Sieve: Attention-based Sampling of End-to-End Trace Data in Distributed Microservice Systems

Zicheng Huang, Pengfei Chen*, Guangba Yu, Hongyang Chen, Zibin Zheng
School of Data and Computer Science
Sun Yat-sen University
Guangzhou 510006, China
*Email: {huangzch8, yugb5, chenhy95}@mail2.sysu.edu.cn, {*chenpf7, zhzibin}@mail.sysu.edu.cn*

***Abstract*—End-to-end tracing plays an important role in understanding and monitoring distributed microservice systems. The trace data are valuable to help find out the anomalous or erroneous behavior of the system. However, the volume of trace data is huge leading to a heavy burden on analyzing and storing them. To reduce the volume of trace data, the sampling technique is widely adopted. However, existing uniform sampling approaches are unable to capture uncommon traces that are more interesting and informative. To tackle this problem, we design and implement *Sieve*, an online sampler that aims to bias sampling towards uncommon traces by taking advantage of the attention mechanism. The evaluation results on the trace datasets collected from real-world and experimental microservice systems show that *Sieve* is effective to increase sampling probabilities of the structurally and temporally uncommon traces and reduce the storage space to a large extent by taking a low sampling rate.**

***Index Terms*—End-to-end tracing, Weighted sampling, Microservice, Robust Random Cut Forest**

I. Introduction

With a microservice architecture, an application is decoupled into a group of loosely distributed fine-grained services with complex interactions [1], bringing great challenges to operate microservice systems. Distributed tracing plays an important role in profiling, diagnosing, and debugging microservice systems. By instrumenting the components of the system, trace data are generated to record the execution paths of a request. The trace data can be leveraged to obtain the complex dependencies between microservices as well as to detect and explain anomalous behavior. Although traces are of great help, they are often produced in a large volume and costly for storage. The microservice architecture often comprises hundreds to thousands of microservices. For example, WeChat system accommodates more than 3000 services running on over 20000 machines and the total amount of requests is normally $10^{10} \sim 10^{11}$ on a daily basis [2], which produces dozens of Tera Bytes trace data per day. However, most of them are similar and redundant. It is not necessary to store so many redundant traces. Only a fraction of traces that are helpful to operators are those from corner cases [3].

To reduce the storage cost of traces, some sampling techniques are proposed. For distributed tracing systems such as Zipkin[1] and Jaeger[2], they use head-based sampling to decide whether to keep the trace or not. The sampling decision is made at the beginning of the trace. Then the decision will be propagated with the request to the downstream services. The disadvantage of head-based sampling is the unawareness of what would happen in the following steps. Therefore, the traces are sampled randomly, which will limit the ability to preserve the informative traces. On the other hand, tail-based sampling mitigates the defect of head-based sampling by making sampling decisions at the end of the request. It is able to capture more informative traces by taking the message recorded in the traces like latency or HTTP status code (e.g., 404) into consideration. But the tail-based is still inadequate to detect uncommon traces since it focuses on the individual trace and ignores the difference between current traces and previous traces.

To find out uncommon traces, more attention should be paid to the differences between traces. Some sampling approaches [3], [4] have been proposed following this way. However, these methods only focus on the structural difference and pay no attention to the temporal difference. To make full use of these two types of attention, we design and develop Sieve, an online sampling approach for end-to-end trace data. Sieve uses robust random cut forest (RRCF) [5] which is a variant of the isolation forest [6] to detect uncommon traces. It will pay attention to the traces that are temporally or structurally different from other traces, and raise their sampling probabilities. To achieve the biased sampling scheme, Sieve firstly encodes a trace as a path vector [7] which is a useful form to express the temporal and structural difference and gives the vector an attention score, which represents how much attention Sieve pays to the trace. Finally, the attention score will be used to calculate the sampling probability of the trace.

To evaluate the effectiveness of Sieve, we perform experiments on four trace datasets from different microservice systems. The results show that Sieve is effective to detect uncommon traces and reduce the cost of storage to a large extent, robust to the degree of uncommonness and parameter settings, suitable for the sampling scenario in a large-scale

[1]https://zipkin.io/
[2]https://www.jaegertracing.io/

microservice system.

The contribution of this paper is four-fold shown as follows.

- We introduce the attention mechanism which focuses on the temporal and structural differences to capture uncommon traces.
- We propose the path vector encoding method to encode the trace into a vector that incorporates the temporal and structural information.
- We provide a biased sampling scheme based on attention score. The sampling probabilities of uncommon traces are high, while the ones of common traces are pretty low, which reduces the storage space significantly.
- We design and implement Sieve to conduct online sampling at a low cost. The effectiveness is evaluated in several microservice systems.

The rest of the paper is organized as follows. Section II presents our motivation. Section III depicts the design of Sieve in detail. Section IV introduces the implementation of Sieve. Section V shows the experimental evaluation. In section VI, we review the previous related work. Section VII concludes this paper.

II. MOTIVATION

According to the specification of OpenTracing[3], a trace is a direct acyclic graph including multiple spans which are correlated by the causal relationships. A span represents the work done by a service and records the execution latency, tags and logs, etc [8]. Trace data is a kind of semi-structured text varying from hundreds of bytes to millions of bytes. Moreover, they are produced at a high speed. A real-world telecommunication enterprise can produce more than 2 million traces in one minute and an electronic payment company can produce 50GB traces in one day. By analyzing these data, operators can uncover performance issues [9]–[11], detect anomalies [7], [12] and locate root causes of faults [8], [13]–[15]. The huge amount of trace data makes it too expensive for storage and analysis. Therefore, a sampling method that preserves the useful traces and discards the useless traces is necessary to reduce the overhead.

Trace data contain abundant information including structural information and temporal information. The structural information (i.e., calling relationship, number of spans) and the temporal information (i.e., request latency) are two indicators of unusual behavior. Existing sampling approaches do not take the structural and temporal information into account explicitly. It is problematic to judge a trace in such a way because the trace with high latency is not necessarily anomalous (e.g., a request may access a database) and the trace with a usual structure is not necessarily normal (e.g. the request latency is high). A microservice system usually handles many types of requests. Each type of request is related to a pattern. The pattern can be simply depicted as the combination of structural information and temporal information embedded in a trace. Here, we call such information as temporal and structural attention. The traces with the same pattern are produced by the same type of request. Hence, it is reasonable to compare the trace with other traces with the same pattern in order to find uncommon traces. Fig. 1 illustrates the temporal and structural attention.

[3]https://opentracing.io/

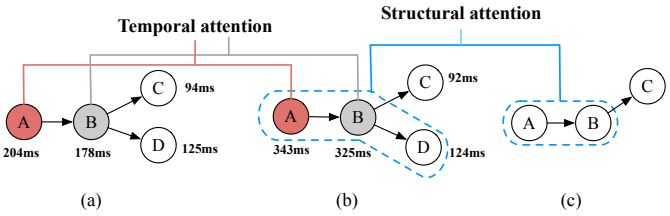


Fig. 1. An example of temporal and structural attention. (a) denotes a normal trace. (b) denotes a trace with temporal attention since the latency of Service A and B increases significantly. (c) represents a trace with structural attention since Service D is disconnected from Service B.

