Spell: Online Streaming Parsing of Large Unstructured System Logs

Min Du, *Student Member, IEEE* and Feifei Li, *Member, IEEE*

Abstract—System event logs have been frequently used as a valuable resource in data-driven approaches to enhance system health and stability. A typical procedure in system log analytics is to first parse unstructured logs to structured data, and then apply data mining and machine learning techniques and/or build workflow models from the resulting structured data. Previous work on parsing system event logs focused on offline, batch processing of raw log files. But increasingly, applications demand online monitoring and processing. As a result, a streaming method to parse unstructured logs is needed. We propose an online streaming method Spell, which utilizes a longest common subsequence based approach, to parse system event logs. We show how to dynamically extract log patterns from incoming logs and how to maintain a set of discovered message types in streaming fashion. An enhancement to find more accurate message types is also proposed. We also propose and evaluate a method to automatically discover semantic meanings for parameter fields identified by Spell. We compare Spell against state-of-the-art methods to extract patterns from system event logs on large real data. The results demonstrate that, compared with other log parsing alternatives, Spell shows its superiority in terms of both efficiency and effectiveness.

Index Terms—Log parsing, log data, system logs

1 Introduction

The increasing complexity of modern computer systems has become a significant limiting factor in deploying and managing them. Being able to be alerted and mitigate the problem right away has become a fundamental requirement in many computer systems. As a result, automatically detecting anomalies upon happening in an online fashion is an appealing solution. Data-driven methods based on machine learning and data mining techniques are heavily employed to understand complex system behaviors, for example, exploring machine data for automatic pattern discovery and anomaly detection. System logs, as a universal data source that contains important information such as execution paths and program running status, are valuable assets in assisting these data-driven system analytics, in order to gain insights that are useful to enhance system health, stability, and usability.

The effectiveness of system log mining has been validated by recent literature. Logs could be used to detect execution anomalies [1], [2], [3], monitor network failures [4], or even find software bugs [5]. Researchers have also used system logs to discover and diagnose performance problems [6]. Logs contain intrinsic underlying information that could help to understand system behaviors [7].

To alleviate the pain of diving into massive *unstructured* log data, in most prior work, the first and foremost step is to automatically *parse the unstructured system logs to structured data* [1], [2], [3], [5]. There have been a substantial study on how to achieve this, for example, using regular expressions [8], leveraging the source code [5], or parsing purely based on system log characteristics using data mining approaches such as clustering and iterative partitioning [1], [9], [10], [11]. Nevertheless, except the approach that uses regular expressions which requires domain-specific expert knowledge [8], hence, does not work for general purpose system log parsing, or the approach that leverages the source code [12] which is often unavailable, none of the previous methods could achieve truly online parsing in a streaming fashion. Some work claimed "online" processing, but with the requirement of doing some extensive offline processing first [13], or using regular expressions to remove certain fields [14], and only then matching log entries with the data structures and patterns previously identified.

Furthermore, previous methods that are tuned for a specific type of system log may work terribly on a new format or type of system logs. For example, OpenStack is a very popular open source cloud infrastructure. Its logs contain various formats that are not present in previous system logs, such as JSON format. OpenStack log analysis is important for automatic problem solving but current work remains manually parsing message types as the first step [15]. Our method is designed as a general-purpose streaming log parsing method, hence, it's system, type, or format agnostic. In our evaluation, we have collected OpenStack raw log messages as our test data, and we obtain ground truth message types by parsing them from the source code of OpenStack; the results show that our method works substantially better than all previous methods.

There is also an increasing demand to properly manage and store system logs [16]. Thus log management systems (LMS) are in great need and becoming widely deployed in

• *The authors are with the School of Computing, University of Utah, Salt Lake City, UT 84112. E-mail: {mind, lifeifei}@cs.utah.edu.*

Manuscript received 2 Oct. 2017; revised 30 Sept. 2018; accepted 6 Oct. 2018. Date of publication 11 Oct. 2018; date of current version 4 Oct. 2019. (Corresponding author: Min Du.)
Recommended for acceptance by C. Ordonez.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TKDE.2018.2875442

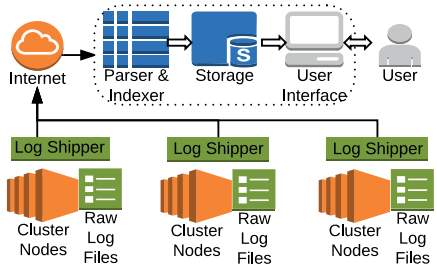


Fig. 1. A typical modern log management system architecture.

@timestamp	message
2016-06-12T09:53:13.372-07:00	container-server: Started child 15783
2016-06-12T09:53:13.371-07:00	container-server: Started child 15779
2016-06-12T09:53:12.861-07:00	object-server: Started child 15802
2016-06-12T09:53:12.861-07:00	object-server: Started child 15801
2016-06-12T09:53:12.629-07:00	account-server: Started child 15861
2016-06-12T09:53:12.627-07:00	account-server: Started child 15860

Fig. 2. Screen-shot of log messages on http://logstash.openstack.org/.

recent years (e.g., ELK by Elastic.co). A typical architecture of an LMS is shown in Fig. 1. On each node, a log shipper forwards log entries to a centralized server, which often contains a log parser, a log indexer, a storage engine and a user interface. In such systems the default log parser only parses simple schema information such as timestamp and hostname. The log entry itself is treated as an unstructured text value (hence, the need for ElasticSearch to support approximate string search). During peak workloads, it is common for thousands of log messages to arrive per minute. This could be overwhelming for an online processing based on an unstructured, text search approach.

In contrast, a *structured* approach is to parse the event logs into structured data that are much easier to query, summarize and aggregate. Consider raw log messages in Fig. 2, instead of using them as raw, unstructured text data which is used by the Logstash project for OpenStack, it is much easier for an end user (and a data-driven analytical process) to understand and use the *structured data* presented in Table 1.

Log entries are produced by the "print" statements in a system program's source code. As such, we can view a log entry as a collection of ("message type", "parameter value") pairs. For example, a log printing statement *printf("File %d finished.", id);* contains a *constant* message type *File finished* and a *variable* parameter value which is the file id. Hence, the goal of a *structured log parser* is to identify the message type *File * finished*, where * stands for the place holder for variables (parameter values).

Contributions. In this paper, we propose Spell, a structured Streaming Parser for Event Logs using an LCS (longest common subsequence) based approach. Spell parses unstructured log messages into structured message types and parameters in an online streaming fashion. The time complexity to process each log entry e is close to linear (to the size of e).

With streaming, real-time message type and parameter extraction produced by Spell, not only it provides a concise, intuitive summary for end users, but the logs are also represented by clean structured data to be processed and analyzed further using advanced data analytics methods by downstream analysts. Using three state-of-the-art methods to automatically extract message types and parameters from raw log files as the competing baseline, our study shows that compared with state-of-the-art methods, Spell outperforms them in terms of both efficiency and effectiveness. Furthermore, Spell is also able to infer a semantic meaning for each parameter field, to help users understand the log data.

TABLE 1
An Example of Structured Message Type Summary

Message Type	Frequency	Parameters
container-server: Started child *	2	15779, 15783
object-server: Started child *	2	15801, 15802
account-server: Started child *	2	15860, 15861
...	...	...

The rest of this paper is organized as follows. Section 2 provides the problem formulation and a literature survey. Section 3 presents our Spell algorithm and a number of optimizations. Section 4 discusses several limitations and extensions to Spell. Section 5 evaluates our method using large real system logs. Finally, Section 6 concludes the paper.

2 Preliminary and Background

2.1 Problem Formulation

System event logs are a universal resource that exists practically in any system. We use *system event logs* to denote the free-text audit trace generated by the system execution (typically in the `/var/log` folder). A log message or a log record/entry refers to one line in the log file, which is produced by a *log printing statement* in the source code of a user or kernel program running on or inside the system.

Our goal is to parse each log entry e into a collection of *message types* (and parameter values). Here each message type in e has a one-to-one mapping with a log printing statement in the source code producing the raw log entry e. For example, a log printing statement: `printf("Temperature %s exceeds warning threshold \n", tmp);` may produce several log entries such as: `Temperature (41C) exceeds warning threshold` and `Temperature (43C) exceeds warning threshold`, where the parameter values are `41C` and `43C` respectively, and the message type is: `Temperature * exceeds warning threshold.`

Formally, a *structured log parser* is defined as follows:

Definition 1 (Structured Log Parser). *Given an ordered set of log entries (ordered by timestamps),* log $= \{e_1, e_2, \ldots, e_L\}$, *that contain m distinct message types produced by m different log printing statements from p different programs, where the values of m and p (and the printing statements and the source code of these programs) are* unknown, *a structured log parser is to parse* log *and produce all* message types *from those m log printing statements. We consider each log printing statement contains a single, unique message type that may have multiple parameters.*

A structured log parser is the first and foremost step for most automatic and smart log mining and data-driven log analytics solutions, and also a useful and critical step for managing logs in a log management system (LMS).

Our objective is to design a *streaming* structured log parser such that it makes only one pass over the log and processes each log entry in an online, streaming fashion continuously.

Without loss of generality, we assume that the size of each log entry is $O(n)$ words.

2.2 Related Work

Mining interesting patterns from raw system logs has been an active research field for over a decade. Two major efforts in this area include generating features from raw logs to apply various data analytics, e.g., [2], [3], [5], and building execution models from system logs followed by comparing it with future system executions, e.g., [1], [7]. There are also efforts in mining workflow models from concurrent logs [2], [3], [8], by first parsing unstructured logs, and then building workflow models to identify dependencies and concurrencies from the interleaved log traces (originated from different programs).

