LogSed: Anomaly Diagnosis through Mining Time-weighted Control Flow Graph in Logs

Tong Jia
Peking University
jia.tong@pku.edu.cn

Lin Yang
IBM Research
ylyang@cn.ibm.com

Pengfei Chen
IBM Research
cpfchen@cn.ibm.com

Ying Li
Peking University
li.ying@pku.edu.cn

Fanjing meng
IBM Research
mengfj@cn.ibm.com

Jingmin Xu
IBM Research
xujingm@cn.ibm.com

Abstract— **Detecting execution anomalies is very important to monitoring and maintenance of cloud systems. People often use execution logs for troubleshooting and problem diagnosis, which is time consuming and error-prone. There is great demand for automatic anomaly detection based on logs. In this paper, we mine a time-weighted control flow graph (TCFG) that captures healthy execution flows of each component in cloud, and automatically raise anomaly alerts on observing deviations from TCFG. We outlined three challenges that are solved in this paper, including how to deal with the interleaving of multiple threads in logs, how to identify operational logs that do not contain any transactional information, and how to split the border of each transaction flow in the TCFG. We evaluate the effectiveness of our approach by leveraging logs from an IBM public cloud production platform and two simulated systems in the lab environment. The evaluation results show that our TCFG mining and anomaly diagnosis both perform over 80% precision and recall on average.**

Keywords-interleaved logs; time-weighted control flow graph mining; Anomaly diagnosis

I. Introduction

Software systems are getting increasingly large and complex especially cloud systems that often contain hundreds of components, and support a large number of concurrent users. One particular challenge for large scale software systems is problem diagnosis. That is, when problems occur, how to quickly diagnose problems and identify root causes. Logs are straightforward and common source of information for problem diagnosis. Typically, administrators manually check log files and search for problem-related log lines. However, in today's large scale systems, logs can be overwhelmingly large. For instance, in some large-scale systems that provide global services, the amount of daily log data could reach tens of terabytes (TBs). A Microsoft service system even generates over 1 petabyte (PB) of logs every day[13]. On the other hand, problems of today's systems can be cross-component and cross-service, it is hard to get root causes based on certain "error" logs. Therefore, manually diagnosing problems can be time consuming and error-prone.

Meanwhile, modern software systems, especially cloud applications, become more and more emphasize on their scalability. This makes concurrency and asynchronous the standard design principles. The usage of asynchronous and non-blocking services in cloud poses difficulties to applications' administrating and troubleshooting. For traditional multi-threaded applications, log lines generated by the same requesting serving could be identified by context information, e.g., thread id (TID) or process id (PID), supported by standard logging libraries. On the contrary, logs generated by non-blocking applications lack this context information as one thread or process serves more than one request by multiplexing. Therefore, logs generated by concurrent request servings are interleaved together. The higher concurrency an application supports, the more significant this interleave exhibits.

In this paper, we propose a finer black-box approach for anomaly diagnosis with a graph-based model. Different from prior works, we overcome simplistic assumptions made in prior works including 1) logs contain a task/thread/transaction ID that stitches logs together in each transaction; 2) logs contain multiple different shared parameters that can tie logs together in a transaction; 3) systems are controllable to perform a white-box log mining approach. Furthermore, to further diagnose latency anomalies, we add a time weight recording regular transition time between two nodes to edges of the CFG model. This modified model is called time-weighted control flow graph (TCFG) in this paper.

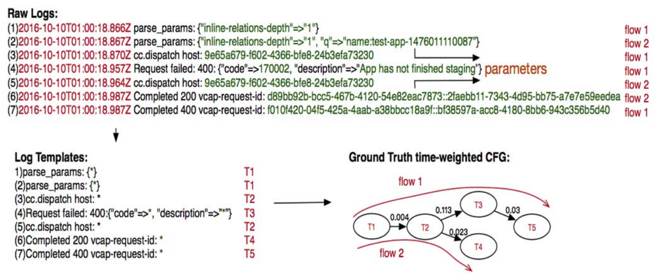
We evaluate the effectiveness of our approach by leveraging logs from an IBM public cloud production platform and two simulated systems in the lab environment. Experiment result shows that our TCFG mining performs over 80% precision and 70% recall on average and our anomaly diagnosis performs near 90% precision and over 80% recall.

The contributions of this paper are as follows:

1. We propose LogSed, an approach to mine time-weighted control flow graphs (TCFG) from interleaved logs of a system without any prior system knowledge or assumptions.
2. We aware that prior works have a hidden assumption that all logs from multiple components represent different transaction flows. In fact, a large number of logs represent individual event messages, such as dispatching events, heartbeat, etc. we propose a two-phase method based on clustering algorithm to identify and filter these logs before mining TCFG in order to improve TCFG quality.
3. We evaluate both the effectiveness of TCFG mining algorithm and the effectiveness of anomaly diagnosis on three log sets from a IBM public cloud production platform and two simulated systems in the lab environment.

This paper is organized as follows. Section 2 further describes our motivation and challenges in detail. Section 3 describes our approach. Section 4 describes our experiment and evaluation. Section 5 analyzes the related work and Section 6 concludes this paper.

2159-6190/17 $31.00 © 2017 IEEE
DOI 10.1109/CLOUD.2017.64



Raw Logs:

```
(1)2016-10-10T01:00:18.866Z parse_params: {"inline-relations-depth"=>"1"}                                     flow 1
(2)2016-10-10T01:00:18.867Z parse_params: {"inline-relations-depth"=>"1", "q"=>"name:test-app-1476011110087"}  flow 2
(3)2016-10-10T01:00:18.870Z cc.dispatch host: 9e65a679-f602-4366-bfe8-24b3efa73230                            flow 1
(4)2016-10-10T01:00:18.957Z Request failed: 400: {"code"=>170002, "description"=>"App has not finished staging"} parameters  flow 1
(5)2016-10-10T01:00:18.964Z cc.dispatch host: 9e65a679-f602-4366-bfe8-24b3efa73230                            flow 2
(6)2016-10-10T01:00:18.987Z Completed 200 vcap-request-id: d89bb92b-bcc5-467b-4120-54e82eac7873::2faebb11-7343-4d95-bb75-a7e7e59eedea flow 2
(7)2016-10-10T01:00:18.987Z Completed 400 vcap-request-id: f010f420-04f5-425a-4aab-a38bbcc18a9f::bf38597a-acc8-4180-8bb6-943c356b5d40 flow 1
```

