TraStrainer: Adaptive Sampling for Distributed Traces with System Runtime State

HAIYU HUANG, Sun Yat-sen University, China
XIAOYU ZHANG, Huawei, China
PENGFEI CHEN, ZILONG HE, ZHIMING CHEN, GUANGBA YU, and HONGYANG CHEN, Sun Yat-sen University, China
CHEN SUN, Huawei, China

Distributed tracing has been widely adopted in many microservice systems and plays an important role in monitoring and analyzing the system. However, trace data often come in large volumes, incurring substantial computational and storage costs. To reduce the quantity of traces, trace sampling has become a prominent topic of discussion, and several methods have been proposed in prior work. To attain higher-quality sampling outcomes, biased sampling has gained more attention compared to random sampling. Previous biased sampling methods primarily considered the importance of traces based on diversity, aiming to sample more edge-case traces and fewer common-case traces. However, we contend that relying solely on trace diversity for sampling is insufficient, system runtime state is another crucial factor that needs to be considered, especially in cases of system failures. In this study, we introduce TraStrainer, an online sampler that takes into account both system runtime state and trace diversity. TraStrainer employs an interpretable and automated encoding method to represent traces as vectors. Simultaneously, it adaptively determines sampling preferences by analyzing system runtime metrics. When sampling, it combines the results of system-bias and diversity-bias through a dynamic voting mechanism. Experimental results demonstrate that TraStrainer can achieve higher quality sampling results and significantly improve the performance of downstream root cause analysis (RCA) tasks. It has led to an average increase of 32.63% in Top-1 RCA accuracy compared to four baselines in two datasets.

CCS Concepts:
• Software and its engineering → Cloud computing; Software reliability; Software performance.

Additional Key Words and Phrases: distributed tracing, biased sampling, microservice

ACM Reference Format:

1 INTRODUCTION

Industrial microservice systems operate in a highly unpredictable and dynamic setting [46, 53]. To effectively monitor and manage these systems, distributed tracing [37] has been widely adopted. By implementing frameworks like OpenTelemetry [30] and SkyWalking [38], end-to-end paths...
of requests through service instances can be recorded as trace data. These distributed traces offer valuable insights for assessing risks, troubleshooting, and comprehending the system’s intricacies [7, 39]. Site Reliability Engineers (SREs) extensively leverage them to profile the environment [14], detect anomalies [25, 27, 51], and diagnose failures [10, 26, 35, 45].

Although distributed traces are helpful for analysis, their quantities and storage costs are significant [21]. Performing subsequent analysis on the complete set of traces is also computationally expensive [16]. To address this, trace sampling techniques [11, 16, 20, 21, 37, 52] have been introduced to selectively capture traces of interest.

A common sampling approach used by tracing systems like Jaeger [17] and Zipkin [2] is uniform random sampling [21]. This method determines whether a trace should be sampled at the beginning of the trace, thus is also known as head-based sampling. However, different traces have varying analytical value within the system, and head-based sampling cannot preserve more valuable traces [16]. To address this, tail-based sampling has been proposed and adopted in previous works [11, 16, 20, 21]. This method makes sampling decisions at the end of the request, allowing for the capture of complete trace information to determine if it is valuable. Thus it is also known as biased trace sampling.

However, previous biased sampling methods [11, 16, 20, 21, 52] have not fully considered sampling preferences, especially neglect the influence of system runtime state. These methods were only based on a core intuition: setting sampling preferences based on the specificity and diversity of traces, tending to keep more edge-case traces and fewer common-case traces. This viewpoint makes sense because edge-case traces are more interesting and informative [21]. However, common-case traces are also worth analyzing, as demonstrated in §2.2. Moreover, methods solely based on trace diversity exhibit the same sampling preference (i.e., favoring rare traces) at any given point in system operation, disregarding the crucial context of system runtime state. In fact, the characteristics of a “valuable trace” also change when the system state changes, as shown in Fig. 1.

**TraStrainer Approach.** To select more valuable traces while reducing storage and computational costs, we propose an online biased trace sampling approach called TraStrainer. The core idea behind TraStrainer is to take into account both system runtime state and trace diversity, adaptively providing a more comprehensive sampling preference. TraStrainer simultaneously takes system metrics and traces as inputs. Trace Encoder (§4.1) automatically encodes the coming trace into vector representations based on the system metrics. The generated vector representations contain both structural and state information of the trace, with each system metric corresponding to a specific dimension in the vector. System Bias Extractor (§4.2) determines the valuable dimensions based on the fluctuations of the system metrics and obtains a preference vector. System-Biased
TraStrainer: Adaptive Sampling for Distributed Traces with System Runtime State

2 BACKGROUND AND MOTIVATION

2.1 Background

Distributed traces. A trace corresponds to a series of actions triggered by a request within the system. It consists of a series of service operations (spans), and span events [32] have been incorporated in some tracing frameworks [30] to enhance the information associated with each span, as shown in Fig. 2(a). The structure of a trace reflects the order and hierarchy of the invocations among services. The status information such as invocation latency and event annotations on each span also provide crucial insights for subsequent analysis [15]. Despite the high analytical value of traces, the computational and storage overheads are also significant concerns [20].

Trace sampling. Large-scale industrial microservice systems generate millions to billions of traces daily [18, 37]. Analyzing and storing all traces incurs significant costs, making it impractical to meet low-latency requirements [3, 33, 37, 55]. Additionally, not all traces hold valuable insights for analysis [21]. Hence, sampling traces to filter out those with lower analytical value becomes a

(a) Distributed Trace Data

(b) System Metrics

Fig. 2. Trace data and system metrics.

Sampler (§ 4.3) and Diversity-Biased Sampler (§ 4.4) online calculate sampling probability from two different perspectives. Finally, Composite Sampler (§ 4.5) utilizes a dynamic voting mechanism to make the final sampling decision.

To evaluate the effectiveness and efficiency of TraStrainer, we constructed two datasets, one from 13 real-world production microservice systems and the other from two widely-used microservices benchmarks, OnlineBoutique [12] and TrainTicket [9]. We conducted a comprehensive evaluation of TraStrainer, including the quality of sampling results, performance in downstream tasks, and sampling efficiency. The experimental results demonstrated that compared to four baseline methods, TraStrainer was able to identify more valuable traces within the same budget and improve the performance of downstream analysis. Furthermore, TraStrainer exhibited the better efficiency than two online biased sampling methods.

Contributions. In summary, this study makes the following contributions.

• We present TraStrainer, an online biased trace sampler. By considering both system runtime state and trace diversity, TraStrainer offers a more comprehensive approach to adaptively determine sampling preferences.

• We propose an interpretable and automated method for trace representation. The generated trace vector includes both the structural and state information of the trace, with each system metric corresponding to a specific dimension in the vector.

• We combined different sampling methods with several classical trace-based analysis approaches to investigate the impact of different sampling methods on downstream root cause analysis tasks.

