Highlights

DistAD: Software Anomaly Detection Based on Execution Trace Distribution
Shiyi Kong, Jun Ai, Minyan Lu, Shuguang Wang, W. Eric Wong

- A measure of software runtime states based on software execution trace distribution is defined in this work to achieve the software runtime anomaly detection.

- A deep learning framework based on bi-direction long short term memory neural networks and an improved loss function based on KS-test are proposed to obtained a classified model under one class learning modes.

- Empirical study is conducted on a widely used database, named Cassandra. The results prove the feasibility of proposed method. The overheads are also evaluated in the empirical study.
DistAD: Software Anomaly Detection Based on Execution Trace Distribution

Shiyi Kong\(^a\), Jun Ai\(^b\), Minyan Lu\(^a\*,\) Shuguang Wang\(^a\) and W. Eric Wong\(^b\)

\(^a\)School of Reliability and Systems Engineering, Beihang University, No.37, Xueyuan Road, Haidian District, Beijing, 100191, China
\(^b\)Department of Computer Science, University of Texas at Dallas, 2601 N Floyd Road, Richardson, 75083, TX, USA

ARTICLE INFO

Keywords:
software anomaly detection
internal indicator
execution trace distribution
LSTM

ABSTRACT

Modern software systems have become increasingly complex, which makes them difficult to test and validate. Detecting software partial anomalies in complex systems at runtime can assist with handling unintended software behaviors, avoiding catastrophic software failures and improving software runtime availability. These detection techniques aim to identify the manifestation of faults (anomalies) before they ultimately lead to unavoidable failures, thus, supporting the following runtime fault-tolerant techniques. In this work, we propose a novel anomaly detection method named DistAD, which is based on the distribution of software runtime dynamic execution traces. Unlike other existing works using key performance indicators, the execution trace is collected during runtime via intrusive instrumentation. Instrumentation are controlled following a sampling mechanism to avoid excessive overheads. Bi-directional Long Short-Term Memory (Bi-LSTM), an architecture of Recurrent Neural Network (RNN) is used to achieve the anomaly detection. The whole framework is constructed under a One-Class Neural Network (OCNN) learning mode which can help eliminate the limits of lacking for enough labeled samples and the data imbalance issues. A series of controlled experiments are conducted on a widely used database system named Cassandra to prove the validity and feasibility of the proposed method. Overheads brought about by the intrusive probing are also evaluated. The results show that DistAD can achieve more than 70% accuracy and 90% recall (in normal states) with no more than 2 times overheads compared with unmonitored executions.

1. Introduction

With the development of computer science and software engineering, software has become increasingly complex. The increasing complexity of the systems makes them become much more difficult to test and validate. As emphasized in a technique report produced by the Software Engineering Institute of Carnegie Mellon University, “only 20% of the errors are discovered by the end of code development and unit testing, while 80% of the errors are discovered at or after integration testing”[1]. These residual defects are usually “complex faults” which are difficult to detect during software verification and validation (V&V). Their error triggers and propagation conditions are more complex than “simple defects”, which makes them difficult to reveal and more likely to survive from V&V phases[2]. Once triggered, these residual defects can lead to unexpected software failures and result in property losses.

Software anomaly detection and failure prediction methods are proposed to warn the anomalies brought by the residual defects during software executions, and the unexpected failures and their negative influence can be mitigated via further failure prevention techniques [3, 4]. Timely and accurately detecting the issues at software runtime periods when the software is abnormal (online anomaly detection), and predicting the possible failure through the internal or external manifestations of the software before failures occurring (online failure prediction)[5], are the key tasks of IT operations for the modern software systems and are known as algorithmic IT operations (AIOps)[6].

Software runtime anomaly detection methods (also called outlier detection) are used to monitor the software during their execution, collect the runtime information to extract the features as anomaly indicators, and then feed them into pre-trained machine learning models to detect anomalies[7]. The features, also called symptoms or indicators, are the characteristics of software states and behaviors that indicate anomalies or oncoming failures. The machine learning models are trained using historical data obtained in the previous execution process. There are two major concerns in the above processes:

- What runtime information should be collected as the features?
- What models should be selected to conduct the detection work?

At the same time, surrounding these two core problems, there are also many derivative issues that need to be solved in engineering practice. For example, some of these issues include mitigating the overheads brought by intrusive monitoring, improving the model’s performance, and dealing with the problems resulting from insufficient labeled samples.

We have reviewed the state-of-the-art of anomaly detection and failure prediction areas. After the literature review,
we found that there were four main types of indicators used for anomaly detection:

- Indicators from internal of the software objects, such as execution traces, function/method calls and duration, and interruptions[8]. These indicators should be collected using intrusive probes which may cause non-negligible overheads. Log-based anomaly detection accounts for a large proportion of this type of indicator, as in [9, 10, 11, 12]. The performance of these methods depends heavily on the quality of the logs produced by the software applications.

- Indicators collected from System Internal Environments (SIE)[2], including operating systems performance metrics such as CPU utilization, CPU performance counters, and memory usages[13, 14]. Distributed traces from online distribution service systems also belong to this type of indicator[15, 16]. These indicators can be collected outside of the software, and they cause less overhead than the first type.

- Indicators from the outside platforms. Application services deployed on the online service infrastructure, such as Ali-Cloud, Azure, or Amazon Web Services and so on should use these indicators to predict node anomalies to prevent unexpected failures caused by node failures[17].

- Indicators from the user sides. User experiences can help find anomalies that are difficult to detect on the server or developer and operation sides[18, 19, 20].

In addition to the indicators, the classification models used to detect abnormal states are also key parts of software anomaly detection. There are three types of models commonly used in existing research work:

- Statistical patterns obtained from historical data. The threshold is set according to previous experiences and statistical results. Time series forecasting methods are used to forecast whether the peak values of the indicators are over the threshold. Due to the simplest logic rules and the lowest inference time cost, threshold-based models are mostly used in the present AIOps framework, as illustrated in [21].

