The Journal of Systems & Software 143 (2018) 172–186

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Execution anomaly detection in large-scale systems through console log analysis

Liang Bao*[,a], Qian Li[b], Peiyao Lu[a], Jie Lu[a], Tongxiao Ruan[a], Ke Zhang[a]

[a] School of Software, XiDian University, Xi'an 710071, China
[b] School of Economics and Finance, Xi'an Jiaotong University, Xi'an 710061, China

ARTICLE INFO

Keywords:
Log analysis
Execution anomaly detection
Control flow analysis
Trace anomaly index

ABSTRACT

Execution anomaly detection is important for development, maintenance and performance tuning in large-scale systems. System console logs are the significant source of troubleshooting and problem diagnosis. However, manually inspecting logs to detect anomalies is unfeasible due to the increasing volume and complexity of log files. Therefore, this is a substantial demand for automatic anomaly detection based on log analysis. In this paper, we propose a general method to mine console logs to detect system problems. We first give some formal definitions of the problem, and then extract the set of log statements in the source code and generate the reachability graph to reveal the reachable relations of log statements. After that, we parse the log files to create log messages by combining information about log statements with information retrieval techniques. These messages are grouped into execution traces according to their execution units. We propose a novel anomaly detection algorithm that considers traces as sequence data and uses a probabilistic suffix tree based method to organize and differentiate significant statistical properties possessed by the sequences. Experiments on a CloudStack testbed and a Hadoop production system show that our method can effectively detect running anomalies in comparison with existing four detection algorithms.

1. Introduction

Nowadays, large-scale systems often consist of hundreds of software components running on thousands of computing nodes. Their runtime data are continuously collected and stored in log files, and analyzed to identify causes and locations of problems in case of system malfunctions and failures. Logs are particularly useful to these large, concurrent and asynchronous systems, because their execution space is too large to sample at testing time, and many untested, or sometimes illegal system behaviors may be observed only after these systems have been put into operation (Mariani and Pastore, 2008).

With the ever increasing size and complexity of system logs, analyzing these logs manually has been experienced as a cumbersome, labor intensive, and error prone task. This is because logs contain a lot of noise, and that key log events are often separated by hundreds of unrelated log messages. There is thus a great demand for automatic anomaly detection approaches based on log analysis.

To analyze log files, operators typically create ad hoc scripts to search for keywords such as "error" or "critical", but this has been proven to be insufficient for determining problems (Jiang et al., 2009; Oliner and Stearley, 2007). Rule-based processing (Prewett, 2003) is a popular improvement, but its problem is that these rules are usually hard to devise and maintain (Xu, 2010). Another direction of improvement is to adapt full text search tools, such as the commercial analysis tool (Splunk, 2018). Although these tools have significant advantages in terms of performance and usability, they still require operators to provide keywords for searching. Unfortunately, the keywords selection process is manual, and often beyond operators' knowledge.

Since unusual log messages often indicate the source of the problem, it is natural to formalize log analysis as an anomaly detection problem in machine learning (Xu et al., 2009c). However, previous work shows that while a single log message, or critical log message is a poor predictor of execution anomalies, a problem often manifests as an abnormality in the relationships among different types of log messages (Jiang et al., 2009; Xu et al., 2009c). Therefore, we apply control flow analysis to the source code to understand all possible combinations of log messages, which guides the whole anomaly detection process thereafter. The augmentation of adding control flow analysis to anomaly detection is an important part of our contribution. Note that given the ubiquitous presence of open-source software in many production systems, the need for source code is not a practical drawback (Xu et al., 2008).

* Corresponding author.
E-mail address: baoliang@mail.xidian.edu.cn (L. Bao).

https://doi.org/10.1016/j.jss.2018.05.016
Received 25 August 2017; Received in revised form 13 May 2018; Accepted 15 May 2018
Available online 17 May 2018
0164-1212/

In this paper, we propose a general four-step approach that integrates control flow analysis, text parsing, and statistical methods to detect execution anomalies in large-scale systems through free-text console logs, without any manual intervention. Specifically, our work involves the following four contributions:

- A formal definition of execution anomaly detection problem through console logs, and some related terminology such as log statement, log message, etc.;
- A technique for analyzing source code to recover the reachability structure of all log statements. This structure reveals all possible combinations of log messages and can be subjected to analysis by our detection algorithm;
- Demonstration of a novel probabilistic suffix tree based statistical method that effectively detects unusual patterns or anomalies in a large collection of log messages;
- Experiments on a testbed running CloudStack and a Hadoop production system show that our method can effectively detect running anomalies in comparison with existing four detection algorithms.

The reminder of this paper is organized as follows. We start off by describing our assumptions and some key observations, presenting the problem formulation, and providing an overview of our approach in Section 2, followed by a detailed explanation of our source code analysis approach in Section 3. Sections 4 and 5 describe log parsing and trace extraction techniques. Section 6 presents our solution for anomaly detection. Section 7 describes the experimental setup and evaluates our approach. We summarize related work in Section 8, and conclude with a discussion of our approach and possible improvements in Section 9.

2. Overview

In this section, we give two basic assumptions about log messages and report some key observations first, and then present the problem formulation and the workflow of our approach.

2.1. Assumptions

Our assumptions are primarily about data normalization of log messages. Due to the characteristics like multi-threading and networking of large-scale systems, much noise is introduced in console log generation and collection process. The two most notable kinds of noise are the inaccuracy of message ordering as well as the random interleaving of messages from multiple threads or processes (Xu, 2010). To avoid noise interference and focus to the problem itself, we assume that the log messages have been cleaned and normalized before analysis.

Assumption 1. log messages satisfy total ordering relation on time dimension. This assumption implies the accuracy of message ordering in log messages: there exists a global clock and each log message must contain a time stamp of global time. This assumption is adopted by many previous researches (Tan et al., 2009; Vaarandi and Pihelgas, 2015), and can be satisfied easily in many prevalent logging libraries by simply activating the option that enables to write log messages to a single log output file.

Assumption 2. the information about execution unit is recorded in each log message. In background of complex large-scale systems, random interleaving of messages from multiple threads or processes is natural and must be taken into consideration. To deal with this issue, this assumption tries to identify the origin of each log message. Specifically, we assume that the specific running information (usually the thread identifier) is written in each log message, which means that we can easily divide log messages into different groups by execution units they belong to. This is a reasonable assumption because many popular logging libraries, such as (Log4j, 2018; Google Glog, 2018), can output information about execution unit for each log message and this option is opened by default.

2.2. Some key observations

Anomalous execution traces are buried in the millions of lines of free-text console logs. We need to identify different execution traces from logs and create the high quality feature (i.e. the numerical representation) for each trace that is understandable by an algorithm to distinguish anomalous traces from normal ones. The following three key observations lead to our solution to this problem.

1. Most of log messages indicate the normal operations of a system.

In production systems, most of the operations are normal, and generate normal log sequences. Unusual log messages often indicate the source of problem and hence are rare. This observation is quite natural and many previous work formalizes log analysis as an *anomaly detection* problem in machine learning (Xu, 2010; Chandola et al., 2009).

2. Source code defines the structures and orders of log messages.

This observation bears to meanings: (1) Although console logs appear in free text form, they are actually structured because they are generated entirely from the log statements. The structure of each log message is defined in the log statements, which can be analyzed and recovered from program source code. (2) The order of log messages within a single process or thread is determined by the control flow and invocation relationships of related log statements in source code.

Consider a real Hadoop console log snippet and the source code that generated it in Fig. 1. For the 4th log message "*Max vcores capability of resources in this cluster 50*", one can easily concludes that "*Max vcores capability of resources in this cluster*" is a constant string and "50" is a variable string according to the log statement *LOG.info("Max vcores capability of resources in this cluster" + maxVCores)*. Actually, we can use Xu's method (Xu, 2010) to get all possible log message template strings from the source code and match these templates to each log message to recover its structure in a high accuracy.

Furthermore, this log excerpt also reflects one possible execution trace of the system, and in turn indicates flows of controls within a process or a thread. Specifically, the log statements labeled as 1, 5, 6, 8, 9 and 3 are executed sequentially to generate the six log messages in Fig. 1.

3. The amount of log statements is relatively small and most of these statements are rarely called.

In Xu et al. (2009c), Xu studied source code and logs from 22 widely deployed open source systems. They found that about 1%-5% of code lines are logging calls in most of the systems. We extend Xu's work by conducting additional statistical evaluation on 10 popular large-scale open source software systems belonging to cloud operating system (2 systems), NoSQL databases (3 systems) and big data computing frameworks (5 systems). According to the statistics that are summarized in Table 1, we find that 0.3%-3% of code lines are log statements. Based on these two independent investigations, the amount of log statements is relatively small compared to the total lines of code in the system, which means that we can perform the thorough control flow analysis for these log statements with a relatively small cost.

During our investigation, another notable phenomenon is observed that most of the log statements represent erroneous execution paths (Xu et al., 2009a; Bezerra and Wainer, 2013). Since log messages usually indicate the normal operations of a system, we can safely conclude that most of these log statements are rarely executed. The result of our experiments confirms this observation.

2.3. Problem formulation

In general, anomaly detection refers to the problem of finding patterns in data that do not conform to expected behavior, and anomalies are patterns in data that do not conform to a well-defined notion of normal behavior (Chandola et al., 2009).

```
2015-04-08 13:22:17,634 INFO org.apache.hadoop.yarn.applications.distributedshell.ContainerLaunchFailAppMaster(main:null): Initializing ApplicationMaster
2015-04-08 13:22:18,461 INFO org.apache.hadoop.yarn.applications.distributedshell.ApplicationMaster(main:null): Starting ApplicationMaster
2015-04-08 13:22:18,558 INFO org.apache.hadoop.yarn.applications.distributedshell.ApplicationMaster(main:null): Max vcores capabililty of resources in this cluster 50
...
2015-04-19 17:59:06,289 INFO org.apache.hadoop.yarn.applications.distributedshell.ApplicationMaster(main:null): Application completed. Stopping running container
2015-04-19 17:59:06,659 INFO org.apache.hadoop.yarn.applications.distributedshell.ApplicationMaster(main:null): Application completed. Signaling finish to RM
2015-04-19 18:00:00,221 INFO org.apache.hadoop.yarn.applications.distributedshell.ContainerLaunchFailAppMaster(main:null): Application Master completed successfully.
                                                                                                                                                    exiting
```

```
public class ContainerLaunchFailAppMaster extends
ApplicationMaster {
 public static void main(String[] args) {
  boolean result = false;
  try {
   ContainerLaunchFailAppMaster appMaster=
    new ContainerLaunchFailAppMaster();
   1. LOG.info("Initializing ApplicationMaster:");
   boolean doRun = appMaster.init(args);
   if (!doRun) {
    System.exit(0);
   }
   result = appMaster.run();
   appMaster.finish();
  } catch (Throwable t) {
   2. LOG.fatal("Error running ApplicationMaster", t);
   System.exit(1);
  }
  if (result) {
   3. LOG.info("Application Master completed
successfully. exiting");
   System.exit(0);
  } else {
   4. LOG.info("Application Master failed. exiting");
   System.exit(2);
  }
 }
}
```

```
public class ApplicationMaster {
public void run() throws YarnException, IOException, InterruptedException {
  5. LOG.info("Starting ApplicationMaster");
  ...;
  int maxVCores = response.getMaximumResourceCapability().getVirtualCores();
  6. LOG.info("Max vcores capabililty of resources in this cluster " + maxVCores);
  if (containerMemory > maxMem) {
   7. LOG.info("Container memory specified above max threshold of cluster."
    + " Using max value." + ", specified=" + containerMemory + ", max="
    + maxMem);
   containerMemory = maxMem;
  }
 }
 protected void finish() {
  8. LOG.info("Application completed. Stopping running containers");
  nmClientAsync.stop();
  9. LOG.info("Application completed. Signalling finish to RM");
  ...;
  try {
   amRMClient.unregisterApplicationMaster(appStatus, appMessage, null);
  } catch (YarnException ex) {
   10. LOG.error("Failed to unregister application", ex);
  } catch (IOException e) {
   11. LOG.error("Failed to unregister application", e);
  }
  amRMClient.stop();
 }
}
```

Fig. 1. Hadoop console log snippet.

Table 1
Console logging in large-scale systems.

