector output by the classification layer, $y \in \{0,1\}$ be the ground truth label, and the class weight vector $\mathbf{w} = [w_0, w_1]$. The loss for a single training sample is defined as:

$$L(\mathbf{z}, y) = -w_y \cdot \log \left(\frac{e^{z_y}}{\sum_{j=0}^1 e^{z_j}} \right) = -w_y \cdot \left(z_y - \log \sum_{j=0}^1 e^{z_j} \right) \quad (13)$$

, where w_y denotes the weight assigned to the class.

In the experiment, the weight $w_0 = 1.0$ is assigned to the regular class and the weight $w_l = 4.0$ is assigned to the anomalous class. This approach suggests that the model focuses more on the anomalous class to reduce the bias caused by the skewed distribution of the regular class in the training set.

4.1.2 AdamW Optimizer & Learning Rate Scheduling

The proposed model exploits the AdamW optimizer [33], which decouples the weight decay term from the gradient updating to ensure consistent and effective regularization. The optimizer is configured with a weight decay of 10^{-2} and an epsilon of 10^{-8} . The initial learning rate is 5×10^{-4} and scheduled by ReduceLROnPlateau scheduler with the factor set to 0.1

4.2 Benchmark Datasets

4.2.1 Public Log Datasets and Dataset Split

In our experiments, open-source public log datasets are adopted to evaluate the performance of the proposed model and compare it with baseline models. The adopted datasets are BGL and HDFS [31]. Table 1 summarizes these datasets. The BGL dataset is a publicly available collection of logs from a Blue Gene/L supercomputer system at Lawrence Livermore National Laboratory (LLNL) in Livermore, California. The HDFS (Hadoop Distributed File System) dataset is publicly available and was generated by running Hadoop-based MapReduce jobs on over 200 Amazon EC2 nodes.

The dataset is split into training (70%), validation (15%), and test (15%) sets. The training set was used to fit the model, the validation set was used for hyperparameter tuning and early stopping, and final performance was reported on the unseen test set to estimate generalization.

4.3 Evaluations

4.3.1 Evaluation metrics

To evaluate the performance of models, widely adopted Precision, Recall, and F1-score metrics are used in this paper. Let True Positive (TP) be the number of anomalies that the model correctly detected. False Positive (FP) represents the number of regular sessions that are mistakenly identified as anomalous ones. True Negative (TN) refers to the number of regular sessions that the model correctly identified as regular ones. False Negative (FN) is the number of anomalous sessions that the model does not recognize.

Precision denotes the percentage of correctly predicted anomalies over all anomalous predictions. $Precision = \frac{TP}{TP + FP}$

Recall represents the percentage of correctly predicted anomalies over all real anomalies. $Recall = \frac{TP}{TP + FN}$

F1-score shows the harmonic mean of the Precision and the Recall.

$$F1-score = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

4.3.2 Comparison with Baseline Models

The performance metrics of the proposed model are compared with those of the following models against HDFS and BGL public datasets, shown in Table 4. Those results, except AllLinLog, are cited from FSMFLog [22]. DeepLog [17], LogCluster [10], LogAnomaly [34], SwissLog [35], PLElog [36], and FSMFLog [22] all employ a log parser to transform unstructured system logs to structured data. AllLinLog is the only one that does not utilize log parsing. The proposed model achieves the highest F1 Score on both the HDFS and BGL datasets.

Table 4: Comparison of various methods on public datasets

Methods	HDFS			BGL		
	Precision	Recall	F1-score	Precision	Recall	F1-score
DeepLog	0.879	0.812	0.844	0.823	0.807	0.815
LogCluster	0.851	0.745	0.794	0.578	0.829	0.681
LogAnomaly	0.965	0.834	0.895	0.859	0.912	0.885
SwissLog	0.960	0.885	0.921	0.941	0.852	0.894
PLELog	0.923	0.734	0.818	0.911	0.811	0.858
FSMFLog	0.981	0.952	0.966	0.972	0.951	0.961
AllLinLog	0.997	0.995	0.996	0.999	0.999	0.999

4.3.3 Ablation Study of AllLinLog

To evaluate the effectiveness and efficiency of the proposed model, an ablation study is conducted on the BGL dataset, focusing on three architecture configurations: the low-rank approximation factor k , the number of linear self-attention encoder layers, and a comparison with the standard transformer encoder in terms of detection performance and inference time. All evaluations are performed using the same data preprocessing pipeline, loss function configuration, and optimizer settings mentioned in section 4.1.