- Traditional machine learning models trained using historical data. Decision trees and support vector machines (SVMs) are typical examples of this type[22, 23]. In contrast to deep learning methods, which can conduct the representation learning in hidden layers, traditional machine learning methods have simpler architecture and can be more efficient to train and infer. However, they require features that have been finely designed manually.

- Unlike traditional machine learning algorithms depends heavily on the representation of the given data, deep learning methods can automatically learn the representation within several hidden layers[13, 14, 24]. However, deep learning methods cost much more time and computing resources than simple machine learning algorithms, and they also take more time to complete the inference phases after being deployed.

Approaches for anomaly detection can be divided into three groups depending on the use of datasets types. In the first kind, anomalies are detected with no prior knowledge of the data, as some of the pattern-based methods use thresholds from engineering practice [25]. The second group separates the datasets according to nominal data and abnormal data, like the supervised learning methods in ML or DL domains. This type accounts for the largest proportion of existing approaches. The third group takes advantage of the normal data only, and uses one class learning (abbreviated as OCL) methods like clustering methods to identify the normal data from all other data (anomalies)[26, 27]. One class learning method usually contain two parts, 1) The first part includes an algorithm to organize the existing nominal data and produce a learning model. 2) The second part involves a metric to identify whether novel data pieces belong to the normal domain.

The results of the literature review indicate that existing research prefers to use the indicators collected from logs, SIE, and other indicators outside of the software. These indicators are much easier to collect than internal indicators, like low end execution information without introducing too many overheads. However, Li [28] noted that the outside indicators are strongly related to the performance issues and cannot reveal functional issues that usually do not have performance manifestations, which represent large proportions of all failure types. Taking log information as an indicator is heavily limited by the quality of log produced by applications. Internal indicators extracted from low-level execution data should be more related to software runtime status and can reveal more anomaly types, as claimed by Katz in her PhD thesis[25].

The assumption used in this work is that long running software system behaviors follow stable distributions. The long running systems denote the software systems keep alive for a long duration and provide continuous service (i.e. online services systems, databases)[29]. Therefore, this work uses machine learning models to fit the distribution and generate samples following the distribution for a specific time interval. Once the generated predictions are considerably different from the real values, meaning that the current states or behaviors break the stable distribution of software systems, anomalies should occur.

The major contributions of this work are summarized as following:

- A measure of software runtime states based on software execution trace distribution is defined in this work to achieve the software runtime anomaly detection.

- A deep learning framework based on bi-direction long short term memory neural networks and an improved
loss function based on KS-test are proposed to obtained a classified model under one class learning modes[30].

- Empirical study is conducted on a widely used database, named Cassandra. The results prove the feasibility of proposed method. The overheads are also evaluated in the empirical study.

This work is organized as follows: Section 2 introduces the methodology of the entire work, which contains the assumptions, the detection models and the deep learning framework used in this work. Section 3 evaluates the proposed framework with respect to several aspects and demonstrates the feasibility of the methods. We address the threats to validity in Section 4. Section 5 summarizes the whole work and suggests research directions of future work.

2. Methodology

2.1. Problem Formulation

This work proposes the following assumption 1 as the basis of the proposed approach DistAD.

Assumption 1. The states set $S$ of the component set $U$ in long running software systems follows a distribution $P$. Taking a specific time window into consideration, if the states $S'$ of set $U$ have a distribution $P'$ in the time window that is considerably different from the long running distribution $P$, then there should be anomalies in this time window.

The execution state $S$ is denoted as the whole set of each unit state in the software component set $U = \{u_1, u_2, \ldots, u_n\}$. The states $S_{u_i}$ is denoted by the equation 1.

$$S_{u_i} = \{u_1^{(t_1)}, u_1^{(t_2)}, \ldots, u_1^{(t_i)} \ldots \}.$$  (1)

$u_i^{(t_w)}$ denotes the duration ratio of each component $u_i$ in time window $[t_w, t_w + \Delta t]$ and can be calculated as following equation 2.

$$u_i^{(t_w)} = \begin{cases} \frac{t_{end} - t_{start}}{\Delta t}, & t_{start} \leq t_{end} \leq t_{w} + \Delta t \\ \frac{t_w - t_{start}}{\Delta t}, & t_{start} \leq t_{w} \leq t_{end} + \Delta t \\ \frac{t_w + \Delta t - t_{start}}{\Delta t}, & t_{w} \leq t_{start} \leq t_{w} + \Delta t \leq t_{end} \end{cases}.$$  (2)

The equation is then as follows:

$$S = \bigcup_{i=1}^{n} S_{u_i}.$$  (3)

All states of each component make up the states $S$ in equation 3. Considering the assumption 1, there is

$$S \sim P.$$  (4)

Using $f$ as the execution distribution of component set $U$ in any time window $[t_w, t_w + \Delta t]$, then there have

$$f(t_w) = \{u_1^{(t_w)}, u_2^{(t_w)}, \ldots, u_n^{(t_w)}\}. $$  (5)

There also have

$$\sum_{i=1}^{n} u_i = 1.$$  (6)

and

$$S = \bigcup_{i=1}^{+\infty} f(t_i).$$  (7)

The principle of the proposed method is shown in Figure 1. According to the assumption 1, $S$ follows $P$, the latter is a stable hidden distribution that is implicit. For component states $S'$ in an any time interval, there is $S' \subseteq S$. The distribution $P'$ is fitted by the deep learning model, and there should be $S \sim P'$. $S''$ is generated by the trained model according to the time position provided by $S'$, if there is $S'' \sim S'$, then there should be anomalies in software systems.

2.2. Model Framework

This work proposes a novel runtime anomaly detection method based on the above assumption and the software internal low end execution trace distribution is taken as the indicator.

As shown in Figure 2, we sample runtime data of software systems according to the specific sampling rules with two parameters: $\Delta t$ denotes the length of the time windows and $\delta t$ denotes the time interval between each two adjacent time windows. We calculate the trace distributions of all components in time window $[t_i, t_i + \Delta t]$, fed them into the pre-trained deep learning models, get the predicted distribution results of the next time window (depending on the sampling rules), and compare the prediction with the real values inspiring by the Kolmogorov-Smirnov (KS) test to determine whether there is a considerable difference between the two distributions. The above processes comprise our runtime anomaly detection approach.