To achieve effective data-driven log analytics, the first and foremost process is to turn unstructured logs into structured data. Xu et al. [5] uses the schema from log printing statements in the original programs' source code to extract message types. In [8], the raw logs are parsed using predefined, domain-specific regular expressions. There are efforts to make this process more automatic and more accurate. Fu et al. [1] proposes a method to first cluster log entries using pairwise weighted edit distance, and then use several heuristics like the number of distinct contents and entropy at each position for recursively splitting. IPLoM [9] explores several heuristics to iteratively partition system logs, such as log size and the bipartite relationship between words in the same log message. LogTree [10] utilizes the format information of raw logs and applies a tree structure to extract system events from raw logs. LogSig [11] generates system events from textual log messages by searching the most representative message signatures. HELO [13] extracts constants and variables from message bodies, by first using an offline classification step and then performing online clustering based on the template set by the first step. As is compared and evaluated in [17], IPLoM represents the state-of-the-art offline log parser in this direction.

HLAer [18] is a heterogeneous log analysis system which utilizes a hierarchical clustering approach with pairwise log similarity measures to assist log formatting. As an extension of HLAer, LogMine [19] is a hierarchical log parser that is implemented using map-reduce framework, achieving hundreds of times speedup compared with the unscalable method. Different from log parsers like Spell, LogMine has the flexibility to choose from various levels of granularities for *fixed values* and *variables*. However, it does not guarantee that the particular message types composed by the constant strings in log printing statements could be generated at some level. In terms of parallelization, a map-reduce framework is carefully designed to use in different steps of LogMine, e.g., in log clustering to compare each pair of log messages, and in final sequential pattern generation. In contrast, Spell could be done in parallel using multi-threads processing, by simply assigning one thread to a new log entry upon arrival.

All these structured log parsing methods focus on offline batched processing or matching new log entries with previously offline-extracted message types or regular expressions (e.g., from source code). Hence, they *cannot* be used as an online streaming structured log parser. Drain [14] parses log messages using a fixed depth tree in an online fashion, but its first step requires using regular expressions to filter out digits and other parameter values etc., which requires domain knowledge to do so. Even so, Drain is outperformed by Spell which does not require any pre-parsing, as evaluated in Section 5. As the first truly streaming log parser, Spell is initially presented in [20]. Based on that, this journal version has certain unique contributions which mainly include: 1) an advanced index structure (inverted index) to improve efficiency; 2) split/merge heuristics to improve effectiveness; 3) parameters semantic inference to help users' understanding; 4) a parallel implementation of Spell to further speedup its processing; 5) more detailed analysis and experiments to fully evaluate different versions of Spell, and to compare with more previous methods.

There are also commercial and open source softwares on log management and analysis. Splunk is a leading log management system that offers a suite of solutions to find useful information from machine data. ELK offers a rich set of open-sourced tools that could gather logs from distributed nodes, and then index, store, for users to query/visualize. All these tools provide interface to parse logs upon their arrival. However, their parsers are based on regular expressions defined by end users. The system itself can only parse very simple and basic structured schema such as timestamp and hostname, while log messages are treated as unstructured text values.

3 Spell: Streaming Structured Log Parser

We now present Spell, a streaming structured log parser for system event logs. Since a basic building block for Spell is a longest common subsequence (LCS) algorithm, hence, Spell stands for Streaming structured Parser for Event Logs using LCS. In what follows, we first review the LCS problem.

3.1 The LCS Problem

Suppose Σ is a universe of alphabets (e.g., a-z, 0-9). Given any sequence $\alpha = \{a_1, a_2, a_3, \ldots, a_m\}$, such that $a_i \in \Sigma$ for $1 \leq i \leq m$, a subsequence of α is defined as $\{a_{x_1}, a_{x_2}, a_{x_3}, \ldots, a_{x_k}\}$, where $\forall x_i$, $x_i \in \mathbb{Z}^+$, and $1 \leq x_1 < x_2 < \cdots < x_k \leq m$. Let $\beta = \{b_1, b_2, b_3, \ldots, b_n\}$ be another sequence such that $b_j \in \Sigma$ for $j \in [1, n]$. A subsequence γ is called a common subsequence of α and β iff it is a subsequence of each. The longest common subsequence (LCS) problem for input sequences α and β is to find longest such γ. For instance, sequence $\{1, 3, 5, 7, 9\}$ and sequence $\{1, 5, 7, 10\}$ yields an LCS of $\{1, 5, 7\}$.

The LCS problem has a long history and its variants were also extensively studied [21], [22], [23], [24], [25]. Its applications include diff utility used to find the difference of two files, and version control systems such as Git.

Inspired by the usefulness of finding LCS in these applications, we observe that an LCS-based method can be developed to efficiently and effectively extract message types from raw system logs. This is a seemingly natural idea, yet has not been explored by existing literature. Our key observation is that, if we view the output by a log printing statement (which is a log entry) as a *sequence*, in most log printing statements, the constant that represents a message type often takes a majority part of the sequence and the parameter values take only a small portion. If two log entries are produced by the same log printing statement stat, but only differ by having different parameter values, the LCS of the two sequences is very likely to be the constant in the code stat, implying a message type.

The merit of using the LCS formulation to parse system event logs, compared to previously mentioned clustering

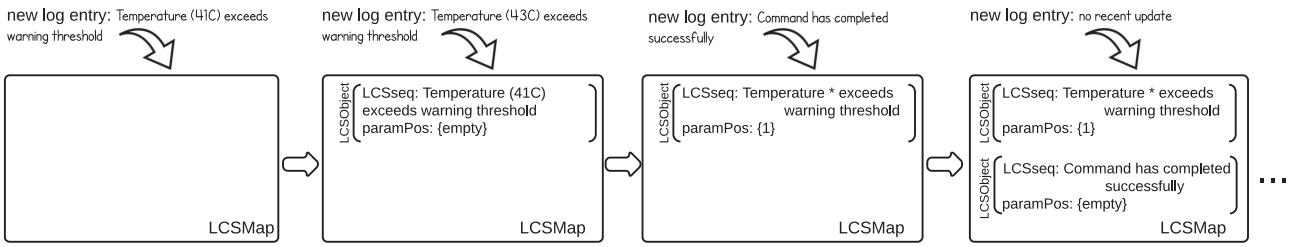


Fig. 3. Basic workflow of Spell.

and iterative partitioning methods, is that we could still derive a particular message type *even with very few instances of log entries produced by its log statement*. The other benefit of using the LCS approach is that, instead of relying on an offline batched approach that is utilized by all existing methods, it is possible to parse logs using a streaming approach, which is the main challenge we will describe next.

3.2 Basic Notations and Data Structure

In a log entry e, we call *each word a token*. A log entry e could be parsed to a set of tokens using system defined (or as user input) delimiters according to the format of the log. In general common delimiters such as space and equal sign are sufficient to cover most cases. After tokenization of a log, each log entry is translated into a "token" sequence, which we will use to compute the longest common subsequence, i.e., $\Sigma = \{\text{tokens from} e_1\} \cap \{\text{tokens from } e_2\} \cdots \cap \{\text{tokens from } e_L\}$. Each log entry is assigned a unique line id which is initialized to 0 and auto-incremented for the arrival of a new log entry.

We create a data structure called *LCSObject* to hold currently parsed LCS sequences and its related metadata information. We use *LCSseq* to denote a sequence that's the LCS of multiple log messages (also known as an LCS sequence), which, in our setting, is a candidate for the message type of those log entries. That said, each *LCSObject* contains an *LCSseq*, and a list of parameter positions called *paramPos* that indicates where in the LCSseq are the place holders for parameter values (the positions of `*`) with respect to each log entry. Finally, we store all currently parsed LCSObjects into a list called *LCSMap*. When a new log entry e_i arrives, we first compare it with all LCSseq's in existing LCSObjects in LCSMap, then based on the results of these comparisons, either assign an existing LCSObject as its message type, or compute a new LCSObject and insert it into LCSMap.

3.3 Basic Workflow

Our algorithm runs in a streaming fashion, as shown in Fig. 3. Initially, the LCSMap list is empty. When a new log entry e_i arrives, it is first parsed into a token sequence s_i, using a pre-defined set of delimiters. After that, we compare s_i with the LCSseq's from all LCSObjects in the current LCSMap, to see if s_i "matches" one of the existing LCSseq's or we need to create a new LCSObject and insert it into LCSMap. This basic workflow is captured in Algorithm 1. Next we will explain each step in detail.

Get New LCS. Given a new log sequence s produced by the tokenization of a new log entry e, we search through LCSMap. For the ith LCSObject, suppose its LCSseq is q_i, we compute the value ℓ_i, which is the length of the LCS(q_i, s). While searching through the LCSMap, we keep the largest ℓ_i value and the index to the corresponding LCSObject. In the end, if $\ell_j = \max(\ell_i' s)$ is greater than a threshold τ (by default, $\tau = |s|/2$, where $|s|$ denotes the length of a sequence s, i.e., *number of tokens* in a log entry e), we consider the LCSseq q_j and the new log sequence s having the same message type. The intuition is that the LCS of q_j and s is the maximum LCS among all LCSObjects in the LCSMap, and the length of LCS(q_j, s) is at least half the length of s; hence, unless the total length of parameter values in e is more than half of its size, which is very unlikely in practice, the length of LCS(q_j, s) is a good indicator whether the log entries in the jth LCSObject (which share the LCSseq q_j) share the same message type with e or not (which would be LCS(q_j, s)).

Algorithm 1. Spell

Input: raw system log entries arriving in a streaming fashion
Output: LCSMap
  init LCSMap as a list of LCSObjects and is initially empty;
  while a new log entry e arrives **do**
    parse e to a log token sequence s;
    newLCS= LCSsearch(s, LCSMap); // *details later*
    if newLCS is not NULL **then**
      updateLCSMap(newLCS, LCSMap);
    end if
  end while
  return LCSMap;

If there are multiple LCSObjects having the same max ℓ values, we choose the one with the smallest $|q_j|$ value, since it has a higher set similarity value with s. Then we use *backtracking* to generate a new LCS sequence to represent the message type for all log entries in the jth LCSObject and e. Note when using backtracking to get the new LCSseq of q_j and s, we mark '`*`' at the places where the two sequences disagree, as the place holders for parameters, and consecutive adjacent '`*`'s are merged into one '`*`'. For instance, consider the following two sequences:

q_j =`Command Failed on: node-235 node-236`
s =`Command Failed on: node-127`

Use backtracking to get the new LCSseq of these two, the result would be: `Command Failed on: *`. We can easily prove that this backtracking method gives LCS(q_j, s). Once this is done, we update the LCSseq of the jth LCSObject from q_j to LCS(q_j, s), and assign the jth LCSObject to e as its message type.

