This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2762673, IEEE Transactions on Dependable and Secure Computing

Towards Automated Log Parsing for Large-Scale Log Data Analysis

Pinjia He, Jieming Zhu, Shilin He, Jian Li, and Michael R. Lyu, *Fellow, IEEE*

Abstract—Logs are widely used in system management for dependability assurance because they are often the only data available that record detailed system runtime behaviors in production. Because the size of logs is constantly increasing, developers (and operators) intend to automate their analysis by applying data mining methods, therefore structured input data (e.g., matrices) are required. This triggers a number of studies on log parsing that aims to transform free-text log messages into structured events. However, due to the lack of open-source implementations of these log parsers and benchmarks for performance comparison, developers are unlikely to be aware of the effectiveness of existing log parsers and their limitations when applying them into practice. They must often reimplement or redesign one, which is time-consuming and redundant. In this paper, we first present a characterization study of the current state of the art log parsers and evaluate their efficacy on five real-world datasets with over ten million log messages. We determine that, although the overall accuracy of these parsers is high, they are not robust across all datasets. When logs grow to a large scale (e.g., 200 million log messages), which is common in practice, these parsers are not efficient enough to handle such data on a single computer. To address the above limitations, we design and implement a parallel log parser (namely POP) on top of Spark, a large-scale data processing platform. Comprehensive experiments have been conducted to evaluate POP on both synthetic and real-world datasets. The evaluation results demonstrate the capability of POP in terms of accuracy, efficiency, and effectiveness on subsequent log mining tasks.

Index Terms—System Management, Log Parsing, Log Analysis, Parallel Computing, Clustering.

1 Introduction

Large-scale distributed systems are becoming the core components of the IT industry, supporting daily use software of various types, including online banking, e-commerce, and instant messaging. In contrast to traditional standalone systems, most of such distributed systems run on a 24×7 basis to serve millions of users globally. Any non-trivial downtime of such systems can lead to significant revenue loss [1], [2], [3], and this thus highlights the need to ensure system dependability.

System logs are widely utilized by developers (and operators) to ensure system dependability, because they are often the only data available that record detailed system runtime information in production environment. In general, logs are unstructured text generated by logging statements (e.g., *printf()*, *Console.Writeline()*) in system source code. Logs contain various forms system runtime information, which enables developers to monitor the runtime behaviors of their systems and to further assure system dependability.

With the prevalence of data mining, the traditional method of log analysis, which largely relies on manual inspection and is labor-intensive and error-prone, has been complemented by automated log analysis techniques. Typical examples of log analysis techniques on dependability assurance include anomaly detection [4], [5], program verification [6], [7], problem diagnosis [8], [9], and security assurance [10], [11]. Most of these log analysis techniques comprise three steps: log parsing, matrix generation, and log mining (Fig. 1). The performance of log parsing plays an important role in various log analysis frameworks in terms of both accuracy and efficiency. The log mining step usually accepts structured data (e.g., a matrix) as input and reports mining results to developers. However, raw log messages are usually unstructured because they are natural language designed by developers. Typically, a raw log message, as illustrated in the following example, records a specific system event with a set of fields: timestamp, verbosity level, and raw message content.

```
2008-11-09 20:46:55,556 INFO dfs.DataNode$PacketResp
onder: Received block blk_3587508140051953248 of siz
e 67108864 from /10.251.42.84
```

Log parsing is usually the first step of automoated log analysis, whereby raw log messages can be transformed into a sequence of structured events. A raw log message, as illustrated in the example, consists of two parts, namely a *constant* part and a *variable* part. The constant part constitutes the fixed plain text and represents the corresponding event type, which remains the same for every event occurrence. The variable part records the runtime information, such as the values of states and parameters (e.g., the block ID: blk_-1608999687919862906), which may vary among different event occurrences. The goal of log parsing is to automatically separate the constant part and variable part of a raw log message (also known as log de-parametrization), and to further match each log message with a specific event type (usually denoted by its constant part). In the example, the event type can be denoted as "*Received block * of size * from **", where the variable part is identified and masked using asterisks.

Traditional log parsing approahes rely heavily on manually customized regular expressions to extract the specific

- *The authors are with the Department of Computer Sciences and Engineering, The Chinese University of Hong Kong, Shatin, NT, Hong Kong. E-mail: {pjhe, jmzhu, slhe, jianli, lyu}@cse.cuhk.edu.hk.*

Manuscript received xx xx, xxxx; revised xx xx, xxxx.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2762673, IEEE Transactions on Dependable and Secure Computing

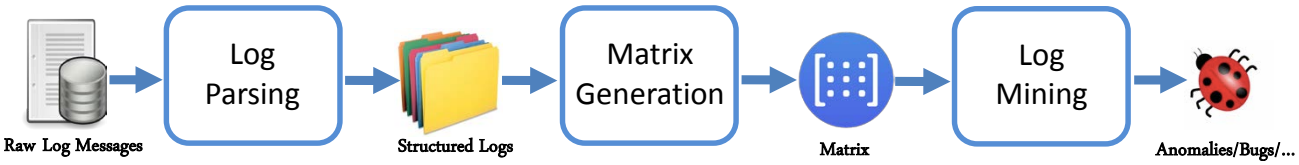

Fig. 1: Overview of Log Analysis

log events (e.g., SEC [12]). However, this method becomes inefficient and error-prone for modern systems for the following reasons. First, the volume of log grows rapidly; for example, it grows at a rate of approximately 50 GB/h (120∼200 million lines) [13]. Manually constructing regular expressions from such a large number of logs is prohibitive. Furthermore, modern systems often integrate open-source software components written by hundreds of developers [4]. Thus, the developers who maintain the systems are usually unaware of the original logging purpose, which increases the difficulty of the manual method. This problem is compounded by the fact that the log printing statements in modern systems update frequently (e.g., hundreds of new logging statements every month [14]); consequently, developers must regularly review the updated log printing statements of various system components for the maintenance of regular expressions.

Recent studies have proposed a number of automated log parsing methods, including SLCT [15], IPLoM [16], LKE [5], LogSig [17]. Despite the importance of log parsing, we find a lack of systematic evaluations on the accuracy and efficiency of the automated log parsing methods available. The effectiveness of the methods on subsequent log mining tasks is also unclear. Additionally, there are no other ready-to-use tool implementations of these log parsers (except for SLCT [15], which was released more than 10 years ago). In this context, practitioners and researchers must implement log parsers by themselves when performing log mining tasks (e.g., [6], [9]), which is a time-consuming and redundant effort.

To fill the significant gap, in this paper, we conduct a comprehensive evaluation of four representative log parsers and then present a parallel log parser that achieves state-of-the-art performance in accuracy and efficiency.

More specifically, we study SLCT [15], IPLoM [16], LKE [5], and LogSig [17], which are widely used log parsers in log analysis. We do not consider source code-based log parsing [4], because, in many cases, the source code is inaccessible (e.g., in third party libraries). By using five real-world log datasets with over 10 million raw log messages, we evaluate the log parsers' performance in terms of accuracy (i.e., F-measure [18], [19]), efficiency (i.e., execution time), and effectiveness on a log mining task (i.e., anomaly detection [4] evaluated by detected anomaly and false alarm). We determine that, although the overall accuracy of these log parsing methods is high, they are not robust across all datasets. When logs grow to a large scale (e.g., 200 million log messages), these parsers fail to complete in reasonable time (e.g., one hour), and most cannot handle such data on a single computer. We also find that parameter tuning costs considerable time for these methods, because parameters tuned on a sample dataset of small size cannot be directly employed on a large dataset.

To address these problems, we propose a parallel log parsing method, called POP, that can accurately and efficiently parse large-scale log data. Similar to previous papers [5], [15], [16], [17], POP assumes the input is single-line logs, which is the common case in practice. To improve accuracy in log parsing, we employ iterative partitioning rules for candidate event generation and hierarchical clustering for event type refinement. To improve efficiency in processing large-scale data, we design POP with linear time complexity in terms of log size, and we further parallelize its computation on top of Spark, a large-scale data processing platform.

We evaluate POP on both real-world datasets and large-scale synthetic datasets with 200 million lines of raw log messages. The evaluation results show the capability of POP in achieving accuracy and efficiency. Specifically, POP can parse all the real-world datasets with the highest accuracy compared with the existing methods. Moreover, POP can parse our synthetic HDFS (Hadoop Distributed File System) dataset in 7 min, whereas SLCT requires 30 min, and IPLoM, LKE, and LogSig fail to terminate in reasonable time. Moreover, parameter tuning is easy in POP because the parameters tuned on small sample datasets can be directly applied to large datasets while preserving high parsing accuracy.

- This is the first work that systematically evaluates the performance of current log parsers in terms of accuracy, efficiency, and effectiveness on subsequent log mining tasks.
- It presents the design and implementation of a parallel log parsing method (POP), which can parse large-scale log data accurately and efficiently.
- The source code of both POP and the studied log parsers have been publicly released [20], allowing for easy use by practitioners and researchers for future study.

Extended from its preliminary conference version [21], the paper makes several major enhancements: the design and implementation of a parallel log parsing method (POP); the evaluation of POP's performance in terms of accuracy, efficiency, and effectiveness on subsequent log mining tasks; efficiency evaluation of the state-of-the-art parsers on large-scale synthetic datasets; discussion highlighting the usage of POP in practice; and code release of POP for reproducible research.