• We have implemented TraStrainer and constructed two datasets to validate the quality of its sampled data and its effectiveness in improving downstream analysis tasks. Moreover, the experimental results show that our method outperforms state-of-the-art sampling methods.
crucial aspect of operations in microservice systems. The current sampling methods for traces can be broadly categorized into two types: head-based sampling [2, 17, 37] and biased sampling [11, 16, 20, 21, 52]. As mentioned in § 1, head-based sampling often results in lower quality traces being retained. Therefore, our focus lies on biased sampling.

**System metrics.** The system metrics are a series of time-series data that reflect the runtime state of the system [19]. They typically consist of four components: timestamp, node or pod affiliation, metric type, and value, as shown in Fig. 2(b). During production, due to inevitable reliability and performance issues, the system state undergoes real-time changes [44], and these changes are often manifested by fluctuations in the system metrics [24]. For example, during an overload attack, Node A’s disk becomes paralyzed, which is manifested by a sharp increase in the DiskUsage metric on Node A. By analyzing the fluctuations of the metrics, we can infer the runtime state of the system.

### 2.2 Motivation

As mentioned in § 2.1, distributed traces are extensively helpful in profiling, diagnosing, and debugging [10, 26, 35, 45]. Therefore, the quality of the sampled traces largely determines the effectiveness of downstream analysis [47]. Prior to this, several biased samplers [11, 16, 20, 21, 52] were proposed to improve the quality of sampled traces. However, we found that the previous biased sampling approaches did not generate truly high-quality traces for downstream analysis, as demonstrated in our experimental findings (§ 5.5). After analyzing the experimental results, we concluded that this was primarily due to insufficient settings in the sampling preferences of the previous approaches. These approaches solely considered trace value based on diversity, favoring edge-case traces over common-cases.

**Solely favoring edge-case traces is not sufficient.** This is because: (1) Some common-case traces can be related to root causes. For instance, if there is an issue where a thread pool becomes exhausted, the cause may be excessive requests simultaneously issued to the thread pool. The traces associated to this issue may not exhibit any anomalies and are considered common-case traces. (2) Common-case traces are also helpful in downstream analysis methods. For example, trace-based root cause analysis algorithms [26, 45] typically need to collect a certain number of common-case traces to learn the normal invocation patterns and then identify and locate root causes based on this normal pattern. Some downstream approaches [23, 43] require common-case traces to conduct spectrum analysis [34]. (3) When encountering a large number of edge-case traces, it is necessary to determine which ones are more significant. In actual production scenarios, when system failures happen, there are often numerous abnormal traces [45]. If the anomaly rate exceeds the budget sampling rate, it becomes impossible to capture all the edge-case traces. In such scenarios, more specific rules are required to capture the more valuable portions of these traces.

**System runtime state plays an important role in determining valuable traces.** As a real-world issue case of Huawei is shown in Fig. 3: During the time window [a, b], there was an
increase in concurrent requests to the MySQL Server on Node A, including some full table queries, which led to an increase in CPU utilization on Node A. After time b, the CPU utilization reached full capacity, causing Node A to be unavailable. As a result, all requests passing through Node A exhibited abnormal behavior, generating a large number of abnormal traces.

The manual analysis process conducted by SREs for this failure is as follows: When receiving a system alert at time b, SREs examined the system state over a recent period and discovered the increased CPU utilization on Node A during the time window [a, b]. They further filtered and analyzed the traces passing through Node A, eventually identifying the root cause as an overload attack on the MySQL Server during the [a, b] time period.

For previous samplers that only considered trace diversity [16, 21], the structure and status information of the traces passing through Node A during the [a, b] time window actually appeared normal. Since these traces were considered common-cases, their sampling probability remained low during this time period, resulting in insufficient traces available for analysis and potentially missing the traces related to the root cause event that manual analysis can be aware of. This scenario shows major shortcomings of existing state-of-the-art samplers. It motivates us to consider the system state to dynamically increase the sampling probability of traces related to issues within the time window [a, b], based on the fluctuation of CPU utilization in Node A.

**Goal.** Given the discussion above, our goal is to implement biased trace sampling in a more comprehensive way, which considers not only the trace diversity, but also the system runtime state.

### 2.3 Problem Formulation

We define the problem of biased trace sampling with system runtime state as follows. Given the current system state metrics denoted as $\mathcal{M}$ and the traces collected over a period of time represented by $\mathcal{T}$, we aim to determine a sampling function $Sp(\beta, \mathcal{M}, \mathcal{T}, t)$ for a given budget sampling rate $\beta$. This function maintains an average rate of $\beta$ and calculates a biased sampling probability $\rho$ for each trace $t$,

$$Sp(\beta, \mathcal{M}, \mathcal{T}, t) \rightarrow \rho, \mathcal{T}'.$$  \hspace{1cm} (1)

Subsequently, the decision to retain trace $t$ is based on the sampling probability.

### 3 OVERVIEW

This section presents the design of TraStrainer, an adaptive online biased trace sampler. The objective of TraStrainer is to capture higher-quality traces based on system runtime state and trace diversity, and sample them with a higher probability. The overall architecture of TraStrainer is illustrated in Fig. 4. TraStrainer consists of two main phases: runtime data preprocessing and comprehensive sampling. In the runtime data preprocessing phase, for each coming trace, Trace Encoder (§ 4.1) automatically encodes it in an interpretable manner guided by metrics. The generated vector representation includes both structural and status information of the trace, with each system metric corresponding to a specific dimension in the vector. Simultaneously, System Bias Extractor...
Fig. 5. The process of encoding traces in two parts: status-related and structure-related.

(§ 4.2) dynamically calculates the anomaly degree of each system metric. This degree serves as the preference weight for the corresponding metric dimension. The larger the anomaly degree, the higher the preference weight. The preference weights for each dimension form a preference vector that determines which dimensions are more valuable.

During the online comprehensive sampling phase, TraStrainer takes into account the current coming trace vector and a look-back window of previous trace vectors. It makes a sampling decision that considers both the system bias and the diversity of the traces. On one hand, System-Biased Sampler (§ 4.3) calculates the sampling probability for the coming trace based on the preference vector obtained from System Bias Extractor, resulting in a system-biased sampling outcome. On the other hand, Diversity-Biased Sampler (§ 4.4) determines the diversity-biased sampling outcome by considering the structural and state differences between the coming trace and the previous traces. Finally, Composite Sampler (§ 4.5) uses a dynamic voting mechanism to make the final sampling decision based on the sampling budget and the current sampling frequency.

4 DETAILED DESIGN

4.1 Trace Encoder

Trace is not originally machine-readable and requires encoding into a machine-friendly format (e.g., vector) for next-step analysis (e.g., clustering, sampling) [16]. However, previous approaches suffered from insufficient manual intervention and a lack of interpretability. In order to address these issues, Trace Encoder proposes an automated trace encoding method guided by system metrics, which ensures that the dimensions of the resulting trace vector are interpretable.