Shiyi Kong, Jun Ai, Minyan Lu, Shuguang Wang, W. Eric Wong: Preprint submitted to Elsevier
DistAD: Software Anomaly Detection Based on Execution Trace Distribution

Software Execution Process - Time Aspect

Figure 2: Sketch map of the runtime anomaly detection method.

Execution trace distribution calculating & Time windows sampling

Distribution based DL models & runtime anomaly detection

The framework of the method is shown in Figure 3. There are four main parts in the framework.

First, the runtime execution trace information is collected according to the sampler with parameters $\Delta t$ and $\delta t$. Then the trace sequences are transformed into distribution datasets according to the component composition of the software systems.

Second, the datasets are fed into a deep learning model consisting of a bi-direction long short term memory recurrent neural network (Bi-LSTM) and a full connected network layers. The KS-Loss proposed in this work is used as the loss function to measure the errors in the training process.

Third, unlike the above two processes which function offline, we generate the runtime samples in this phase. The calculations are the same as the offline data acquisitions.

Finally, the runtime samples are taken as the input of the pre-trained DL models. The outputs of the models are the prediction distribution values and they are fed into a decision-making model to determine whether there are any anomalies as shown in Figure 2. Warnings are generated if anomalies are detected.

2.2.1. The datasets for deep learning model training

The first layer of deep-learning algorithms is Bi-LSTM. The input of Bi-LSTM is a three-dimensional matrix, and the three dimensions of the matrix are batch size, input size and time steps, as Figure 4 shows.

Each point in Figure 4 denotes a duration time ratio of a component in a time window, marked as $u^{i}_{t_{w}}$. The batch size is the number of samples, which refers to sequences consisting of distribution data of continuous time windows.
Samples \( l_i \) are the subsets of \( L \). \( L \) is segmented into several parts according to the time interval boundaries, which can be seen as follows:

\[
L = \bigcup_{i=1}^{m} l_i. \tag{8}
\]

The input size denotes the number of features in the datasets, which equals to the number of components.

It should be noted that the lengths of segments (samples) \( l \) are different from each other. When feeding them into DL algorithms via batch processing, we should pad the sequences to make them be of the same length in the same batch.

### 2.2.2. Deep Learning Framework

The deep learning framework used in this work is shown in Figure 5. There are two main parts in the framework, Bi-LSTM and fully connected networks.

Binary-direction long short term memory (Bi-LSTM) has been proven to be with great ability and qualities when dealing with time series data. There are four states in LSTM models, known as follows:

\[
\begin{align*}
z &= \tanh(W \times \text{cat}(x^t, h^{t-1}T)), \tag{9} \\
z^f &= \sigma(W^f \times \text{cat}(x^t, h^{t-1}T)), \tag{10} \\
z^l &= \sigma(W^l \times \text{cat}(x^t, h^{t-1}T)), \tag{11} \\
z^o &= \sigma(W^o \times \text{cat}(x^t, h^{t-1}T)). \tag{12}
\end{align*}
\]

The cell state is calculated as Equation 13.

\[
e^t = z^fo + z^l ⊕ z. \tag{13}
\]

The final outputs are calculated as follows:

\[
\begin{align*}
h^t &= z^2\tanh(e^t), \tag{14} \\
y^t &= \tanh(W^t \times h^t + b^t). \tag{15}
\end{align*}
\]

Two LSTM layers with different directions are used in this framework to help the model learn the sequence relationships.

#### 2.2.3. KS loss function

The loss function used in the training process is inspired by the Kolmogorov-Smirnov test.

The Cumulative Distribution Function (CDF) of a given distribution \( X \) is defined as Equation 16.

\[
F(x) = Pr(X \leq x). \tag{16}
\]

The loss function \( \mathcal{L}_{ks} \) is defined as following Equation 17.

\[
\mathcal{L}_{ks} = \max(|F(x)_{\text{pred}} - G(x)_{\text{real}}|). \tag{17}
\]

\( F(x)_{\text{pred}} \) denotes the CDF of the prediction values from the trace distribution of a time window, and \( G(x)_{\text{real}} \) denotes the real values of the same time window. The CDF can be calculated as follows equation 18.

\[
F(x) = \sum_{i \leq x} u_{i}(tw), \quad i, x \in U. \tag{18}
\]

\( U \) denotes an ordered sequence of all system components. \( i, x \) denotes the index of components. If the index of component \( i \) is less than the index of component \( x \), there is \( i \leq x \).

### 3. Evaluations

In this work, we aim to address the following research questions:

**RQ1** Does the execution trace is predictable?

**RQ2** How does the proposed approach perform on different sampling parameters and compared with existing methods?

**RQ3** How do sampling parameters \( \Delta t \) or \( \delta t \) affect the monitoring overheads?

### 3.1. Experimental Setup

#### 3.1.1. Datasets

In this study, two common used open source software objects are used to conduct the evaluation. The first one is `grep` in Linux[31]. Another is Apache Cassandra\(^1\), which is an open-source NoSQL distribution database system from Apache Software Foundation.

For `grep`, we use a well designed test suite provided by [31] which contains 470 tasks. The test suite can cover almost all functions in `grep`.

`grep` is instrumented using fork functions provided by the GCC compiler [32]. Due to the limited software scales, we instrument all functions of the grep and record all traces of each task. All tasks can be processed in 5 seconds after instrumentation. The number of functions called during each task varies from 3000 times to 6000 times, which are equal to the length of the corresponding time series for the deep learning model.

\(^1\)Apache Cassandra, which is available at [https://cassandra.apache.org](https://cassandra.apache.org)
For Cassandra, we deploy a two-node database system in the experimental environment. Each computer is an Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz with 8GB memory and a 1TB hard driver. The operating system is Ubuntu 18.04 LTS. The Yahoo! Cloud Serving Benchmark (YCSB) is used to connected with Cassandra systems and performs different level workloads[33].

YCSB is provided with 5 benchmark core workloads from "Workload A" to "Workload E". The 5 core workloads contain different proportions of read and write operations which represent several common database usage scenarios.