(A) Effect of Low Rank Approximation Factor k

The linear self-attention encoder replaces full self-attention with a low-rank approximation to address the quadratic complexity of computation time and storage requirements. To assess the optimal value for k , anomaly detection is performed on BGL dataset with k in range of $\{4, 8, 16, 32, 64, 128\}$ while keeping all other configurations of model as constants which include the model embedding dimension set as 128, the layer of linear self-attention encoder set as one and the number of heads set as 4. The result is shown as Figure 10, which reveals that the optimal value of k is 32, with an F1-score of 0.99888.

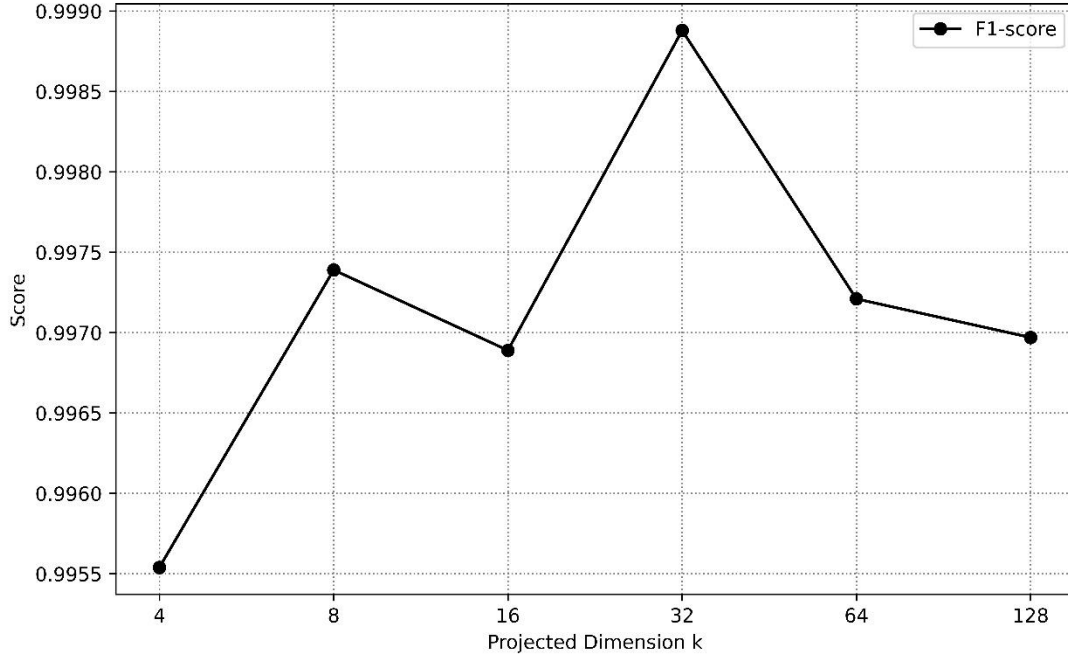


Figure 10. The effect of different low-rank approximation factor k

(B) Effect of Linear Self-Attention Layer Depth

The proposed model can stack multiple linear self-attention encoder layers as a complex anomaly detection framework. To find the optimal configuration for log-based anomaly detection tasks, the proposed model is evaluated with different layer configurations against the BGL dataset. The evaluation begins with a single encoder layer comprising four attention heads, and subsequently increases to four encoder layers with 32 heads, while maintaining other model parameters constant. Based on the results depicted in Figure 11, the optimal depth of the encoder layer for the proposed model is determined to be one, with an F1-score of 0.99888.

