# SwissLog: Robust and Unified Deep Learning Based Log Anomaly Detection for Diverse Faults

Xiaoyun Li, Pengfei Chen*, Linxiao Jing, Zilong He and Guangba Yu
School of Data and Computer Science, Sun Yat-sen University
Guangzhou, China
Email: {lixy223, jinglx3, hezlong, yugb5}.mail2.sysu.edu.cn, *chenpf7.mail.sysu.edu.cn

***Abstract*—Log-based anomaly detection has been widely studied and achieves a satisfying performance on stable log data. But, the existing approaches still fall short meeting these challenges: 1) Log formats are changing continually in practice in those software systems under active development and maintenance. 2) Performance issues are latent causes that may not be detected by trivial monitoring tools. We thus propose SwissLog, namely a robust and unified deep learning based anomaly detection model for detecting diverse faults. SwissLog targets at those faults resulting in log sequence order changes and log time interval changes. To achieve that, an advanced log parser is introduced. Moreover, the semantic embedding and the time embedding approaches are combined to train a unified attention based Bi-LSTM model to detect anomalies. The experiments on real-world datasets and synthetic datasets show that SwissLog is robust to the changing log data and effective for diverse faults.**

***Keywords*-deep learning; log parsing; anomaly detection; BERT**

## I. Introduction

For large-scale software systems, especially those deployed on cloud servers, it is vital to enhance system health and stability. Both external faults (e.g., malicious attack, node disconnection) and internal software bugs (e.g., an infinite loop, incorrect configuration) may deliver to unexpected system aborts. All of these failures are regarded as anomaly. A large-scale halt of cloud servers can lead to the failure of downstream services, customers drain, and even huge economic loss. Take an anomaly in cloud server for example. During an upgrade, a snippet of error code caused I/O hang in many running instances. Millions of services, especially e-commerce services and financial services built on top of cloud servers, suffered huge economics loss from this anomaly [1]. Anomaly detection is therefore required to alarm immediately and mitigate the impact of an anomaly.

Log data is an extensively available data resource that records system states and critical events at runtime in all kinds of software systems. Developers generally utilize log data to obtain the system status, detect anomaly and locate root causes. The hidden abundant information offers a good view to analyze system problems. Hence by mining log information in a large amount of log data, data-driven methods can help to enhance system health, stability, and availability. As the scale and complexity of modern computer systems increase, log data is generated in explosion. For example, there are more than 50 GB of logs generated per hour [2]. It is a crucial challenge to process such a large amount of log data. Instead of error-prone and time-consuming manual work, an effective and efficient data-driven log processing tool is an urgent need.

There are a large body of data-driven methods that automatically detect anomalies. Principal Component Analysis (PCA) based methods [3], Invariant Mining-based (IM) methods [4], and workflow-based methods [5] are typical automated algorithms to detect anomaly based on log data. With the prevalence of deep learning, deep learning-based methods are gradually applied to anomaly detection such as DeepLog [6], LogAnomaly [7], LogRobust [8]. They present remarkable results than previous methods in anomaly detection.

But, the existing approaches are built based on some strong assumptions which are not satisfied in the real-world production environment. There are two major challenges when applying the methods mentioned above in the production environment. 1) Changing logs: log formats are changing constantly in practice in those software systems under active development and maintenance. Kabinna, et al. [9] and Zhang, et al. [8] discussed log instability in their prior works. The empirical study shows that there are around 20-45% logs changed throughout the software system lifetime. 2) Underlying performance issues: performance issues are the common manifestation of partial failures [10], which refers that partial functionalities are broken, but not all of them. Indeed, partial failures are behind many real-world outages [1], [11]–[14], hence not latent problems that developers can ignore. Consequently, we can dig up partial failures through detecting performance issues.

To address the above challenges, we propose SwissLog, a robust and unified deep learning based log anomaly detection for diverse faults. SwissLog is robust and versatile like Swiss Army Knife. From real-world log data, we find two common types of log changes when an anomaly occurs. In this paper, we name those faults resulting in sequence order changes as sequential log anomalies, and time interval changes as performance issues, respectively.

SwissLog consists of four stages: log parsing, sentence embedding, and Attention-based Bidirectional Long Short-Term Model (Attn-based Bi-LSTM), and anomaly detection. SwissLog introduces a novel log parsing approach that extracts log data by mapping valid words in a dictionary without losing the semantic meaning of sentences. To detect performance issues, we additionally combine temporal information, namely the time interval between log statements, with semantic information in log data. SwissLog employs Bidirectional

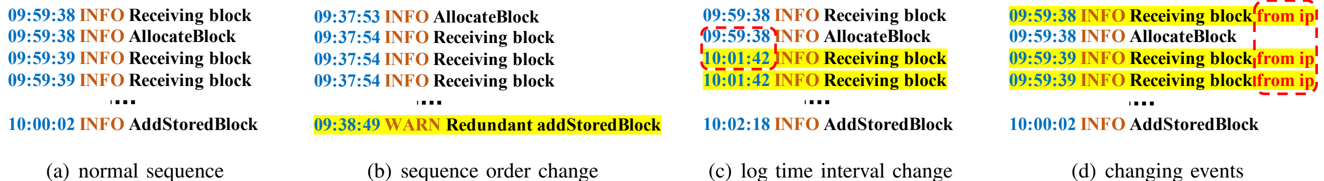



Fig. 1. Four types of log sequences with different changes.

Encoder Representation from Transformers (BERT) [15] to encode semantic information, namely log templates. Then SwissLog utilizes a novel time embedding method to encode temporal information. Next, Attn-based Bi-LSTM receives the concatenation of semantic embedding and time embedding and learns the fixed pattern of log data. Finally, an alarm occurs if detecting an anomaly. We conduct experiments on real-world log datasets and synthetic datasets to evaluate the effectiveness and robustness of SwissLog for detecting diverse faults.

The contributions of this paper are four-fold:

- We propose a novel approach that parses log messages based on a dictionary. Particularly, it does not require any parameter tuning process. To our best knowledge, we are the first to propose a log parsing method based on the dictionary.
- We introduce BERT to encode log templates which is robust to changing log formats.
- We combine time embedding and semantic embedding approaches to detect sequential log anomalies and performance issues by a unified deep learning model. The performance issues are rarely studied in log analysis before.
- We implement SwissLog and evaluate it on real-world datasets and synthetic datasets. The results prove the effectiveness and robustness of SwissLog for detecting diverse faults.

The remainder of this paper is organized as follows. Sec. II illustrates the motivation. Sec. III shows an overview and details of SwissLog. We present our evaluation results and discussions in Sec. IV and Sec. V. The related work is summarized in Sec. VI. Finally, we conclude this paper in Sec. VII.

## II. Motivation

Log data is a substantially available data source recording system states and significant events at runtime. It is intuitive to observe system status and inspect potential anomaly. One of the normal sequences is shown in Fig. 1(a). We can observe that the beginning of a normal sequence is to allocate and receive blocks. After a series of operations, this block is eventually added to the set of stored blocks, which means the end of a cycle.

With the increasing complexity and scale of distributed systems, complicated log data and various types of the anomalies are constantly coming out. In this section, we show out the observation from log data in a real-world production environment and analyze the requirements of a robust anomaly detection under a large-scale production environment.

### A. Diverse Faults

A large-scale system inevitably encounters faults, resulting in log pattern changes. We target at two types of log changes in practice, as shown in Fig. 1(b) and Fig.1(c). We omit some unimportant log statements and only show the key information (i.e. time, verbosity level, simplified log statement).

*a) Sequence order change:* An abnormal sequence against the normal one in Fig. 1(a) is depicted in Fig. 1(b), where the abnormal log statement is highlighted in yellow. In this case, the system received a redundant *addStoredBlock* request, causing the sequence order change. Therefore, sequential log anomalies can be generally observed from its abnormal sequence order. Prior works [3], [4], [6]–[8], [16] mostly focus on sequential log anomalies and recognize them by detecting abnormal log sequence order .