As mentioned in § 2.1, a trace is a tree-like topology structure that contains valuable information in both its status and structure. Therefore, we can divide the encoding of a trace into two parts: status-related and structure-related, as shown in Fig. 5.

Encode the status-related part of the trace. When disregarding the structure of a trace, we can perceive it as a bag of spans, denoted as $S = \{s_1, ..., s_n\}$. Each span, denoted as $s_i$, represents a unit of work or operation within the trace. It includes essential information such as duration, status code, the associated node or pod, as well as additional details provided by event annotations. The information appended by event annotations can be complex and diverse. Previous approaches [16, 26] mostly relied on manual selection of relevant information based on human experience. However, this manual approach is costly and requires redefining rules when new fields are introduced. We leverage system metrics to automatically determine the features we need.

**Definition 1 (Related sub span bag $S_m$).** For a metric $m(m.nodename, m.typename)$ and a trace $t$, let $S = \{s_1, ..., s_n\}$ to be the span bag of $t$. We use $S_m = \{s_{m1}, ..., s_{mn}\}$ to denote the related sub span bag of $m$, where $s_{mi}$ must satisfy that $s_{mi}.node = m.nodename$ and $m.typename$ related to $s_{mi}.annotation$. 
**Definition 2 (Feature value $f_m$).** For the related sub span bag $S_m = \{s_{m1}, ..., s_{mn}\}$ of the metric $m$, we use $S_a = \{s_{a1}, ..., s_{an}\}$ to denote the abnormal span in $S_m$, where $s_{ai}$ must satisfy that $s_{ai}.\text{status}$ is abnormal. The feature value $f_m$ of $m$ is calculated as follows:

$$f_m = (|S_a| + 1) \ast \sum_{i=1}^{n} s_{mi}.\text{duration}. \quad (2)$$

As mentioned in § 2.1, each key of the system state metric $m$ includes the type of the metric and the node it belongs to (e.g., SQL connection time on node $C$), which can be represented by a tuple $(m.\text{node}, m.\text{type})$. For a trace $t$ and its corresponding span bag $S$, we select a subset of spans that are relevant to the metric $m$ as the related sub span bag $S_m$. $S_m$ represents the set of all behaviors in trace $t$ that are related to metric $m$ (e.g., span 3 and 4 are the related spans of $m_1$ in Fig. 5). Therefore, we calculate the statistical analysis of the spans in $S_m$ to obtain the feature value $f_m$ of $t$ in the corresponding dimension of $m$. We measure $f_m$ by calculating the total duration and the number of exceptions in the spans of $S_m$.

**Encode the structure-related part of the trace.** The trace structure shows the order and hierarchy of service operations. We call it an invocation tree. Because there are many asynchronous calls in the production environment, the order of spans at the same level on the trace can change [15]. Therefore, we focus on the invocation hierarchy of spans on the trace, rather than the strict order of spans at the same level. For a trace $t$, each layer of its invocation tree is encoded as a feature in the vector representation. Each span in a layer is represented by its parent span, method name, and argument (referred to as $pma$). If there is no span for a trace at a certain depth, the corresponding position is filled with null.

**Dimension scalability and reduction.** When new metrics are added or additional traces occur, Trace Encoder can automatically expand the vector dimensions without the need for additional manual intervention. For example, for newly introduced metrics, it only requires automatically identifying the related sub span bag as mentioned earlier and performing the necessary statistics. However, the vector expansion mechanism has a side effect of continuously increasing the dimension of the trace vector. To address this, we have adopted a dimension reduction strategy. For the status-related part, if it is observed that two metrics consistently have the same values for their dimensions over a period of time, those dimensions are merged. For the structure-related part, the reason for dimension redundancy is that a certain trace has a very deep call hierarchy but no longer occurs in the future. To address this, we only focus on recent traces. If the dimensions with greater depth have cleared corresponding values across all recent traces, those dimensions are removed.

### 4.2 System Bias Extractor

As mentioned in § 2.2, the runtime state of the system has a significant impact on sampling preferences. System Bias Extractor adaptively determines which metrics are more worthy of attention. It assesses the anomaly degree of various metrics by comparing the current system metric values with the expected values. It then generates a dimension preference vector that represents the preference score of each metric at the present moment.

**Assess the anomaly degree of metrics.** As described in § 2.1, each metric $m$ consists of a series of time-series data, represented as $m = \{(t_1, v_1), ..., (t_n, v_n)\}$. Our goal is to calculate online the anomaly degree $\alpha$ of each metric at the current time $t_k$, based on the historical time-series data within a look-back window $[t_1, ..., t_{k-1}]$. We consider two alternative approaches: (i) statistical methods and (ii) neural network models. Given that business operations in real production environments often exhibit periodic patterns, it is crucial to learn the historical variations of the metrics in order to better assess the current degree of fluctuations. Statistical methods such as boxplot [8] and
3-sigma detection [57] can only provide certain statistical measures based on historical data, which fail to learn the historical patterns. Therefore, we opt for the neural network model approach, which can better learn the waveform patterns in historical data.

Due to the online nature of the System Bias Extractor, complex models such as LSTM [36] and Transformer [41] variants are unable to meet the low latency requirements [11]. Therefore, we need a lighter model to fulfill this task. Taking inspiration from recent advancements in time series forecasting (TSF) [5], we have opted for a linear forecasting model. Zeng et al. [49] have demonstrated that linear models outperform Transformer-based methods in time series prediction tasks. We have adopted the DLinear algorithm [49], which combines a Decomposition scheme used in Autoformer [42] and FEDformer [54] with linear layers, and made adjustments for our specific task. This model takes a historical time series data window of a certain metric as input and outputs the expected value of the current time point, denoted as $v_k'$. The difference between the actual value $v_k$ and the expected value $v_k'$ is used to measure the degree of the anomaly, represented as $\alpha = \frac{|v_k' - v_k|}{\max(v_k', v_k)}$. An example of assessing the anomaly degree of metrics is shown in Fig. 6.

Form the preference vector. After obtaining the current anomaly degree $\alpha_i$ for each metric $m_i$, we consider $\alpha_i$ as the preference score $p_i$ for that metric. The preference scores of all metrics $M$ form the preference vector $P$ for the current moment, which is represented as $P = [p_1, ..., p_n]$.

### 4.3 System-Biased Sampler

The intuition behind System-Biased Sampler is that we prioritize traces that are more relevant to the fluctuations within the system. To achieve this, we maintain a look-back window consisting of recently collected traces. Based on the statistical measures of the traces within the look-back window on each dimension, we calculate an attention score vector for the coming trace. We then take the dot product of this attention score vector with the current system preference vector to determine the system-biased sampling probability.