Cassandra is implemented using the JAVA programming languages, and thus, we using a JAVA tracing tool named BTrace for instrumentation and collection of runtime trace information.

The database system runs for 10 days as a cycle. We set different sampling parameters of $\Delta t$ and $\delta t$ and run them for different cycles. Beside these sampling rules. The effects on overheads and prediction performance of different sampling rules are also studied in this paper.

The training processes are conducted on a workstation deployed with Ubuntu 20.04 LTS, which has 2 CPU (Intel(R) Xeon(R) Gold 5220R CPU @ 2.20GHz) nodes, 128GB memory and a NVIDIA Quadro GV100 Graphic Card.

3.1.2. Measurements
In this section, we introduce the metrics used to measure the performance of the anomaly detection method.

Firstly, several widely-used evaluation metrics are employed, namely accuracy, precision, recall, and F1-measure. Those measurements are derived from the confusion matrix as shown in Figure 6.

**Accuracy** measures the percentage of samples predicted correctly, as equation 19.

\[
\text{accuracy} = \frac{TP + TN}{TP + TN + FP + FN}.
\]

(19)

**Precision** measures the percentage of anomalies truly occurring among all predicted anomalies, as equation 20.

\[
\text{precision} = \frac{TP}{TP + FP}.
\]

(20)

**Recall** measures the percentage of anomalies that are correctly predicted among all anomalies truly occurring, and can be calculated using equation 21.

\[
\text{recall} = \frac{TP}{TP + FN}.
\]

(21)

**F1-measure** is the harmonic mean of precision and recall, as equation 22.

\[
F_1 - \text{measure} = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}.
\]

(22)

---

2 YCSB is available at https://ycsb.site
3 BTrace is available at https://github.com/btraceio/btrace
3.2. Evaluation of Trace Predictability (RQ1)

Many researchers study runtime anomaly detection based on external indicators, which take the majority of existing runtime anomaly detection methods. After the literature review, rare content failure has been studied. We agree with Katz’s opinions in [25]. She asserts that software content failures are more about software systems inheriting logical errors rather than resources-related errors so that they usually cannot be detected using external indicators that are most related to resource usage. Internal indicators that reveal software execution processes should be more related to software content anomalies.

In our previous work, we proved that low end execution indicators have strong relationships with software failures, such as times, duration[28] and runtime entropy[34]. In the offline evaluations, these models can accurately detect software anomalies.

In following parts of this section, we attempt to demonstrate that the execution traces of software applications are predictable (RQ 1). On the basis of the trace predictability, we construct the distribution-based anomaly detection methods in this work.

In this section, we take execution traces as time series and use LSTM models to fit their trends to prove the predictability of the execution traces (from the aspects of method calls and duration).

3.2.1. Predictability of Method Calls

Method/function invocation prediction refers to the problems that predict the next function/method in the execution trace.

Function/method names are extracted from the execution traces and constructed into time series obeying the orders of their appearance. To simplify the training process, we use a word embedding process. The dictionary is set by using all function names, and encoding the name in traces with one hot encoding. The one hot encoding process is shown in Figure 7.

![Figure 7: Illustrations of the one hot encoding process in this work.](image)

The training process is shown in Figure 8. We set different levels of targets to evaluate the predictability of the method/function invocations, as Figure 8 shows. The parameter named interval denoted as the interval between inputs and outputs.

![Figure 8: Method LSTM Network.](image)

Taking sets \( X = \{x_1, x_2, \ldots, x_n\} \) as inputs and \( Y = \{y_1, y_2, \ldots, y_n\} \) as outputs, if we set \( \hat{X} = \{x_2, x_3, \ldots, x_{n+1}\} \) as the target to fit a map \( f : X \rightarrow \hat{X} \), then the \( x_{n+1} \) could be used as the prediction of the next step of \( X \), and \( \hat{X} \) as the prior knowledge to make the prediction \( x_{n+1} \). The deviation between \( Y \) and \( \hat{X} \) denoted as \( \mathcal{L}(Y, \hat{X}) \) can be used to optimize the model parameters. Here \( \mathcal{L}(\cdot) \) represents the Cross Entropy Loss.

If we set the target as \( \hat{X} = \{x_6, x_7, \ldots, x_{n+5}\} \) with an interval = 5, the prediction could be \( \hat{X} \) with the prior knowledge \( X \). Then there is a period of lead time between the historical data and prediction results.

The dataset grep contains 470 time series, and we feed them into the LSTM model with the parameters listed in Table 1. We set interval to 1, 5, and 10 in this section.

A variable optimal learning rate is used to replace the constant learning rate to eliminate the oscillation of the loss values in order to make the loss converge rapidly. For every 80 epochs, the learning rate is multiplied by 0.95. Cross Entropy Loss (CE-loss) is used in this training process.

The results are shown in Figure 10. The loss value converges uniformly after the 150th epochs. The evaluation results are shown in Figure 11. More details of the evaluation results are shown in Table 2 and Figure 9. As the interval length increases, the accuracy decreases, but it remains acceptable.

The results in Table 2 show that the model can predict the next method/function name after current one accurately. When trying to predict next 5 or 10 function names, there is limited accuracy loss.

The above results have shown that the trace of software execution traces are predictable. In the following parts we...
prove that the duration of each method/function is also predictable.

### 3.2.2. Predictability of the Method Duration

A similar training framework is used to evaluate the predictability of method duration. The difference is that we use the method duration time as the target.

The inputs of the LSTM model are the same as those used in method name prediction. The outputs of the LSTM model become a sequence of duration times of functions that have been called in succession. The targets are set to a sequence of duration times that after several steps of the current function. In contrast to the interval values, the model owns different abilities to predict the next several method durations. Similar to the method name prediction phases, we set the value of interval to 1, 5 and 10. The duration dataset is collected using fork functions provided by the GCC compiler, and the time unit is set to nanoseconds (ns).

To measure the difference between outputs (prediction values) and the target (real values), we introduce Percentage Loss \( L_p \) and use it to optimize model parameters the Percentage Loss is shown in Equation 23.

\[
L_p = \frac{|\hat{Y}_{output} - Y_{target}|}{Y_{target}}. \tag{23}
\]

Where \( \hat{Y}_{output} \) denotes the outputs of the LSTM model and \( Y_{target} \) denotes the target that the model tries to follow.