If none of the existing q_i's shares an LCS with s that is at least $|s|/2$ in length, we create a new LCSObject for e in LCSMap, and set its LCSseq as s itself. The entire LCSsearch algorithm for getting new LCS is shown in Algorithm 2.

Add New LCS to Current LCSMap. This step is pretty straight-forward. After the new LCS sequence has been calculated from last step, if it exists, the related LCSObject is update in LCSMap, otherwise we create a new LCSObject using the new log sequence and add it to LCSMap.

Algorithm 2. LCSsearch

Input: the new log sequence s and current LCSMap
Output: a new LCS sequence newLCSseq (could be empty)
  init maxℓ=0;
  init minoldℓ=MAX_INT;
  init candidate pointer c to NULL;
  for each LCSObject$_i$ in LCSMap **do**
    $\ell_i = |\text{LCS}(q_i, s)|$; //$q_i$ *is the LCSseq of LCSObject*$_i$
    if (maxℓ $<$ ℓ_i) or (maxℓ==ℓ_i and minoldℓ $>$ $|q_i|$) **then**
      maxℓ $= \ell_i$;
      minoldℓ $= |q_i|$;
      set $c = i$;
    end if
  end for
  if maxℓ $< |s|/2 + 1$ or maxℓ $< |q_c|/2 + 1$ **then**
    assign NULL to newLCSseq, -1 to c, and return;
  else
    use backtracking to get the LCS(q_c, s);
    return newLCSseq=LCS(q_c, s) and c;
  end if

This completes the basic procedures in Spell, and many standard logs could already be successfully parsed using this method. Nevertheless, we can further improve its efficiency and effectiveness, especially for the cases when message types differ only a little bit, or for logs having many parameters.

3.4 Improvement on Efficiency

In this section we show how to achieve nearly optimal time complexity for most incoming log entries (i.e., linear to $|s|$, the number of tokens in the token sequence s of a log entry e). In our basic method in Section 3.3, when a new log entry arrives, we'll need to compute the length of its LCS with each existing message type. Suppose each log entry is of size $O(n)$ for some small constant n (i.e., $n = |s|$), it takes $O(n^2)$ time to compute LCS of a log entry and an existing message type (using a standard dynamic programming (DP) formulation). Let m' be the number of currently parsed message types in LCSMap. The method in section 3.3 leads to a time complexity of $O(m' \cdot n^2)$ for each new log entry.

Note that since the number of possible tokens in a complex system could be large, we cannot apply techniques that compute LCS or MLCS efficiently by assuming a limited set of alphabets [26], [27], i.e., by assuming small $|\Sigma|$ values.

A key observation is that, for a vast majority of new log entries (over 99.9 percent in our evaluation), their message types are often already present in currently parsed message types (stored by LCSMap). Hence instead of computing the LCS between a new log entry and each exiting message type, we adopt a pre-filtering step to find if its message type already exists, which reduces to the following problem:

For a new string σ and a set of current strings $strs = \{str_1, str_2, \ldots str_m\}$, find the longest str_i such that $LCS(\sigma, str_i) = str_i$, and return true if $|str_i| >= \frac{1}{2}|\sigma|$.

In our problem setting, each string is a set of tokens and we simply view each token as a character.

1) *Simple loop approach.* A naive method is to simply loop through $strs$. For each str_i, maintain a pointer p_i pointing to the head of str_i, and another pointer pt pointing to the head of σ. If the characters (or tokens in our case) pointed to by p_i and pt match, advance both pointers; otherwise only advance pointer pt. When pt has gone to the end of σ, check if p_i has also reached the end of str_i. A pruning can be applied which is to skip str_i if its length is less than $\frac{1}{2}|\sigma|$. The worst time complexity for this approach is $O(m \cdot n)$.
2) *Inverted list approach.* To avoid going through the entire $strs$ set, we use an inverted index [28] based approach which could skip checking many str_is. The inverted index I is built over $strs$, as shown in Fig. 4. For each unique character in $strs$, record its position $(i, pos\_in\_str_i)$ in all str_is that it appears in, sorted by i. Given this index I, the procedure to find the subsequence of σ is as follows
   i) For each character in σ, find if it presents in I. If not, then treat that character as a parameter and skip; otherwise assign a unique, auto-increment id to the matching inverted list (e.g., $1, 2, 3$ on each arrow in Fig. 4 for A, B, and C).
   ii) For such matching lists, scan them using a sort-merge join style method. Instead of trying to find equal items across all lists (as that in sort-merge join), we try to find, by following the order of assigned ids to these lists, if any column with the same string id i in the inverted index matrix could form a subsequence of σ. For example in Fig. 4, (1,1) from the list for A, (1,2) from the list for B, and (1,3) from the list for C follow the order of assigned ids *and* share the same string id value 1 *and* form a str_i that's a proper subsequence of σ (by checking if the position values are properly ordered). Note that this step will return *all* str_is in $strs$ that satisfy $str_i = LCS(\sigma, str_i)$.
   iii) For all str_is returned by last step, find the longest one and check if its length is greater than $\frac{1}{2}|\sigma|$. This completes the inverted index lookup procedure. In this approach, all str_is are considered, and the time complexity is only $O(c \cdot n)$, where c is the average length of each inverted list, i.e., the average number of duplicated characters in the set $strs$.
3) *Prefix tree approach.* Another possible approach is to use a prefix tree [29]. In particular, we index str_is in $strs$ using a prefix tree, and prune away many candidates.

An example is shown in Fig. 5 where $strs = \{ABC, ACD, AD, EF\}$, and they are indexed by a prefix tree T. Instead of checking σ against every str_i in $strs$, we first check tree T and see if there is an existing str_i *that is a subsequence of* σ. If such a str_i is identified, we apply the length filter (i.e., check if $|str_i| > \frac{1}{2}|\sigma|$). As shown in Fig. 5, suppose σ=*ABPC*. Then by comparing each character of σ with each node of T, we could efficiently prune most branches in T, and mark the characters in σ that do not match any node in T as parameters in a message type. In this case, we will successfully identify ABC as the message type for σ, and P as its parameter.

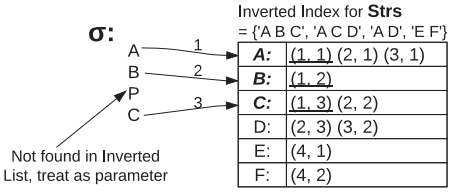


Fig. 4. Find the subsequence of σ using inverted index.

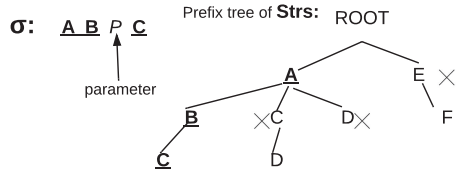


Fig. 5. Find the subsequence of σ using Prefix Tree.

For most log entries, it is highly likely that their message types already exist in tree T, so Spell will stop here, and the time complexity is only $O(n)$. This is optimal, since we have to go through every token in a log entry at least once. However, this approach only guarantees to return a str_i if such $str_i = LCS(\sigma, str_i)$ exists. In contrast to the inverted list approach, it *does not* guarantee that the returned str_i is the longest one among all str_is that satisfy $str_i = LCS(\sigma, str_i)$. For example, if σ=*DAPBC* while $strs = \{DA, ABC\}$, the prefix tree returns DA instead of ABC.

In practice, we find that although the prefix tree approach does not guarantee to find the longest message type, its returned message type is almost identical to the results of simple loop and inverted list methods. That's because parameters in each log record tend to appear near the end. In fact one of the state-of-art offline methods [1] finds message types by using weighted edit distance and assigns more weight to the token closer to end as parameter position. In particular, the evaluation results show that for the Los Alamos HPC log with 433,490 log records, for each new log entry, the message type returned by the prefix tree approach (if found), is 100 percent equal to the results returned by the simple loop and inverted list methods. But there also exist cases where the returned message type by prefix tree is fewer than $\frac{1}{2}$ number of tokens ($\frac{1}{2}|s|$) for a new log entry e while e's message type still already exists in LCSMap.

That said, the *complete pre-filtering step in* Spell *is*, for each new log entry e, first find its message type using prefix tree, and if not found, apply the inverted index lookup. In our evaluation section we compare Spell with pre-filtering with the naive LCS method (i.e., computing LCS between e and every existing message type using $O(mn^2)$ time), which shows that Spell with our pre-filtering step produces almost equally good results for all logs but with much less cost.

For log entries (fewer than 0.1 percent in our evaluation) that do not find message types using the pre-filtering step, we have to compare the new log entry e with all existing message types to see if a new message type could be generated. However, instead of computing LCS between each message type q and e (and then find the length of this LCS), we first compute their *set similarity score* using Jaccard similarity. Only for those message types that have more than half common elements (i.e., tokens) with e do we compute their LCS. Then if their LCS length exceeds $\frac{1}{2}|s|$, we adjust that message type, and adjust inverted index I and prefix tree T accordingly. Otherwise e is a new message type, so we simply create a new LCSObject in LCSMap, as well as update I and T.

3.4.1 Parallelization

Spell is embarrassingly parallel on a server with multi-core processors. For instance, the new log entry could be compared with each message type concurrently.

As previously mentioned, in large scale log datasets used in our evaluation, which are generated by real world production systems, for over 99.9 percent new log entries, their message types already exist in current LCSMap. Only for the remaining fewer than 0.1 percent log entries, does LCSMap needs to be updated, either by adding a new LCSObject or updating an existing LCSObject. Therefore, to process all log entries, LCSMap only needs to have exclusive update operations for less than 0.1 percent of the times; all others are read operations which could be done in parallel. We further design a parallel version of Spell using readers-writer lock [30], where a write lock needs to be acquired exclusively while multiple read locks could be acquired simultaneously. The modified basic workflow is presented in Algorithm 3. All differences from Algorithm 1 are written in italic. Also, the inverted list and prefix tree data structures presented in Section 3.4 only need to be updated when LCSMap does. Hence, the related write operations could be viewed as a procedure inside function "updateLCSMap", while read operations inside "LCSsearch" in Algorithm 3.