Calculate the attention score vector. During the sampling process, we maintain a dynamic look-back window, denoted as $W = [t_1, ..., t_k]$, which consists of the most recent $k$ trace vectors. Here, we only consider the status-related part of each trace vector, denoted as $t_i = [f_{i1}, ..., f_{in}]$. For each dimension $i$, we calculate the mean $\mu_i$ and standard deviation $\sigma_i$ of the values taken by the previous $k$ trace vectors in that dimension. When considering the coming trace $t_{k+1}$, the attention score $a_i$ measures the increase in resource utilization in dimension $i$ compared to the historical traces, using Equation 4.3. The attention scores for all dimensions on the coming trace form the attention score vector, denoted as $A = [a_1, ..., a_n]$.

$$a_i = \frac{|f_{ik+1} - \mu_i|}{\sigma_i}.$$ (3)

---

Fig. 6. An example of assessing the anomaly degree of metrics.
Fig. 7. The process of the dynamic voting mechanism that takes the budget into account to combine the sampling results from the two previous samplers.

Calculate the system-biased sampling probability. When a coming trace obtains a higher attention score in dimensions with higher preference scores, it is more deserving of being kept. Once we obtain the attention score vector $\mathcal{A}(t_{k+1})$ for the coming trace $t_{k+1}$, we perform a dot product with the current preference vector $\mathbf{p}$, and then apply a $\tanh$ function to map the dot product result to the $[0, 1]$ range. This yields the system-biased sampling probability $p_s(t_{k+1})$.

$$p_s(t_{k+1}) = \frac{2}{1 + e^{-2 \mathbf{p} \cdot \mathcal{A}(t_{k+1})}} - 1. \quad (4)$$

4.4 Diversity-Biased Sampler

The goal of Diversity-Biased Sampler is to identify edge-case traces and assign them a higher sampling probability. To achieve this, we cluster the trace vectors within the look-back window and calculate the mass of each cluster. When a new trace arrives, we locate the closest cluster and calculate the diversity-biased sampling probability for the coming trace based on its similarity to the closest cluster and the mass of the cluster.

Cluster traces within the look-back window. To calculate the uncommon degree of the coming trace, we first establish patterns for previous traces. For a look-back window $W = \{t_1, ..., t_k\}$, we cluster the trace vectors within it. Assuming we obtain $n$ clusters $C = \{c_1, ..., c_n\}$ after clustering, we calculate the number of traces included in each cluster, which serves as the mass of each cluster, denoted as $Ma = \{ma_1, ..., ma_n\}$.

Calculate the diversity-biased sampling probability. For the coming trace $t_{k+1}$, we calculate its Jaccard similarity [29] with each cluster and consider the cluster with the highest similarity as the closest cluster, denoted as $c'_{k+1}$. The mass of $c'_{k+1}$, denoted as $ma'_{k+1}$, reflects the commonality of $c'_{k+1}$. The Jaccard similarity $si(t_{k+1})$ between $t_{k+1}$ and $c'_{k+1}$ reflects their matching degree. Therefore, the smaller the value of $ma'_{k+1}$ and $si(t_{k+1})$, the larger the uncommonness of the coming trace $t_{k+1}$. We use the following equation to calculate the diversity-biased sampling probability $p_d(t_{k+1})$.

$$p_d(t_{k+1}) = \frac{1}{\sum_{i=1}^{k+1} \frac{1}{ma_i \cdot si(t_i)}}. \quad (5)$$

4.5 Composite Sampler

Composite Sampler provides a more comprehensive measure of the importance of a trace by taking into account both system state and trace diversity. It also considers the impact of the budget on the sampling strategy and utilizes a dynamic voting mechanism to make the final sampling decision.
After **System-Biased Sampler** and **Diversity-Biased Sampler** provide their respective sampling probabilities $p_s(t)$ and $p_d(t)$ for the coming trace $t$, *TraStrainer* generates a random number between $[0, 1]$. It then compares this random number to $p_s(t)$ and $p_d(t)$ separately. If the sampling probability $p(t)$ for a certain aspect is greater than the random number, the sample result for that aspect is considered "True", otherwise it is considered "False".

**Dynamic Voting Mechanism.** In order to align the overall sampling rate with the expected budget, we have implemented a dynamic voting mechanism to combine the sampling results from the two previous samplers. Using a look-back window, we dynamically track the recent sampling frequency, denoted as $\theta$, and compare it with the budget sampling rate, denoted as $\beta$. If $\theta$ is greater than $\beta$, we need to enforce stricter sampling rules. In this case, we utilize an AND gate as the voting mechanism, meaning that *TraStrainer* will only sample the trace when both samplers yield a "True" result. On the other hand, if $\theta$ is smaller than $\beta$, we can relax the sampling rules. In this scenario, we employ an OR gate as the voting mechanism, indicating that *TraStrainer* will sample the trace as long as at least one sampler produces a "True" result, as shown in Fig. 7. The sampling decision ultimately determines whether the coming trace $t$ will be stored or discarded.

## 5 EXPERIMENT EVALUATION

We conducted experiments to evaluate how well *TraStrainer* performs in biasing sampling towards problem-related and edge-case traces. We also assessed its effectiveness for downstream root cause analysis tasks and analyzed the contribution of considering both sampling factors simultaneously. Through these studies, we aim to answer the following research questions (RQs).

- **RQ1**: How does the quality of the traces sampled by *TraStrainer* compare to the baseline approaches?
- **RQ2**: How effective is *TraStrainer* in downstream trace-based root cause analysis compared with baseline approaches?
- **RQ3**: How much does considering both system runtime state and trace diversity contribute to the effectiveness of *TraStrainer*?
- **RQ4**: How efficient is the sampling of *TraStrainer*?

### 5.1 Datasets

We evaluated *TraStrainer* using two datasets, namely $\mathcal{A}$ and $\mathcal{B}$. Dataset $\mathcal{A}$ consists of real-world data generated from 13 production microservice systems of Huawei. Dataset $\mathcal{B}$ is derived from two widely-used microservices benchmarks, OnlineBoutique [12] and TrainTicket [9]. Table 1 shows the detailed information of the two datasets.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Microservice Benchmark</th>
<th>Trace Number</th>
<th>Metric Number</th>
<th>Batch Number</th>
<th>Uncommon Label Rate</th>
<th>Problem-related Label Rate</th>
<th>Fault Types Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{A}$</td>
<td>13 Production Microservice systems</td>
<td>909,797</td>
<td>121</td>
<td>62</td>
<td>2.5%</td>
<td>2.5%</td>
<td>6</td>
</tr>
<tr>
<td>$\mathcal{B}$</td>
<td>OnlineBoutique and TrainTicket</td>
<td>112,000</td>
<td>32</td>
<td>56</td>
<td>5.0%</td>
<td>5.0%</td>
<td>5</td>
</tr>
</tbody>
</table>
and problem-related traces, with an average label rate of 2.5%. They also annotated the actual root causes for each batch to evaluate the accuracy of downstream root cause analysis.