The remainder of this paper is organized as follows. Section 2 presents the overview of log parsing, and Section 3 introduces our parallel log parsing method. The evaluation results are reported in Section 4. We discuss limitations and practical usage of POP in Section 5. We then introduce the

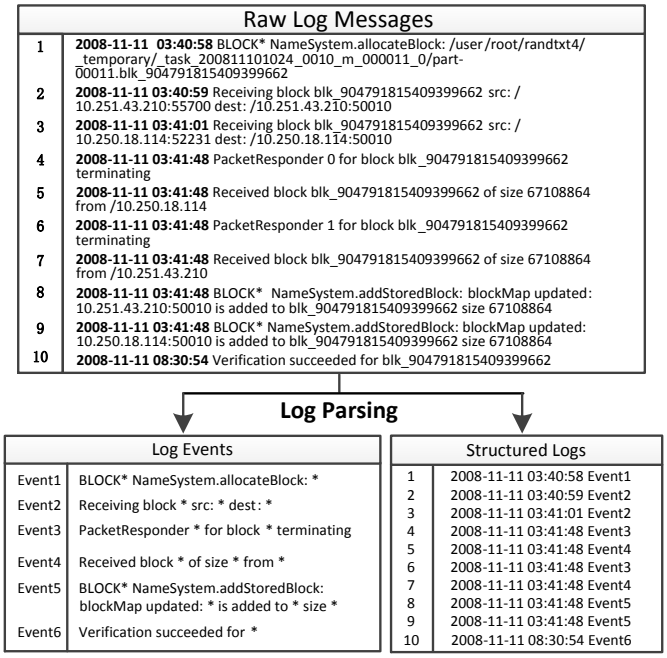

Fig. 2: Overview of Log Parsing

related work in Section 6, and finally conclude this paper in Section 7.

2 Overview of Log Parsing

The goal of log parsing is to transform raw log messages into a sequence of structured events, which facilitates subsequent matrix generation and log mining. Fig. 2 illustrates an overview of log parsing process. As shown in the figure, there are ten HDFS raw log messages collected on the Amazon EC2 platform [4]. In real-world cases, a system can generate millions of such log messages per hour. The output of log parsing is a list of *log events* and *structured logs*.

Log events are the extracted templates of log messages, for example, "*Event 2: Receiving block * src: * dest: **". In practice, typically we employ POP on historical logs to generate log events, which can be used to parse the logs in system runtime. Struture logs are a sequence of events with field of interest (e.g., timestamp). The structure logs can be easily transformed to a matrix, or directly processed by the subsequent log mining tasks (e.g., anomaly detection [4], deployment verfication [7]).

Log parsing has been widely studied in recent years. Among all the log parsers, we choose four representative ones (SLCT [15], IPLoM [22], LKE [5], LogSig [17]), which are in widespread use for log mining tasks. Details of these parsers are provided in our supplementary report [23].

3 Parallel Log Parsing (POP)

From the implementation and systematic study of log parsers introduced in Section 2, we observe that a good log parsing method should fulfill the following requirements: (1) Accuracy. The parsing accuracy (i.e., F-measure) should be high. (2) Efficiency. The running time of a log parser should be as short as possible. (3) Robustness. A log parsing method needs to be consistently accurate and efficient on logs from different systems.

[18:03:38] chrome.exe, 4381 bytes sent, 6044 bytes received, lifetime **09:14**
[16:49:08] chrome.exe, 464 bytes sent, 1101 bytes received, lifetime **<1 sec**

Fig. 3: Proxifier Log Samples

Thus, we design a *p*arallel l*o*g *p*arsing method, namely **POP**, to fulfill the above requirements. POP preprocesses logs with simple domain knowledge (step 1). It then hierarchically partitions the logs into different groups based on two heuristic rules (step 2 and 3). For each group, the constant parts are extracted to construct the log event (step 4). Finally, POP merges similar groups according to the result of hierarchical clustering on log events (step 5). We design POP on top of Spark [24], [25], a large-scale data processing platform using the parallelization power of computer clusters, and all computation-intensive parts of POP are designed to be highly parallelizable.

3.1 Step 1: Preprocess by Domain Knowledge

According to our study on the existing log parsers, simple preprocessing using domain knowledge can improve parsing accuracy, so raw logs are preprocessed in this step. POP provides two preprocessing functions. First, POP prunes variable parts according to simple regular expression rules provided by developers, for example, removing block ID in Fig. 2 by "blk_[0-9]+". For all datasets used in our experiment, at most two rules are defined on a dataset. This function can delete variable parts that can be easily identified with domain knowledge. Second, POP allows developers to manually specify log events based on regular expression rules. This is useful because developers intend to put logs with certain properties into the same partition in some cases. For example, Fig. 3 contains two log messages from Proxifier dataset. The two logs will be put into the same partition by most of the log parsing methods. However, developers may want to count the session with less than 1 second lifetime separately. In this case, POP can easily extract the corresponding logs based on the regular expression ".*<1 sec.*". Note that the simple regular expressions used in this step require much less human effort than those complex ones used by traditional methods to match the whole log messages.

3.2 Step 2: Partition by Log Message Length

In this step, POP partitions the remaining logs into nonoverlapping groups of logs. POP puts logs with the same log message length into the same group. By log message length, we mean the number of tokens in a log message. This heuristic, which is also used by IPLoM [22], is based on the assumption that logs with the same log event will likely have the same log message length. For example, log event "Verification succeeded for *" from HDFS dataset contains 4 tokens. It is intuitive that logs having this log event share the same log message length, such as "Verification succeeded for blk_1" and "Verification succeeded for blk_2". This heuristic rule is considered coarse-grained, so it is possible that log messages in the same group have different log events. "Serve block * to *" and "Deleting block * file *" will be put into the same group in step 2, because they

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2762673, IEEE Transactions on Dependable and Secure Computing

both contain 5 tokens. This issue is addressed by a fine-grained heuristic partition rule described in step 3. Besides, it is possible that one or more variable parts in the log event contain variable length, which invalidates the assumption of step 2. This will be addressed by hierarchical clustering in step 5.

3.3 Step 3: Recursively Partition by Token Position

In step 3, each group is recursively partitioned into subgroups, where each subgroup contains logs with the same log event (i.e., same constant parts). This step assumes that if the logs in a group having the same log event, the tokens in some token positions should be the same. For example, if all the logs in a group have log event "Open file *", then the tokens in the first token position of all logs should be "Open". We define *complete token position* to guide the partitioning process.

Notations: Given a group containing logs with log message length n, there are n token positions. All tokens in token position i form a token set TS_i, which is a collection of distinct tokens. The cardinality of TS_i is defined as $|TS_i|$. A token position is *complete* if and only if $|TS_i| = 1$, and it is defined as a *complete token position*. Otherwise, it is defined as an *incomplete token position*.

Our heuristic rule is to recursively partition each group until all the resulting groups have enough complete token positions. To evaluate whether complete token positions are enough, we define *Group Goodness (GG)* as following.

$$GG = \frac{\#CompleteTokenPositions}{n}. \quad (1)$$

A group is a *complete group* if $GG > GS$, where GS stands for *Group Support*, a threshold assigned by developers. Otherwise, the group is an incomplete group. In this step, POP recursively partitions the groups if the current group is not a complete group.

Algorithm 1 provides the pseudo code of step 3. POP regards all groups from step 2 as incomplete groups (line 1). Incomplete groups are recursively partitioned by POP to generate a list of complete groups (lines 4∼24). For each incomplete group, if it already contains enough complete token positions, it is moved to the complete group list (lines 6∼8). Otherwise, POP finds the split token position, which is the token position with the lowest cardinality among all incomplete token positions. Because of its lowest cardinality, tokens in the split token position are most likely to be constants. Then POP calculates *Absolute Threshold (AT)* and *Relative Threshold (RT)* (line 11∼12). A token position with smaller *AT* and *RT* is more likely to contain constants. For example, in Fig. 4, column (i.e. token position) 1 and 2 have smaller *AT* (2) and *RT* (0.5), so they are more likely to contain constants compared with column 3, whose *AT* is 4 and *RT* is 1. Note that we only need to calculate *AT* and *RT* for the split token position. We demonstrate *AT* and *RT* for all the columns in Fig. 4 for better explanation. Thus, POP regards the tokens as variables only when both *AT* and *RT* are larger than manually defined thresholds (i.e., $splitAbs$ and $splitRel$ respectively). If all tokens in the split token position are variables, POP moves the current group to the complete group list, because it could not be further partitioned (line 13∼15). Otherwise, POP partitions the current group into $|TS_s|$ resulting groups based on the token value in the split token position (line 17). Among all the result groups, the complete groups are added into the complete group list, while the incomplete ones are added to the incomplete group list for further partitioning (line 18∼22). Finally, the complete group list is returned, where logs in each group share the same log event type.

Algorithm 1 POP Step 3: Recursively partition each group to complete groups.

