DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning

Min Du, Feifei Li, Guineng Zheng, Vivek Srikumar
School of Computing, University of Utah
{mind, lifeifei, guineng, svivek}@cs.utah.edu

ABSTRACT

Anomaly detection is a critical step towards building a secure and trustworthy system. The primary purpose of a system log is to record system states and significant events at various critical points to help debug system failures and perform root cause analysis. Such log data is universally available in nearly all computer systems. Log data is an important and valuable resource for understanding system status and performance issues; therefore, the various system logs are naturally excellent source of information for online monitoring and anomaly detection. We propose DeepLog, a deep neural network model utilizing Long Short-Term Memory (LSTM), to model a system log as a natural language sequence. This allows DeepLog to automatically learn log patterns from normal execution, and detect anomalies when log patterns deviate from the model trained from log data under normal execution. In addition, we demonstrate how to incrementally update the DeepLog model in an online fashion so that it can adapt to new log patterns over time. Furthermore, DeepLog constructs workflows from the underlying system log so that once an anomaly is detected, users can diagnose the detected anomaly and perform root cause analysis effectively. Extensive experimental evaluations over large log data have shown that DeepLog has outperformed other existing log-based anomaly detection methods based on traditional data mining methodologies.

CCS CONCEPTS

•**Information systems** → *Online analytical processing;* •**Security and privacy** → *Intrusion/anomaly detection and malware mitigation;*

KEYWORDS

Anomaly detection; deep learning; log data analysis.

1 INTRODUCTION

Anomaly detection is an essential task towards building a secure and trustworthy computer system. As systems and applications get increasingly more complex than ever before, they are subject to more bugs and vulnerabilities that an adversary may exploit to launch attacks. Such attacks are also getting increasingly more sophisticated. As a result, anomaly detection has become more challenging and many traditional anomaly detection methods based on standard mining methodologies are no longer effective.

System logs record system states and significant events at various critical points to help debug performance issues and failures, and perform root cause analysis. Such log data is universally available in nearly all computer systems and is a valuable resource for understanding system status. Furthermore, since system logs record noteworthy events *as they occur* from actively running processes, they are an excellent source of information for online monitoring and anomaly detection.

Existing approaches that leverage system log data for anomaly detection can be broadly classified into three groups: PCA based approaches over log message counters [39], invariant mining based methods to capture co-occurrence patterns between different log keys [21], and workflow based methods to identify execution anomalies in program logic flows [42]. Even though they are successful in certain scenarios, none of them is effective as a universal anomaly detection method that is able to guard against different attacks in an online fashion.

This work proposes DeepLog, a data-driven approach for anomaly detection that leverages the large volumes of system logs. The key intuition behind the design of DeepLog is from natural language processing: we view log entries as elements of a sequence that follows certain patterns and grammar rules. Indeed, a system log is produced by a program that follows a rigorous set of logic and control flows, and is very much like a natural language (though more structured and restricted in vocabulary). To that end, DeepLog is a deep neural network that models this sequence of log entries using a Long Short-Term Memory (LSTM) [18]. This allows DeepLog to automatically learn a model of log patterns from normal execution and flag deviations from normal system execution as anomalies. Furthermore, since it is a learning-driven approach, it is possible to incrementally update the DeepLog model so that it can adapt to new log patterns that emerge over time.

Challenges. Log data are unstructured, and their format and semantics can vary significantly from system to system. It is already challenging to diagnose a problem using unstructured logs even after knowing an error has occurred [43]; online anomaly detection from massive log data is even more challenging. Some existing methods use rule-based approaches to address this issue, which requires specific domain knowledge [41], e.g., using features like "IP address" to parse a log. However, this does not work for general purpose anomaly detection where it is almost impossible to know a priori what are *interesting features* in different types of logs (and to guard against different types of attacks).

Anomaly detection has to be timely in order to be useful so that users can intervene in an ongoing attack or a system performance issue [10]. Decisions are to be made in streaming fashion. As

CCS'17, Oct. 30–Nov. 3, 2017, Dallas, TX, USA.
 ISBN 978-1-4503-4946-8/17/10…$15.00
DOI: http://dx.doi.org/10.1145/3133956.3134015

a result, offline methods that need to make several passes over the entire log data are not applicable in our setting [22, 39]. We would also like to be able to detect *unknown* types of anomalies, rather than gearing towards specific types of anomalies. Therefore, previous work [44] that use both normal and abnormal (for specific types of anomalies) log data entries to train a binary classifier for anomaly detection is not useful in this context.

Another challenge comes from concurrency. Clearly, the order of log messages in a log provides important information for diagnosis and analysis (e.g., identify the execution path of a program). However, in many system logs, log messages are produced by several different threads or concurrently running tasks. Such concurrency makes it hard to apply workflow based anomaly detection methods [42] which use a workflow model for a single task as a generative model to match against a sequence of log messages.

Lastly, each log message contains rich information such as a log key and one or more metric values, as well as its timestamp. A holistic approach that integrates and utilizes these different pieces of information will be more effective. Most existing methods [22, 32, 39, 41, 42, 44] analyze only one specific part of a log message (e.g., the log key) which limits the types of anomalies they can detect.

Our contribution. A Recurrent Neural Network (RNN) is an artificial neural network that uses a loop to forward the output of last state to current input, thus keeping track of history for making predictions. Long Short-Term Memory (LSTM) networks [13, 18, 27] are an instance of RNNs that have the ability to remember long-term dependencies over sequences. LSTMs have demonstrated success in various tasks such as machine translation [35], sentiment analysis [8], and medical self-diagnosis [20].

Inspired by the observation that entries in a system log are *a sequence of events produced by the execution of structured source code (and hence can be viewed as a structured language)*, we design the DeepLog framework using a LSTM neural network for online anomaly detection over system logs. DeepLog uses not only log keys but also metric values in a log entry for anomaly detection, hence, it is able to capture different types of anomalies. DeepLog only depends on a small training data set that consists of a sequence of "normal log entries". After the training phase, DeepLog can recognize normal log sequences and can be used for online anomaly detection over incoming log entries in a streaming fashion.

Intuitively, DeepLog implicitly captures the potentially nonlinear and high dimensional dependencies among log entries from the training data that correspond to normal system execution paths. To help users diagnose a problem once an anomaly is identified, DeepLog also builds workflow models from log entries during its training phase. DeepLog separates log entries produced by concurrent tasks or threads into different sequences so that a workflow model can be constructed for each separate task.

Our evaluation shows that on a large HDFS log dataset explored by previous work [22, 39], trained on only a very small fraction (less than 1%) of log entries corresponding to normal system execution, DeepLog can achieve almost 100% detection accuracy on the remaining 99% of log entries. Results from a large OpenStack log convey a similar trend. Furthermore, DeepLog also provides the ability to incrementally update its weights during the detection phase by incorporating live user feedback. More specifically, DeepLog provides a mechanism for user feedback if a normal log entry is incorrectly classified as an anomaly. DeepLog can then use such feedback to adjust its weights dynamically online over time to adapt itself to new system execution (hence, new log) patterns.

2 PRELIMINARIES

2.1 Log parser

We first parse unstructured, free-text log entries into a structured representation, so that we can learn a sequential model over this structured data. As shown by several prior work [9, 22, 39, 42, 45], an effective methodology is to extract a "log key" (also known as "message type") from each log entry. The log key of a log entry e refers to the string constant k from the print statement in the source code which printed e during the execution of that code. For example, the log key k for log entry e = *"Took 10 seconds to build instance."* is k = *Took * seconds to build instance.*, which is the string constant from the print statement *printf("Took %f seconds to build instance.", t)*. Note that the parameter(s) are abstracted as asterisk(s) in a log key. These metric values reflect the underlying system state and performance status. Values of certain parameters may serve as identifiers for a particular execution sequence, such as `block_id` in a HDFS log and `instance_id` in an OpenStack log. These identifiers can group log entries together or untangle log entries produced by concurrent processes to separate, single-thread sequential sequences [22, 39, 42, 45]. The state-of-the-art log parsing method is represented by Spell [9], an *unsupervised* streaming parser that parses incoming log entries in an online fashion based on the idea of LCS (longest common subsequence).

Past work on log analysis [22, 39, 42, 44] have discarded timestamp and/or parameter values in a log entry, and only used log keys to detect anomalies. DeepLog stores parameter values for each log entry e, as well as the time elapsed between e and its predecessor, into a vector $\overrightarrow{v}_e$. This vector is used by DeepLog in addition to the log key. An example is given in Table 1, which shows the parsing results for a sequence of log entries from multiple rounds of execution of virtual machine (VM) deletion task in OpenStack.

2.2 DeepLog architecture and overview

The architecture of DeepLog is shown in Figure 1 with three main components: the log key anomaly detection model, the parameter value anomaly detection model, and the workflow model to diagnose detected anomalies.

Training stage. Training data for DeepLog are log entries from normal system execution path. Each log entry is parsed to a log key and a parameter value vector. The log key sequence parsed from a training log file is used by DeepLog to train a log key anomaly detection model, and to construct system execution workflow models for diagnosis purposes. For each distinct key k, DeepLog also trains and maintains a model for detecting system performance anomalies as reflected by these metric values, trained by the parameter value vector sequence of k.

Detection stage. A newly arrived log entry is parsed into a log key and a parameter value vector. DeepLog first uses the log key

log message (log key underlined)	log key	parameter value vector
t_1 Deletion of **file1** complete	k_1	$[t_1 - t_0$, file1Id$]$
t_2 Took **0.61** seconds to deallocate network ...	k_2	$[t_2 - t_1, 0.61]$
t_3 VM Stopped (Lifecycle Event)	k_3	$[t_3 - t_2]$
...	...	...

Table 1: Log entries from OpenStack VM deletion task.

