Practical Root Cause Localization for Microservice Systems via Trace Analysis

Zeyan Li*¶, Junjie Chen†, Rui Jiao*¶, Nengwen Zhao*¶,
Zhijun Wang‡, Shuwei Zhang‡, Yanjun Wu‡, Long Jiang‡, Leiqin Yan‡,
Zikai Wang§, Zhekang Chen§, Wenchi Zhang §, Xiaohui Nie*¶, Kaixin Sui§, Dan Pei*¶,
*Tsinghua University, †College of Intelligence and Computing, Tianjin University, ‡China Minsheng Bank, §BizSeer
¶Beijing National Research Center for Information Science and Technology (BNRist)

***Abstract*—Microservice architecture is applied by an increasing number of systems because of its benefits on delivery, scalability, and autonomy. It is essential but challenging to localize root-cause microservices promptly when a fault occurs. Traces are helpful for root-cause microservice localization, and thus many recent approaches utilize them. However, these approaches are less practical due to relying on supervision or other unrealistic assumptions. To overcome their limitations, we propose a more practical root-cause microservice localization approach named *TraceRCA*. The key insight of *TraceRCA* is that a microservice with more abnormal and less normal traces passing through it is more likely to be the root cause. Based on it, *TraceRCA* is composed of trace anomaly detection, suspicious microservice set mining and microservice ranking. We conducted experiments on hundreds of injected faults in a widely-used open-source microservice benchmark and a production system. The results show that *TraceRCA* is effective in various situations. The top-1 accuracy of *TraceRCA* outperforms the state-of-the-art unsupervised approaches by 44.8%. Besides, *TraceRCA* is applied in a large commercial bank, and it helps operators localize root causes for real-world faults accurately and efficiently. We also share some lessons learned from our real-world deployment.**

I. Introduction

Microservice architecture is the latest trend in software service and is used by an increasing number of systems due to its faster delivery, better scalability, and greater autonomy [1]. A modern microservice system consists of dozens to thousands of microservices deployed on hundreds to thousands of servers [2], [3]. Although extensive efforts have been devoted to quality assurance, microservice systems are typically fragile due to their large scale and complexity [1]. Moreover, microservice system faults could cause enormous economic loss and damage user satisfaction. For example, the loss of one-hour downtime for `Amazon.com` on Prime Day in 2018 (its biggest sale event of the year) is up to $100 million [4]. Therefore, once a fault happens for microservice systems, the urgent demand is to localize and mitigate it as soon as possible.

Over the years, many approaches have been proposed in the field [5]–[12], including invocation-based and trace-based approaches. The invocation-based approaches assume that the adjacent microservices with abnormal invocations are more likely to be the root causes [5], [8]–[10], [12]. However, due to the complex dependencies and fault propagation among microservices, the anomaly invocations between adjacent microservices are not sufficient to reflect the locations of root causes (see Section V). The trace-based approaches overcome the above limitation by correlating all the microservices involved in a trace instead of just the adjacent ones. Here, all the invocations realizing the same user request form a *trace*. However, the existing trace-based approaches (*i.e.*, MicroScope [7], TraceAnomaly [13], and MEPFL [11]) still suffer from some practical issues. More specifically, MicroScope uses directed acyclic graphs to represent the dependency among microservices, but in practice, there often exist dependency cycles. For example, we confirmed many in the production system studied in Section V, and *Train-Ticket* studied in Section IV. TraceAnomaly focuses on detecting structural or latency anomalies of traces but ignores other metrics. Both TraceAnomaly and MicroScope localize root-cause microservices by assuming a fixed anomaly propagation pattern. MEPFL trains a supervised machine learning model to predict the root-cause microservices with a training corpus built by fault injection. Its effectiveness heavily depends on the high coverage of all fault types, achieving which is impractical. Therefore, it is still required for a more practical and better root-cause microservice localization approach.