**Dataset Setup.** The dataset is generated from two open-source microservice benchmarks: OnlineBoutique [12] and TrainTicket [9], which have been widely used in previous studies [6, 22, 45, 48, 50, 56]. We deployed them on a Kubernetes [1] platform consisting of 12 virtual machines. Each virtual machine is equipped with an 8-core 2.10 GHz CPU, 16GB memory, and runs on Ubuntu 18.04 OS. To collect traces, we utilized Opentelemetry Collector [31] and stored them in Grafana Tempo [13]. In order to simulate latency or reliability issues in the microservice system, we injected a total of 56 faults using Chaosblade [4] into these two microservice benchmarks. We collected corresponding trace data and metric data, resulting in 56 batches. The fault types included CPU contention, CPU consumed, network delay, code exception, and error return. The dataset comprises a total of 112,000 traces and 32 metrics. We annotated the uncommon traces and problem-related traces based on the injected fault positions and types, with an average label rate of 5.0%.

### 5.2 Baselines

**Baseline Sampling Approaches.** To evaluate the effectiveness and efficiency of **TraStrainer**, we compare it with four baseline sampling methods as follows.

- **Random** is the head-based sampling approach that decides whether to capture each trace with equal probability.
- **HC** [20] is an offline tail-based sampling approach using hierarchical clustering, which groups traces by label counting to conduct sampling.
- **Sifter** [21] is an online tail-based sampling approach, which approximates the distributed system’s common-case behavior and samples new traces based on how well represented they are.
- **Sieve** [16] is an online tail-based sampling approach, which uses robust random cut forest (RRCF) to detect uncommon traces and sample them with a high probability.

**Baseline Downstream Analysis Approaches.** Root cause analysis (RCA) is a common task performed in downstream analysis. To evaluate the impact of sampling methods on the analysis results, we combined the following three state-of-the-art trace-based RCA methods with different sampling approaches.

- **TraceAnomaly** [26] leverages deep learning to learn normal trace patterns offline, and then detects abnormal traces and identifies root causes online.
- **TraceRCA** [23] utilizes spectrum analysis to identify root cause services by analyzing the proportion of normal and abnormal invocations.
- **MicroRank** [45] identifies and ranks root causes by combining personalized PageRank method and spectrum analysis.

**Variants of **TraStrainer**.** To evaluate the impact of combining system state and trace diversity, we create the following two variants and conduct ablation experiments.

- **TraStrainer w/o M** is a variant that only considers trace diversity to set sampling preferences. We achieve this by using the sampling results from the Diversity-Biased Sampler as the final sampling outcome.
- **TraStrainer w/o D** is a variant that considers only the system runtime state to set sampling preferences. We achieve this by using the sampling results from the System-Biased Sampler as the final sampling outcome.

### 5.3 Evaluation Metrics

To evaluate the quality of the sampling results, we rely on two metrics: proportion and diversity. Both of them have been widely utilized in previous studies [11, 16, 51] and are defined as follows.
- **Proportion** (PRO) reflects the ability of an approach to bias towards valuable traces. Let $T$ be the total number of labeled traces, and $t$ denote the number of labeled traces that are sampled. The proportion of labeled traces can be calculated as: $\text{PRO} = \frac{t}{T}$. In our experiments, we had three variants based on different labels: the proportion of uncommon traces, the proportion of problem-related traces, and the proportion of traces that are both uncommon and problem-related.

- **Diversity** (DIV) reflects the ability of an approach to provide a representative sample. We denote the sampled traces obtained after sampling as $ST = \{s_{t1}, ..., s_{tn}\}$. Clustering them based on execution path yields trace patterns $TP = \{tp_{t1}, ..., tp_{tm}\}$. The diversity of the sampling results is simply the number of trace patterns obtained, which is denoted as $\text{DIV} = m$.

To assess the effectiveness of downstream root cause analysis, we employ three commonly used evaluation metrics: $A@1$, $A@3$, and $MRR$. These metrics serve as standard measures for evaluating the performance of RCA methods [25, 45] and are defined as follows.

- **Top-k accuracy** ($A@k$) represents the probability that the true root cause is included in the top-k positions of the results. Let $rc_i$ be the root cause of the $i$-th issue, $\text{Rank}^k_i$ be the top-k result list for the $i$-th issue. $A@k$ can be calculated as: $A@k = \frac{1}{|I|} \sum_{i=1}^{|I|} \mathbb{1}_{rc_i \in \text{Rank}^k_i}$. Higher values of $A@k$ indicate better accuracy.

- **Mean reciprocal rank** ($MRR$) represents the inverse of the rank of the first identified root cause. If the actual root cause is not present in the result list, the rank is considered to be infinity and its reciprocal is zero. Let $r_i$ be the rank of the root cause in the returned list for the $i$-th issue. The calculation for $MRR$ is: $MRR = \frac{1}{|I|} \sum_{i=1}^{|I|} \frac{1}{r_i}$. Higher values of $MRR$ indicate better accuracy.

### 5.4 RQ1: Sampling Quality of TraStrainer

To evaluate the sampling quality of TraStrainer, we compare its sampling results with four baseline methods. Our evaluation of sampling quality is divided into two aspects: bias sampling and representative sampling.

#### 5.4.1 Bias Sampling

Our first set of experiments evaluates TraStrainer’s ability to bias towards uncommon traces and problem-related traces. We conducted experiments on both dataset $A$ and $B$. To investigate the impact of different budgets, we set five gradient budget sampling rates: 0.1%, 1%, 2.5%, 5%, and 10%. These rates encompass budgets that are both higher and lower than the label rate. As the online sampling approach does not allow for precise fixed sampling rates, we followed the approach used in previous study [16]. If the final sampling quantity exceeded the budget, we removed a portion of the sampled traces in reverse order until the sampling result met the budget. Conversely, if the sampling quantity fell short of the budget, we randomly selected additional traces from the unsampled ones to match the budget. We conducted repeated experiments with the same settings for each batch and recorded the average proportion of the results.

- **Bias towards uncommon traces.** Fig. 8 (a1) and (b1) show the proportions of uncommon traces sampled by different methods across various budget settings in two datasets. TraStrainer and Sieve exhibit similar performance, outperforming the other three approaches. Random sampling, with no specific preference, achieves proportions roughly equivalent to the budget sampling rate. HC and Sifter, which solely consider trace structure while disregarding time delays, lack the ability to identify uncommon traces that only differ in execution time. Therefore, their proportions on both datasets do not exceed 0.7.

  We conducted a detailed comparison between TraStrainer and Sieve, both of which consider both time delays and structure during encoding, resulting in superior recognition of uncommon traces compared to other methods. When the budget is low, TraStrainer performs better, thanks to its metric-based encoding approach, which effectively distinguishes anomalies. However, when
Fig. 8. Proportions of uncommon, related, and uncommon-related traces sampled by different approaches across different budget settings in two datasets.

the budget is around the label rate (2.5% in dataset $A$ and 5% in $B$), TraStrainer’s proportion is lower than Sieve. This is because, in addition to bias towards uncommon traces, TraStrainer also exhibits a bias towards problem-related traces, which include some common traces. When the budget exceeds twice the label rate, both TraStrainer and Sieve achieve a proportion of 1, meaning all uncommon traces are captured.