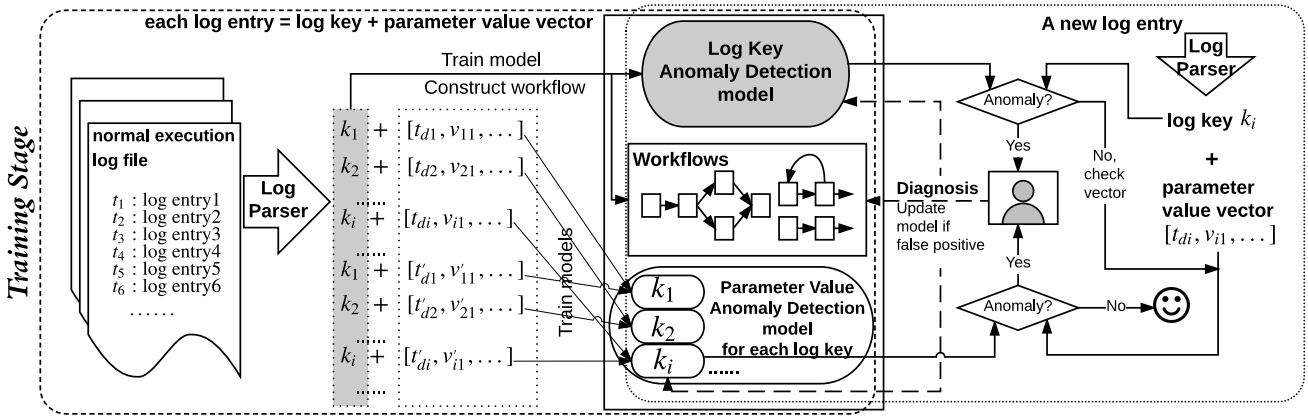


Figure 1: DeepLog architecture.

anomaly detection model to check whether the incoming log key is normal. If yes, DeepLog further checks the parameter value vector using the parameter value anomaly detection model for that log key. The new entry will be labeled as an anomaly if either its log key or its parameter value vector is predicted being abnormal. Lastly, if it is labeled being abnormal, DeepLog's workflow model provides semantic information for users to diagnose the anomaly. Execution patterns may change over time or were not included in the original training data. DeepLog also provides the option for collecting user feedback. If the user reports a detected anomaly as false positive, DeepLog could use it as a labeled record to incrementally update its models to incorporate and adapt to the new pattern.

2.3 Threat model

DeepLog learns the comprehensive and intricate correlations and patterns embedded in a sequence of log entries produced by normal system execution paths. Henceforth, we assume that system logs themselves are *secure and protected*, and an adversary cannot attack the integrity of a log itself. We also assume that an adversary cannot modify the system source code to change its logging behavior and patterns. That said, broadly speaking, there are two types of attacks that we consider.

(1) Attacks that lead to system execution misbehavior and hence anomalous patterns in system logs. For instance, Denial of Service (DoS) attacks which may cause slow execution and hence performance anomalies reflected in the log timestamp differences from the parameter value vector sequence; attacks causing repeated server restarts such as Blind Return Oriented Programming (BROP) attack [5] shown as too many server restart log keys; and any attack that may cause task abortion such that the corresponding log sequence ends early and/or exception log entries appear.

(2) Attacks that could leave a trace in system logs due to the logging activities of system monitoring services. An example is suspicious activities logged by an Intrusion Detection System (IDS).

3 ANOMALY DETECTION

3.1 Execution path anomaly

We first describe how to detect execution path anomalies using the log key sequence. Since the total number of distinct print statements (that print log entries) in a source code is constant, so is the total number of distinct log keys. Let $K = \{k_1, k_2, \ldots, k_n\}$ be the set of distinct log keys from a log-producing system source code.

Once log entries are parsed into log keys, the log key sequence reflects an execution path that leads to that particular execution order of the log print statements. Let m_i denote the value of the key at position i in a log key sequence. Clearly, m_i may take one of the n possible keys from K, and is strongly dependent on the most recent keys that appeared prior to m_i.

We can model anomaly detection in a log key sequence as a multi-class classification problem, where each distinct log key defines a class. We train DeepLog as a multi-class classifier over recent context. The input is a history of recent log keys, and the output is a *probability distribution over the n log keys from K*, representing the probability that the next log key in the sequence is a key $k_i \in K$.

Figure 2 summarizes the classification setup. Suppose t is the sequence id of the next log key to appear. The input for classification is a window w of the h most recent log keys. That is, $w = \{m_{t-h}, \ldots, m_{t-2}, m_{t-1}\}$, where each m_i is in K and is the log key from the log entry e_i. Note that the same log key value may appear several times in w. The output of the training phase is a model of the conditional probability distribution $\Pr[m_t = k_i|w]$ *for each* $k_i \in K (i = 1, \ldots, n)$. The detection phase uses this model to make a prediction and compare the predicted output against the observed log key value that actually appears.

Training stage. The training stage relies on a small fraction of log entries produced by normal execution of the underlying system. For each log sequence of length h in the training data, DeepLog

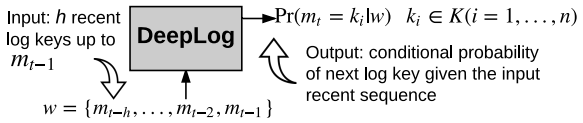


Figure 2: An overview of log key anomaly detection model.

updates its model for the probability distribution of having $k_i \in K$ as the next log key value. For example, suppose a small log file resulted from normal execution is parsed into a sequence of log keys: $\{k_{22}, k_5, k_{11}, k_9, k_{11}, k_{26}\}$. Given a window size $h = 3$, the *input sequence* and the *output label* pairs to train DeepLog will be: $\{k_{22}, k_5, k_{11} \rightarrow k_9\}$, $\{k_5, k_{11}, k_9 \rightarrow k_{11}\}$, $\{k_{11}, k_9, k_{11} \rightarrow k_{26}\}$.

Detection stage. DeepLog performs anomaly detection in an online, streaming setting. To test if an incoming log key m_t (parsed from an incoming log entry e_t) is to be considered normal or abnormal, we send $w = \{m_{t-h}, ..., m_{t-1}\}$ to DeepLog as its input. The output is a probability distribution $\Pr[m_t|w] = \{k_1 : p_1, k_2 : p_2, ..., k_n : p_n\}$ describing the probability for each log key from K to appear as the next log key value given the history.

In practice, multiple log key values may appear as m_t. For instance, if the system is attempting to connect to a host, then m_t could either be '*Waiting for * to respond*' or '*Connected to **'; both are normal system behavior. DeepLog must be able to learn such patterns during training. Our strategy is to sort the possible log keys K based on their probabilities $\Pr[m_t|w]$, and treat a key value as normal if it's among the top g candidates. A log key is flagged as being from an abnormal execution otherwise.

3.1.1 Traditional N-gram language model. The problem of ascribing probabilities to sequences of words drawn from a fixed vocabulary is the classic problem of *language modeling*, widely studied by the natural language processing (NLP) community [24]. In our case, each log key can be viewed as a word taken from the vocabulary K. The typical language modeling approach for assigning probabilities to arbitrarily long sequences is the N-gram model. The intuition is that a particular word in a sequence is only influenced by its recent predecessors rather than the entire history. In our setting, this approximation is equivalent to setting $\Pr(m_t = k_i|m_1, \ldots, m_{t-1}) = \Pr(m_t = k_i|m_{t-N}, \ldots, m_{t-1})$ where N denotes the length of the recent history to be considered.

For training, we can calculate this probability using relative frequency counts from a large corpus to give us maximum likelihood estimates. Given a long sequence of keys $\{m_1, m_2, \ldots, m_t\}$, we can estimate the probability of observing the i^{th} key k_i using the relative frequency counts of $\{m_{t-N}, \ldots, m_{t-1}, m_t = k_i\}$ with respect to the sequence $\{m_{t-N}, \ldots, m_{t-1}\}$. In other words, $\Pr(m_t = k_i|m_1, \ldots, m_{t-1}) = \text{count}(m_{t-N}, \ldots, m_{t-1}, m_t = k_i)/\text{count}(m_{t-N}, \ldots, m_{t-1})$. Note that we will count these frequencies using a sliding window of size N over the entire key sequence.

To apply the N-gram model in our setting, we simply use N as the history window size, i.e., we set $h = N$ in our experiments when the N-gram model is used where h is the history sliding window size as depicted in Figure 2. We use this as a baseline method.

3.1.2 The LSTM approach. In recent years, *neural* language models that use recurrent neural networks have been shown to be highly effective across various NLP tasks [3, 25]. Compared to a N-gram language model, a LSTM-based one can encode more intricate patterns and maintain long-range state over a sequence [34]. Complex patterns and interleaving log entries from concurrent tasks in a system log can render a traditional language model less effective. Thus, DeepLog uses a LSTM neural network [18] for anomaly detection from a log key sequence.

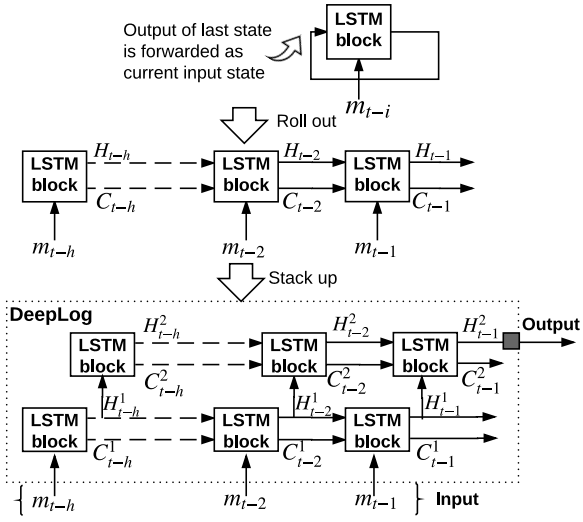


Figure 3: A detailed view of log key anomaly detection model using stacked LSTM.

Given a sequence of log keys, a LSTM network is trained to maximize the probability of having $k_i \in K$ as the next log key value as reflected by the training data sequence. In other words, it learns a probability distribution $\Pr(m_t = k_i|m_{t-h}, \ldots, m_{t-2}, m_{t-1})$ that maximizes the probability of the training log key sequence.