Input: a list of log groups from step 2: $logGroupL$; and algorithm parameters: $GS, splitRel, splitAbs$
Output: a list of complete groups: $completeL$

```
1: incompleteL ← logGroupL
2: completeL ← List( )                          ▷ Initialize with empty list
3: curGroup ← first group in incompleteL
4: repeat
5:     Set n ← |curGroup|
6:     if IsComplete(curGroup, GS)= true then
7:         Add curGroup to completeL
8:         Remove curGroup from incompleteL
9:     else
10:        Find the split token position s
11:        Compute AT ← |TS_s|
12:        Compute RT ← |TS_s|/n
13:        if AT > splitAbs and RT > splitRel then
14:            Add curGroup to completeL
15:            Remove curGroup from incompleteL
16:        else
17:            Partition curGroup to several resultGroup based
               on the token value in split token position
18:            for all resultGroup do
19:                if IsComplete(resultGroup, GS)= true then
20:                    Add resultGroup to completeL
21:                else
22:                    Add resultGroup to incompleteL
23:    curGroup ← next group in incompleteL
24: until incompleteL is empty

25: function IsComplete(group, gs)
26:    Compute token sets for token positions in group
27:    Compute GG                                  ▷ by Equation 1
28:    if GG > gs then
29:        return true
30:    else
31:        return false
```

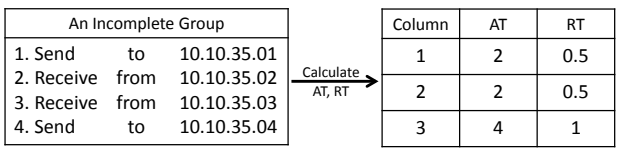

Fig. 4: An Example of *AT*, *RT* Calculation

3.4 Step 4: Generate Log Events

At this point, the logs have been partitioned into nonoverlapping groups by two heuristic rules. In this step, POP scans all the logs in each group and generates the corresponding log event, which is a line of text containing constant parts and variable parts. The constant parts are represented by tokens and the variable parts are represented by wildcards. To decide whether a token is a constant or

a variable, POP counts the number of distinct tokens (i.e., $|TS|$) in the corresponding token position. If the number of distinct tokens in a token position is one, the token is constant and will be outputted to the corresponding token position in a log event. Otherwise, a wildcard is outputted.

3.5 Step 5: Merge Groups by Log Event

To this end, logs have been partitioned into nonoverlapping complete groups, and each log message is matched with a log event. Most of the groups contain logs that share the same log event. However, some groups may be over-parsed because of suboptimal parameter setting, which causes false negatives. Besides, it is possible that some variable parts in a log event have variable length, which invalidates the assumption in step 2. This also brings false negatives.

To address over-parsing and further improve parsing accuracy, in this step, POP employs hierarchical clustering [26] to cluster similar groups based on their log events. The groups in the same cluster will be merged, and a new log event will be generated by calculating the Longest Common Subsequence (LCS) [27] of the original log events. This step is based on the assumption that if logs from different groups have the same log event type, the generated log event texts from these groups should be similar. POP calculates Manhattan distance [28] between two log event text to evaluate their similarity. Specifically,

$$d(a,b) = \sum_{i=1}^{N} |a_i - b_i|, \tag{2}$$

where a and b are two log events, N is the number of all constant token values in a and b, and a_i means the occurrence number of the i-th constant token in a. We use Manhattan distance because it assigns equal weight to each dimension (i.e., constant). This aligns with our observation that all constants are of equal importance in log parsing. Besides, Manhattan distance is intuitive, which makes parameter tuning easier. POP employs complete linkage [29] to evaluate the distance between two clusters, because the resulted clusters will be compact, which avoids clustering dissimilar groups together. The only parameter in this step is $maxDistance$, which is the maximum distance allowed when the clustering algorithm attempts to combine two clusters. The algorithm stops when the minimum distance among all cluster pairs is larger than $maxDistance$.

3.6 Implementation

To make POP efficient in large-scale log analysis, we build it on top of Spark [24], [25], a large-scale data processing platform. Specifically, Spark runs iterative analysis programs with orders of magnitude faster than Hadoop Mapreduce [30]. The core abstraction in Spark is Resilient Distributed Datasets ($RDDs$), which are fault-tolerant and parallel data structures representing datasets. Users can manipulate RDDs with a rich set of Spark operations called $transformations$ (e.g., map, filter) and $actions$ (e.g, reduce, aggregate). Calling transformations on an RDD generates a new RDD, while calling actions on an RDD reports calculation result to users. Spark employs lazy evaluation, so that transformations on RDDs will not be executed until an action is called. At that time, all preceding transformations are executed to generate the RDD, where the action is then evaluated. We build POP on top of Spark because it is good at parallelizing identical computation logic on each element of a dataset, and it directly uses the output of one step in memory as the input to another. In our case, an RDD can represent a log dataset, where each element is a log message. POP can be parallelized by transformations and actions, because each POP step requires computation-intensive tasks that cast identical computation logic to every log message. To parallelize these tasks, we invoke Spark operations with specially designed functions describing the computation logic. In the following, we will introduce the Spark operations we applied for the five POP steps.

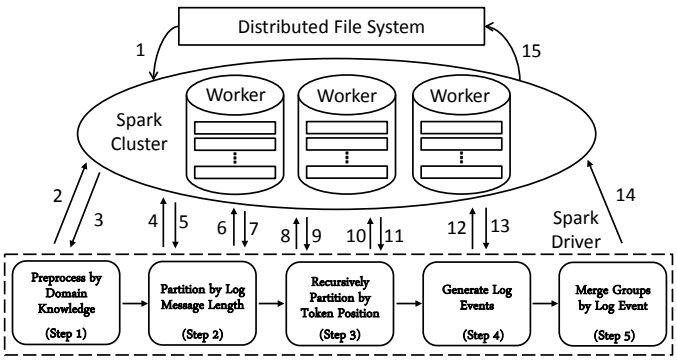

Fig. 5: Overview of POP Implementation

The implementation of POP on Spark is illustrated in Fig. 5. The five rounded rectangles at the bottom represent the five steps of POP, where the numbered arrows represent the interactions between the main program and the $Spark\ cluster$. The main program is running in $Spark\ driver$, which is responsible for allocating Spark tasks to $workers$ in the Spark cluster. For a POP Spark application, in step 1, we use $textFile$ to load the log dataset from a distributed file system (e.g., HDFS) to Spark cluster as an RDD (arrow 1). Then, we use map to preprocess all log messages with a function as input describing the preprocessing logic on single log message (arrow 2). After preprocessing, we $cache$ the preprocessed log messages in memory and return an RDD as the reference (arrow 3). In step 2, we use $aggregate$ to calculate all possible log message length values (arrow 4) and return them as a list (arrow 5). Then for each value in the list, we use $filter$ to extract log messages with the same log message length (arrow 6), which is returned as an RDD (arrow 7). Now we have a list of RDDs. In step 3, for each RDD, we employ $aggregate$ to form the token sets for all token positions (arrow 8∼9) as described in Section 3.3. Based on the token sets and pre-defined thresholds, the driver program decides whether current RDD could be further partitioned or not. If yes, we use $filter$ to generate new RDDs and add them into the RDD list (arrow 10∼11). Otherwise, we remove it from the list and pass the RDD to step 4. In step 4, we use $reduce$ to generate log events for all RDDs (arrow 12∼13). When all log events have been extracted, POP runs hierarchical clustering on them in main program. We use $union$ to merge RDDs based on the clustering result (arrow 14). Finally, merged RDDs are outputted to the distributed file system by $saveAsTextFile$ (arrow 15).

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2762673, IEEE Transactions on Dependable and Secure Computing

The implementation of this specialized POP is non-trivial. First, Spark provides more than 80 operations and this number is increasing quickly due to its active community. We need to select the most suitable operations to avoid unnecessary performance degradation. For example, if we use $aggregateByKey$ in step 2 and step 3 instead of $aggregate$, the running time will be one order of magnitude longer. Second, we need to design tailored functions as input for Spark operations, such as $aggregate$ and $reduce$. Though we use $aggregate$ in both step 2 and step 3, different functions have been designed. The source code of POP has been release [20] for reuse. Note that the existing log parser can also be parallelized, but they require non-trivial efforts.

4 Evaluation

This section presents our evaluation methodology first. Then we evaluate the performance of log parsers in terms of their accuracy, efficiency, and effectiveness on subsequent log mining tasks in different sub-sections. For each of these three evaluation items, we first explain the evaluation study on representative log parsers and their implication; then we analyze the evaluation results of POP.

4.1 Study Methodology

Log Datasets. We used five real-world log datasets, including supercomputer logs (BGL and HPC), distributed system logs (HDFS and Zookeeper), and standalone software logs (Proxifier). Table 1 provides the basic information of these datasets. *Log Size* column describes the number of raw log messages, while *#Events* column is the number of log event types. Since companies are often reluctant to release their system logs due to confidentiality, logs are scarce data for research. Among the five real-world log datasets in Table 1, three log datasets are obtained from their authors. Specifically, BGL is an open dataset of logs collected from a BlueGene/L supercomputer system at Lawrence Livermore National Labs (LLNL), with 131,072 processors and 32,768GB memory [31]. HPC is also an open dataset with logs collected from a high performance cluster at Los Alamos National Laboratory, which has 49 nodes with 6,152 cores and 128GB memory per node [32]. HDFS logs are collected in [4] by engaging a 203-node cluster on Amazon EC2 platform. To enrich the log data for evaluation purpose, we further collected two datasets: one from a desktop software Proxifier, and the other from a Zookeeper installation on a 32-node cluster in our lab. In particular, the HDFS logs from [4] have well-established anomaly labels, each of which indicates whether a block has anomaly operations. Specifically, the dataset with over 10 million log messages records operations on 575,061 blocks, among which 16,838 are anomalies.

Log Parser Implementation. Among the four studied log parsing methods, we only find an open-source implementation on SLCT in C language. To enable our evaluations, we have implemented the other three log parsing methods in Python and also wrapped up SLCT as a Python package. Currently, all our implementations have been open-source as a toolkit on Github [20].