*b) Log time interval change:* Another kind of fault is performance issues, whose example is shown in Fig. 1(c). In contrast to abnormal sequence order, those blocks with performance issues usually keep the same sequence order as the normal one. However, performance issues slow down the execution time of specific tasks according to their faulty components. For example, the receiving block in line 3 has a 3000-millisecond latency which is caused by the network congestion. The performance issue here is manifested in the time interval change. Such performance issues are like buried land mines that may trigger catastrophic outages. Therefore, the topic of detecting performance issues gains lots of attention recently [10], [17]–[20]. But the existing approaches employ a static analysis to find performance bugs or an intrusive method to detect them. Either they are difficult to detect performance issues at runtime or they slow down system performance. If we detect performance issues by mining time interval changes in log data, the above problems are accordingly solved.

### B. Changing Events

Modern software systems that need an automated log processing tool are probably under active developments and maintenance. Kabinna, et al. [9] examined the stability of logging statements via empirical study. They find that 20-45% of the logging statements change throughout the whole lifetime. Zhang, et al. [8] also conducted a similar empirical study on Microsoft Service X. As reported, up to 30.3% logs are changed in the latest version. Two main possible factors that cause log changes are: 1) Developers add new log statements to source codes. 2) Developers add a few new features and modify the content of log statements. Extra words are thus attached to log data while not changing its meaning. Fig. 1(d) shows a common case of changing events.

String "from ip" is added to the log statement while it keeps the original meaning. The state of the art method to detect anomaly in changing events is LogRobust [8].

On the basis of the observation in a real-world production environment, we propose SwissLog, a robust and unified deep learning based log anomaly detection model for diverse faults. It can detect sequence order change and log time interval change manifested in log data. Also, it is robust to changing events.

## III. Design of SwissLog

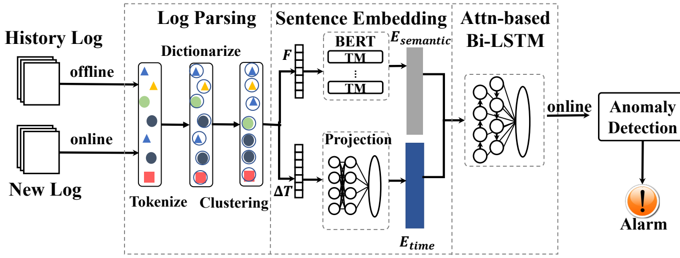



Fig. 2. The overview of SwissLog

We first begin with an overview of SwissLog which is presented in Fig. 2. SwissLog comprises two phases, namely the offline processing phase and the online processing phase. Each phase includes log parsing, sentence embedding, Attn-based Bi-LSTM stage and the online phase particularly contains anomaly detection stage. Firstly, SwissLog adopts a novel log parsing method and extracts multiple templates by tokenizing, dictionarizing, and clustering history log data. These templates are kept as natural sentences instead of event ids. We link those log statements with the same identifiers or simply use a sliding window to construct log sequences named "sessions". And then the log sequence is transformed into semantic information and temporal information. SwissLog uses BERT encoder to encode semantic information $\boldsymbol{F}$ into embedding $\boldsymbol{E}_{context}$ and projects temporal information $\Delta \boldsymbol{T}$ onto embedding $\boldsymbol{E}_{time}$. The concatenation of semantic embedding $\boldsymbol{E}_{semantic}$ and time embedding $\boldsymbol{E}_{time}$ as input is fed into Attn-based Bi-LSTM to learn the features of normal, abnormal and performance-anomalous log sequence. At runtime, the online phase also executes the same workflow as the offline phase. Finally, the pre-trained SwissLog model predicts if a log sequence is an anomaly or not. An alarm will be raised once an anomaly is detected.

We next introduce four stages of SwissLog including log parsing (Sec. III-A), sentence embedding (Sec. III-B), Attn-based Bi-LSTM (Sec. III-C), anomaly detection (Sec. III-D) in detail.

### A. Log Parsing

In this part, we briefly introduce the design of our log parser. Log statements in Fig. 3 are readable because most of the words in it are valid words, which can be looked up in a dictionary. Due to the reading ability of the human brain, most similar logs can be visually split into the same group. Leveraging this feature, a dictionary-based approach naturally addresses the log parsing issue. In the following parts, we illustrate the log parser in SwissLog step by step.

*1) Step 1: Tokenize and Preprocess using Delimiters:* In each log statement $e$, we define a slice of log statement as *token*. How to tokenize a complete log statement into appropriate *tokens* is a critical problem in the dictionary-based approach since the parsing result largely depends on it. The logging system is more likely to use special delimiters such as colon, and quotation marks to separate strings. For better tokenization, we thus utilize five special delimiters, namely {, . ; : "} attained from empirical study, to tokenize log statements.

Given a dictionary $D = \{w_1, w_2, ..., w_n\}$, such that every word $w_i$ can be identified as a valid word. After tokenizing, we first check that if tokens of log statement $e$ are in dictionary $D$. Then we get the *wordset*, a multiset of valid words $dword = \{d_1, d_2, ..., d_m\}$, where $\forall d_i \in D$. An example is shown in Fig. 3 Step 1. When a raw log message "Received block blk_560063894682806537 of size 67108864 from /10.251.194.129" arrives, it will be separated into 11 *tokens*. After searching in the dictionary, *'Received', 'block', 'of', 'size', 'from'* are identified as valid words. Particularly for log-specific concatenated words like "PowerDown", we import an external package *wordninja* [21] to split it into "Power" and "Down" based on the unigram frequencies in English Wikipedia. Finally, we obtain the *wordset* $dword$ containing valid words.

*2) Step 2: Cluster Logs by Wordset:* The goal of this step is to cluster similar log statements with the same *wordset*. When a new *wordset* $dword$ arrives, SwissLog looks for the matched group for it. If a group is matched, SwissLog puts *wordset* $dword$ into it. Otherwise, SwissLog creates a new cluster for *wordset* $dword$. Assume that $dword_1$, $dword_2$, $dword_3$ are the *wordset* of log statements $e_1$, $e_2$, $e_3$, respectively. Since the log statement $e_1$ and $e_2$ have different *wordset*, SwissLog creates the new cluster $C_1$ and $C_2$ for them separately. Observed that *wordset* $dword_3$ is identical with $dword_1$, log statement $e_3$ is consequently categorized into cluster $C_1$.

Sometimes, a valid word occurs multiple times in one log statement. For example, "120 bytes sent, 80 bytes received". The word `bytes` occurs twice in this log statement, which is easily confused with those log statements with only one `bytes`. Taking the word occurrence into account, we especially use count set $fword$ to store *wordset* occurrence. Hence, only when the *wordset* $dword$ and occurrence $fword$ are completely identical, can the two log statements be categorized into the same cluster.

*3) Step 3: Mask Variable with LCS:* The goal of the masking layer is to distinguish the constant part and variable part in a cluster. Common sequence of log statements in the same cluster can be regarded as a constant part, while the changing part can be viewed as the variable part. Next, we introduce token-level Longest Common Sequence (LCS) to help us mask all variable parts in a cluster with *.

LCS is to find the longest sequence among a sequence set. An LCS example is shown in Step 3 of Fig. 3. Assume there are four log statements A, B, C, D in the cluster $C_1$. We firstly define token-level subsequence. Suppose $\Sigma$ is a universe of *tokens*. Given any sequence $\alpha = \{a_1, a_2, ..., a_m\}$,

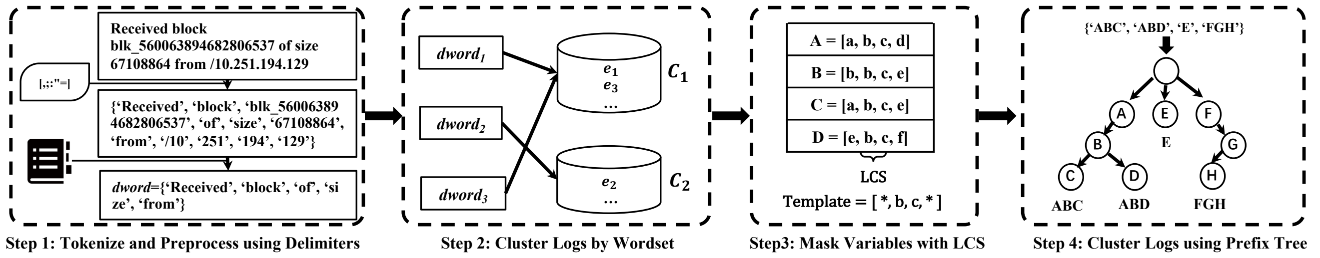



Fig. 3. The workflow of log parsing

such that $a_i \in \Sigma$. Then a subsequence of $\alpha$ is defined as $\{a_i,\ a_{i+1},\ ...,\ a_j\}$, where $i \in \mathbb{Z}^+$ and $1 \leq i \leq j \leq m$. A common subsequence is a subsequence of both sequence $\alpha_1$ and $\alpha_2$. For instance, two common subsequence of A and C are $\{a, b, c\}$. The token-level LCS of A, B, C, D is $\{b, c\}$.