The loss values in the training processes are shown in Figure 12 and Figure 13. The loss function converges uniformly to nearly 0.45. More details of the results are shown in Table 3.

After the loss converging uniformly, there are also near 45% errors between the prediction values and the real values. However, the method execution duration usually is measured by nanoseconds, a confidence interval \((0.55t, 1.45t)\) is an acceptable prediction result, where \( t \) denotes the execution time of the method/function which is to be predicted.

The evaluation results show that the execution trace of software functions are predictable both function orders and duration.

Function name prediction shows higher accuracy than duration prediction, since software structure is a more static property than duration, and the latter depends much more on runtime usages. The accuracy of prediction models is dropped with the increasing intervals.

Although we can find software runtime issues directly via software execution, extremely high overheads are required to record the whole execution trace, which heavily affects the system performance. On the basis of the predictability of the execution trace and the relationships between trace and software states, a framework of online anomaly detection is proposed which can eliminate the monitoring overheads and maintain the equal prediction accuracy. Sampling method is used to eliminate the intrusive monitoring overheads.

### 3.3. Evaluation on the Performance of the Proposed Framework under Different Sampling Rules (RQ2)

The proposed framework uses sampling to eliminate the overheads, however, sampling will lead to the loss of prediction accuracy. So we first build a benchmark without sampling, all methods are probed in this scenarios. We collect the traces, split them into different time windows with a time interval equals zero. Data distributions are calculated according to the length of time windows.

First, we take a replacement by using function/method execution counts rather than the physical time to simplify the data acquisition, splitting and calculation.
The replacement is conducted as shown in Figure 14. Counters are set at each entrance or exit of the functions. After every $d$ counters (equals $\Delta t$), the monitoring process would start, the monitoring process will last for $D$ counters (equals $\Delta t$). Then, the time window length should be $D$ and the interval should be $d$. Thus, the physical time of the length and interval of each time window can be different.

In this section, the influences of different sampling rules on performance are evaluated.

Figure 15 shows the loss curves of different sampling parameter pairs for both on the training sets and the testing sets.

During the training process, the parameters of the neural networks are optimized by minimizing the weighted summations of mean values $\mu$ and the variations $\sigma$ of $KS$-Loss (calculated using equation 17) among the whole batches, using Equation 24. Cause there is big numerical difference between $\mu$ and $\sigma$, we set a weighting factor $\beta$ to calculate the weighted summations, in the training process, we set $\beta = 50$.

$$
L = \mu_{\text{train}} + \beta \cdot \sigma_{\text{train}}.
$$

The variance analysis method is used to analyze the importance of $\Delta t$ and $\delta t$, and the results are shown in Table 4.

The variance analysis shows that $\Delta t$ has a more important influence on model training than $\delta t$. During the over experiments, $\Delta t = 1000$ outperforms than the other two conditions. $\delta t$ has rare impact on the training process, we have set $\delta t$ from 0 to $1 \times 10^5$ (number of function calls).

To analyze the influence on degree of model overfitting, we calculated the difference between train loss and test loss.
Table 4
Results of variance analysis on influence on loss.

|      | df  | sum_sq  | mean_sq  | F        | PR(>F)       |
|------|-----|---------|----------|----------|--------------|
| C(D) | 2.0 | 0.003645| 0.001823 | 10.495036| 0.002316     |
| C(d) | 6.0 | 0.001275| 0.000213 | 1.223827 | 0.359392     |
| Residual | 12.0 | 0.002084| 0.000174 | NaN      | NaN          |

Table 5
Results of the variance analysis on influence on overfitting.

|      | df  | sum_sq  | mean_sq  | F        | PR(>F)       |
|------|-----|---------|----------|----------|--------------|
| C(D) | 2.0 | 0.003057| 0.001528 | 35.950958| 0.000009     |
| C(d) | 6.0 | 0.000290| 0.000048 | 1.135515 | 0.399175     |
| Residual | 12.0 | 0.000510| 0.000043 | NaN      | NaN          |

Figure 13: Evaluation results on method duration prediction.

Figure 14: Using execution time counters and replacing the physical time.

loss, marked as $\mathcal{V}$ to measure the over-fitting degree, using equation 25.

$$
\mathcal{V} = |\mathcal{L}_{\text{train}} - \mathcal{L}_{\text{test}}|.
$$

(25)

Figure 17 shows the heat map of $\mathcal{V}$ and Table 5 shows the results of the variance analysis with respect to the influence of sampling parameters brought to $\mathcal{V}$.

Table 6
The statistical results of KS-Loss under $\Delta t = 1000$ and $\delta t = 100$.

| Average | median | Variance |
|---------|--------|----------|
| 0.2292  | 0.2233 | 0.0013   |

The results show that the sampling interval $\delta t$ has rare influence on model loss optimizing and over-fitting degree. As for sampling duration $\Delta t$, the over-fitting problem becomes serious as with the increase of $\Delta t$. We set the largest $\delta t = 1 \times 10^5$ in this study to make sure there are enough samples for deep learning training process.

Figure 18 shows the scatter map of the evaluation results (the values ks-loss). The datasets (with sampling parameters $\Delta t = 1000$ and $\delta t = 100$) used in figure 18 contains failures at the end. We can see an obvious change between $1.1 \times 10^5$ and $1.2 \times 10^5$. Figure 19 shows the local enlarged view of figure 18. Table 6 shows the statistical results when $\Delta t = 1000$ and $\delta t = 100$, the results under other parameters show similar trends. Figure 20 shows the distribution of the KS-Loss under normal working states.

In this situation ($\Delta t = 1000$ and $\delta t = 100$), In normal working states, the proportion of KS-Loss of time windows in the whole sets change with the values of $\beta$ in the threshold formula, as Figure 21 shows. When $\beta = 13$, more than 95% of the data points are under the threshold. For $\beta = 131$, there are more than 99% of the data points under the threshold.

$$
\mathcal{T} = \mu_{\mathcal{L}} + \beta \times \sigma_{\mathcal{L}}
$$

(26)

The decision making module in our runtime anomaly detection framework is constructed by checking if the number of the data points surpassing the threshold larger than 5%. The threshold is set as equation 26, where $\beta = 13$ as equation 21 shows. When the proportion of outliers larger than the significant level, then there should be failures in the applications.