Log Templates:

```
1)parse_params: {*}                                  T1
(2)parse_params: {*}                                 T1
(3)cc.dispatch host: *                               T2
(4)Request failed: 400:{"code"=>*, "description"=>"*"}  T3
(5)cc.dispatch host: *                               T2
(6)Completed 200 vcap-request-id: *                  T4
(7)Completed 400 vcap-request-id: *                  T5
```

Figure 1. Logs and ground truth TCFG

II. Background and Challenges

A time-weighted control flow graph(TCFG) is a directed graph consisting of edges and nodes and each edge has a time weight recording the transition time. This TCFG stitches together various log message types or *templates* and represents the baseline healthy system state and is used to flag deviations from expected behaviors at runtime. A *template* is an abstraction of a print statement in source code, which manifests itself in logs with different embedded parameter values in different executions. Represented as a set of invariant keywords and parameters (denoted by parameter placeholder *), a template can be used for summarization of multiple log lines. The TCFG is such a graph where the nodes are templates and the edges represent the transition from one template to another. In addition, every log has a *timestamp* indicating its print time, thus the difference between two log *timestamps* represents the program execution time between the two logs. This difference is recorded as the time weight of each edge in the TCFG. Figure 1 shows an example of logs, log templates and a ground truth TCFG. Each log has some invariant keywords and some variable parameters (shown in green) and log templates only reserve invariant keywords. Nodes in the TCFG are different log templates, edges represent how each control flow passes between nodes, and weight of edges indicates the transition time between two nodes.

To mine a CFG structure from execution logs, some prior works ([1][23][24][25]) assume that the set of nodes are known in advance and recorded events contain a task/transaction ID which ties together the sets of events occurring for each execution of a workflow/process. These assumptions do not hold in many distributed systems including multiple components interacting with various third-party services. Some prior works ([15][17]) make a step forward that they do not require a certain transaction id, instead, they adopt multiple ids such as uuid, thread id, 32 char id, etc. to tie events together. However, this assumption does not hold either, in fact, through analysis on OpenStack[18], 35% logs do not contain any id information[17]. Other works ([5][6]) leverage a white-box approach where source code is available and a duplicated controllable environment is available. In real production systems, source code is usually inaccessible and request tracing operations such as probing are not allowed.

Besides these assumptions, prior works have another hidden assumption that all logs from different components represent different transaction flows. In fact, many logs characterize independent events, thus these logs are independent as well. We divide logs in distributed systems into two types: transactional log and operational log. Transactional log characterizes request/transaction execution logic. Operational log usually characterizes independent events such as heartbeat, dispatching messages, etc. For instance, taking Cloud Foundry[19] as an example, amongst the several primary components, *cloud controller* is responsible for managing the lifecycle of applications which handles requests like start or stop an application. Logs of *cloud controller* record the execution steps of each request including finding containers, parsing parameters, reporting request execution status, etc. HM9000 monitors, determines and reconciles applications to determine their state. We look into the logs of HM9000 listener, and notice that each log represents an independent event for receiving and saving a heartbeat from other components. Note that sometimes transactional logs and operational logs are mixed together in one component. A typical example is that most components execute transaction flows while reporting heartbeat information periodically. Transactional logs can be well mapped to a CFG-liked structure while operational logs cannot because of the lack of causal relationships between logs. Therefore, operational logs need to be identified and filtered before mining CFG models, otherwise, they will bring ineffective and meaningless results.

Our goal is to propose a black-box TCFG mining approach so as to perform anomaly diagnosis. We summarize and highlight several challenges we aim to overcome.

First, given the absence of transaction ids, mining template sequences from interleaved log traces of multi-threaded traces of the same component becomes a hard problem. Without clearly distinguishable transaction demarcations, which is a prerequisite for a suite of process mining techniques proposed in literature, the challenge lies in separating the noise from interleaved sequences. For instance, in Figure 1, in the absence of transaction flow ids, it is non-trivial to separate the execution path of each flow and build the TCFGs. T5 in flow1 immediately follows T4 in flow2 and gives the false impression of an edge existing between nodes T4 and T5.

Second, in order to fully recover the transaction flow through TCFG, it is necessary to split the border of each flow, that is, to identify the start node and the end node of a transaction flow in the TCFG. For instance, in Figure 1, T1 is the start node of flow1 and T5 is the end node of flow1. In prior work with transaction ids, it is easy to solve this problem that the start node is where a transaction id first appears and the end node is the last node this transaction id appears. However, without these specific identifiers, it is hard to split the borders of transaction flows where the end node of flow1 may connect with its start node, because flow1 may execute many times in sequence and T1 always occurs after T5 just like other correlated templates such as T2 and T3.

At last, heterogeneous logs include transactional logs and operational logs in which operational logs represent independent system events and lack of causal correlations with each other. If we do not filter components that print

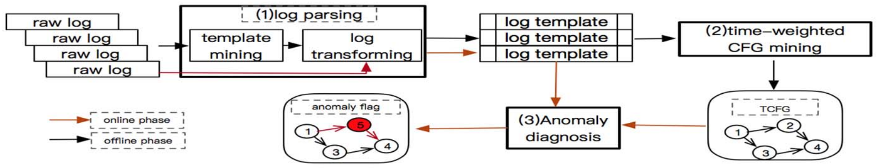


Figure 2. Approach Overview

mostly operational logs, our TCFG mining approach will generate a completely ineffective and meaningless model for this component, which brings huge noises for performing anomaly diagnosis. However, it is still a problem of splitting mixed transactional logs and operational logs, we will discuss this problem in the future work.

III. Our Approach

Figure 2 shows the overview of our approach. Offline phase aims at mining TCFG from logs while online phase aims to flag the deviation between execution log sequences and the mined TCFG model. To support the workflow, we present three major procedures including log parsing, time-weighted CFG mining and anomaly diagnosis. Log parsing is responsible for mining templates in offline phase and matching logs to a certain template in online phase. After mining templates, log parsing succeeds another process of identifying operational logs. Time-weighted CFG (TCFG) mining is responsible for generating TCFG through mining interleaved logs. Anomaly diagnosis is responsible for comparing the online execution log sequence with the mined TCFG to locate anomalies. Next subsections will describe these procedures in detail.

