Mining Invariants from Console Logs for System Problem Detection

Jian-Guang LOU[1] Qiang FU[1] Shengqi YANG[2] Ye XU[3] Jiang LI[1]

[1]Microsoft Research Asia
Beijing, P. R. China
{jlou, qifu, jiangli}@microsoft.com

[2]Dept. of Computer Science
Beijing Univ. of Posts and Telecom
v-sheyan@microsoft.com

[3]Dept. of Computer Science
Nanjing University, P.R. China
v-yexu@microsoft.com

Abstract

Detecting execution anomalies is very important to the maintenance and monitoring of large-scale distributed systems. People often use console logs that are produced by distributed systems for troubleshooting and problem diagnosis. However, manually inspecting console logs for the detection of anomalies is unfeasible due to the increasing scale and complexity of distributed systems. Therefore, there is great demand for automatic anomaly detection techniques based on log analysis. In this paper, we propose an unstructured log analysis technique for anomaly detection, with a novel algorithm to automatically discover program invariants in logs. At first, a log parser is used to convert the unstructured logs to structured logs. Then, the structured log messages are further grouped to log message groups according to the relationship among log parameters. After that, the program invariants are automatically mined from the log message groups. The mined invariants can reveal the inherent linear characteristics of program work flows. With these learned invariants, our technique can automatically detect anomalies in logs. Experiments on Hadoop show that the technique can effectively detect execution anomalies. Compared with the state of art, our approach can not only detect numerous real problems with high accuracy but also provide intuitive insight into the problems.

1 Introduction

Most software systems generate console log messages for troubleshooting. The console log messages are usually unstructured free-form text strings, which are used to record events or states of interest and to capture the system developers' intent. In general, when a job fails, experienced system operators examine recorded log files to gain insight about the failure, and to find the potential root causes. Especially for debugging distributed systems, checking the console logs to locate system problems is the most applicable way because the instrumentation or dump based approaches may make a system behave differently from its daily execution and introduce overhead.

We are now facing an explosive growth of large-scale Internet services that are supported by a set of large server clusters. The trend of cloud computing also drives the deployment of large-scale data centers. Typical systems such as those of Google, Amazon and Microsoft consist of thousands of distributed components including servers, network devices, distributed computing software components, and operating systems. Due to the increasing scale and complexity of these distributed systems, it becomes very time consuming for a human operator to diagnose system problems through manually examining a great amount of log messages. Therefore, automated tools for problem diagnosis through log analysis are essential for many distributed systems.

Several research efforts have been made in the design and development of automatic tools for log analysis. Most of the traditional automatic tools detect system problems by checking logs against a set of rules that describe normal system behaviors. Such rules are manually predefined by experts according to their knowledge about system design and implementation. SEC [1], Logsurfer [2] and Swatch [3] are three typical examples of a rule-based log analysis tool. However, it is very expensive to manually define such rules because a great amount of system experts' efforts are required. Besides, a modern system often consists of multiple components developed by different groups or even different companies, and a single expert may not have complete knowledge of the system; therefore, constructing the rules needs close cooperation of multiple experts, which brings more difficulties and costs. Furthermore, after each upgrade of the system, the experts need to check or modify the predefined rules again. In summary, manually defining rules for detecting problems from logs is expensive and inefficient.

Recently, there have appeared some statistic learning based automatic tools that analyze console logs, profiles and measurements for system monitoring and trouble shooting. Such approaches extract features from logs, traces or profiles, then use statistical techniques, such as subspace analysis [5, 6, 13], clustering and classification algorithms [7, 8], to automatically build models, and then identify failures or problems according to the learned models. However, most of the learned models are black box models that cannot be easily understood by human operators [5]. They may detect anomalies in a high dimensional feature space, but can hardly provide intuitive and meaningful explanations for the detected anomalies.

In this paper, we aim to automatically mine constant linear relationships from console logs based on a statistical learning technique. Such relationships that always hold in system logs under different inputs and workloads are considered as program invariants. These linear relationships can capture the normal program execution behavior. If a new log breaks certain invariants, we say an anomaly occurs during the system execution. Here is a simple example of invariant: in the normal executions of a system, the number of log messages indicating "Open file" is usually equal to the number of log messages corresponding to "Close file", because each opened file will be closed at some stage eventually. Such relationships are often well utilized when we manually check problems in log files. If it is broken, the operator can know there must be a system anomaly of the file operation (e.g. file handler leak) and safely speculate where the problem is. With this observation, we propose an approach to automatically detect system anomalies based on mining program invariants from logs. Unlike other statistical based approaches, program invariant has a very clear physical meaning that can be easily understood by human operators. It can not only detect system anomalies but also give a meaningful interpretation for each detected anomaly. Such interpretation associates the anomaly with the execution logic, which can significantly help system operators to diagnose system problems.

In our approach, we first convert unstructured log messages to structured information, including message signatures and parameters, by using a log parser. Then, the messages are grouped based on the log parameters. Based on the message groups, we discover sparse and integer invariants through a hypothesis and testing framework. In addition, the scalability and efficiency issues of the invariant search algorithm are discussed, and some techniques to reduce the computational cost are introduced. In brief, the main contribution of our work can be summarized as follows:

- We propose a method to automatically identify a set of parameters that correspond to the same program variable (namely cogenetic) based on the parameter value range analysis.
- We propose a method to discover sparse and integer invariants that have very clear physical meanings associated with system execution. The computational complexity of our algorithm can be significantly reduced to fit real-world large-scale applications.
- We apply the mined invariants to detect system anomalies. By checking the broken invariants, our method can provide helpful insights about execution problems.

The paper is organized as follows. In Section 2, we briefly introduce the previous work that is closely related to ours. Section 3 provides the basic idea and an overview of our approach. In Section 4, we briefly describe the log parsing method that we have used. In Section 5, we first relate multiple parameters to a program variable, and then group log messages to obtain message count vectors. In Section 6, we mainly present the invariant searching algorithm. We give a simple anomaly detection method in Section 7. Section 8 gives some experimental results on two large scale systems. Finally, we conclude the paper in Section 9.

2 Related Work

Recently, statistical machine learning and data mining techniques have shown great potential in tackling the scale and complexity of the challenges in monitoring and diagnosis of large scale systems. Several learning based approaches have been proposed to detect system failures or problems by statistically analyzing console logs, profiles, or system measurements. For example, Dickenson et al. [7] use classification techniques to group similar log sequences to a set of classes based on some string distance metrics. A human analyst examines one or several profiles from each class to determine whether the class represents an anomaly. Mirgorodskiy et al. [8] also use string distance metrics to categorize function-level traces, and identify outlier traces as anomalies that substantially differ from the others. Yuan et al. [9] first extract n-grams as features of system call sequences, and then use Support Vector Machine (SVM, a supervised classification algorithm) to classify traces based on the similarity of the traces of known problems. Xu et al. [5, 6] preprocess log messages to extract message count vectors as the log features, and detect anomalies using Principal Component Analysis (PCA). From the point of view of a human operator, the above statistical tools build models of a black box style, and they can hardly provide human operators with intuitive insights about abnormal jobs and anomalies. In [5, 6], the authors try to remedy the defect by learning a decision tree and to visualize the detection results. However, the decision tree is still somehow incomprehensible to human operators, because it does not directly relate to execution flow mechanisms. In this paper, we use program invariants to characterize the behavior of a system. Unlike the black box models, program invariants often provide intuitive interpretations of the detected anomalies.