**Bias towards problem-related traces.** Fig. 8 (a2) and (b2) show the proportions of problem-related traces. TraStrainer outperforms the four baselines in both datasets significantly. Sieve, Sifter, and HC exhibit similar proportion growth to random when the budget is high, with proportions below 0.5 in both datasets. TraStrainer takes system runtime state into account when setting sampling preferences. When the budget equals the label rate, the proportion for problem-related traces is around 0.6. When the budget is twice the label rate, the proportion approaches 1.0, meaning that all problem-related traces are captured.

**Bias towards uncommon problem-related traces.** Fig. 8 (a3) and (b3) show the proportions of traces that are both uncommon and problem-related. TraStrainer outperforms the baseline methods with better results even at lower budgets. When the budget rate equals the label rate (1% in dataset $A$ and 2.5% in $B$), TraStrainer achieves a proportion above 0.9 in both datasets, while the other baselines remain below 0.55. This indicates that TraStrainer not only prefers uncommon traces at lower budgets but also shows a stronger preference for problem-related traces within the uncommon traces.

5.4.2 **Representative Sampling.** Our next set of experiments assess the diversity of TraStrainer’s sampling results and its capability to prioritize underrepresented and abnormal request types. Similar to the previous experiments, we conducted comparisons with baselines on two datasets at various budget sampling rates. Additionally, we evaluated the sampling effectiveness of two variants of TraStrainer.
Table 2. Comparison of the result diversity of different sampling approaches across different budget settings.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Dataset A</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Dataset B</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.1%</td>
<td>1.0%</td>
<td>2.5%</td>
<td>5.0%</td>
<td>10.0%</td>
<td>0.1%</td>
<td>1.0%</td>
<td>2.5%</td>
<td>5.0%</td>
<td>10.0%</td>
<td>10.0%</td>
</tr>
<tr>
<td>Random</td>
<td>5.95</td>
<td>28.03</td>
<td>36.77</td>
<td>48.42</td>
<td>61.68</td>
<td>6.61</td>
<td>21.16</td>
<td>27.77</td>
<td>32.79</td>
<td>37.56</td>
<td></td>
</tr>
<tr>
<td>HC</td>
<td>7.27</td>
<td>28.11</td>
<td>40.68</td>
<td>52.77</td>
<td>67.60</td>
<td>9.04</td>
<td>27.56</td>
<td>33.86</td>
<td>39.02</td>
<td>41.84</td>
<td></td>
</tr>
<tr>
<td>Sifter</td>
<td>9.24</td>
<td>44.31</td>
<td>58.07</td>
<td>74.23</td>
<td>85.62</td>
<td>12.05</td>
<td>35.50</td>
<td>43.85</td>
<td>44.02</td>
<td>45.06</td>
<td></td>
</tr>
<tr>
<td>Sieve</td>
<td>13.70</td>
<td>98.24</td>
<td>151.27</td>
<td>155.10</td>
<td>161.24</td>
<td>13.70</td>
<td>51.27</td>
<td>54.44</td>
<td>54.44</td>
<td>54.44</td>
<td>54.44</td>
</tr>
<tr>
<td>TraStrainer w/o M</td>
<td>15.7</td>
<td>101.54</td>
<td>158.27</td>
<td>162.40</td>
<td>166.35</td>
<td>14.92</td>
<td>53.50</td>
<td>54.44</td>
<td>54.44</td>
<td>54.44</td>
<td>54.44</td>
</tr>
<tr>
<td>TraStrainer w/o D</td>
<td>8.06</td>
<td>28.53</td>
<td>42.66</td>
<td>55.26</td>
<td>73.23</td>
<td>6.05</td>
<td>17.73</td>
<td>23.03</td>
<td>27.34</td>
<td>32.45</td>
<td></td>
</tr>
<tr>
<td>TraStrainer</td>
<td>14.65</td>
<td>95.87</td>
<td>139.84</td>
<td>153.61</td>
<td>159.15</td>
<td>14.34</td>
<td>47.34</td>
<td>53.42</td>
<td>54.44</td>
<td>54.44</td>
<td>54.44</td>
</tr>
</tbody>
</table>

Fig. 9. Distribution of different APIs in the sampling results.

**Diversity of the sampling result.** Table 2 shows the diversity of sampling results. It can be observed that *TraStrainer* w/o M achieves the best performance in both datasets, demonstrating the effectiveness of *TraStrainer*’s metric-based trace encoding in distinguishing different trace patterns and ensuring fair sampling. Sieve follows closely behind as it considers both structure and latency during encoding. The diversity of *TraStrainer*’s sampling results is slightly lower than the previous two approaches because *TraStrainer* also takes preferences for related traces into account. On average, *TraStrainer*’s diversity is only slightly lower than *TraStrainer* w/o M by less than 5%. HC and Sifter, lacking consideration for latency, exhibit noticeably lower diversity in their sampling results compared to the previous three approaches. *TraStrainer* w/o D, which does not consider diversity preferences, shows no clear pattern in the diversity of its sampling results.

**Representative ability.** We selected a batch from dataset A and B, respectively. In dataset A, the traces consist of 8 different high-level API calls, with the following distribution ratios: 41%, 12%, 11%, 10%, 9%, 7%, 6%, and 4%. Among them, API-4 and API-8 are considered as exceptional interfaces. In dataset B, the traces consist of 5 different high-level API calls, with the following distribution ratios: 68%, 11%, 9%, 8%, and 4%. API-4 is considered as an exceptional interface in this case. We conducted 50 repeated experiments using the same setup and recorded the average distribution ratios of the sampling results for each approach.

Fig. 9 shows the distribution ratios of different APIs in the sampling results. It can be observed that *TraStrainer* enhances the representation of less frequently used APIs, resulting in a more balanced sampling. Additionally, considering the system’s state, *TraStrainer* also improves the
representation of APIs that are related to system issues and experience deterioration. In comparison, HC and Sifter only slightly increase the representation of low-frequency APIs, while Sieve achieves balanced sampling results but fails to enhance the representation of deteriorating APIs.

5.5 RQ2: Effectiveness of TraStrainer for Downstream Root Cause Analysis.

Root cause analysis (RCA) is a typical downstream analysis task. We evaluate the impact of sampling strategies on the effectiveness of RCA approaches. We use the sampled traces obtained through the sampling method as input for the RCA method to combine them, conducting experiments on two datasets. To explore the performance of sampling at different budgets, we conducted experiments at budget sampling rates of 0.1%, 1.0%, and 2.5%. The average results on the two datasets are shown in Table 3. It is evident that the variant employing TraStrainer as the sampling strategy achieves the best results for each RCA methods, in terms of $A@1$, $A@3$, and $MRR$.