To develop an online sampler, we should solve the following challenges to balance the sampling quality and the sampling overhead.

- **Trace Representation.** A trace is a human-readable text that cannot be processed by algorithms directly. We should encode a trace into a form that can be processed by algorithms. The main principle of trace encoding is to find a representation that can reflect our attention to distinctive traces. Moreover, trace encoding should incorporate its structural and temporal information.
- **Biased Sampling with Attention.** The volume of trace data is huge, but the informative traces (i.e., uncommon traces) only take up a pretty small part. The uniform sampling scheme with a low sampling rate often picks out common traces and leaves out uncommon traces. A biased sampling scheme should be designed for such an unbalanced dataset to preserve more uncommon traces.
- **Constant Time and Space Complexity.** The design of an online sampler requires fast sampling and low memory consumption. The workload might be continuous and heavy, thus the sampling overhead should adapt to the workload volume.

III. SYSTEM DESIGN

This section describes the design of Sieve. Sampling aims to preserve traces that might have useful information for debugging or diagnosis and to discard redundant traces. In reality, the microservice system runs steadily most of the time. Therefore, most traces share common characteristics and are similar to each other. Uncommon executions are rare and we should pay more attention to them. How to pick out those rare patterns and sample them with high probabilities are our goals. Sieve makes use of the isolation-based approach to discover uncommon executions. To achieve that, Sieve uses a path vector encoder to extract the structural and temporal features from the trace data and builds path vectors. Then the path vector is sent to the adaptive scorer and gets an attention score. Finally, the biased sampler makes a sampling decision

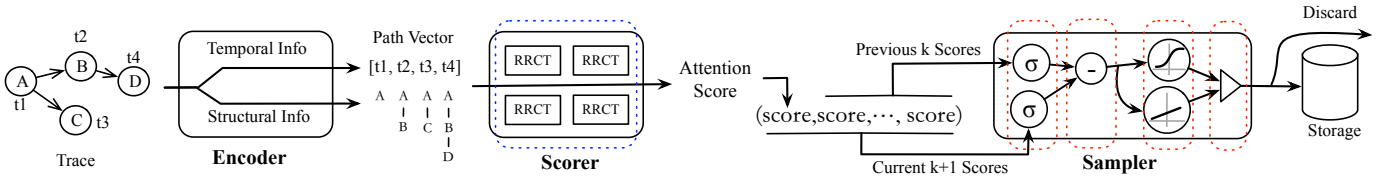


Fig. 2. The sampling workflow of Sieve. Every span of the trace is labeled by a letter and has a latency attribute. The core of the scorer module is a random cut forest which is comprised by RRCT (i.e., robust random cut tree). In the sampler module, the first red box contains the operators to calculate the variance of scores, the second red box contains an operator to conduct subtraction, the third red box contains the operators to map the attention score to the sampling probability in different ways, depending on the result of the subtraction operator. The fourth red box contains an operator to make a sampling decision. Finally, the filtered traces are stored persistently. While other traces are discarded.

based on the current score and the previous scores. Fig. 2 shows the overview of Sieve's sampling workflow.

A. Path Vector Encoder

A trace records the execution path of a request. The first span represents the request entry which is called a root span. Starting from the root span A, we can navigate to the child span B which depends on A and now we have a path $A \rightarrow B$. When we navigate to the child span C of B, we have another path $A \rightarrow B \rightarrow C$. After we traverse all the spans in the trace, we get a path set P, each of which starts from the root span A. For each path p in P, it is associated with the latency l_t of the tail span of path p. With the path set P and the corresponding latency set L of a trace, we can build the path vector $\mathbf{x} = (x_1, ..., x_n)$ of the trace. Each index of $\mathbf{x}$ is associated with a path. The index i associates with the path p_i and the value x_i in index i is assigned with the latency l_i. Different traces may contain different path sets. For a certain trace, if it does not contain the path p_i associated with the index i, then -1 is assigned to x_i of its path vector $\mathbf{x}$, which means it is an invalid index. Fig. 3 shows an example of path vector encoding. In this way, we incorporate structural and temporal information into a vector. The valid indexes of a path vector reveal the path set of the trace and the values in the valid indexes reveal the latency set of the trace. Therefore, when we compare two traces with different structures, the value in the invalid index will make them distinguishable. When we compare two traces with the same structure but different latency, the value in the valid index will make them distinguishable likewise.

Beyond the path and latency features, Sieve can extend the path vector to incorporate more meaningful features that help distinguish different types of traces. For example, the number of spans will add more information to help diagnose structural uncommonness. The request status code which is usually recorded in the trace, will play an important role in discovering anomalous traces. We only need to concatenate these features to form a more informative trace encoding and it will benefit the isolation procedure which will be introduced in the next subsection.

B. Adaptive Scorer

1) Isolation: Since traces are encoded into path vectors, finding out uncommon traces is equivalent to finding out

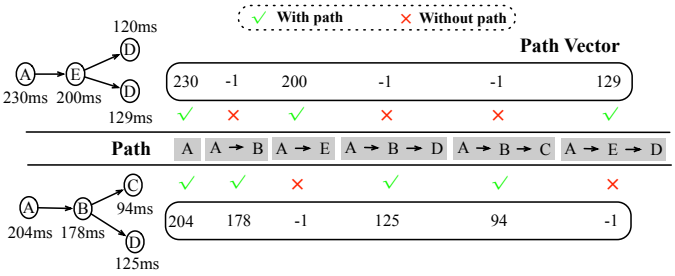


Fig. 3. The example of path vector encoding of two traces. Sieve extracts all the paths that starting from the root span. In the path vector, each index associates with one path, and the value in the index is the latency of the tail span of the associated path. If a trace does not contain a path, the value in the corresponding index is set to -1. If more than one same paths exist in a trace, the maximum latency amongst these paths is selected for encoding.

uncommon path vectors. The uncommonness means deviation from the majority. Therefore, the distributions of the uncommon and common traces are quite different. Sieve makes use of the difference and conducts a partitioning procedure to isolate the uncommon from the common traces. Given a path vector set $X \subset R^N$, we select a partition over some dimension to split X repeatedly until every path vector is isolated from each other. The uncommon path vectors in X will be isolated earlier and easier due to their difference and minority [6]. A tree is an appropriate data structure to perform isolation because we can partition X recursively from top to down and finally place every path vector in the leaf node. The length of a path from the root to a leaf node measures the number of partition conducted to isolate the path vector. Hence, the uncommon path vector will have a shorter path than that of the common path vector.

According to the above observations, we adopt RRCF to achieve the isolation of uncommon traces. RRCF is suitable for handling the outlier detection in the streaming data and the capability is leveraged by Sieve to discover uncommon traces. Sieve builds an enhanced RRCF model to figure out the attention scores of a trace. The original RRCF model can only process data with fixed dimensions while the path vector varies with different lengths. When there is one trace containing paths that never appeared before, the dimension will be extended by appending -1 to other path vectors. Making RRCF adapt to the dimension variation is one of our innovations.