Compared with traditional LCS problem, SwissLog focuses on token-level LCS. After clustering by *wordset*, log statement $e_1$ and $e_3$ are in the same cluster $C_1$. The input of Step 3 involves all *tokens*, not only those words in the vocabulary, but also those words out of vocabulary. Token-level LCS of cluster $C_1$ can be found as {‘Receiving’, ‘block’, ‘src’, ‘dest’}, so the masking result of this cluster is “Receiving block * src: * dest: *”.

Template:
Disconnecting: Too many authentication failures for * [preauth]

**e4**: Disconnecting: Too many authentication failures for admin [preauth]
**e5**: Disconnecting: Too many authentication failures for root [preauth]

Fig. 4. An example of prefix tree

*4) Step 4: Cluster Logs using Prefix Tree:* After masking variable, an important issue cannot be ignored. Given an example, templates in Fig. 4 come from OpenSSH log data [22]. We observe that the difference between log statement $e_4$ and $e_5$ is a user name, which is `admin` in $e_4$ and `root` in $e_5$, respectively. In this case, the variable part involves valid words, thus the two templates are viewed as different templates after clustering log by *wordset*. The prefix tree has been applied in log analysis before [23], [24], here we employ it so as to avoid the above cases.

The Prefix Tree, an ordered tree data structure, is often used to store a dynamic set. The root of the prefix tree points to an empty string and all the descendants of a node in the prefix tree have a common prefix string with that node. Step 4 in Fig. 3 shows an example of a prefix tree structure. Keys are listed in the nodes and final string values are below them. Given a set of string $strs = \{ABC, ABD, E, FGH\}$, they are indexed by the prefix tree. String *ABD* traverses the whole tree starting from the root to check if there exists a common prefix. Then it finds *ABC*, so their leaf nodes point to the same parent node $B$. While string *E* and *FGH* branch out because they have no common prefix.

Before clustering, we first sort all *wordset* in an alphabetical order, which largely helps to reduce the prefix tree construction time. Also, we place * as the first rank before all alpha order. Instead of searching prefix, our approach needs to find out common preceding subsequence. Each *token* (i.e., ‘Disconnecting’ in Fig. 4) in *wordset* $dword$ can be treated as an element, then we utilize the prefix tree to find common preceding subsequence. In this way, the example shown in Fig. 4 can be eventually clustered into one template.

### B. Sentence Embedding

The ultimate goal of anomaly detection is to detect diverse faults that we have described in Sec. II-A. We can observe that it is insufficient only with semantic information to detect multiple types of faults. Therefore, we also introduce temporal information as features to complement the anomaly detection approach. After log parsing, we construct sessions by correlating log with the same identifiers or sliding windows. We transform the sequence into semantic information $T$ and temporal information $\Delta T$. Then we encode these two kinds of information with the following methods.

Case 1: “Expected quotacontroller.Sync to still be running but it is **blocked**. %v”,err

Case 2: “`{"metadata":{"ownerReferences":[{"apiVersion":"%s","kind":"%s","name":"%s","uid":"%s","controller":true,"**block**OwnerDeletion":true}],"uid":"%s"}}`” ,m.controllerKind.GroupVersion(), m.controllerKind.Kind, m.Controller.GetName(), m.Controller.GetUID(), rs.UID)

Fig. 5. Two log cases extracted from Kubernetes

*1) Semantic Embedding:* Log formats are under active evolution. Yet, the key meaning of changing log statements stays unchanged as we discussed in Sec. II-B. Sentence embedding is therefore introduced to encode templates into vectors to preserve the key meaning of log statements. Word2Vec [25] has been widely used in existing approaches to transform words of log statements into vectors. But it only performs the limited utility meeting the case in Fig. 5. There are two log cases extracted from Kubernetes source codes. Both case 1 and 2 contain the word `block`. `block` in case 1 is a `verb` which means to prevent something from happening, developing, or making progress. While `block` in case 2 is a `noun` which represents that there exists a data block to be processed. It produces the same word embedding with Word2Vec for the word `block`. It will probably confuse downstream works and lead to false alarming. To overcome the challenges of polysemous words and changing events in log data, we need an advanced word embedding approach.

The pre-trained language representation gains considerable progress in the NLP field, especially BERT developed by Google. Google released the pre-trained language model which has trained on Wikipedia corpus and Book corpus. Compared to other embedding methods, the large pre-trained language model provides a sufficient word database to encode words more precisely. As described in the initial paper of BERT [15], there are two usages based on specific downstream tasks: fine-tuning and feature extraction. We adopt the latter to get the semantic embedding.

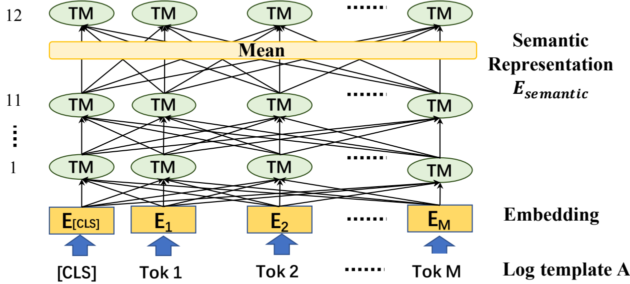



Fig. 6. The structure of BERT

Fig. 6 shows a simplified structure of BERT. As we only execute the feature extraction part of BERT, the rest of BERT will not be shown in this paper. Log template A is first tokenized into $M$ tokens as listed in Fig. 6 (Tok means Token). BERT particularly adds a [CLS] token at the beginning of the sentence which refers to the starting position of a sentence. The embedding layer generates an embedding vector $\boldsymbol{E}_i$ for each token including [CLS], where $i$ refers to the $i$th word in sentence. Then embedding vectors $\boldsymbol{E}_i$ are fed into transformer encoders (TM in Fig. 6) as model inputs. Unlike other embedding methods, a self-attention layer is added in the transformer encoder to acquire other word information in log statements. Therefore, when processing a log statement, the attention mechanism builds a correlation among all other words in this statement. After that, the output of self-attention is transferred to two feed-forward layers to learn further position and word vector relation.

SwissLog leverages an off-the-shelf service *bert-as-service* [26] which uses BERT as a sentence encoder and runs it as a service. We utilize BERT base model [27] which contains a 12-layer of transformer encoders and 768-hidden units of each transformer. Each output per token from each layer can be used as a word embedding. The first layer is close to the initial word embedding while the last layer may be biased to the training of downstream tasks. Choosing a word embedding from these is then a trade-off. Xiao, et al. [26] did research on this problem and suggests to generate word embedding in the last second layer. Hence, we take the average of the hidden state of encoding layer on the time axis to get the final semantic embedding $\boldsymbol{E}_{semantic}$.

*2) Time Embedding:* Existing approaches are difficult to detect performance issues at runtime. Moreover, the intrusive detection results in the performance slowdown. Log-based performance issue detection is then a non-intrusive and real-time approach. To detect the log time interval change shown in Fig. 1(c), we particularly introduce the temporal information. We calculate the time difference $\Delta t$ between two events $e_1$ and $e_2$, and then obtain a temporal differential sequence $\Delta \boldsymbol{T} = \{\Delta t_1, \Delta t_2, ..., \Delta t_i, ...\}$, where $i$ refers to the time axis in time series. Additionally, minus one is used to pad the beginning of the time series. For example, we obtain temporal sequence $\Delta \boldsymbol{T} = \{-1, 0, 3, 0, ...\}$ in seconds in Fig. 1(c). Intuitively, we can observe that $\Delta t$ is closely related to the former event $e_1$. For example, the IO task shows a smaller $\Delta t$ while the scheduling task shows a greater $\Delta t$. Even in the normal operation, the time interval vibrates in a task-related time range. To mitigate this task-related time vibration issue, we transform $\Delta t$ into $\theta = \frac{1}{\Delta t}$. Also, we standardize all temporal data by removing the mean and scaling to unit variance so as to receive a trainable data.