A. Log parsing

Template mining technique is mature and widely utilized in most research work. Our approach leverages the method described in [32] to generate a template set. Then each log is replaced by its certain template. Note that in online phase, templates have already been mined, and online logs are mapped to the mined templates immediately.

Then we aim to filter operational logs. To figure out the difference between operational logs and transactional logs, we manually examine operational logs from tens of different components of an IBM public cloud production platform, and we demonstrate the following two observations. First, operational log templates are independent with others. For transactional logs, each template is a part of the transaction flow, thus each template has causal correlation with its predecessor templates and successor templates in the transaction flow. For operational logs, each log template characterizes a certain event, these events are independent. Second, operational logs usually have a similar structure in the same component. For instance, in a message dispatcher component such as a router or a load balancer, each log interprets an event with the same message type (a typical example is nginx server logs) that emphasizes the request transfer destination, method, etc. Intuitively, these logs should belong to similar log templates or even the same template.

Based on the two observations, we propose a two phase method for operational log filtering. To illustrate our method, we first describe several definitions. Logs are arranged by timestamp to form a log sequence. The predecessors of log A indicates other logs whose timestamp is before A, and the direct predecessor means the certain log whose timestamp is nearest to A in the predecessors of A. Similarly, the successors of log A indicates other logs whose timestamp is after A, and the direct successor means the nearest log after log A. The predecessors and successors are all called vicinities. For the first phase, our method prepares an independence test for each log template. An intuitive solution is that we check the direct predecessor and direct successor of all logs of the same template, and if a large portion of these logs have direct predecessor of the same template or successor of the same template, the three templates have a great chance to represent as a serial part of a transaction, this template as well as the template of its predecessors and successors are labeled as transactional templates. However, since logs from multiple transaction flows may interleaved together, the direct predecessor or successor can be noisy from logs of other interleaved transaction flows.

We overcome this problem by checking several predecessors and successors of a log rather than direct predecessor and direct successor. The intuition behind is that no matter how much the noise is, correct predecessor and successor of a log in the same transaction flow will appear in the near vicinities. For a log template, our independence checking phase first extracts several nearest predecessors and successors(vicinities) of each occurrence of the reference template. Then we record the occurrences of each template appeared in the nearest vicinities of the reference template. If all templates occur very few times in the vicinities (determined by a small threshold), the reference template is identified as operational template. Note that we need to use a relatively small threshold (20% here) to reduce as many false positives as possible. Consequently, small threshold brings a relatively large false negatives, but we have a chance to deal with false negatives in the second phase.

For the second phase, we propose a clustering-based method. Since templates of operational logs are usually similar in a component, we apply an unsupervised clustering algorithm DBSCAN[28] on the templates based on Levenshtein distance[27]. Then we count the occurrences of all templates in each cluster and then compute the occurrence percentage of every cluster. At last, we set a high threshold (say 70% here) to check if most logs belong to the same template cluster, that is, whether there is a dominating template cluster. If so, the templates in the dominating template cluster are labeled as operational templates.

At last, if most logs (decided by an adjustable threshold q) of a component are identified as operational logs, we filter this component before mining TCFGs.

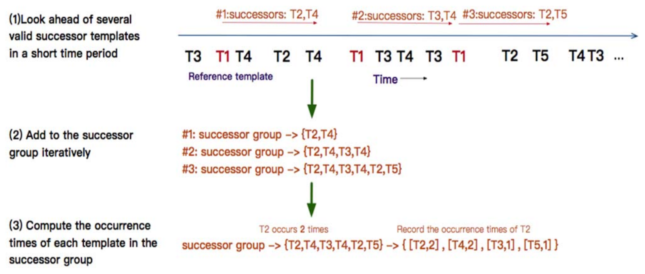

(1)Look ahead of several valid successor templates in a short time period
#1:successors: T2,T4
#2:successors: T3,T4
#3:successors: T2,T5
T3 T1 T4 T2 T4 T1 T3 T4 T3 T1 T2 T5 T4 T3 ...
Reference template
Time
(2) Add to the successor group iteratively
#1: successor group -> {T2,T4}
#2: successor group -> {T2,T4,T3,T4}
#3: successor group -> {T2,T4,T3,T4,T2,T5}
(3) Compute the occurrence times of each template in the successor group
T2 occurs 2 times
Record the occurrence times of T2
successor group -> {T2,T4,T3,T4,T2,T5} -> { [T2,2] , [T4,2] , [T3,1] , [T5,1] }

Figure 3. Successor group generation for FS group computation using template time-series

B. *Time-weighted CFG mining*

Three challenges exist in this step. First, to robustly mine the edges between log templates when operating on interleaved logs. Second, to compute the interval time of the transition edge in the TCFG. Third, to split the border of each flow in the TCFG. We will discuss our solutions in the next subsections.

1) Time-weighted CFG edge mining

We propose a two-stage TCFG edge mining approach. In the first stage, we compute frequent successor groups called FS groups of each template, and the second stage leverages this FS group to mine only the immediate successors of each template. The intuition behind our two-stage algorithm is that the occurrence of immediately succeeding log of a reference template in the input log stream may be noises resulting from interleaving, however, logs of correct immediately succeeding template of the reference template will finally appear in a short period.