RRCF is built on many RRCTs, which are built from scratch and finally grow to a fixed size. There are two stages for an

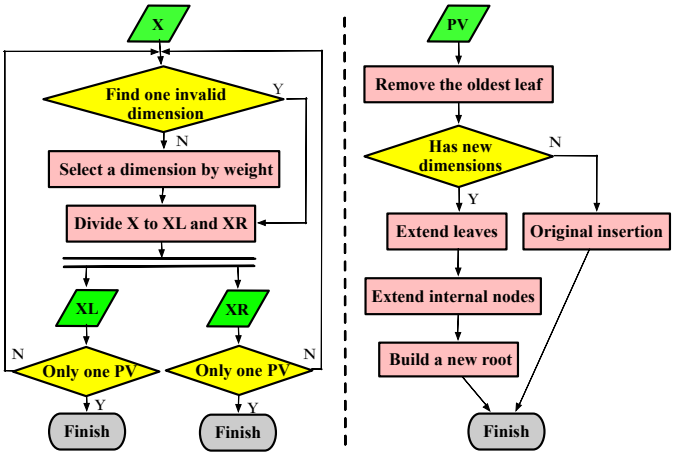


Fig. 4. The workflow of the construction stage (left) and the maintenance stage (right). X is a set of path vectors. XL and XR are the subsets of X. PV is the path vector.

RRCT, namely the construction stage and the maintenance stage. Fig. 4 shows the workflows of both stages. Suppose the final size of each RRCT is m and the dimension of the path vector is n, Sieve uses these m path vectors to build RRCTs in the construction stage. m path vectors form a path vector set $X = \{\mathbf{x}_i = (x_{i1}, ..., x_{in}) | \mathbf{x}_i \in X, 1 \leq i \leq m\}$ and each RRCT is built by partitioning X recursively. Starting from the root of an RRCT, Sieve selects a cutting dimension $j (1 \leq j \leq n)$ that is most likely to isolate the uncommon. The method to select cutting dimensions is another improvement on the original RRCF model. The structural distinction is considered first since this information is distinctive to distinguish different clusters of path vectors. A dimension j is named invalid dimension if, for some path vector, the index j is invalid. Sieve selects one invalid dimension as the cutting dimension. If there are no invalid dimensions, Sieve selected the cutting dimension according to the dimension weight. The weight of dimension j is calculated in the following way,

$$w_j = \frac{\underset{1\leq i\leq m}{max}\ x_{ij} - \underset{1\leq i\leq m}{min}\ x_{ij}}{\sum_{j=1}^{n} \underset{1\leq i\leq m}{max}\ x_{ij} - \underset{1\leq i\leq m}{min}\ x_{ij}}. \quad (1)$$

The probability that dimension j is selected is proportional to w_j, thus the dimension that has the largest difference will be selected with the highest probability. After selecting the cutting dimension j, a cutting value q_j is selected. If the cutting dimension j is an invalid index for some path vectors, the cutting value q_j is always set to -0.5. Otherwise, the cutting value q_j is selected between the maximum and minimum value in the selected dimension j randomly. The subset $XL = \{\mathbf{x}_i = (x_{i1}, ..., x_{in}) | \mathbf{x}_i \in X, x_{ij} \leq q_j\}$ whose value in index j is equal or less than q_j is assigned to the left child of root. The rest $XR = X \setminus XL$, whose value in index j is greater than q_j is assigned to the right child of the root. Then Sieve partitions the left child and right child in the above way until every leaf contains only one path vector. Fig. 5 shows the partition procedure of an RRCT.

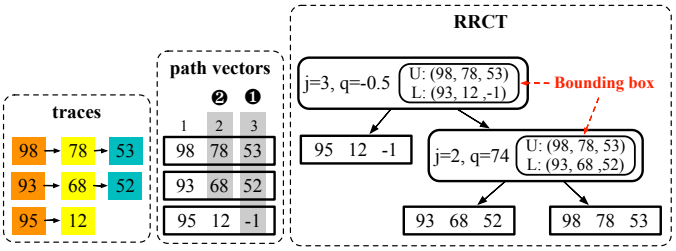


Fig. 5. Illustration of the building procedure of one RRCT. The blocks with different colors represent different spans. The number inside a block represents the latency. The dimension denoted by ❶ is an invalid index for the third path vector so dimension 3 is selected as the cutting dimension first. The first and second path vectors are partitioned again on the right child of RRCT's root. The dimension denoted by ❷ is selected since it is quite different from others. The bounding box records the upper and lower bound of the values on each dimension.

At the end of the construction stage, the maintenance stage begins. With an incoming path vector, the leaf containing the path vector of the trace produced earliest is removed. Besides the cutting dimension and the cutting value, each internal node of RRCT maintains a bounding box of its leaves. The bounding box records the upper and lower bound of the values on each dimension. With the removal of the oldest leaf, each RRCT is adjusted by replacing the parent of the removed leaf with the removed leaf's sibling and updating the bounding box of the internal node above the sibling. Fig. 6 shows an example of leaf removal. Then the incoming path vector is inserted into each RRCT. If the path vector has new dimensions, the insertion is handled in three steps. Firstly, Sieve extends the path vectors that are already in the tree by appending -1 to the path vectors. Secondly, Sieve extends the bounding boxes in the same way as path vector expansion. Finally, Sieve generates a new root with its cutting dimension set to one new dimension, its cutting value set to -0.5, its left child set to the old root, its right child set to the incoming path vector, and set the new root's bounding box. Fig. 7 shows the insertion of the incoming path vector with a new dimension. If the path vector has no new dimensions, Sieve performs the insertion using the original method of RRCT, which is referred to in [5].

The size of RRCT is constant during the maintenance stage since Sieve keeps the latest m path vectors by performing the removal and insertion operations on each RRCT. Therefore,

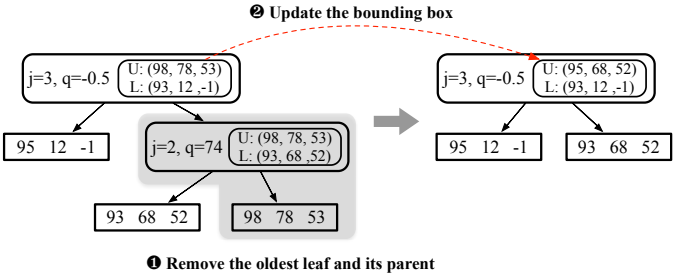


Fig. 6. Illustration of the removal of the oldest leaf.

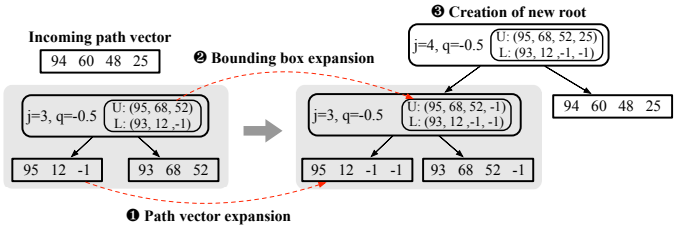


Fig. 7. Illustration of the insertion of path vector with a new dimension.

Sieve evaluates a path vector in the context of the distribution of the latest m path vectors, so Sieve can adapt to the changes of the traces.