However, 1-dimension temporal data exhibits limited information. It is better to extend 1-dimension temporal data to a high dimension of time embedding. Li, et al. [28] has proposed a time-dependent event representation method. Inspired by their work, we encode $\theta$ using soft one-hot encoding.

The first step is to project the scalar value $\theta$ onto a d-dimension vector space. As presented in Eq. 1, we multiply $\theta$ with a randomly-initialized weight vector $\boldsymbol{W}$ and then add a randomly-initialized biases vector $\boldsymbol{b}$, where $p$ is the projection size. After the above linear transformation, we apply a softmax function to catch the importance vector $\boldsymbol{s}$ of the obtained projection vector. The function $softmax(\cdot)$ is used to re-scale a tensor, making its elements lie in the range $[0, 1]$ and sum to 1 along with a selected dim.

$$\boldsymbol{s} = softmax(\theta^i \boldsymbol{W} + \boldsymbol{b}), \text{where } \boldsymbol{W} \in \mathbb{R}^p,\ \boldsymbol{b} \in \mathbb{R}^p \quad (1)$$

Then we weight all rows in the randomly-initialized embedding matrix $\boldsymbol{E}_s$ with the vector values in $s$. It is better to have the same dimension $d$ as semantic embedding $\boldsymbol{E}_{semantic}$. Finally, we get the time embedding vector $\boldsymbol{E}_{time}$.

$$\boldsymbol{E}_{time} = \boldsymbol{s}\boldsymbol{E}_s, \text{where } \boldsymbol{E}_s \in \mathbb{R}^{p \times d} \quad (2)$$

### C. Attn-based Bi-LSTM

After sentence embedding, each log message is transformed into a semantic vector $\boldsymbol{E}_{semantic}$ and a time embedding vector $\boldsymbol{E}_{time}$. We obtain the concatenation $\boldsymbol{V} = concat(\boldsymbol{E}_{semantic}, \boldsymbol{E}_{time})$, so each log sequence is represented as a list of vectors (like $[\boldsymbol{V}_1, \boldsymbol{V}_2, ..., \boldsymbol{V}_T]$). Taking such vectors as input, SwissLog adopts the Attn-based Bi-LSTM neural network for detecting diverse anomalies, as shown in Fig. 7.

The LSTM network, a variant of Recurrent Neural Network (RNN), is capable of capturing contextual information for sequential data. Incorporating gating mechanisms, LSTM can have the ability to remove or add information to the cell state and finally decide what information to go through. It allows neural networks to dynamically exhibit temporal behavior. The LSTM network consists of three layers: input layer, hidden neurons layer and output layer. At each time step, LSTM calculates the new cell state $c_t$ and new hidden state $\boldsymbol{h}_t$ using the input state $\boldsymbol{V}_t$ and transferred hidden state $\boldsymbol{h}_{t-1}$. Bi-LSTM is an extension of LSTM. It particularly adds a hidden neurons

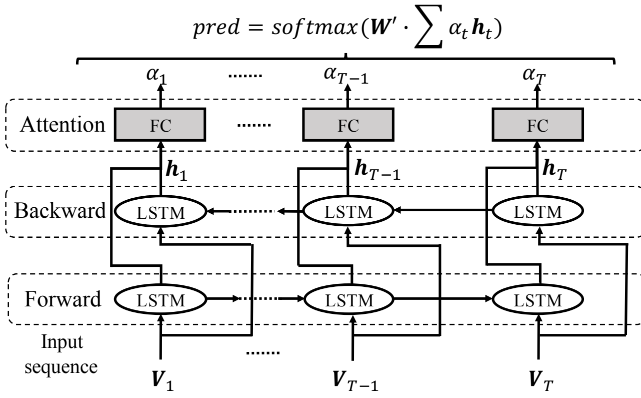



Fig. 7. The architecture of Attn-Bi-LSTM.

layer in a backward direction and calculates each hidden state $h_t$ at time $t$ through concatenating from both directions as input to output layers.

Like verbosity level in log statements, different log statements show different importance in a log sequence. To mitigate the impact of noisy or unimportant log statements, attention mechanisms are therefore introduced to Bi-LSTM to assign different weights to different log statements. Noisy or unimportant log statements will tend to be given low attentions. The attention function $\alpha_t$ at time $t$ is implemented with a fully connected layer (i.e., FC layer in Fig. 7), which performs the following calculation,

$$\alpha_t = tanh(\boldsymbol{W'}^{\alpha}_t \cdot \boldsymbol{h}_t). \quad (3)$$

Here, $\boldsymbol{W'}^{\alpha}_t$ denotes the trainable weight matrix of the attention layer at time $t$. The function $tanh(\cdot)$ is kind of an activation function. Then, all the hidden states multiply their corresponding $\alpha_t$ and are further summed to get a summarized hidden state vector. Finally, a prediction output is calculated by applying a softmax layer to the summarized hidden state vector. The computation is formulated in Eq. 4, with $\boldsymbol{W'}$ representing the softmax layer weight.

$$pred = softmax(\boldsymbol{W'} \cdot (\sum_{t=0}^{T} \alpha_t \cdot \boldsymbol{h}_t)) \quad (4)$$

At the training stage, we calculate the cross-entropy as the loss function and use the Adam optimizer [29] to train the networks. The cross-entropy is formulated in Eq. 5, where $\boldsymbol{y}^{(i)}$ denotes the one-hot representation of the label (normal or abnormal) of the $i^{th}$ log sequence and $\hat{\boldsymbol{y}}^{(i)}$ refers to its prediction.

$$H\left(\boldsymbol{y}^{(i)}, \hat{\boldsymbol{y}}^{(i)}\right) = -\sum_{j=1}^{2} y_j^{(i)} \log \hat{y}_j^{(i)}. \quad (5)$$

### D. Anomaly Detection

In the offline phase, we obtain a pre-trained Attn-based Bi-LSTM model for anomaly detection using history log. When a set of new log statements arrives, it first goes through log parsing and sentence embedding. Then the obtained vectors as input are fed into the pre-trained model. Finally, the Attn-based Bi-LSTM can detect if an anomaly occurs. Pay attention to that SwissLog makes decisions based on a session of log statements correlated by a common identifier such as block ID. Therefore, an anomaly can be robustly reported until the session is closed. In other words, SwissLog works in a near real-time mode like LogRobust [8] and LogAnomaly [7].

## IV. Experimental Evaluations

In this section, we evaluate the effectiveness and robustness of SwissLog for diverse faults by answering the following questions:

- RQ1: How effective and robust is the proposed log parser?
- RQ2: How effective is the BERT encoder on anomaly detection? Do other log parsers perform as well as the proposed log parser using BERT encoder?
- RQ3: How robust is SwissLog on those log data with changing events?
- RQ4: Can SwissLog detect performance issues? How sensitive is SwissLog to log time deviations?

All experiments in this paper are conducted on a server equipped with two 24-core CPU, 128GB RAM, and one NVIDIA GeForce GTX 1080 Ti GPU.

### A. Experiments Setting

*1) Datasets:* In this paper, we evaluate the proposed approach on the following datasets.