In this paper, we propose a practical trace-based root-cause microservice localization approach called *TraceRCA*. The insight of *TraceRCA* is that a microservice with more abnormal traces and less normal traces passing through it is more likely to be the root-cause microservice. Similar insights are widely and successfully used in other domains such as spectrum-based program debugging [14]–[19] and multi-dimensional root cause localization [20], [21]. We also directly validated it in microservice systems (see Fig. 4). To apply the insight into root-cause microservice localization, we firstly detect abnormal traces. Here, *TraceRCA* infers trace's normality based on its member invocations' normality, which is detected by our designed unsupervised multi-metric anomaly detection method. Since not all metrics are related to the concerned fault and irrelevant anomalies could exist in any metrics, we adaptively select useful metrics by testing whether each metric's underlying distribution changes after the fault. The second stage in *TraceRCA* is to mine suspicious root-cause microservice sets satisfying the insight. Mining suspicious

†Junjie Chen is the corresponding author.

978-0-7381-3207-5/21/$31.00 ©2021 IEEE

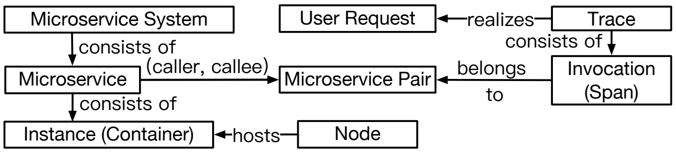


Fig. 1: Relationship of some basic concepts

microservice sets rather than *microservices* makes *TraceRCA* more practical because some faults only affect traces that pass through a specific set of microservices. Finally, *TraceRCA* calculates a suspicious score for each microservice in the mined suspicious microservice sets. Based on the number of traces containing incoming and outcoming abnormal invocations, we dynamically infer the anomaly propagation pattern for calculating the suspicious scores rather than assume a fixed one. Based on the suspicious scores, *TraceRCA* ranks all the microservices so that operators can mitigate faults earlier.

We conducted an extensive study to evaluate *TraceRCA* based on a popular open-source microservice benchmark (*Train-Ticket* [22]) and an Internet service provider's production microservice system. *Train-Ticket* is one of the most extensive open-source microservice benchmarks. We used 222 faults of 10 different categories in total. To our best knowledge, this is the most large-scale study in the field w.r.t. the number of faults, the number of fault types, and the scale of benchmarks. The experimental results show that *TraceRCA* ranks the root-cause microservices at top-1 in 83% of all faults and significantly outperforms the state-of-the-art unsupervised approach by 44.8%. We applied *TraceRCA* to a large service-oriented production system in a large commercial bank and share the lessons learned from our deployment in Section V.

The main contributions are summarized as follows:

- Based on a straightforward and simple insight, we design a novel unsupervised and lightweight root-cause microservice localization approach via trace analysis.
- We design an unsupervised multi-metric trace anomaly detection method, which adaptively selects useful features for each fault and conducts invocation anomaly detection and trace anomaly inference based on the selected features.
- We conduct the most large-scale experimental study based on 2 benchmarks with 222 faults in 10 different categories. The experimental results demonstrate the effectiveness and efficiency of *TraceRCA*. We share lessons learned from the deployment of *TraceRCA* in a large production system.

II. Basic Concepts

This section introduces some basic concepts, the relationship of which is shown in Fig. 1.

A **microservice system** is a system structured with microservice architecture. **Microservice architecture** is an architecture style that organizes a system as many lightweight, loosely-coupled, and independently-deployed services, called **microservices** [23]. For example, *Train-Ticket* [22] is organized as many microservices, such as *UI*, *seat*, *train*, *station*, *order*, and *price*. Each **microservice** has one or more **instances** hosted on **nodes** (physical or virtual machines). Each node can host many containers and microservices. Each microservice can also be hosted on different nodes.