2) Attention Score: After the construction of RRCT, uncommon path vectors will have a shorter path from the root to leaf. Therefore, they are at the shallower layers of the tree. RRCT assigns an attention score to a path vector according to its depth in the tree. Intuitively, the significance of uncommonness should increase as the tree becomes deeper. Given $depth(l,t)$ which gets the depth of leaf l in tree t, the attention score s_i of the path vector $\mathbf{x}_i$ in leaf l' of tree t_j is calculated as follows

$$s_i = \frac{\underset{l \in Leaves}{max} \ depth(l, t_i)}{depth(l', t_j)}. \tag{2}$$

Each RRCT in RRCF represents a different partition scheme for the trace data. When a path vector gets a high score from most of the trees, it is likely to be an uncommon one with a high probability. The final score assigned to the path vector is calculated by averaging the scores given by all the RRCTs. Given an RRCF including k RRCTs, the final score of a path vector is

$$s = \frac{s_1 + ... + s_k}{k}. \tag{3}$$

3) Dimension Reduction: Sieve is designed to be adaptive to the new traces with execution paths that never existed before. With the path vector expansion mechanism introduced in section III-B1, RRCT can handle path vectors with new dimensions. These path vectors will be isolated from other path vectors. However, the path vector expansion mechanism has a side effect and will cause the dimension of the path vector to continue to grow. The curse of dimensionality impacts Sieve's online sampling ability severely so the dimension reduction technique is necessary.

The cause of the dimensionality curse is that Sieve records all execution paths it has seen up to now, while the RRCT only keeps the path vectors of the most recent traces. The execution paths of these traces account for a small part of the total execution paths Sieve has seen. Since one execution path is associated with one dimension, there are plenty of dimensions that are invalid for all the path vectors in the RRCT. In Fig. 8, the left figure shows the drastic growth of the invalid dimensions, especially when an RRCT is of a small size. All these invalid dimensions make no influence on the partition because they have never been selected as a cutting dimension. Hence, Sieve can remove these invalid dimensions safely.

Besides the removal of the dimensions that are invalid for all the path vectors in the RRCT, Sieve adopts a more aggressive approach to remove the dimension that satisfies the following two criteria: the dimension is not the cutting dimension; the variance of the values in the dimension is lower than 0.1. The dimensions meeting the criteria have a tiny chance to be selected as the cutting dimension. We name them as weak dimensions due to their almost zero contribution to the partition. Note that the invalid dimensions that Sieve can remove safely are a kind of weak dimension. Sieve treats them differently. The right figure in Fig. 8 shows the number of weak dimensions during the sampling procedure of 1000 traces. Different from the complete removal of the invalid dimensions, Sieve temporarily removes the weak dimensions and keeps monitoring the variance of the values in the weak dimensions. When the variance is greater than 0.1, the weak dimension is changed to the normal dimension and all the path vectors in the RRCT will be extended with such a normal dimension.

C. Biased Sampler

Sieve calculates the sampling probability of a trace based on its attention score and the scores of previous traces. Sieve keeps a sliding window containing k most recent scores and the current score. In the sliding window, Sieve calculates the variance var_k of the past k scores and the variance var_{k+1} of the $k+1$ scores. The difference between var_{k+1} and var_k indicates the deviation of current score s_{k+1} from the distribution of the past k scores. If the difference degree exceeds a threshold h, Sieve adopts a sigmoid function to greatly raise the sampling probability of the current trace. Otherwise, Sieve calculates the sampling probability of the current trace in proportion to its weight in the sliding window. Given a sliding window $W = [s_1, ..., s_{k+1}]$, the sampling probability of the current trace is calculated in the following way where $\overline{W}$ is the mean value of W:

$$f(s_{k+1}) = \begin{cases} \dfrac{1}{1 + e^{2\overline{W} - s_{k+1}}} & \text{if } var_{k+1} - var_k > h, \\ \dfrac{s_{k+1}}{\sum_{i=1}^{k+1} s_i} & \text{if } var_{k+1} - var_k \leq h \end{cases} \tag{4}$$

The trace whose score has a significant increase to the variance in the sliding window will get a sampling probability close to 1. The common trace gets a low sampling probability due to its low attention score. It seems that the selection of threshold h plays an important role in controlling the sample size. We show the impact of threshold in the evaluation section. After getting the sampling probability $f(s_{k+1})$, Sieve generates a random number between $[0, 1]$ and compares it to $f(s_{k+1})$. If $f(s_{k+1})$ is no less than the random number, Sieve samples the trace, otherwise Sieve drops it.

D. Online Sampling

Sieve is a real-time online sampler to achieve a high sampling probability for the uncommon trace. When the streaming

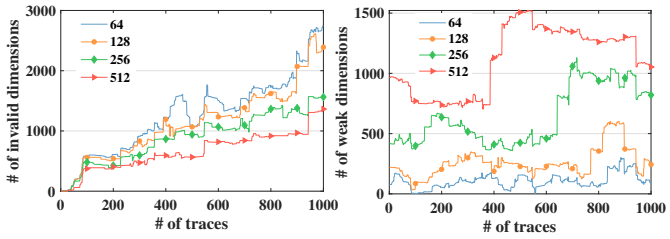


Fig. 8. The number of invalid dimensions and weak dimensions. Different lines represent different RRCT sizes. The x-axis shows the number of traces that are processed by Sieve.

Algorithm 1 Online Sampling Algorithm

Input: The trace t; the RRCF $rrcf$; the sliding window sw of the previous k scores; the threshold h.

Output: The sampling decision $decision$; the updated sliding window sw; the updated RRCF $rrcf$.

$x \leftarrow encode(t)$
$scores \leftarrow []$
for all $rrct \in rrcf$ **do**
    $rrct.remove\_oldest()$
    $rrct.insert(x)$
    $s \leftarrow rrct.score(x)$
    $scores.append(s)$
$avg \leftarrow average(scores)$
$sw.append(avg)$
if $variance(sw) - variance(sw[1:k]) > h$ **then**
    $p \leftarrow \frac{1}{1 + e^{2 * average(sw) - avg}}$
else
    $p \leftarrow \frac{avg}{\sum_{i=1}^{k+1} sw[i]}$
$sw.remove(sw[0])$
if $random(0, 1) < p$ **then**
    $decision \leftarrow True$
else
    $decision \leftarrow False$
return $decision, sw, rrcf$

traces come, Sieve collects enough traces, encodes them into path vectors, and constructs the RRCF, then Sieve enters the maintenance stage. With more incoming traces, before inserting a new coming trace, the oldest trace is removed. For every trace inserted into the trees, the attention score is evaluated by all RRCTs of the RRCF. Next, the sampling probability is calculated using equation (4). Finally, Sieve samples a trace according to its sampling probability. Algorithm I describes the sampling procedure at the maintenance stage.

IV. IMPLEMENTATION

We have implemented Sieve in python based on an open-source implementation[4] of RRCF. We modify the original RRCF model to fit the online sampling scenario. We improve the cutting dimension selection method by giving priority to invalid dimensions and new dimensions. Therefore, the structural uncommonness is considered first to accelerate isolation. We replace the original scoring scheme with our scheme which is introduced in Section III-B2 because of the poor performance of the original one. We enhance the RRCF model with the path vector expansion mechanism to enable the ability to process vectors of indefinite length, which is introduced in III-B1. To solve the curse of dimensionality, we implement the dimension reduction scheme by removing the invalid dimensions and weak dimensions, which is introduced in III-B3.