System	Ver[a]	Lang[b]	Logger	LOC[c]	LOL[d]
Cloud Operating System					
OpenStack[e]	Liberty	Python	custom	1,288,283	37,487
CloudStack	4.3.2	Java	log4j	1,931,120	9391
NoSQL Database					
HBase	0.94.27	Java	log4j	458,784	3426
MangoDB	3.2.0	C++	custom	654,227	10,914
Cassandra	3.0.0	Java	slf4j	442,651	1204
Big Data Computing Framework					
Hadoop	2.3.0	Java	log4j	1,385,440	9612
Spark	1.4.0	Scala	custom	483,503	2058
Hive	1.2.1	Java	slf4j	1,056,781	3433
Storm	0.10.0	Java	slf4j	134,108	565
Samza	0.10.0	Java	log4j	10,311	161

[a] Ver = version of the software.
[b] Lang = programming language.
[c] LOC = lines of codes in the system.
[d] LOL = number of log statements in the system.
[e] Only 6 core services are included in our statistics.

In the context of execution anomaly detection in software systems through console logs, the goal of this problem is to find anomalous execution traces in log files that do not conform to the frequent normal execution traces. According to Bezerra and Wainer (2013), anomalous execution traces often consist of incorrect executions or some acceptable but rare executions (such as system initiation).

Some previous studies try to define what is an anomalous execution in a system, but few of them give the formal definition. Chandola et al. (2009) discussed that there is no formal definition of anomaly, only intuitions that guide the development of different algorithms and techniques. One of the most important intuitions is that anomalies are data points that have low probability of occurring (given the appropriate generative model for the "normal" data). Bezerra and Wainer (2013) thought that each of the anomalous execution is "infrequent" among the set of all executions, although the whole set of anomalous executions may not be that infrequent.

In this section, we introduce some terminology first, and then define the *execution anomaly detection problem* formally.

Log statement. The term *log statement* abstracts the common structure of all log printing statements in the source code. A log statement is modeled as a 4-tuple $ls = (id, loc, cons, vars)$, where

- *id* is the unique identifer used to distinguish different log statements.
- *loc* contains the source file name and line number in which a log statement is located.
- *cons* represents the constant strings in a log statement.
- *vars* denotes the variables in a log statement.

Consider the log statement *LOG.info("Max vcores capability of resources in this cluster" + maxVCores);* in Fig. 1, its formal definition is summarized in Table 2.

Log message. A log message is a complete line in a log file, which describes a notable execution event of a specific system. Formally, a log message lm is defined as a 5-tuple $lm = (ln, ts, lv, pv, ct)$, where

- *ln* is the line number of this log message in a log file.
- *ts* denotes the time stamp of this message.
- *lv* describes the level (or severity) of this log message, example values are DEBUG, INFO, WARN, ERROR, and FATAL.

Table 2
Definition of a log statement in Fig. 1.

LOG.info("Max vcores capability of resources in this cluster" + maxVCores)	
id	6
loc	ApplicationMaster.java/936
cons	Max vcores capability of resources in this cluster
vars	*maxVCores*

Table 3
Definition of a log message in Fig. 1.

2015-04-08 13:22:18,558 INFO org.apache.hadoop.yarn.applications.distributedshell.ApplicationMaster: Max vcores capabililty of resources in this cluster 50		
	ln	3
	tm	2015-04-08 13:22:18,558
	lv	INFO
pv	*ls*	*see Table 2*
	tid	main:null
	ct	Max vcores capabililty of resources in this cluster 50

- $pv = (ls, tid)$ represents the provenance of this message, where *ls* is the log statement printing this log message (static part), *tid* is the identifer of the running thread that enables the log printing activity (dynamic part).
- *ct* is the content of this message, i.e. a paragraph of free text which describes the meaning of the log message in a way that human can understand.

Consider the third log message in Fig. 1, its formal definition is listed in Table 3.

Homologous log message. Given two log messages lm_i and lm_j, we say lm_i and lm_j are *homologous*, denoted by $lm_i \sim lm_j$, iff:

$$lm_i.\ pv.\ ls = lm_j.\ pv.\ ls$$

the homology concerns about the origin of log messages, i.e. the log statements that can generate log messages. A pair of homologous log messages may be generated by executing the same log statement in two different running threads respectively, or by executing the same log statement twice in one thread.

We then define a utility function called $countL(lm_i, lm_j)$ to judge and count the homology for any two log messages:

$$countL(lm_i, lm_j) = \begin{cases} 1, & lm_i \sim lm_j, \\ 0, & otherwise. \end{cases} \quad (1)$$

Log. Based on the Assumption 2 in Section 2.1, A log $L = \{lm_1, lm_2, ...\}$ is defined as a partially ordered set of log messages with a partial order "$\preceq$", where:

$$\preceq \triangleq lm_i.\ ts \le lm_j.\ ts \quad i \le j,\ lm_i, lm_j \in L$$

it is obvious that the binary relation "$\preceq$" over L is reflexive, antisymmetric, and transitive.

Trace. An execution trace (short for trace) T is a subset of L, which also holds the partial order "$\preceq$".

Homologous trace. Given two traces T_i and T_j, if there exist a *bijection* f: $T_i \to T_j$, where:

$$\forall\ lm_k \in T_i,\ lm_k \sim f(lm_k)$$

we say that T_i and T_j are *homologous*, denoted by $T_i \approx T_j$. Similarly, a pair of homologous traces may be generated by executing the same sequence of log statements in two different running threads respectively, or by executing the same sequence of log statements twice in one thread.

To judge and collect the homology for any two traces, we also define a utility function called $countT(T_i, T_j)$:

$$countT(T_i, T_j) = \begin{cases} 1, & T_i \approx T_j, \\ 0, & otherwise. \end{cases} \quad (2)$$

Valid trace. Given a trace T and the corresponding source code S, if there exists a relation "imply" between S and T, denoted by $S \vdash T$, we say that the trace T *valid*. Here "imply" means that T is one of all possible traces which can be generated by the source code S.

Based on these concepts, we can define the execution anomaly detection problem whose goal is to find the anomalous execution trace in a log file.

Execution anomaly detection problem. Given (1) a log file L consisting of log messages, (2) the source code S that generates L, and (3) a threshold $freq_{max}$ represents the term "infrequent", how to find the set $A = \{T\}$ of all anomalous execution traces that $\forall\ T \in A$, $\frac{|HT|}{|VT|} \le freq_{max}$? Here $HT = \{T' | T' \subseteq L \wedge T' \approx T\}$ represents the set of all homologous traces to T, $VT = \{T' | T' \subseteq L \wedge S \vdash T'\}$ is the set of all valid traces in L.

The execution anomaly detection problem poses a unique set of challenges:

First, how to find all valid execution traces in a log file? In other word, given a log file L and the source code S that can generate L, an algorithm must be designed to find VT, the set of all valid traces in L, efficiently. Suppose L has n log messages, the number of all possible traces in L is $2^n - 1$, which equals to the number of non-empty subsets of L. Since a typical log file of the real-world system often contains millions or even billions of log messages, it is infeasible to enumerate each trace T in L and determine whether $S \vdash T$.

Second, how to find a reasonable set of anomalous execution traces in a log file? To obtain a good result, one needs to design an elaborate detection algorithm and carefully choose the value of $freq_{max}$ concurrently, which is really a challenging task.

2.4. Our approach

The main idea of our approach is to leverage control flow analysis on the source code for valid traces identification. The approach is inspired by the control flow graph (CFG) (Allen, 1970), in which the control flow relationships are expressed in a directed graph. In our setting, the log statements correspond to the nodes in the graph and the control flows and function calls between log statements corresponds to edge between the nodes in the graph. Finally, we propose a novel algorithm based on *path anomaly index* to detect anomaly in log files. The results of some extensive experiments have shown that our algorithm can obtain better result in comparison with four state-of-art detection algorithms.

Fig. 2 illustrates the four stages in our approach for execution anomaly detection of large-scale systems through console log analysis.

1. Source code analysis. This step takes the program source code as input. It extracts the *log statement*, the formal structure defined in Section 2.3, for each log printing statement in source code, and then generates the reachability graph to reveal the reachable relations for any two log statements. Fig. 3 shows the example from Fig. 1 consisting of a 6-line log file and the corresponding source code (with 11 log printing statements), and how we process it. We first abstract each log statement into four parts, as shown in Fig. 3(b), and then take the whole source code as input and generate the reachability graph for log statements, as shown in Fig. 3(c).

2. Log parsing. This step is to parse the log file and extract necessary information for log messages. Specifically, each log message is parsed using some predefined regular expressions to get its line number, time stamp, event level, provenance and text content. Note that sometimes the provenance of a log message may be missing in the log file. In such case, we can apply the method proposed by Xu et al. (2009c) to extract all possible log message template strings from log statements in the source code, and then associate each log message with the log statement

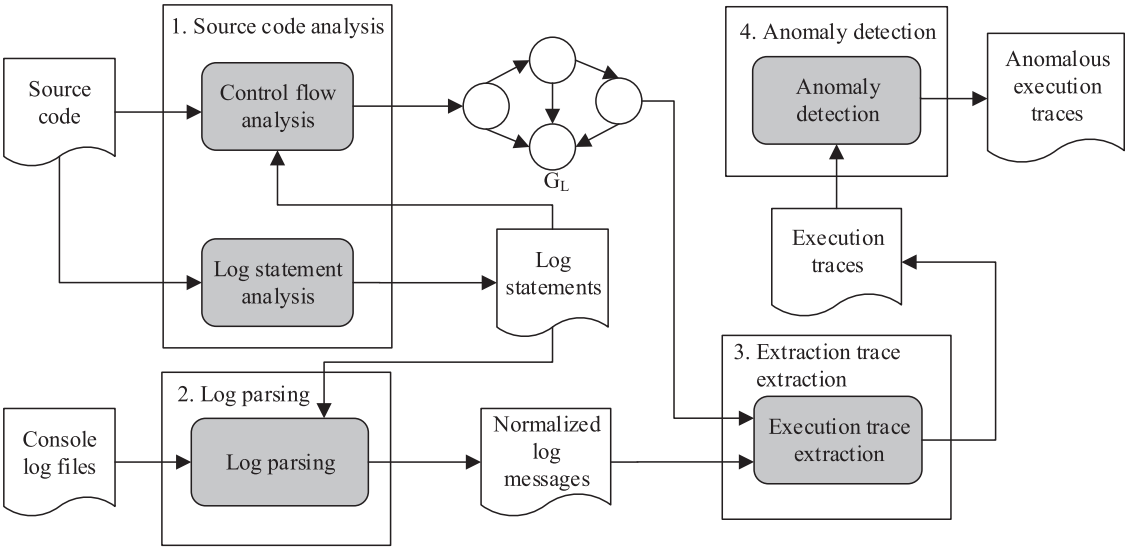


Fig. 2. Overview of our approach.

with a high accuracy. Finally, the chosen log statement and the corresponding thread identifier constitute the provenance of this log message. In our example, the log lines are abstracted into six different log messages, as shown in Fig. 3(d). For each log message, we identify the linkage to its associated log statement, and record the corresponding thread identifer. We use such linkage to associate log messages with formerly generated reachability graph, and group the related log messages together by thread identifers during trace extraction phase.

3. Execution trace extraction. The purpose of this step is to differentiate a log file into many execution traces. To deal with this problem, we propose an execution trace extraction algorithm that partitions the log messages first, and then extract the different traces according to the reachable relations revealed in reachability graph. Our algorithm can avoid the problem of *time window selection*, which has been proven to be a very tricky issue in previous work (Xu, 2010). For the example shown in Fig. 3, our algorithm extracts 578 different traces according to the thread identifers and reachable relations, and classifies them into seven different categories, as shown in Fig. 3(e).

4. Anomaly detection. In this step, we first define *trace anomaly index* for each trace and then propose a novel algorithm to perform anomaly detection. Our algorithm is inspired by the Yang's work (Yang and Wang, 2003), it considers execution traces as sequence data and uses a method of sequence clustering analysis, where a variation of the suffix tree, namely probabilistic suffix tree, is employed to organize and explore significant statistical properties possessed by the sequences. Fig. 3(f) shows the resulting graph representing the trace occurrence probability. Given two traces T_1 and T_6, the trace anomaly index of T_1 is 0.754, and of T_2 is 885.984 according our algorithm (see Section 6), which means that T_1 has much greater possibility to be an anomalous execution trace.