When a microservice system realizes a **user request**, microservices invoke each other with some specific application programming interfaces (API). A microservice could contain tens to hundreds of APIs. An **invocation** (a.k.a. **span**) belongs to a specific microservice *caller*→*callee* pair (referred to as **microservice pair**). All invocations realizing the same user request form a **trace**. An industrial microservice system is commonly equipped with distributed tracing systems, which tracks the execution of a request across services, *i.e.*, traces [24]. For example, when the button *Buy* is clicked in *Train-Ticket*, a *user request* is sent to buy a ticket. First, microservice *UI* makes an API call to microservice *verification*. This API call is an *invocation* belonging to the *microservice pair UI*→*verification*. Each click on the button *Buy* will trigger an invocation belonging to *UI*→*verification*, *i.e.*, there may exist many invocations belonging to a microservice pair. After that, *UI* would call other microservices (*e.g.*, *payment*) and trigger many other invocations. All the invocations triggered by a click on the button *Buy*, which realize the same user request, form a *trace*.

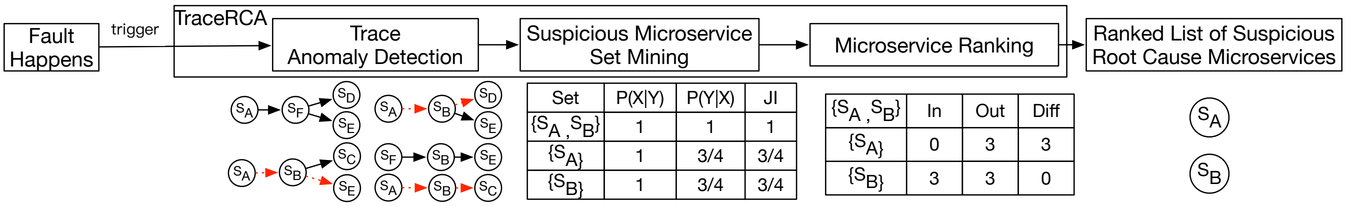
III. Approach

In this section, we present the detail of *TraceRCA*. By analyzing many real faults and summarizing the manual diagnosis process, we obtain the key insight of *TraceRCA*: a microservice with more abnormal traces and less normal traces passing through it is more likely to be the root cause. It is simple and effective, and it holds in various situations. In particular, our insight holds for partly faulty microservices (*e.g.*, only one container of the root cause occurs faults) and multi-root-cause faults, which are validated by our experiments (see Table III and Table IV). Though we focus on microservice systems, our insight can also successfully applied in similar architectures like service-oriented architectures (see Section V).

As shown in Fig. 2, *TraceRCA* contains three stages. When a fault happens, *TraceRCA* is triggered to localize the root-cause microservices. First, *TraceRCA* detects abnormal traces (Section III-A) with our unsupervised multi-metric anomaly detection method. Then, we propose using a unified metric to measure how much a microservice set satisfied the insight (Section III-B). *TraceRCA* utilizes frequent pattern mining techniques to reduce the search space. Finally, *TraceRCA* ranks all microservices based on the mined suspicious sets (Section III-C). In this way, operators can check microservices one by one according to our ranking so that the root cause can be identified more rapidly.

A. Trace Anomaly Detection

We first design a *multi-metric invocation* anomaly detection method to obtain the normality of each invocation and then infers trace's normality according to its member invocations' normality. Here, *TraceRCA* detects abnormal traces through inference from abnormal invocations rather than directly detect abnormal traces because traces are variable-length, leading to



Set	P(X|Y)	P(Y|X)	JI
$\{S_A, S_B\}$	1	1	1
$\{S_A\}$	1	3/4	3/4
$\{S_B\}$	1	3/4	3/4

$\{S_A, S_B\}$	In	Out	Diff
$\{S_A\}$	0	3	3
$\{S_B\}$	3	3	0

Fig. 2: Overview of *TraceRCA*. There are five traces, and the red dotted arcs represent abnormal invocations. The microservice set $\{S_A, S_B\}$ is mined as the most suspicious set. Then the microservices are ranked based on our suspicious scores.

low efficiency or low accuracy if transforming them to fixed-length vectors. In particular, our anomaly detection method is *unsupervised* to avoid the limitation of supervised approaches and make *TraceRCA* more practical.