Algorithm 3. Spell (A Parallel Version)

Input: raw system log entries arriving in a streaming fashion; *global readers-writer lock RWLock*
Output: LCSMap
init LCSMap as a list of LCSObjects and is initially empty;
create a thread pool with a fixed size equal to number of CPU cores;
while a new log entry e arrives **do**
assign a thread from thread pool to execute:
parse e to a log token sequence s;
Acquire read lock RWLock;
newLCS= LCSsearch(s, LCSMap);
Release read lock RWLock;
if newLCS is not NULL **then**
Acquire write lock RWLock;
updateLCSMap(newLCS, LCSMap);
Release write lock RWLock;
end if
end while
return LCSMap;

3.5 Improvement on Effectiveness

The Spell method as presented already achieves a significantly better score on all effectiveness measures (e.g., F-measure and Accuracy), as we will show in evaluation. For a small portion of message types that Spell fails to recognize, we further improve its effectiveness through refinement techniques. Such refinement takes place when the number of processed log entries has reached a user-defined threshold, whenever the system is idle or under-utilized (e.g., arrival rates of new log entries become low), and/or right before the message type lookup procedure completes. The refinement is achieved via two steps, namely, *split* and *merge*.

Split procedure. Consider the following log entries:

```
boot (command 1880) Error: Console-Busy Port
already in use
boot (command 2359) Error: Console-Busy Port
already in use
wait (command 3964) Error: Console-Busy Port
already in use
```

By the basic Spell as illustrated so far, the message types extracted would be a single message type:

```
* (command *) Error: Console-Busy Port already
in use
```

However the correct message types should be:

```
boot (command *) Error: Console-Busy Port already in use
wait (command *) Error: Console-Busy Port already in use
```

We observe that, as stated in previous literature [1], [9], the *token positions that contribute to message types have fewer number of unique tokens*. By utilizing this feature, we introduce a split method that is applied to each LCSObject in LCSMap. For this purpose, we introduce another field in an LCSObject called *params* which contains key-value pairs that store all parameter values (as keys), seen from the history, at each parameter position, and how many times each parameter value has been seen (as values). These key-value pairs can be easily produced and/or updated during the backtracking process to produce LCS.

In particular, for each LCSObject O, O has a paramPos list that contains the token positions as place holders for parameters for the message type LCSseq of O. So the basic idea of *split* is to leverage the above observation and split some existing parameters, and include the parameter values into message type(s). Specifically, for each LCSObject O in LCSMap, we go through O.params, its key-value pair parameter list, and count the number of unique tokens at each parameter position. We find the position having the least number of unique tokens. If the number of unique tokens in that position is less than a split threshold, and no token contains any digit value, we consider that position will contribute to a message type. In that case we will split O to several new LCSObjects, with that parameter position in O.LCSseq being replaced by a unique token in the same position in each new LCSObject's LCSseq.

For example, in the above example, the first parameter position has only 2 unique tokens, while the second parameter position has many (in the actual log file). So the first parameter position has a high probability being part of a message type. The split procedure will replace the first parameter '*' in the LCSseq with each unique token (`boot` and `wait` in this example) at the same position, leading to two new LCSObjects with two new LCSseqs as above.

Merge Procedure. Spell works perfectly with well formatted log messages, where the number of tokens in message types is more than that in parameters, and parameters vary in different log messages. Recall that Spell uses a threshold to distinguish whether the LCS of two sequences is a valid message type; there could be situations where the number of parameters in one log entry is so many such that this threshold based approach cannot extract the message type properly. Consider the case below:

```
Fan speeds ( 3552 3534 3375 4354 3515 3479 )
Fan speeds ( 3552 3534 3375 4299 3515 3479 )
Fan speeds ( 3552 3552 3391 4245 3515 3497 )
Fan speeds ( 3534 3534 3375 4245 3497 3479 )
Fan speeds ( 3534 3534 3375 4066 3497 3479 )
```

Clearly, they should lead to a single message type:

```
Fan speeds: ( * )
```

However, each of the log messages contains 10 tokens, and the message type has only 4 tokens (including parentheses). The message types output by Spell as presented so far is:

```
Fan speeds ( * 3552 * 3515 * )
Fan speeds ( 3534 3534 3375 * 3497 3479 )
```

Though their LCS has correctly identified `Fan speeds`, its length is less than the threshold τ we used to qualify an LCSseq as a message type. So in this case, Spell will identify the above example as two distinct message types.

To address this problem (when the number of parameters in a log entry is too many), we introduce a merge procedure, which first clusters current LCSObjects that might have the same message type together, and then count the number of distinct tokens at each position of LCSseqs from these LCSObjects. In particular, we partition LCSMap into a set of clusters, until each cluster satisfies the followings:

i) For a subset of token positions, at each such position, there is only one unique token for all LCSObject.LCSseq at this position from this cluster. For example, in previously mentioned example, for different message types in one cluster, all log tokens at the first and second positions should be the same: `Fan` and `speeds`.

ii) For every other token position, the number of unique tokens of all LCSObject.LCSseq from this cluster exceeds a merge threshold. The merge threshold is different for positions having numbers and positions having only strings. Currently we consider if any token in one position has digits in it (might be ids or values), then that position could be a parameter position, which works pretty well in practice. For instance, in previous example, except `Fan speeds ()`, all other positions contain different values of digits.

After this partition step, we consider all LCSObjects inside one cluster have the same message type. So we merge them into a single LCSObject, assign the positions from (ii) as parameter positions, and the tokens from (i) are inserted into LCSseq as part of the message type. As in the above example, the positions having only one unique token are positions 1, 2, 3, 10, i.e., `Fan speeds: ()`, and the rest having many unique tokens are identified as parameter positions.

3.6 Complexity Analysis

Originally, with the basic workflow of Spell shown in Fig. 3, when a new log entry arrives, we first try to find if its message type exists in current LCSMap, and update LCSMap if not. With the improvements proposed in Sections 3.4 and 3.5, when a new log entry arrives, we first search if a path in prefix tree (i.e., an existing message type) serves the purpose of its message type; if not, we further find its message type in inverted index; finally, if its message type is still not found, we would compare it with all message types in LCSMap using naive dynamic programming approach, and update LCSMap accordingly. Split and merge procedures could be applied periodically to clean up current parsed message types, or simply after all log entries are generated.

In this section, we analyze Spell's complexities with all improvements applied.

3.6.1 Time Complexity

Spell ensures that the size of LCSMap increases by one *only when* a new message type has been identified; otherwise, an existing LCSObject will be assigned to a new log entry, with an updated message type if necessary. This guarantees that

LCSMap size is at most the number of total message types (which is m) that could be produced by the corresponding source code, which is a constant. At the beginning of Section 3.4, we've shown that for the basic Spell, the time complexity for each new log entry is $O(m \cdot n^2)$, since the naive dynamic programming method to compute LCS between a log entry and a message type is $O(n^2)$ for log entries of size $O(n)$, whereas our backtracking method is often cheaper and we only do it with a target message type in LCSMap which has the longest LCS length with respect to the new log entry and the length exceeds a threshold, thus it is computed at most once for each new log message.

With the pre-filtering step, for each log entry, we'll first try to find its message type in prefix tree, then inverted index, and only for the small portion that are still not located, the LCSMap needs to be compared against. For L log records, suppose the number of log records that fail to find message types in pre-filtering step is F, and the number of log records that are returned in inverted index step is I. The amortized cost for each log record is only $O(n + \frac{(I+F)}{L} \cdot c \cdot n + \frac{F}{L} \cdot m \cdot n^2)$, where c is the average number of duplicated tokens in all message types, m is the number of message types and n is the log record length. In our evaluation, $\frac{(I+F)}{L} < 0.01$ and $\frac{F}{L} < 0.001$, thus the cost for each log record to find its message type in Spell is approximately only $O(n)$ in practice.

For the complexity of the split method, note that the split threshold is a small constant, so the total size of an adjusted LCSMap is at most a small constant times of the previous LCSMap size, which is still $O(m)$. The split method itself clearly takes only $O(m)$ cost. In the merge procedure, the LCSMap is first repeatedly partitioned until each cluster satisfies the conditions to be merged. Note this is done over currently parsed message types. Hence, the partition cost is at most the number of currently parsed message types, $O(m)$, and the LCSMap size only reduces after a merge step. Note that the split/merge heuristics are executed very infrequently (only once in our evaluation), thus their impacts to the amortized cost on each log entry could be mostly ignored.

3.6.2 Space Complexity

Spell keeps certain data structures to store message types parsed so far, such that incoming new log entries could be compared against. As mentioned in Section 3.6.1, the number of message types stored in LCSMap is in the order of m, which is the number of log printing statements in source code, i.e., a constant. A message type is a sequence of tokens, e.g., `Temperature * exceeds warning threshold`, where the average number of tokens per message type is typically fewer than 100, and the average length of characters per token is typically fewer than 10. Therefore, suppose the memory allocated for each character is 1 byte, it only takes less than 1 kilobyte to store one message type on average. Thus, the total memory to store all message types in LCSMap is $O(m)$ kilobytes.

Section 3.6.1 proposes to use prefix tree and inverted index to improve Spell's efficiency. These are index structures built on top of all existing message types in LCSMap, which get updated only when LCSMap does. For prefix tree, many log tokens are shared across message types. As shown in Fig. 5, to store message types *A B C; A C D; A D; E F* (10 tokens in total), the prefix tree only needs to allocate memory for tokens *A B C C D D E F* (8 tokens in total). Although prefix tree introduces new memory usage to store relevant pointers at each node, the added memory together is still $O(m)$ kilobytes. The case is similar for inverted index. As shown in Fig. 4, to store message types *A B C; A C D; A D; E F* (10 tokens in total), inverted list only needs to allocate memory for all *distinct* tokens across all message types, which are *A B C D E F* in this case (a total of 6), and position values pairs, which have the same amount as the number of tokens in LCSMap. To summarize, by adding prefix tree and inverted list for efficiency improvement, the extra memory introduced is still $O(m)$ kilobytes, i.e., in the same order of storing all message types in LCSMap.