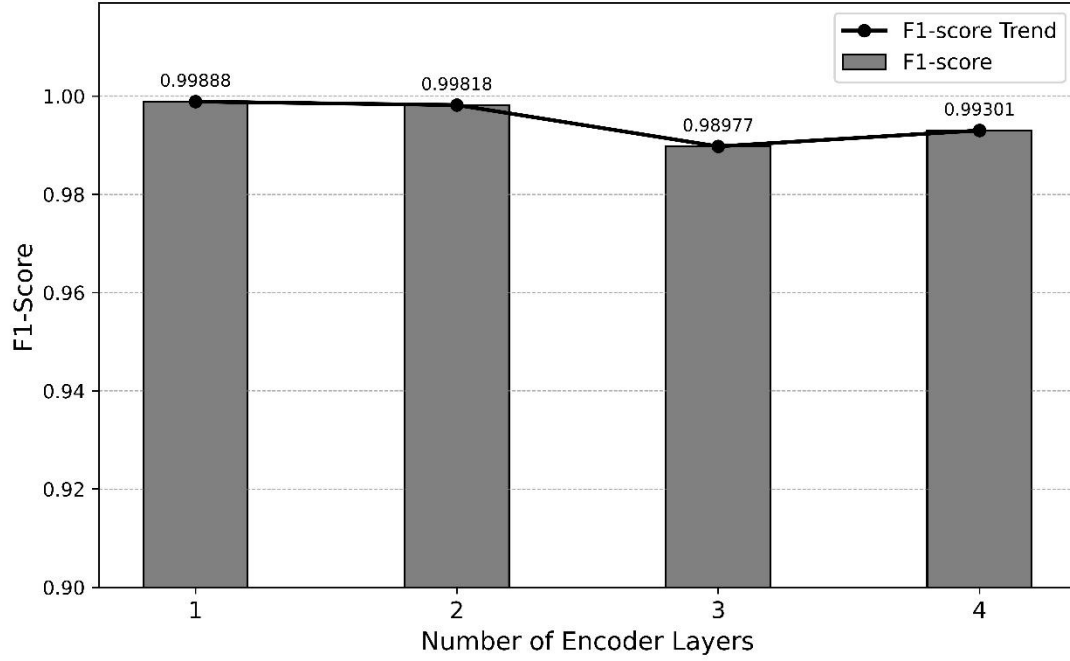


Figure 11. Impact of encoder layer depth on F1-score

(C) Comparison with Standard Transformer Encoder

The proposed model employs a linear self-attention encoder to reduce the computational complexity of full self-attention from quadratic to linear complexity by incorporating a low-rank approximation. To evaluate the cost incurred by the low-rank approximation, the encoder is replaced with a standard Transformer encoder, and the detection performance is compared between the two different encoders against the BGL dataset. From the results shown in Figure 12, the linear self-attention encoder decreases computational complexity at the cost of a 0.079% drop in F1-score compared to the standard Transformer encoder (drop from 0.99967 to 0.99888).