1) Multi-Metric Invocation Anomaly Detection: In a microservice system, there are various metrics (a.k.a features). For example, in *Train-Ticket* (see Section IV-A), we use latency and HTTP status of each invocation, and CPU usage, memory usage, network receive/send throughput, and disk read/write throughput of each microservice as the features for trace anomaly detection. However, when a fault occurs, not all of the features are affected by the concerned fault. Due to wrong user inputs or just random fluctuation, some anomalies of irrelevant features could exist and become noise for anomaly detection, which harms the detection accuracy. Therefore, our method is designed with two steps: 1) adaptively selecting useful features for each fault; 2) detecting abnormal invocations based on the selected features. An overview of our invocation anomaly detection method is shown in Fig. 3.

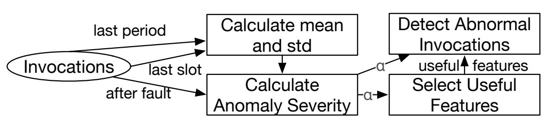


Fig. 3: Overview of our invocation anomaly detection method

Intuitively, if a feature is related to the concerned fault, there should be more abnormal invocations with respect to it after the fault occurs. Therefore, we determine whether a feature is useful by testing whether the distribution of normal and abnormal invocations with respect to it changes after the fault occurs. For this purpose, we need to determine the normality of each invocation with respect to each feature. Note that we only consider the historical invocations of the same microservice pair (see Fig. 1) to which this invocation belongs because the underlying distributions with respect to the same feature can vary vastly for different microservice pairs.

For a feature of a specific microservice pair (denoted as f), we use the mean and standard deviation of the historical invocations to model its normal state, which has been proved to be effective and is widely used in previous works [20], [25]–[29]. The mean (denoted as μ_f) defines the expected normal value of f, and the standard deviation (denoted as σ_f) is used to determine the probability that actual values deviate from the mean. With μ_f and σ_f, how much the feature value of an invocation (denoted as v_f) deviates from its normal state can be denoted as $\alpha=\frac{|v_f-\mu_f|}{\sigma_f}$, which is called *anomaly severity*. Hence the larger α is, the more likely it is abnormal.

We use the following techniques to ensure the calculation of μ_f and σ_f robust and efficient. First, the historical data used to calculate μ_f and σ_f is twofold: all the historical invocations of the same microservice pair 1) in the *last slot* and 2) in the same slot of the *last period* (*e.g.*, last day or week). Last-slot historical data capture the normal state of the feature in just a few minutes before the fault happens, and last-period historical data captures that in previous days or weeks. Small dip or spike could deviate from their last-period normal states insignificantly due to the variation among periods [30]. However, the deviation should be obvious compared with last-slot data; otherwise, they are not dips or spikes. If the abnormal metric changes gradually, the last-slot normal state can be biased since the metrics already change in the last slot. However, the last-period normal states, which are too far from the fault and thus not affected, should not be biased. Thus utilizing both last-slot and last-period invocations makes *TraceRCA* more robust and practical. In our implementation, the slot length is always the same as that of the current analyzing fault, and the period is chosen to be a day since daily periodicity is very common due to the periodicity of user behaviors. Second, to eliminate bias introduced by historical anomalies, we exclude invocations in all known previous faulty durations. Third, for the sake of efficiency, μ_f and σ_f are maintained in an online manner rather than calculated after a fault happens. More specifically, we update μ_f and σ_f periodically (typically per minute) with the latest coming data.

As mentioned before, if a feature is useful, there should be more abnormal invocations with respect to this feature, which should have large anomaly severities. Thus the average anomaly severity of all invocations with respect to this feature should be large. Therefore, the average anomaly severity of all invocations after the fault happens with respect to a useful feature, which is denoted as α_{after}, should be larger than that of the historical invocations, which is denoted as α_{before}. As a result, a feature is considered useful if $\alpha_{after}-\alpha_{before}>\delta_{fs}\cdot\alpha_{before}$, where δ_{fs} is a given threshold. The default value of δ_{fs} is 10%, and its impact is discussed in Section IV-D.