Besides the essential memory usage to store message types in LCSMap and other data structures, Spell also needs certain temporary memory, e.g., in order to use dynamic programming to compare a new log entry with each existing message type. Also, for split and merge procedures, Spell needs to store (parameter value, occurrence) pairs for each parameter position, as described in Section 3.5, which is only stored in LCSMap, not prefix tree or inverted index. What's more, for the purpose of our paper evaluation, we also store line ids for each log entry to indicate which message type it belongs to. The summed memory usage besides message types could be up to megabytes in our evaluation.

To summarize, the memory cost to store message types is $O(m)$ kilobytes, which includes all message types storage in LCSMap, and the whole memory usage of prefix tree and inverted index, where m is the number of log printing statements in the source code (a constant), and is typically up to hundreds. However, the memory cost to store parameter values and line ids could be up to megabytes, making the added memory of prefix tree and inverted index ($O(m)$ kilobytes) negligible.

3.7 Semantics Inference of Parameter Fields

Besides the ability to accurately extract message types from raw log messages, Spell is also capable of automatically inferring a semantic meaning for each parameter position. Not only does this help end users to understand system status with the summarized view of message types provided by Spell (as in Table 1), but it also provides insights for root cause analysis with relevant message types diagnosed by further data analytics schemes.

A semantic meaning for a parameter position is simply the type (or nature) of parameter values that position holds. For example, for log message `Temperature (41C) exceeds warning threshold`, the message type is `Temperature * exceeds warning threshold` and the semantic meaning for the parameter position `*` is `temperature`. The semantic meaning for this particular case is easy to infer, as indicated by the noun prior to the parameter position. However, in many cases it is difficult to infer the semantic meanings of a parameter field simply based on the associated message type itself. For instance, for log message `Command Failed on: node-235 node-236`, the message type is `Command Failed on: *`. Although human can easily make a guess in this case for the semantic meaning based on the parameter values, e.g., the semantic meaning of this parameter is "node(s)", it is not obivious for a parsing algorithm to automatically infer a semantic meaning only based on the message type (i.e., `Command Failed on: *`).

To arm Spell with the ability of recognizing parameter semantics based on parameter values, we adopt the concept

of "named-entity recognition (NER)" [31] from natural language processing (NLP), and design a two-step procedure for Spell to automatically generate semantic meanings for parameter positions.

Step 1. There are cases where the semantic meaning of a parameter field could be inferred from the message type it associates with. For example, `temperature` in the above example. The question is which word in a message type do we pick up as a candidate for parameter semantics, and under what condition that word could be used as a semantic meaning. Our observations are as follows. First, it is obvious that a word next to (prior or after) a parameter position is mostly related to the meaning of that position, e.g., `temperature *` and `* seconds`. Second, if the word is a noun or a non-standard English word, it most likely represents the semantic meaning for that parameter position. The reason for "non-standard English word" is that it might be a system-specific name. For example, in OpenStack log message type `Total usable vcpus: *`, "vcpus" means "virtual CPUs", which is a term used for Virtual Machines. Based on these two observations, if either the word prior or after a parameter position is a noun or a non-standard English word, we would add it as a possible semantic meaning for that position. A parameter position may have two potential semantic meanings if both words prior and after it satisfy our criteria. This is acceptable because the purpose of semantic meaning extraction is to help users understand the summarized message types, hence we should present both possibilities to users if the algorithm is not certain which one it should be.

Step 2. For other cases when a semantic meaning cannot be found using Step 1, such as `Command Failed on: *`, we leverage the ideas used in entity resolution to generate parameter semantics. The concept of entity resolution is to identify whether two entities are in fact representations of the same object, by comparing the similarities of underlying attributes of these two entities. We could treat a parameter position as an entity and all parameter values at that position as its underlying attributes. With that, we could find out whether two parameter positions have the same semantic meaning by comparing the similarities of their parameter values. Then, if two parameter positions have the same semantic meaning after entity resolution, of which one has been assigned a semantic meaning in Step 1, the remaining unknown one could be given the same meaning as the known one. Specifically, for each parameter position, all parameter values belong to that position form a group. We use the semantic meaning of a parameter position to denote the name of its associated group. Then the task becomes to find representative names for those unnamed groups. For example, for message type `Targeting nodes: *`, the last parameter position has an associated group with a name "node" and a group of parameter values, e.g., {`node-235, node-236, node-231`}. Our idea is as follows. For each unnamed group, we find possible names by checking its intersection with each of the named groups. If an intersection size is large enough (e.g., more than half of the unnamed group size), then we use the name of the corresponding named group as a potential candidate for the unnamed group. For instance, for message type `Command failed on: *`, the last parameter position has an unnamed group C which contains parameter values of {`node-235, node-236, node-230`}. By comparing C with an named group called "node", which has values of {`node-235, node-236, node-231`}, we find the intersection of the two is {`node-235, node-236`}, larger than half of C's size, so the name "node" could be used to denote group C, which represents the semantic meaning of the last parameter position in message type `Command failed on: *`. In this step, we compare each pair of groups, and assign all possible names as potential semantic meanings for a group, i.e., a parameter position. Similar with Step 1, one parameter position could have multiple semantic meanings.

We evaluate this two-step procedure in Section 5.3 on various types of system logs, which shows that it is able to effectively resolve semantic meanings for most of the parameter positions.

4 Remarks

Parsing each log message to extract their message type, though a vital step for many further data analysis, is not an easy task. It should be noted that no automatic approach is perfect for all possible logs. For example, even the approach that extracts log schema from the source code [5] that produces the corresponding log in the first place cannot achieve 100 percent accuracy. We'll show in our evaluation that Spell with split and merge has achieved over 90 percent accuracy in most cases, and even close to 100 percent, without the access to any source code.

We'd also like to highlight that the LCS-based construction is not similar to the edit distance based approach as presented previously in [1]. A crucial difference of the two is that the LCS sequence of two log messages is naturally a message type, which makes streaming log parsing possible.

Note that, stack traces and/or minidumps generated accompanied with system errors compose an important part of system logs. Although particularly interesting for error diagnosis, these are not the focus of Spell. The purpose of Spell log parser is for *automatic* system log analysis (e.g., automatic anomaly detection), or to provide a summarized view for end users while numerous log entries are being generated each second. However, stack traces are typically used for *post-analysis* after a known error, and are too important to be summarized. Thus, although Spell is capable of summarizing system logs based on their similarity whether they are event console logs or minidumps, currently we only focus on event console logs that are automatically generated by log printing statements in the source code.

It is straight-forward to integrate Spell into the open-source ELK stack. Instead of parsing logs using user-defined regular expressions, we could do automatic message type parsing upon each log entry's arrival. Existing ELK stack only works as a key word search engine. With the streaming structured log parsing feature provided by Spell, ELK has the potential to become a full-fledged log data warehouse and is able to provide end-users with much more sophisticated operations such as roll-up and drill-down, as in a traditional data warehouse setting for structured data, through its Kibana interface. What's more, this happens while new log entries continue to arrive in a streaming fashion.

We also want to highlight that Spell has been successfully applied in an automatic log anomaly detection system named DeepLog [32]. Because Spell is capable of extracting message types in an online fashion without any offline preprocessing, it is a natural fit to such online anomaly detection systems which are agnostic to log formats.

TABLE 2
Parameters for All Three Algorithms

Spell
message type threshold τ = 0.5
split threshold = 8
merge threshold = 10

CLP
edit distance weight ν = 10
cluster threshold ς = 5
private contents threshold ϱ = 4

IPLoM
file support threshold = 0.01
partition support threshold = 0
lower bound = 0.1
upper bound = 0.9
cluster goodness threshold = 0.34
Drain
depth = 3~4
st (similarity threshold) = 0.3~0.5

5 Evaluation

In this section, we evaluate the efficiency and effectiveness of Spell, by comparing it with previous popular and state-of-the-art log parsing algorithms, on multiple real log datasets with different formats. All experiments were executed on a Linux machine with an 8-core Intel(R) Core(TM) i7-3770 CPU @ 3.40 GHz computer. We'll show that Spell not only is able to parse logs in an online streaming fashion *using no prior knowledge like accessing source code*, but also has outperformed the competing offline methods in terms of both efficiency and effectiveness.

We compare Spell with three log parsing methods: IPLoM [9], [33] (state-of-the-art offline algorithm), an offline clustering-based log parser [1] which we refer to as CLP, and a fixed-depth tree online log parser called Drain [14]. The idea of IPLoM is to partition the entire log into multiple clusters, where each cluster represents a set of log entries printed by the same print statement. The partition is done using a simple 3-step heuristic: first partition by each log record length; then partition each cluster by the token position having least distinct tokens; and finally partition by the bipartite mapping between tokens in each cluster. It is so far the most lightweight offline automatic log parsing algorithm. CLP, on the other hand, is a frequently used algorithm as the first step of multiple log mining efforts [1], [2], [3], [7]. It also partitions the log into clusters, while by first clustering using weighted edit distance, and then repeatedly partitioning until all clusters satisfy the heuristic - each position either has the same token, or is a parameter position. Similar to Spell, Drain utilizes a tree structure to guide log template search. Unlike Spell, its first step is to preprocess by domain knowledge, i.e., removing tokens matching specific regular expressions. Drain maintains a fixed-depth tree where the first layer nodes represent log entry length, whose subtrees are the paths of first several tokens (e.g., first 3 tokens) of different log messages. Finally, each leaf node contains multiple log groups to match with.

Table 2 shows the default values of key parameters used for each algorithm. For parameters with recommended values that were clearly stated in the original papers, such as all parameters for IPLoM [9], we simply adopt those values. For others that were not clearly specified, we tested the corresponding method with different values until we get the best result (for the same log data) as reported for that method in the original paper. Note that Drain uses different parameters for different log datasets, details of which could be found in [14], whereas Table 2 shows parameter ranges.