Figure 3 illustrates our design. The top of the figure shows a single LSTM block that reflects the recurrent nature of LSTM. Each LSTM block remembers a state for its input as a vector of a fixed dimension. The state of an LSTM block from the previous time step is also fed into its next input, together with its (external) data input (m_{t-i} in this particular example), to compute a new state and output. This is how historical information is passed to and maintained in a single LSTM block.

A series of LSTM blocks form an unrolled version of the recurrent model in one layer as shown in the center of Figure 3. Each cell maintains a hidden vector H_{t-i} *and* a cell state vector C_{t-i}. Both are passed to the next block to initialize its state. In our case, we use one LSTM block for each log key from an input sequence w (a window of h log keys). Hence, a single layer consists of h unrolled LSTM blocks.

Within a single LSTM block, the input (e.g. m_{t-i}) and the previous output (H_{t-i-1}) are used to decide (1) how much of the previous cell state C_{t-i-1} to retain in state C_{t-i}, (2) how to use the current input and the previous output to influence the state, and (3) how to construct the output H_{t-i}. This is accomplished using a set of gating functions to determine state dynamics by controlling the amount of information to keep from input and previous output, and the information flow going to the next step. Each gating function is parameterized by a set of weights to be learned. The expressive capacity of an LSTM block is determined by the number of memory

units (i.e. the dimensionality of the hidden state vector H). Due to space constraints, we refer the reader to NLP primers (e.g., [12]) for a formal characterization of LSTMs.

The training step entails finding proper assignments to the weights so that the final output of the sequence of LSTMs produces the desired label (output) that comes with inputs in the training data set. During the training process, each input/output pair incrementally updates these weights, through loss minimization via gradient descent. In DeepLog, an input consists of a window w of h log keys, and an output is the log key value that comes right after w. We use the *categorical cross-entropy loss* for training.

After training is done, we can predict the output for an input ($w = \{m_{t-h}, \ldots, m_{t-1}\}$) using a layer of h LSTM blocks. Each log key in w feeds into a corresponding LSTM block in this layer.

If we stack up multiple layers and use the hidden state of the previous layer as the input of each corresponding LSTM block in the next layer, it becomes a deep LSTM neural network, as shown at the bottom of Figure 3. For simplicity, it omits an input layer and an output layer constructed by standard encoding-decoding schemes. The input layer encodes the n possible log keys from K as one-hot vectors. That is, a sparse n-dimensional vector $\overrightarrow{u}_i$ is constructed for the log key $k_i \in K$, such that $\overrightarrow{u}_i[i] = 1$ and $\overrightarrow{u}_i[j] = 0$ for all other $j \neq i$. The output layer translates the final hidden state into a probability distribution function using a standard multinomial logistic function to represent $\Pr[m_t = k_i|w]$ for each $k_i \in K$.

The example at the bottom of Figure 3 shows only two hidden layers, but more layers can be used.

3.2 Parameter value and performance anomaly

The log key sequence is useful for detecting execution path anomalies. However, some anomalies are not shown as a deviation from a normal execution path, but as an irregular parameter value. These parameter value vectors (for the same log key) form a parameter value vector sequence, and these sequences from different log keys form a multi-dimensional feature space that is important for performance monitoring and anomaly detection.

Baseline approach. A simple approach is to store all parameter value vector sequences into a matrix, where each column is a parameter value sequence from a log key k (note that it is possible to have multiple columns for k depending on the size of its parameter value vector). Row i in this matrix represents a time instance t_i.

Consider the log entries in Table 1 as an example. There are 3 distinct log key values in this example, and the sizes of their parameter value vectors are 2, 2, and 1 respectively. Hence, row 1 in this matrix represents time instance t_1 with values [$t_1 - t_0$, file1Id, null, null, null]. Similarly, row 2 and row 3 are [null, null, $t_2 - t_1$, 0.61, null] and [null, null, null, null, $t_3 - t_2$] respectively.

We may also ask each row to represent a range of time instances so that each row corresponds to multiple log messages within that time range and becomes less sparse. But the matrix will still be very sparse when there are many log key values and/or exists some large parameter value vectors. Furthermore, this approach introduces a delay to the anomaly detection process, and it is also difficult to figure out a good value for the length of each range.

Given this matrix, many well-known data-driven anomaly detection methods can be applied, such as principal component analysis (PCA) and self-organizing maps (SOM). They are useful towards capturing *correlation* among different feature dimensions. However, a major limitation of this method in the context of log data is that often times the appearance of multiple log keys at a particular time instance is equally likely. For instance, the order of k_1 and k_2 in Table 1 is arbitrary due to concurrently running tasks. This phenomena, and the fact that the matrix is sparse, render these techniques ineffective in our setting. Lastly, they are not able to model *auto-correlation* that exists in a parameter value vector sequence (regular patterns over time in a single vector sequence).

Our approach. DeepLog trains a parameter value anomaly detection model by viewing each parameter value vector sequence (for a log key) as a separate time series.

Consider the example in Table 1. The time series for the parameter value vector sequence of k_2 is: $\{[t_2 - t_1, 0.61], [t'_2 - t'_1, 1]\}$. Hence, our problem is reduced to anomaly detection from a multi-variate time series data. It is possible to apply an LSTM-based approach again. We use a similar LSTM network as shown in Figure 3 to model a multi-variate time series data, with the following adjustments. Note that a separate LSTM network is built for the parameter value vector sequence of each distinct log key value.

Input. The input at each time step is simply the parameter value vector from that timestamp. We normalize the values in each vector by the average and the standard deviation of all values from the same parameter position from the training data.

Output. The output is a real value vector as a prediction for the next parameter value vector, based on a sequence of parameter value vectors from recent history.

Objective function for training. For the multi-variate time series data, the training process tries to adjust the weights of its LSTM model in order to minimize the error between a prediction and an observed parameter value vector. Thus, mean square loss is used to minimize the error during the training process.

Anomaly detection. The difference between a prediction and an observed parameter value vector is measured by the mean square error (MSE). Instead of setting a magic error threshold for anomaly detection purpose in an ad-hoc fashion, we partition the training data to two subsets: the model training set and the validation set. For each vector $\overrightarrow{v}$ in the validation set, we apply the model produced by the training set to calculate the MSE between the prediction (using the vector sequence from before $\overrightarrow{v}$ in the validation set) and $\overrightarrow{v}$. At every time step, the errors between the predicted vectors and the actual ones in the validation group are modeled as a Gaussian distribution.

At deployment, if the error between a prediction and an observed value vector is within a high-level of confidence interval of the above Gaussian distribution, the parameter value vector of the incoming log entry is considered normal, and is considered abnormal otherwise.

Since parameter values in a log message often record important system state metrics, this method is able to detect various types of performance anomalies. For example, a performance anomaly may reflect as a "slow down". Recall that DeepLog stores in each parameter value vector the time elapsed between consecutive log entries. The above LSTM model, by modeling parameter value vector as a multi-variate time series, is able to detect unusual patterns in one

or more dimensions in this time series; the elapsed time value is just one such dimension.

3.3 Online update of anomaly detection models

Clearly, the training data may not cover all possible normal execution patterns. System behavior may change over time, additionally depending on workload and data characteristics. Therefore, it is necessary for DeepLog to incrementally update weights in its LSTM models to incorporate and adapt to new log patterns. To do this, DeepLog provides a mechanism for the user to provide feedback. This allows DeepLog to use a false positive to adjust its weights. For example, suppose $h = 3$ and the recent history sequence is $\{k_1, k_2, k_3\}$, and DeepLog has predicted the next log key to be k_1 with probability 1, while the next log key value is k_2, which will be labeled as an anomaly. If user reports that this is a false positive, DeepLog is able to use the following input-output pair $\{k_1, k_2, k_3 \rightarrow k_2\}$ to update the weights of its model to learn this new pattern. So that next time given history sequence $\{k_1, k_2, k_3\}$, DeepLog can output both k_1 and k_2 with updated probabilities. The same update procedure works for the parameter value anomaly detection model. Note that DeepLog does not need to be re-trained from scratch. After the initial training process, models in DeepLog exist as several multi-dimensional weight vectors. The update process feeds in new training data, and adjusts the weights to minimize the error between model output and actual observed values from the false positive cases.

4 WORKFLOW CONSTRUCTION FROM MULTI-TASKS EXECUTION

Each log key is the execution of a log printing statement in the source code, while a task like `VM creation` will produce a sequence of log entries. Intuitively, the order of log entries produced by a task represents an execution order of each function for accomplishing this task. As a result, we can build a workflow model as a finite state automaton (FSA) to capture the execution path of any task. This workflow model can also be used to detect execution path anomalies, but it is less effective compared to DeepLog's LSTM model due to its inability to capture inter-task dependencies and non-deterministic loop iterations. However, the workflow model is very useful towards enabling users to *diagnose* what had gone wrong in the execution of a task when an anomaly has been detected.

Given a log sequence generated by the repeated executions of a task, there have been several works exploring the problem of workflow inference [4, 21, 42]. CloudSeer [42] represents the state of the art in anomaly detection using a workflow model. CloudSeer has several limitations. Firstly, the anomalies it can detect are limited to log entries having "ERROR" logging level and log entries not appearing. Furthermore, its workflow model construction requires a log file with repeated executions of only one single task. Other previous works [4, 21] on workflow construction from a log file also suffer from this limitation. In practice, a log file often contains interleaving log entries produced by multiple tasks and potentially concurrently running threads within a task.

4.1 Log entry separation from multiple tasks

An easy case is when multiple programs concurrently write to the same log (e.g., Ubuntu's system log). Often each log entry contains the name of the program that created it. Another easy case is when the process or task id is included in a log entry. Here, we focus on the case where a user program is executed repeatedly to perform different, but logically related, tasks within that program. An important observation is that tasks do not overlap in time. However, the same log key may appear in more than one task, and concurrency is possible within each task (e.g., multiple threads in one task).