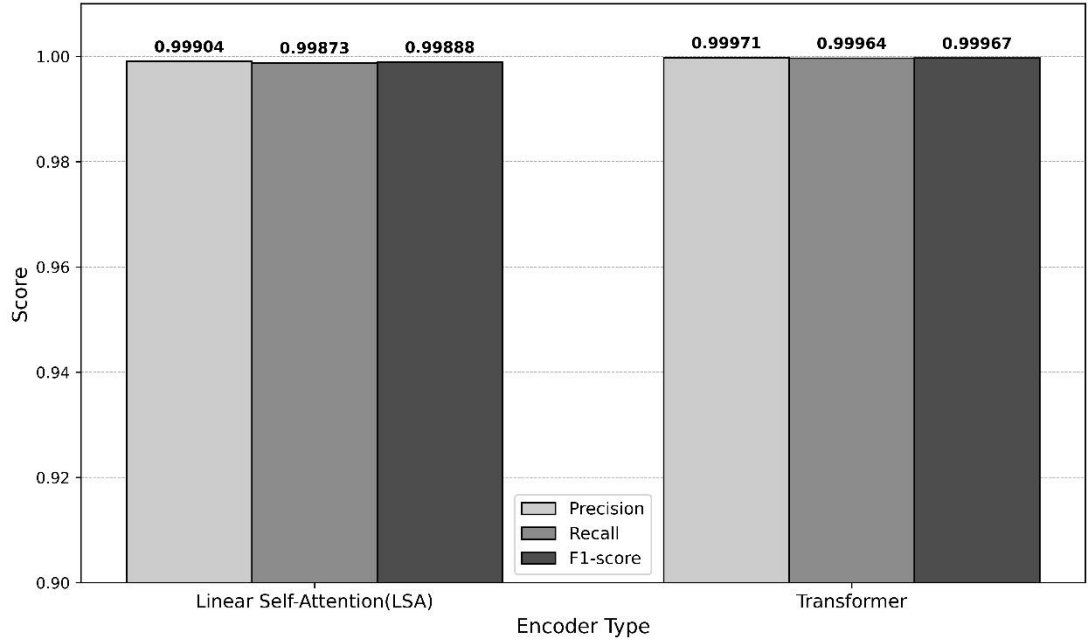


Figure 12. Comparison of anomaly detection performance between the linear self-attention encoder and the transformer encoder

On the other hand, as Table 5 illustrates, the linear self-attention encoder is faster and more efficient than the Transformer encoder. It achieves significant complexity reductions in both inference time and memory consumption. This study begins with a batch size of one for real-time anomaly detection, then increases the batch size to 16 to simulate devices collecting more log sessions and performing anomaly detection in batches. For measuring memory usage, `torch.cuda.max_memory_allocated` is used to determine the peak memory allocation. This is because PyTorch, by design, pre-allocates and preserves memory during operation, so we can't measure the delta values of memory before and after inference. For the inference time benchmark, the inference is executed 100 times, and then the average time is derived.

Table 5: Benchmark analysis: transformer encoder vs. linear self-attention (LSA) encoder (Batch size 1 & 16)

Encoder	Batch size	Peak Memory used	Avg Inference Time	Throughput
Transformer	1	893.22 MB	4.7541 ms	210.35 samples/sec
LSA	1	709.09 MB	0.8143 ms	1228.07 samples/sec
Transformer	16	3824.70 MB	51.2782 ms	312.02 samples/sec
LSA	16	911.70 MB	3.9677 ms	4032.52 samples/sec

From Table 6, it can be observed that the linear self-attention encoder outperforms the Transformer encoder in memory efficiency. It can also be observed that the fact described in Table 7 is that the linear self-attention encoder scales inference time much more efficiently than the Transformer encoder, and the Transformer encoder’s inference time increases by approximately 10.8 times when the batch size is increased to 16. The linear self-attention encoder’s inference time only grows by about 4.9 times, indicating that it handles large batches more efficiently. Furthermore, Table 8 shows that the linear self-attention encoder achieves 12.9 times the throughput of the Transformer encoder at a batch size of 16. These results indicate that the Transformer encoder struggles with large batches due to its quadratic computational complexity of self-attention. In contrast, the linear self-attention encoder demonstrates significantly higher efficiency due to its linear computational complexity.

Table 6: Memory efficiency comparison between transformer encoder and linear self-attention (LSA) encoder

Encoder	Batch Size	Memory Growth (1 $\rightarrow$ 16)
Transformer	1 $\rightarrow$ 16	893.22 MB $\rightarrow$ 3824.70 MB (+2,931.48 MB)
LSA	1 $\rightarrow$ 16	709.09 MB $\rightarrow$ 911.70 MB (+202.61 MB)

Table 7: Inference time efficiency comparison between transformer encoder and linear self-attention (LSA) encoder

Encoder	Batch Size	Inference Time Increase (1 $\rightarrow$ 16)
Transformer	1 $\rightarrow$ 16	4.7541 ms $\rightarrow$ 51.2782ms (10.8x increase)
LSA	1 $\rightarrow$ 16	0.8143 ms $\rightarrow$ 3.9677ms (4.9x increase)

Table 8: Throughput comparison between transformer encoder and linear self-attention (LSA) encoder

Encoder	Batch Size	Throughput (samples/sec)
Transformer	1 $\rightarrow$ 16	210.25 $\rightarrow$ 312.02
LSA	1 $\rightarrow$ 16	1228.07 $\rightarrow$ 4032.52