The log datasets used are shown in Table 3. Besides the two supercomputer logs that were commonly used for evaluation by previous work [1], [2], [3], [7], [9], [33], we also used log datasets from more recent popular systems: HDFS log and OpenStack log. HDFS is a widely used file system and its log datasets have been analyzed for automatic log anomaly detection in multiple works [2], [5], [12], where the log parsing part is done either by using source code template or offline processing. OpenStack is a popular open source cloud infrastructure. We setup an experiment in CloudLab [34], run a script to repeatedly create project, network, virtual machine, and run various virtual machine tasks for a week. A total of 106,838 log records are generated by OpenStack's four major components: Nova, Neutron, Keystone and Ceilometer. We find the corresponding message types from OpenStack's source code, and use regular expressions to match each log entry with its message type to generate the ground truth to be compared with.

TABLE 3
Log Datasets (Count: The Total Number of Log Entries)

Log type	Count	Message type ground truth
Los Alamos HPC log[1]	433,490	available online[2]
BlueGene/L log[1]	4,747,963	available online[3]
HDFS log	100,000	obtained from Xu etc. [5]
OpenStack Cloud log	106,838	parsed from source code

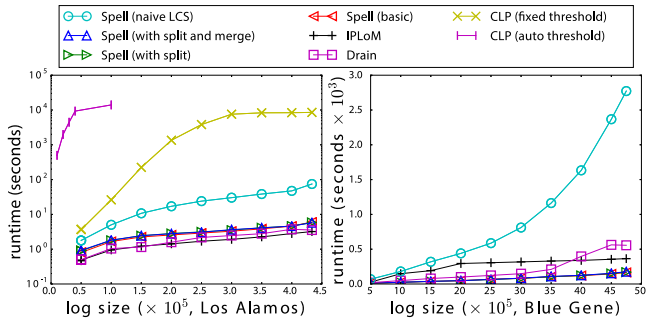


Fig. 6. Efficiency comparison of different methods.

5.1 Efficiency of Spell

Fig. 6 shows the total runtime of different methods when log size (the number of log records) grows bigger. Note that we tested different alternatives of the Spell method:

- Spell (naive LCS): compute the LCS of a new log entry and each existing message type using DP.
- Spell (basic): Spell with the pre-filtering step.
- Spell (with split): Spell with pre-filtering and split.
- Spell (with split and merge): Spell with pre-filtering and split and merge.

Fig. 6 left shows the results on Los Alamos HPC Log. *Note that runtime is measured by logarithm scale*. To parse the entire log with 433,490 entries, Spell with naive LCS is about 75 seconds while it's only 5 seconds with pre-filtering. For Spell with pre-filtering, adding either split or merge or both does not introduce significant overhead; their runtime are nearly identical to Spell with pre-filtering. But running split/merge, however, helps to improve the effectiveness measures as we'll see later.

1. CFDR Data, https://www.usenix.org/cfdr-data
2. Los Alamos National Lab HPC Log message types, https://web.cs.dal.ca/~makanju/iplom/hpc-clusters.txt
3. BlueGene/L message types, https://web.cs.dal.ca/~makanju/iplom/bgl-clusters.txt

TABLE 4
Amortized Cost of Each Message Type Lookup Step in Spell (unit: milliseconds)

	prefix tree	inverted index	naive LCS
Los Alamos HPC log	0.006	0.015	0.175
BlueGene/L log	0.011	0.077	0.580

TABLE 5
Number (Percentage) of Log Entries Returned by Each Step

	Los Alamos HPC log	BlueGene/L log
prefix tree	397,412 (91.68%)	4,457,719 (93.89%)
inverted index	35,691 (8.23%)	288,254 (6.07%)
naive LCS	387 (0.09%)	1,990 (0.042%)

IPLoM shows the best efficiency which is similar to Drain, whereas Spell (with pre-filtering) is only slightly slower (within seconds). The CLP method has the worst efficiency (2-4 orders of magnitude slower than IPLoM and Spell). We tested two variants of CLP:

1) CLP (auto threshold): it automatically sets the cluster threshold ς by k-means clustering. When log size is bigger than 100,000, it's already too slow to run to completion.
2) CLP (fixed threshold): it uses a fixed threshold 5 calculated from smaller log file, which significantly improves the runtime. However it's still much slower than other methods. In later experiments we only use CLP with fixed threshold if applicable, as CLP with auto threshold is simply too expensive.

Fig. 6 right shows the results on Blue Gene Log. *The runtime in this figure is measured in normal decimal scale.* We didn't include CLP in this experiment: even CLP with fixed threshold is too slow to finish as the Blue Gene log has nearly 5 million entries.

Here the advantage of our pre-filtering step is clearly demonstrated. In particular, Spell with pre-filtering has outperformed IPLoM in terms of efficiency. With prefix tree, when the log size grows much faster than the number of message types, most log entries will find a match in prefix tree, and return immediately. Then for the majority of the rest, the message types could be found in inverted index. Only for a small amount of log records that are not matched in pre-filtering step, we will compare it with each existing message type. Noticeably, the runtime of Spell (naive LCS) increases exponentially. That's because when log size grows bigger, more message types also show up, and when each new log entry comes, it may need to be compared with a larger number of message types. This result clearly shows the importance of the pre-filtering step and how it has effectively mitigated the efficiency issues in the basic Spell method.

The amortized cost for each log entry to find its message type using different lookup method in the pre-filtering step is shown in Table 4 (in milliseconds). Recall that for each log entry, Spell first tries to find its message type in prefix tree, then inverted index, and finally uses naive LCS if not found in previous two steps. Table 5 shows the number (percentage) of log entries that are returned in each step, showing that over 91 percent could be processed in prefix tree in $O(n)$ time, and over 99.9 percent in total could be processed by prefix tree and inverted index combined. The expensive naive LCS computation is only applied to less than 0.1 percent of log entries. Hence much overhead is reduced by pre-filtering step, we'll show later that it provides almost identical results with the costly naive LCS method.

TABLE 6
Runtime on HDFS and OpenStack Logs (Unit: Seconds; CLP: with Fixed Threshold; Spell 1: Spell with Split; Spell 2: Spell with Split and Merge)

	CLP	IPLoM	Drain	Spell	Spell1	Spell2
HDFS	3786.83	5.76	2.12	2.80	2.90	2.99
OpenStack	21053.22	8.34	3.13	4.22	5.23	5.52

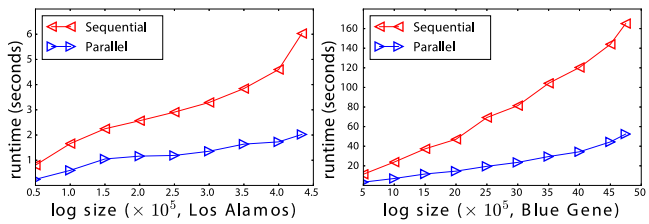


Fig. 7. Runtime for sequential and parallel implementations of Spell.

The runtime comparisons on HDFS and OpenStack log datasets are shown in Table 6, which shows similar results that Spell is as efficient as IPLoM and Drain, and much more efficient than CLP.

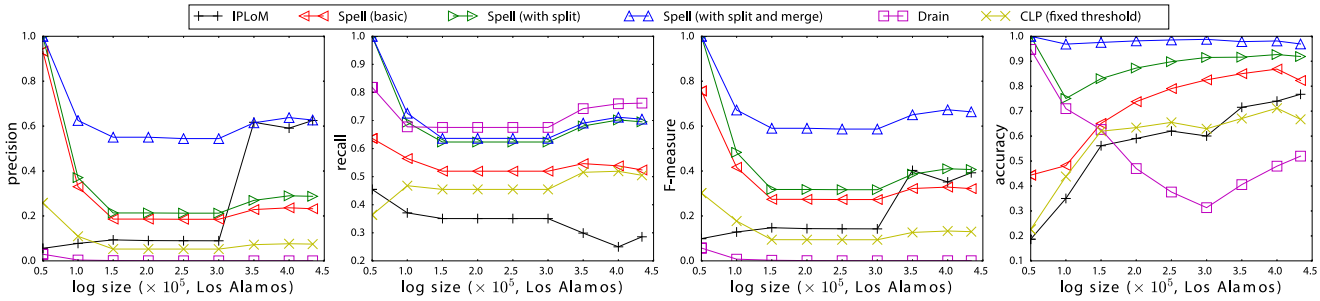
5.1.1 Parallelization Evaluation

To further improve Spell's efficiency, Algorithm 3 presents a parallel implementation of Spell, by utilizing CPU's multi-core structure. In this experiment, we simply allocate 8 threads to utilize 8 CPU cores on our machine. The thread pool and readers-writer lock are implemented using Boost C++ libraries [35]. Fig. 7 shows the performance of the parallel implementation compared with a sequential version. This seemingly naive implementation achieves significant runtime improvement, which shows Spell's potential to be improved further with finer granularity locks and on a more advanced platform.

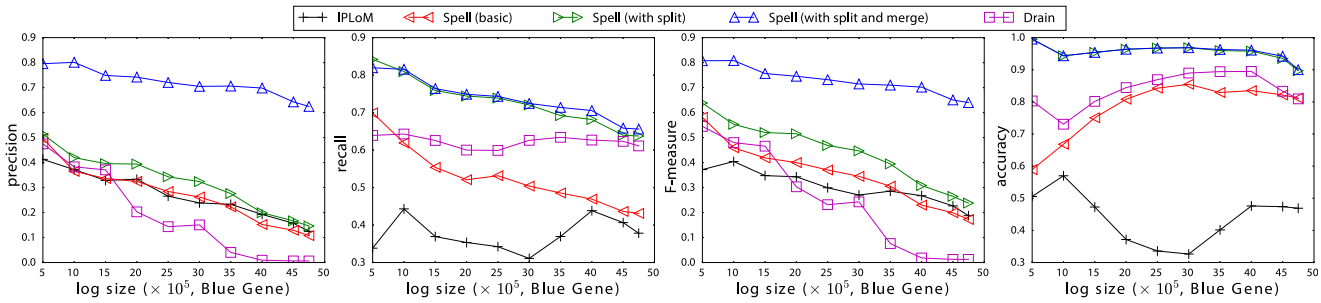
Note that parallelization may change the order of new log entries' updates to LCSMap, thus the final accuracy and F-measure could be different from the results of the sequential implementation. We compare the results and find the difference negligible, because of the abundant log messages there are for each message type.