Evaluation Metric. We use F-measure [18], [19], a commonly-used evaluation metric for clustering algorithms,

TABLE 1: Summary of Our System Log Datasets

System	Log Size	Length	#Events	Description
BGL	4,685,142	10∼102	376	BlueGene/L Supercomputer
HPC	433,490	6∼104	105	High Performance Cluster (Los Alamos)
HDFS	11,175,629	8∼29	29	Hadoop Distributed File System
Zookeeper	74,380	8∼27	80	Distributed System Coordinator
Proxifier	10,108	10∼27	8	Proxy Client

to evaluate the parsing accuracy of log parsing methods. The definition of parsing accuracy is as the following.

$$Parsing\ Accuracy = \frac{2 * Precision * Recall}{Precision + Recall}, \quad (3)$$

where $Precision$ and $Recall$ are defined as follows:

$$Precision = \frac{TP}{TP + FP}, Recall = \frac{TP}{TP + FN}, \quad (4)$$

where a true positive (TP) decision assigns two log messages with the same log event to the same log group; a false positive (FP) decision assigns two log messages with different log events to the same log group; and a false negative (FN) decision assigns two log messages with the same log event to different log groups. If the logs are under-partitioning, the precision will be low because it leads to more false positives. If a log parsing method over-partitions the logs, its recall will decrease because it has more false negatives. Thus, we use F-measure, which is the harmonic mean of precision and recall, to represent parsing accuracy. To obtain the ground truth for the parsing accuracy evaluation, we split the raw log messages into different groups with the help of manually-customized regular expressions.

Experimental Setting. The experiments of systematic evaluation on existing log parsers are run on a Linux server with Intel Xeon E5-2670v2 CPU and 128GB DDR3 1600 RAM, running 64-bit Ubuntu 14.04.2 with Linux kernel 3.16.0. Experiments of POP are run on Spark 1.6.0 with YARN as the cluster controller on 32 physical machines. The cluster has 4TB memory and 668 executors in total. All 32 physical machines are inter-connected with 10Gbps network switch. In our experiment, unless otherwise specified, we use 16 executors, each of which has 25G memory and 5 executor cores. We set Kryo as the Spark serializer because it is significantly faster and more compact than the default one [33]. The parameter setting follows the experience of Cloudera [34], a leading software company that provides big data software, services and supports. To avoid bias, each experiment is run 10 times and the averaged result is reported.

4.2 Accuracy of Log Parsing Methods

In this section, we first evaluate the accuracy of all five log parsing methods. Then we study whether these log parsers can consistently obtain high accuracy on large datasets if parameters tuned on small datasets are employed.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2762673, IEEE Transactions on Dependable and Secure Computing

4.2.1 Parsing Accuracy

In this section, we first evaluate the parsing accuracy of existing parsers with and without preprocessing. Then the parsing accuracy of POP is explained.

To study the accuracy of log parsing methods, we use them to parse real-world logs. Similar to the existing work [17], we randomly sample 2k log messages from each dataset in our evaluation, LKE and LogSig cannot parse large log datasets in reasonable time (e.g., LogSig requires 1 day to parse the entire BGL data). For each experiment, we use the same 2k log messages for all 10 executions. These 2k datasets have been released on our project website [20]. The results are reported in Table 2 (i.e., the first number in each cell). As shown in the table, the accuracy of existing log parsers is larger than 0.8 in many cases. Besides, the overall accuracy on HDFS, Zookeeper and Proxifier datasets is higher than that on BGL and HPC. We find that this is mainly because BGL and HPC logs involve much more event types, and they have more various log length range compared with HDFS, Zookeeper and Proxifier.

LKE has an accuracy drop on HPC dataset. This is because almost all the log messages are grouped into a single cluster in the first step of LKE, which aggressively groups two clusters if any two log messages between them have a distance smaller than a specified threshold. BGL contains a lot of log messages with "*generating core.**" as log event, such as "generating core.1" and "generating core.2". LogSig tends to separate these two log messages into different clusters, because 50% of them are different (core.1 and core.2). This causes LogSig's low accuracy on BGL. Particularly, IPLoM employs heuristic rules based on the characteristics of log messages, while other log pasers rely on typical data mining models. However, we find that IPLoM enjoys the superior overall accuracy, which implies the importance of studying the unique characteristics of log data.

Instead of directly parsing the raw log messages, developers may conduct preprocessing basd on their domain knowledge. To figure out the effectiveness of preprocessing on log parsing, we study the impact of preprocessing on parsing accuracy. Specifically, obvious numerical parameters in log messages (i.e., IP addresses in HPC&Zookeeper&HDFS, core IDs in BGL, and block IDs in HDFS) are removed. Preprocessing is mentioned in LKE and LogSig, but its effectiveness has not been studied.

In Table 2, the second number in each cell represents the accuracy of log parsing methods on preprocessed log data. In most cases, accuracy of parsing is improved. Preprocessing greatly increases the accuracy SLCT on BGL, LKE on HDFS, and LogSig on BGL (in bold). However, preprocessing could not improve the accuracy of IPLoM. It even slightly reduces IPLoM's accuracy on Zookeeper. This is mainly because IPLoM considers preprocessing internally in its four-step process. Unnecessary preprocessing can lead to over-partitioning.

After preprocessing, most of the methods have high overall parsing accuracy (larger than 0.90 in many cases). But none of them consistently lead to very accurate parsing results on all datasets. Specifically, SLCT obtains 0.86 accuracy on HPC; IPLoM has 0.64 accuracy on HPC; LKE encounters 0.17 accuracy on HPC, 0.70 accuracy on BGL, and 0.81 accuracy on Proxifier; LogSig obtains 0.87 accuracy on HPC and 0.84 accuracy on Proxifier.

TABLE 2: Parsing Accuracy of Log Parsing Methods (Raw/Preprocessed)

	BGL	HPC	HDFS	Zookeeper	Proxifier
SLCT	**0.61/0.94**	0.81/0.86	0.86/0.93	0.92/0.92	0.89/-
IPLoM	0.99/0.99	0.64/0.64	0.99/1.00	0.94/0.90	0.90/-
LKE	0.67/0.70	0.17/0.17	**0.57/0.96**	0.78/0.82	0.81/-
LogSig	**0.26/0.98**	0.77/0.87	0.91/0.93	0.96/0.99	0.84/-
POP	**0.99**	**0.95**	**1.00**	**0.99**	**1.00**

Findings. Simple preprocessing using domain knowledge (e.g., removal of IP address) improves log parsing accuracy. With preprocessing, existing log parsers can achieve high overall accuracy. But none of them consistently generates accurate parsing results on all datasets.

To evaluate the accuracy of POP, we employ it to parse the same 2k datasets. For dataset BGL, HPC, HDFS and Zookeeper, we set GS to 0.6, $splitAbs$ to 10, $splitRel$ to 0.1, $maxDistance$ to 0. Parameter tuning is intuitive because all these parameters have physical meanings. Developer can easily find the suitable parameter setting with basic experience on datasets. For dataset Proxifier, we set GS to 0.3, $splitAbs$ to 5, $splitRel$ to 0.1, $maxDistance$ to 10. The parameter setting of Proxifier is different because it contains much fewer log events (i.e., 8 as described in Table 1) compared with others. Besides, we extract log messages containing text "<1 sec" in step 1 of POP, which simulates the practical condition described in Section 3.1.

The results are presented in the last line of Table 2. We observe that POP delivers the best parsing accuracy for all these datasets. For datasets that has relatively few log events (e.g., HDFS and Proxifier), its parsing accuracy is 1.00, which means all the logs can be parsed correctly. For datasets that has relatively more log events, POP still delivers very high parsing accuracy (0.95 for HPC). POP has the best parsing accuracy because of three reasons. First, POP will recursively partition each log group into several groups until they become complete groups. Compared with other log parsers based on heuristic rules (e.g., SLCT), POP provides more fine-grained partitioning. Second, POP merges similar log groups based on the extracted log event, which amends over-partitioning. Third, POP allows developers to manually extract logs with certain properties, which reduces noise for the partitioning process.

4.2.2 Parameter Tuning

The accuracy of log parsers is affected by parameters. For large-scale log data, it is difficult to select the most suitable parameters by trying different values, because each run will cause a lot of time. Typically, developers will tune the parameters on a small sample dataset and directly apply them on large-scale data.

To evaluate the feasibility of this approach, we sampled 25 datasets from the original real-world datasets. Table 3 shows the number of raw log messages in these 25 sample datasets, where each row presents 5 sample datasets generated from a real-world dataset.

TABLE 3: Log Size of Sample Datasets

BGL	400	4k	40k	400k	4m
HPC	600	3k	15k	75k	375k
HDFS	1k	10k	100k	1m	10m
Zookeeper	4k	8k	16k	32k	64k
Proxifier	600	1200	2400	4800	9600

TABLE 4: Parsing Accuracy of POP on Sample Datasets in Table 3 with parameters tuned on 2k datasets

BGL	0.98	0.99	0.99	0.89	0.89
HPC	0.95	0.97	0.96	0.96	0.97
HDFS	1.00	0.99	0.99	0.99	0.99
Zookeeper	0.99	0.99	0.99	0.99	0.99
Proxifier	1.00	1.00	1.00	0.99	0.99

We apply parameters tuned on 2k datasets. In Fig. 6, we evaluate the accuracy of the log parsers on the datasets presented in Table 3 employing these parameters. The results show that IPLoM performs consistently in most cases except a 0.15 drop on Proxifier. SLCT varies a lot on HPC and Proxifier. The accuracy of LKE is volatile in Zookeeper because of its aggressive clustering strategy. LogSig obtains consistent accuracy on datasets with limited types of events, but its accuracy fluctuates severely on datasets with many log events (i.e., BGL and HPC).

Findings. Parameter tuning is time-consuming for existing log parsing methods except IPLoM, because they could not directly use parameters tuned on small sampled data for large datasets.