The detection performance of the models obtained from the training processes are evaluated under different sampling
parameters. The validation datasets are collected from both the Normal Domain and the Failure Domain. The datasets contain 1230 normal data series (called Normal Samples) and 430 Abnormal data series (called Abnormal Samples). The failures in the Abnormal Samples include out of memory (OOM), running out of handlers, and bad infinity loop.

The evaluation results are shown in Table 7. The evaluation results using four measures are shown in Figure 22. In the model training phases, sampling interval values show limited influences on models’ convergence. However, the results in evaluation phases reveal that with the increase in the sampling interval, precision (similar to other three measures), declines rapidly.

A large sampling duration (such as 2000 counters) will produce a more stable results. A time window that is too short (short duration) like $\Delta t = 100$ may lack sufficient data to capture the distribution then lead to poorer detect ability. Combined with the training process, when $\Delta t = 1000$, the model can recall more normal samples, which is consistent with the result found in the training phases that $\Delta t = 1000$ leads to the minimum loss in all parameter pairs.

On the one hand, we apply a one class learning mode in our detection framework, the recall rates of abnormal samples are smaller than the recall of normal samples. On the other hand, as the figure 23 shows, we labeled all the data series collected from abnormal traces as abnormal samples, making them more prevalent than the real abnormal data series. Since Cassandra is a large scale database system, partial failures may not impact the normal execution of other parts, these parts of data pieces are also labeled as abnormal. Since the one class learning model can identify this data pieces as normal ones, which lead to a less recall rates of abnormal samples. Labeling the abnormal data samples precisely to obtain a more precise evaluations is still an open issue in this domain.

Shiyi Kong, Jun Ai, Minyan Lu, Shuguang Wang, W. Eric Wong: Preprint submitted to Elsevier
DistAD: Software Anomaly Detection Based on Execution Trace Distribution

Table 7
Evaluation results.

| Δt | measures |          |          |          |          |          |
|----|----------|----------|----------|----------|----------|----------|
|    |          |         | 0        | 100      | 500      | 1000     |
| 100| precision| 0.742489| 0.747826 | 0.768924 | 0.818356 | 0.745308 |
|    | recall   | 0.703252| 0.699187 | 0.784553 | 0.695935 | 0.452033 |
|    | accuracy | 0.599398| 0.60241 | 0.665663 | 0.660241 | 0.479518 |
|    | F1-measure| 0.722338| 0.722689 | 0.77666  | 0.752197 | 0.562753 |
| 1000| precision| 0.765943| 0.75945  | 0.764505 | 0.777691 | 0.695431 |
|    | recall   | 0.917886| 0.898374 | 0.910569 | 0.810569 | 0.556911 |
|    | accuracy | 0.731325| 0.713855 | 0.725904 | 0.687952 | 0.490964 |
|    | F1-measure| 0.731325| 0.713855 | 0.725904 | 0.687952 | 0.490964 |
| 2000| precision| 0.782051| 0.758491 | 0.747212 | 0.699187 | 0.490964 |
|    | recall   | 0.99187 | 0.817073 | 0.817073 | 0.747967 | 0.490964 |
|    | accuracy | 0.789157| 0.671687 | 0.659639 | 0.611446 | 0.490964 |
|    | F1-measure| 0.835059| 0.823091 | 0.831169 | 0.79379 | 0.455406 |

Table 8
Comparison with OmniAnomaly.

|          | precision | recall   | accuracy | F1-measure |
|----------|-----------|----------|----------|------------|
| DistAD #1| 0.765943  | 0.917886 | 0.731325 | 0.731325   |
| DistAD #2| 0.818356  | 0.695935 | 0.660241 | 0.660241   |
| OmniAnomaly | 0.696790 | 0.859999 | 0.627674 | 0.627684   |

Figure 16: Heat map of the loss values on testing datasets with different Δt and δt.

Since different indicators are used, it is difficult to compare our work with most existing work based on key performance indicators. To make an approximate comparison, we also collected the KPI information when we create the validation datasets simultaneously. We reproduce the model from [35] for our datasets, and the results are shown in Table 8. The results of DistAD #1 is when Δt = 1000 and δt = 0, and the results of DistAD #2 is when Δt = 100 and δt = 1000.

The model of OmniAnomaly is deployed without fine-tuning parameters for the datasets. Therefore, the results are slightly different from the original paper on other datasets (based on key performance indicators). The comparison shows that DistAD outperform than the OmniAnomaly under several sampling scenarios.

The results show that distribution of execution trace can be used to detect the anomalies at runtime with an acceptable accuracy.
Table 9
Overheads evaluation results.

| $\delta t$ (ms) | 100 | 1000 | 2000 |
|----------------|-----|------|------|
| runtime (ms)   | 4379.40 | 4404.13 | 4396 |
| throughput (ops/sec) | 231.05 | 236.08 | 236.78 |
| runtime (ms)   | 4327.72 | 4405.66 | 4365.45 |
| throughput (ops/sec) | 232.55 | 238.01 | 245.65 |
| runtime (ms)   | 4375.2 | 4382.1 | 4404.52 |
| throughput (ops/sec) | 231.17 | 235.89 | 236.12 |
| runtime (ms)   | 4355.15 | 4411.92 | 4343.78 |
| throughput (ops/sec) | 232.21 | 229.08 | 243.28 |
| runtime (ms)   | 3831.95 | 4360.89 | 4363.22 |
| throughput (ops/sec) | 263 | 231.32 | 234.65 |
| runtime (ms)   | 3740.66 | 4376.33 | 4357.08 |
| throughput (ops/sec) | 269.26 | 230.58 | 235.82 |
| runtime (ms)   | 2058.22 | 4320.26 | 4338.04 |
| throughput (ops/sec) | 357.05 | 268.8 | 250.58 |

Figure 18: The scatter view of the loss difference between prediction distribution values and real distribution.