Another set of algorithms [15, 16, 17] use Finite State Automaton (FSA) models to represent log sequences, which is more easily understood by operators. For example, SALSA [15] examines Hadoop logs to construct FSA models of the Datanode module and TaskTracker module. In the work of Cotroneo et al. [16], FSA models are first derived from the traces of Java Virtual Machine. Then, logs of unsuccessful jobs are compared with the inferred FSA models to detect anomalies. In [17], the authors also construct a FSA to characterize the normal system behaviors. A new trace is compared against the learned FSA to detect whether it is abnormal. However, these papers do not discuss interleaved logs which are prevalent in distributed systems. It is much more difficult to learn state machines from interleaved logs. Our analysis is based on message groups, which is not affected by the interleaving patterns.

Mining program invariants is a very important step in our approach. There are some research efforts related to this subject. Ernst et al. developed Daikon [10] to discover program invariants for supporting program evolution. Daikon can dynamically discover invariants at specific source code points by checking the values of all program variables in the scope. Jiang et al. proposed a search algorithm to infer likely invariants in distributed systems [12]. Rather than searching the invariants of program variables, their algorithm searches invariant pair-wise correlations between two flow intensities, such as traffic volume and CPU usage that are monitored in distributed systems. They also proposed an EM algorithm in [11], and extended their work to mine correlations among multiple flow intensities. In contrast with these methods, we mine invariant relationships among the counts of log message types, which present the characteristics of the program execution flow. In addition, we focus on sparse and integer invariants that can reveal the essential relations of the system execution logic and are easily understood by human operators.

3 The Approach

3.1 Invariants in textual logs

In general, a program invariant is a predicate that always holds the same value under different workloads or inputs. Program invariants can be defined from various aspects of a system, including system measurements (e.g. CPU and network utilization [11]) and program variables [10]. Besides the program variables and system measurements, program execution flows can also introduce invariants. With the assumption that log sequences provide enough information for the system execution paths, we can obtain invariants of program execution flows through analyzing log sequences. A simple example of program execution flow is shown in Fig. 1. At each stage of *A, B, C, D*, and *E*, the system prints a corresponding log message. We assume that there are multiple running instances that follow the execution flow shown in Figure 1. Even different instances may execute different branches and their produced logs may interleave together; the following equations should always be satisfied:

$$c(A) = c(B) = c(E) \tag{1}$$

$$c(B) = c(C) + c(D) \tag{2}$$

where $c(A)$, $c(B)$, $c(C)$, $c(D)$, $c(E)$ denote the number of log messages *A, B, C, D*, and *E* in the logs respectively. Each equation corresponds to a specific invariant of the program execution flow, and the validity of such invariants is not affected by the dynamics of the workloads, the difference of system inputs or the interleaving of multiple instances. In this paper, we call them *execution flow invariants*. There are mainly two reasons that we look at linear invariants among the counts of different type of log messages. First, linear invariants encode meaningful characteristics of system execution paths. They universally exist in many standalone or distributed systems. Second, an anomaly often manifests a different execution path from the normal ones. Therefore, a violation of such relations (invariants) means a program execution anomaly. Because log sequences record the underlying execution flow of the system components, we believe there are many such linear equations, i.e. invariants, among the log sequences. If we can automatically discover all such invariants from the collected historical log data, we can facilitate many system management tasks. For example,

- By checking whether a log sequence violates the invariants, we can detect system problems. As mentioned above, a violation of an invariant often means an anomaly in the program's execution.
- Each invariant contains a constraint or an attribute of a system component's execution flow. Based on the related execution flow properties of the broken invariants, system operators can find the potential

causes of failure.

- The invariants can help system operators to better understand the structure and behavior of a system.

3.2 Invariant as a Linear Equation

An invariant linear relationship can be presented as a linear equation. Given m different types of log messages, a linear relationship can be written as follows:

$$a_0 + a_1x_1 + a_2x_2 + \cdots + a_mx_m = 0$$

where x_j is the number of the log messages whose type index is j; $\theta = [\mathrm{a}_0, \mathrm{a}_1, \mathrm{a}_2, \cdots, \mathrm{a}_\mathrm{m}]^\mathrm{T}$ is the vector that represents the coefficients in the equation. So, an invariant can be represented by the vector θ. For example, the invariant of equation (2) can be represented by the vector $\theta = [0,0,1,-1,-1,0]^\mathrm{T}$. Here, the message type indexes of A to E are 1 to 5 respectively. Obviously, independent vectors correspond to different linear relations, so represent different invariants. Given a group of log sequences L_i, $i = 1, \ldots, n$, that are produced by past system executions, we count the number of every type of log messages in every log sequence, x_{ij}, $j = 1, \ldots, m$. Here, x_{ij} is the number of log messages of the j^{th} log message type in the i^{th} log sequence. If none of those log sequences contains failures or problems, and all of log sequences satisfy the invariant, then we have the following linear equations:

$$a_0 + a_1x_{i1} + \cdots + a_mx_{im} = 0, \forall i = 1, \ldots, n \qquad (3)$$

Let us denote

$$\boldsymbol{X} = \begin{bmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1m} \\ 1 & x_{21} & x_{22} & \ddots & x_{2m} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_{n1} & x_{n2} & \cdots & x_{nm} \end{bmatrix}$$

Then the formula (3) can be reformed as a matrix expression (4). That is to say, every invariant vector θ should be a solution of the equation:

$$\boldsymbol{X}\theta = 0 \qquad (4)$$

Formula (4) shows the principle characteristic of the invariants under the condition that all collected history logs do not contain any failure or problem. In practice, a few collected log sequences may contain failures or problems, which will make equation (4) not be precisely satisfied. We will discuss how to deal with such a problem in Section 6. Currently, we just focus on explaining the basic ideas and key concepts related to the execution flow invariant.

We derive two sub-spaces according to the matrix $\boldsymbol{X}$: the row space of matrix $\boldsymbol{X}$, which is the span of the row vectors of $\boldsymbol{X}$, and the null space of matrix $\boldsymbol{X}$, which is the orthogonal complement space to the row space. Formula (4) tells us that an invariant θ can be any vector in the null space of matrix $\boldsymbol{X}$. In this paper, we call the null space of matrix $\boldsymbol{X}$ the ***invariant space*** of the program. Each vector in the invariant space represents an execution flow invariant, and we call the vector in the invariant space the ***invariant vector***. On the other hand, any linear combination of the invariant vectors is also an invariant vector.