3. Source code analysis

The source code analysis step takes program source code as input. It extracts the useful information for each log statements in source code, and generates the reachability graph that reveals the reachable relations for all log statements.

3.1. Log statement analysis

Since log messages are generated entirely from a relatively small set of log statements in the system, these statements can be used as the "schema" for parsing log messages later. This step leverages the technique of source code analysis to extract and form the set of formalized *log statements* defined in Section 2.3 for all log statements. Log statements can recover the inherit structures of the associated log messages, which help us to accurately parse all possible log messages and eliminate most heuristics and guesses for log parsing used by existing solutions.

Xu's approach (Xu, 2010) is adopted here to analyze source code and extract log statements. The key to this approach is to generate the source code's abstract syntax tree (AST) and do type inference recursively on all objects appearing in the log printing statements. Note that this process is programming language dependent, so we use the AST implementations built into the open-source Eclipse IDE for both Java and C/C++ codes, while dynamic scripting languages like Python has built-in functionality to analyze AST.

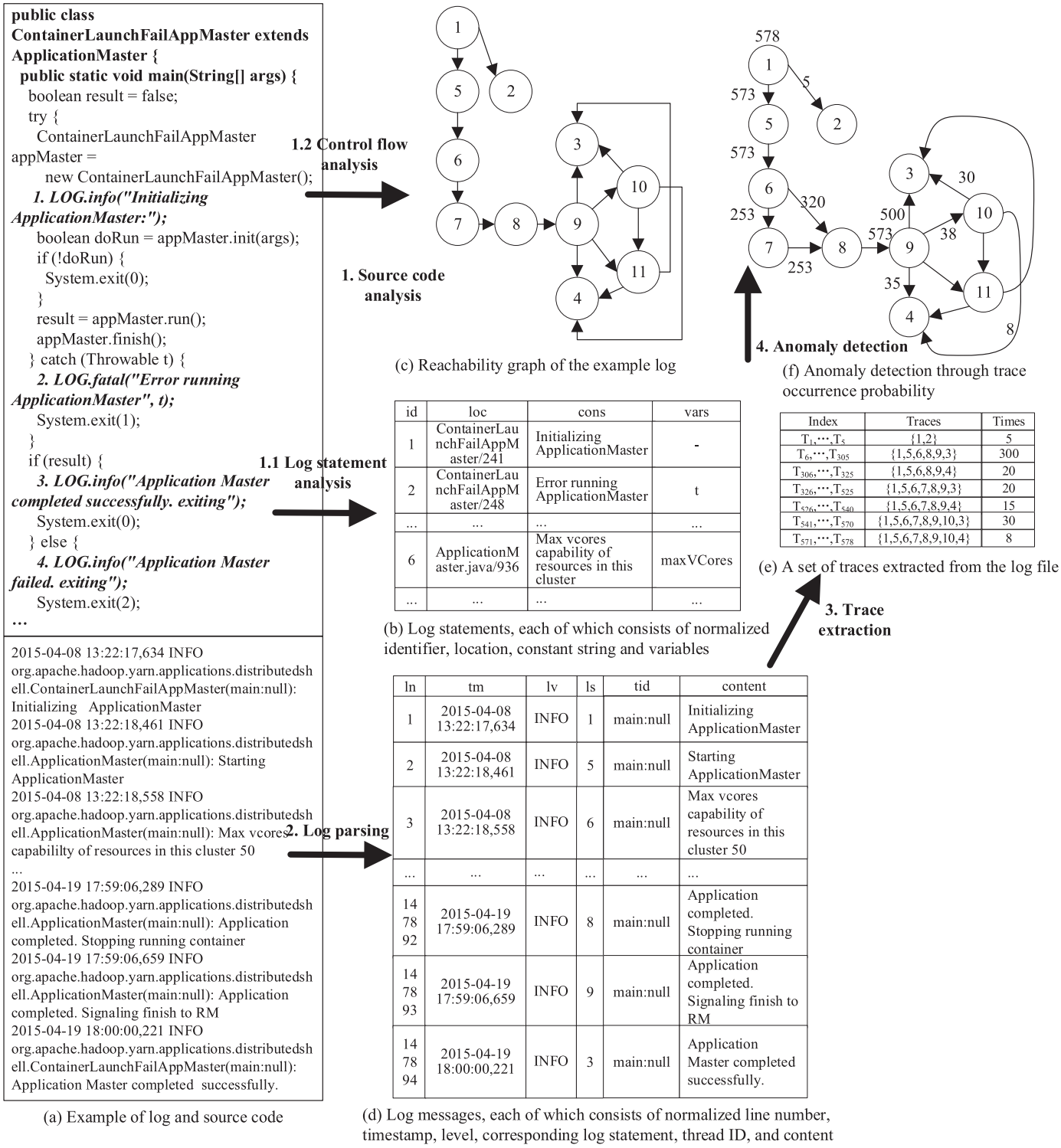
3.2. Control flow analysis

In this step, we focus on constructing reachable relations through control flows and function-calls between log statements, which are unexploited in the most of prior work. We first define the control flow graph with function calls that extends the control flow graph (CFG) (Allen, 1970) a bit further to add function calls as edges to the graph. After that, we propose the reachability graph for log statements to express such reachable relations. Finally, the detailed process of control flow analysis are described, which takes source code as input and generate the reachability graph for log statements.

Control flow graph with function calls. A control flow graph with function calls can be denoted by $G_F = (V_F, E_F)$ where V_F is the set of nodes $\{v_{f_1}, v_{f_2}, ..., v_{f_n}\}$ in the graph and E_F is the set of directed edges $\left\{\left(v_{f_i}, v_{f_j}\right), (v_{f_k}, v_{f_l}), ...\right\}$. Each node in V_F is either a basic block or a function call statement. Each edge is represented by an ordered pair $\left(v_{f_i}, v_{f_j}\right)$ of nodes which indicates that a directed edge, a control flow path or a function call, goes from v_{f_i} to v_{f_j}.

Reachability graph for log statements. A reachability graph for log statements can be denoted by $G_L = (V_L, E_L)$ where V_L is the set of nodes $\{v_{l_1}, v_{l_2}, ..., v_{l_n}\}$ in the graph and E_L is the set of directed edges $\{(v_{l_i}, v_{l_j}), (v_{l_k}, v_{l_m}), ...\}$. Each node in V_L is a log statement. Each edge is represented by an ordered pair (v_{l_i}, v_{l_j}) of nodes which indicates that there exists a reachable relation from v_{l_i} to v_{l_j} through some control flow paths or function calls.

Five steps are involved in the control flow analysis to generate G_L



id	loc	cons	vars
1	ContainerLaunchFailAppMaster/241	Initializing ApplicationMaster	-
2	ContainerLaunchFailAppMaster/248	Error running ApplicationMaster	t
...	...	...	...
6	ApplicationMaster.java/936	Max vcores capability of resources in this cluster	maxVCores
...	...	...	...

Index	Traces	Times
$T_1,\cdots,T_5$	{1,2}	5
$T_6,\cdots,T_{305}$	{1,5,6,8,9,3}	300
$T_{306},\cdots,T_{325}$	{1,5,6,8,9,4}	20
$T_{326},\cdots,T_{525}$	{1,5,6,7,8,9,3}	20
$T_{526},\cdots,T_{540}$	{1,5,6,7,8,9,4}	15
$T_{541},\cdots,T_{570}$	{1,5,6,7,8,9,10,3}	30
$T_{571},\cdots,T_{578}$	{1,5,6,7,8,9,10,4}	8

ln	tm	lv	ls	tid	content
1	2015-04-08 13:22:17,634	INFO	1	main:null	Initializing ApplicationMaster
2	2015-04-08 13:22:18,461	INFO	5	main:null	Starting ApplicationMaster
3	2015-04-08 13:22:18,558	INFO	6	main:null	Max vcores capability of resources in this cluster 50
...	...	...	...	...	...
1478 92	2015-04-19 17:59:06,289	INFO	8	main:null	Application completed. Stopping running container
1478 93	2015-04-19 17:59:06,659	INFO	9	main:null	Application completed. Signaling finish to RM
1478 94	2015-04-19 18:00:00,221	INFO	3	main:null	Application Master completed successfully.

Fig. 3. An example of our approach for detecting anomalies in log messages from Fig. 1.

from source code S:

1. Scan all functions in S recursively from its entry functions. For each function, generate a control flow graph $G_C = (V_C, E_C)$ for it. Repeat this process to form a set of CFGs $\{G_{C_1}, G_{C_2}, ..., G_{C_n}\}$ for S;

2. Initiate a control flow graph with function calls $G_F = (V_F, E_F)$, where $V_F = \{G_{C_1}.V_C \cup G_{C_2}.V_C \cup ... \cup G_{C_n}.V_C\}$, $E_F = \{G_{C_1}.E_C \cup G_{C_2}.E_C \cup ... \cup G_{C_n}.E_C\}$.

3. Given any two CFGs G_{C_i} and G_{C_j}, if there exists a function call goes from a node $v_{c_i} \in G_{C_i}$ to another node $v_{c_j} \in G_{C_j}$, add the edge (v_{c_i}, v_{c_j}) to $G_F.E_F$. Repeat this step until all function calls are processed.

4. Initiate a reachability graph for log statements $G_L = (V_L, E_L)$, where $G_L.E_L = \varnothing$, $G_L.V_L = LS$ and $LS = \{ls_1, ls_2, ..., ls_m\}$ is the set of all log statements.

5. For any two nodes $v_{l_i}, v_{l_j} \in G_L.V_L$, perform Floyd-Warshall algorithm (Floyd, 1962) on G_F to detect whether v_{l_i} and v_{l_j} are reachable. If so, add the edge (v_{l_i}, v_{l_j}) to $G_L.E_L$. Repeat this step until all nodes in G_L are processed.

Consider the Hadoop example showed in Fig. 1, its reachability graph for log statements is shown in Fig. 4. Each labeled node in graph

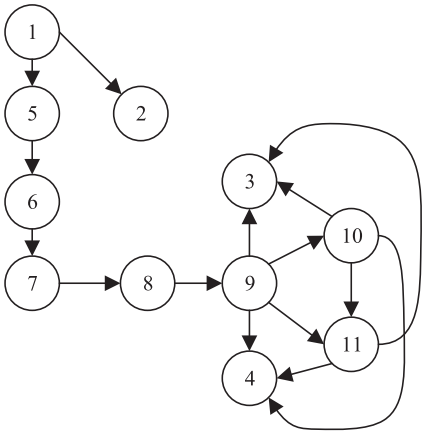


Fig. 4. Reachability graph for Fig. 1.

is associated with the same numbered log statement in Fig. 1, and each directed edge in graph indicates a possible reachable relation between a pair of log statements.

3.3. Complexity analysis of reachability graph generation

We shall start with some notations, and analyze the time and space complexity of reachability graph generation process step by step.

Suppose N is the line of code, F denotes the number of functions in the source code, and L represents the number of log statements, we have:

(1) The first step is to construct a control flow graph $G_C = (V_C, E_C)$ for each function. According to Allen (1970), the time complexity of this process is $O(N^2)$, and the space complexity is $O(N)$.

(2) The second step is to add function call information to each of the control flow graph. This process needs to go through all control flow paths, so its time complexity is $O(N)$, and space complexity is $O(N)$.

(3) The third step is to attach function call information to different graphs. The time complexity of this process is $O(N*F)$ since we have to check every statement and every function call, and the space complexity is $O(N)$;

(4) To initiate the reachability graph for log statements, we should check every log statement. The time complexity of this process is $O(L)$, and the space complexity is $O(L)$;

(5) The last step is to provide reachability information for each pair of log statements. To achieve this, we need to apply Floyd-Warshall algorithm (Floyd, 1962) to each node in G_C and G_F. The time complexity of this process is $O(N^3)$, and the space complexity is $O(N^2)$;

In summary, because the aforementioned five steps are executed sequentially, the final time complexity of the reachability graph generation process is $O(N^3)$, and the space complexity is $O(N^2)$.

4. Log parsing

Two operations are involved in the log parsing step: finding the matching log statement for each log message and extracting according information using the statement.