### 3.4. Evaluation on monitoring overheads (RQ3)

As Katz addressed in her thesis [25], monitoring overheads under a limited level on those systems that involve a high level of human-machine interaction could be absorbed by the latency from networks, operations, and clients. We consider a database system, such as Cassandra, to belong to this kind, although it requires real-time feedback.

In the overheads evaluation parts, YCSB is used to generate the workload. The workload is executed 500 times on Cassandra, and the average runtime of each repeat workload and the throughput obtained from the client side (in this situation, YCSB) are shown in Table 9, Figure 24 and 25.

The average runtime without monitoring is 2100.98 ms and the throughput is 368.5 ops/sec (operations per seconds).

The results show that the overheads are more closely related with the ratio of $\Delta t$ and $\delta t$, as the figure 26 shows. As with the increase of the $\delta t/\Delta t$, runtime overheads will decrease so there is an increase in throughput and a decrease in average runtime of jobs. As claimed in [25], the overheads caused by intrusive instrument can be absorbed by the interactions between the clients and the server.

The increase in the sampling interval causes the loss of prediction accuracy, and there should be a trade-off when apply the method to real practice. $\delta t/\Delta t < 3$ is a proper interval for Cassandra in this work, which ensure more than 70% precision and the overheads are approximately 2 times. As a contrast, the overheads caused by an all method monitoring on grep could be 20 times. Since grep doesn’t need interactions in the testing environments and the monitoring overheads could not be absorbed by network communications or clients operations.

### 4. Threats to Validity

In this section, some potential threats to the validity of our research are presented.

#### 4.1. Internal Validity

Threats to the internal validity of this work related to the re-implementation of OmniAnomaly and the data pieces labeling methods. Since the author of OmniAnomaly made their code publicly available, we rebuilt the framework in our environment. Changes in the environment may cause different results. In the model training phases, their are many super-parameters need to be defined, and each super-parameter can cause different results. The abnormal data marking methods are also different. On the KPI datasets, we use the methods introduced in [35]. On the execution datasets, we marked the anomalies similar to the KPI data, once there are enough abnormal data points, all data pieces are marked as abnormal. However, unlike KPI data, the trace shows the complete execution states of the applications where only partial data points (or series) are abnormal, and the other data pieces should be marked as normal. Identifying abnormal data pieces rather than marking all data abnormal will be part of our future work.

#### 4.2. External Validity

External validity indicates the degree of generalizability of the results to other software systems. Empirical studies are conducted on grep and Cassandra. grep is too small to prove the detectability, so we use it when performing the motivation experiments, such as proof of the predictability of the execution trace. Cassandra has enough scales, thus,
we use it to perform the other evaluation works. However, there are so many other software types in the real world. More software types should be used to prove the feasibility of runtime anomaly detection using a low end execution distribution.

5. Conclusion

In contrast to performance based indicators, trace distribution based indicators in DistAD is a novel indicator from inheriting of software applications that can achieve the runtime anomaly detection.

In this work, the predictability of software runtime execution trace is proven first. A framework for distribution based anomaly detection is proposed and proven the feasibility of DistAD by conducting a series of empirical studies. Deep learning methods are used to learn the patterns of software runtime states in a one class learning fashion. KS-Loss, defined based on KS-test, is used in the deep learning phases to accelerate the training process. Unlike general CE loss functions or MSE loss, KS loss relax the constraints appropriately and can converge more quickly and uniformly.

Empirical study is conducted on a widely used database system name Cassandra. The performance of DistAD is evaluated in terms of for well-known measures, precision, recall, accuracy and F1-measure. We also evaluated the overheads brought by the instrumentation using client-side measures. The results show that DistAD can achieve a more than 70% precision and 90% recall (on recall of normal states) with no more than 2 times of overheads compared with un-probed applications. The tradeoff off between the overheads and performance should be carefully considered when applying the method to actual engineering practice.

In future work, we plan to improve the deep learning architecture to make the detection more accurate. Evaluation methods for the framework in the product environment should also be studied. The proposed framework can also be applied only on the critical parts of the whole application to improve the accuracy and further eliminate the overheads.

References

[1] P. H. Feiler, J. B. Goodenough, A. Gurfinkel, C. B. Weinstock, L. Wrage, Reliability validation and improvement framework, Technical Report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213, 2012.
[2] M. Grottke, K. S. Trivedi, Software faults, software aging and software rejuvenation (special survey: New development of software reliability engineering), The Journal of Reliability Engineering Association of Japan 27 (2005) 425–438.
[3] S. He, J. Zhu, P. He, M. R. Lyu, Experience report: System log analysis for anomaly detection, in: 2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE), 2016, pp. 207–218. doi:10.1109/ISSRE.2016.21.
[4] Y. Chen, M. Ying, D. Liu, A. Alim, F. Chen, M.-H. Chen, Effective online software anomaly detection, in: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2017, Association for Computing Machinery, New York, NY, USA, 2017, p. 136–146.
Figure 22: Evaluation results of proposed framework under different sampling rules.

Figure 23: The relationships among different parts of the datasets.

Figure 24: Runtime evaluation under different sampling parameters.

[5] F. Salfner, M. Lenk, M. Malek, A survey of online failure prediction methods, ACM Comput. Surv. 42 (2010).
[6] Y. Dang, Q. Lin, P. Huang, AIOps: Real-world challenges and research innovations, in: 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), 2019, pp. 4–5.
[7] V. Hodge, J. Austin, A survey of outlier detection methodologies, Artificial Intelligence Review 22 (2004) 85–126.
[8] L. Li, M. Lu, T. Gu, A systematic modeling approach for failure indicators of complex software-intensive systems, in: 2018 12th International Conference on Reliability, Maintainability, and Safety (ICRMS), IEEE, 2018, pp. 43–51.
[9] R. B. Yadav, P. S. Kumar, S. V. Dhavale, A survey on log anomaly detection using deep learning, in: 2020 8th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO), 2020, pp. 1215–1220. doi:10.1109/ICRITO48877.2020.9197818.
[10] L. Yang, J. Chen, Z. Wang, W. Wang, J. Dong, W. Zhang, Semi-supervised log-based anomaly detection via probabilistic label estimation, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), 2021, pp. 1448–1460. doi:10.1109/ICSE43902.2021.00130.
[11] A. Farzad, T. A. Gulliver, Unsupervised log message anomaly detection, ICT Express 6 (2020) 229–237.
**Figure 25:** Throughput evaluation under different sampling parameters.