**Real-world Datasets.** Logpai [30] adopts 16 real-world log datasets ranging from distributed systems, supercomputers, operating systems, mobile systems, server applications, to standalone software including HDFS, Hadoop, Spark, Zookeeper, BGL, HPC, Thunderbird, Windows, Linux, Android, HealthApp, Apache, Proxifier, OpenSSH, OpenStack, and Mac. The above log datasets are provided by LogHub [22]. Each dataset contains 2,000 log samples with its ground truth tagged by a rule-based log parser. Besides sampled datasets, we select datasets collected from three representative systems to evaluate the proposed approach. The details are shown in Tab. I. HDFS log dataset is collected from a 203-node cluster on Amazon EC2 platform [3], containing 11,175,629 raw log messages. BGL dataset is a supercomputing system log dataset collected by Lawrence Livermore National Labs (LLNL) [31]. The Android dataset provided by Loghub [22] is log data recording Android framework states.

TABLE I
The Details of Log Datasets

| Log Type | #Messages | #Templates | Labeled |
|---|---|---|---|
| HDFS [32] | 11,175,629 | 30 | Yes |
| Blue Gene /L [33] | 4,747,963 | 377 | Yes |
| Android [32] | 30,348,042 | 76,923 | No |

**Synthetic HDFS Data.** To evaluate SwissLog, we synthesize new test datasets by simulating the real-world situation discussed in Sec. II. Two possible types that may occur are injected in HDFS log data illustrated as below:

- Changing events. Only unimportant words are inserted or removed, without changing the key meaning of sentences, in evolving log data. Therefore, the labels of changing log data stay unchanged. We apply this injection with a

TABLE II
COMPARISONS OF SWISSLOG AND SOTA OF PARSING ACCURACY ON DIFFERENT LOG DATASETS

| Dataset | | HDFS | Hadoop | Spark | Zookeeper | BGL | HPC | Thunderbird | OpenStack | Mac |
|---|---|---|---|---|---|---|---|---|---|---|
| Parsing Accuracy | **SwissLog** | **1.000** | **0.992** | **0.997** | **0.985** | **0.970** | **0.910** | **0.992** | **1.000** | 0.840 |
| | SOTA | 1.000 | 0.957 | 0.994 | 0.967 | 0.963 | 0.903 | 0.955 | 0.871 | **0.872** |
| Dataset | | Windows | Linux | Andriod | HealthApp | Apache | Proxifier | OpenSSH | Average | |
| Parsing Accuracy | **SwissLog** | **1.000** | **0.869** | **0.954** | **0.901** | **1.000** | **0.990** | **1.000** | **0.962** | |
| | SOTA | 0.997 | 0.701 | 0.919 | 0.822 | 1.000 | 0.967 | 0.925 | 0.865 | |

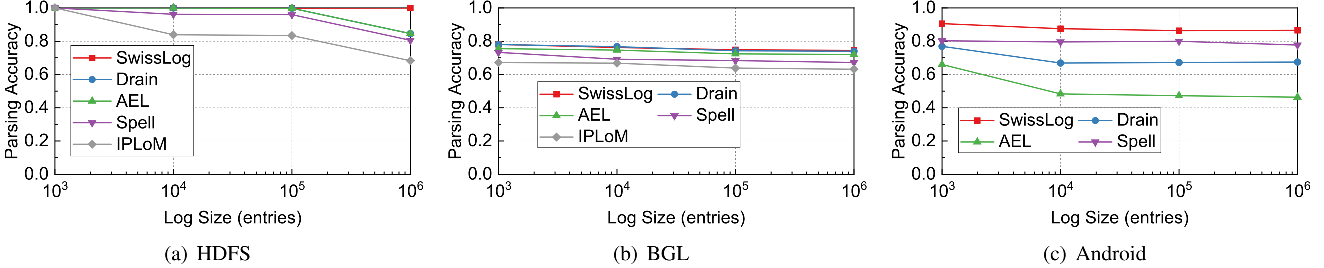



Fig. 8. Comparisons among different log parsers of parsing accuracy on different volumes of logs

specific ratio ranging from 5% to 30% to the original HDFS log data. Also, we tag the changed log message as a new log template key.

- Performance issues. Only those performance issues that do not change the log sequence order are considered. Therefore, log sequence order also stay unchanged. We apply the time interval latency injection to mimic CPU hog, memory hog, disk write burn and network delay with ratio 5% to those log whose original time interval is less than 2. We label the injected sessions with performance issues.

For simplicity, we name the dataset injected with changing events as *TestingEvent* and performance issues as *TestingPerf*.

*2) Evaluation Metrics:* We leverage the widely used metrics, namely *Precision*, *Recall*, and *F1-score* to measure the effectiveness of anomaly detection in SwissLog. Besides, the parsing accuracy (PA) metric is introduced to qualify the effectiveness of an automated log parser. Compared to previous metrics, evaluation using PA is more rigorous because partially matched templates are also considered as incorrect. The detailed definitions of them are as follows, where $TP, FP, FN$ represent True Positive, False Positive, and False Negative respectively.

- **Parsing Accuracy**: $PA = \frac{count(correct\ event\ ID\ group)}{count(all\ event\ ID\ group)}$. The ratio of correctly parsed log messages over the total number of log messages.
- **Precision**: $P = \frac{TP}{TP+FP}$. The percentage of correctly detected anomalies amongst all detected anomalies.
- **Recall**: $R = \frac{TP}{TP+FN}$. The percentage of correctly detected anomalies amongst all real anomalies.
- **F1-Score**: $F1 = \frac{2*P*R}{P+R}$. The harmonic mean of Precision and Recall.

*3) Implementation and Parameters Setting:* 6,000 normal and 6,000 abnormal blocks from real-world datasets are randomly sampled for training. The neural network is trained using Adam optimizer [29]. We use a weight decay of 0.0001 and set the initial learning rate to 0.001. We set the hidden dim to 128. The training epoch is 30 and the mini batch size is set to 32. We use the cross-entropy as the loss function. We implement SwissLog with Python 3.7, Pytorch 1.3.

### B. RQ1: The Effectiveness and Robustness of Log Parser

To answer RQ1, we utilize a sampled dataset and a large dataset to figure out the effectiveness and robustness of SwissLog. We first construct a dictionary and utilize an English corpus including 5.2 million sentences, which is accessible on [34]. After splitting this corpus with the space delimiter, we collect 588,054 distinct words. Noting that not every occurred word is valid (e.g., location name), we set an occurrence threshold to filter common valid words. The dictionary finally remains only 18,653 common words. In the evaluation, we will use these 18,653 common words as the dictionary $D$ to recognize valid words.

The sampled dataset is a quick and effective to test the effectiveness and robustness of log parsers. Therefore, we compare SwissLog with 14 log parsers in the LogPai benchmark [30] spanning 16 datasets. The results are shown in Tab. II. Due to the limited space, we only present the state-of-the-art (SOTA) result (i.e, the best score of the specific dataset shown in the LogPai benchmark [30]). In particular, the better result between SOTA and SwissLog is highlighted in bold font with a gray block.

Overall, we observe that SwissLog shows almost the best PA in all datasets except the Mac logs. Even more, SwissLog can parse HDFS, BGL, Windows, Apache, OpenSSH datasets with 100% accuracy. Noting that we only utilize 2,000 log messages for testing, thus a 100% accuracy is possible to achieve. The average of SwissLog is up to 0.962, which is much more than other log parsers by 10%. From this remarkable result, we can indicate that the dictionary-based log parsing method is close to the visual reflection of humans, it consequently achieves a better result.

However, SwissLog shows an unsatisfactory performance on the Mac logs. Consider three templates of Mac logs shown

PM response took <*> ms (<*>, powerd)
PM response took <*> ms (<*>, QQ) → <*>
PM response took <*> ms (<*>, WeChat)

Fig. 9. An example of Mac log templates

in Fig. 9. At first glance, it is not hard for us to classify these three templates into one category, in that the difference is only one word, namely the service name, *powerd*, *QQ*, *WeChat*. In this case, according to the different characteristics of services, they should be divided into three templates. However, in other cases, where we can simply treat them as the variable part, three templates should be merged into one template. We need to adjust the dictionary used in preprocessing step according to different template discriminant.

Besides the sampled dataset, we further evaluate SwissLog towards three large datasets. The comparison of parsing accuracy is shown in Fig. 8. The horizontal axis represents log size which increases in logarithm and the vertical axis denotes parsing accuracy over different amounts of log statements. We compare SwissLog with the best four log parsers in LogPai benchmark [30], namely Drain, AEL, Spell, and IPLoM. It is worth noting that IPLoM is excluded on the Android dataset since it consumes too much time to finish in parsing the data.