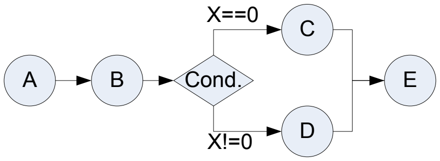

Figure 1. An execution flow example.

Sparseness: Although any vector in the invariant space represents an execution flow invariant, an arbitrary invariant vector with many non-zero coefficients often does not directly correspond to the meaningful work flow structures of a program and cannot be well understood by system operators.

Generally, the whole work flow of a program consists of a lot of elementary work flow structures, such as sequential structures, branched structures, and looping structures. The elementary work flow structures are often much simpler than the whole work flow structures and can be easily understood by system operators. As in the example shown in Figure 1, the sequential structure of A to B, the branch structure of B to C or D, and the joint structure of C or D to E are elementary work flow structures that compose the whole work flow. The invariants corresponding to the elementary work flow structures can usually give system operators intuitive interpretations of the system execution flow. For example, the invariant of $c(B) = c(C) + c(D)$ tells us that there may be a join or branch structure in the workflow.

Because the elementary work flow structures in the program are often quite simple, the invariants corresponding to such elementary work flow structures may involve only a few types of log messages. Therefore, compared to the number of all log message types, the number of message types involved in an elementary invariant is often very small, that is, the vector representations of such elementary invariants are often very sparse. Accordingly, the sparse vectors in the invariant space may correspond to elementary invariants, and general vectors in the invariant space may be linear combinations of the elementary invariant vectors. For example, the equations (1) and (2) represent the elementary invariants that can directly reflect the elementary work flow structures in Figure 1. Each of them involves only two or three types of log messages. However, their linear combination, i.e. the invariant $c(A) + 3c(B) - 2c(E) - 2c(C) - 2c(D) = 0,$ does not directly correspond to a concrete work flow structure in Figure 1 and is not easily understood by system operators.

In this paper, we assume that the number of log sequences is larger than the number of log message types. This assumption is reasonable because the number of log message types is constant and limited, while many logs can be collected while the system is running. If the dimension of the invariant space is r, then the dimension of the row space is $(m + 1 - r)$ because the dimension of the whole space is $(m + 1)$. Therefore, we can always find a vector with no more than $(m + 2 - r)$ non-zero coefficients in the invariant space. That is to say, the number of non-zero coefficients in a sparse invariant vector should be at most $(m + 1 - r)$, or it is not viewed as sparse because we can always find out an invariant vector with $(m + 2 - r)$ non-zero coefficients. We denote the upper bound of the number of non-zero coefficients for sparse invariant vectors as $K(\boldsymbol{X})$, and $K(\boldsymbol{X}) = m + 1 - r$. In real systems, the dimension of the row space is often quite small, so $K(\boldsymbol{X})$ is small too. For example, by investigating a lot of software systems, the authors of [5] observed that the effective dimensions of all row spaces are less than 4.

Compactness: For a set of invariant vectors, it is called a *redundant invariant vector set* if there is at least one invariant vector in the set that can be a linear combination of the other invariant vectors in the same set. On the other hand, if the set does not contain any invariant vector that can be a linear combination of the other invariant vectors in the same set, we call the set a *compact invariant vector set*. Because invariant vectors are essentially equivalent to invariants, it is natural to say that an invariant set is compact if its corresponding invariant vector set is compact, and vice versa. For example, the set $\{c(A) = c(B), c(A) = c(E), c(E) = c(B)\}$ is a redundant set, because the invariant $c(E) = c(B)$ can be deduced from the other two invariants in the set. On the other hand, the set $\{c(A) = c(B), c(A) = c(E), c(B) = c(C) + c(D)\}$ is a compact invariant set. Obviously, a redundant invariant set contains redundant information. If the dimension of the invariant space is r, there exists at most r different invariants satisfying that each of them cannot be a linear combination of the others. Therefore, for any compact invariant set C, the number of invariants in the set, i.e. $|C|$., is not larger than r.

Integer constraint: Another important property of the program execution flow invariants is that all coefficients are integer values. The reason is that all elementary work flow structures, such as sequence, branch, and join, can be interpreted by the invariant vectors whose elements are all integers. For example, the invariant vectors represented in equations (1) and (2) are all integer values, i.e. $[0,1,-1,0,0,0]^T$, $[0,0,1,0,0,-1]^T$ and $[0,0,1,-1,-1,0]^T$. At the same time, integer invariants are easily understood by human operators. In this paper, we aim to automatically mine the largest compact sparse integer invariant set of a program. In the remainder of this paper, the term "invariant" is used to refer to "sparse integer invariant" unless otherwise stated.

3.3 Practical Challenges

In real world systems, some collected historical log sequences contain failures or noise, i.e. they are abnormal log sequences. There also may be some partial log sequences, which are generally caused by inconsistent log data cuts from continuously running system (such as large-scale Internet Services). An invariant may be broken by these log sequences, because the abnormal execution flows are different from the normal execution flows. The results of this are that some of the equations in formula (3) may be not satisfied. With the assumption that failure logs and noise polluted logs only take up a small portion of the historical log sequences (e.g. <5%), we can find all invariants by searching the sparse resolutions of Equ. (4). This can be realized by minimizing the value of $\|\boldsymbol{X}\theta\|_{\boldsymbol{0}}$. Here, $\|\boldsymbol{X}\theta\|_{\boldsymbol{0}}$ equals the number of log sequences that violates the invariant θ.

Generally speaking, minimizing the value of $\|\boldsymbol{X}\theta\|_{\boldsymbol{0}}$ is an NP-Hard problem [19]. To find a sparse invariant with k non-zero coefficients, the computational cost is about $O(C_m^k)$. Fortunately, in many systems, log messages may form some groups in which the log messages contain the same program variable value. Such groups usually represent meaningful workflows related to the specific program variable. For example, log messages containing an identical user request ID can form a group that represents the request handling execution flow in the program; there is a strong and stable correlation among log messages within the same group. On the other hand, inter-group log message types are often not obviously correlated. Furthermore, the number of log message types in each message group is usually much smaller than the total number of log message types. If we can divide all log messages into different groups properly and mine the invariants on different kinds of groups respectively, the search space of the algorithm can be largely reduced. There are some systems in which log messages do not contain such parameters. Just as prior work [5], our approach does not target these systems.

Even with the grouping strategy, the computational cost of invariant searching is still quite large. We try to further reduce the computational cost by introducing early termination and pruning strategies which will be discussed in Section 6.3.

3.4 Workflow of our approach

Figure 2 shows the overall framework of our approach, which consists of four steps: log parsing, log message grouping, invariant mining, and anomaly detection. We will provide further explanation in the corresponding sections.

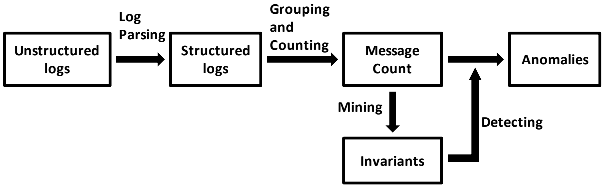

Figure 2. The overall framework of our approach.