5.2 Effectiveness of Spell

In this section we evaluate the effectiveness of Spell. After parsing, the log file is processed into multiple clusters, where each cluster represents one message type with the associated log records (as produced by the corresponding log parsing method). A parsed message type is considered as correct *if all and only if* all log records printed by that message type (as identified through the ground truth) are clustered together. We run each method, compare the results with the ground truth generated by matching each log entry with its true message type from Table 3, and calculate four measures to evaluate on: Precision, Recall, F-measure and Accuracy. Precision means among all the message types generated, how many match the true message types from ground truth. Recall is the percentage of "number of correct message types generated" over "total number of true message types in



(a) Los Alamos log



(b) Blue Gene log

Fig. 8. Effectiveness comparison of different methods.

ground truth". F-measure is a combination of Precision and Recall, which is $\frac{2 \cdot Precision \cdot Recall}{Precision + Recall}$. Finally, Accuracy indicates the total number of log entries that are parsed to correct message types over the number of total processed log records. Note that, Accuracy measures at a per log entry level, while all others measure at a per message type level. As is shown later, Accuracy is in general higher than Precision, which is intuitive because a message type is more likely to be parsed correctly if the number of log entries having that message type is big, due to more variations for parameter values.

Fig. 7 shows the comparison on supercomputer logs. With more log entries, number of message types also increases; and they don't necessarily show up uniformly over time. Hence, the effectiveness of a method does not necessarily show a steady trend as log grows. We can see that in almost all charts, Spell without merge/split already outperforms CLP, IPLoM and Drain. Spell with only split has further improved the effectiveness (and quite significantly in some cases). In the end, Spell with merge and split has consistently and significantly outperformed all other methods on the final measures of F-measure and Accuracy for both data sets.

More specifically, in terms of Precision, *Spell with merge* has greatly improved its performance in this measure compared with Spell without merge. That's because merge will merge multiple message types into fewer ones, which reduces the total number of message types generated, the denominator of the Precision measure. The sudden jump of IPLoM from point 3.0×10^5 to 3.5×10^5 in Fig. 8a is due to its clustering method. In IPLoM, if the size of one cluster is too small compared to the total log size (controlled by the "file support threshold" in Table 2), that whole cluster is grouped into an "outlier" cluster, which reduces the total message types being generated. When log size grows from 3.0×10^5 to 3.5×10^5, for some clusters, the percentage of each cluster size over whole log size becomes below the threshold, hence grouped into one cluster, leading to the boost in Precision. For Drain, since its fixed-depth tree is constructed on raw log entries, instead of log templates as in Spell, a significant amount of log messages cannot be matched and are treated as log templates (clusters) themselves. Hundreds of thousands of log clusters are generated because of this, rendering Precision to almost 0.

In terms of Recall, *Spell with split* has greatly improved its performance compared to basic Spell. This is because that some similar message types are parsed as one message type by LCS, and split helps significantly in this case.

F-measure shows that Spell without split and merge can already outperform other methods, and using split/merge could further improve its score. Spell achieves much better Accuracy than other methods. Spell with split and merge could achieve almost 100 percent accuracy. IPLoM Accuracy is acceptable in Fig. 8a for Los Alamos log, and becomes very poor in Fig. 8b for Blue Gene log. CLP does not achieve very good results for these HPC logs due to the facts that: 1) the cluster threshold ς is fixed to save running time; 2) a large portion of HPC logs only contain one word like "`running`" and "`down`", so their weighted edit distances could be small enough to be (wrongfully) clustered into one group. Drain's F-measure is close to 0 because of its low Precision due to too many log templates being generated.

Note that the pre-filtering step in Spell may miss an existing message type t for a new log entry e if $LCS(t, s) \neq t$ but $|LCS(t, s)| > |LCS(t', s)|$ when there is another existing message type t' that satisfies $t' = LCS(t', s)$, where s is the token sequence of e. To evaluate such potential degrade to the effectiveness due to the pre-filtering step, we show a comparison in Table 7. The result shows that Spell with pre-filtering has achieved an accuracy nearly the same as that using only naive LCS. This means the pre-filtering step has almost no downgrade effect on the parsing results though it greatly reduces the parsing overhead.

The effectiveness comparisons on HDFS and OpenStack logs are shown in Table 8. Our Spell method works remarkably better than CLP and IPLoM on all measures, especially on OpenStack log. We looked into the log in details to find

TABLE 7
Comparison of Spell with and without Pre-Filter
(#TM: Number of True Message Types Found)

Spell	pre-filtering	Los Alamos log #TM	Los Alamos log Accuracy	BlueGene/L log #TM	BlueGene/L log Accuracy
basic	False	55	0.8228	165	0.8118
	True	55	0.8228	164	0.8118
with split	False	73	0.9190	239	0.8955
	True	73	0.9190	238	0.8924
with split and merge	False	74	0.9692	247	0.9019
	True	74	0.9692	242	0.8946

TABLE 8
Effectiveness Measures on HDFS and OpenStack Logs
(Spell1: Basic Version; Spell2: with Split;
Spell3: with Split and Merge)

		CLP	IPLoM	Drain	Spell1	Spell2	Spell3
HDFS	Precision	0.0001	0.0423	0.4063	0.6842	0.6667	0.8421
	Recall	0.2353	0.5294	0.6842	0.7647	0.8235	0.9412
	F-measure	0.0002	0.0783	0.5098	0.7222	0.7368	0.8889
	Accuracy	0.0111	0.7758	0.9948	0.7768	0.7768	0.9994
OpenStack	Precision	0.0002	0.0000	0.0003	0.5238	0.4167	0.4762
	Recall	0.4444	0.0714	0.5	0.7857	0.7143	0.7143
	F-measure	0.0003	0.0001	0.0006	0.6286	0.5263	0.5714
	Accuracy	0.3687	0.0082	0.8599	0.9886	0.9818	0.9818

out the reason for this giant gap. Here are some typical log records. 1) `[req...] Host * has more disk space than database expected`. CLP fails to recognize the similarity of these logs because of the multiple request ids at the beginning of each log entry, and its heuristic believes parameters happen more likely later. 2) `Identity response: "error": "message": *, "code": *, "title": *`. This is a typical JSON format log message of OpenStack. The parameter after `message` could have variable length, so IPLoM's first heuristic which partitions by length splits this cluster into different parts. Drain has very good accuracy on OpenStack dataset but very poor Precision and F-measure, which is also because a parameter may happen at the beginning of a log message, thus being used to construct the fixed-depth tree, leading to more log templates being generated. Drain has good performance on HDFS dataset, part of the reason is that Drain removes parameters matching numbers and block ids first, which are the majority of parameters in HDFS log datasets.

5.2.1 Sensitivity Analysis

Fig. 9 shows how sensitive Spell is to its input parameters listed in Table 2, by varying one parameter at a time while fixing the others. We have also tried other combinations and the observations are similar. First, Spell is generally stable to split and merge thresholds, in an evaluated range of $[5, 9]$ and $[10, 14]$ respectively. Second, some measurements could be sensitive to message type threshold τ, for example: 1) accuracy of Blue Gene log, and F-measure of HDFS log, where the difference of the best and worst results is within 20 percent; 2) runtime of Blue Gene log, due to its large size and complex log patterns. Note that, as τ increases, fewer message types may be generated, thus it takes less time for a new log entry to find its match in existing index structures, but more log entries will need to go through the time-consuming dynamic programming matching process (i.e., naive LCS). That's why the runtime decreases a little as τ increases to 0.50, but increases significantly as it grows further.

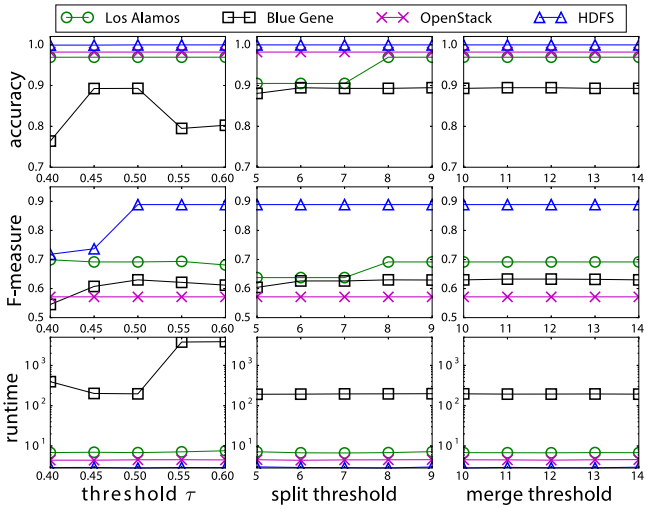


Fig. 9. Sensitivity analysis of Spell.

5.3 Parameter Semantics Inference

We implement the 2-step procedure described in Section 3.7 to learn semantic meanings of parameter positions. Upon the arrival of each log message, Spell first extracts its message type, and then adds its parameter values to the corresponding sets of parameter positions, i.e., each parameter position has a set (group) of parameter values appearing at that position. Finally, Spell applies the 2-step procedure on parsed message types and parameter value groups to generate a semantic meaning for each parameter position.

In step 1, we analyze all message types. For each parameter position in each message type, if the word before or after it is a noun or non-English word, we assign the word as one of its parameter meanings. To find out whether a word is a noun or non-English word, we leverage NLTK corpus [36]. After this step, some parameter positions are labeled with meaningful semantics (i.e., corresponding parameter value groups are "named") and others are not.