The most computationally intensive operation here is to find the matching log statement, because there there can be thousands of log statements and millions or even billions of log messages, and we need to match each log message to the set of log statements. In order to accelerate this matching process, we compile all log statements into an Apache Lucene reverse index (McCandless et al., 2010), which allows us to quickly associate any log message with the corresponding log statement. The whole index is only a few MB in size and can easily be loaded into memory in advance.

Once the index is loaded, for each log message *lm* in a log file, the whole process of log parsing goes as following:

1. Constructing a simple regular expression to extract the line number (*ln*), time stamp (*ts*), level (*lv*), thread identifer (*pv.tid*) and content (*ct*) of *lm*.

2. Constructing an index query from *lm* by removing all numbers and special symbols in *lm.ct*.

3. Searching for the index with the query constructed, and rank results by relevance score.

4. From the list of relevance-ranked candidates (i.e log statements) returned by the reverse-index search, associating the highest-ranked log statement to *lm.pv.ls*.

It is quite straightforward that once the reverse index is distributed to each of the parser node, the parsing step is immediately parallel, because parsing of each message is independent of other messages. We implement the log parser as a Hadoop Map-Reduce job by replicating the index to every worker node and partitioning the log among the workers, achieving near linear speedup (Xu et al., 2009c).

5. Execution trace extraction

After log parsing step, each log message is normalized to the standard format and is associated with a log statement. The goal of this step is to differentiate a log file into many execution traces: given a parsed log file consists of n log messages $L = \{lm_1, lm_2, ..., lm_n\}$, we need to find an appropriate partition TS of L that consists of m traces $TS = \{T_1, T_2, ..., T_m\}$, where $T_i \subseteq L$, and $\forall T_i,\ T_j \in TS,\ T_i \bigcap T_j = \varnothing$ $(1 \le i, j \le m)$.

To deal with this problem, we propose an execution trace extraction algorithm that partitions the log messages first, and then extract different traces according to the reachable relations revealed in reachability graph for log statements. The algorithm is explained in Algorithm 1. In this algorithm, we first divide the log messages into partitions according to their *tid*s, and sort log messages in each partition by their time stamps to form "$\preceq$" relations (line 2–3). After this step, log messages having the same *tid* will belong to the same sorted partition. Then, we traverse all log messages in each group from the beginning and determine whether the two adjacent messages are reachable in G_L (line 5–10). If they do in G_L, we add them to the same trace T and continue the traversing process (line 11–14). Otherwise, the newly added log message doe not belong to T, we thus stop this traversing process and add T to the set of traces TS (line 16–17). The algorithm stops when all groups of log messages are processed and the set of traces TS is generated simultaneously. The time complexity of our execution trace extraction algorithm is $O(n*log(n))$, where n is the number of log messages.

Note that once we divide the log messages into groups based on their thread identifers, it will result in inaccurate log sequences in each group. This is because the same thread can process multiple different tasks one after the other, and all these different tasks are unrelated and should not be grouped into the same execution trace. To deal with this issue, after grouping operation, we further extract the possible different execution traces by traversing every log message in each group from the beginning and checking the reachability between every pair adjacent messages. If a pair of such messages is reachable in G_L, they should evolve in a control flow belonging to the same task, i.e. the same execution trace, and we continue this process to add more log messages (line 5–14 in Algorithm 1). Otherwise, we stop this round of traversing process, create a execution trace T including all previous log messages but the last one (line 16–17 in Algorithm 1), and then start a new traversing process recursively with the newly added log message (line 6 in Algorithm 1).

6. Anomaly detection

We have extracted and differentiated traces from a log file in the last section, how to detect anomaly from these traces are definitely the key

Input: a log file with n messages $L = \{lm_1, lm_2, ..., lm_n\}$;
the reachability graph for log statements $G_L = (V_L, E_L)$.
Output: the set of execution traces $TS = \{T_1, T_2, ..., T_m\}$.

1: Initiate $TS=\varnothing$, currently identified trace $T=\varnothing$;
2: According to the *tid* in each log message, divide L into k partitions $P = \{P_1, P_2, ..., P_k\}$, $(P_i \subseteq L, 1 \le i \le k)$;
3: Sort log messages in each group by their time stamps;
4: **for** each partition $P_i = \{lm_1^{P_i}, lm_2^{P_i}, ...\}$ in P **do**
5: **while** $P_i \neq \varnothing$ **do**
6: $lm_i^{P_i} \leftarrow first(P_i)$;
7: $T \leftarrow T \bigcup \{lm_i^{P_i}\}$;
8: **while** $P_i \neq \varnothing$ **do**
9: $lm_j^{P_i} \leftarrow next(P_i, lm_i^{P_i})$;
10: **if** $(lm_i^{P_i}.ls, lm_j^{P_i}.ls) \in E_L$ **then**
11: $P_i \leftarrow P_i - \{lm_i^{P_i}\}$;
12: $T \leftarrow T \bigcup \{lm_j^{P_i}\}$;
13: $lm_i^{P_i} \leftarrow next(P_i, lm_j^{P_i})$;
14: $P_i \leftarrow P_i - \{lm_j^{P_i}\}$;
15: **else**
16: $TS \leftarrow TS \bigcup \{T\}$;
17: $T \leftarrow \varnothing$;
18: **break**;
19: **end if**
20: **end while**
21: **end while**
22: **end for**

Algorithm 1. Extracting execution trace from a log file.

to our approach. In this section, we first define the *trace anomaly index* for each trace and then propose a novel anomaly detection algorithm to solve this problem.

As the set of traces is a partition of all log messages, we can draw the following conclusion according to the first observation in Section 2.2: most of the traces indicate normal operations and anomalous traces are rare and often indicate unusual operations (Xu et al., 2009a; Bezerra and Wainer, 2013; Chandola et al., 2009). Based on this conclusion, we can distinguish between normal and anomalous traces by measuring the similarity between traces. The idea here is straightforward: all normal traces are similar in values of some features, and these values may be quite different for anomalous traces.

We define the *trace anomaly index* to measure the similarity between different traces. This definition is inspired by Yang and Wang (2003), in which they derive a conditional probability distribution (CPD) of the next symbol given a preceding data sequence segment and used to characterize sequence behavior and to support the similarity measurement.

The key to our approach is that normal execution traces should subsume to the same probability distribution of symbols (conditioning on the preceding segment of a certain length), while anomalous execution traces may follow different underlying probability distributions. This feature, typically referred to as *short memory*, indicates that, for a certain execution trace, the empirical probability distribution of the next log message given the preceding segment can be accurately approximated by observing no more than the last L log messages in that segment. By extracting and maintaining significant patterns characterizing normal/anomalous clusters, one can easily determine whether a execution trace should belong to a cluster by calculating the likelihood of (re)producing the trace under the probability distribution that characterizes the given cluster. In the context of anomaly detection problem, this CPD-based method is more accurate and robust than simply counting occurrences and frequencies of log messages in each trace.

Trace anomaly index. Given a set of traces $TS = \{T_1, T_2, ..., T_n\}$, the trace anomaly index for any trace $T_i = \{lm_1, lm_2, ..., lm_m\}$, denoted by $\mathscr{A}_{T_i}$, is defined as follows:

$$\mathscr{A}_{T_i} = -\frac{\mathscr{F}_{T_i}}{\mathscr{F}_a} * \left(|T_i| \times \frac{1}{\log(p_1^{T_i} \times p_2^{T_i} \times ... \times p_m^{T_i})} \right) \tag{3}$$

where

- $\mathscr{F}_{T_i}$ denotes the frequency of T_i, i.e. the number of homologous traces with T_i in TS.

$$\mathscr{F}_{T_i} = \sum_{j=1}^{|TS|} countT(T_i, T_j) \tag{4}$$

- $\mathscr{F}_a$ denotes the average frequency of TS, i.e. the average number of homologous traces for each trace in TS.

$$\mathscr{F}_a = \frac{\sum_{j=1}^{|TS|} \mathscr{F}_{T_j}}{|TS|} \tag{5}$$

- $|T_i|$ is the length of the T_i, i.e. the number of log messages in T_i.
- $(p_1^{T_i} \times p_2^{T_i} \times ... \times p_m^{T_i})$ represents the trace occurrence probability that accumulates from the start log message to the stop log message in T_i.

$$\begin{cases} p_j^{T_i} = \dfrac{\sum_{k=1}^{|TS|} \sum_{l=1}^{|T_k|} countL(lm_l^{T_k}, lm_j^{T_i})}{|TS|}, j = 1 & (6) \\ \dfrac{\sum_{k=1}^{|TS|} \sum_{l=1}^{|T_k|-1} countT((lm_l^{T_k}, lm_{l+1}^{T_k}), (lm_{j-1}^{T_i}, lm_j^{T_i}))}{\sum_{k=1}^{|TS|} \sum_{l=1}^{|T_k|} countL(lm_l^{T_k}, lm_{j-1}^{T_i})} & (7) \end{cases}$$

We propose our anomaly detection algorithm based on the definition of trace anomaly index. The detailed steps of our algorithm is listed in Algorithm 2. The algorithm first calculates the frequency for each trace T_i in TS according to the Eq. (4) (line 2–4), and then updates the average frequency $\mathscr{F}_a$ of TS according to Eq. (5) (line 5). For each log message lm_j in T_i, its occurrence probability is computed according to Eq. (6) (when $j = 1$) or Eq. (7) (when $j \in [2, |TS_i|]$) (line 7–13). The value of anomaly index for each trace T_i can be obtained according to Eq. (3) (line 14). Given a pre-defined threshold $freq_{\max}$, the first $\lfloor m * freq_{\max} \rfloor$ traces with smaller anomaly index values are returned as the candidates of anomalous traces.

Consider again the example shown in Fig. 1, suppose we have extracted a set of 578 traces from a raw log files. Table 4 lists the index for each trace and the number of log statements involved in it, and shows the occurrence times for each trace. According to our algorithm, given two traces T_1 and T_6, we have $\mathscr{F}_{T_1} = 5$; $\mathscr{F}_{T_6} = 300$; $\mathscr{F}_a = 6.43$; $p_1^{T_1} = \frac{578}{578}$, $p_2^{T_1} = \frac{5}{578}$; $p_1^{T_6} = \frac{578}{578}$, $p_2^{T_6} = \frac{573}{578}$, $p_3^{T_6} = \frac{573}{573}$, $p_4^{T_6} = \frac{320}{573}$, $p_5^{T_6} = \frac{573}{573}$, $p_6^{T_6} = \frac{500}{573}$. We visualize the process of calculating trace occurrence probability in Fig. 5 by attaching the occurrence times of these trace to our *reachability graph* shown in Fig. 4. Finally, we have $A_{T_1} = 0.754$ and $A_{T_6} = 885.984$, which means that T_1 has the greater possibility to be a anomalous execution trace.

7. Experiment and evaluation

We have implemented our approach and conducted extensive experiments using CloudStack (2018) and Hadoop (2018). In this section, we first describe our experiment setup, and then present the experimental results to prove the efficiency and effectiveness of the proposed approach.

Input: a set of traces $TS = \{T_1, T_2, ..., T_m\}$ with m traces;
the threshold $freq_{max}$.
Output: the set of anomalous traces ATS.

Initiate $\mathcal{F}_a \leftarrow 0$; $ATS \leftarrow \varnothing$;
all $\{\mathcal{F}_{T_i}\} \leftarrow 0$; all $\{\mathcal{A}_{T_i}\} \leftarrow 0$; all $\{p_i\} \leftarrow 0$;
for each T_i in TS **do**
  update $\mathcal{F}_{T_i}$ according to Eq. 4;
end for
update $\mathcal{F}_a$ according to Eq. 5;
for each T_i in TS **do**
  for each lm_j in T_i **do**
    if j=1 **then**
      update p_j according to Eq. 6;
    else
      update p_j according to Eq. 7;
    end if
  end for
  update $\mathcal{A}_{T_i}$ according to Eq. 3;
end for
sort $\{\mathcal{A}_{T_i}\}$ in ascending order;
for i=1 to $\lfloor m * freq_{max} \rfloor$ **do**
  select the ith elements in $\{\mathcal{A}_{T_i}\}$;
  add the corresponding trace to ATS;
end for

Algorithm 2. Anomaly detection algorithm.

Table 4
A set of traces extracted from a raw log file.