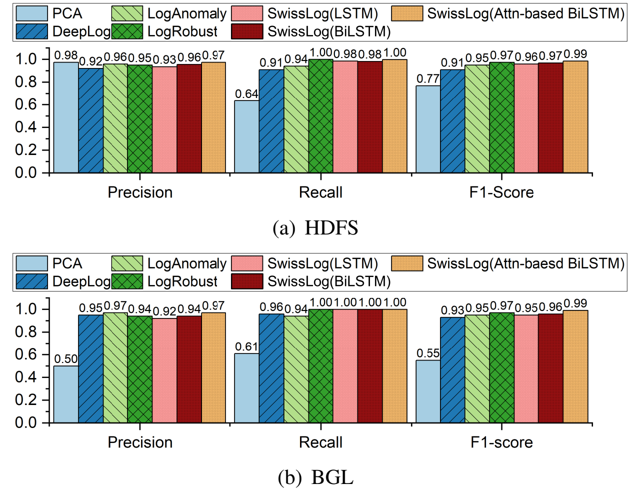
From Fig. 8, we observe that SwissLog outperforms other log parsers on HDFS and Android dataset while it is slightly higher than others on BGL dataset. When the volume of logs increases, the parsing accuracy of SwissLog drops very slightly. Also, there is no parameter tuning procedure in SwissLog. Hence, we can indicate from the above results on the sampled and large datasets that SwissLog has excellent effectiveness and robustness with different volumes and different types of log statements.

### C. RQ2: The Effectiveness of Semantic-based Model

In this part, we intend to evaluate the effectiveness of semantic-based model in SwissLog. Consequently, we conduct experiments on original datasets HDFS and BGL with two kinds of labels, namely normal sequence, and sequential log anomalies. For the HDFS dataset, we correlate log statements with the same block id named *session* in advance. For the BGL dataset, we apply a sliding window with a length of 20 entries to construct a sequence *session*.

We adopt the proposed log parser of SwissLog to extract log templates. Then we employ one supervised method (i.e., LogRobust [8]), three unsupervised methods (i.e., DeepLog [6], LogAnomaly [7], PCA [3]), and two variants of SwissLog (i.e., with Bi-LSTM and with LSTM) to detect anomaly. DeepLog [6] is a log key-based anomaly detection model and it leverages LSTM to learn the pattern of normal sequence. LogRobust [8] encodes log templates using Word2Vec and leverages Attn-based Bi-LSTM to learn and detect anomaly. LogAnomaly [7] accurately extracts the semantic and syntax information from log templates.

The comparison results of evaluation metrics *Precision/Recall/F1-Score* on different datasets are shown in Fig. 10. A lower *Precision* means that more anomalies cannot be detected while a higher *Recall* means more manual works. Compared with other competitive approaches, SwissLog




(a) HDFS

(b) BGL

Fig. 10. Comparisons of different approaches on different datasets

with Attn-based Bi-LSTM achieves a better balance between *Precision* and *Recall*. It achieves a very high *F1-Score* up to 0.99 and 0.98 in HDFS and BGL, respectively. It is worth noting that Attn-based Bi-LSTM outperforms Bi-LSTM and LSTM in detecting sequential log anomalies. Deep learning based methods perform well in *Precision*. Here the freely changed variable is the BERT encoder. Thus, the results confirm the effectiveness of BERT.

Next, we need to figure out how the proposed log parser affects the anomaly detection model. We select the top 2 log parsers in the LogPai benchmark [30], namely AEL and Drain, to parse the HDFS dataset in Tab. I into log templates. Then these log templates work as input to the anomaly detection model for sequence order changes.

TABLE III
RESULTS OF DIFFERENT LOG PARSERS USING BERT ENCODERS

| LogParser | Precision | Recall | F1-Score |
|---|---|---|---|
| Drain | 0.95 | 0.96 | 0.96 |
| AEL | 0.96 | 0.97 | 0.97 |
| **SwissLog** | **0.97** | **1.00** | **0.99** |

Tab. III presents the comparison results among different log parsers. Compared with other log parsers, SwissLog achieves the best score of 0.99 in *F1-Score*. The biggest difference between SwissLog and other approaches is that SwissLog extracts more valuable valid words which provide wealthy information for sentence embedding. For example, "Exception in receivedBlock" is a piece of a log statement. AEL and Drain treat it as the variable part so that the log template loses lots of valuable semantic information. In contrast, the dictionary-based log parsing method reduces the occurrence of this condition. Through these experiments, we further confirm the effectiveness of the proposed log parser. Hence, we can verify the validity of semantic-based parsing and embedding in SwissLog.

### D. RQ3: The Robustness on Changing Log Data

As we discussed in Sec. II, changing events inevitably occur in modern software systems under active development and maintenance. In this part, we evaluate the effectiveness

of the semantic-based anomaly detection model on changing log data. Two competitive approaches, namely DeepLog and LogRobust are chosen as the baselines. DeepLog leverages log key to identify templates while SwissLog and LogRobust utilize sentence embedding. We use the model trained by the original dataset to predict *TestingEvent* dataset. Since the injected events are changed, we tag them as new templates in DeepLog. The experimental results on *TestingEvent* are shown in Fig. 11.

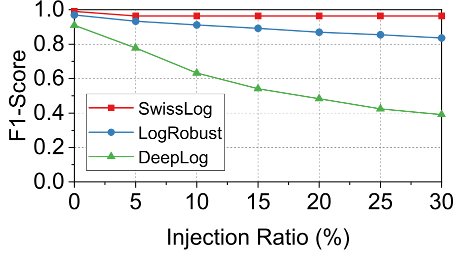



Fig. 11. F1-Score on the dataset TestingEvent

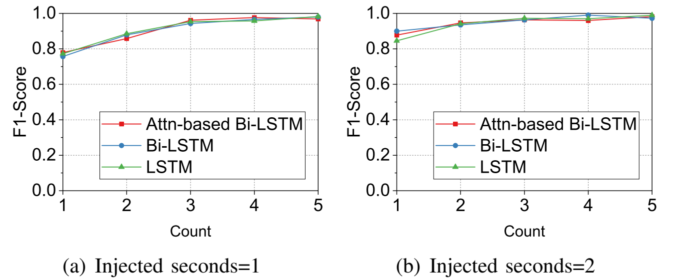
In Fig. 11, the horizontal axis denotes the injection ratio and the vertical axis denotes the *F1-Score* of different anomaly detection models. We can observe that the *F1-Score* of SwissLog, LogRobust, and DeepLog with injection ratio 5% are 0.96, 0.93, 0.78 respectively. Semantic-based models (i.e., LogRobust, SwissLog) achieve a better *F1-Score* than log key-based models (i.e, DeepLog). The reason is that the log key-based model treats those changing events as new templates, which probably results in false alarming. Semantic-based models utilize sentence embedding to encode templates, which extend the 1-dimension sentence array to 2-dimension sentence embedding matrix. Hence, sentence embedding brings the robustness on changing log data as we expected. As the injected ratio increases, the *F1-Score* of LogRobust starts to drop while SwissLog still maintains a high *F1-Score*. For example, under the injection ratio of 30%, the *F1-Score* of SwissLog is higher than 0.9, but LogRobust can only achieve 0.84. Compared to 2-dimension unordered sentence embedding array in LogRobust, BERT encoders in SwissLog capture the contextual information in templates and encode changing log data with similar vectors. Hence, SwissLog with BERT encoders is almost not affected by changing events, showing a good robustness.