In various variants of TraceAnomaly, TraStrainer-based approach significantly improved $A@1$, $A@3$, and $MRR$ by 33.78%, 26.88%, and 31.45%, respectively. TraceAnomaly employs a VAE-based model to detect anomalies and locate root causes, and the quality of the traces used to train this VAE-based model has a significant impact on root cause localization. We observed that when the budget is low, variants of the baselines HC, Sifter, and Sieve performed poorly, even slightly worse than the results based on random sampling. Further analysis of intermediate results revealed that in this phase, the baselines primarily obtained abnormal traces, making it difficult for the VAE-based model to learn normal trace patterns effectively, thus hindering accurate root cause localization. TraStrainer, on the other hand, focuses not only on abnormal traces but also on problem-related traces, enabling better results with lower sampling budgets. As the sampling budget increases, the performance of all variants improves. However, the baselines’ $A@1$ remains consistently below 30%, while TraStrainer achieves an $A@1$ of 54.84% and an $MRR$ of 0.7151.

Both TraceRCA and MicroRank employ spectrum analysis, which means that a microservice with more abnormal traces and fewer normal traces is more likely to be the root cause. In various

<table>
<thead>
<tr>
<th>RCA Approach</th>
<th>Sampling Approach</th>
<th>$A@1$</th>
<th>$A@3$</th>
<th>$MRR$</th>
</tr>
</thead>
<tbody>
<tr>
<td>TraceAnomaly</td>
<td>Random</td>
<td>10.71</td>
<td>9.26</td>
<td>16.67</td>
</tr>
<tr>
<td></td>
<td>HC</td>
<td>9.26</td>
<td>12.96</td>
<td>18.32</td>
</tr>
<tr>
<td></td>
<td>Sifter</td>
<td>11.67</td>
<td>24.07</td>
<td>16.67</td>
</tr>
<tr>
<td></td>
<td>Sieve</td>
<td>8.81</td>
<td>18.52</td>
<td>29.63</td>
</tr>
<tr>
<td>TraStrainer</td>
<td>w/o $M$</td>
<td>11.67</td>
<td>20.37</td>
<td>22.22</td>
</tr>
<tr>
<td>TraStrainer</td>
<td>w/o $D$</td>
<td>12.96</td>
<td>38.89</td>
<td>44.44</td>
</tr>
<tr>
<td>TraceRCA</td>
<td>Random</td>
<td>7.41</td>
<td>20.37</td>
<td>29.63</td>
</tr>
<tr>
<td></td>
<td>HC</td>
<td>9.26</td>
<td>24.07</td>
<td>24.07</td>
</tr>
<tr>
<td></td>
<td>Sifter</td>
<td>8.67</td>
<td>19.63</td>
<td>25.19</td>
</tr>
<tr>
<td></td>
<td>Sieve</td>
<td>18.52</td>
<td>31.48</td>
<td>38.89</td>
</tr>
<tr>
<td>TraStrainer</td>
<td>w/o $M$</td>
<td>18.52</td>
<td>33.33</td>
<td>35.19</td>
</tr>
<tr>
<td>TraStrainer</td>
<td>w/o $D$</td>
<td>24.07</td>
<td>55.56</td>
<td>55.56</td>
</tr>
<tr>
<td>MicroRank</td>
<td>Random</td>
<td>5.56</td>
<td>16.67</td>
<td>27.78</td>
</tr>
<tr>
<td></td>
<td>HC</td>
<td>7.41</td>
<td>18.52</td>
<td>22.22</td>
</tr>
<tr>
<td></td>
<td>Sifter</td>
<td>5.56</td>
<td>18.52</td>
<td>27.78</td>
</tr>
<tr>
<td></td>
<td>Sieve</td>
<td>9.26</td>
<td>25.83</td>
<td>35.19</td>
</tr>
<tr>
<td>TraStrainer</td>
<td>w/o $M$</td>
<td>12.96</td>
<td>16.67</td>
<td>24.07</td>
</tr>
<tr>
<td>TraStrainer</td>
<td>w/o $D$</td>
<td>29.63</td>
<td>42.59</td>
<td>46.30</td>
</tr>
<tr>
<td></td>
<td>TraStrainer</td>
<td>42.59</td>
<td>45.16</td>
<td>50.00</td>
</tr>
</tbody>
</table>

Table 3. Comparison of the effects of different sampling approaches in downstream root cause analysis.
variants of these two methods, TraStrainer-based approach improved A@1, A@3, and MRR by 31.48%, 38.34%, and 28.96%, respectively. We can observe that variants of HC and Sifter show analysis results similar to random sampling methods at various budgets. This is because TraceRCA and MicroRANK determine abnormal invocations based solely on latency, and these two sampling approaches consider only trace structure during encoding, thus providing no assistance to root cause analysis. As for the Sieve-based approach, it performs poorly at lower budgets because spectrum analysis weaves clues from both normal and anomalous traces for root cause localization, and the Sieve-based approach cannot provide a sufficient number of normal traces. TraStrainer consistently achieves good analysis results across different budgets, maintaining an A@3 above 70%. This is due to TraStrainer’s more comprehensive sampling preferences. Even with lower budgets, it discards traces unrelated to the problem rather than valuable traces for analysis, resulting in acceptable analysis performance.

5.6 RQ3: Contribution of the Combination of System Status and Trace Diversity.

We performed ablation experiments to examine the impact of combining system state and trace diversity in the sampling preference setting. We created two variants: TraStrainer w/o M, which solely focuses on bias sampling using trace diversity, and TraStrainer w/o D, which solely considers system state. We replicated partial experiments from RQ1 and RQ2 on these two variants and analyzed the experimental results.

Fig. 10 shows the distribution of different APIs in the sampling results sampled by various variants in dataset B. It is evident that the sampling results of TraStrainer w/o D exhibit a significant imbalance in the distribution ratios of different APIs. Specifically, the distribution ratio of deteriorating interface API-4 exceeds 70%, while the underrepresented API-5 is completely absent in the samples. Although TraStrainer w/o M produces relatively balanced sampling results on average, it fails to bias towards the more crucial deteriorating API-4. On the other hand, TraStrainer not only enhances the diversity of the sampling results but also exhibits a stronger inclination towards the problematic API.

Table 3 shows the effectiveness comparison of different variants of TraStrainer in downstream analysis. It can be seen that TraStrainer w/o D performs better overall than TraStrainer w/o M, suggesting that prioritizing problem-related traces yield better results for analysis tasks compared to solely focusing on uncommon traces. When analyzing the effects under different budgets, we notice that at a low budget (0.1%), TraStrainer w/o D has a significantly lower A@1 compared to TraStrainer. However, at a high budget (2.5%), TraStrainer w/o D approaches the performance of TraStrainer. This is because TraStrainer w/o D’s sampling preference is solely based on problem-relatedness without bias towards uncommon abnormal traces, potentially missing root cause traces at low budgets. As the budget increases, TraStrainer w/o D captures most of the problem-related traces including the culprits, resulting in a noticeable improvement in performance. On the other hand, TraStrainer, which considers both system state and trace diversity, achieves better analysis performance across all budgets. This finding further confirms the motivation, as described in § 2.2, of combining system state and trace diversity to enhance the effectiveness of trace sampling.

Fig. 10. Distribution of different APIs in the sampling results of different variants.
5.7 RQ4: Efficiency of TraStrainer.