5. CONCLUSION

AllLinLog is a streamlined log-based anomaly detection model that leverages linear-complexity self-attention to reduce computational overhead when processing long log sessions. Not only does the proposed model resolve the out-of-vocabulary issue, but it also eliminates the need for an imperfect and complex log parser by leveraging the GPT-4 BPE tokenizer. Unlike other research that discards dynamic parameter features in favor of semantic features or truncates input log tokens to meet the computational complexity constraints of the detection engine, the proposed model preserves all the contextual information without any loss or truncation of input log messages by integrating a linear self-attention encoder. To mitigate bias in the class-imbalanced public datasets, the proposed model applies random oversampling during dataset preparation. With all these enhancements, the proposed model outperforms the existing log-parsed log-based anomaly detection models in experimental evaluations against public datasets.

Although the BPE tokenizer can resolve the OOV issue, it will tokenize unfamiliar technical terms or parameters (e.g., IP addresses) into sub-words. This process can distort important contextual information of tokens and increase input sequence length. The next step in the proposed method is to develop a byte-level tokenization methodology that can learn and segment log messages without breaking parameters and unfamiliar technical terms into sub-words, thereby better preserving all features of the system logs.

REFERENCE

1. Roku Protecting your Roku account. In: Roku Blog.
<https://www.roku.com/blog/protecting-your-roku-account>. Accessed 28 Feb 2025
2. Shu X, Tian K, Ciambone A, Yao D (2017) Breaking the Target: An Analysis of Target Data Breach and Lessons Learned
3. Zhu J, He S, Liu J, et al (2019) Tools and Benchmarks for Automated Log Parsing. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). pp 121–130
4. Thaler S, Menkonvski V, Petkovic M (2017) Towards a neural language model for signature extraction from forensic logs. In: 2017 5th International Symposium on Digital Forensic and Security (ISDFS). pp 1–6
5. Jiang ZM, Hassan AE, Flora P, Hamann G (2008) Abstracting Execution Logs to Execution Events for Enterprise Applications (Short Paper). In: 2008 The Eighth International Conference on Quality Software. pp 181–186
6. He P, Zhu J, Zheng Z, Lyu MR (2017) Drain: An Online Log Parsing Approach with Fixed Depth Tree. In: 2017 IEEE International Conference on Web Services (ICWS). pp 33–40
7. He P, Zhu J, Xu P, et al (2018) A Directed Acyclic Graph Approach to Online Log Parsing
8. Nagappan M, Vouk MA (2010) Abstracting log lines to log event types for mining software system logs. In: 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010). pp 114–117
9. Dai H, Li H, Chen C-S, et al (2022) Logram: Efficient Log Parsing Using nn-Gram Dictionaries. *IEEE Transactions on Software Engineering* 48:879–892.
<https://doi.org/10.1109/TSE.2020.3007554>
10. Vaarandi R, Pihelgas M (2015) LogCluster - A data clustering and pattern mining algorithm for event logs. In: 2015 11th International Conference on Network and Service Management (CNSM). pp 1–7
11. Du M, Li F (2016) Spell: Streaming Parsing of System Event Logs. In: 2016