(*Stage 1*) The FS group of a template is a set of other templates that are observed to statistically temporally co-occur with the reference template. FS groups of each reference template can be computed by analyzing the time-series of each template. The FS group computation consists of successor group generation step and noise filtering step. Given the time-series of each template, the successor group generation aims at mining all the successors of the reference template occurred in the log stream in a short time period. Figure 3 shows an example of successor group generation step. First, our approach looks ahead of several valid successor templates of each occurrence of the reference template in a short time period *t*. A small *t* will lead to eliminating correct successors while a large *t* will bring too many noises. The value selection of *t* depends on our needs for a higher precision or a higher recall. Second, it adds the successors of each occurrence of the reference template into the successor group. Note that if a template occurs more than once as the successor of an occurrence of the reference template in the log stream, this template is only added once to the successor group. For instance, in Figure 3, T1 is an example of the reference template. For the first occurrence of T1 in the log stream, its successors are T4, T2 and T4 in time-series where T4 occurred twice. When we add T2 and T4 to the successor group, T4 is only counted once. The intuition behind is that if T4 is the immediate successor of T1 in a transaction flow, T4 will occur once right after T1, thus the

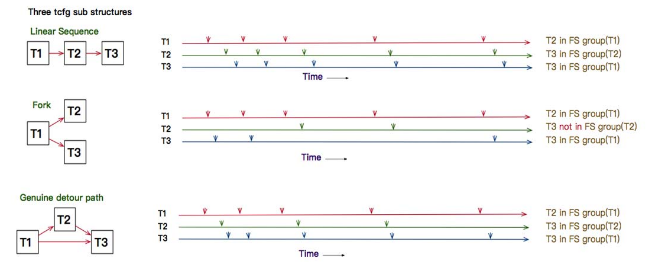


Figure 4. Time series in log stream of three types of TCFG sub-structures

other T4 is certainly noise from interleaving. Third, it computes and records the occurrence times of each successor template in the successor group, thus the final successor group includes multiple successor templates and their occurrence times. For instance, Figure 3 shows three occurrences of T1 in a log stream, T2 appears in the successors of the first and third occurrences of T1, thus is added twice to the successor group of T1 and the occurrence times of T2 is recorded as 2.

In the noise filtering step, our approach is based on the occurrence times of each template in the successor group. As mentioned before, logs of immediate successors of the reference template will finally appear in a short period while noisy log templates randomly appear. Therefore, the succeeding templates must have a higher frequency, i.e, more occurrence times in the successor group. Let us denote the occurrence times of the reference template T_r as N, the number of templates in the successor group as n, the occurrence times of template T_i in the successor group as M_i where $i \in (1, n)$. We compute the possibility of succeeding correlation between the i_{th} template in the successor group and the reference template as:

$$P(T_i | T_r) = \frac{M_i}{N}$$

Then, we set a filtering threshold Θ. For each template T_i, if $P(T_i | T_r) > \Theta$, T_i is added to the final FS group. Otherwise, T_i is filtered. Note that Θ should be relatively small to lead to a loose filtering rule, because forks are common in transaction flows where the transaction flow may transfer from the reference template to multiple different succeeding templates. The succeeding possibility of these templates is less than 100% even without any noise.

(*Stage 2*) The second stage of our TCFG edge mining approach leverages the pre-computed FS group of each template. During mining FS group, instead of mining the immediate succeeding templates, our approach also mines lots of redundant templates of the downstream descendants. Therefore, this stage aims to determine the immediate succeeding templates of each template and recover the TCFG sub-structure near each reference template. Considering that the FS group of T1 includes T2 and T3, here comes the problem of which template is the immediate successor. There exists three sub-structures– linear sequence, fork, and genuine detour path. Figure 4 describes the three fundamental TCFG sub-structures (time weights have been removed for simplicity), and how the time-series of the templates in those sub-structures relate to each other. In the linear sequence, T2 always follows T1 while T3 follows T2 regularly. In the fork, T2 and T3 have a possibility to follow T1. In the genuine detour path, sometimes T1, T2 and T3 form a linear sequence, sometimes T3 follows T1 immediately. Among the three sub-sequences, it is easy to identify fork when T3 is not in the FS group of T2, because T3 have no succeeding correlation with T2. However, it is hard to identify linear sequence and genuine detour path, because T3 will appear in the FS group of T2 in both cases.

Existing works ([22][25][29]) ether do not demonstrate this problem, or remove the succeeding occurrences of templates that are not in FS group of T1 in the log stream, and then check which template (T2 or T3 in our example) occurs immediately succeeding T1, and this template is identified as the immediate succeeding template of T1. Similarly, it does the same operation on T2 so as to distinguish immediate succeeding template of T2. However, templates in the FS group can also be noisy, figure 5 shows a log stream example of a linear sequence. *N* in red denotes filtered noise through FS group mining. When T3 occurs immediately succeeding T1 as an interleaving noise, T3 can be easily identified as the immediate succeeding template of T1.

Our solution to this problem is simple that we expand existing solution to a statistical and probabilistic model. Our assumption is that the interleaving noises inside the FS group are rare. Let us denote the occurrence times of the reference template T1 as N_1, the occurrence times of T3 that immediately succeed T1 as N_3. The possibility of T3 immediately succeeding T1 is defined as as:

$$P(T_3|\, T_1) = \frac{N_3}{N_1}$$

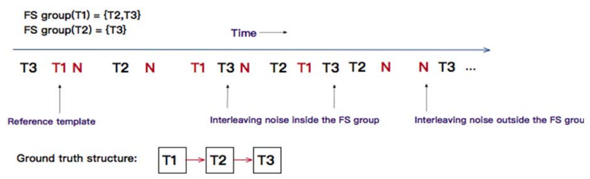
Then we set a very small threshold ∂ (10% here) to determine the existence of an edge from T1 to T3, i.e., whether the sub-structure is linear sequence or genuine detour path.

For each template, once the members of FS group and the successor edges are computed, the temporal vicinity of each template is confirmed. Further, the temporal vicinity of each template can be stitched together to construct the desired structure of TCFG.

2) Edge time weight computation

A time weight on each edge in the TCFG denotes the transition time between two templates, in other words, the overall execution time between two templates. This time weight can be utilized for latency problem diagnosis. For instance, if the time between two consistent logs in a transaction exceeds the time weight recorded in the TCFG, a latency problem is reported and located.

As mentioned at first, input log data for TCFG mining is from the normal running system, thus we assume that the behavior of these logs represents the healthy status of the system. For each two adjacent templates T_i and T_j in the TCFG, we look into the log stream and find all the occurrences of T_i. Then we look ahead a time period *t* (detailed description is in FS group computation stage in TCFG edge mining phase) from each occurrence of T_i and find the first succeeding T_j. In this way, we obtain many <T_i, T_j> pairs in the log stream. After that, we compute the



FS group(T1) = {T2,T3}
FS group(T2) = {T3}
Time
T3 T1 N T2 N T1 T3 N T2 T1 T3 T2 N N T3 ...
Reference template
Interleaving noise inside the FS group
Interleaving noise outside the FS grou
Ground truth structure:
T1
T2
T3

Figure 5. An example of different interleaving noises

difference between the timestamp of T_i and the timestamp of T_j in each pair. The maximum time difference of all <T_i, T_j> pairs is recorded as the time weight. Therefore, if the execution time between T_i and T_j is larger than the time weight, it denotes that we can't find such large latency between T_i and T_j when the system is healthy, thus this is certainly a latency problem.