Next in step 2, for each of the remaining parameter positions, we check the intersection between its parameter value group and each of the named parameter value groups. If an intersection size is larger than half of the unnamed group size, then Spell assigns the semantic meaning of the named group to that unnamed parameter position.

We apply this approach to all log datasets in Table 3. A parameter position could have multiple semantic meanings since an unnamed group may have large intersections with multiple named groups (e.g., "src", "dest" and "IP"), and we treat a parameter position as correctly parsed if at least one semantic meaning makes sense. The evaluation results are shown in Fig. 10, which include inference results using only step 1, and results using both step 1 and step 2. Y axis denotes the percentage of parameter positions that are assigned with correct semantics. Fig. 10 illustrates that the 2-step procedure we propose could reveal meaningful semantics for most of the parameter positions, where step 2 largely boosts the accuracy. For OpenStack log, there is a "request id" at the beginning of each log message which

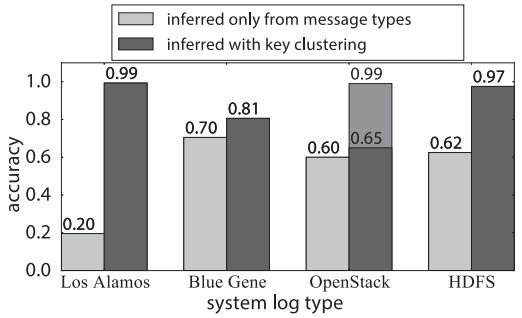


Fig. 10. Percentage of parameter semantics that are correctly inferred.

could be treated as a parameter position. Spell fails to find a semantic meaning for this because there's no meaningful word next to it in each message type, and no other parameter positions contain similar identifiers. Thus, if we treat such cases as un-discovered parameter positions, the parsing accuracy is only 65 percent. However, since such identifiers only appear at the beginning and have the same format "req-***", if we apply this prior knowledge, the parsing accuracy could be as high as 99 percent.

This component serves as a post analysis after all message types are parsed out by Spell main log parser. A python implementation takes on average about 10 seconds for Los Alamos log, and about 1 minute for Blue Gene log, of which 8 seconds are used to load NLTK word repository.

6 Conclusions

We present a streaming structured log parser, Spell, for parsing large system event logs in a streaming fashion. Spell works perfectly for online system log mining and monitoring. We propose pre-filtering to improve Spell's efficiency, and split and merge to improve Spell's effectiveness. Experiments over real system logs have clearly demonstrated that Spell has outperformed the state-of-the-art methods in terms of both efficiency and effectiveness. For future work, we plan to investigate deep learning and natural language processing techniques for advanced log parsing and log understanding.

Acknowledgments

The authors thank the support from NSF SaTC grants 1801446 and 1514520. Feifei Li is also supported in part by NSFC grant 61729202.

References

[1] Q. Fu, J.-G. Lou, Y. Wang, and J. Li, "Execution anomaly detection in distributed systems through unstructured log analysis," in *Proc. IEEE Int. Conf. Data Mining*, 2009, pp. 149–158.

[2] J.-G. Lou, Q. Fu, S. Yang, Y. Xu, and J. Li, "Mining invariants from console logs for system problem detection," in *Proc. USENIX Annu. Tech. Conf.*, 2010, pp. 24–24.

[3] J.-G. Lou, Q. Fu, S. Yang, J. Li, and B. Wu, "Mining program workflow from interleaved traces," in *Proc. ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2010, pp. 613–622.

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

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

[6] K. Nagaraj, C. Killian, and J. Neville, "Structured comparative analysis of systems logs to diagnose performance problems," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2012, pp. 26–26.

[7] Q. Fu, J.-G. Lou, Q. Lin, R. Ding, D. Zhang, and T. Xie, "Contextual analysis of program logs for understanding system behaviors," in *Proc. Work. Conf. Mining Softw. Repositories*, 2013, pp. 397–400.

[8] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy, "Inferring models of concurrent systems from logs of their behavior with csight," in *Proc. Int. Conf. Softw. Eng.*, 2014, pp. 468–479.

[9] A. A. Makanju, A. N. Zincir-Heywood, and E. E. Milios, "Clustering event logs using iterative partitioning," in *Proc. ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2009, pp. 1255–1264.

[10] L. Tang and T. Li, "LogTree: A framework for generating system events from raw textual logs," in *Proc. IEEE Int. Conf. Data Mining*, 2010, pp. 491–500.

[11] L. Tang, T. Li, and C.-S. Perng, "Logsig: Generating system events from raw textual logs," in *Proc. Conf. Inf. Knowl. Manag.*, 2011, pp. 785–794.

[12] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan, "Online system problem detection by mining patterns of console logs," in *Proc. IEEE Int. Conf. Data Mining*, 2009, pp. 588–597.

[13] A. Gainaru, F. Cappello, S. Trausan-Matu, and B. Kramer, "Event log mining tool for large scale HPC systems," in *Proc. 17th Int. Conf. Parallel Process.*, 2011, pp. 52–64.

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

[15] X. Yu, P. Joshi, J. Xu, G. Jin, H. Zhang, and G. Jiang, "Cloudseer: Workflow monitoring of cloud infrastructures via interleaved logs," in *Proc. ACM Int. Conf. Architectural Support Program. Languages Operating Syst.*, 2016, pp. 489–502.

[16] Z. Cao, S. Chen, F. Li, M. Wang, and X. S. Wang, "LogKV: Exploiting key-value stores for event log processing," in *Proc. Conf. Innovative Data Syst. Res.*, 2013.

[17] P. He, J. Zhu, S. He, J. Li, and M. R. Lyu, "An evaluation study on log parsing and its use in log mining," in *Proc. IEEE Int. Conf. Depend. Syst. Netw.*, 2016, pp. 654–661.

[18] H. C. Xia Ning, Geoff Jiang and K. Yoshihira, "HLAer: A system for heterogeneous log analysis," in *Proc. SDM Workshop Heterogeneous Learn.*, 2014.

[19] H. Hamooni, B. Debnath, J. Xu, H. Zhang, G. Jiang, and A. Mueen, "Logmine: Fast pattern recognition for log analytics," in *Proc. Conf. Inf. Knowl. Manag.*, 2016, pp. 1573–1582.

[20] M. Du and F. Li, "Spell: Streaming parsing of system event logs," in *Proc. IEEE Int. Conf. Data Mining*, 2016, pp. 859–864.

[21] D. S. Hirschberg, "A linear space algorithm for computing maximal common subsequences," *Commun. ACM*, vol. 18, pp. 341–343, 1975.

[22] J. W. Hunt and T. G. Szymanski, "A fast algorithm for computing longest common subsequences," *Commun. ACM*, vol. 20, pp. 350–353, 1977.

[23] D. S. Hirschberg, "Algorithms for the longest common subsequence problem," *J. ACM*, vol. 24, no. 4, pp. 664–675, 1977.

[24] L. Bergroth, H. Hakonen, and T. Raita, "A survey of longest common subsequence algorithms," in *Proc. 7th Int. Symp. String Process. Inf. Retrieval*, 2000, pp. 39–48.

[25] Y. Wu, L. Wang, D. Zhu, and X. Wang, "An efficient dynamic programming algorithm for the generalized LCS problem with multiple substring exclusive constraints," *J. Discrete Algorithms*, vol. 26, pp. 98–105, 2014.

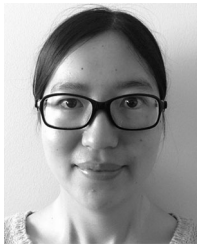
[26] Y. Li, H. Li, T. Duan, S. Wang, Z. Wang, and Y. Cheng, "A real linear and parallel multiple longest common subsequences (MLCS) algorithm," in *Proc. ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2016, pp. 1725–1734.

[27] Y. Li, Y. Wang, Z. Zhang, Y. Wang, D. Ma, and J. Huang, "A novel fast and memory efficient parallel mlcs algorithm for longer and large-scale sequences alignments," in *Proc. IEEE Int. Conf. Data Eng.*, 2016, pp. 1170–1181.

[28] Wikipedia contributors, "Inverted index — Wikipedia, the free encyclopedia," 2017. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Inverted_index&oldid=815397313, Accessed on: Jul. 17, 2018.

[29] Wikipedia contributors, "Trie — Wikipedia, the free encyclopedia," 2018. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Trie&oldid=848098286, Accessed: Jul. 17, 2018.

[30] Wikipedia contributors, "Readerswriter lock — Wikipedia, the free encyclopedia," 2018. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Readers%E2%80%93writer_lock&oldid=844122301, Accessed: Jul. 9, 2018.
[31] D. Nadeau and S. Sekine, "A survey of named entity recognition and classification," *Linguisticae Investigationes*, vol. 30, no. 1, pp. 3–26, Jan. 2007.
[32] M. Du, F. Li, G. Zheng, and V. Srikumar, "DeepLog: Anomaly detection and diagnosis from system logs through deep learning," in *Proc. ACM SIGSAC Conf. Comput. Commun. Security*, 2017, pp. 1285–1298.
[33] A. Makanju, A. N. Zincir-Heywood, and E. E. Milios, "A lightweight algorithm for message type extraction in system application logs," *IEEE Trans. Knowl. Data Eng.*, vol. 24, no. 11, pp. 1921–1936, Nov. 2012.
[34] CloudLab. [Online]. Available: https://www.cloudlab.us/, Accessed: 2017.
[35] B. Schäling, *The Boost C++ Libraries*. 2011.
[36] NLTK, "nltk.corpus package," [Online]. Available: http://www.nltk.org/api/nltk.corpus.html, Accessed: 2017.



Min Du received the bachelor's and master's degrees from Beihang University, and the PhD degree from the School of Computing, University of Utah, in 2018. She is currently a postdoctoral scholar in the EECS Department, UC Berkeley. Her research interests include big data analytics and machine learning security.

Feifei Li received the BS degree in computer engineering from Nanyang Technological University, in 2002, and the PhD degree in computer science from Boston University, in 2007. He is currently an associate professor with the School of Computing, University of Utah. His research interests include database and data management systems and big data analytics.