The experimental results of POP are shown in Table 4 and Fig. 6. We observe that the accuracy of POP is very consistent for all datasets. The accuracy on Zookeeper is 0.99 for all 5 sampling levels, which indicates the parameters tuned on 2k sample dataset lead to nearly the same parsing results. For HPC, HDFS and Proxifier, the fluctuation of the accuracy is at most 0.02, while the accuracy is at least 0.95. For BGL, the accuracy has a 0.1 drop for the last two sampling levels. But POP can still obtain 0.89 accuracy in these two levels, while 0.1 is not a large drop compared with existing parsers in Fig. 6. Compared with existing methods, POP is the only parser that obtains high accuracy consistently on all datasets using the parameters tuned on small sampled data.

4.3 Efficiency of Log Parsing Methods

In this section, we evaluate the efficiency of all five log parsing methods. Specifically, we first measure the running time of these log parsers on 25 sampled datasets with varying number of log messages (i.e., log size) in Table 3. Second, we evaluate the running time of these log parsers on synthetic datasets containing over 200 million log messages, which is comparable to large-scale modern production systems [13].

Note that running time in this paper means the time used to run log parsers (i.e., training time). In addition to training time, we measure the efficiency for parsing a new log message, which is 173μs for BGL, 108μs for HPC, 36μs for HDFS, 29μs for Zookeeper, and 20μs for Proxifier. The matching process relies on regular expressions, thus its time depends on the number of log events and their lengths. The matching time is similar for different log parsers.

4.3.1 Running Time on Real-World Datasets

In Fig. 7, we evaluate the running time of the log parsing methods on all datasets by varying the number of raw log messages (i.e., log size). Notice that as the number of raw log messages increases, the number of events becomes larger as well (e.g., 60 events in BGL400 while 206 events in BGL40k). Fig. 7 is in logarithmic scale, so we can observe the time complexity of these log parsers from the slope of the lines. As show in the figure, the running time of SLCT and IPLoM scale linearly with the number of log messages. They both could parse 10 million HDFS log messages within five minutes. However, as the slopes show, their running time increases fast as the log size becomes larger, because they are limited by the computing power of a single computer. The fast increasing speed can lead to inefficient parsing on production level log data (e.g., 200 million log messages). The running time of LogSig also scales linearly with the number of log messages. However, it requires much running time (e.g, 2+ hours for 10m HDFS log messages), because its clustering iterations are computation-intensive and its word pair generation step is time-consuming. The time complexity of LKE is $O(n^2)$, where n is the number of raw log messages, which makes it unable to handle some real-world log data, such as BGL4m and HDFS10m. Running time of some LKE experiments is not plotted because LKE could not terminate in reasonable time (i.e., days or even weeks).

Findings. Clustering-based log parsers require much running time on real-world datasets. Heuristic rule-based log parsers are more efficient, but their running time increases fast as the log size becomes larger. These imply the demand for parallelization.

The time complexity of POP is $O(n)$, where n is the number of raw log messages. In step 1, step 2 and step 4, POP traverses all log messages once so the time complexity for these steps are all $O(n)$. In step 3, POP may scan some log messages more than once due to recursion. However, in the case of log parsing, the recursion depth can be regarded as a constant because it will not increase as the number of log messages, which remains small in all our datasets. Thus, the time complexity of step 3 is also $O(n)$. Finally, the time complexity of step 5 is $O(m^2 \log m)$, where m is the number of log events. We do not consider it in the time complexity of POP, because m is far less than n. So the time complexity of POP is $O(n + n + n + n + m^2 \log m) = O(n)$.

The "SinglePOP" lines represent the running time of the nonparallel implementation of POP on different datasets. We can observe that the running time of SinglePOP is even shorter than the parallel implementation of POP. Because the nonparallel implementation of POP does not require any data transportation between nodes, which is required by parallel programs. Besides, the parallel implementation needs to deploy the runtime environment (e.g., set up the nodes that will be used) at the beginning, though automatically, will cost some constant time.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2762673, IEEE Transactions on Dependable and Secure Computing

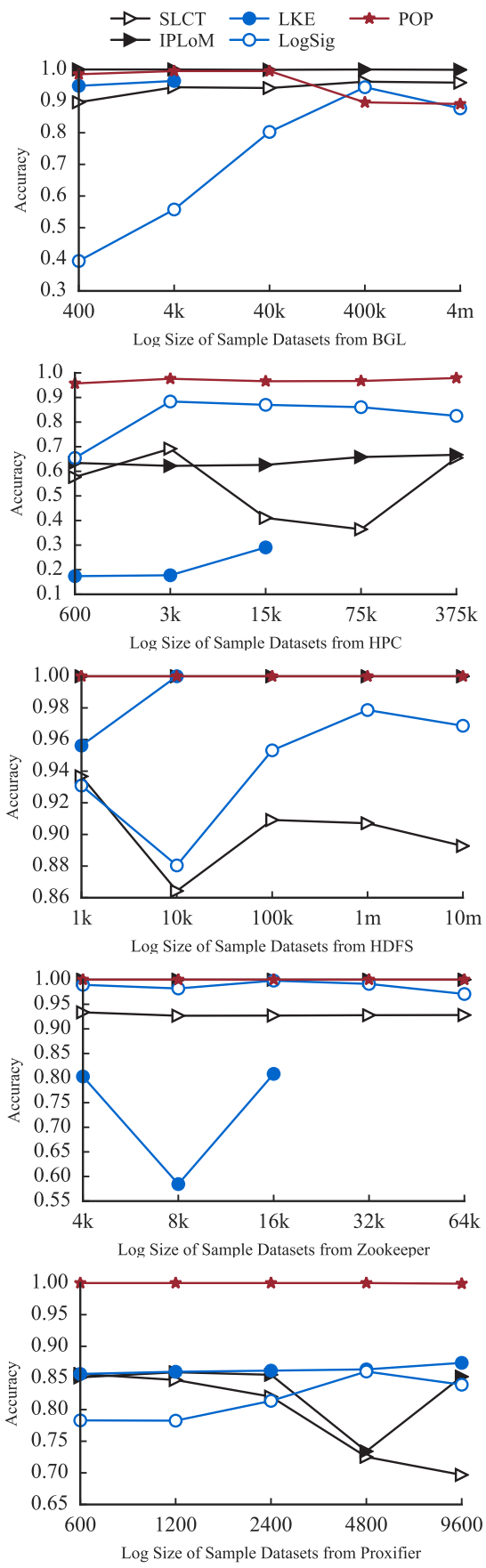

Fig. 6: Parsing Accuracy on Datasets in Different Size

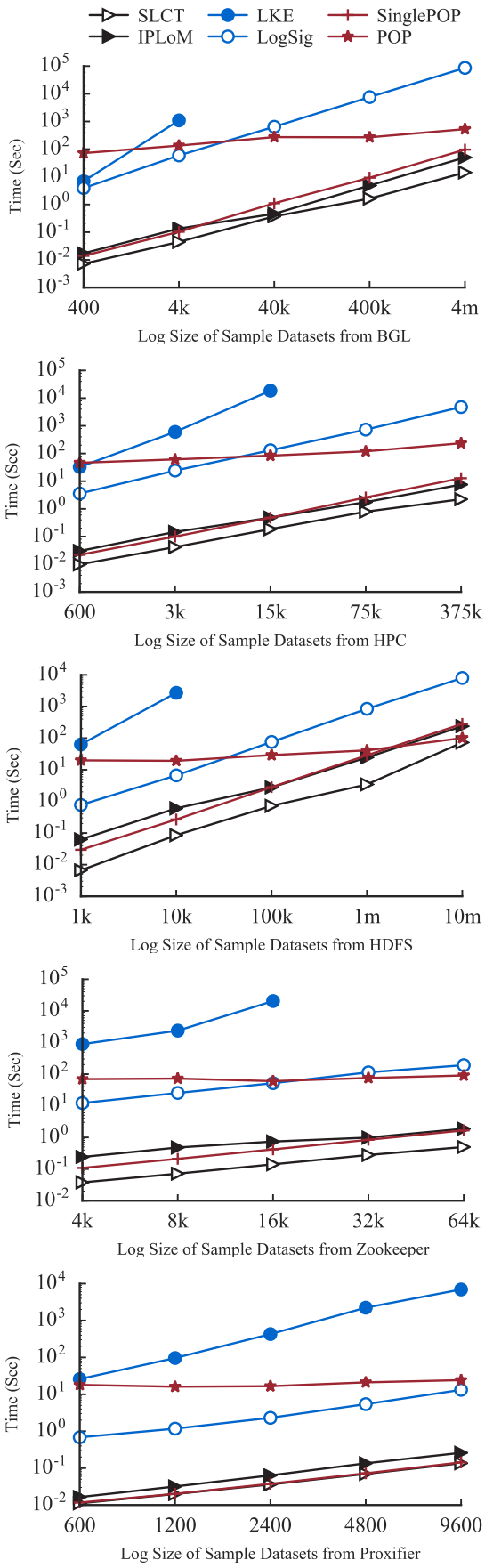

Fig. 7: Running Time of Log Parsing Methods on Datasets in Different Size

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2762673, IEEE Transactions on Dependable and Secure Computing

TABLE 5: Running Time of POP (Sec) on Sample Datasets in Table 3

BGL	71.87	134.48	271.98	268.12	527.63
HPC	46.24	61.29	83.81	119.81	234.92
HDFS	19.82	19.17	29.14	41.03	100.58
Zookeeper	69.62	72.22	60.07	75.56	90.69
Proxifier	18.00	16.08	16.60	21.07	24.22

The experimental results of POP are presented in Table 5 and Fig. 7. Fig. 7 shows that POP has the slowest increasing speed of running time as the log size becomes larger. Its increasing speed is even much better (i.e., slower) than linear parsers (i.e., SLCT, IPLoM, LogSig). For a few cases, the running time of POP even decreases when the log size becomes larger. This is mainly caused by two reasons. First, a larger dataset could benefit more from parallelization than a smaller one. Second, it is possible that a smaller dataset requires deeper recursion in step 3 of POP, which increases its running time. Compared with the existing methods, POP enjoys the slowest running time increase because of its $O(n)$ time complexity and its parallelization mechanism. It can parse a large amount of log messages very fast (e.g., 100 seconds for 10 million HDFS log messages). Although its running time is slower than IPLoM and SLCT in some cases, POP turns out to be more efficient for two reasons. First, as we can observe from Fig. 7, the running time increase of POP is the slowest, so POP will be faster than other log parsers when log size is larger. For example, POP is faster than IPLoM on 10m HDFS dataset. Second, the efficiency of IPLoM and SLCT is limited by computing power or/and memory of single computer, while POP is able to utilize multiple computers.