3) Transaction border splitting

In multi-thread systems, when a thread finishes a transactional task, it then reports its status and wait for the scheduler to check thread pool status, analyze new coming requests and at last assign the next task. Therefore, in general, interval time between two tasks is much longer than the execution time of each step inside one task. We can leverage this feature to split the border of a transaction flow, despite that the interleaving of logs printed by multiple threads may dilute this feature.

When the end node of a transaction flow is connected to its start node, it will form a circle in the TCFG, thus we focus on the circles in the TCFG. Let us denote templates on the path of the reference circle as <T_1, T_2, ..., T_n> where *n* is the number of templates. Each edge on the path determines a transition denoted as a pair <T_i, T_j> where $i, j \leq n$. Here we can't use the time weight on each edge to represent the transition time, because in a few circumstances an execution step in a transaction flow may occupy a much longer time because of resource locks, parameter settings, etc. This will confuse the real execution time of the step. For each edge, we record the transition time of each occurrence of edge <T_i, T_j> in the log stream which can be formulated as $S_{<Ti,\, Tj>}=\{t_1, t_2, ..., t_m\}$. We define a time distribution function $f_{<T_i,T_j>}(t)$ to describe the transition time distribution of the edge between T_i and T_j . For t_k in $S_{<Ti,\, Tj>}$ where $k \leq m$, we introduce a smoothing function $g_k(t)$ which is computed as follows:

$$g_k(t) = \begin{cases} 1, & if\ t \in (t_k - \varepsilon, t_k + \varepsilon) \\ 0, & else \end{cases}$$

where ε is a very small parameter. The intuition behind is that if the transition time is t_k, it is reasonable that transition time shifts slightly. Then the time distribution function $f_{<T_i,T_j>}(t)$ is computed based on the smoothing function:

$$f_{<T_i,T_j>}(t) = \sum_{k=1}^{m} g_k(t)$$

Through this function, we obtain the distribution of transition time between two adjacent templates in the TCFG. The spike of $f_{<T_i,T_j>}(t)$ denotes that a maximum number of occurrence pairs of T_i and T_j have an interval time around

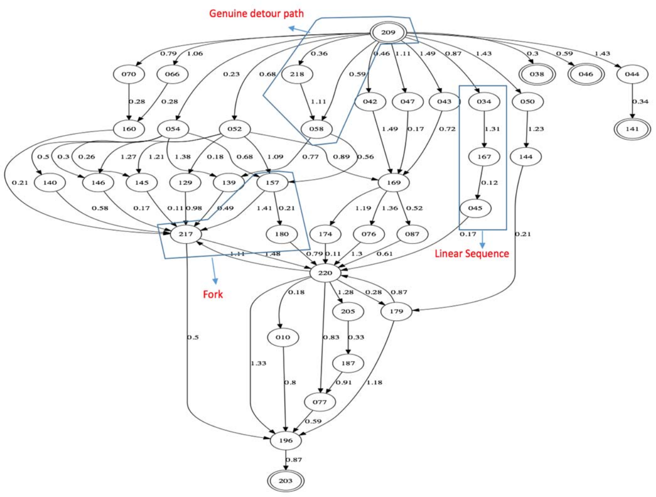


Figure 6. The mined TCFG from execution logs of an IBM public cloud platform.

Table 1. The rate of identified operational logs in two distinct components

Two phases	1st component (operational log set)	2nd component (transactional log set)
1st phase	14.6%	0.7%
2nd phase	79.6%	0%
Total	94.2%	0.7%

the corresponding value of *t* denoted as *t'*. We choose *t'* as the transition time between T_i and T_j.

Next, we can check if there exists an outlier edge whose transition time is much longer than others. This can be implemented as inspecting if the largest transition time is several times larger than others with an adjustable threshold.

C. *Anomaly diagnosis*

In the online phase, the input data includes the mined TCFG from offline phase and a log set printed in a period when the system performs abnormal or a period that system administrators doubt there is a problem hide in. The output is anomaly flag in the TCFG.

The basic idea for anomaly diagnosis is to compare online log stream with the mined TCFG to diagnose the deviation. We start with the start templates in the TCFG because these logs denote the beginning of multiple transaction flows. Then for each occurrence of a start template, we check if the immediate succeeding template in the TCFG appears within an expected time lag interval in the online log stream, then check for the next succeeding template, and so on iteratively. During this time for string together multiple logs, we mainly diagnose for three types of anomalies including *sequence anomaly*, *redundancy anomaly* and *latency anomaly*. A sequence anomaly is raised when none of the children of a parent node is seen within an expected time lag interval in the log stream. A redundancy anomaly is raised when unexpected logs occur that cannot be mapped to any node in the temporal path of the TCFG. An unexpected log can be obvious abnormal log that cannot be matched to any template or a redundant occurrence of a log template. A latency

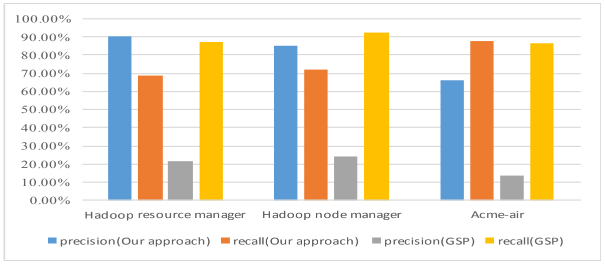


Figure 7. Precision/Recall for TCFG mining

anomaly is raised when the child of a parent node is seen but the interval time exceeds the time weight recorded on the edge.