V. EVALUATION

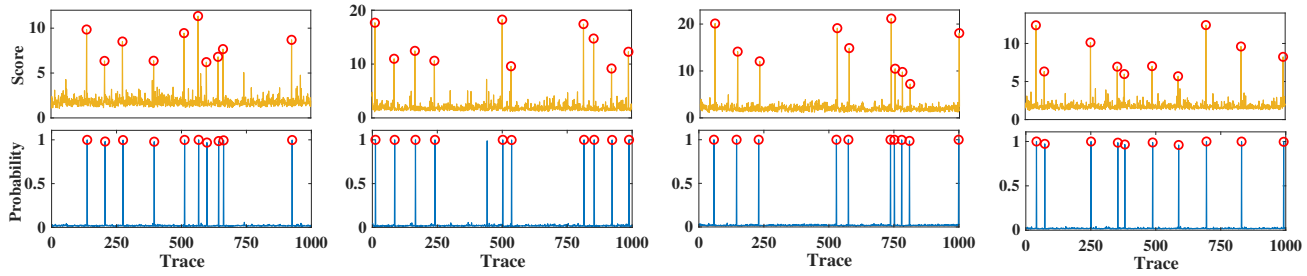
In this section, we carry out evaluations on four trace datasets to show the effectiveness of Sieve. In addition, we evaluate Sieve's sensitivity to the degree of uncommonness and its parameters including the number and size of RRCT, the threshold. To evaluate the performance of Sieve, we compare its sampling result to the hierarchical clustering method proposed in [3]. By default, the number of RRCT is set to 50, and the size of RRCT is set to 128; the length of sliding window is set to 50 and the threshold is set to 0.3. These parameters can be tuned in different systems. The datasets we use are illustrated as follows:

- **A simulated microservice system.** The traces are generated by Virtual War Room (VWR) [16]. We use VWR to simulate a microservice system composed of 6 microservices and inject two types of faults (i.e., network delay and early stop) into it, collect trace data with OpenTracing. There are 34167 traces in total, 32592 of them are normal, 66 of them are with high network delay, 1509 of them are with early stop. The name VWR is used to refer to this dataset in the following part.
- **A Real-world microservice system.** The dataset is generated by a real microservice application deployed in the private cloud environment of an ISP and prepared for an international AIOps challenge[5]. There are 13 microservices in the system, some of which have multiple instances. Different types of faults are injected into different instances to make diverse anomalies. The dataset has 164340 normal traces and 4092 abnormal traces. The name AIOps is used to refer to this dataset in the following part.
- **Online Boutique microservice benchmark.** Online Boutique is a microservice demo application consists of a 10-tier microservice. We instrumented the application with OpenCensus[6] and deployed it on a Kubernetes cluster with 8 nodes. We run the workload on the benchmark to emit traces and two types of faults were injected. One fault will delay the response of an email microservice and the other will break down the product catalog microservice. 2000 traces are collected and 99

[4]https://github.com/kLabUM/rrcf

[5]http://iops.ai/dataset_list/

[6]https://opencensus.io/



(a) VWR (b) AIOps (c) Boutique (d) TC

Fig. 9. Biased Sampling towards temporal uncommonness. The red circle denotes the attention scores and the sampling probability of the uncommon traces.

TABLE I
LATENCY SETTING

Dataset	Spans	Latency	
		Common	***Uncommon***
VWR	6	200 $\sim$ 400ms	$>$ 100000ms
AIOps	58	100 $\sim$ 200ms	500 $\sim$ 1000ms
Boutique	28	$<$ 200ms	$>$ 500ms
TC	15	22 $\sim$ 30ms	50 $\sim$ 66ms

traces are anomalous. The name Boutique is used to refer to this dataset in the following parts.

- **Real production traces.** This dataset comprises 6561 traces from a large telecommunication enterprise. The traces can be grouped into 10 different API types. The longest trace contains 451 spans. The name TC is used to refer to this dataset in the following part.

A. Biased Sampling

We first show the ability of our solution to bias sampling towards uncommon traces which often indicate anomalies. We show how Sieve makes use of temporal attention and structural attention to find out uncommon traces.

Temporal attention. We replay 1000 traces which consist of 990 traces whose latency is in a normal range and 10 traces whose latency deviates from the normal distribution. The structures (i.e. the number of span, causal relationship) of these 1000 traces are the same. The specific settings of the different datasets are listed in Table I.

Fig. 9(a) shows the attention scores and the sampling probabilities. It shows that uncommon traces have higher scores because they are easier to be isolated and placed in the shallower layer of the tree. The sampling probability is extremely high for uncommon traces, namely 0.990 on average. Conversely, the sampling probability of common traces is very low and the average is 0.018. The significant difference between the sampling probabilities is due to the different policies we take when the difference of score variances is above/below the threshold. When a score makes the difference of variances exceed the threshold, the sigmoid function makes the sampling probability converge to 1 rapidly. For most traces, their sampling probabilities are calculated linearly and are proportional to the length of the sliding window. Therefore, the sampling probabilities of the common traces fluctuate around 0.02 when the sliding window is of length 50.

We conduct a similar experiment on the AIOps dataset. Fig. 9(b) shows the scores and sampling probabilities. The attention scores of the minority are significantly higher than that of the majority. The sampling probability of the minority is 0.999 on average and the counterpart of the majority is 0.019. In the second figure of Fig. 9(b), we notice one common trace has a sampling probability close to 1. The reason for this false positive is the selection of sub-optimal threshold which is 0.3 in the experiment. But the sub-optimal threshold only adds a few more samples and has little effect on the sampling result, which we will show later.

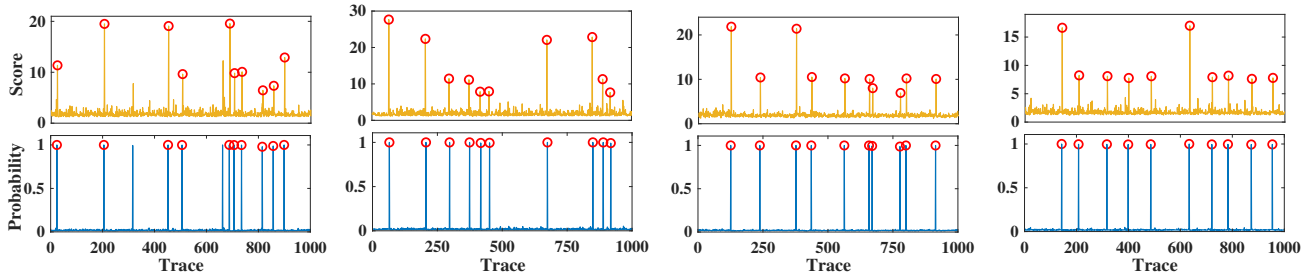
We conduct a similar experiment again on Boutique and TC respectively. Fig. 9(c) shows the result of Boutique and Fig. 9(d) shows the result of TC. Both figures illustrate that the Sieve detects all the uncommon traces and raises their sampling probabilities enormously. Even for the small gap between the common and uncommon like the latency setting of TC, Sieve is effective to distinguish them.