Finally, based on the selected useful features, we detect abnormal invocations. An invocation is abnormal if it is abnormal with respect to any useful feature. For this purpose, we need to determine the normality of all invocations with respect to each useful feature. Utilizing the anomaly severity mentioned above, *TraceRCA* considers an invocation abnormal with respect to a feature if the anomaly severity is greater than a given threshold, *i.e.*, $\alpha>\delta_{ad}$. The default value of δ_{ad} is 1, and its impact is discussed in Section IV-D.

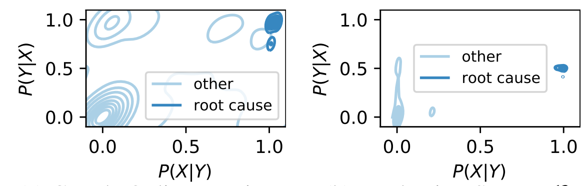
2) Trace anomaly inference: Based on the detected abnormal invocations, we infer the normality of traces. If at least one member invocation of a trace is determined abnormal in the last stage, this trace is determined abnormal. In this way, we can take as many abnormal traces into consideration for root cause localization as possible, which makes *TraceRCA* robust while keeping efficient enough (see Section IV-E).

B. Suspicious Root-Cause Microservice Set Mining

After trace anomaly detection, we mine suspicious *microservice sets* satisfying our insight rather than *microservices*. It is because in practice, sometimes only those traces that pass through a specific *microservice set* are affected by a fault. For example, a buggy API in microservice S_1 is triggered only by invocations from S_2, but many other microservices also invoke S_1. In such a case, the fractions of abnormal traces passing through S_1 or S_2 would be small, but that of abnormal traces passing through both would be large. We did not emulate such cases in Section IV due to the limitation of fault injection. Besides, we do not mine microservice sequences or subgraphs since *TraceRCA* focuses on localizing root cause *microservices*, and mining sequences or subgraphs is redundant for this purpose and harms the efficiency.

Specifically speaking, we propose two key metrics to evaluate how a microservice set satisfies the insight: 1) the support of a microservice set in abnormal traces (denoted as $P(X|Y)$ where X and Y denote the sets of those traces passing through all microservice in the set and all abnormal traces respectively, and $P(\cdot)$ denotes probability), which represents the percentage of those traces passing through all microservices in the set among all abnormal traces; 2) the confidence of a microservice set (denoted as $P(Y|X)$), which represents the percentage of abnormal traces among all traces passing through all microservices in the set. Note that "support" always refers to $P(X|Y)$ in this paper unless otherwise specified. Based on these two metrics, we propose that a microservice is a root-cause microservice if it has both high $P(X|Y)$ and high $P(Y|X)$. We validate the relationship between these two metrics and root-cause microservice sets based on two benchmarks: 1) 25 faults based on Google Online Boutique [31] (**not used** to evaluate *TraceRCA* in Section IV to avoid circle in proving) 2) 22 faults from a production system ($\mathcal{B}$ in Section IV-A). We estimated the distribution of $P(X|Y)$ and $P(Y|X)$ of root-cause and non-root-cause microservice sets by kernel density estimation. As shown in Fig. 4, the two metrics of root-cause microservice sets concentrated on the right-top corner, which means both metrics are larger than non-root-cause microservice sets. The good performance in Section IV-B also supports our insight.

The number of potential root-cause microservice sets is exponential to that of potential microservices, and thus evaluating the two metrics on all microservice sets is impractical. To reduce the search space, we first identify those microservice sets with high supports by an efficient frequent pattern mining algorithm, FP-growth [32]. FP-Growth skips the time-consuming candidate generation process entirely and uses a divide-and-conquer strategy plus a special data structure called



(a) Google Online Boutique (b) Production System $\mathcal{B}$

Fig. 4: The distribution of $P(X|Y)$ and $P(Y|X)$

FP-Tree. A frequent pattern is a pattern that appears in a dataset frequently [32]. In other words, every microservice set (*i.e.*, pattern) whose support ($P(X|Y)$) is greater than a given threshold (*i.e.*, appearing frequently), denoted as δ_{spt}, is frequent. For example, in Fig. 2, the abnormal traces are $\{S_A, S_B, S_D, S_E\}$, $\{S_A, S_B, S_C, S_D\}$ and $\{S_A, S_B, S_C, S_E\}$, and thus the most frequent microservice set is $\{S_A, S_B\}$, the support of which is 1. We set δ_{spt} to be 10%, which is relatively low to ensure localization accuracy. If any microservice set has a support less than 10%, it can hardly contain root causes because most abnormal traces are unrelated to it. The impact of δ_{spt} is discussed in Section IV-D.