IV. Experiment and Evaluation

A. *Datasets*

We evaluate the TCFG mining phase and anomaly diagnosis phase separately with one real-world log dataset and two lab environment log datasets. Detailed information of our datasets are as follows:

IBM public cloud platform log set: the cloud platform is a Cloud Foundary[19] based PaaS platform providing cloud services to millions of customers all over the world. It outputs 800 million logs per day. We choose about 300 GB logs of ten hours from the system as our dataset.

Hadoop log set: Hadoop [30] is a popular open source distributed computing platform. We set up a lab environment of Hadoop version 2.7 with 5 nodes in which one for resource manager, one for HDFS and the other three for node manager. We simulate several hundred concurrent jobs using several common benchmarks including WordCount, kmeans clustering algorithm， inverted index, etc. We obtained about 1 GB logs as our dataset.

Acme-Air log set: Acme-Air is an online service of a fictitious airline which is commonly used as a benchmark. Acme-Air is composed of several services including user authentication, querying airlines, booking tickets, etc. We deploy the system in our lab environment, and set the logging level as "debug" so as to generate more detailed logs. We simulate several hundreds of concurrent requests to different services and obtain 450 MB interleaved logs of multiple services as our dataset.

B. *Operational log identification*

We leverage the IBM public cloud platform log set for our evaluation of operational log identification. We select logs from 49 major components that print over 99% percent logs in which our method identifies and filters 15 components that print mostly operational logs. Then we manually examine logs from the 15 components and find these logs are exactly operational logs, which shows our method is effective.

Table 1 shows the rate of identified operational logs in two example components in which the first one prints mostly operational logs and the other prints transactional logs. For the first component with most operational logs, we identified and filtered 14.6% logs in the first phase. In the second phase,

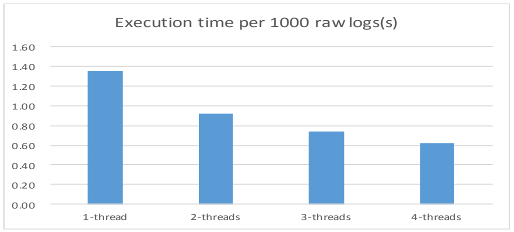


Figure 8. Execution time while running with multiple threads

after clustering, 93.2% of the remaining logs are divided to one group. For transactional component, very few logs are identified as operational logs. Note that in the second phase, largest group occupies only 23.7% of all logs. We observe that the deviation between results of operational component and transactional component is very obvious, thus the final threshold *q* in our method is not sensitive to the filtering result.

C. *TCFG mining evaluation*

We choose IBM cloud platform log set to give a first overview of the effectiveness of our TCFG mining techniques. Figure 6 shows a mined complete TCFG from logs of a major component of the chosen IBM cloud platform. In the figure, double circles denote start nodes or end nodes of transaction flows, which shows our approach is able to split the border of different transaction flows. We also observe that it models all types of sub-structures including genuine detour path, fork and linear sequence.

1) *Effectiveness of TCFG mining*

Here we evaluate the effectiveness of our mined TCFG. We use the standard Precision/Recall metrics for evaluating the TCFG. Recall is the fraction of the ground truth edges that were mined. Precision is the fraction of mined edges that were in the ground truth. Since we need ground-truth TCFG for our evaluation, we utilize Hadoop log set and acme-air log set. We first generate a ground truth TCFG structure manually through looking into the source code and submit a single request to the system to track the transaction flow.

We choose classic frequent sequence mining algorithm as our baseline. Considering interleaved logs as a sequence based on log timestamps, frequent sequence mining algorithms seem to be able to mine frequent patterns from the log sequence. Therefore, a frequent sequence mining algorithm can be applied as an attempt to mine transaction flows from interleaved logs. We leverage GSP[31] algorithm in our experiment, because GSP has two important features. First, it adds time periods to specify the minimum/maximum time interval between adjacent elements in a pattern. Second, it allows elements to present in multiple transactions so as to mine complex structures like fork and genuine detour path. After running GSP algorithm, we mix the mined frequent patterns into a CFG structure by adding an edge to each adjacent element in these patterns.

Results are shown in Figure 7. For both Hadoop resource manager and node manager, the precision is up to 90% percent while the recall is unsatisfied. For Hadoop log set, we notice that our approach mines the transactions of managing resource containers, jobs, and applications successfully, for instance, a container's life circle from New to Allocated to Acquired and finally get to Running. However, our approach neglects the system transaction flow of operations such as security check, function active, logging system setup, etc. This is because these transaction flows appear very few times in the log stream many of which only execute once when the resource manager starts its service. Our approach mistakes these seldom log templates as noise during our noise filtering. As for Acme-Air log set, the result is opposite with a relatively high recall and low precision. This is because when the logs from different services are mixed together based on the timestamp, the interleaving degree is very high leading to many redundant edges, thus generate a lot of false positives.

As for results from GSP, we obtain over 100 frequent patterns from each dataset. After mixing them into a CFG, we find that GSP adds a lot of redundant edges from one node to its down stream descendants. On the other hand, the result of GSP is very comprehensive including most correct transaction flows. Therefore, GSP performs a very good recall and very poor precision. Compared with our approach, GSP performs a better recall, however, GSP result is somehow like a fully connected graph which is meaningless for anomaly diagnosis. Our approach performs a much better precision.

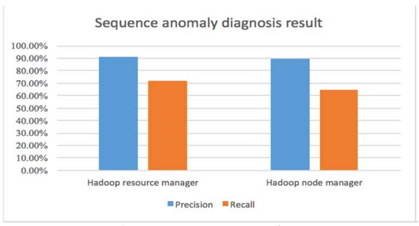
2) *Efficiency of TCFG mining*

We evaluate the efficiency of our TCFG mining approach here. Note that most of our approach is based on statistical model with very few iterations, furthermore, the computation of FS group for each template can be computed in parallel because the whole computation for each template is completely independent without waiting for other intermediate results. Therefore, our approach is scalable for concurrent executions. We evaluate the processing time per 1000 logs running on multiple threads. Results are shown in Figure 8. We observe that when running with 4 threads, the execution time of our approach has been cut by half. Note that our testing environment is a VM on a server with 4 CPU cores and 8GB memory, the efficiency will be further improved while using more powerful computing resources.