Log parsing. In most systems, log messages are usually unstructured free-form text messages, and are difficult to be directly processed by a computer. In the log parsing step, we convert a log message from an unstructured text string to a tuple-form representation that consists of a timestamp, a message signature (i.e. a constant free form text string to represent a log message type), and a list of message parameter values.

Log message grouping and counting. Once parameter values and log message signatures are separated from all log messages, we first automatically discover whether a set of parameters correspond to the same program variable. Then, we group log messages that contain the same value of the same program variable together. For example, the log messages containing the same request ID value are grouped together. As mentioned above, dividing log messages into some close inner-related groups can largely reduce the computational cost. For each message group, we count the number of log messages for each message type to obtain a message count vector for further processing.

Invariant mining. Next, we try to find a compact sparse integer invariant set for each type of the log message groups. Message groups extracted according to the same program variable are considered as the same type of group. For example, the group of log messages with request ID #1# and the group of log messages with request ID #2# are the same type of message groups. In this paper, we combine a brute force searching algorithm and a greedy searching algorithm to make the searching process tractable.

Anomaly detection. We apply the obtained set of invariants to detect anomalies. A log sequence that violates an invariant is labeled as an anomaly.

4 Log Parsing

Each log message often contains two types of information: a free-form text string that is used to describe the semantic meaning of the recorded program event and parameters that are often used to identify the specific system object or to record some important states of the current system.

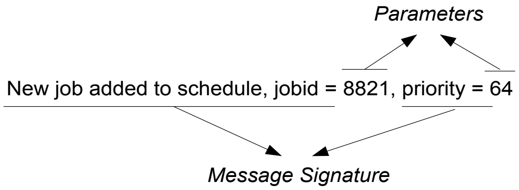

Figure 3. A log message contains two types of information: a message signature and parameters.

In general, log messages printed by the same log-print statement in the source code are the same type of messages because they all correspond to the same execution point and record the same kind of program events with the same semantic meaning. Different types of log messages are usually used to record different program events with different semantic meanings, and printed by different log-printed statements. Naturally, we can use the free-form text string in the log-print statement as a signature to represent the log message type. Therefore, a *message signature* corresponds to the constant content of all log messages that are printed by the same log-print statement. Parameter values are the recorded variable values in the log-print statement, and they may vary in different executions. For example, in Figure 3, the log message signature is the string “New job added to schedule, jobId =, priority =”, and the parameter values are “8821” and “64”.

The log parsing aims to extract message signatures and parameters from the original log messages. If the source code of the target program is available, the method proposed in [5] can automatically parse the log messages with a very high precision. However, in some systems, the source code is not available because it is usually stripped of programs' distribution packages. Therefore, it is still necessary to design a log parser that does not depend on any source code. In this paper, we use the algorithm that we have previously published [14] to extract message signatures and parameter values from log messages. It can achieve an accuracy of more than 95% [14]. Because the log parser is not the focus of this paper, we do not discuss the details of the algorithm.

Once the message signatures and parameter values are extracted from the original log messages, we can convert the unstructured log messages to their corresponding structured representations. For a log message m, we denote the extracted message signature as $K(m)$, the number of parameters as $PN(m)$, and the ith parameter's value as $PV(m,i)$. After message signature and parameter value extraction, each log message m with its time stamp $T(m)$ can be represented by a tuple $\left[T(m), K(m), PV(m,1),\ PV(m,2),\ \dots, PV\big(m, PN(m)\big)\right]$, we call such tuples the *tuple-form representations* of the log messages.

5 Log Message Grouping

The above log parsing procedure helps us to extract the message signature and parameter values for each log message. Each parameter is uniquely defined by a pair consisting of a message signature and a position index. Taking Figure 3 as an example, the pair ("*New job added to schedule, jobid=[],priority=[]* ", 1) defines a parameter, and its value is 8821 in the log messages. Note that a parameter is an abstract representation of a printed variable in the log-print statement, while a parameter value is the concrete value in a specific log message.

Developers often print out the same important program variable in multiple log-print statements. Therefore, multiple parameters may correspond to the same program variable, and we call these parameters cogenetic parameters. For example, a program variable, i.e. request ID, can appear as a parameter in different log message types, and these message types are often related to the execution flow of the request processing. Traditional log analysis tools heavily depend on application specific knowledge to determine whether a set of parameters correspond to the same program variable. However, in practice, most operators may not have enough knowledge about the implementation details. In this paper, we automatically determine whether two parameters are cogenetic. Our algorithm is based on the following observations:

- In a log bunch[1], if two parameters (e.g. P_a and P_b) either have the same value ranges (i.e. $V_r(P_a) = V_r(P_b)$), or one parameter's value range is a subset of the other's (e.g. $V_r(P_a) \subseteq V_r(P_b)$), then they are cogenetic (denoted as $P_a \cong P_b$). Here, the value range of a parameter ($V_r(P_a)$) is defined by the set of all distinct values of the parameter in the log bunch.
- Two parameters with a large joint set $V_r(P_a) \cap V_r(P_b)$ (namely the overlapped value range) will have a high probability to be cogenetic. For two parameters that are not cogenetic, they may have a few identical values in log messages by chance. However, if they have a lot of identical values, it is unlikely that it happens by chance. Therefore, a large overlapped value range often means that two parameters are likely cogenetic.
- The larger the length of each parameter value (i.e. the number of characters of the value in a log message) in the joint set $V_r(P_a) \cap V_r(P_b)$ is, the higher the probability the parameters are cogenetic. Intuitively, it will be more difficult for two parameter values with a large length to be identical by chance.

For each parameter, we first count its value range in each historical log bunch through scanning log messages one by one. Then, we check whether there are pair wise cogenetic relationships using Algorithm 1. For two parameters, if the size of the overlapped value range is larger than a threshold, and the minimal length of the parameter value is larger than 3 characters (Note that we use the text string form of the parameter values in the log messages for analysis), we can determine that the parameters are cogenetic. Finally, based on the obtained pair-wise cogenetic relationships, we can gather a set of parameters that are cogenetic into a parameter group. The detail of the algorithm is described in Algorithm 1.

According to the above algorithm, we obtain a set of parameter groups, with the parameters in each group being cogenetic. Intuitively speaking, each parameter

[1] In this paper, we can collect log messages several times as the target program runs under different workloads. At each time, one or more log files may be collected from distributed machines. The set of all the collected log messages at one time of collection are defined as *a log bunch*.

group corresponds to a program variable. We group log messages with the same program variable values together. Specifically, for each group of cogenetic parameters denoted as A, we group together the log messages that satisfy the following condition: the messages contain the parameters belonging to the specified parameter group A, and the parameter's values in the log messages are all the same.