4.3.2 Running Time on Large-Scale Synthetic Datasets

In this section, we evaluate the running time of log parsers on very large synthetic datasets, which are randomly generated from BGL and HDFS. These two datasets are representative because they include log datasets with a lot and a few log events respectively. BGL has more than 300 log events, while HDFS has 29. The synthetic datasets are generated from the real-world datasets. For example, to generate a 200m synthetic dataset from HDFS dataset, we randomly select a log message from the dataset each time, and repeat this random selection process 200 million times. Fig. 8 presents the experimental results in linear scale.

In this figure, a result is neglected if its running time is larger than one hour, because we want to evaluate the effectiveness of these log parsers in production environment (e.g., 120∼200 million log messages per hour [13]). Thus, experimental results of SLCT, IPLoM and POP are plotted, while LKE and LogSig require more than one hour on these datasets. The running time increase of IPLoM is the fastest among the plotted three. It requires more than an hour for two datasets generated from HDFS; therefore, they are not plotted. Besides, IPLoM requires more than 16G memory when the synthetic dataset contains 30m or more log messages for both BGL and HDFS. Because IPLoM needs to load the whole dataset into memory, and it creates extra data of comparable size in runtime. SLCT is more efficient than IPLoM, and it requires the least time on BGL datasets.

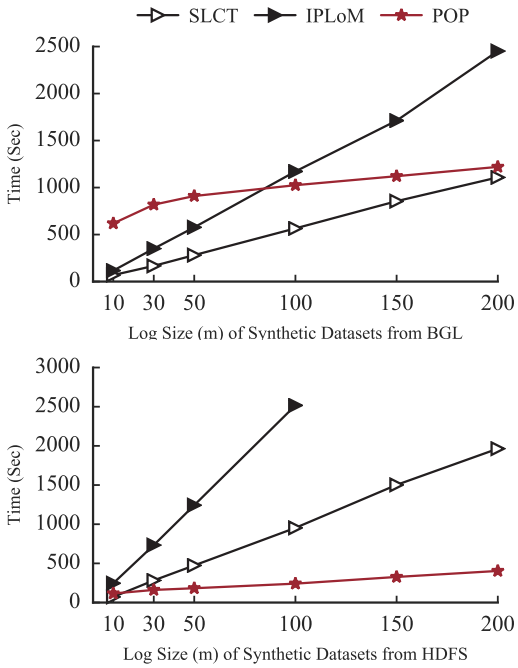

Fig. 8: Running Time on Synthetic Datasets

SLCT only requires two passes across all log data, and it is implemented in C instead of Python. However, its running time increases fast as the log size becomes larger, because SLCT is limited by the computing power of single computer.

Findings. Clustering-based log parsers cannot handle large-scale log data. Heuristic rule-based log parsers are efficient, but they are limited by the computing power or/and memory of a single computer.

For POP, we use 64 executors on BGL datasets and 16 executors on HDFS datasets, each of which has 16G memory and 5 executor cores. We use more executors on BGL datasets because they require more recursive partitioning in step 3. We set 16G memory because this is a typical memory setting for a single computer. We observe that POP has the slowest growth speed among all three methods. Besides, POP requires the least running time for HDFS datasets. Though SLCT requires less time for BGL datasets, its running time increases faster than POP, which is shown by their comparable results on 200m log message dataset generated from BGL. Thus, POP is the most suitable log parser for large-scale log analysis, given that the size of logs will become even larger in the future.

4.4 Effectiveness of Log Parsing Methods on Log Mining: A Case Study

Log mining tasks usually accept structured data (e.g., matrix) as input and report mining results to developers, as described in Fig. 1. If a log parser is inaccurate, the generated structured logs will contain errors, which can further ruin the input matrix of subsequent log mining tasks. A log mining task with erroneous input tends to report biased results. Thus, log parsing should be accurate

TABLE 6: Anomaly Detection with Different Log Parsing Methods (16,838 Anomalies)

	Parsing Accuracy	Reported Anomaly	Detected Anomaly	False Alarm
SLCT	0.83	18,450	10,935 (64%)	7,515 (40%)
LogSig	0.87	11,091	10,678 (63%)	413 (3.7%)
IPLoM	0.99	10,998	10,720 (63%)	278 (2.5%)
POP	0.99	10,998	10,720 (63%)	278 (2.5%)
Ground truth	1.00	11,473	11,195 (66%)	278 (2.4%)

enough to ensure the high performance of subsequent log mining tasks.

To evaluate the effectiveness of log parsing methods on log mining, we apply different log parsers to tackle the parsing challenge of a real-world anomaly detection task. This task employs Principal Component Analysis (PCA) to detect anomalies. Due to the space limit, the technical details of this anomaly detection task is described in our supplementary report [23]. There are totally 16,838 anomalies in this task, which are found manually in [4]. We re-tune the parameters of the parsers for better *parsing accuracy*. LKE is not employed because it could not handle this large amount of data (10m+ lines) in reasonable time. Table 6 demonstrates the evaluation results. *Reported anomaly* is the number of anomalies reported by log mining model (i.e., PCA) while adopting different log parsers in the log parsing step. *Detected anomaly* means the number of true anomalies detected by PCA. *False alarm* is the number of wrongly detected anomalies. *Ground truth* is an anomaly detection task with exactly correct parsed results. Notice that even the ground truth could not detect all anomalies because of the boundary of the PCA anomaly detection model.

From Table 6, we observe that LogSig and IPLoM lead to nearly optimal results on the anomaly detection task. However, SLCT does not perform well in anomaly detection with its acceptable parsing accuracy (0.83). It reports 7,515 false alarms in anomaly detection, which introduces extensive human effort on inspection. Futhermore, the parsing accuracy of SLCT (0.83) and LogSig (0.87) is comparable, but the performance of anomaly detection using LogSig as parser is one order of magnitude better than that using SLCT. Anomaly detection task using LogSig only reports 413 false alarms. These reveal that anomaly detection results are sensitive to some critical events, which are generated by log parsers. It is also possible that F-measure, despite pervasively used in clustering algorithm evaluation, may not be suitable to evaluate the effectiveness of log parsing methods on log mining.

Findings. Log parsing is important because log mining is effective only when the parsing result is accurate enough. Log mining is sensitive to some critical events. 4% errors in parsing could even cause one order of magnitude performance degradation in anomaly detection.

The parameters of POP in this experiment are the same as those tuned for 2k HDFS datasets. We observe that the accurate parsed results of POP are effective from the perspective of this anomaly detection task. Although there

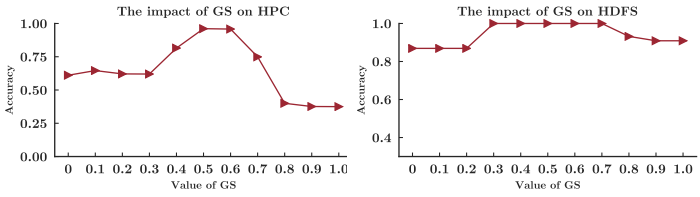

Fig. 9: Impact of GS

are still 37% non-detected anomalies, we think this is the limitation of the anomaly detection model PCA. Because the anomaly detection task with ground truth as input provides comparable performance, where 34% anomalies are not detected. Note that although the performance of the anomaly detection task with POP as input is the same as that of IPLoM in Table 6, their parsing results are different.

4.5 Parameter Sensitivity

To study the impact of parameters, we evaluate the accuracy of POP while varying the value of the studied parameter. All the sensitivity experiments are run on the 2k datasets, which are the datasets used to evaluate the accuracy of the parsers in Section 4.2. Due to the space limit, we only demonstrate the results of parameter GS on HPC and HDFS here in Fig. 9, while the remaining results are provided in our supplementary report [23]. Similar to the parameter setting in our accuracy experiments, we set $splitRel$ to 0.1, $splitAbs$ to 10, $maxDistance$ to 0. We observe that the accuracy of POP peaks for all datasets if we set GS in range [0.5, 0.6]. When GS is smaller, a log group is easier to get shipped to step 4 without further partitioning, which may lower the accuracy because we may put log messages with different log events into the same log group. Thus we can observe the relatively lower accuracy in range [0, 0.3] on HPC and range [0, 0.2] on HDFS. When GS is larger, a log group has higher probability to go through partitioning process in step 3, which may lower the accuracy because we may put log messages with the same log event into different log groups. Thus we can observe the relatively lower accuracy for range [0.8, 1.0] on HPC and range [0.8, 1.0] on HDFS. For dataset BGL, Zookeeper, and Proxifier, POP's accuracy is consistently high (larger than 0.9) under all GS values. POP is also not sensitive to $splitRel$ and $splitAbs$ in our experiments. For $maxDistance$, setting a too large value will cause accuracy drop.