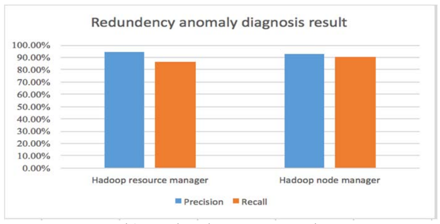
D. *Anomaly diagnosis evaluation*

We also use the standard Precision/Recall metrics for evaluating the anomaly diagnosis performance. Recall is the fraction of anomalies that were correctly diagnosed in all anomalies. Precision is the fraction of correct anomalies in all diagnosed anomalies. To evaluate the capability of anomaly diagnosis, we mock anomalies in log sets and apply fault injection techniques in our experiment. These mocked or injected anomalies cover the three types of anomalies including *sequence anomaly*, *redundancy anomaly* and *latency anomaly* mentioned in section 3.3. We design two different experiments on Hadoop log set and Acme-Air log set separately.

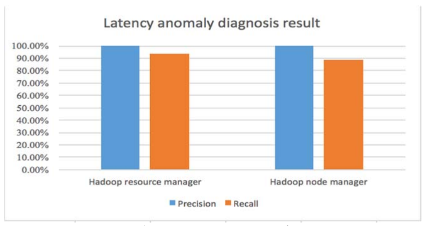
In the first experiment, for each type of anomalies, we inject anomalies to a small fraction (approx. 5%) of nodes and edges in the TCFG, in other words, we mock a few anomalies on the occurrences of these nodes in the log set. Results are shown in Figure 9.



9 a) Sequence anomaly



9 b) Redundancy anomaly



9 c) Latency anomaly

Figure 9. Evaluation results of different types of anomaly

Table 2. Evaluation result for anomaly diagnose on Acme-Air system

Services	Sequence anomalies	Redundancy anomalies	Latency anomalies	Diagnosed anomalies	F/P	F/N
Login	3	2	5	8	1	3
Query flights	3	3	4	10	2	2
Book flights	3	4	3	11	3	2
Check bookings	4	4	2	13	3	0

For sequence anomalies (Figure 9 a)), we observe that precision is very high of approx. 90%. However, the recall is unsatisfied of around 70%. The misses happen when the anomalies are generated at parent nodes of popular merge points. This is because, Despite of the injected sequence anomaly at the parent node, a popular merge point is observed to occur via the path of another parent. At this point, the anomaly diagnosis phase is unable to differentiate the multiple paths to the merge point and will not raise the anomaly.

For redundancy anomalies (Figure 9 b)), the precision and recall scores are overall quite satisfactory. We observe that the cause of misses is that some anomalies are injected to the end node of a genuine detour path, multiple occurrences of the end node will be regarded as the child of the start node, thus will not flag anomalies. False alerts are due to the fact that some rare transaction flows are regarded as unexpected logs and will flag unreal anomalies.

For latency anomalies (Figure 9 c)), the precision and recall scores are very high. The cause of a few misses is that latency problem can be diluted through interleaving. For instance, assuming transition between A and B has a latency problem in the log stream, however, several B are succeeding A due to interleaving of multiple transactions. The last B that exceeds the time weight is the correct successor of A, but during anomaly diagnosis, the previous B which satisfies the latency requirement will be selected as the successor. In this way, a latency anomaly is missed.

In the second experiment, we apply fault injection techniques to the system. We choose 4 major services as injection point, and randomly trigger different types of anomalies ten times for each service. We abort the current execution of requests to simulate sequence anomalies, drop the program into exception handler to simulate redundancy anomalies, and add sleep sentences to the fault point in the program to simulate latency problems.

Results are shown in table 1. The column "Services" shows the four injection point consisting of login, query flights, book flights and check bookings. The column "diagnosed anomalies" represents the number of anomalies our approach identifies. The evaluation metrics are false positives and false negatives shown in "F/P" and "F/N" separately. Among all the 40 problems, 13 of them have related to redundancy anomalies, 14 of them have related to latency anomalies and the other 13 related to sequence anomalies. Our approach detects 33 problems of the total 40 injected problems while mistakenly report 7 problems. These results show a precision of 78.57% and a recall of 82.5%.

V. Related Work

A. Anomaly diagnosis via log analysis

Analyzing logs for problem detection and identification has been an active research area [1~17]. These work first parse logs into log templates based on static code analysis or clustering mechanism, and then build problem diagnosis models. These models can be divided into two categories. *Template frequency based model* usually count the number of different templates in a time window, and set up a vector for each time window. Then it utilizes methods such as machine learning algorithms to distinguish outliers [3~8]. This model is simple for implementation, however, it can not provide help for problem identification and diagnosis. If a problem is detected, engineers still need to manually search logs to find the cause. *Graph-based model* is the current research hotspots. It extracts template sequence at first, and then generating a graph-based model (CFG structure) to compare with log sequences in production environment to detect conflicts ([1][10][11][13][15][17]). This model has three advantages: 1) it can diagnose problems that deeply buried in log sequences such as performance degradation, 2) it can provide engineers with the context log messages of problems, 3) it can provide engineers with the correct log sequence and tell engineers what should have happened.

B. Mining CFG from execution logs

To track the execution path in logs, existing work share an assumption that the behavior of each component is stable, thus the content and the sequence of logs do not change. A first class of existing work assumes there exists some unique identifiers such as task id or thread id, and each task or thread indicate a type of execution path. Qingwei Lin, etc. [13]and Fu Q, etc.[1] propose problem identification approach for Hadoop systems. They use task id to split log events, and extract features from log sequences in different task ids. Then when problems occur, they are able to largely reduce the effort of log searching for engineers. Yu X, etc. [15] and Tak B C, etc. [17] propose methods for extracting proper execution paths. They utilize different ids including uuid, request id, host id, etc. to correlate log events. A second class of approaches leverage classical process mining ([23][24][25][29]) approaches. They operate on the prerequisite of clearly demarcated transactions and require an input dataset comprising a set of such transactions.

VI. Conclusion and Future Work

We have presented LogSed, an approach of diagnosing anomalous run-time behaviors in cloud systems from execution logs. We outlined three challenges that are solved in this paper, including how to deal with the interleaving of multiple threads in logs, how to identify operational logs that do not contain any transactional information, and how to split the border of each transaction flow in the TCFG. Evaluation result shows that our approach performs a good result.