Algorithm 1. Log parameter grouping

1. For each log parameter (defined by its message signature and its position index), we enumerate its value range (i.e. all distinct values) within each log bunch.
2. For every two parameters (e.g. P_a and P_b) that satisfy the following rules, we conclude that they are cogenetic parameters.
   - For every log bunch, we have $V_r(P_a) \subseteq V_r(P_b)$ or $V_r(P_a) \supseteq V_r(P_b)$.
   - $min(|V_r(P_a)|, |V_r(P_b)|) \geq 10$
   - Each value in $V_r(P_a)$ and $V_r(P_b)$ contains at least 3 characters.
3. We use the following rules to identify the group of cogenetic parameters:
   - If $P_a \cong P_b$ and $P_a \cong P_c$, we can conclude that P_a, P_b, and P_c are cogenetic parameters.

6 Invariant Mining

After the log message grouping step, we can obtain a set of log message groups for each program variable. Each message group describes a program execution path related to the program variable. Since the logging points are chosen by developers, log messages are often very important for problem diagnosis. We collect the log message groups corresponding to the same program variable together and discover their invariants as described in Algorithm 2. At first, for each log message group, we count the number of log messages for each log message type in the message group to obtain one message count vector. For all log messages groups that are related to the same program variable, we can extract a set of message count vectors. The message count vectors that correspond to the same program variable form the count matrix $\boldsymbol{X}$ (Eq. 4). Then, we need to identify the invariant space and the row space of $\boldsymbol{X}$ by using singular value decomposition and analysis. Next, we find the sparse invariant vectors in the invariant space. To find a sparse invariant vector with k non-zero coefficients with a small value of k (e.g, <5 in most cases), we can use a brute force search algorithm to obtain the optimal solution. However, when k is large, the brute force algorithm has to search in a huge searching space. In this case, we use a greedy algorithm [19] to obtain an invariant candidate. Finally, we validate the found invariants using the collected historical logs.

6.1 Estimate the invariant space

Once we have constructed the matrix $\boldsymbol{X}$ from the collected historical logs, we can estimate the invariant space by singular value decomposition (SVD) operation.

Instead of the energy ratio, we use the support ratio as a criterion to determine the invariant space (and, at the same time, the row space). It can directly measure the matching degree between the collected logs and the invariants. For an invariant, the support ratio is defined as the percentage of the log message groups that do not break the invariant. Specifically, we first use SVD to obtain the right-singular vectors. After that, we evaluate the right-singular vectors one by one in increasing order of singular values to check whether they are a part of the invariant space. For a right-singular vector v_i, if there are more than 98% log message groups satisfying the condition $|X_j v_i| < \epsilon$, we treat v_i as a validated invariant. Otherwise, it is an invalidated invariant. Here X_j is a message count vector of the message group j, ϵ is a threshold. The right-singular vector with the smallest singular value is evaluated first. Then, the vector with the second smallest singular value is evaluated, and so on. If a singular vector is verified as an invalidated invariant, the evaluation process is terminated. The invariant space is a span of all right-singular vectors that have been validated during this process. In our implementation, the threshold ϵ is selected as 0.5 ($\approx \sqrt{4}/4$) because most of our invariants at most contain 4 non-zero coefficients.

6.2 Invariant searching

In this section, we introduce an invariant searching algorithm which aims to find a compact set of program invariants based on a log message count matrix $\boldsymbol{X}$. Because we have little knowledge about the relationship between different log message types, we try any hypotheses of non-zero coefficients in different dimensions to construct a potential sparse invariant, and then continue to validate whether it fits with the historical log data.

Specifically, we define an invariant hypothesis as its non-zero coefficient pattern $\{p_j, j = 1,2,\cdots,k\}$, where p_j is the index of the non-zero coefficient of the invariant hypothesis, and $0 \leq p_j < p_{j+1} \leq m$. For any non-zero coefficient pattern, we check whether there is an invariant, i.e. $\{a_{p_j}, j = 1,2,\cdots,k\}$. There are two steps. At first, we try to obtain an invariant candidate $\hat{\theta}$ that satisfies the given non-zero coefficient pattern and minimizes the value of $\|\hat{\boldsymbol{X}}\hat{\theta}\|_0$, namely $\hat{\theta} = \text{argmin}_\theta(\|\hat{\boldsymbol{X}}\theta\|_0)$. Here, $\hat{\boldsymbol{X}}$ is a matrix that contains only k column vectors of matrix $\boldsymbol{X}$ whose column indexes are $\{p_j, j = 1,2,\cdots,k\}$. We ignore other columns in X for constructing $\hat{\boldsymbol{X}}$ because those columns correspond to zero coefficients of the invariant vectors. Because an optimization operation over zero norm is often not tractable, we estimate $\hat{\theta}$ through $\hat{\theta} = \text{argmin}_\theta(\|\hat{\boldsymbol{X}}\theta\|_2)$. The coefficients of the estimated $\hat{\theta}$ are often within the range of $(-1.0, 1.0)$. In order to obtain integer invariant candidates, we scale up $\hat{\theta}$ to make its minimal non-zero coefficient equal to an integer l , and round other non-zero coefficients to an integer accordingly. In this paper, we set $l = 1,2,\cdots,p$ respectively. Therefore, we obtain p integer invariant candidates. Then, we verify each of them by checking its support ratio based on the log message groups. If there is an invariant whose support ratio is larger than γ, we set it as a validated invariant, otherwise, we conclude that there is no invariant that satisfies the non-zero coefficients pattern $\{p_j, j = 1,2,\cdots,k\}$. Here, γ is a user defined threshold, which is set as 98% in our experiments. In our implementation, we can handle all cases that we have studied by selecting $p = 3$. A large value of p is often not necessary, because most complex invariants are linear combinations of simple local invariants.

Algorithm 2. Mining Invariants

1. For all message groups related to a specific parameter group, we construct the matrix X using their message count vectors, and estimate the dimension of the invariant space (denoted as r).
2. We use a brute force algorithm to search invariants that contains k non-zero coefficients, where k increases from 1 to 5 in turn. The algorithm exits when one of following conditions is satisfied:
   - r independent invariants have been obtained.
   - $k > (m - r + 1)$
3. If $(m - r + 1) > 5$ and no early terminate condition has be satisfied, we use a greedy algorithm [19] to find potential invariants for *k* >*5*.

6.3 Computational Cost and Scalability

In general, it is an NP-Hard problem to find a sparse invariant vector. The computational complexity of the above search algorithm is about $O(\sum_{i=1}^{m-r+1} C_m^i)$. Although it has been largely reduced from the computational cost of full search space (i.e. $O(\sum_{i=1}^{m} C_m^i)$), it is still not a trivial task if the number of dimensions of matrix $\boldsymbol{X}$'s row space (i.e. $m - r + 1$) is large. Fortunately, in real world systems, the dimensions of the row spaces are often very small, which helps us to avoid the problem of combinatorial explosion. For example, in Table 1, we list the row space dimensions of different types of log message groups. Many of them are not larger than 4. Therefore, the computational cost can usually be controlled below $O(\sum_{i=1}^{4} C_m^i)$.

In addition, most real world sparse invariants often only contain 2 or 3 non-zero coefficients. Because we can obtain at most *r* independent invariants, we do not need to search the combinations of 5 or more non-zero coefficients if we have obtained *r* independent invariants when *k*<5. For example, the 4th row of Table 3 is such a case. This allows us to terminate the search process early, and to reduce the computational cost.