Structural attention. We evaluate the effectiveness of Sieve to uncommon traces with structural attentions. 1000 traces including 990 structurally common traces and 10 structurally uncommon traces are replayed. To avoid the impact of temporal deviations, the latency of the traces is restricted within a specific range. Table II shows the settings of traces from different datasets. The 1000 traces are shuffled and then fed into Sieve.

Fig. 10(a) shows attention scores and sampling probabilities of traces from VWR. The sampling probabilities of common traces are much lower than that of uncommon traces. The average of the former is 0.020, while the latter is 0.996. There are two false positives in the second figure. When we inspect

TABLE II
TRACE STRUCTURE SETTING

Dataset	Latency	Spans	
		Common	***Uncommon***
VWR	$<$ 200ms	6	{4,5}
AIOps	400 $\sim$ 450ms	58	59
Boutique	$<$ 200ms	28	18
TC	25 $\sim$ 35ms	15	14



(a) VWR (b) AIOps (c) Boutique (d) TC

Fig. 10. Biased Sampling towards structural uncommonness. The red circle denotes the attention score and the sampling probability of uncommon traces.

the traces corresponding to the two false positives, we find them both have extremely low latency below 10ms which makes them deviate from the distribution of latency. Hence, Sieve regards the two traces are uncommon and gives them high attention scores.

We adopt the AIOps dataset to conduct a similar experiment. In Fig. 10(b), there are no false positives due to the small range of latency. The averaged sampling probabilities of the common traces is 0.018 and the counterpart of the uncommon traces is 0.997.

We experiment again on the trace from Boutique. Fig. 10(c) shows that all uncommon traces are detected by Sieve and will be sampled with a high probability. The average sampling probabilities of the common and the uncommon are 0.018 and 0.997 respectively.

For the TC dataset, we select 990 traces with 15 spans as the common ones and 10 traces with 14 traces as the uncommon ones. The latencies of them are within 25ms to 35ms. The uncommon and the common are almost the same in structure, except that the common has one more span. Fig. 10(d) shows that Sieve detects all the uncommon traces. Therefore, we conclude that Sieve is sensitive to the subtle difference in structure.

B. Sensitivity

We evaluate Sieve's sensitivity to the degree of uncommonness and its parameters including the number and size of RRCT, and the threshold. We focus on different degrees of temporal deviations because it is more challenging to sample the temporal uncommonness which does not severely deviate from the distribution than to sample the structural uncommonness. In the following experiments, we regard the uncommon traces, whose attention scores make the difference of variance exceed the threshold, as True Positive (TP), and the common traces, whose attention scores make the difference of variance exceeds the threshold, as False Positive (FP).

We select 1000 traces from AIOps with different ranges of latency for the common and the uncommon traces. To avoid the effect of structural deviation, the traces we selected is of the same structure. The latency of 990 traces is below 200ms. The latency of the rest 10 traces is above 200ms. We randomly insert the 10 traces into the 990 traces. Then the 1000 traces

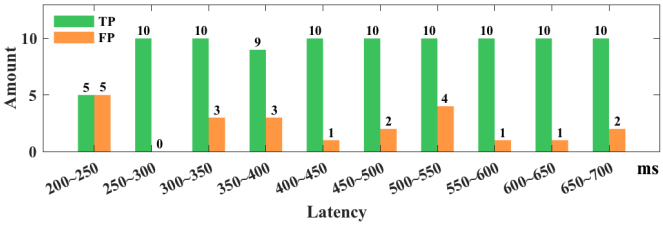


Fig. 11. Sieve's sensitivity to the degree of uncommonness.

TABLE III
TRACE COMPOSITION

Type	Spans	Latency	Amount
common	58	$0 \sim 300ms$	990
uncommon	58	$> 400ms$	5
	59	$0 \sim 300ms$	2
	7	$0 \sim 300ms$	3

are streaming to Sieve and we record the number of TP and FP. To vary the degree of uncommonness, we change the latency range of the uncommon traces and repeat the steps above. Common traces and the position of insertion remain unchanged.

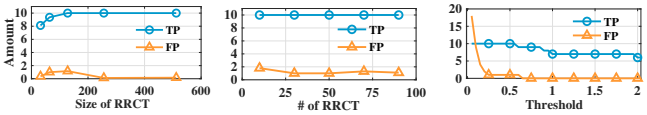
Fig. 11 demonstrates true positives and false positives that Sieve recognizes under various degrees of uncommonness. Sieve detects all the 10 uncommon traces in most scenarios. The scenario in which Sieve performs not so well is in the range $200 \sim 250$ms. Since the latency of the common traces is below 200ms, the boundary between common and uncommon is fuzzy. As the difference becomes more distinguishable, Sieve improves its detection ability rapidly and keeps the false positive rate at a low level.

Our next set of experiments evaluate Sieve's sensitivity to its parameters including the number and size of RRCT, the threshold. We select 1000 traces from AIOps. The composition of 1000 traces is listed in Table III. 990 common traces with 58 spans whose latency is below 300ms are selected. The rest 10 traces consist of two types of traces. 5 temporally uncommon traces with 58 spans have latency above 400ms. The latency of 2 structurally uncommon traces with 59 spans and 3 structurally uncommon traces with 7 spans is below 300ms.

Fig. 12(a) shows the results when the size of RRCT is set

TABLE IV
TRACE COMPOSITION OF DIFFERENT DATASET

Type	VWR		AIOps		Boutique		TC		Amount
	Spans	Latency	Spans	Latency	Spans	Latency	Spans	Latency	
common	6	$0 \sim 200ms$	58	$0 \sim 300ms$	28	$0 \sim 200ms$	29	$0 \sim 40ms$	990
uncommon	6	$> 100000ms$	58	$> 400ms$	28	$> 500ms$	29	$60 \sim 90ms$	5
	5	$0 \sim 200ms$	59	$0 \sim 300ms$	18	$0 \sim 200ms$	30	$0 \sim 40ms$	5
	7	$0 \sim 300ms$	4	$0 \sim 200ms$					



(a) Size of RRCT (b) Number of RRCT (c) Threshold

Fig. 12. Sieve's sensitivity to the its parameters.

to 32, 64, 128, 256, 512 respectively. Fig. 12(b) presents the results when the number of RRCT is set to 10, 30, 50, 70, 90 respectively. Fig. 12(c) shows the results of different thresholds varying from 0.05 to 2.05. The results of Fig. 12(a) and Fig. 12(b) illustrate that the size and number of RRCT have a little effect on its accuracy and the performance of Sieve is stable under different configurations. But the threshold has a direct influence on Sieve's accuracy. When the threshold is lower than 0.55, the number of FP decreases significantly and the number of TP remains the maximum. When the threshold is between 0.6 and 0.85, the number of the TP remains stable with a slight decrease. And the number of FP decreases to 0. When the threshold is above 1, the number of TP decreases to 7 and remains stable. From the above results, we can conclude that Sieve is not sensitive to the threshold in a limited range and it is encouraged to set a low threshold to get a high coverage of TP with a small increase of FP.