Consider OpenStack administrative logs as an example. For each VM instance, its life cycle contains `VM creation`, `VM stop`, `VM deletion` and others. These tasks do not overlap, i.e., `VM stop` can only start after `VM creation` has completed. However, the same log key may appear in different tasks. For example, a log message *"VM Resumed (Lifecycle Event)"* may appear in `VM creation`, `VM start`, `VM resume` and `VM unpause`. There could be concurrently running threads inside each task, leading to uncertainty in the ordering of log messages corresponding to one task. For instance, during `VM creation`, the order of two log messages *"Took * seconds to build instance"* and *"VM Resumed (Lifecycle Event)"* is uncertain.

Our goal is to separate log entries for different tasks in a log file, and then build a workflow model for each task based on its log key sequence. That said, the input of our problem is the entire log key sequence parsed from a raw log file, and the output is a set of workflow models, one for each task identified.

4.2 Using DeepLog's anomaly detection model

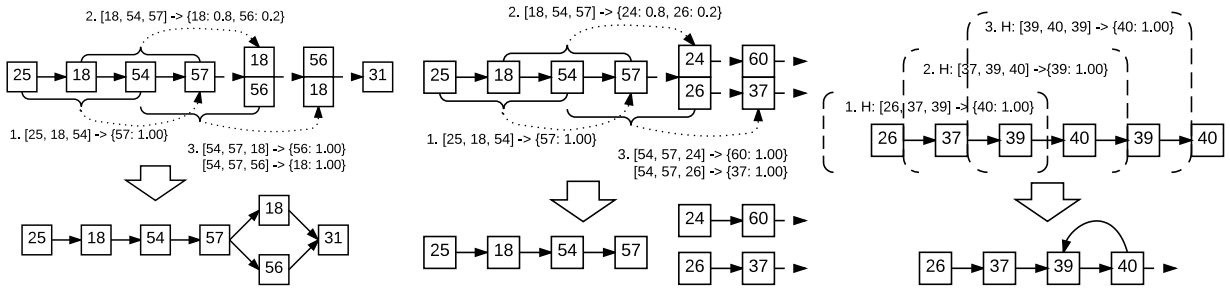
4.2.1 Log key separation. Recall that in DeepLog's model for anomaly detection from log keys, the input is a sequence of log keys of length h from recent history, and the output is a probability distribution of all possible log key values. An interesting observation is that its output actually encodes the underlying workflow execution path.

Intuitively, given a log key sequence, our model predicts what will happen next based on the execution patterns it has observed during the training stage. If a sequence w is never followed by a particular key value k in the training stage, then $\Pr[m_t = k|w] = 0$. Correspondingly, if a sequence w is *always* followed by k, then $\Pr[m_t = k|w] = 1$. For example, suppose on a sequence "25→54", the output prediction is "{57:1.00}", we know that "25→54→57" is from one task. A more complicated case is when a sequence w is to be followed by a log key value from a group of different keys; the probabilities of these keys to appear after w sum to 1.

To handle this case, we use an idea that is inspired by small invariants mining [21].

Consider a log sequence "54→57", and suppose the predicted probability distribution is "{18: 0.8, 56: 0.2}", which means that the next step could be either "18" or "56". This ambiguity could be caused by using an insufficient history sequence length. For example, if two tasks share the same workflow segment "54→57", the first task has a pattern "18→54→57→18" that is executed 80% of the time, and the second task has a pattern "31→54→57→56" that is executed 20% of the time. This will lead to a model that predicts "{18: 0.8, 56: 0.2}" given the sequence "54→57".

We can address this issue by training models with different history sequence lengths, e.g., using $h = 3$ instead of $h = 2$ in this case. During workflow construction, we use a log sequence length that leads to a more certain prediction, e.g. in the above example the



(a) An example of concurrency detection **(b) An example of new task detection** **(c) An example of loop identification**

Figure 4: Examples of using LSTM for task separation and workflow construction.

sequence "18→54→57" will lead to the prediction {18: 1.00} and the sequence "31→54→57" will lead to the prediction {56: 1.00}.

If we have ruled out that a small sequence is a shared segment from different tasks (i.e., increasing the sequence length for training and prediction doesn't lead to more certain prediction), the challenge now is to find out whether the multi-key prediction output is caused by *either* concurrency in the same task *or* the start of a different task. We call this a *divergence point*.

We observe that, as shown in Figure 4a, if the divergence point is caused by concurrency in the same task, a common pattern is that keys with the highest probabilities in the prediction output will appear one after another, and the certainty (measured by higher probabilities for less number of keys) for the following predictions will increase, as keys for some of the concurrent threads have already appeared. The prediction will eventually become certain after all keys from concurrent threads are included in the history sequence.

On the other hand, if the divergence point is caused by the start of a new task, as shown in Figure 4b, the predicted log key candidates ("24" and "26" in the example) will not appear one after another. If we incorporate each such log key into the history sequence, the next prediction is a deterministic prediction of a new log key (e.g., "24→60", "26→37"). If this is the case, we stop growing the workflow model of the current task (stop at key "57" in this example), and start constructing workflow models for new tasks. Note that the two "new tasks" in Figure 4b could also be an "if-else" branch, e.g., "57→if (24→60→...) else (26→37→...)". To handle such situations, we apply a simple heuristic: if the "new task" has very few log keys (e.g., 3) and always appears after a particular task T_p, we treat it as part of an "if-else" branch of T_p, otherwise as a new task.

4.2.2 Build a workflow model. Once we can distinguish divergence points caused by concurrency (multiple threads) in the same task and new tasks, we can easily construct workflow models as illustrated in Figure 4a and Figure 4b. Additional care is needed to identify loops. The detection of a loop is actually quite straightforward. A loop is always shown in the initial workflow model as an unrolled chain; see Figure 4c for an example. While this workflow chain is initially "26→37→39→40→39→40", we could identify the repeated fragments as a loop execution (39→40 in this example).

4.3 Using density-based clustering approach

4.3.1 Log key separation. Another approach is to use a density-based clustering technique. The intuition is that log keys in the same task always appear together, but log keys from different tasks may not always appear together as the ordering of tasks is not fixed during multiple executions of different tasks. This allows us to cluster log keys based on co-occurrence patterns, and separate keys into different tasks when co-occurrence rate is low.

Table 2: Co-occurrence matrix within distance d

	k_1	...	k_j	...	k_n
k_1	$p_d(1,1)$		$p_d(1,j)$		
...					
k_i	$p_d(i,1)$		$p_d(i,j) = \frac{f_d(k_i,k_j)}{d \cdot f(k_i)}$		
...					
k_n	$p_d(n,1)$		$p_d(n,j)$		

In a log key sequence, the distance d between any two log keys is defined as the number of log keys between them plus 1. For example, given the sequence $\{k_1, k_2, k_2\}$, $d(k_1, k_2) = [1, 2], d(k_2, k_2) = 1$ (note that there are two distance values between the pair (k_1, k_2)).

We build a co-occurrence matrix as shown in Table 2, where each element $p_d(i,j)$ represents the probability of two log keys k_i and k_j appearing within distance d in the input sequence. Specifically, let $f(k_i)$ be the frequency of k_i in the input sequence, and $f_d(k_i, k_j)$ be the frequency of pair (k_i, k_j) appearing together within distance d in the input sequence. We define $p_d(i,j) = \frac{f_d(k_i,k_j)}{d \cdot f(k_i)}$, which shows the *importance* of k_j to k_i.

For example, when $d = 1$, $p_1(i,j) = \frac{f_1(k_i,k_j)}{f(k_i)} = 1$ means that for every occurrence of k_i, there must be a k_j next to it. Note that in this definition, $f(k_i)$ in the denominator is scaled by d because while counting co-occurrence frequencies within d, a key k_i is counted by d times. Scaling $f(k_i)$ by a factor of d ensures that $\sum_{j=1}^{n} f_d(i,j) = 1$ for any i. Note that we can build multiple co-occurrence matrices for different distance values of d.

With a co-occurrence matrix for each distance value d that we have built, our goal is to output a set of tasks $TASK = (T_1, T_2, ...)$. The clustering procedure works as follows. First, for $d = 1$, we check if any $p_1(i,j)$ is greater than a threshold τ (say $\tau = 0.9$), when it does, we connect k_i, k_j together to form $T_1 = [k_i, k_j]$. Next, we recursively check if T_1 could be extended from either its head or tail. For example, if there exists $k_x \in K$ such that $p_1(k_i, k_x) > \tau$, we further check if $p_2(k_j, k_x) > \tau$, i.e., if k_j and k_x have a large co-occurrence probability within distance 2. If yes, $T_1 = [k_x, k_i, k_j]$, otherwise we will add $T_2 = [k_i, k_x]$ to $TASK$.

This procedure continues until no task T in $TASK$ could be further extended. In the general case when a task T to be extended

has more than 2 log keys, when checking if k_x could be included as the new head or tail, we need to check if k_x has a co-occurrence probability greater than τ with each log key in T up to distance d', where d' is the smaller of: i) length of T, and ii) the maximum value of d that we have built a co-occurrence matrix for. For example, to check if $T = [k_1, k_2, k_3]$ should connect k_4 at its tail, we need to check if $\min(p_1(k_3, k_4), p_2(k_2, k_4), p_3(k_1, k_4)) > \tau$.

The above process connects sequential log keys for each task. When a task $T_1 = [k_i, k_j]$ cannot be extended to include any single key, we check if T_1 could be extended by two log keys, i.e., if there exists $k_x, k_y \in K$, such that $p_1(k_i, k_x) + p_1(k_i, k_y) > \tau$, or $p_1(k_j, k_x) + p_1(k_j, k_y) > \tau$. Suppose the latter case is true, the next thing to check is whether k_x and k_y are log keys produced by concurrent threads in task T_1. If they are, $p_d(k_j, k_x)$ always increases with larger d values, i.e., $p_2(k_j, k_x) > p_1(k_j, k_x)$, which is intuitive because the appearance ordering of keys from concurrent threads is not certain. Otherwise k_x and k_y do not belong to T_1, thus we add $T_2 = [k_j, k_x]$ and $T_3 = [k_j, k_y]$ into $TASK$ instead.