To pick a suitable value, we could first set the parameter to a reasonable value according to its physical meaning. Then we tune it on a small sample dataset by evaluating the resulting accuracy. After finding the best parameter, we can apply it to the original dataset.

4.6 Observations

Among the existing log parsers, LKE has quadratic time complexity, while the running time of others scales linearly with the number of log messages. LogSig is accurate on most datasets. IPLoM is accurate and efficient on small datasets. SLCT requires the least running time. Although these widely used log parsing methods have their own merits, none of them can perform accurately and efficiently on various modern datasets. First, SLCT is not accurate

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2762673, IEEE Transactions on Dependable and Secure Computing

enough. Because of its relatively low parsing accuracy, in our case study in Section 4.4, the false alarm rate of the subsequent anomaly detection task increases to 40%, which causes 7,515 false positives. Secondly, LKE and LogSig cannot handle large-scale log data efficiently. Specifically, LKE has quadratic time complexity, while LogSig needs computation-intensive iterations. Moreover, LKE and LogSig both require non-trivial parameter tuning effort. Finally, IPLoM cannot efficiently handle large-scale log data (e.g., 200 million log messages) due to the limited computing power and memory of a single computer. Our proposed POP is the only log parser that performs accurately and efficiently on all the datasets.

5 Discussions

In this section, we discuss the limitations of this work and provide some potential directions for future exploration.

Diversity of dataset. Not all datasets (two out of five) used in our evaluation are production data, and the results may be limited by the representativeness of our datasets. This is mainly because public log data is lacking. As a result, we cannot claim that our results are broadly representative. However, Zookeeper and HDFS are systems widely adopted by companies for their distributed computing jobs. We believe these logs could reflect the logs from industrial companies to some extent. We also mitigate this issue by generating many sample datasets from the original ones, where each sample dataset has different properties, such as log size and the number of log events. The proposed parser POP at least has consistent accuracy and efficiency on all these datasets, which demonstrates its robustness. Besides, we thank those who release log data [4], [31], [32], which greatly facilitates our research.

Diversity of log mining tasks. Results of effectiveness of log parsing methods are evaluated on anomaly detection, which may not generalize to other log mining tasks. This is mainly because public real-world log mining data with labels is scarce. However, the anomaly detection task evaluated is an important log mining task widely studied [35], [36], which is presented in a paper [4] enjoying more than 300 citations. Besides, even conducting evaluation on one log event mining task, the result reveals that an accurate log parser is of great importance for obtaining optimal log mining performance. We will consider to extend our methodology on more log parsing data and different log mining tasks in our future work.

Logging of Event ID. Log parsing process can also be improved by recording event ID in logs in the first place. This approach is feasible because developers who design the logging statement know exactly the corresponding log event. Thus, adding event ID to logging statement is a good logging practice [37] from the perspective of log mining. Event ID adding tools that can automatically enrich logging statements may greatly facilitate the log parsing process.

Training Log Data Usually we hope to train our log parser on as many logs as possible. This can increase the generalizability of the results obtained by POP. This is also why we propose a parallel log parsing method that aims for parsing large-scale logs. However, we agree that in case we have too many historical logs for processing, sampling is an effective way. We suggest two methods to sample training data. (1) Using the latest logs. This sampling method is more likely to get the newest log events produced by new-version systems. (2) Collecting the logs periodically (e.g., collecting the logs every single day). This sampling method can allow the variability of logs. The quantity of sample logs depends on the training time we can afford. For example, in case of POP, if we want to finish the training process in 7 minutes for HDFS logs, then we can use the latest 200 million log messages.

Log Event Changes. Logs change over time, a log message may not be matched by the current list of log events. To solve this problem, developers can use POP to periodically retrain on new training data to update the list. In runtime, if a log message is not matched by any log events, we mark it as "other events" and recorded. When retraining, the developer can retrain on the log messages marked as "other events", and add the new log events to the log event list. To avoid the burst of not-matched logs (e.g., a billion times), we can maintain a counter to remember the number of log messages marked as "other events" after the latest training. If it is larger than a threshold, an alarm is reported to call for retraining.

POP for Big Data. We propose the parallel log parser POP in the manuscript because the existing nonparallel log parsers and SinglePOP cannot handle the large volume of logs generated by modern systems in the big data era. We can observe from the Fig. 7 that the increasing speed of SinglePOP's running time (i.e., slope) is faster than POP as the log size becomes larger. The running time of SinglePOP will be longer than that of POP on production level log data (e.g., over 200m log messages). For example, the running time of SinglePOP is already larger than that of POP on the 10m HDFS dataset as illustrated. Thus, although SinglePOP is efficient, we need POP, a parallel design on top of Spark, to handle production level log data efficiently.

6 Related Work

Log Management: With the prevalence of distributed systems and cloud computing, log management becomes a challenging problem because of security assurance requirements and the huge volume of log data. Hong et al. [38] design a framework to sanitize search logs with strong privacy guarantee and sufficiently retained utility. Zawoad et al. [39] propose a scheme to reveal cloud users' logs for forensics investigation while preserving their confidentiality. Meanwhile, to assist log analysts in searching, filtering, analyzing, and visualizing a mountain of logs, some promising solutions, such as commercial Splunk [40], and open-source Logstash [41], Kibana [42], have been provided. These solutions provide many plugins/tools for monitoring and analyzing popular system logs (e.g., TCP/UDP, Apache Kafka) and present stunning visualization effects. However, their log parsing procedures are mainly based on prior knowledge and require user-defined matching patterns (e.g., regular expressions). In this paper, we propose a automated log parsing method that can accurately and efficiently parse production-level log data. Besides, the evaluation of log parsing methods gives developers deeper insights on the log parsing procedure.

Log Analysis: Logs, as an important data source, are in widespread use to ensure system dependability. For anomaly detection, Xu et al. [4] propose a PCA-based model, which is trained by system logs, to detect runtime anomalies. Kc et al. [43] detect anomalies by using both coarse-grained and fine-grained log features. As for program verification, Beschastnikh et al. [6] propose Synoptic to construct a finite state machine from logs as system model. Shang et al. [7] analyze logs from both pseudo and cloud environment to detect deployment bugs for big data analytics applications. Log analysis also facilitates system security assurance. Gu et al. [11] leverage system logs to build an attack detection system for cyber infrastructures. Oprea et al. [10] employ log analysis to detect early-stage enterprise infection. Besides, Pattabiraman et al. [44] design an assertion generator based on execution logs to detect application runtime errors. Log analysis is also employed in structured comparative analysis for performance problem diagnosis [9] and time coalescence assessment for failure reconstruction [45]. As shown in our experiemnts, the accuracy and efficiency of log parsing could have great impact on the whole log analysis tasks. Thus, we believe our parallel log parsing approach could benefit future studies on dependability assurance with log analysis.

Log Parsing: Log parsing has been widely studied in recent years. Xu et al. [4] propose a log parser based on source code analysis to extract log events from logging statements. However, source code is often unavailable or incomplete to access, especially when third-party components are employed. Recent work proposes data-driven log parsers (e.g., SLCT [15], IPLoM [22], LKE [5], LogSig [17]), in which data mining techniques are employed. But an open-source implementations of log parsers is still lacking. Many researchers (e.g., [6], [9], [46], [47]) and practitioners (as revealed in StackOverflow questions [48], [49]) in this field have to implement their own log parsers to deal with their log data. Our work not only provides valuable insights on log parsing, but also releases open-source tool implementations on the proposed log parser POP and four representative log parsers

Empirical Study: Empirical studies have attracted considerable attraction in recent years, because the empirical results could usually provide useful insights and direct suggestions to both academic researchers and industrial practitioners. In particular, Yuan et al. [8], [50] conduct an empirical study on the logging practices in open-source systems. Based on their findings, they provide actionable suggestions for improvement and a tool to identify potential unlogged exceptions. Besides, the logging practices in industry has been studied in some recent work [51], [52]. Our work extends the previous conference paper [21], which is an empirical study on log parsing and its subsequent use in log mining.

7 Conclusion

This paper targets automated log parsing for the large-scale log analysis of modern systems. Accordingly, we conduct a comprehensive study of four representative log parsing methods characterizing their accuracy, efficiency and effectiveness on subsequent log mining tasks. Based on the result of the comprehensive study, we propose a parallel log parsing method (POP). POP employs specially designed heuristic rules and hierarchical clustering algorithm. It is optimized on top of Spark by using tailored functions for selected Spark operations. Extensive experiments are conducted on both synthetic and real-world datasets, and the results reveal that POP can perform accurately and efficiently on large-scale log data. POP and the four studied log parsers have been publicly released to make them reusable and thus facilitate future research.