Table 1. Low dimensionality of row space

Message group of related object identifier	m	$\boldsymbol{m - r + 1}$
Hadoop logs with MapTask ID	7	3
Hadoop logs with ReduceTask ID	3	2
Hadoop logs with MapTask Attempt ID	28	4
Hadoop logs with ReduceTask Attempt ID	25	6
Hadoop logs with JVM ID	7	2

Table 2. Reduce computational cost

Message group of related object identifier	Original search space	Result search space
Hadoop logs with MapTask ID	63	37
Hadoop logs with ReduceTask ID	6	6
Hadoop logs with MapTask Attempt ID	24157	3310
Hadoop logs with ReduceTask Attempt ID	15275	730
Hadoop logs with JVM ID	28	16

Furthermore, we can reduce the computational cost by skipping the searching on some hypothesis candidates. As discussed in Section 3, any linear combination of invariants is also an invariant. Therefore, we need not search the invariants that can be a linear combination of the detected ones. Then, the search space can be largely reduced by skipping the search on such combinations. At the same time, the skipping strategy also guarantees the compactness of our discovered invariants. Table 2 shows the effectiveness of our early termination and skipping strategy. The numbers of hypotheses in the original search space (i.e. $O(\sum_{i=1}^{m-r+1} C_m^i)$) are listed in the second column. The third column contains the size of the search space after applying the early termination and the skipping strategy. By comparing the search space size in the two columns, we can find that the early termination and the skipping strategy largely reduce the search space, especially for the message groups with high dimension values.

In our implementation, we only search the invariant hypotheses up to 5 non-zero coefficients. If no early termination condition is met, we then find potential invariants by using a greedy algorithm [19] on the message types that do not appear in the existing invariants. However, the greedy algorithm cannot guarantee to find all invariants in logs. The overall algorithm is presented in Algorithm 2.

From the view of scalability, there are a huge amount of log messages in a large scale system with thousands of machines. Therefore, the row number n of matrix X is often very large. Directly applying SVD on matrix X is often not scalable. Fortunately, the number of message types (i.e. m) is usually limited, and it does not increase as the system scales up. We can replace the SVD operation by an Eigen Value Decomposition (EVD) operation to calculate the right-singular vectors, because $\boldsymbol{X} = U\Lambda V^T \Rightarrow \sum = \boldsymbol{X}^T\boldsymbol{X} = V\Lambda^T\Lambda V^T$. Here, matrix $\sum = \boldsymbol{X}^T\boldsymbol{X}$ is a $m \times m$ matrix. It can be easily calculated by a MapReduce-like distributed program. Similarly, we can also use an EVD operation to estimate $\hat{\theta}$ based on a matrix $\hat{\sum} = \hat{\boldsymbol{X}}^T\hat{\boldsymbol{X}}$ for each invariant hypothesis. The matrix $\hat{\sum}$ can directly be calculated from the matrix $\sum$. At the same time, the support ratio counting procedure can also be easily performed in a distributed manner. Therefore, our algorithm can be easily scaled up. In addition, most program invariants do not depend on the scale of the system. We can learn invariants from the logs of a small scale system deployment, and then use these invariants to detect problems of a large scale deployment.

7 Problem Detection

Once the program invariants are discovered, it is straightforward to detect problems from console logs. For a new input console log, we first convert the unstructured log messages to tuple-form representations using the log parser, and then group log messages and calculate a count vector for each message group. After that, we check every message count vector with its related learned invariants. The message group whose message count vector violates any one of its related invariants is considered as an anomaly.

The automatically mined invariants by our approach reflect the elementary execution flows. These program invariants often provide intuitive and meaningful information to human operators, and help them to locate problems on the fine granularity. Therefore, we relate each detected anomaly with the invariants that it breaks so as to provide insight cues for problem diagnosis. Operators can check which invariants are broken by an anomaly, and how many anomalies are raised by the violation of a specific invariant.

8 Case Study and Comparision

In this section, we evaluate the proposed approach through case studies on two typical distributed computing systems: Hadoop and CloudDB, a structured data storage service developed by Microsoft. We first set up a testing environment and collect console logs. And then, we begin our experiments of anomaly detection on these two systems. The detection results are presented and analyzed in the following subsections. Unlike CloudDB, Hadoop is a publicly available open-source project. The results on Hadoop are easy to be verified and reproduced by third parties. Therefore, we give more details about the results on Hadoop.

8.1 Case Study on Hadoop

Test Environment Setup: Hadoop [18] is a well-known open-source implementation of Google's Map-Reduce framework and distributed file system (GFS). It enables distributed computing of large scale, data-intensive and stage-based parallel applications.

Our test bed of Hadoop (version 0.19) contains 15 slave workstations (with 3 different hardware configurations) and a master workstation, and all these machines are connected to the same 1G Ethernet switch. We run different Hadoop jobs including some simple sample applications, such as WordCount and Sort, on the test bed. The WordCount job counts the word frequency in some random generated input text files, and the Sort job sorts the numbers in the input files. During the running of these jobs, we randomly run some resource intensive programs (e.g. CPUEater) on the slave machines to compete for CPU, memory and network resources with Hadoop jobs. Rather than active error injection, we hope such intensive resource competition can expose inherent bugs in Hadoop. We collect the produced log files of these jobs 4 times at different time points. Each time, we put the collected log files into a single file folder. The log data is not uniformly distributed in these folders. The smallest folder contains about 116 megabytes, and the largest folder contains about 1.3 gigabytes. The log messages in each folder are considered as a log bunch. There are totally about 24 million lines of log messages.

Results of Parameter Grouping: Our parameter grouping algorithm identifies several parameter groups as meaningful program variables. By manually checking the log messages and the parameters, we find that they corresponded to the following meaningful program variables: Map/Reduce Task ID, Map/Reduce Task Attempt ID, Block ID, and JVM ID, Storage ID, IP address and port, and write data size of task shuffling. These variables include both object identifiers (such as Task ID) and system states (such as IP address and Port). It is interesting that the packet size during shuffling operations is also detected as a parameter group. We discover one invariant from its related message group, and learn that the number of MAPRED_SHUFFLE operations is equal to the number of messages of "Sent out bytes for reduce:## from map:## given from with (#,#)".