Index	Traces	Frequency
$T_1, ..., T_5$	{1, 2}	5
$T_6, ..., T_{305}$	{1, 5, 6, 8, 9, 3}	300
$T_{306}, ..., T_{325}$	{1, 5, 6, 8, 9, 4}	20
$T_{326}, ..., T_{525}$	{1, 5, 6, 7, 8, 9, 3}	200
$T_{526}, ..., T_{540}$	{1, 5, 6, 7, 8, 9, 4}	15
$T_{541}, ..., T_{570}$	{1, 5, 6, 7, 8, 9, 10, 3}	30
$T_{571}, ..., T_{578}$	{1, 5, 6, 7, 8, 9, 10, 4}	8

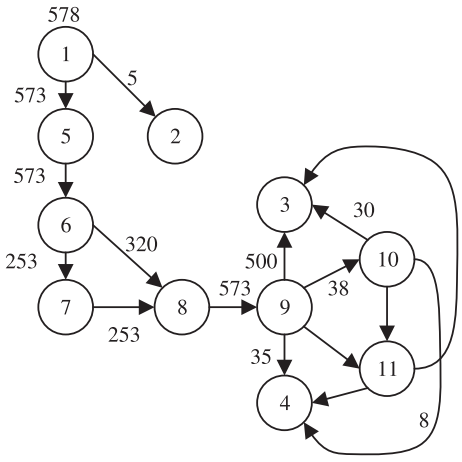


Fig. 5. The visualized process of calculating trace occurrence probability.

7.1. Experiment setup

Our experiments are conducted on our university's public cloud computing environment, a production cloud computing infrastructure, which operates in a similar way as Amazon EC2 (Amazon elastic compute cloud, 2018). Each host has two 10-core E5-2650L V2 1.7GHz CPUs and 256GB memory, and runs CentOS 6.4 64-bit with KVM 3.0.3. The guest VMs also run CentOS 6.4 64-bit.

We first use Apache CloudStack as a case study application. In our experiment, we deploy a small-scale Apache CloudStack system for deploying and managing a simple network of virtual machines. The system has been running continuously for 17 days, resulting in 1,008,950 log messages, the size of data set is 185MB.

The second data set comes from a large-scale Hadoop HDFS production system for big data analytics. We extract all 59,708,045 log messages of this system in 10 days and the size of this data set is about 5.6GB.

7.2. Baseline algorithms

To evaluate the performance of our CFG approach, we compare it with four state-of-the-art algorithms, namely PCA (Xu et al., 2009a), SAMP (Bezerra and Wainer, 2013), CLSTR (Dickinson et al., 2001), and SRCH. We provide a brief description for each algorithm and report its hyperparameters (if necessary) as follows:

PCA [1] is an unsupervised machine learning based algorithm for mining system console logs to automatically detect system runtime problems. In this algorithm, logs are parsed by combining source code analysis with information retrieval to create composite features, then these feature are analyzed using principal component analysis (PCA) to detect operational problems. We follow the suggested settings in Xu et al. (2009a) and Xu (2010).

SAMP [2] is a frequent model based algorithms for detecting anomaly logs in process aware systems. The SAMP algorithm is based on the idea that a sample of the log should not contain an anomaly, since they are infrequent. Thus if one selects a sample of the log, mine the model from it, anomalies should not be instances of the model. Thus all infrequent that are not instances of the mined model are considered an anomaly. We use the hyperparameters suggested in Bezerra and Wainer (2013).

CLSTR is a cluster filtering based anomaly detection algorithm. The rationale for this algorithm is that if failures in a population of executions are infrequent and have unusual profiles, then it should be possible to cluster the population so as to isolate a significant proportion of the failures in small clusters. CLSTR uses adaptive sampling method and designs a dissimilarity metrics giving extra weight to unusual profile features. We have implemented the CLSTR using Java, and set the values of hyperparameters, such as clustering techniques, number of cluster, and sampling strategy, as suggested in Dickinson et al. (2001).

SRCH represents the keyword search based anomaly detection. We hired six administrators, 3 for CloudStack and 3 for Hadoop, from industry to perform keyword search based anomaly detection process, each of which has 2–3 years of operation and maintenance experience. Specially, for each experiment, we first allow these administrators to install their favorite full text search tools for log analysis, and then give them two hours to detect anomaly (which is about 2 times more than running time of our CFG algorithm). Last, we take the average of their performance as the results of SRCH approach.

We have implemented CFG algorithm using Java, and the source code of the algorithm is available online.[3] In CFG algorithm, the choice of the $freq_{max}$ for anomaly detection has been studied in previous work (Lakhina et al., 2004; Xu et al., 2009a; Bezerra and Wainer, 2013) and

[1] Source code is available from https://github.com/xuw/logm.
[2] Source code was downloaded from http://www.fabiobezerra.pro.br/logs/.
[3] https://github.com/sselab/LogAnalysis.

we follow standard recommendations and choose $freq_{max} = 0.1$ in our experiments. We also found that our detection results are not sensitive to this parameter choice.

7.3. Evaluation measures

In order to discuss the quality metric between different approaches, we adopt the standard metrics for a binary classifier in our experiment. The classes of a binary classifier are usually referred to as "positive" and "negative". In our case, "positive" refers to being an anomaly, and "negative" to being a normal trace. We have:

- A true positive (*TP*) is a trace that is classified as positive by the approach and it is indeed positive.
- A false positive (*FP*) is a trace that is classified as positive by the approach, but it is really a negative trace (i.e. a normal trace).
- A false negative (*FN*) is a trace that is classified as negative by the system, but it is really a positive trace (i.e. an anomaly).

The metrics for quality of each approach are:

- $Precision = TP/(TP + FP)$
- $Recall = TP/(TP + FN)$
- $f\text{-}measure = \frac{2 \times Precision \times Recall}{Precision + Recall}$

Note that the goal of anomaly detection is to find anomalous traces in log messages, so we will ignore the *true negative* indicator and the *accuracy* metric in our evaluation.

7.4. Cloudstack experiment results

In the experiment, we deploy an unmodified CloudStack 4.1.2 distribution on a cluster with 10 virtual nodes, and turn on the *DEBUG* level logging. We invite 20 volunteers to perform many daily operations, such as turning on virtual machines, creating images, destroying virtual machines, etc., on the system. We keep the experiment running for 17 days and collect 1,008,950 log messages. After that, we ask five CloudStack experts (two from our university and three from industry) to search for anomalous operations in these messages. They find 44 different abnormal operations that can be classified into 10 categories. Such results are adopted as our ground truth of this experiment, as shown in Table 5.

The four algorithms are executed on this data set are also shown in Table 5. We observe directly that the CFG algorithm can detect a large fraction of anomalies (36 of 44 anomalies) in the data set. Table 6 shows the detailed results according to our evaluation measures. The statistical analysis of the results shows that in terms of precision, the PCA based algorithm gets the best score, followed by CFG and SRCH, and finally the sampling algorithm is similar to the clustering. This is because PCA based algorithm doesn't make any mistake (i.e. $FP = 0$) although it finds out only 12 anomalies.

Table 5
Actual and detected anomalies using different algorithms.

Cat.	Anomaly Description (# of log messages[a])	ACT[b]	PCA	SAMP	CLSTR	SRCH	CFG
1	Fail to create a VM (36)	13	0	0	0	0	13
2	Fail to start virtual router (4)	10	10	10	10	0	10
3	Fail to upload a template (10)	2	0	0	0	2	0
4	Fail to register an ISO (7)	5	0	5	5	5	5
5	Fail to create a user (5)	1	1	0	1	1	1
6	Host disconnected (19)	1	1	0	1	0	1
7	Fail to add a primary storage (8)	1	0	0	0	0	1
8	Fail to create a new volume (17)	17	5	0	5	0	5
9	Fail to add accounts to the project (12)	5	0	0	0	5	0
10	Exception in upgrading virtual router template (1)	1	1	0	0	1	0
	Total	44	12	20	18	19	36

[a] # of log messages: the number of log messages in an anomaly.
[b] ACT: the number of actual anomalies labeled by experts (ground truth).

Table 6
CloudStack detection results.

Measures	PCA	SAMP	CLSTR	SRCH	CFG
TP	12	20	18	14	36
FP	0	37	61	12	20
FN	32	24	26	0	8
Precision	1.0000	0.3509	0.2278	0.5385	0.6429
Recall	0.2727	0.4545	0.4091	1.0000	0.8182
f-measure	0.4286	0.3960	0.2927	0.7000	0.7200

In terms of recall, SRCH outperforms all other four algorithms, followed by CFG, SAMP, CLSTR, and finally PCA. The difference for the SAMP and the CLSTR algorithm is not significantly different and all other differences are. The reason why SRCH obtains the best result is that human administrator didn't make any mistake in treating normal log messages as anomalous trace in the experiment.

In terms of f-measure, the order is CFG, SRCH, PCA, SAMP and finally CLSTR. The the difference between CFG and SRCH, and PCA and SAMP is not significative, all other are. CFG achieves 67.99% improvement over PCA, 81.82% improvement over SAMP, 145.99% improvement over CLSTR, and 2.86% improvement over SRCH. We can conclude from Table 6 that CFG achieves stable and significant improvements compared with the other four algorithms.

It is worth noting that our algorithm does report 20 false positives, and we submit them to the CloudStack experts for manual inspection. Upon closer examination, they tell us that 15 of 20 operations are initializing operations, which are indeed *rare* situations compared to the other normal operations, but are *normal* by the system design. Eliminating this type of "rare but normal" false positive requires domain expert knowledge. As a future direction, we are combining our approach with the semi-supervised machine learning techniques that can take feedback from human experts and further improve our results.

Another interesting observation is that CFG algorithm outperforms in some categories, such as "Fail to create a VM (Cat.1)", "Fail to add a primary storage (Cat.7)", and "Fail to create a new volume (Cat.8)", but performs as bad as others in "Fail to upload a template (Cat.3)" and "Fail to add accounts to the project (Cat.9)". After examining the intermediate and final results of CFG algorithm, we find that CFG can obtain more promising results when detecting *contextual anomalies* (Chandola et al., 2009) (Cat.7) and *collective anomalies* (Chandola et al., 2009) (Cat.1 and Cat.8), this is because our *reachability graph* can model the complicated control flows buried in log messages accurately. Unfortunately, CFG is sensitive to the content and quality of log messages, and performs poorly when missing part of traces messages in log files (e.g. some log messages are missing in Cat.3) or blending the log messages with unexpected messages (e.g. Java exception outputs are mixed in Cat.9).

Finally, we observe that SRCH has a good performance in detecting *point anomalies* (Chandola et al., 2009) (Cat.10), and is of strong fault tolerance (Cat.3 and Cat.9). However, it shows poor performance in catching the long-periodic and complex-structure anomalies (Cat.1, 6, and 8). We can find the obvious complementarity of SRCH and CFG methods from Table 6, which makes our CFG algorithm a good and practical choice to aid manual anomaly detection process.

7.5. Hadoop experiment results

In this experiment, we collect over 59 million lines of logs from a Hadoop HDFS production system for big data analytics. The system is

Table 7
Hadoop detection results.

Measures	PCA	SAMP	CLSTR	SRCH	CFG
TP	121	176	188	34	324
FP	200	4502	8405	33	214
FN	223	168	156	12	20
Precision	0.3769	0.0376	0.0219	0.5075	0.6022
Recall	0.3517	0.5116	0.5465	0.7391	0.9419
f-measure	0.3639	0.0701	0.0421	0.6018	0.7347

equipped with an customized Hadoop-2.3.0-cdh5.1.2 distribution, and is deployed on a cluster with 200 virtual nodes (in 20 physical rack-mounted servers). We turn on the *INFO* level logging and run the experiment for 10 days. The typical tasks of this system are file operations and different types of map-reduce jobs.

Because of the enormous number of log messages, it is hard to label the whole dataset manually directly and obtain the ground truth. So we first apply five algorithms to the dataset one after another, and then ask human experts to check the correctness of the results. Table 7 summarizes the results of this experiment. Generally speaking, our method is superior to PCA, SAMP, CLSTR and SRCH approaches in precision, recall and f-measure.