C. Performance Comparison

To evaluate the sampling quality of Sieve, we compare the sampling result of Sieve to that of the hierarchical clustering (HC) method. The composition of the 1000 traces for different datasets is listed in Table IV. For the sake of comparison, we set a sampling budget, i.e., the sample size, and sample traces until the budget runs out. For Sieve, it is designed for online sampling and is not restricted by a budget inherently. To make use of the budget, Sieve samples traces in the following way. If Sieve runs out of the budget, then it stops processing the remaining traces; if Sieve still has a budget after the 1000 traces are processed, then it samples traces randomly from the traces which are not sampled until the budget runs out. We vary the budget from 10 to 100 and measure the quality of sampling by calculating the proportion of uncommon traces in the sample. We repeat experiments 50 times with the same setting and record the average proportion.

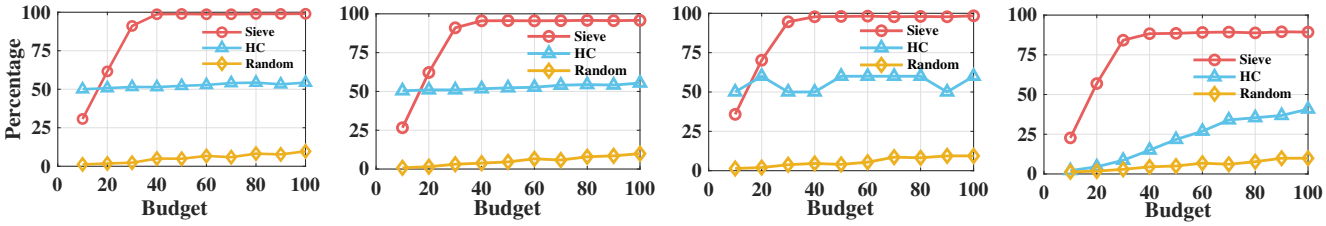
Fig. 13 shows the proportion of uncommon traces sampled by random, HC, and Sieve, under different settings of budget. Sieve achieves the best performance in four datasets. The proportion of random sampling is approximately equal to the sampling rate. With the increase of budget, the proportion of HC grows slowly and does not exceed 60%. Since HC can only detect the structural uncommonness, 5 structurally uncommon traces are sampled and the other 5 temporally uncommon traces are omitted. For Sieve, the proportion is lower than that of HC at the beginning because of its online sampling scheme. Though the sampling probabilities of common traces are low, they have a huge quantity in the trace stream and will consume a large part of the budget if the uncommon traces do not occur in the early stage. As the budget increases, the proportion of Sieve grows and converges closely to 100% rapidly. The result on TC is different from other datasets for the reason that neither the temporal uncommonness nor the structural uncommonness is not explicitly considered. The effectiveness of HC is largely reduced, while Sieve still has an ideal performance.

D. Representative sampling

We evaluate Sieve's ability to obtain a representative sample, which comprises traces of different patterns. Traces of TC can be divided into 10 groups. We use 6561 traces of TC as the population and test Sieve's ability to get a sample from different groups. In order to compare Sieve with HC and random sampling, we set the sample size of the latter two methods to the number of samples that Sieve produces. The experiment is repeated 50 times. Table V shows the number of traces in different groups and the sampling results using different methods. Sieve is more effective to preserve uncommon traces than HC. For the very few traces, Sieve preserves most of them. On the contrary, Sieve drops most of the common traces. To evaluate Sieve's storage saving, we conduct experiments with the other three datasets to observe the sampling rate. Table VI records sampling results. The results show that the reduction of traces is up to 97.5%, which brings considerable storage savings.

E. Overhead

We measure the time Sieve spends on sampling. The number of RRCT is set to 20 for its approximate performance to that of the setting of 50. Other parameters remain unchanged. The sampling latency comprises three parts, namely the time spending on path vector encoding, the time spending on calculating attention score, the time spending on calculating



(a) VWR (b) AIOps (c) Boutique (d) TC

Fig. 13. The proportion of uncommon traces sampled by three different methods in different trace datasets.

TABLE V
RESULTS OF REPRESENTATIVE SAMPLING

	API-1	API-2	API-3	API-4	API-5	API-6	API-7	API-8	API-9	API-10
Population	11	16	48	119	135	210	325	571	607	4519
Sieve	11	16	41.2	64	51	31.2	9.4	15.2	10.8	65.4
HC	5.9	16	5.44	17.22	19.44	32.64	42.24	45.92	42.84	87.56
Random	0.42	0.68	2.3	5.38	6.34	10.78	16.12	27.28	29.7	217

TABLE VI
SAMPLING RATES OF DIFFERENT DATASETS

	Population	Sample	Sampling Rate
VWR	34167	85	2.5%
AIOps	168432	7602	4.5%
Boutique	2000	114	5.9%

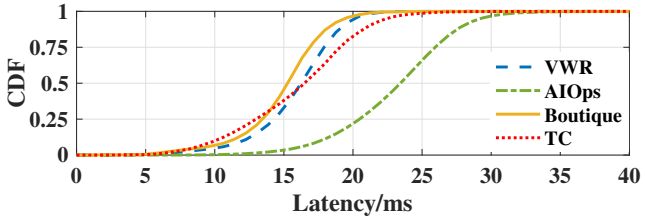


Fig. 14. The cumulative distribution of sampling latency.

sampling probability. Fig. 14 shows the cumulative distribution function of latency when Sieve processes different datasets. The latency varies between 3ms and 33ms. Even for the production traces with hundreds of spans, Sieve is efficient to process them.

VI. RELATED WORK

Debugging the distributed systems is notoriously challenging due to the interactions between various components which may run on different machines. Numerous distributed tracing systems are designed to help understand the systems and provide more observability when something wrong happens.

Magpie [17] is able to capture the control flow and resource consumption of requests to construct a concise workload model of the distributed system. Mike Y.Chen et al. [18] have proposed a request path based method to manage failure and evolution in large distributed systems. They apply the statistical technique to the collected path to infer system behavior. X-Trace [19] is a tracing framework that focuses on tracing applications at different network layers, to provide a comprehensive view of the system's behavior. Dapper [8] is Google's production distributed systems tracing platform and shares conceptual similarities with Magpie and X-Trace. Dapper has achieved a number of new features including sampling and the degree of application-level transparency to make it more appropriate in production. In industry, open-source or commercial distributed tracing systems have emerged following the tracing model of Dapper, such as Jaeger, Lightstep. Systems like [10], [13], [20]–[24] detect the performance degradation and pinpoint the root cause. MEPFL [15] trains prediction models to predict latent error and locate fault for microservice applications by making use of the trace level and microservice level features extracted from the trace log. Some of these features may be helpful for Sieve to improve its ability of uncommonness detection.

Sampling becomes necessary in distributed tracing. Faced with the high volume of trace data, Dapper uses a random sampling scheme with a sampling rate less than 0.1%. Though Dapper reduces the storage cost enormously in this way, the informative traces that account for a tiny part will lose. JCallGraph [25] only samples successful traces and preserves all the traces of failure invocation. Although it reduces enormous network bandwidth consumption on trace transmission, but the traces with high latency will be lost. Martin Bauer et al. [26] introduce trace sampling into conformance checking. They continuously sample traces until no trace with new information for conformance is found. The hierarchical clustering sampling scheme [3] is able to bias the sampling to maximize the diversity of traces in terms of the number of spans. It is neither a proper solution for online sampling nor unable to detect traces with temporal uncommonness. Sifter [4] uses traces to builds a low-dimension model to approximate the system's common-case behavior and bias sampling towards traces that are poorly captured by the model. Since Sifter only focuses on the structure of traces, the traces with uncommon temporal characteristics will be ignored. In the database literature, the problem of sampling a small portion of data to represent the

whole dataset exists as well. Ying Yan, et al. proposed an error-bounded stratified sampling scheme to reduce the sample size with the knowledge of data distribution [27].