IEEE 16th International Conference on Data Mining (ICDM). pp 859–864

12. Le V-H, Zhang H (2021) Log-based Anomaly Detection Without Log Parsing. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp 492–504
13. Fu Y, Yan M, Xu Z, et al (2022) An empirical study of the impact of log parsers on the performance of log-based anomaly detection. *Empir Software Eng* 28:6. <https://doi.org/10.1007/s10664-022-10214-6>
14. Chen M, Zheng AX, Lloyd J, et al (2004) Failure diagnosis using decision trees. In: International Conference on Autonomic Computing, 2004. Proceedings. pp 36–43
15. Breier J, Branišová J (2015) Anomaly Detection from Log Files Using Data Mining Techniques. In: Kim KJ (ed) *Information Science and Applications*. Springer, Berlin, Heidelberg, pp 449–457
16. Lin Q, Zhang H, Lou J-G, et al (2016) Log clustering based problem identification for online service systems. In: Proceedings of the 38th International Conference on Software Engineering Companion. Association for Computing Machinery, New York, NY, USA, pp 102–111
17. Du M, Li F, Zheng G, Srikumar V (2017) Anomaly Detection from Log Files Using Data Mining Techniques. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. Association for Computing Machinery, New York, NY, USA, pp 1285–1298
18. Lu S, Wei X, Li Y, Wang L (2018) Detecting Anomaly in Big Data System Logs Using Convolutional Neural Network. In: 2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech). pp 151–158
19. Vaswani A, Shazeer N, Parmar N, et al (2017) Attention is All you Need. In: *Advances in Neural Information Processing Systems*. Curran Associates, Inc.
20. Guo H, Yuan S, Wu X (2021) LogBERT: Log Anomaly Detection via BERT. In: 2021 International Joint Conference on Neural Networks (IJCNN). pp 1–8
21. Qi J, Huang S, Luan Z, et al (2023) LogGPT: Exploring ChatGPT for Log-Based

- Anomaly Detection. In: 2023 IEEE International Conference on High Performance Computing & Communications, Data Science & Systems, Smart City & Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys). pp 273–280
22. Niu W, Li Z, He Z, et al (2024) FSMFLog: Discovering Anomalous Logs Combining Full Semantic Information and Multifeature Fusion. *IEEE Internet of Things Journal* 11:4442–4453. <https://doi.org/10.1109/JIOT.2023.3300690>
 23. Guo H, Yang J, Liu J, et al (2024) LogFormer: a pre-train and tuning pipeline for log anomaly detection. In: *Proceedings of the Thirty-Eighth AAAI Conference on Artificial Intelligence and Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence and Fourteenth Symposium on Educational Advances in Artificial Intelligence*. AAAI Press, pp 135–143
 24. Khan ZA, Shin D, Bianculli D, Briand LC (2024) Impact of log parsing on deep learning-based anomaly detection. *Empir Software Eng* 29:139. <https://doi.org/10.1007/s10664-024-10533-w>
 25. Guan W, Cao J, Qian S, et al (2025) LogLLM: Log-based Anomaly Detection Using Large Language Models
 26. Xiao R, Li W, Lu J, Jin S (2024) ContextLog: Non-Parsing Log Anomaly Detection With All Information Preservation and Enhanced Contextual Representation. *IEEE Transactions on Network and Service Management* 21:4750–4762. <https://doi.org/10.1109/TNSM.2024.3400283>
 27. Ma X, Zou H, He P, et al (2025) On the Influence of Data Resampling for Deep Learning-Based Log Anomaly Detection: Insights and Recommendations. *IEEE Transactions on Software Engineering* 51:243–261. <https://doi.org/10.1109/TSE.2024.3513413>
 28. Child R, Gray S, Radford A, Sutskever I (2019) Generating Long Sequences with Sparse Transformers
 29. Kitaev N, Kaiser Ł, Levskaya A (2020) Reformer: The Efficient Transformer
 30. Wang S, Li BZ, Khabsa M, et al (2020) Linformer: Self-Attention with Linear Complexity
 31. Zhu J, He S, He P, et al (2023) Loghub: A Large Collection of System Log Datasets for AI-driven Log Analytics. In: *2023 IEEE 34th International*

Symposium on Software Reliability Engineering (ISSRE). pp 355–366

32. Sennrich R, Haddow B, Birch A (2016) Neural Machine Translation of Rare Words with Subword Units
33. Loshchilov I, Hutter F (2019) Decoupled Weight Decay Regularization
34. Meng W, Liu Y, Zhu Y, et al (2019) LogAnomaly: Unsupervised Detection of Sequential and Quantitative Anomalies in Unstructured Logs. In: Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence. International Joint Conferences on Artificial Intelligence Organization, Macao, China, pp 4739–4745
35. Li X, Chen P, Jing L, et al (2020) SwissLog: Robust and Unified Deep Learning Based Log Anomaly Detection for Diverse Faults. In: 2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE). pp 92–103
36. Yang L, Chen J, Wang Z, et al (2021) PLELog: Semi-Supervised Log-Based Anomaly Detection via Probabilistic Label Estimation. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). pp 230–231