The precision of our method is nearly double that of PCA based algorithm, and is of 18.67% improvement over SRCH. This is because our approach finds out the most anomalies again compared with other algorithms and produces almost as many false positives as PCA based algorithm. Although SRCH generates as less as 33 false positives, it finds out only 34 true positives. Such results indicate that keyword search based approach suffers from efficiency problem and the quality of results declines rapidly, because manually analyzing a large collection of log messages is experienced as a cumbersome, labor intensive, and error prone task. On the other hand, SAMP and CLSTR produce so many false positives (4502 and 8405), their precision is extremely low.

In terms of recall, CFG is better than all other three algorithms, followed by SRCH, CLSTR, SAMP, and finally PCA. After examining the detection results generated by PCA based algorithm, we find that most of its false negatives are anomalies that occur quite a number of times. For example, two hard disks in Hadoop *Datenodes* are down for some reason, and lose all data blocks located in them. The *Namenode* keeps finding the data blocks by scanning their indexes and generate tens of thousands of log messages. The PCA based algorithm fail to classify these messages as positives because they follow a very similar pattern and repeat so many times. Our approach does not suffer from such problems, because all log messages are first organized into different traces, and then compared with each other at the global level. This can be implemented easily by attaching each trace to our *reachability graph* prepared in advance.

In terms of f-measure, the order is CFG, SRCH, PCA, SAMP and finally CLSTR. CFG achieves 22.09% performance improvement over SRCH, and 101.90% improvement over PCA. The f-measure values of SAMP and CLSTR are one magnitude less than CFG, SRCH and PCA, and their difference is not significative.

7.6. Overhead of control flow analysis

On drawback of our algorithm is that it needs to parse entire source code to construct the reachability graph of system to reveal the reachable relations for any two log statements. As discussed in 3.3, the time complexity of this process is $O(N^3)$, and the space complexity is $O(N^2)$. In CloudStack experiments, it takes 61 minutes to generate the graph, and 2.8 GB of memory to store the graph; and for Hadoop experiments, the numbers are 69 min and 3.5 GB of memory, respectively. Although the time overhead is relatively large comparing with SAMP and CLSTR, this process is totally offline and has nothing to do with log files. For each system, we need only scan the source code once, construct the reachability graph for the system, and store it in the database. Whenever necessary, we can directly load the graph from the database and put it into service immediately.

7.7. Threats to validity

Internal validity: To increase internal validity, we performed a controlled benchmark experiment by inviting 20 volunteers to perform many daily operations on the target system for 17 days, and generated 1,008,950 log messages. Such method can avoid misleading effects of specifically selected test cases. In addition, we asked five experts to search for abnormal operations in these log messages independently, and they must have an agreement for each identified anomaly. Such results are reliable and can be treated as the ground truth.

In our experiments, the value of the anomaly threshold $freq_{max}$, which controls the ratio between anomalous and normal traces, is set to 0.1. However, the value of this parameter is domain-specific, and can be set by a domain expert. Nevertheless, in cases where the precise value of this ratio is unknown, using the value suggested by previous studies (Xu et al., 2009a; Bezerra and Wainer, 2013; Lakhina et al., 2004) seems to be a reasonable choice. In addition, we tried multiple values of the parameter $freq_{max}$ in our experiments and observed that the detection results are not sensitive to this parameter choice.

External validity: We aimed at increasing external validity by choosing programs from different domains and having totally different functions. Furthermore, we used programs that are deployed and used in the real-world. Nevertheless, we are aware that the results of our evaluations are not automatically transferable to all programs. In addition to our benchmark program, the strong and exhaustive evaluations (over 10 days of running a production system on a cluster with 20 physical servers) indicate that our approach works in many practical application scenarios.

8. Related work

Logs of software systems contain a wealth of information to help detect system anomaly (Oliner et al., 2012). However, manually inspecting system logs to detect anomalies is unfeasible due to the increasing scale and complexity of distributed systems. Therefore, there is a great demand for automatic anomaly detection techniques based on log analysis, and lots of existing work is proposed in this area. Such work can be classified into five categories: rule-based, workflow-based, time series analysis-based, learning-based, and graph-based anomaly detection.

8.1. Rule-based anomaly detection

Rule-based anomaly detection techniques (Prewett, 2003; Cinque et al., 2013; Hansen and Atkins, 1993; Oprea et al., 2015; Rouillard, 2004; Roy et al., 2015; Yamanishi and Maruyama, 2005; Yen et al., 2013) learn rules that capture the normal behavior of a system. A test instance that is not covered by any such rule is considered as an anomaly (Chandola et al., 2009). For example, Beehive (Yen et al., 2013) identifies potential security threats from logs by unsupervised clustering of data-specific features, and then manually labeling outliers. Oprea et al. (2015) used belief propagation to detect early-stage enterprise infection from DNS logs. PerfAugur (Roy et al., 2015) is designed specifically to find performance problems by mining service logs using specialized features such as predicate combinations. Despite the accuracy, rule-based approaches are limited to specific application scenarios and also require domain expertise.

8.2. Workflow-based anomaly detection

The main idea of workflow-based approaches is to construct program workflow models from execution traces in log files (Lou et al.,

2010a). There are a set of existing research efforts (Ammons et al., 2002; Lo et al., 2009; Cotroneo et al., 2007; Lorenzoli et al., 2008; Walkinshaw and Bogdanov, 2008) on inferring Finite State Automatons (FSA) based workflow models from execution traces for software testing and debugging. These algorithms are mostly extended from the popular *k-Tails* algorithm first proposed by Biermann and Feldman (1972). In Lou et al. (2010a), Lou et al. proposed a three-step algorithm to automatically discover concurrent workflows from interleaved event traces that record system events during system execution. Yu et al. (2016) presented CloudSeer, a lightweight non-intrusive approach for log-based workflow monitoring in cloud infrastructures. CloudSeer first builds an automaton for the workflow of each management task based on normal executions, and then it checks interleaved log sequences against a set of automata for workflow divergences in a streaming manner. Finally, it outputs task-automaton instances representing possible erroneous sequences as hints for further diagnosis. Wu et al. (2017) proposed a data-driven approach to construct a workflow model from system logs without a priori domain knowledge about the system. Chuah et al. (2010) developed FDiag, a diagnostics tool to reconstruct event order and establish correlations among events which indicate the root causes of a given failure from very large syslogs. Despite intense research, it has been shown that workflow offers limited advantage for anomaly detection (Fu et al., 2009; Yu et al., 2016). Instead, a major utility of workflows is to aid system diagnosis (Beschastnikh et al., 2014; Lou et al., 2010a).

8.3. Time series analysis-based anomaly detection

Many existing work regards the entire log as a single sequence of repeating messages and mines it with time series analysis methods. Tsao (1983); Iyer et al. (1986), and Hansen et al. (1988) developed the *tupling* concept to perform trend analysis and fault prediction of system event logs. Yamanishi and Maruyama (2005) modeled syslog sequences as a mixture of Hidden Markov Models (HMM) to find messages that are likely to be related to critical failures. Lin and Siewiorek (1990) proposed a failure prediction heuristic named Dispersion Frame Technique (DFT) to extract both transient and intermittent error processes from system error logs. Bezerra and Wainer (2013) compared three frequent model based algorithms for detecting anomaly logs in process aware systems. However, treating a log as a single time series does not perform well in large-scale software systems with multiple independent processes that generate interleaved logs (Xu et al., 2009c).

8.4. Learning-based anomaly detection

Machine learning is a prevalent method used by many existing work to detect anomaly. According to the type of data involved and the machine learning techniques employed, anomaly detection approaches can be further classified into two sub-categories: *supervised anomaly detection* and *unsupervised anomaly detection* (He et al., 2016).

8.4.1. Supervised anomaly detection

Supervised learning methods first infer a model from labeled training data, and then use this model to discriminate normal and anomalous log traces. Many supervised methods, such as regression, decision tree, support vector machine (SVM), and deep learning, are used in the anomaly detection context (He et al., 2016). For example, (Farshchi et al., 2015) adopted a regression-based analysis technique to find the correlation between an operation's activity logs and the operation activity's effect on cloud resources. The correlation model is then used to derive assertion specifications, which can be used for runtime verification of running operations and their impact on resources. Wang (2005) employed the multinomial logistic regression technique to identify potential risk factors that are statistically significantly associated with anomalous behaviors. Mok et al. (2010) proposed a random effects logistic regression model to predict anomaly detection. Unlike the previous studies on anomaly detection, a random effects model was applied, which accommodates not only the risk factors of the exposures but also the uncertainty not explained by such factors.

Decision tree was first applied to failure diagnosis for web request log system in Chen et al. (2004). Stein et al. (2005) used a genetic algorithm to select a subset of input features for decision tree classifiers, with a goal of increasing the detection rate and decreasing the false alarm rate in network intrusion detection. Fu et al. (2013) first constructed a concept lattice graph to mine execution patterns and to model relationships among different execution patterns, and then proposed a decision-tree based technique to learn branch conditions that can determine which code branches the system will take at bifurcation points. Amor et al. (2004) offered an experimental study of the use of naive Bayes and decision tree in intrusion detection, and showed that even if having a simple structure, naive Bayes provide very competitive results.

In anomaly detection via SVM, if a new instance is located above the hyperplane, it would be reported as an anomaly, while marked as normal otherwise. Liang et al. (2007) collected event logs over an extensive period from IBM BlueGene/L, and developed a SVM-based prediction model to detect system failures and compared it with other methods. Steinwart et al. (2005) proposed a modified SVM model for anomaly detection, and reported more promising experimental results compared with other commonly used SVM models. Li et al. (2003) proposed an one-class SVM based approach for anomaly detection using abstracted user audit logs. Shon et al. (2005) introduced a internet anomaly detection framework combining genetic algorithm (GA) and SVM, where GA is used for feature selection and SVM is used for classification. Perdisci et al. (2006) proposed new feature extraction method, and use an ensemble of one-class SVM classifiers to detect network anomalies.

Deep learning is a class of supervised machine learning algorithms that use a cascade of multiple layers of nonlinear processing units for feature extraction and transformation (Deng et al., 2014). Inspired by the observation that entries in a system log are a sequence of events produced by the execution of structured source code, (Du et al., 2017) designed the DeepLog framework using a Long Short-Term Memory (LSTM) neural network for online anomaly detection over system logs. Zhang et al. (2016) proposed a feature representation of coding logs during each time epoch in a low-dimensional vector space, and designed a LSTM-based anomaly prediction system.

Besides, many hybrid methods are also proposed. Muniyandi et al. (2012) introduced an anomaly detection method cascading k-Means clustering and the C4.5 decision tree methods for classifying anomalous and normal activities in a computer network. Farid et al. (2010) presented a new hybrid learning algorithm for adaptive network intrusion detection using naive Bayesian classifier and ID3 algorithm, which analyzes the large volume of network data and considers the complex properties of attack behaviours to improve the performance of detection speed and detection accuracy.

Supervised methods usually achieve high precision, while the recall varies over different datasets and window settings (He et al., 2016). Their main downside is that unknown anomalies not in training data may not be detected. Furthermore, anomalous data are hard to obtain for training (Du et al., 2017).

8.5. Unsupervised anomaly detection

Unlike supervised methods, unsupervised methods derive a model to describe hidden structure from unlabeled data. Unsupervised methods are more applicable in real-world production environment due to the lack of labels (He et al., 2016). Common unsupervised approaches include various clustering methods, principal component analysis (PCA), association rule mining and etc. Lin et al. (2016b) designed a two-phases and clustering-based method called LogCluster to

identify online system problems. Dickinson et al. (2001) experimentally evaluated the effectiveness of using cluster analysis of execution profiles to find failures among the executions induced by a set of potential test cases. Chen et al. (2002) proposed Pinpoint to support automatic problem determination. Pinpoint traces requests as they travel through a system, detects component failures internally and end-to-end failures externally, and performs data clustering analysis over a large number of requests to determine the combinations of components that are likely to be the cause of failures. Lim et al. (2008) discussed their experiences with applying clustering based log mining techniques to characterize the behavior of large enterprise telephony systems. Meng et al. (2017) proposed DriftInsight, a practical behavior anomaly detection system for large-scale clouds in production. DriftInsight adopts an unsupervised clustering algorithm named DBSCAN Ester et al. (1996) to aggregate multidimensional metrics into a one-dimension component state metric while keep semantics of these metrics for problem diagnosis. Vaarandi (2003) presented a density based clustering algorithm for log files to build log file profiles, to find frequent special patterns from log files, and to detect anomalous log file lines. They proposed LogCluster (Vaarandi and Pihelgas, 2015) in their latest work, which implements data clustering and line pattern mining for textual event logs. Makanju et al. (2009) presented the IPLoM (Iterative Partitioning Log Mining) algorithm for the mining of clusters from event logs. Shang et al. (2013) proposed an approach for detecting potential deployment failures/anomalies of big data analytics applications after deployment. The approach abstracts the platform's execution logs from both the small (pseudo) and large scale cloud deployments, groups the related abstracted log lines into execution sequences for both deployments, then examines and reports the differences between the two clusters of execution sequences. Yasami and Mozaffari (2010) introduced a host-based combinatorial method based on k-Means clustering and ID3 decision tree learning algorithms for unsupervised classification of anomalous and normal activities in computer network ARP traffic.