Finally, for each task T in $TASK$, we eliminate T if its sequence is included as a sub-sequence in another task.

4.3.2 Build workflow model. Once a log key sequence is separated out and identified for each task, the workflow model construction for a task follows the same discussion from Section 4.2.2.

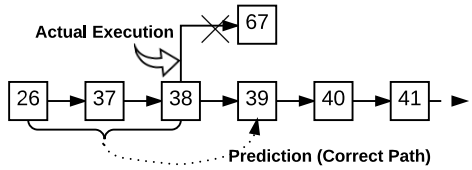
4.4 Using the workflow model

4.4.1 Set parameters for DeepLog model. In Section 3.1, we've shown that DeepLog requires several input parameters, in particular, it needs the length of history sequence window h (for training and detection), and the number of top g log keys in the predicted output probability distribution function to be considered normal.

Setting a proper value for h and g is problem dependent. Generally speaking, larger h values will increase the prediction accuracy because more history information is utilized in LSTM, until it reaches a point where keys that are far back in history do not contribute to the prediction of keys to appear. At this point continuing to increase h does not hurt the prediction accuracy of LSTM, because LSTM is able to learn that only the recent history in a long sequence matters thus ignore the long tail. However, a large h value does have a performance impact. More computations (and layers) are required for both training and prediction, which slows down the performance of DeepLog. The value of g, on the other hand, regulates the trade-off between true positives (anomaly detection rate) and false positives (false alarm rate).

The workflow model provides a guidance to set a proper value for both h and g. Intuitively, h needs to be just large enough to incorporate necessary dependencies for making a good prediction, so we can set h as the length of the shortest workflow. The number of possible execution paths represents a good value for g, hence, we set g as the maximum number of branches at all divergence points from the workflows of all tasks.

4.4.2 Using workflow to diagnose detected anomalies. Whenever an anomaly is detected by DeepLog, the workflow model can be used to help diagnose this anomaly and understand how and why it has happened. Figure 5 shows an example. Using a history sequence [26, 37, 38], the top prediction from DeepLog is log key 39 (suppose $g = 1$), however the actual log key appeared is 67, which is an anomaly. With the help of a workflow model for this task, users could easily identify the current execution point in the corresponding workflow, and further discover that this error happened right after *"Instance destroyed successfully"* and before *"Deleting instance files *"*, which means that this error occurred during cleanup after destroying a VM.



37: instance: * Terminating instance
38: instance: * Instance destroyed successfully
39: instance: * Deleting instance files *
40: instance: * Deletion of * complete
41: instance: * Took * seconds to destroy the instance on the hypervisor
67: instance: * Error from libvirt during unfilter. Code=* Error=*

Figure 5: Anomaly diagnosis using workflow.

4.5 Discussion

Previous works [4, 21, 42] focused on constructing workflows from multiple executions of just *one task*. The basic idea in their approach follows 3 steps: 1) mine temporal dependencies of each pair of log keys; 2) construct a basic workflow from the pairwise invariants identified in step 1; 3) refine workflow model using the input log key sequence. A major limitation is that they are not able to work with a log sequence that contains multiple tasks or concurrent threads in one task, which is addressed by our study. Our task separation methodology also provides useful insights towards the workflow construction for each task.

5 EVALUATION

DeepLog is implemented using Keras [6] with TensorFlow [2] as the backend. In this section, we show evaluations of each component and the overall performance of DeepLog, to show its effectiveness in finding anomalies from large system log data.

5.1 Execution path anomaly detection

This section focuses on evaluating the *log key anomaly detection* model in DeepLog. We first compare its effectiveness on large system logs with previous methods, and then investigate the impact of different parameters in DeepLog.

5.1.1 Previous methods. Previous work on general-purpose log anomaly detection follows a similar procedure: they first extract a log key from each log message, and then perform anomaly detection on the log key sequence.

The *Principal Component Analysis (PCA)* method [39] assumes that there are different "sessions" in a log file that can be easily identified by a session id attached to each log entry. It first groups log keys by session and then counts the number of appearances of each log key value inside each session. A session vector is of size n, representing the number of appearances for each log key in K in that session. A matrix is formed where each column is a log key, and each row is one session vector. PCA detects an abnormal vector (a session) by measuring the projection length on the residual subspace of transformed coordinate system. This approach is shown to be more effective than its online counterpart

online PCA [38] especially in reducing false positives, but this is clearly an offline method and cannot be used for online anomaly detection. The implementation is open-sourced by [17].

Invariant Mining (IM) [22] constructs the same matrix as the PCA approach does. IM first mines small invariants that could be satisfied by the majority of vectors, and then treats those vectors that do not satisfy these invariants as abnormal execution sessions. This approach is shown to be more effective than an earlier work [11] which utilizes workflow automata. The implementation is open-sourced by [17].

TFIDF is developed in [44]. Although its objective is for IT system failure prediction, which is different from anomaly detection as shown in [39]. Nevertheless, we still included this method in our evaluation as it also uses a LSTM-based approach. There are several key differences. TFIDF groups log keys by time windows (each time window is defined by a user parameter), and then models each time window (called "epoch") using a TF-IDF (term-frequency, inverse document frequency) vector. The Laplace smoothing procedure it uses requires the knowledge of the total number of epochs (hence the entire log file). TFIDF constructs a LSTM model as a *binary classifier*, which needs both *labeled normal and abnormal* data for training. Not only are anomaly log entries hard to obtain, but also, new types of anomalies that are not included in training data may not be detected. In contrast, DeepLog trains its LSTM model to be a *multi-class classifier*, and *only requires normal data to train*.

CloudSeer is a method designed *specifically for multi-user OpenStack log* [42]. It builds a workflow model for each OpenStack VM-related task and uses the workflow for anomaly detection. Though it achieves acceptable performance on OpenStack logs (a precision of 83.08% and a recall of 90.00% as reported in the paper), this method does not work for other types of logs (e.g., HDFS log) where the patterns of log keys are much more irregular. For example, CloudSeer only models log keys that *"appear the same number of times"* in every session. In HDFS logs, only 3 out of 29 log keys satisfy this criterion. Furthermore, this method cannot separate log entries for different tasks in one log into separate sequences. It relies on multiple identifiers to achieve this, which is not always possible for general-purpose logs. Thus it is not compared against here.

5.1.2 Log data sets and set up.

HDFS log data set. It is generated through running Hadoop-based map-reduce jobs on more than 200 Amazon's EC2 nodes, and labeled by Hadoop domain experts. Among 11, 197, 954 log entries being collected, about 2.9% are abnormal, including events such as "write exception". This was the main data set firstly used by an offline PCA-based [39] method, and subsequently used by several other work including online PCA [38] and IM-based [22] methods. Details of this dataset could be found in [38, 39].

OpenStack log data set. We deployed an OpenStack experiment (version Mitaka) on CloudLab [30] with one control node, one network node and eight compute nodes. Among 1, 335, 318 log entries collected, about 7% are abnormal. A script was running to constantly execute VM-related tasks, including VM creation/deletion, stop/start, pause/unpause and suspend/resume. VM tasks were scheduled with the pattern of a regular expression *(Create (Stop Start) {0,3} (Pause Unpause) {0,3} (Suspend Resume) {0,3} Delete)+*. A VM life cycle starts with "VM create" and ends with "VM delete", while task pairs such as "Stop-Start", "Pause-Unpause" and "Suspend-Resume" may randomly appear from 0 to 3 times within a life cycle. `INFO` level logs from nova-api, nova-scheduler and nova-compute were collected and forwarded for analysis using Elastic Stack [33]. Three types of anomalies were injected at different execution points: 1) neutron timeout during VM creation; 2) libvirt error while destroying a VM; 3) libvirt error during cleanup after destroying a VM.

Set up. To execute PCA-based and IM-based methods, we group log entries into different sessions by an identifier field, which for HDFS log is `block_id` and for OpenStack log is `instance_id`. Each session group is a life cycle of one block or a VM instance respectively. We then parse each log entry into a log key. DeepLog can be applied directly on log keys to train its weights and subsequently be used to detect anomalies, while other methods require one more step. They need to count the number of appearances for each distinct log key within each session, and build a matrix where each column is a distinct log key (so there will be n columns) and each row represents a session vector, and the value of a cell V_{ij} in the matrix represents the count of log key k_j in the i-th session.

DeepLog needs a small fraction of normal log entries to train its model. In the case of HDFS log, only less than 1% of normal sessions (4,855 sessions parsed from the first 100,000 log entries compared to a total of 11,197,954) are used for training. Note that DeepLog can pinpoint which log entry (with its corresponding log key) is abnormal, but in order to use the same measures to compare with competing methods, we use "session" as the granularity of anomaly detection, i.e., a session C is considered an abnormal session as long as there exists at least one log key from C being detected abnormal.

Table 3 summarizes the two data sets. Note that PCA and IM are unsupervised offline methods that do not require training data, whereas DeepLog only needs a training data produced by normal system execution, and TFIDF requires both normal and abnormal data to train.

Log data set	Number of sessions		n: Number of log keys
	Training data (if needed)	Test data	
HDFS	4,855 normal; 1,638 abnormal	553,366 normal; 15,200 abnormal	29
OpenStack	831 normal; 50 abnormal	5,990 normal; 453 abnormal	40

Table 3: Set up of log data sets (unit: session).

In addition to the number of false positives (FP) and false negatives (FN), we also use standard metrics such as Precision, Recall and F-measure. Precision=$\frac{TP}{TP+FP}$ (TP stands for true positive) shows the percentage of true anomalies among all anomalies detected; Recall=$\frac{TP}{TP+FN}$ measures the percentage of anomalies in the data set (assume that we know the ground-truth) being detected; and F-measure=$\frac{2\cdot Precision\cdot Recall}{Precision+Recall}$ is the harmonic mean of the two.