For the future work, we aim improve the recall of our TCFG mining algorithm by leveraging other models. Furthermore, we will correlate TCFGs of different components and hope to generate a whole picture for distributed systems that contains multiple transaction flows across different components. We also aware that identifying operational logs from transactional logs is a problem, we hope to make a study on these logs and combine text features and frequency features to solve this problem.

Acknowledgment

This work is supported by Key Program of National Natural Science Foundation of China (Grant No. 61232005).

References

[1] Fu Q, Lou J G, Wang Y, et al. Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis[C]. Data Mining, 2009. ICDM '09. Ninth IEEE International Conference on. IEEE, 2009:149-158.

[2] Chuah E, Jhumka A, Narasimhamurthy S, et al. Linking Resource Usage Anomalies with System Failures from Cluster Log Data[C]// Proceedings of the 2013 IEEE 32nd International Symposium on Reliable Distributed Systems. IEEE Computer Society, 2013:111-120.

[3] Lim C, Singh N, Yajnik S. A log mining approach to failure analysis of enterprise telephony systems[C]. Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on. IEEE, 2008:398-403.

[4] Kc K, Gu X. ELT: Efficient Log-based Troubleshooting System for Cloud Computing Infrastructures[J]. Proceedings of the IEEE Symposium on Reliable Distributed Systems, 2011, 11(1):11-20.

[5] Xu W, Huang L, Fox A, et al. Online System Problem Detection by Mining Patterns of Console Logs[C]. Data Mining, IEEE International Conference on. IEEE, 2009:588-597

[6] Xu Wei, Huang Ling, Fox Armando, et al. Detecting large-scale system problems by mining console logs[C]. International Conference on Machine Learning. 2010:37-46.

[7] Du S, Cao J. Behavioral anomaly detection approach based on log monitoring[C]. International Conference on Behavioral, Economic and Socio-Cultural Computing. IEEE, 2015.

[8] Wang C, Chen J, Liu X, et al. An improved deep log analysis method based on data reconstruction[C]. IEEE International Conference on Cloud Computing and Intelligence Systems. IEEE, 2014.

[9] Salfner F, Tschirpke S. Error log processing for accurate failure prediction[C]. Usenix Workshop on the Analysis of System Logs. USENIX Association, 2008.

[10] Juvonen A, Hamalainen T. An Efficient Network Log Anomaly Detection System Using Random Projection Dimensionality Reduction[C]. 2014 6th International Conference on New Technologies, Mobility and Security (NTMS). 2014:1-5.

[11] Babenko A, Mariani L, Pastore F. AVA: automated interpretation of dynamically detected anomalies[C]. Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, Il, Usa, July. 2009:237-248.

[12] Yen T F, Oprea A, Onarlioglu K, et al. Beehive: Large-scale log analysis for detecting suspicious activity in enterprise networks[C]. Computer Security Applications Conference. 2013:199-208.

[13] Qingwei Lin, Hongyu Zhang, et al. Log Clustering based Problem Identification for Online Service Systems[C]. International Conference on Software Engineering. IEEE Press, 2016

[14] Liu Y, Pan W, Cao N, et al. System anomaly detection in distributed systems through MapReduce-Based log analysis[C]. Advanced Computer Theory and Engineering (ICACTE), 2010, 2010:V6-410 - V6-413.

[15] X. Zhao, et al., lprof: A nonintrusive request flow profiler for distributed systems[C]. in Proc. 11th Symp. Oper. Syst. Des. Implementa- tion, 2014, pp. 629–644.

[16] Yu X, Joshi P, Xu J, et al. CloudSeer: Workflow Monitoring of Cloud Infrastructures via Interleaved Logs[J]. Acm Sigops Operating Systems Review, 2016, 50(2):489-502.

[17] Tak B C, Tao S, Yang L, et al. LOGAN: Problem Diagnosis in the Cloud Using Log-Based Reference Models[C]. IEEE International Conference on Cloud Engineering. IEEE, 2016:62-67.

[18] OpenStack. http://www.openstack.org

[19] Cloud Foundry. https://www.cloudfoundry.org/

[20] Bilski M. Migration from blocking to non-blocking web frameworks[D]. Dept. Computer Science & Engineering, Blekinge Institute of technology, 2015.

[21] Lou J G, Fu Q, Wang Y, et al. Mining dependency in distributed systems through unstructured logs analysis[C]. Acm Sigops Operating Systems Review, 2010, 44(1):91-96.

[22] Animesh Nandi, Atri Mandal, Shubham Atreja, Gargi B. Dasgupta, Subhrajit Bhattacharya, Anomaly Detection Using Program Control Flow Graph Mining From Execution Logs[C],KDD,2016.

[23] W.V.derAalst, T.Weijters, and L.Maruster. Workflow mining: Fdis Discovering process models from event logs[C]. In *TKDE*, 2004.

[24] C.W.Gunther and W.M.vanderAalst. Fuzzy mining-adaptive process simplification based on multi-perspective metrics[C]. In *BPM*, 2007.

[25] J.G. Lou, Q. Fu, S. Yang, J. Li, and B. Wu. Mining program workflow from interleaved traces[C]. In *KDD*, 2010.

[26] H.Mannila, H.Toivonen, and A.I.Verkamo. Discovery of frequent episodes in event sequences[C]. In *DMKD*, 1997.

[27] Levenshtein V I. Binary codes capable of correcting deletions, insertions and reversals[J]. Problems of Information Transmission, 1965, 1(1):707-710.

[28] Ester M, Kriegel H P, Sander J, et al. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise[C]. 1996:226--231.

[29] I.Beschastnikh, Y.Brun, M.D.Ernst, A.Krishnamurthy, and T.E. Anderson. Mining temporal invariants from partially ordered logs[C]. In *SLAML*, 2011.

[30] Hadoop. https://hadoop.apache.org

[31] Srikant R, Agal R. Mining sequential patterns: Generalizations and performance improvements[J]. Lecture Notes in Computer Science, 1996, 31(6):176--184.

[32] Vaarandi R. A data clustering algorithm for mining patterns from event logs[C]. Ip Operations & Management. IEEE, 2003:119-126.