PCA is a statistical method that projects high-dimension data to a new coordinate system composed of k principal components, where k is set to be less than the original dimension. Xu et al. proposed a PCA based method to mine log files to automatically detect system runtime problems in a series of articles (Xu et al., 2008; 2009c; 2009b; 2009a; Xu, 2010). In their method, system logs are parsed by combining source code analysis with information retrieval to create composite features, then these feature are analyzed using PCA to detect operational problems.

Program invariants (Ernst et al., 2001) are the linear relationships that always hold during system running even with various inputs and under different workloads. Intuitively, invariants mining could uncover the linear relationships between multiple log events that represent system normal execution behaviors. If such linear relationships are violated in a log file consisting of many log events, it will be marked abnormal. Lou et al. (2010b) proposed an algorithm to automatically discover program invariants in logs. If the message count vector of a message group extracted from a log file violates any one of its related invariants, it is considered as an anomaly immediately. Hangal and Lam (2002) introduced DIDUCE, an automatic anomaly detector, to track down software bugs. DIDUCE first formulates hypotheses of program invariants by instrumenting a program and observing its behavior as it runs, and then reports violations to help users to catch software bugs as soon as they occur. Abreu et al. (2008) studied the utility of generic invariants in their capacity of error detectors within a spectrum-based fault localization (SFL) approach aimed to diagnose program defects in the operational phase. Sahoo et al. (2008) present iSWAT, a system that uses invariants to improve the coverage and latency of existing hardware error detection techniques. The basic idea of iSWAT is to use training inputs to create likely invariants based on value ranges of selected program variables and then use them to identify faults at runtime. Lin et al. (2016a) propose iDice, an automated algorithm that helps support engineers identify the effective combinations that are associated with emerging issues. They formulated the problem of identifying emerging issues as a pattern mining problem and designed several pruning techniques to significantly reduce the search space. Vaarandi (2004) presented a breadth-first frequent itemset mining algorithm for mining frequent patterns from event logs. The algorithm combines the features of well-known breadth-first and depth-first algorithms, and also takes into account the special properties of event log data.

Unsupervised methods generally achieve inferior performance against supervised methods (Du et al., 2017; He et al., 2016). Among them, invariants mining manifests as a promising model with stable, high performance according to the experiment results in He et al. (2016), but the methods of invariants mining and clustering are less efficient and need further optimizations for speedup.

8.6. Graph-based anomaly detection

Most relevant to our work is graph-based anomaly detection. Typically, one first extracts template sequence at first, and then generates a graph-based model to compare with log sequences in production systems to detect conflicts. For instance, Fu et al. (2009) converted free form text messages in existing log files to log keys first, and learned a Finite State Automaton (FSA) from training log sequences to model the normal work flow for each system component. With these learned models, they can automatically detect anomalies in newly input log files. Babenko et al. (2009) presented a technique, called Automata Violation Analyzer (AVA), that automatically analyzes anomalous events identified by dynamic analysis techniques that infer FSA. AVA produces multiple interpretations of the detected anomalies, prioritizes them according to likelihood to explain differences between correct and failing executions, and presents the final result to testers. Zhao et al. (2014) introduced lprof, a profiling tool that automatically reconstructs the execution flow of each request in a distributed application. lprof infers the request-flow entirely from runtime logs and thus does not require any modifications to source code. Tak et al. (2016) presented LOGAN, a log analytics tool for problem diagnosis in cloud platform. LOGAN uses graph-based templates and modules to support log correlation analysis and log comparison, and is able to provide diagnosis capability beyond common search based method. Jiang et al. (2008) developed an approach that recognizes the internal structure of log lines. The approach uses graph-based clone detection techniques to abstract each log line to its corresponding execution event. Jia et al. (2017a) proposed an approach for automatic anomaly detection based on logs. They first mine a hybrid graph model that captures normal execution flows inter and intra services, and then raise anomaly alerts on observing deviations from the hybrid model. In their another related work (Jia et al., 2017b), they proposed LogSed, an approach of diagnosing anomalous run-time behaviors in cloud systems from execution logs. LogSed mines 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.

Graph-based model has three advantages (Jia et al., 2017b): (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.

9. Conclusions and future work

We propose a general approach to execution anomaly detection for large scale software systems via the analysis of their console logs. We first give some related terminology and a formal definition of execution anomaly detection problem. Using source code as a reference to understand the possible logical relationships among log messages, we are

able to segment logs accurately. The accuracy in log segmentation allows us to extract the complete execution traces containing different types of log messages from log files, which are proven to be very important (Jiang et al., 2009) yet are usually ignored due to the difficulties in log segmentation. By treating each trace as a data sequence segment, we propose a novel probabilistic suffix tree based statistical method that can detect unusual patterns or anomalies effectively. Experiments on a CloudStack cluster and a Hadoop production system show the superiority of our method in comparison with existing three popular detection algorithms. We plan to explore many possible directions in our future research, including: (1) developing online detection algorithms instead of current postmortem analysis; and (2) introducing machine learning approaches to automatically distinguish "rare but normal" traces and real execution anomalies.

Acknowledgments

The authors would like to thank Liyong Zhang for his great suggestions on the early draft of the paper. Our thanks also go to the reviewers for their hard work and feedbacks. This work is supported by the National Natural Science Foundation of China (Grant Nos. 61202040 and 71201121) and the Fundamental Research Funds for the Central Universities (Grant No. JB171005).

References

Abreu, R., González, A., Zoeteweij, P., van Gemund, A.J., 2008. Automatic software fault localization using generic program invariants. Proceedings of the 2008 ACM symposium on Applied computing. ACM, pp. 712–717.
Allen, F.E., 1970. Control flow analysis. ACM Sigplan Notices. Vol. 5. ACM, pp. 1–19.
Amazon elastic compute cloud, 2018. http://aws.amazon.com/ec2.
Ammons, G., Bodík, R., Larus, J.R., 2002. Mining specifications. ACM Sigplan Notices 37 (1), 4–16.
Amor, N.B., Benferhat, S., Elouedi, Z., 2004. Naive bayes vs decision trees in intrusion detection systems. Proceedings of the 2004 ACM Symposium on Applied Computing. ACM, pp. 420–424.
Apache cloudstack, 2018. https://cloudstack.apache.org (Accessed on February 25th 2018).
Apache hadoop, 2018. http://hadoop.apache.org (Accessed on February 25th 2018).
Apache log4j, 2018. http://logging.apache.org/log4j (Accessed on February 25th 2018).
Babenko, A., Mariani, L., Pastore, F., 2009. Ava: automated interpretation of dynamically detected anomalies. Proceedings of the Eighteenth International Symposium on Software Testing and Analysis. ACM, pp. 237–248.
Beschastnikh, I., Brun, Y., Ernst, M.D., Krishnamurthy, A., 2014. Inferring models of concurrent systems from logs of their behavior with csight. Proceedings of the 36th International Conference on Software Engineering. ACM, pp. 468–479.
Bezerra, F., Wainer, J., 2013. Algorithms for anomaly detection of traces in logs of process aware information systems. Inf. Syst. 38 (1), 33–44.
Biermann, A.W., Feldman, J.A., 1972. On the synthesis of finite-state machines from samples of their behavior. IEEE Trans. Comput. 100 (6), 592–597.
Chandola, V., Banerjee, A., Kumar, V., 2009. Anomaly detection: a survey. ACM Comput. Surv. (CSUR) 41 (3), 15.
Chen, M., Zheng, A.X., Lloyd, J., Jordan, M.I., Brewer, E., 2004. Failure diagnosis using decision trees. Proceedings of the International Conference on Autonomic Computing. IEEE, pp. 36–43.
Chen, M.Y., Kiciman, E., Fratkin, E., Fox, A., Brewer, E., 2002. Pinpoint: problem determination in large, dynamic internet services. Proceedings. International Conference on Dependable Systems and Networks, 2002. DSN 2002. IEEE, pp. 595–604.
Chuah, E., Kuo, S.-h., Hiew, P., Tjhi, W.-C., Lee, G., Hammond, J., Michalewicz, M.T., Hung, T., Browne, J.C., 2010. Diagnosing the root-causes of failures from cluster log files. Proceedings of the International Conference on High Performance Computing (HiPC). IEEE, pp. 1–10.
Cinque, M., Cotroneo, D., Pecchia, A., 2013. Event logs for the analysis of software failures: a rule-based approach. IEEE Trans. Softw. Eng. 39 (6), 806–821.
Cotroneo, D., Pietrantuono, R., Mariani, L., Pastore, F., 2007. Investigation of failure causes in workload-driven reliability testing. Proceedings of the Fourth International Workshop on Software Quality Assurance: in Conjunction with the 6th ESEC/FSE Joint Meeting. ACM, pp. 78–85.
Deng, L., Yu, D., et al., 2014. Deep learning: methods and applications. Found. Trends® Signal Process 7 (3–4), 197–387.
Dickinson, W., Leon, D., Podgurski, A., 2001. Finding failures by cluster analysis of execution profiles. Proceedings of the 23rd International Conference on Software Engineering. IEEE Computer Society, pp. 339–348.
Du, M., Li, F., Zheng, G., Srikumar, V., 2017. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. ACM, pp. 1285–1298.
Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D., 2001. Dynamically discovering likely program invariants to support program evolution. IEEE Trans. Softw. Eng. 27 (2), 99–123.
Ester, M., Kriegel, H.-P., Sander, J., Xu, X., et al., 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. Proceedings of the Kdd. 96. pp. 226–231.
Farid, D.M., Harbi, N., Rahman, M.Z., 2010Farid, Harbi, Rahman. Combining naive Bayes and decision tree for adaptive intrusion detection. International Journal of Network Security & Its Applications 12–25.
Farshchi, M., Schneider, J.-G., Weber, I., Grundy, J., 2015. Experience report: anomaly detection of cloud application operations using log and cloud metric correlation analysis. Proceedings of the IEEE 26th International Symposium on Software Reliability Engineering (ISSRE). IEEE, pp. 24–34.
Floyd, R.W., 1962. Algorithm 97: shortest path. Commun. ACM 5 (6), 345.
Fu, Q., Lou, J.-G., Lin, Q., Ding, R., Zhang, D., Xie, T., 2013. Contextual analysis of program logs for understanding system behaviors. Proceedings of the 10th Working Conference on Mining Software Repositories. IEEE Press, pp. 397–400.
Fu, Q., Lou, J.-G., Wang, Y., Li, J., 2009. Execution anomaly detection in distributed systems through unstructured log analysis. Proceedings of the Ninth IEEE International Conference on Data Mining. ICDM'09. IEEE, pp. 149–158.
Google glog, 2018. https://github.com/google/glog (Accessed on February 25th 2018).
Hangal, S., Lam, M.S., 2002. Tracking down software bugs using automatic anomaly detection. Proceedings of the 24rd International Conference on Software Engineering. ICSE 2002. IEEE, pp. 291–301.
Hansen, J. P., et al., 1988. Trend analysis and modeling of uni/multi-processor event logs. CMU Center for Dependable Systems Technical Report, 1–89.
Hansen, S.E., Atkins, E.T., 1993. Automated system monitoring and notification with swatch. Proceedings of the LISA. 93. pp. 145–152.
He, S., Zhu, J., He, P., Lyu, M.R., 2016. Experience report: system log analysis for anomaly detection. Proceedings of the IEEE 27th International Symposium on Software Reliability Engineering (ISSRE). IEEE, pp. 207–218.
Iyer, R., Young, L., Sridhar, V., 1986. Recognition of error symptoms in large systems. Proceedings of 1986 ACM Fall Joint Computer Conference. IEEE Computer Society Press, pp. 797–806.
Jia, T., Chen, P., Yang, L., Li, Y., Meng, F., Xu, J., 2017a. An approach for anomaly diagnosis based on hybrid graph model with logs for distributed services. Proceedings of the IEEE International Conference on Web Services (ICWS). IEEE, pp. 25–32.
Jia, T., Yang, L., Chen, P., Li, Y., Meng, F., Xu, J., 2017b. Logsed: anomaly diagnosis through mining time-weighted control flow graph in logs. Proceedings of the IEEE 10th International Conference on Cloud Computing (CLOUD). IEEE, pp. 447–455.
Jiang, W., Hu, C., Pasupathy, S., Kanevsky, A., Li, Z., Zhou, Y., 2009. Understanding customer problem troubleshooting from storage system logs. Proceedings of the USENIX FAST. Vol. 9. pp. 43–56.
Jiang, Z.M., Hassan, A.E., Hamann, G., Flora, P., 2008. An automated approach for abstracting execution logs to execution events. J. Softw. Evol. Process 20 (4), 249–267.
Lakhina, A., Crovella, M., Diot, C., 2004. Diagnosing network-wide traffic anomalies. ACM SIGCOMM Computer Communication Review. Vol. 34. ACM, pp. 219–230.
Li, K.-L., Huang, H.-K., Tian, S.-F., Xu, W., 2003. Improving one-class SVM for anomaly detection. Proceedings of the International Conference on Machine Learning and Cybernetics. Vol. 5. IEEE, pp. 3077–3081.
Liang, Y., Zhang, Y., Xiong, H., Sahoo, R., 2007. Failure prediction in ibm bluegene/l event logs. Proceedings of the Seventh IEEE International Conference on Data Mining, 2007. ICDM 2007. IEEE, pp. 583–588.
Lim, C., Singh, N., Yajnik, S., 2008. A log mining approach to failure analysis of enterprise telephony systems. Proceedings of the IEEE International Conference on Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE, pp. 398–403.
Lin, Q., Lou, J.-G., Zhang, H., Zhang, D., 2016a. idice: problem identification for emerging issues. Proceedings of the IEEE/ACM 38th International Conference on Software Engineering (ICSE). IEEE, pp. 214–224.
Lin, Q., Zhang, H., Lou, J.-G., Zhang, Y., Chen, X., 2016b. Log clustering based problem identification for online service systems. Proceedings of the 38th International Conference on Software Engineering Companion. ACM, pp. 102–111.
Lin, T.Y., Siewiorek, D.P., 1990. Error log analysis: statistical modeling and heuristic trend analysis. IEEE Trans. Reliab. 39 (4), 419–432.
Lo, D., Mariani, L., Pezzè, M., 2009. Automatic steering of behavioral model inference. Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering. ACM, pp. 345–354.
Lorenzoli, D., Mariani, L., Pezzè, M., 2008. Automatic generation of software behavioral models. Proceedings of the 30th International Conference on Software Engineering. ACM, pp. 501–510.
Lou, J.-G., Fu, Q., Yang, S., Li, J., Wu, B., 2010a. Mining program workflow from interleaved traces. Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM, pp. 613–622.
Lou, J.-G., Fu, Q., Yang, S., Xu, Y., Li, J., 2010b. Mining invariants from console logs for system problem detection. Proceedings of the USENIX Annual Technical Conference.
Makanju, A.A., Zincir-Heywood, A.N., Milios, E.E., 2009. Clustering event logs using iterative partitioning. Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM, pp. 1255–1264.
Mariani, L., Pastore, F., 2008. Automated identification of failure causes in system logs. Proceedings of the 19th International Symposium on Software Reliability Engineering, 2008. ISSRE 2008. IEEE, pp. 117–126.
McCandless, M., Hatcher, E., Gospodnetic, O., 2010. Lucene in Action: Covers Apache Lucene 3.0. Manning Publications Co.
Meng, F.J., Zhang, X., Chen, P., Xu, J.M., 2017. Driftinsight: detecting anomalous behaviors in large-scale cloud platform. Proceedings of the IEEE 10th International Conference on Cloud Computing (CLOUD). IEEE, pp. 230–237.