VII. CONCLUSION

Distributed tracing in microservice systems becomes common. Moreover, sampling is essential to reduce the expense of storage facing trace data with several Tera Bytes per day. We design and implement Sieve to fulfill the requirements of biased online sampling. With the attention mechanism, Sieve is sensitive to temporally and structurally uncommon traces and samples them in a high probability, while most common traces are discarded to reduce the redundancy. The evaluation of different trace datasets shows that Sieve is robust to the degree of uncommonness. Compared to state-of-the-art approaches, Sieve can increase the probability of uncommon traces and reduce storage space tremendously. Moreover, the overhead of Sieve is low enough to work in real-time sampling.

ACKNOWLEDGMENT

The research is supported by the National Key Research and Development Program of China (2019YFB1804002), the Key-Area Research and Development Program of Guangdong Province (No. 2020B010165002), the National Natural Science Foundation of China (No. 61802448, No. U1811462), the Basic and Applied Basic Research of Guangzhou (No. 202002030328), and the Natural Science Foundation of Guangdong Province (No. 2019A1515012229). The corresponding author is Pengfei Chen.

REFERENCES

[1] J. Lin, P. Chen, and Z. Zheng, "Microscope: Pinpoint performance issues with causal graphs in micro-service environments," in *Service-Oriented Computing*. Springer International Publishing, 2018, pp. 3–20.

[2] H. Zhou, M. Chen, Q. Lin, Y. Wang, X. She, S. Liu, R. Gu, B. C. Ooi, and J. Yang, "Overload control for scaling wechat microservices," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2018, p. 149–161.

[3] P. H. B. Las-Casas, J. Mace, D. O. Guedes, and R. Fonseca, "Weighted sampling of execution traces: Capturing more needles and less hay," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*. ACM, 2018, pp. 326–332.

[4] P. H. B. Las-Casas, G. Papakerashvili, V. Anand, and J. Mace, "Sifter: Scalable sampling for distributed traces, without feature engineering," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*. ACM, 2019, pp. 312–324.

[5] S. Guha, N. Mishra, G. Roy, and O. Schrijvers, "Robust random cut forest based anomaly detection on streams," in *Proceedings of the 33nd International Conference on Machine Learning (ICML)*, vol. 48, 2016, pp. 2712–2721.

[6] F. T. Liu, K. M. Ting, and Z.-H. Zhou, "Isolation-based anomaly detection," *ACM Trans. Knowl. Discov. Data*, vol. 6, no. 1, Mar. 2012.

[7] P. Liu, H. Xu, Q. Ouyang, R. Jiao, Z. Chen, S. Zhang, J. Yang, L. Mo, J. Zeng, W. Xue, and D. Pei, "Unsupervised detection of microservice trace anomalies through service-level deep bayesian networks," in *31st IEEE International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2020, pp. 48–58.

[8] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," Tech. Rep., 2010.

[9] S. Kavulya, S. Daniels, K. R. Joshi, M. A. Hiltunen, R. Gandhi, and P. Narasimhan, "Draco: Statistical diagnosis of chronic problems in large distributed systems," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE Computer Society, 2012, pp. 1–12.

[10] Y. Gan, Y. Zhang, K. Hu, D. Cheng, Y. He, M. Pancholi, and C. Delimitrou, "Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2019, pp. 19–33.

[11] Y. Gan, S. Dev, D. Lo, and C. Delimitrou, "Sage: Leveraging ml to diagnose unpredictable performance in cloud microservices," *ML for Computer Architecture and Systems*, 2020.

[12] M. M. Z. Zadeh, M. Salem, N. Kumar, G. Cutulenco, and S. Fischmeister, "Sipta: Signal processing for trace-based anomaly detection," in *2014 International Conference on Embedded Software (EMSOFT)*. ACM, 2014, pp. 6:1–6:10.

[13] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. A. Brewer, "Pinpoint: Problem determination in large, dynamic internet services," in *2002 International Conference on Dependable Systems and Networks (DSN)*. IEEE Computer Society, 2002, pp. 595–604.

[14] Z. Cai, W. Li, W. Zhu, L. Liu, and B. Yang, "A real-time trace-level root-cause diagnosis system in alibaba datacenters," *IEEE Access*, vol. 7, pp. 142 692–142 702, 2019.

[15] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, D. Liu, Q. Xiang, and C. He, "Latent error prediction and fault localization for microservice applications by learning from system trace logs," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE*. ACM, 2019, pp. 683–694.

[16] H. Chen, P. Chen, and G. Yu, "A framework of virtual war room and matrix sketch-based streaming anomaly detection for microservice systems," *IEEE Access*, vol. 8, pp. 43 413–43 426, 2020.

[17] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using magpie for request extraction and workload modelling," in *6th Symposium on Operating System Design and Implementation (OSDI)*. USENIX Association, 2004, pp. 259–272.

[18] M. Y. Chen, A. J. Accardi, E. Kiciman, D. A. Patterson, A. Fox, and E. A. Brewer, "Path-based failure and evolution management," in *1st Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX, 2004, pp. 309–322.

[19] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "X-trace: A pervasive network tracing framework," in *4th Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX, 2007.

[20] P. Reynolds, C. E. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat, "Pip: Detecting the unexpected in distributed systems," in *3rd Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX, 2006.

[21] R. R. Sambasivan, A. X. Zheng, M. D. Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. R. Ganger, "Diagnosing performance changes by comparing request flows," in *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2011.

[22] E. Thereska, B. Salmon, J. D. Strunk, M. Wachs, M. Abd-El-Malek, J. C. López-Hernández, and G. R. Ganger, "Stardust: tracking activity in a distributed storage system," in *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS/Performance*. ACM, 2006, pp. 3–14.

[23] M. Ma, J. Xu, Y. Wang, P. Chen, Z. Zhang, and P. Wang, "Automap: Diagnose your microservice-based web applications automatically," in *The Web Conference (WWW)*. ACM / IW3C2, 2020, pp. 246–258.

[24] X. Zhou, X. Peng, T. Xie, J. Sun, W. Li, C. Ji, and D. Ding, "Delta debugging microservice systems," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*. ACM, 2018, pp. 802–807.

[25] H. Liu, J. Zhang, H. Shan, M. Li, Y. Chen, X. He, and X. Li, "Jcallgraph: tracing microservices in very large scale container cloud platforms," in *International Conference on Cloud Computing*. Springer, 2019, pp. 287–302.

[26] M. Bauer, H. Van der Aa, and M. Weidlich, "Estimating process conformance by trace sampling and result approximation," in *International Conference on Business Process Management*. Springer, 2019, pp. 179–197.

[27] Y. Yan, L. J. Chen, and Z. Zhang, "Error-bounded sampling for analytics on big sparse data," *Proc. VLDB Endow.*, vol. 7, no. 13, pp. 1508–1519, 2014.