Table 3. Invariants found in Hadoop logs

Message groups of related object identifiers	Invariants (≤3 coef.)	Invariants (≥4 coef.)
Hadoop logs with MapTask ID	3	0
Hadoop logs with ReduceTask ID	1	0
Hadoop logs with MapTask Attempt ID	21	3
Hadoop logs with ReduceTask Attempt ID	17	0
Hadoop logs with Data Block ID	9	0
Hadoop logs with JVM ID	5	0
Hadoop Logs with Storage ID	3	0
Logs with IP/port	4	0
Logs with task write packet size	1	0

Learned Invariants: In the Hadoop experiments, we discover 67 invariants in total. 64 of them only contain at most three non-zero coefficients, and 3 invariants have 4 non-zero coefficients. Table 3 shows the numbers of the learned invariants for different program variables. To validate our learned invariants, we manually verify our learned program invariants by carefully studying the Hadoop source code, the documents on MapReduce, and the sample logs. By comparing with the real program work flows, we find that our discovered invariants correctly describe the inherent linear relationships in the work flow. No false positive invariant is found. To vividly illustrate our discovered invariants, we present an example - the learned ternary invariant of the MapTask log group. The invariant equation is $c(L_{113}) + c(L_{114}) = c(L_{90})$, where L_{113}, L_{114}, and L_{90} are the log message types of "*Choosing data-local task ##*", "*Choosing rack-local task ##*", and "*Adding task '##' to tip ##, for tracker '##'*" respectively. In our test environment, all 16 machines are connected to a single Ethernet switch, and they are configured as one rack. Therefore, for each MapTask, it selects its data source from either local disc or local rack. The above invariant correctly reflects this property because the equation shows that each "*Adding task*" corresponds to either a "*data-local*" or a "*rack-local*". This proves our claim in Section 3 that invariants encode the properties of work flow structures.

Table 4. Detected true problems in Hadoop

Anomaly Description	PCA based Method	Our Method
Tasks fail due to heart beat lost.	397	779
A killed task continued to be in RUNNING state in both the JobTracker and that TaskTracker for ever	730	1133
Ask more than one node to replicate the same block to a single node simultaneously	26	26
Write a block already existed	25	25
Task JVM hang	204	204
Swap a JVM, but mark it as unknown.	87	87
Swap a JVM, and delete it immediately	211	211
Try to delete a data block when it is opened by a client	3	6
JVM inconsistent state	73	416
The pollForTaskWithClosedJob call from a Jobtracker to a task tracker times out when a job completes.	3	3

Anomaly Detection: We use the learned invariants to detect anomalies by checking whether a log sequence breaks a program invariant. By manually checking these detected anomalies, we find that there are 10 types of different execution anomalies, which are listed in Table 4. Note that each anomaly in Table 4 corresponds to a specific pattern corresponding to a certain set of violated invariants, and its description is manually labeled by our carefully studying the source code and documents of Hadoop. Many of them are caused by the loss of the heart beat message from Tasktracker to Jobtracker. Our method also detects some subtle anomalies in Hadoop. For example (the 4th row of Table 4), we detect that Hadoop DFS has a bug that asks more than one node to send the same data block to a single node for data replication. This problem is detected because it violates a learned invariant of "count('Receiving block ##') = count('Deleting block file ##')". A node receives more blocks than it finally deletes, and some duplicated received blocks are dropped.

At the same time, we find that our approach can well handle the problems that cause the confusion in the traditional keyword based log analysis tools. Here is one typical example. In Hadoop, TaskTracker often logs many non-relevant logs at info level for DiskChecker$DiskErrorException. According to Apache issue tracking HADOOP-4936, this happens when the map task has not created an output file, and it does not indicate a running anomaly. Traditional keyword-based tools may detect these logs as anomalies, because they find the word *Exception*. This confused many users. Unlike keyword-based tools, our approach can avoid generating such false positives.

Table 5. False positives

False Positive Description	PCA Method	Our Method
Killed speculative tasks	585	1777
Job cleanup and job setup tasks	323	778
The data block replica of Java execution file	56	0
Unknown Reason	499	0

Just like all unsupervised learning algorithms, our approach does detect some false positives. As shown in Table 5, we detect two types of false positives. Hadoop has a scheduling strategy to run a small number of speculative tasks. Therefore, there may be two running instances of the same task at the same time. If one task instance finishes, the other task instance will be killed no matter which stage the task instance is running at. Some log groups produced by the killed speculative task instances are detected as anomalies by our approach, because their behaviors are largely different from normal tasks. The other false positives come from job cleanup and setup tasks. Hadoop schedules two tasks to handle job setup and cleanup related operations. Since these two tasks, i.e. job setup task and job cleanup task, print out the same type of log messages as map tasks. Many users are confused by these logs. Because their behaviors are quite different from the normal worker map tasks, our approach also detects them as anomalies.

Comparison with the PCA Based Algorithm: We compared our approach with the PCA based algorithm of [5]. Because our running environment, work load characteristics, and Hadoop version are different from the experiments in [5], we cannot directly compare our results with theirs. We implement their algorithm and test it on our data set. From Table 4 and Table 5, we can find that both algorithms can detect the same types of anomalies, which is reasonable because both approaches utilize the inherent linear characteristics of the console logs. In some cases, our approach can detect more anomalies than the PCA based approach. If a set of log messages appear in almost all log message groups, the PCA based algorithm will ignore them by giving a very small TF/IDF weight. Therefore, the PCA based algorithm cannot detect the anomalies exposed as abnormal relationships among the log message types. For example, in a case of "*JVM inconsistent state*" (refer to the 10th row of Table 4), our algorithm detects the anomaly because the message "*Removed completed task ## from*" abnormally appears twice for the same task instance (i.e. breaking an invariant of one message for each task). However, the PCA based algorithm cannot detect these anomalies because it ignores the message. On the whole, in our test data, our approach can detect all the anomalies that can be detected by the PCA based method.

Unlike the PCA based approach, our invariant based approach can give human operators intuitive insight of an anomaly, and help them to locate anomalies in finer granularity. For example, the fact that an anomaly of "*a task JVM hang*" (refer to the 5th row of Table 4) breaks the invariant of "count('JVM spawned') = count('JVM exited')" can give operators a very useful cue to the understanding of the anomaly: a JVM spawned but does not exit, which may indicate the JVM is hung. At the same time, because the anomaly does not break "count('JVM spawned') = count('JVM with ID:# given task:#')", we can conclude that the JVM got hung after it was assigned a MapReduce task. However, the PCA based approach can only tell operators that the feature vector of the anomaly is far away from the normal feature space, i.e. the residential value of the feature vector is larger than a certain threshold value. This can hardly help operators to diagnose the problems, and they have to check the original log sequence carefully for problem analysis. In the PCA based approach, the decision tree technique makes its decision based on the count of one type of log messages at each step of the decision tree [5]. In contrast, our invariant based approach utilizes the numerical relationships among different types of log messages.

The 4th row of Table 5 shows another advantage of our approach. In order to rapidly distribute the Java executable files (e.g. JAR file) of jobs, Hadoop sets the replica number of these files according to the number of slave machines. For example, in our test environment, the replica number of a job JAR file is set as 15. The PCA based algorithm detects them as anomalies, because 15 is far from the normal replica number (e.g. 3 in most systems) of the data block. Our approach does not detect them as anomalies because their work flows do not break any invariant. There are some other false positive cases (refer to the 5th row of Table 5), e.g. some log groups of successful tasks, in the results of PCA based algorithm. We speculate that these false positives may be caused by the different characteristics of work load (WordCount and Sort are different), but currently, we do not know the exact reason. It seems that the PCA based method is more sensitive to the workload and environment. Our algorithm is much more robust. Furthermore, our algorithm only detects two types of false positives, while the PCA based method detects more than 4 types (we believe that the unknown reason false positives belong to many different types.). We argue that, rather than the number of false positives, the number of false positive types is more important for a problem detection tool. In fact, when the tool detects an anomaly, if a human operator marks it as a false positive, the tool can automatically suppress to pop up false positives of the same type. Therefore, a tool with few types of false positives can reduce the operator's workload.