Mok, M.S., Sohn, S.Y., Ju, Y.H., 2010. Random effects logistic regression model for anomaly detection. Expert Syst. Appl. 37 (10), 7162–7166.
Muniyandi, A.P., Rajeswari, R., Rajaram, R., 2012. Network anomaly detection by cascading k-means clustering and c4.5 decision tree algorithm. Procedia Eng. 30, 174–182.
Oliner, A., Ganapathi, A., Xu, W., 2012. Advances and challenges in log analysis. Commun. ACM 55 (2), 55–61.
Oliner, A., Stearley, J., 2007. What supercomputers say: a study of five system logs. Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2007. DSN'07. IEEE, pp. 575–584.
Oprea, A., Li, Z., Yen, T.-F., Chin, S.H., Alrwais, S., 2015. Detection of early-stage enterprise infection by mining large-scale log data. Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). IEEE, pp. 45–56.
Perdisci, R., Gu, G., Lee, W., 2006. Using an ensemble of one-class svm classifiers to harden payload-based anomaly detection systems. Proceedings of the Sixth International Conference on Data Mining, 2006. ICDM'06. IEEE, pp. 488–498.
Prewett, J.E., 2003. Analyzing cluster log files using logsurfer. Proceedings of the 4th Annual Conference on Linux Clusters. Citeseer.
Rouillard, J.P., 2004. Real-time log file analysis using the simple event correlator (sec). Proceedings of the LISA. Vol. 4. pp. 133–150.
Roy, S., König, A.C., Dvorkin, I., Kumar, M., 2015. Perfaugur: Robust diagnostics for performance anomalies in cloud services. Proceedings of the IEEE 31st International Conference on Data Engineering (ICDE). IEEE, pp. 1167–1178.
Sahoo, S.K., Li, M.-L., Ramachandran, P., Adve, S.V., Adve, V.S., Zhou, Y., 2008. Using likely program invariants to detect hardware errors. Proceedings of the IEEE International Conference on Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE, pp. 70–79.
Shang, W., Jiang, Z.M., Hemmati, H., Adams, B., Hassan, A.E., Martin, P., 2013. Assisting developers of big data analytics applications when deploying on hadoop clouds. Proceedings of the 2013 International Conference on Software Engineering. IEEE Press, pp. 402–411.
Shon, T., Kim, Y., Lee, C., Moon, J., 2005. A machine learning framework for network anomaly detection using SVM and ga. Proceedings from the Sixth Annual IEEE SMC Information Assurance Workshop, 2005. IAW'05. IEEE, pp. 176–183.
Splunk, 2018. http://www.splunk.com (Accessed on February 2018).
Stein, G., Chen, B., Wu, A.S., Hua, K.A., 2005. Decision tree classifier for network intrusion detection with ga-based feature selection. Proceedings of the 43rd Annual Southeast Regional Conference-Volume 2. ACM, pp. 136–141.
Steinwart, I., Hush, D., Scovel, C., 2005. A classification framework for anomaly detection. J. Mach. Learn. Res. 6 (Feb), 211–232.
Tak, B.C., Tao, S., Yang, L., Zhu, C., Ruan, Y., 2016. Logan: problem diagnosis in the cloud using log-based reference models. Proceedings of the IEEE International Conference on Cloud Engineering (IC2E). IEEE, pp. 62–67.
Tan, J., Pan, X., Kavulya, S., Gandhi, R., Narasimhan, P., 2009. Mochi: visual log-analysis based tools for debugging hadoop. HotCloud.
Tsao, M.M., 1983. Trend analysis and fault prediction. Department of Computer Science, Carnegie-Mellon University Ph.D. thesis.
Vaarandi, R., 2003. A data clustering algorithm for mining patterns from event logs. Proceedings of the 3rd IEEE Workshop on IP Operations & Management, 2003. (IPOM 2003). IEEE, pp. 119–126.
Vaarandi, R., 2004. A breadth-first algorithm for mining frequent patterns from event logs. Intelligence in Communication Systems. Springer, pp. 293–308.
Vaarandi, R., Pihelgas, M., 2015. Logcluster-a data clustering and pattern mining algorithm for event logs. Proceedings of the 11th International Conference on Network and Service Management (CNSM). IEEE, pp. 1–7.
Walkinshaw, N., Bogdanov, K., 2008. Inferring finite-state models with temporal constraints. Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering. IEEE Computer Society, pp. 248–257.
Wang, Y., 2005. A multinomial logistic regression modeling approach for anomaly intrusion detection. Comput. Secur. 24 (8), 662–674.
Wu, F., Anchuri, P., Li, Z., 2017. Structural event detection from log messages. Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM, pp. 1175–1184.
Xu, W., 2010. System problem detection by mining console logs. University of California, Berkeley Ph.D. thesis.
Xu, W., Huang, L., Fox, A., Patterson, D., Jordan, M.I., 2009a. Largescale system problem detection by mining console logs. Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP) ACM, pp. 117–132.
Xu, W., Huang, L., Fox, A., Patterson, D., Jordan, M., 2009b. Online system problem detection by mining patterns of console logs. Proceedings of the Ninth IEEE International Conference on Data Mining, 2009. ICDM'09. IEEE, pp. 588–597.
Xu, W., Huang, L., Fox, A., Patterson, D., Jordan, M.I., 2009c. Detecting large-scale system problems by mining console logs. Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles. ACM, pp. 117–132.
Xu, W., Huang, L., Fox, A., Patterson, D.A., Jordan, M.I., 2008. Mining console logs for large-scale system problem detection. SysML 8, 4.
Yamanishi, K., Maruyama, Y., 2005. Dynamic syslog mining for network failure monitoring. Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining. ACM, pp. 499–508.
Yang, J., Wang, W., 2003. Cluseq: efficient and effective sequence clustering. Proceedings of the 19th International Conference on Data Engineering. IEEE, pp. 101–112.
Yasami, Y., Mozaffari, S.P., 2010. A novel unsupervised classification approach for network anomaly detection by k-means clustering and id3 decision tree learning methods. J. Supercomput. 53 (1), 231–245.
Yen, T.-F., Oprea, A., Onarlioglu, K., Leetham, T., Robertson, W., Juels, A., Kirda, E., 2013. Beehive: large-scale log analysis for detecting suspicious activity in enterprise networks. Proceedings of the 29th Annual Computer Security Applications Conference. ACM, pp. 199–208.
Yu, X., Joshi, P., Xu, J., Jin, G., Zhang, H., Jiang, G., 2016. Cloudseer: Workflow monitoring of cloud infrastructures via interleaved logs. ACM SIGPLAN Notices. Vol. 51. ACM, pp. 489–502.
Zhang, K., Xu, J., Min, M.R., Jiang, G., Pelechrinis, K., Zhang, H., 2016. Automated it system failure prediction: a deep learning approach. Proceedings of the IEEE International Conference on Big Data (Big Data). IEEE, pp. 1291–1300.
Zhao, X., Zhang, Y., Lion, D., Ullah, M.F., Luo, Y., Yuan, D., Stumm, M., 2014. lprof: A non-intrusive request flow profiler for distributed systems. Proceedings of the OSDI. Vol. 14. pp. 629–644.

Liang Bao is currently an Associate Professor in the School of Software at Xidian University. His research interests include big data, cloud computing and software engineering. His research in computing develops machine learning based solutions to predict the execution of and optimize the performance of big data analytics frameworks in public cloud computing environments. His research in cloud computing and big data develops a big data analytics framework to help users to find implicit patterns or rules from large collections of IoT datasets. Dr. Bao's work has been supported by various funding agencies, including the National Science Foundation of China, the Ministry of Industry and Information of China, the Ministry of Science and Technology of China, and many companies and research institutes. He has published over 30 research articles in highly reputed conference proceedings and journals.

Qian Li (Ph.D.) is an associate professor of the school of economics and finance, Xi'an Jiaotong University. Her primary research interests include data analytics, behavioral finance, corporate finance and portfolio management. Dr. Li's work has been supported by various funding agencies, including the National Science Foundation of China, the Ministry of Education of Humanities and Social Science project of China, and many companies and research institutes. She has published over 20 research articles in highly reputed conference proceedings and journals.