By default, we use the following parameter values for DeepLog: $g = 9$, $h = 10$, $L = 2$, and $\alpha = 64$ and investigate their impacts in our experiments. Recall g decides the cutoff in the prediction output to be considered normal (i.e., the g log key values with top-g probabilities to appear next are considered normal), and h is the window size used for training and detection. L and α denote the number of layers in DeepLog and the number of memory units in one LSTM block respectively. For all other methods, we explored

	PCA	IM	TFIDF	N-gram	DeepLog
false positive (FP)	277	2122	95833	1360	833
false negative (FN)	5400	1217	1256	739	619

Table 4: Number of FPs and FNs on HDFS log.

their parameter space and report their best results. When the N-gram method is used, we set $N = 1$ unless otherwise specified since this shows the best performance for the N-gram method.

5.1.3 Comparison. Table 4 shows the number of false positives and false negatives for each method on HDFS data. PCA achieves the fewest false positives, but at the price of more false negatives. Figure 6a shows a more in-depth comparison using recall, precision and F-measure. Note that TFIDF is *omitted* from this figure because of limited space and its very poor relative performance.

Clearly, DeepLog has achieved the best overall performance, with an F-measure of 96%. Our baseline solution N-gram also achieves good performance when history length is 1. But, its performance drops dramatically as history window becomes longer. In contrast, LSTM-based approach is more stable as shown in Section 5.1.4.

Figure 6b investigates the top-g approach used by DeepLog's prediction algorithm. Let D_t be the set of top-g log key values predicted by DeepLog at t, and m_t be the actual log key value appeared in the data at t. To see the impact of this strategy, we study the CDF of $\Pr[m_t \in D_t]$ for different g values. Among over 11,000,000 log keys (that are labeled as normal) to predict, 88.9% of DeepLog's top prediction matches m_t exactly; and 96.1% m_t's are within DeepLog's top 2 predictions. When $g = 5$, 99.8% of normal m_t's are within D_t, meanwhile the anomaly detection rate is 99.994% (only one anomalous session was undetected).

Figure 7a shows the performance over OpenStack data set. The PCA approach shows reasonable performance on this data set but with low precision (only 77%), whereas even though IM has achieved a perfect recall in this case, it has very poor precision of only 2% (almost all VM instances are detected as abnormal executions). This is because that OpenStack logs were generated randomly as described in Section 5.1.2. Note that how many times that log keys like *(Stop Start)* may appear in a life cycle of a VM (defined by a pair of *Create* and *Delete*) is uncertain. This makes it really hard for IM to find the "stable small invariants" for anomaly detection.

To test this hypothesis, we generated a second data set with a deterministic pattern like *(Create Delete)+*, resulting in a total of 5,544 normal VM executions and 170 anomalous ones. We denote this data set as OpenStack II and the result is shown in Figure 7b. IM performs very well on this data set with more regular patterns. However the recall for the PCA method drops to only 2% in this case because the normal pattern in the data is too regular, rendering PCA method which detects anomalies by variance not working.

On the other hand, DeepLog demonstrates excellent performance on both OpenStack logs with a F-measure of 98% and 97% respectively. Lastly, it is also important to note that PCA and IM are *offline* methods, and they *cannot* be used to perform anomaly detection per log entry. They are *only able to detect anomaly at session level*, but the notion of session may not even exist in many system logs.

5.1.4 Analysis of DeepLog. We investigate the performance impact of various parameters in DeepLog including: g, h, L, and α. The results are shown in Figure 8. In each experiment, we varied the values of one parameter while using the default values for others.

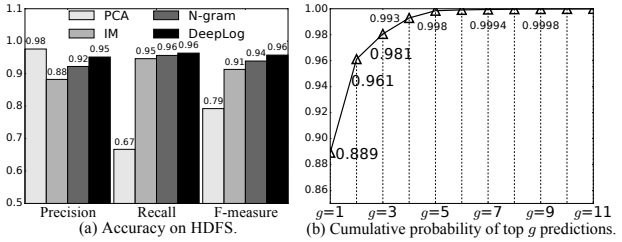


Figure 6: Evaluation on HDFS log.

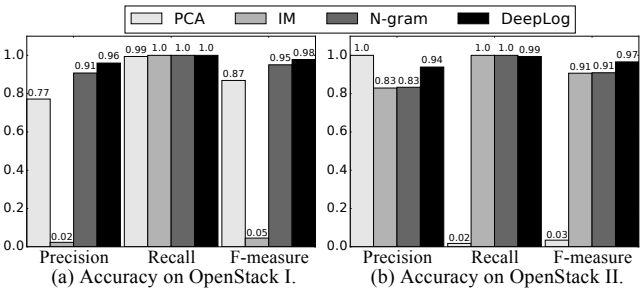


Figure 7: Evaluation on OpenStack log.

In general, the performance of DeepLog is fairly stable with respect to different values, i.e., it is not very sensitive to the adjustment of any one or combinations of these parameter values. This makes DeepLog easy to deploy and use in practice. The results are fairly intuitive to understand as well. For example, Figure 8c shows that a larger g value leads to higher precision but lower recall. Thus, g could be adjusted to achieve higher true positive rate or lower false positive rate. Lastly, DeepLog's prediction cost per log entry is only around 1 millisecond on our standard workstation, which could be further improved by better hardware such as using a GPU.

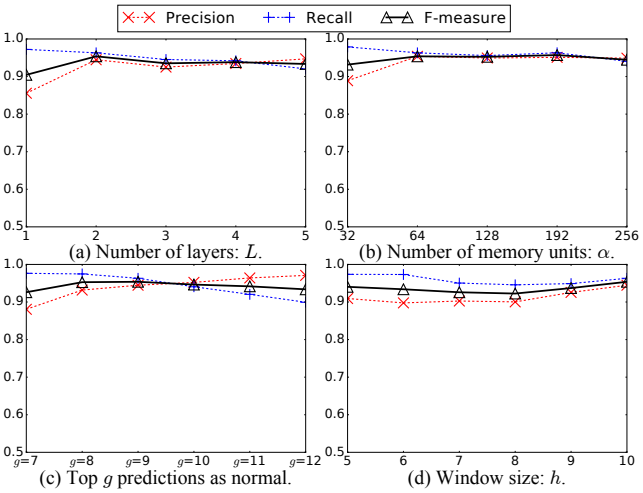


Figure 8: DeepLog performance with different parameters.

5.2 Parameter value and performance anomaly

To evaluate the effectiveness of DeepLog at detecting parameter value and performance (including elapsed time between log entries)

anomalies, we used system logs from the OpenStack VM creation task. This data set includes both types of anomalies: performance anomaly (late arrival of a log entry) and parameter value anomaly (a log entry with a much longer VM creation time than others).

Experiment setup. As before, we deployed an OpenStack experiment on CloudLab, and wrote a script to simulate that multiple users are constantly requesting VM creations and deletions. During OpenStack VM creation, an important procedure is to copy the required image from controller node to a compute node (where the VM will be created). To simulate a performance anomaly which could be possibly caused by a DoS attack, we throttle the network speed from the controller to compute nodes at two different points, to see if these anomalies could be detected by DeepLog.

Anomaly detection. As described in Section 3.2, we separate log entries into two sets, one set is for model training and the other set (called the validation set) is to apply the model to generate the Gaussian distribution of MSEs (mean square error). In subsequent online detection phase, for every incoming parameter value vector $\overrightarrow{v}$, DeepLog checks if the MSE between $\overrightarrow{v}$ and the predication output (a vector as well) from its model is within an acceptable confidence interval of the Gaussian distribution of MSEs from the validation set.

Figure 9 shows the detection results for the parameter value vectors of different log keys, where x-axis represents the id of the VM being created (i.e., different VM creation instances), and y-axis represents the MSE between the parameter value vector and the prediction output vector from DeepLog. The horizontal lines in each figure are the confidence interval thresholds for the corresponding MSE Gaussian distributions. Figure 9a and 9b represent two log keys where their parameter value vectors are normal during the entire time. Figure 9c and 9d illustrate that the parameter value vectors for keys 53 and 56 are successfully detected as being abnormal at exactly the two time instances where we throttled the network speed (i.e., injected anomalies).

For each abnormal parameter value vector detected, we identified the value that differs the most with the prediction, to identify the abnormal column (feature). We found out that the two abnormal parameter value vectors for key 53 are due to unusually large elapsed time values. On the other hand, key 56 is "*Took * seconds to build instance.*", and not surprisingly, its two abnormal parameter value vectors were caused by unusually large values (for seconds).

5.3 Online update and training of DeepLog

Section 5.1 has demonstrated that DeepLog requires a very small training set (less than 1% of the entire log) and does not require user feedback during its training phase. But it is possible that a new system execution path may show up during detection stage, which is also normal, but is detected as anomalous since it was not reflected by the training data. To address this issue, this section evaluates the effectiveness of DeepLog's online update and training module as described in Section 3.3. We demonstrate this using the difference in detection results with and without incremental updates, in terms of both effectiveness and efficiency.

5.3.1 Log data set. The log data set used in this section is Blue Gene/L supercomputer system logs [1], which contains 4,747,963 log

[1]CFDR Data, https://www.usenix.org/cfdr-data

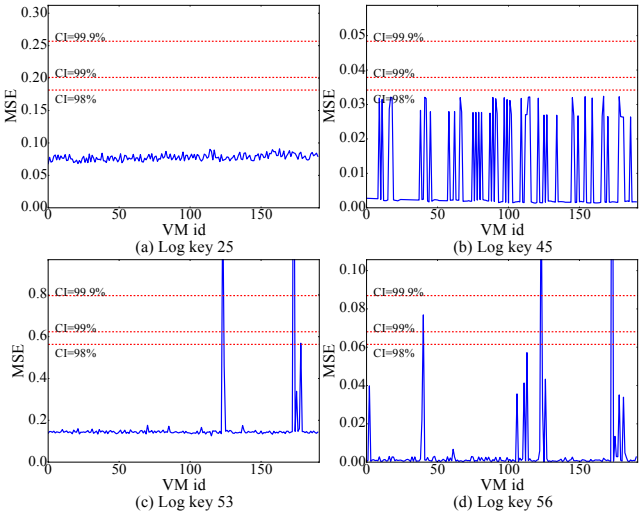


Figure 9: Anomaly detection for parameter value vectors with different confidence intervals (CIs).

entries, of which 348,460 entries are labeled as anomalies. We chose this data set because of an important characteristic: many log keys only appeared during a specific time period. This means that the training data set may not contain all possible normal log keys, let alone all possible normal execution patterns.