8.2 Case Study on CloudDB

MS CloudDB is a structured data storage service developed for internal usage in Microsoft. It can scale up to tens of thousands of servers and runs on commodity hardware. It is capable of auto-failover, auto-load balancing, and auto-partitioning. In our experiments, we use the log messages of Fabric and CASNode levels, which implement the protocols and life cycles of distributed storage nodes, to learn invariants and detect potential anomalies. About 12 million log messages are analyzed in the experiment. We first manually construct some work flow models based on the documents provided by the product group. Due to the insufficiency of documents, not all work flows involved in these two levels are constructed. Then, we compare the invariants automatically mined by our approach (266 invariants are learned in this experiment) with the manually constructed work flow models. The mined invariants not only correctly reflect the real work flows, but also help us to find out some mistakes in the manually constructed work flow models that are caused by the misunderstanding of some content in the documents.

Table 6. Detected anomalies in CloudDB

Anomaly Description	PCA Method	Our Method
Data store operation finished without client response	0	2
Service message lost	8	8
Refresh config message lost	0	2
LookupTableUpdate message lost	0	1
AddReplicaCompleted message lost	1	8
Fail to close channel	2	67
No response for an introduce request	0	2
Send depart message exception	0	2
Add primary failed	0	2

After that, we also use the learned invariants to perform anomaly detection. Table 6 summarizes the detected anomalies in the experiment. By comparing Table 6 and Table 4, we can obtain a similar conclusion as that in Section 8.1 about the performances of the two methods. As mentioned in Section 8.1, the PCA based algorithm fails to detect lots of anomalies because it gives a very small TF/IDF weight to each routine message.

9 Conclusions

In this paper, we propose a general approach to detecting system anomalies through the analysis of console logs. Because the log messages are usually free form text strings that can hardly be analyzed directly, we first convert unstructured log messages to structured logs in tuple-form representations. The parameters that represent the same program variable are classified into a parameter group. We identify parameter groups by analyzing the relationships among the value ranges of the parameters. Then, according to these parameter groups, we classify the log messages to log message groups, and construct message count vectors. After that, we mine the sparse, integer valued invariants from the message count vectors. The mined invariants can reflect the elementary work flow structures in the program. They have physical meanings, and can be easily understood by operators. Finally, we use the discovered invariants to detect anomalies in system logs. Experiments on large scale systems such as Hadoop and CloudDB have shown that our algorithm can detect numerous real problems with high accuracy, which is comparable with the state of art approach [5]. In particular, our approach can detect anomalies with finer granularity and provide human operators with insight cues for problem diagnosis. We believe that this approach can be a powerful tool for system monitoring, problem detection, and management.

Acknowledgements

We thank Wei XU from UC Berkley for his helpful discussion, and our colleagues, Zheng Zhang, Lidong Zhou and Zhenyu Guo, for their valuable comments on the project. We also thank our paper shepherd, Emin Gün Sirer, and the anonymous reviewers for their insightful comments and advice.

10 References

[1] J. P. Rouillard, *Real-time Log File Analysis Using the Simple Event Correlator (SEC),* In Proc. of the 18th Usenix LISA’04, Nov. 14-19, 2004.

[2] J. E. Prewett, *Analyzing Cluster Log Files Using Logsurfer*, In Proc. of Annual Conference on Linux Clusters, 2003.

[3] S. E. Hansen, and E. T. Atkins, *Automated System Monitoring and Notification with Swatch*, In Proc. of the 7th Usenix LISA’93, 1993.

[4] A. Oliner and J. Stearley. *What supercomputers say: A Study of Five System Logs*. In Proc. of IEEE DSN’07, 2007.

[5] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, *Detecting Large-Scale System Problems by Mining Console Logs*, In Proc. of ACM SIGOPS SOSP’09, Big Sky, MT, Oct. 11-14, 2009.

[6] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan, *Mining Console Logs for Large-Scale System Problem Detection*, In Proc. of SysML, Dec. 2008.

[7] W. Dickinson, D. Leon, and A. Podgurski, *Finding Failures by Cluster Analysis of Execution Profiles*, In Proc. of ICSE, May 2001.

[8] A.V. Mirgorodskiy, N. Maruyama, and B.P. Miller, *Problem Diagnosis in Large-Scale Computing Environments*, In Proc. of the ACM/IEEE SC 2006 Conference, Nov. 2006.

[9] C. Yuan, N. Lao, J.R. Wen, J. Li, Z. Zhang, Y.M. Wang, and W. Y. Ma, *Automated Known Problem Diagnosis with Event Traces*, In Proc. of EuroSys’06, Apr. 2006.

[10] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, *Dynamically Discovering Likely Program Invariants to Support Program Evolution*, In IEEE Trans. on Software Engineering, pp. 99-123, Vol.27, No.2, Feb. 2001.

[11] H. Chen, H. Cheng, G. Jiang, and K. Yoshihira, *Exploiting Local and Global Invariants for the Management of Large Scale Information Systems*, In Proc. of ICDM’08, Pisa, Italy, Dec. 2008.

[12] G. Jiang, H. Chen, and K. Yoshihira, *Efficient and Scalable Algorithms for Inferring Likely Invariants in Distributed Systems*, In IEEE Trans. on Knowledge and Data Engineering, pp. 1508-1523, Vol.19, No. 11, Nov. 2007.

[13] H. Chen, G. Jiang, C. Ungureanu, and K. Yoshihira, *Failure Detection and Localization in Component Based Systems by Online Tracking*, In Proc. of SIGKDD, pp. 750-755, 2005.

[14] Q. Fu, J.-G. Lou, Y. Wang, and J. LI*, Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis*, In Proc. of ICDM, Florida, Dec. 2009.

[15] J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasim-han, *SALSA: Analyzing Logs as State Machines*, In Proc. of WASL, Dec. 2008.

[16] D. Cotroneo, R. Pietrantuono, L. Mariani, and F. Pastore, *Investigation of Failure Causes in Workload Driven Reliability Testing*, In Proc. of the 4th Workshop on Software Quality Assurance, Sep. 2007.

[17] G. Jiang, H. Chen, C. Ungureanu, and K. Yoshihira. Multi-*resolution Abnormal Trace Detection Using Varied-length N-grams and Automata*, In Proc. of ICAC, Jun. 2005.

[18] Hadoop. http://hadoop.apache.org/core.

[19] D. Donoho, Y. Tsaig, I. Drori, and J. Starck, *Sparse Solutions of Underdetermined Linear Equations by Stagewise Orthogonal Matching Pursuit*, Technical Report of Department of Statistics, Standford, TR2006-02, 2006.