**Figure 26:** Overheads changes with the $\delta t/\Delta t$.

---

12. S. Gholamian, P. A. S. Ward. On the naturalness and localness of software logs. in: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), 2021, pp. 155–166. doi:10.1109/MSR52588.2021.00028.

13. N. Zhao, H. Wang, Z. Li, X. Peng, G. Wang, Z. Pan, Y. Wu, Z. Feng, X. Wen, W. Zhang, K. Sui, D. Pei. An empirical investigation of practical log anomaly detection for online service systems, in: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2021, Association for Computing Machinery, New York, NY, USA, 2021, p. 1404–1415. doi:10.1145/3468264.3473933.

14. N. Zhao, J. Chen, Z. Yu, H. Wang, J. Li, B. Qiu, H. Xu, W. Zhang, K. Sui, D. Pei. Identifying bad software changes via multimodal anomaly detection for online service systems, in: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2021, Association for Computing Machinery, New York, NY, USA, 2021, p. 527–539. doi:10.1145/3468264.3468543.

15. A. Bento, J. Correia, R. Filipe, F. Araujo, J. Cardoso. Automated analysis of distributed failures: Challenges and research directions, Journal of Grid Computing 19 (2021) 1–15.

16. J. Bogatinovski, S. Nedelkoski, J. Cardoso, O. Kao. Self-supervised anomaly detection from distributed traces, in: 2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC), 2020, pp. 342–347. doi:10.1109/UCC45958.2020.300554.

17. Y. Li, Z. M. J. Jiang, H. Li, A. E. Hassan, C. He, R. Huang, Z. Zeng, M. Wang, P. Chen. Predicting node failures in an ultra-large-scale cloud computing platform: An aiops solution, ACM Trans. Softw. Eng. Methodol. 29 (2020).

18. W. Zheng, H. Lu, Y. Zhou, J. Liang, H. Zheng, Y. Deng. ifeedback: exploiting user feedback for real-time issue detection in large-scale online service systems, in: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2019, pp. 352–363.

19. L. Li, X. Zhang, X. Zhao, P. Zhao, B. Qiao, S. He, P. Lee, J. Sun, F. Gao, L. Yang, et al.. Fighting the fog of war: Automated incident detection for cloud systems, in: 2021 [USENIX] Annual Technical Conference ([USENIX]) (ATC) 21), 2021, pp. 131–146.

20. Z. Chen, Y. Kang, L. Li, X. Zhang, H. Zhang, H. Xu, Y. Zhou, L. Yang, J. Sun, Z. Xu, et al.. Towards intelligent incident management: why we need it and how we make it, in: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020, pp. 1487–1497.

21. X. Li, D. Nan, G. Huanbo, Z. Zhigang, J. Yong. Application of Intelligent Operation and Maintenance Practice, Publishing House of Electronics Industry, 2021.

22. G. A. Hoffmann, K. S. Trivedi, M. Malek. A best practice guide to resource forecasting for computing systems, IEEE Transactions on Reliability 56 (2007) 615–628.

23. C. Domeniconi, C.-s. Perng, R. Vilalta, S. Ma. A classification approach for prediction of target events in temporal sequences, in: European Conference on Principles of Data Mining and Knowledge Discovery, Springer, 2002, pp. 125–137.

24. I. Goodfellow, Y. Bengio, A. Courville. Deep learning, MIT press, 2016.

25. D. S. S. Katz, Identification of Software Failures in Complex Systems Using Low-Level Execution Data, Ph.D. thesis, Computer Science Department, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, 2020. URL: http://ra.adm.cs.cmu.edu/ anor/2020/CNU-CS-20-129.pdf.

26. R. Domingues, M. Filippon, P. Michiardi, J. Zouaoui. A comparative evaluation of outlier detection algorithms: Experiments and analyses, Pattern Recognition 74 (2018) 406–421.

27. M. A. Pimentel, D. A. Clifton, L. Clifton, L. Tarassenko. A review of novelty detection, Signal Processing 99 (2014) 215–249.

28. L. Li, M. Lu, T. Gu. Extracting interaction-related failure indicators for online detection and prediction of content failures, in: 2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), 2018, pp. 278–285.

29. P. Garafalakis, K. Karamanos, P. Pietzuch, A. Suresh, S. Rao, Medea: Scheduling of long running applications in shared production clusters, in: Proceedings of the Thirteenth EuroSys Conference, EuroSys ’18, Association for Computing Machinery, New York, NY, USA, 2018. doi:10.1145/3190588.3190549.

30. R. Chalapathy, A. K. Menon, S. Chawla. Anomaly detection using one-class neural networks, CoRR abs/1802.06360 (2018).

31. H. Do, S. G. Elbaum, G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact., Empirical Software Engineering: An International Journal 10 (2005) 405–435.

32. S. Kong, M. Lu, L. Li. Tracing error propagation in c/c++ applications, in: 2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C), 2018, pp. 308–315. doi:10.1109/QRS-C.2018.00062.

33. B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, R. Sears. Benchmarking cloud serving systems with ycsb, in: Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC ’10, Association for Computing Machinery, New York, NY, USA, 2010, p. 143–154. doi:10.1145/1807128.1807152.

34. S. Kong, M. Lu, J. Ai, S. Wang. Detection software content failures using dynamic execution information, 2021. arXiv:2110.06629.

35. Y. Su, Y. Zhao, C. Niu, R. Liu, W. Sun, D. Pei. Robust anomaly detection for multivariate time series through stochastic recurrent neural network., KDD ’19, Association for Computing Machinery, New York, NY, USA, 2019, p. 2828–2837. URL: https://doi.org/10.1145/3292500.3330672.