References

[1] The cost of downtime at the world's biggest online retailer (https://www.upguard.com/blog/the-cost-of-downtime-at-the-worlds-biggest-online-retailer).
[2] Downtime, outages and failures - understanding their true costs (http://www.evolven.com/blog/downtime-outages-and-failures-understanding-their-true-costs.html).
[3] Facebook loses $24,420 a minute during outages (http://algerian–news.blogspot.hk/2014/10/facebook-loses-24420-minute-during.html).
[4] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordon, "Detecting large-scale system problems by mining console logs," in *SOSP'09: Proc. of the ACM Symposium on Operating Systems Principles*, 2009.
[5] Q. Fu, J. Lou, Y. Wang, and J. Li, "Execution anomaly detection in distributed systems through unstructured log analysis," in *ICDM'09: Proc. of International Conference on Data Mining*, 2009.
[6] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. Ernst, "Leveraging existing instrumentation to automatically infer invariant-constrained models," in *ESEC/FSE'11: Proc. of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, 2011.
[7] W. Shang, Z. Jiang, H. Hemmati, B. Adams, A. Hassan, and P. Martin, "Assisting developers of big data analytics applications when deploying on hadoop clouds," in *ICSE'13: Proc. of the 35th International Conference on Software Engineering*, 2013, pp. 402–411.
[8] D. Yuan, S. Park, P. Huang, Y. Liu, M. Lee, X. Tang, Y. Zhou, and S. Savage, "Be conservative: enhancing failure diagnosis with proactive logging," in *OSDI'12: Proc. of the 10th USENIX Conference on Operating Systems Design and Implementation*, 2012, pp. 293–306.
[9] K. Nagaraj, C. Killian, and J. Neville, "structured comparative analysis of systems logs to diagnose performance problems," in *NSDI'12: Proc. of the 9th USENIX conference on Networked Systems Design and Implementation*, 2012.
[10] A. Oprea, Z. Li, T. Yen, S. Chin, and S. Alrwais, "Dectection of early-stage enterprise infection by mining large-scale log data," in *DSN'15*, 2015.
[11] Z. Gu, K. Pei, Q. Wang, L. Si, X. Zhang, and D. Xu, "Leaps: Detecting camouflaged attacks with statistical learning guided by program analysis," in *DSN'15*, 2015.
[12] D. Lang, "Using SEC," *USENIX ;login: Magazine*, vol. 38, 2013.
[13] H. Mi, H. Wang, Y. Zhou, M. R. 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, pp. 1245–1255, 2013.
[14] W. Xu, "System problem detection by mining console logs," Ph.D. dissertation, University of California, Berkeley, 2010.
[15] R. Vaarandi, "A data clustering algorithm for mining patterns from event logs," in *IPOM'03: Proc. of the 3rd Workshop on IP Operations and Management*, 2003.
[16] A. Makanju, A. Zincir-Heywood, and E. Milios, "Clustering event logs using iterative partitioning," in *KDD'09: Proc. of International Conference on Knowledge Discovery and Data Mining*, 2009.
[17] L. Tang, T. Li, and C. Perng, "LogSig: generating system events from raw textual logs," in *CIKM'11: Proc. of ACM International Conference on Information and Knowledge Management*, 2011.
[18] C. Manning, P. Raghavan, and H. Schutze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.
[19] Evaluation of clustering. [Online]. Available: http://nlp.stanford.edu/IR-book/html/htmledition/evaluation-of-clustering-1.html
[20] https://github.com/logpai/logparser.

[21] 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 *DSN'16: Proc. of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2016.
[22] A. Makanju, A. Zincir-Heywood, and E. Milios, "A lightweight algorithm for message type extraction in system application logs," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 24, pp. 1921–1936, 2012.
[23] Towards automated log parsing for large-scale log data analysis (supplementary report). [Online]. Available: https://github.com/logpai/logparser/blob/master/logparser/supplementary.pdf
[24] Apache spark (http://spark.apache.org/).
[25] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI'12: Proc. of the 9th USENIX conference on Networked Systems Design and Implementation*, 2012.
[26] J. C. Gower and G. J. S. Ross, "Minimum spanning trees and single linkage cluster analysis," *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, vol. 18, pp. 54–64, 1969.
[27] D. Maier, "The complexity of some problems on subsequences and supersequences," *Journal of the ACM (JACM)*, vol. 25, 1978.
[28] E. F. Krause, *Taxicab Geometry*.
[29] B. S. Everitt, S. Landau, and M. Leese, *Cluster Analysis*.
[30] Apache hadoop (http://hadoop.apache.org/).
[31] A. Oliner and J. Stearley, "What supercomputers say: A study of five system logs," in *DSN'07*, 2007.
[32] L. A. N. S. LLC. Operational data to support and enable computer science research. [Online]. Available: http://institutes.lanl.gov/data/fdata
[33] https://spark.apache.org/docs/latest/configuration.html.
[34] S. Ryza. How-to: Tune your apache spark jobs (part 2). https://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/.
[35] A. Lakhina, M. Crovella, and C. Diot, "Diagnosing network-wide traffic anomalies," in *SIGCOMM'04: Proc. of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 2004, pp. 219–230.
[36] H. Ringberg, A. Soule, J. Rexford, and C. Diot, "Sensitivity of pca for traffic anomaly detection," in *SIGMETRICS'07: Proc. of International Conference on Measurement and Modeling of Computer Systems*, 2007.
[37] F. Salfner, S. Tschirpke, and M. Malek, "Comprehensive logfiles for autonomic systems," in *IPDPS'04: Proc. of the 18th International Parallel and Distributed Processing Symposium*, 2004.
[38] Y. Hong, J. Vaidya, H. Lu, P. Karras, and S. Goel, "Collaborative search log sanitization: Toward differential privacy and boosted utility," *IEEE Transactions on Dependable and Secure Computing (TDSC)*, vol. 12, pp. 504–518, 2015.
[39] S. Zawoad, A. Dutta, and R. Hasan, "Towards building forensics enabled cloud through secure logging-as-a-service," *IEEE Transactions on Dependable and Secure Computing (TDSC)*, vol. 13, pp. 148–162, 2016.
[40] Splunk. http://www.splunk.com.
[41] Logstash. http://logstash.net.
[42] Kibana. http://kibana.org.
[43] K. KC and X. Gu, "Elt: Efficient log-based troubleshooting system for cloud computing infrastructures," in *SRDS'11 Proc. of the 30th IEEE International Symposium on Reliable Distributed Systems*, 2011.
[44] K. Pattabiraman, G. Saggese, D. Chen, Z. Kalbarczyk, and R. Iyer, "Automated derivation of application-specific error detectors using dynamic analysis," *IEEE Transactions on Dependable and Secure Computing (TDSC)*, vol. 8, pp. 640–655, 2011.
[45] C. Di Martino, M. Cinque, and D. Cotroneo, "Assessing time coalescence techniques for the analysis of supercomputer logs," in *DSN'12*, 2012.
[46] 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 Conference on Information and Knowledge Management*, 2016.
[47] S. Banerjee, H. Srikanth, and B. Cukic, "Log-based reliability analysis of software as a service (saas)," in *ISSRE'10: Proc. of the 21st International Symposium on Software Reliability Engineering*, 2010.
[48] What is the best log analysis tool that you used? http://stackoverflow.com/questions/154982/what-is-the-best-log-analysis-tool-that-you-used.
[49] Is there a log file analyzer for log4j files? http://stackoverflow.com/questions/2590251/is-there-a-log-file-analyzer-for-log4j-files.
[50] D. Yuan, S. Park, and Y. Zhou, "Characterizing logging practices in open-source software," in *ICSE'12: Proc. of the 34th International Conference on Software Engineering*, 2012, pp. 102–112.
[51] Q. Fu, J. Zhu, W. Hu, J. Lou, R. Ding, Q. Lin, D. Zhang, and T. Xie, "Where do developers log? an empirical study on logging practices in industry," in *ICSE'14: Companion Proc. of the 36th International Conference on Software Engineering*, 2014, pp. 24–33.
[52] A. Pecchia, M. Cinque, G. Carrozza, and D. Cotroneo, "Industry practices and event logging: assessment of a critical software development process," in *ICSE'15: Proc. of the 37th International Conference on Software Engineering*, 2015, pp. 169–178.

PLACE PHOTO HERE

Pinjia He received the BEng degree in Computer Science and Technology from South China University of Technology, Guangzhou, China. He is currently working towards the PhD degree in computer science and engineering in The Chinese University of Hong Kong, Hong Kong. His current research interests include log analysis, system reliability, software engineering and distributed system. He had served as an external reviewer in many top-tier conferences.

PLACE PHOTO HERE

Jieming Zhu is currently a postdoctoral fellow at The Chinese University of Hong Kong. He received the B.Eng. degree in information engineering from Beijing University of Posts and Telecommunications, Beijing, China, in 2011; the Ph.D. degree from Department of Computer Science and Engineering, The Chinese University of Hong Kong, in 2016. He served as an external reviewer for international conferences and journals including TSC, ICSE, FSE, WWW, KDD, AAAI, SRDS, ISSRE, ICWS, etc. His current research focuses on data monitoring and analytics for system intelligence.

PLACE PHOTO HERE

Shilin He received the BEng degree in Computer Science and Technology from South China University of Technology, Guangzhou, China. He is currently working towards the Ph.D. degree in Computer science and engineering in The Chinese University of Hong Kong, Hong Kong. His current research interests include distributed system, log analysis, and software engineering.

PLACE PHOTO HERE

Jian Li received the BEng degree in Electronic and Information Engineering from University of Electronic Science and Technology of China, Chengdu, China. He is currently working towards the PhD degree in computer science and engineering in The Chinese University of Hong Kong, Hong Kong. His current research interests include data mining, software engineering and distributed system. He had served as an external reviewer in many top-tier conferences.

PLACE PHOTO HERE

Michael R. Lyu is currently a professor of Department of Computer Science and Engineering, The Chinese University of Hong Kong. He received the B.S. degree in electrical engineering from National Taiwan University, Taipei, Taiwan, R.O.C., in 1981; the M.S. degree in computer engineering from University of California, Santa Barbara, in 1985; and the Ph.D. degree in computer science from the University of California, Los Angeles, in 1988. His research interests include software reliability engineering, distributed systems, fault-tolerant computing, and machine learning. Dr. Lyu is an ACM Fellow, an IEEE Fellow, an AAAS Fellow, and a Croucher Senior Research Fellow for his contributions to software reliability engineering and software fault tolerance. He received IEEE Reliability Society 2010 Engineering of the Year Award.