We evaluated the efficiency of TraStrainer through a series of experiments. It is worth noting that in TraStrainer, the System-Bias Extractor and Trace Sampler run in parallel, and Trace Sampler only needs to read the current preference vector. During the efficiency validation, we deployed the sampler part of TraStrainer in a single-process form and conducted it on an Intel Xeon Gold 5318Y 2.10GHz CPU. We measured the average time it took for each trace to go through TraStrainer and make a sampling decision.

Fig. 11(a) shows the distribution of sampling latencies for traces from the two datasets. It is evident that TraStrainer generally exhibits smaller sampling latencies compared to the baselines, ranging from 0.28ms to 14.29ms. The maximum latency is 40.82% to 50.16% lower than the baselines. Two factors will affect the sampling latency: trace size and the number of metrics. Fig. 11(b) and (c) demonstrate the impact of these factors on TraStrainer’s sampling latency. Following the principle of controlling variables, we fixed the metric number at 75 when investigating the effect of trace size, and fixed the trace size at 50 when examining the impact of metric number. It can be observed that sampling latency exhibits a linear increase with trace size, but the maximum value remains below 15ms. When the metric number is low, the sampling latency also increases linearly. However, when the metric number exceeds 100, the growth rate of sampling latency becomes much slower. This is because TraStrainer employs dimension reduction strategies to prevent dimension explosion.

From the above analysis, it can be seen that TraStrainer is efficient enough for online sampling. Additionally, the dimension reduction strategy of TraStrainer can effectively improve the efficiency when there are a large number of metrics.

6 LIMITATIONS

In this work, we only implement biased sampling in a tail-based way. However, the most recent research [52] on trace sampling introduces the concept of retroactive sampling [52], which lazily retrieves trace data to perform biased sampling at earlier stages of the trace lifecycle, aiming to improve the efficiency of biased sampling. Nonetheless, current retroactive sampling approach also solely biases towards edge-cases. Our work proposes a more effective method for setting sampling preferences. We believe that this can be beneficial for various types of biased sampling, not only tail-based sampling but also retroactive sampling.

7 THREATS TO VALIDITY

The main threats to internal validity are associated with the implementation of baselines. For Sieve, we utilize its open-source implementation directly. However, since HC and Sifter lack open-source implementations, we carefully develop our own implementations and use exactly the same libraries based on relevant papers. Additionally, previous work has mentioned that online samplers make essentially random sampling decisions in the initial sampling phase due to the random initialization.
of the internal model [16, 21]. To facilitate result comparison, we allow each online sampler to undergo a warm-up period to ensure that the model is initialized before testing the sampling effectiveness. During the experiments, we use 10% of the traces to warm up the sampler.

The threats to external validity mainly lie in the representativeness of the datasets. Although we used data from 13 production microservice systems and two open-source microservice benchmarks for validation, the number of fault types we tested was still limited (7 in total). There could be more complex fault types in other microservice systems, so further testing is needed to assess the effectiveness of TraStrainer in more complex scenarios.

8 RELATED WORK

Distributed traces have become the primary source of observability and analysis for microservice systems [39], widely utilized for profiling, diagnosing, and debugging. Additionally, due to the large amount of trace data involved, sampling becomes essential in distributed tracing [16]. Below, we provide a brief summary of current methods for trace-based analysis and trace sampling techniques.

Trace-based analysis approaches. In microservice systems, trace analysis serves various purposes like anomaly detection (AD) and root cause analysis (RCA). Nedelkoski et al. [28] introduced a method for anomaly detection that employs a multimodal LSTM network, allowing the model to learn from trace data. TraceAnomaly [26] proposes an anomaly detection and root cause localization approach using Variational Autoencoders (VAE) with trace data. TraceRCA [23] and MicroRank [45] identifies root cause services by examining the ratio of normal to abnormal invocations and applying spectrum analysis. Sage [10] utilizes causal Bayesian networks and employs graphical variational autoencoders to pinpoint root cause microservices. FSF [35] utilizes knowledge of failure propagation and the client-server communication model to deduce root causes.

Trace sampling approaches. Uniform random sampling approach has been utilized by tracing systems like Dapper [37], Jaeger [17] and Zipkin [2]. However, this random sampling method fails to guarantee the quality of the sampled traces. Consequently, subsequent methods have introduced several biased sampling approaches. Las-Casas et al. [20] propose a hierarchical clustering method PERCH to bias the sampling to maximize the diversity of traces based on label counting. However, this approach was primarily designed for offline sampling. Sifter [21] performs an online sampling method, which approximates the common-case model of the traces and biases towards traces that are poorly represented by this model. Sieve [16] provides an online sampler that aims to bias sampling towards uncommon traces by using robust random cut forest. TraceCRL [51] utilizes contrastive learning and graph neural network techniques to encode trace data into vectors, enabling the sampling of traces based on the diversity of these vector representations. SampleHST [11] biases towards the edge-cases based on the distribution of mass scores obtained from a forest of Half Space Trees (HST). Hindsight [32] describes and implements retroactive sampling for capturing traces of symptomatic edge-cases. We can see that previous biased trace sampling approaches were merely different variations based on the fundamental principle of being biased towards edge-case traces, without introducing new perspectives for consideration.

Perspectives of TraStrainer. Compared to existing trace sampling methods, TraStrainer takes a more comprehensive approach to sampling preferences. It not only considers trace diversity but also takes into account the impact of system runtime state on sampling preferences. Previous trace sampling methods were limited to using trace information alone to set sampling preferences. In contrast, TraStrainer considers the information of both the traces and the system state metrics, utilizing multiple modalities of information for better sampling results. Additionally, TraStrainer is designed with the requirements of downstream trace-based analysis methods [10, 23, 26, 28, 45], providing higher-quality sampled data for these methods.


9 CONCLUSION

In this study, we introduce TraStrainer, an online biased sampler that takes into account sampling preferences more comprehensively. The core idea behind TraStrainer is to consider not only trace diversity but also system runtime state, allowing for dynamic adjustment of sampling preferences. TraStrainer employs a more interpretable and comprehensive encoding method for traces and utilizes a dynamic voting mechanism to generate integrated sampling decisions. We construct two datasets and conduct a comprehensive evaluation of TraStrainer, including the quality of sampling results, performance in downstream tasks, and sampling efficiency. The experimental results demonstrate that compared to four baseline methods, TraStrainer can identify more valuable traces within the same budget and improve the performance of downstream analysis methods. Furthermore, TraStrainer exhibits higher efficiency than two other online biased.

Artifact Availability. The implementation and data of TraStrainer are publicly available at [40].

ACKNOWLEDGMENTS

The research is supported by the Guangdong Basic and Applied Basic Research Foundation (No.2023B1515020054) and the National Natural Science Foundation of China (No.62272495). This research is also sponsored by Huawei. The corresponding author is Pengfei Chen.

REFERENCES


Received 2023-09-29; accepted 2024-01-23