5.3.2 Evaluation results. We conducted two experiments, one uses the first 1% normal log entries as training data and the other uses the first 10% log entries for training. In both settings, the remaining 99% or 90% entries are used for anomaly detection. We set $L = 1$, $\alpha = 256$, $g = 6$, $h = 3$.

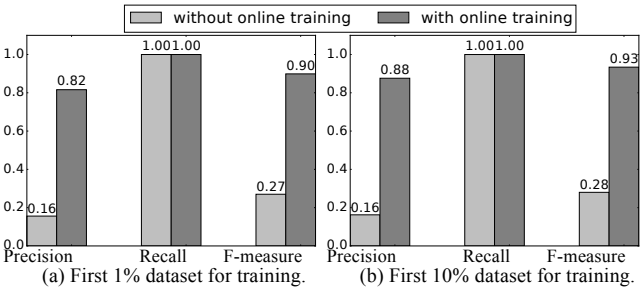


Figure 10: Evaluation on Blue Gene/L log.

Figure 10 shows the results for without and with online training for both experiments. In the case of "without online training", we run DeepLog to test incoming log entries without any incremental update. While for the case of "with online training", we assume there is an end user who reports if a detected anomaly is a false positive. If so, DeepLog uses that sample (now a labeled record) to update its model to learn this new pattern. Figure 10 shows that without online training, with only 1% offline training data, this results in many false positives (hence very low Precision rate). Though increasing its training data to 10% slightly increases the Precision, its performance is still unsatisfactory. On the other hand, DeepLog with online training significantly improves its Precision, and hence F-measure scores. With a true positive rate of 100% (perfect recall) in both settings, online training reduces false positive

rate from 40.1% to 1.7% for 1% training data, and from 38.2% to 1.1% for 10% training data, respectively.

Table 5 shows the amortized cost to check each log entry. For the online training case, we reported time taken for both detection and online update (if an update is triggered). The results show that online update and training does increase the amortized cost per log entry, but only slightly. This is because many log entries will not trigger an update. Note that online update and online detection can be executed in parallel; an update is carried out while the model is using the current weights to continue performing detection.

Table 5: Amortized cost to check each log entry

training data percentage	1%	10%
without online training (milliseconds)	1.06	1.11
with online training (milliseconds)	3.48	2.46

5.4 Security log case studies

Anomalies having log keys that never showed up in normal logs used for training (e.g., "ERROR" or "exception" log messages) are easy to detect. DeepLog can effectively detect much more subtle cases. For example, in HDFS log, *"Namenode not updated after deleting block"* anomaly is shown as a missing log key in a session; and *"Redundant addStoredBlock"* anomaly is shown as an extra log key. This means that for any attack that may cause any change of system behavior (as reflected through logs), it can be detected. In what follows, we investigate system logs containing real attacks to demonstrate the effectiveness of DeepLog.

5.4.1 Network security log. Network security is of vital importance. Both firewall and intrusion detection system (IDS) produce logs that can be used for online anomaly detection.

To test the performance of DeepLog on network security logs, we used the VAST Challenge 2011 data set, specifically, Mini Challenge 2 — Computer Networking Operations [1]. This challenge is to *manually* look for suspicious activities by visualization techniques. It comes with ground truth for anomalous activities. For all anomalies in the ground truth, Table 6 shows the results of DeepLog. The only suspicious activity not being detected is the first appearance of an undocumented computer IP address.

Table 6: VAST Challenge 2011 network security log detection

suspicious activity	**detected?**
Day 1: Denial of Service attack	Yes, log key anomaly in IDS log
Day 1: port scan	Yes, log key anomaly in IDS log
Day 2: port scan 1	Yes, log key anomaly in IDS log
Day 2: port scan 2	Yes, log key anomaly in IDS log
Day 2: socially engineered attack	Yes, log key anomaly in firewall log
Day 3: undocumented IP address	No

The only false positive case happened when DeepLog reported a log message that repeatedly appeared many times in a short period as an anomaly. This is due to an event that suddenly became bursty and printed the same log message many times in a short time range. This is not identified by the VAST Challenge as a suspicious activity.

5.4.2 BROP attack detection. Blind Return Oriented Programming (BROP) attack [5] leverages a fact that many server applications restart after a crash to ensure service reliability. This kind of attack is powerful and practical because the attacker neither relies on access to source code nor binary. A stack buffer overflow vulnerability, which leads server to crash, is sufficient to carry out this attack. In a BROP exploit, the attacker uses server crash as a signal to help complete a ROP attack which achieves executing a shellcode. However, the repeated server restarting activities leave many atypical log messages in kernel log as shown below, which is easily detected by DeepLog.

```
nginx[*]: segfault at * ip * sp * error * in nginx[*]
nginx[*]: segfault at * ip * sp * error * in nginx[*]
nginx[*]: segfault at * ip * sp * error * in nginx[*]
......
```

5.5 Task separation and workflow construction

We implemented the proposed methods in Section 4 and evaluated on a log with various OpenStack VM-related tasks. Both LSTM approach and density-based clustering approach could successfully separate all tasks. The first method requires LSTM; it is a supervised method which requires training data to be provided. The second method uses clustering on co-occurrences of log keys within a certain distance threshold, which is an unsupervised method. Hence, it doesn't require training, but it does require parameter τ as the distance threshold.

Specifically, for density-based clustering approach, with a sufficiently large threshold value $\tau \in [0.85, 0.95]$, there is a clear separation of all tasks. Note that the value of τ cannot be too large (e.g., setting $\tau = 1$), as a background process may produce log entries at random locations that will break log entries from the same task apart.

Next we use a part of the `VM creation` workflow, to show how it provides useful diagnosis of the performance anomaly in Section 5.2. Recall in Section 5.2, parameter value vector anomaly is identified on the time elapsed value of log key 53, and on the parameter position of log key 56 (which represents how many seconds to build instance). As shown in Figure 11, once an anomaly is detected by DeepLog, we know the time taken to build that instance is abnormal, but we don't know why. Then, since the elapsed time between log key 53 and its previous log key is too big, by investigating the workflow model constructed by DeepLog, its previous key is 52: "*Creating image*", so we know that VM creation took longer time than usual because the time to create image was too long. Further investigation following this procedure may reveal that it was caused by slow network speed from control node to compute node.

6 RELATED WORK

Primarily designed for recording notable events to ease debugging, system event logs are abundantly informative and exist practically on every computer system, making them a valuable resource to track and investigate system status. However, since system logs are largely composed of diverse, freeform text, analytics is challenging.

Numerous log mining tools have been designed for different systems. Many use rule-based approaches [7, 15, 28, 29, 31, 32, 40, 41], which, though accurate, are limited to specific application scenarios and also require domain expertise. For example, Beehive [41]

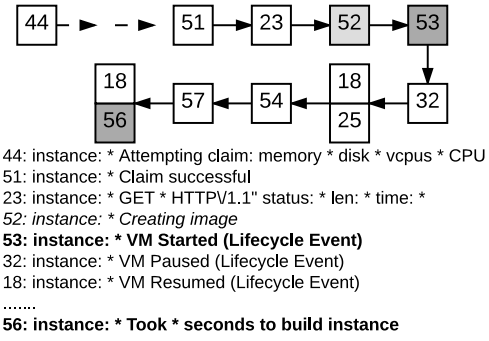


Figure 11: OpenStack VM Creation workflow.

identifies potential security threats from logs by unsupervised clustering of data-specific features, and then manually labeling outliers. Oprea [28] uses belief propagation to detect early-stage enterprise infection from DNS logs. PerfAugur [32] is designed specifically to find performance problems by mining service logs using specialized features such as predicate combinations. DeepLog is a general approach that does not rely on any domain-specific knowledge.

Other generic methods that use system logs for anomaly detection typically apply a two-step procedure. First, a log parser [9, 14, 16, 23, 36, 37] is used to parse log entries to structured forms, which typically only contain "log keys" (or "message types"). Parameter values and timestamps are discarded except for identifiers which are used to separate and group log entries. Then, anomaly detection is performed on log key sequences. A typical way is to generate a numeric vector for each session or time window, by counting unique log keys or using more sophisticated approaches like TF-IDF. The matrix comprising of these vectors is then amenable to matrix-based unsupervised anomaly detection methods such as Principal Component Analysis (PCA) [38, 39] and invariant mining (IM) [22]. Constructing such a matrix is often an offline process, and these methods are not able to provide log-entry level anomaly detection (rather, they can only operate at session level). We refer the reader to [17] for an overview and comparison on these methods.

Supervised methods [17, 44] use normal and abnormal vectors to train a binary classifier that detects future anomalies. A downside of such methods is that unknown anomalies not in training data may not be detected. Furthermore, anomalous data are hard to obtain for training. We have shown in our evaluation that using only a small portion of normal data to train, DeepLog can achieve online anomaly detection with better performance. Moreover, DeepLog also uses timestamps and parameter values for anomaly detection which are missing in previous work.

Workflow construction has been studied largely using log keys extracted from offline log files [4, 11, 21, 42] . It has been shown that workflow offers limited advantage for anomaly detection [11, 42]. Instead, a major utility of workflows is to aid system diagnosis [4, 21]. However, all past work assumes a log file to model only contains repeated executions of *one single task*. In this paper, we propose methods to automatically separate different tasks from log files in order to build workflow models for different tasks.

Besides workflows, other systems that perform anomaly diagnosis using system logs include DISTALYZER [26] that diagnoses system performance issues by comparing a problematic log against a normal one, LogCluster [19] which clusters and organizes historical logs to help future problem identification, and Stitch [45] that extracts different levels of identifiers from system logs and builds a web interface for users to visually monitor the progress of each session and locate performance problems. Note that they are for diagnosis purposes once an anomaly has been detected, and cannot be used for anomaly detection itself.