### E. RQ4: The Effectiveness and Sensitivity of Time Embedding

The ultimate goal of SwissLog is to detect diverse faults including sequential log anomalies and performance issues. To answer RQ4, we need to verify the effectiveness of time embedding in SwissLog using synthetic HDFS dataset *TestingPerf*. We compare SwissLog with those models using different time embedding: 1) 1-dimension raw time (i.e., Raw time in Tab. IV). 2) The mean of $\boldsymbol{E}_{semantic}$ and $\boldsymbol{E}_{time}$ (i.e., Mean in Tab. IV). 3) Log time interval without reciprocal operation (i.e., WO_Reciprocal in Tab. IV). The comparison among them is shown in Tab. IV. We can observe that SwissLog achieves 0.92 in *Precision* while others are less than 0.7. Raw time shows the worst results 0.42 in *F1-Score* since it only contains 1-dimension information. It seems very weak in front of the giants d-dimension $\boldsymbol{E}_{semantic}$. Encoding the log time interval without reciprocal operation also shows an unsatisfactory performance 0.56 in *F1-Score* as expected. The time interval violation is related to the base of the time interval. The reciprocal operation can widen the gap between normal time violation and abnormal time violation. The mean embedding loses part of semantic information and temporal information, therefore it obtains 0.76 in *F1-Score*. According to the result, the effectiveness of time embedding is confirmed.

TABLE IV
RESULTS ON DIFFERENT OPERATION FOR TIME EMBEDDING

| Time embedding | Precision | Recall | F1-Score |
|---|---|---|---|
| Raw time | 0.67 | 0.68 | 0.42 |
| Mean | 0.67 | 0.94 | 0.76 |
| WO_Reciprocal | 0.65 | 0.66 | 0.56 |
| **SwissLog** | **0.92** | **0.99** | **0.95** |




(a) Injected seconds=1 (b) Injected seconds=2

Fig. 12. F1-Score on TestingPerf

The sensitivity to time violation is crucial in detecting performance issues. Here we inject additional 1-second and 2-second latency (i.e., the latency between two consecutive log statements) to mimic performance issues, where the count of injected faults ranges from 1 to 5. Also, we choose the deep learning model Bi-LSTM and LSTM as competitive approaches. As shown in Fig. 12, when the injected number of performance issues and the injected latency increase, all of them achieve a higher *F1-Score*. We can observe from Fig. 12(b) that when injecting two 2-second latency faults, Attn-based Bi-LSTM achieves an excellent result 0.95 in *F1-Score*. In Fig. 12(a), even when injecting one and two time interval changes with only 1-second, Attn-based Bi-LSTM still achieves 0.78 and 0.86 in *F1-Score*, respectively. It reveals the sensitive response of SwissLog to time violation. However, the performance of these three deep learning models seems very similar. That means we can always get a consistent result with a time embedding approach to detect performance issues. Yet, Attn-based Bi-LSTM outperforms Bi-LSTM and LSTM in detecting log sequential anomalies which is shown in RQ2 IV-C. Therefore, we choose Attn-based Bi-LSTM in SwissLog so as to get better performance.

## V. DISCUSSION

**Threats To Validity.** We discuss threats in two aspects: 1) The log parser of SwissLog is based on a dictionary. However, 18,653 common words in our filtered dictionary cannot cover

all of the valid words in logs. A portion of terminology, such as "DataNode" in Hadoop, is not included in them. It is quite challenging to find a suitable dictionary for all software systems generally. Instead, customizing a particular dictionary for a software system accordingly is a better choice. 2) Although the time interval of the normal sequence seems unchanged, different tasks still have different execution times in software systems. The label of task type should also be considered as feature in the future.

**Efficiency.** Efficiency is critical in real-time anomaly detection on large-scale log data. We adopt the metric milliseconds per log statement (ms/l) to measure the efficiency of SwissLog on HDFS dataset which includes 11,175,629 raw log messages. The elapsed time of log parsing, sentence embedding, network training, and anomaly detection are 4.5 ms/l, 2.6 ms/l, 800.0 ms/l, 4.5 ms/l, respectively. Log parsing, sentence embedding and anomaly detection are the three core stages of the online process. All of them achieve 4.5 ms/l within our experimental environments. Therefore, it is reasonable to believe SwissLog can work in real-time anomaly detection on large-scale log data.

## VI. Related Work

**Log Parsing.** Log parsing is the fundamental step of log analysis works which has been widely studied. Xu, et al. [3] and Nagappan, et al. [35] parsed logs by generating regular expressions based on source codes. However, not all projects are open-source online in practice. Moreover, existing log parsing approaches can be divided into several categories. 1) Similarity based clustering: LKE [36], LogSig [37], LogMine [38], SHISO [39] and LenMa [40] compute distances between two log messages or their signature and then cluster them based on similarity. 2) Frequency based clustering: a set of constant items generally occurs frequently in logs, so mining frequency of items is a straightforward way to parse logs automatically. SLCT [41], LFA [42] and LogCluster [43] firstly record frequency of item and then group them into multiple groups. 3) Heuristics by searching tree: Drain [24] and Spell [44] utilize a tree structure to parse log into multiple templates.

**Anomaly Detection.** Existing anomaly detection approaches mainly focus on sequential log anomalies. They can be mainly separated into data mining methods and deep learning methods.

Data mining methods include supervised learning methods and unsupervised learning methods. 1) Supervised: by training labeled log data, supervised methods (e.g., decision tree [45], support vector machines [46], regression-based technique [47]) can learn the fixed pattern of different labeled log. Consequently, they generally achieve a higher score than unsupervised methods. But it is time-consuming to label a large volume amount of history data for training. Moreover, they cannot detect a black swan, which may not be involved in history data. 2) Unsupervised: unsupervised methods take unlabeled history data to train. This kind of methods generally constructs a normal space and an abnormal space for normal sequence and abnormal sequence, respectively [3], [4]. The strength of unsupervised methods is unnecessary to label log data. But similar to supervised methods, a black swan is also hard to detect.

With the prevalence of deep learning, anomaly detection models based on deep learning are widely studied [6]–[8], [16]. Deep learning methods go through parsing log, model training, and model predicting. 1) Log key-based models: log key-based models first parse log statements into templates and tag them with log keys. Du, et al. [6] adopted LSTM while Vinayakumar et, al. [16] trained stacked-LSTM to model the sequential patterns of normal and abnormal sessions. However, when source codes update for a new version, the old-trained log key-based model will treat them as new templates which leads to unsatisfactory performance. 2) Semantic-based models: As log data contains wealthy semantic information of system states, NLP techniques are utilized to analyze log-based anomaly detection. Meng, et al. [7] trained LSTM considering the synonyms and antonyms with word vectors. However, it also takes log count vector as inputs that are not robust to the changing log data. Zhang, et al. [8] leveraged Attention-Based Bi-LSTM to detect anomaly. But Word2Vec and TF-IDF ignore the contextual information in sentences. In our work, we use BERT to capture the contextual semantic meaning in sentences.

## VII. Conclusion and Future Work

Log-based anomaly detection methods help to detect and analyze anomalies. Changing events and performance issues are two major challenges in existing approaches. We propose SwissLog in this paper, a robust and unified anomaly detection model for diverse faults including sequential log anomalies and performance issues. Compared to other approaches, SwissLog employs BERT encoders to encode log templates which can capture contextual information in a log statement. Also, SwissLog utilizes the concatenation of semantic embedding and temporal embedding to train a unified Attn-based Bi-LSTM model for diverse faults. We have conducted experiments on real-world datasets and synthetic datasets to evaluate the effectiveness and robustness of SwissLog. The results show that our approach outperforms others.In the future, we plan to collect more real-world datasets to evaluate SwissLog. Moreover, we will design an effective and flexible incremental updating mechanism to adapt to the new emerging log templates and log sequences.

## VIII. Acknowledgments

The work was supported by the Key-Area Research and Development Program of Guangdong Province (2020B010165002), the National Key Research Development Program of China (2019YFB1804002), the Basic and Applied Basic Research of Guangzhou (202002030328) and the Natural Science Foundation of Guangdong Province (2019A1515012229). The corresponding author is Pengfei Chen.

## References

[1] "Alibaba cloud reports io hang error in north china," https://equalocean.com/technology/20190303-alibaba-cloud-reports-io-hang-error-in-north-china, 2019, [Online].

[2] H. Mi, H. Wang, Y. Zhou, M. R.-T. Lyu, and H. Cai, "Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 6, pp. 1245–1255, 2013.

[3] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *SOSP'09: Proc. of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. ACM, 2009, pp. 117–132.

[4] J.-G. Lou, Q. Fu, S. Yang, J. Li, and B. Wu, "Mining program workflow from interleaved traces," in *SIGKDD'10: Proc. of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2010, pp. 613–622.