7 CONCLUSION

This paper presents DeepLog, a general-purpose framework for online log anomaly detection and diagnosis using a deep neural network based approach. DeepLog learns and encodes entire log message including timestamp, log key, and parameter values. It performs anomaly detection at per log entry level, rather than at per session level as many previous methods are limited to. DeepLog can separate out different tasks from a log file and construct a workflow model for each task using both deep learning (LSTM) and classic mining (density clustering) approaches. This enables effective anomaly diagnosis. By incorporating user feedback, DeepLog supports online update/training to its LSTM models, hence is able to incorporate and adapt to new execution patterns. Extensive evaluation on large system logs have clearly demonstrated the superior effectiveness of DeepLog compared with previous methods.

Future work include but are not limited to incorporating other types of RNNs (recurrent neural networks) into DeepLog to test their efficiency, and integrating log data from different applications and systems to perform more comprehensive system diagnosis (e.g., failure of a MySQL database may be caused by a disk failure as reflected in a separate system log).

8 ACKNOWLEDGMENT

The authors appreciate the valuable comments provided by the anonymous reviewers. Authors thank the support from NSF grants 1314945 and 1514520. Feifei Li is also supported in part by NSFC grant 61729202. We wish to thank all members of the TCloud project and the Flux group for helpful discussion and feedback, especially Cai (Richard) Li, for his valuable input on BROP attack.

REFERENCES

[1] VAST Challenge 2011. 2011. MC2 - Computer Networking Operations. (2011). http://hcil2.cs.umd.edu/newvarepository/VAST%20Challenge%202011/challenges/MC2%20-%20Computer%20Networking%20Operations/ [Online; accessed 08-May-2017].

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, and others. 2016. TensorFlow: A system for large-scale machine learning. In *Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 264–285.

[3] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. 2003. A neural probabilistic language model. *Journal of machine learning research* 3, Feb (2003), 1137–1155.

[4] Ivan Beschastnikh, Yuriy Brun, Michael D Ernst, and Arvind Krishnamurthy. 2014. Inferring models of concurrent systems from logs of their behavior with CSight. In *Proc. International Conference on Software Engineering (ICSE)*. 468–479.

[5] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. 2014. Hacking blind. In *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 227–242.

[6] Franois Chollet. 2015. keras. https://github.com/fchollet/keras. (2015). [Online; accessed 08-May-2017].

[7] Marcello Cinque, Domenico Cotroneo, and Antonio Pecchia. 2013. Event logs for the analysis of software failures: A rule-based approach. *IEEE Transactions on Software Engineering (TSE)* (2013), 806–821.

[8] Andrew M Dai and Quoc V Le. 2015. Semi-supervised sequence learning. In *Proc. Neural Information Processing Systems Conference (NIPS)*. 3079–3087.

[9] Min Du and Feifei Li. 2016. Spell: Streaming Parsing of System Event Logs. In *Proc. IEEE International Conference on Data Mining (ICDM)*. 859–864.

[10] Min Du and Feifei Li. 2017. ATOM: Efficient Tracking, Monitoring, and Orchestration of Cloud Resources. *IEEE Transactions on Parallel and Distributed Systems* (2017).

[11] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. 2009. Execution anomaly detection in distributed systems through unstructured log analysis. In *Proc. IEEE International Conference on Data Mining (ICDM)*. 149–158.

[12] Yoav Goldberg. 2016. A primer on neural network models for natural language processing. *Journal of Artificial Intelligence Research* 57 (2016), 345–420.

[13] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

[14] Hossein Hamooni, Biplob Debnath, Jianwu Xu, Hui Zhang, Guofei Jiang, and Abdullah Mueen. 2016. LogMine: Fast Pattern Recognition for Log Analytics. In *Proc. Conference on Information and Knowledge Management (CIKM)*. 1573–1582.

[15] Stephen E Hansen and E Todd Atkins. 1993. Automated System Monitoring and Notification with Swatch.. In *Proc. Large Installation System Administration Conference (LISA)*. 145–152.

[16] Pinjia He, Jieming Zhu, Shilin He, Jian Li, and Michael R Lyu. 2016. An evaluation study on log parsing and its use in log mining. In *Proc. International Conference on Dependable Systems and Networks (DSN)*. 654–661.

[17] Shilin He, Jieming Zhu, Pinjia He, and Michael R Lyu. 2016. Experience Report: System Log Analysis for Anomaly Detection. In *Proc. International Symposium on Software Reliability Engineering (ISSRE)*. 207–218.

[18] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* (1997), 1735–1780.

[19] Qingwei Lin, Hongyu Zhang, Jian-Guang Lou, Yu Zhang, and Xuewei Chen. 2016. Log clustering based problem identification for online service systems. In *Proc. International Conference on Software Engineering (ICSE)*. 102–111.

[20] Chaochun Liu, Huan Sun, Nan Du, Shulong Tan, Hongliang Fei, Wei Fan, Tao Yang, Hao Wu, Yaliang Li, and Chenwei Zhang. 2016. Augmented LSTM Framework to Construct Medical Self-diagnosis Android. In *Proc. IEEE International Conference on Data Mining (ICDM)*. 251–260.

[21] Jian-Guang Lou, Qiang Fu, Shengqi Yang, Jiang Li, and Bin Wu. 2010. Mining program workflow from interleaved traces. In *Proc. ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD)*.

[22] Jian-Guang Lou, Qiang Fu, Shengqi Yang, Ye Xu, and Jiang Li. 2010. Mining Invariants from Console Logs for System Problem Detection.. In *Proc. USENIX Annual Technical Conference (ATC)*. 231–244.

[23] Adetokunbo AO Makanju, A Nur Zincir-Heywood, and Evangelos E Milios. 2009. Clustering event logs using iterative partitioning. In *Proc. ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD)*. 1255–1264.

[24] Christopher D Manning and Hinrich Schütze. 1999. *Foundations of statistical natural language processing*. MIT Press.

[25] Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernockỳ, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model.. In *Interspeech*, Vol. 2. 3.

[26] Karthik Nagaraj, Charles Killian, and Jennifer Neville. 2012. Structured comparative analysis of systems logs to diagnose performance problems. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 26–26.

[27] Christopher Olah. 2015. Understanding LSTM Networks. (2015). http://colah.github.io/posts/2015-08-Understanding-LSTMs [Online; accessed 16-May-2017].

[28] Alina Oprea, Zhou Li, Ting-Fang Yen, Sang H Chin, and Sumayah Alrwais. 2015. Detection of early-stage enterprise infection by mining large-scale log data. In *Proc. International Conference on Dependable Systems and Networks (DSN)*. 45–56.

[29] James E Prewett. 2003. Analyzing cluster log files using Logsurfer. In *Proc. Annual Conference on Linux Clusters*.

[30] Robert Ricci, Eric Eide, and The CloudLab Team. 2014. Introducing CloudLab: Scientific Infrastructure for Advancing Cloud Architectures and Applications. *USENIX ;login:* 39, 6 (Dec. 2014). https://www.usenix.org/publications/login/dec14/ricci

[31] John P Rouillard. 2004. Real-time Log File Analysis Using the Simple Event Correlator (SEC).. In *Proc. Large Installation System Administration Conference (LISA)*. 133–150.

[32] Sudip Roy, Arnd Christian König, Igor Dvorkin, and Manish Kumar. 2015. Perfaugur: Robust diagnostics for performance anomalies in cloud services. In *Proc. IEEE International Conference on Data Engineering (ICDE)*. IEEE, 1167–1178.

[33] Elastic Stack. 2017. The Open Source Elastic Stack. (2017). https://www.elastic.co/products [Online; accessed 16-May-2017].

[34] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. 2012. LSTM Neural Networks for Language Modeling.. In *Interspeech*. 194–197.

[35] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. In *Proc. Neural Information Processing Systems Conference (NIPS)*. 3104–3112.

[36] Liang Tang and Tao Li. 2010. LogTree: A framework for generating system events from raw textual logs. In *Proc. IEEE International Conference on Data Mining (ICDM)*. 491–500.

[37] Liang Tang, Tao Li, and Chang-Shing Perng. 2011. LogSig: Generating system events from raw textual logs. In *Proc. Conference on Information and Knowledge Management (CIKM)*. 785–794.

[38] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael Jordan. 2009. Online system problem detection by mining patterns of console logs. In *Proc. IEEE International Conference on Data Mining (ICDM)*. 588–597.

[39] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I Jordan. 2009. Detecting large-scale system problems by mining console logs. In *Proc. ACM Symposium on Operating Systems Principles (SOSP)*. 117–132.

[40] Kenji Yamanishi and Yuko Maruyama. 2015. Dynamic syslog mining for network failure monitoring. In *Proc. ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD)*. 499–508.

[41] Ting-Fang Yen, Alina Oprea, Kaan Onarlioglu, Todd Leetham, William Robertson, Ari Juels, and Engin Kirda. 2013. Beehive: Large-scale log analysis for detecting suspicious activity in enterprise networks. In *Proc. International Conference on Dependable Systems and Networks (ACSAC)*. 199–208.

[42] Xiao Yu, Pallavi Joshi, Jianwu Xu, Guoliang Jin, Hui Zhang, and Guofei Jiang. 2016. CloudSeer: Workflow Monitoring of Cloud Infrastructures via Interleaved Logs. In *Proc. ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 489–502.

[43] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. 2010. SherLog: error diagnosis by connecting clues from run-time logs. In *ACM SIGARCH computer architecture news*. ACM, 143–154.

[44] Ke Zhang, Jianwu Xu, Martin Renqiang Min, Guofei Jiang, Konstantinos Pelechrinis, and Hui Zhang. 2016. Automated IT system failure prediction: A deep learning approach. In *Proc. IEEE International Conference on Big Data (IEEE BigData)*. 1291–1300.

[45] Xu Zhao, Kirk Rodrigues, Yu Luo, Ding Yuan, and Michael Stumm. 2016. Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle. In *Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 603–618.