[5] X. Yu, P. Joshi, J. Xu, G. Jin, H. Zhang, and G. Jiang, "Cloudseer: Workflow monitoring of cloud infrastructures via interleaved logs," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 2, pp. 489–502, 2016.

[6] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *SIGSAC'17: Proc. of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 1285–1298.

[7] W. Meng, Y. Liu, Y. Zhu, S. Zhang, D. Pei, Y. Liu, Y. Chen, R. Zhang, S. Tao, P. Sun *et al.*, "Loganomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs." in *IJCAI'19: Proc. of the 28th International Joint Conference on Artificial Intelligence*, 2019, pp. 4739–4745.

[8] X. Zhang, Y. Xu, Q. Lin, B. Qiao, H. Zhang, Y. Dang, C. Xie, X. Yang, Q. Cheng, Z. Li *et al.*, "Robust log-based anomaly detection on unstable log data," in *ESEC/FSE'19: Proc. of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2019, pp. 807–817.

[9] S. Kabinna, C.-P. Bezemer, W. Shang, M. D. Syer, and A. E. Hassan, "Examining the stability of logging statements," *Empirical Software Engineering*, vol. 23, no. 1, pp. 290–333, 2018.

[10] C. Lou, P. Huang, and S. Smith, "Understanding, detecting and localizing partial failures in large system software," in *NSDI'20: Proc. of the 17th USENIX Symposium on Networked Systems Design and Implementation*, 2020, pp. 559–574.

[11] "Gocardless service outage on october 10th, 2017," https://gocardless.com/blog/incident-review-api-and-dashboard-outage-on-10th-october, 2017, [Online].

[12] "Office 365 update on recent customer issues," https://blogs.office.com/2012/11/13/update-on-recent-customer-issues/, 2017, [Online].

[13] "Google compute engine incident 17008," https://status.cloud.google.com/incident/compute/17008, 2017, [Online].

[14] "Twilio billing incident post-mortem: Breakdown, analysis and root cause." https://bit.ly/2V8rurP, 2013, [Online].

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

[16] R. Vinayakumar, K. Soman, and P. Poornachandran, "Long short-term memory based operation log anomaly detection," in *ICACCI'17: 2017 International Conference on Advances in Computing, Communications and Informatics*. IEEE, 2017, pp. 236–242.

[17] P. Huang, C. Guo, L. Zhou, J. R. Lorch, Y. Dang, M. Chintalapati, and R. Yao, "Gray failure: The achilles' heel of cloud-scale systems," in *HotOS'17: Proc. of the 16th Workshop on Hot Topics in Operating Systems*, 2017, pp. 150–155.

[18] P. Huang, C. Guo, J. R. Lorch, L. Zhou, and Y. Dang, "Capturing and enhancing in situ system observability for failure detection," in *OSDI'18: Proc. of the 13th USENIX Symposium on Operating Systems Design and Implementation*, 2018, pp. 1–16.

[19] D. J. Dean, H. Nguyen, X. Gu, H. Zhang, J. Rhee, N. Arora, and G. Jiang, "Perfscope: Practical online server performance bug inference in production cloud computing infrastructures," in *SOCC'14: Proc. of the ACM Symposium on Cloud Computing*, 2014, pp. 1–13.

[20] D. J. Dean, H. Nguyen, P. Wang, X. Gu, A. Sailer, and A. Kochut, "Perfcompass: Online performance anomaly fault localization and inference in infrastructure-as-a-service clouds," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 6, pp. 1742–1755, 2015.

[21] "wordninja," https://github.com/keredson/wordninja, 2020, [Online].

[22] S. H. Pinjia He, Jieming Zhu and M. R. Lyu, "Loghub: A large collection of system log datasets for ai-powered log analytics," in *ESEC/FSE'19: Proc. of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2019.

[23] W. Meng, Y. Liu, F. Zaiter, S. Zhang, Y. Chen, Y. Zhang, Y. Zhu, E. Wang, R. Zhang, S. Tao *et al.*, "Logparse: Making log parsing adaptive through word classification."

[24] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, "Drain: An online log parsing approach with fixed depth tree," in *ICWS'17: 2017 IEEE International Conference on Web Services*. IEEE, 2017, pp. 33–40.

[25] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *ICML'14: Proc. of the 31st International Conference on Machine Learning*, 2014, pp. 1188–1196.

[26] H. Xiao, "bert-as-service," https://github.com/hanxiao/bert-as-service, 2018.

[27] "Bert pretrained models," https://github.com/google-research/bert, 2020, [Online].

[28] Y. Li, N. Du, and S. Bengio, "Time-dependent representation for neural event sequence prediction," *arXiv preprint arXiv:1708.00065*, 2017.

[29] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[30] J. Zhu, S. He, J. Liu, P. He, Q. Xie, Z. Zheng, and M. R. Lyu, "Tools and benchmarks for automated log parsing," in *ICSE(SEIP)'19: Proc. of the 41st International Conference on Software Engineering: Software Engineering in Practice*. IEEE Press, 2019, pp. 121–130.

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

[32] "Loghub datasets," https://zenodo.org/record/3227177, 2019, [Online].

[33] "Bluegene/l message types," https://www.usenix.org/cfdr-data#hpc4, 2019, [Online].

[34] "English corpus," https://storage.googleapis.com/nlp_chinese_corpus/translation2019zh.zip, 2020, [Online].

[35] M. Nagappan, K. Wu, and M. A. Vouk, "Efficiently extracting operational profiles from execution logs using suffix arrays," in *ISSRE'09: Proc. of the 20th International Symposium on Software Reliability Engineering*. IEEE, 2009, pp. 41–50.

[36] Q. Fu, J.-G. Lou, Y. Wang, and J. Li, "Execution anomaly detection in distributed systems through unstructured log analysis," in *ICDM'09: Proc. of the 9th IEEE International Conference on Data Mining*. IEEE, 2009, pp. 149–158.

[37] M. Mizutani, "Incremental mining of system log format," in *SCC'13: 2013 IEEE International Conference on Services Computing*. IEEE, 2013, pp. 595–602.

[38] H. Hamooni, B. Debnath, J. Xu, H. Zhang, G. Jiang, and A. Mueen, "Logmine: Fast pattern recognition for log analytics," in *CIKM'16: Proc. of the 25th ACM International on Conference on Information and Knowledge Management*. ACM, 2016, pp. 1573–1582.

[39] K. Q. Zhu, K. Fisher, and D. Walker, "Incremental learning of system log formats," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 1, pp. 85–90, 2010.

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

[41] R. Vaarandi, "A data clustering algorithm for mining patterns from event logs," in *IPOM'03: Proc. of the 3rd IEEE Workshop on IP Operations & Management*. IEEE, 2003, pp. 119–126.

[42] M. Nagappan and M. A. Vouk, "Abstracting log lines to log event types for mining software system logs," in *MSR'10: Proc. of the 7th IEEE Working Conference on Mining Software Repositories*. IEEE, 2010, pp. 114–117.

[43] R. Vaarandi and M. Pihelgas, "Logcluster-a data clustering and pattern mining algorithm for event logs," in *CNSM'15: Proc. of the 11th International Conference on Network and Service Management*. IEEE, 2015, pp. 1–7.

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

[45] M. Chen, A. X. Zheng, J. Lloyd, M. I. Jordan, and E. Brewer, "Failure diagnosis using decision trees," in *ICAC'04: Proc. of the first International Conference on Autonomic Computing*. IEEE, 2004, pp. 36–43.

[46] Y. Liang, Y. Zhang, H. Xiong, and R. Sahoo, "Failure prediction in ibm bluegene/l event logs," in *ICDM'07: Proc. of the 7th IEEE International Conference on Data Mining*. IEEE, 2007, pp. 583–588.

[47] M. Farshchi, J.-G. Schneider, I. Weber, and J. Grundy, "Experience report: Anomaly detection of cloud application operations using log and cloud metric correlation analysis," in *ISSRE'15: Proc. of the 26th International Symposium on Software Reliability Engineering*. IEEE, 2015, pp. 24–34.