Then, in the reduced search space, we mine microservice sets satisfying our insight by a unified metric, Jaccard Index (JI) [33]. We choose JI rather than any other SBFL (spectrum-based fault localization, see Section VI) technique because 1) JI combines support and confidence (as discussed below), which two perfectly match our insight 2) JI is one of the best SBFL techniques [14]. The intuition of JI is measuring the similarity between X (the traces containing the set)) and Y (the abnormal traces). JI is defined as the fraction of the intersection of X and Y to the union of them. A high JI means X and Y are almost overlapped. JI is a monotonically increasing function of the harmonic mean (denoted as H) of $P(X|Y)$ and $P(X|Y)$: $JI := \frac{P(X\cap Y)}{P(X\cup Y)} = \frac{H(P(X|Y),P(Y|X))}{2-H(P(X|Y),P(Y|X))}$. Therefore, we can mine those microservice sets satisfying our insight, *i.e.*, microservice sets with high $P(Y|X)$ and high $P(X|Y)$, by sorting with JI in descending order and taking top-k sets. We set k to 100 by default, which is large enough to include enough microservice sets for the next step while keeping *TraceRCA* fast enough. The impact of k is discussed in Section IV-D.

C. Microservice Ranking

Although we have suspicious microservice sets that are highly likely to contain root causes, operators need to investigate microservices one by one. Therefore, we calculate a suspicious score for each microservice to rank them. The suspicious score is the combination of each suspicious set's JI score and an in-set suspicious score for each suspicious set.

First, we give an in-set suspicious score for each microservice within each suspicious set containing it. It is calculated by the difference between the numbers of traces that contain incoming/outcoming abnormal invocations on the microservice among all those traces containing the suspicious set. For example, in Fig. 2, in the set $\{A, B\}$, the in-set score of B is $IS(B)$=0=$|3-3|$, and that of A is $IS(A)$=3=$|3-0|$. Considering a trace containing the suspicious set, if it contains both incoming and outcoming abnormal invocations of a microservice, then this microservice is highly likely to be just affected by other microservices, *i.e.*, it is not a causal anomaly, and the anomaly is just propagated through this microservice. Otherwise, if a trace contains only incoming/outcoming abnormal invocations on a microservice, the microservice is highly likely to be the root cause of this trace. The latter would contribute more to the difference than the former, and thus the difference indicates how likely a microservice is a causal anomaly within a suspicious set. Previous trace-based unsupervised approaches [7], [13] assume that the most upstream abnormal service (in call dependency or causal dependency graph) is most likely to be the root cause. For example, in Fig. 2, they would assume S_C, S_D, S_E are the root-cause microservices for the abnormal traces. However, in practice, the propagation of anomaly can be either upwards (*e.g.*, upstream services receive wrong parameters from a downstream service) or downwards (*e.g.*, downstream services wait too long for an upstream service). Our method overcomes the limitation by inferring the anomaly propagation pattern of each fault rather than assume a fixed one. Though our method is limited when the anomaly propagates both upstream and downstream, such cases are so rare in our scenarios (we mainly focus on infrastructure faults) that our method can meet operators' requirements.

For every suspicious set containing a microservice, we combine the set's JI score and the in-set score of the microservice by multiplication. Both multiplication and sum are simple and efficient combination methods, but these two scores are of different scales, and thus, sum is inappropriate. Based on the combination, the final **suspicious score of a microservice** is the maximal combination among all suspicious sets. Although more than one suspicious set can contain a root-cause microservice, a root-cause microservice only affects traces through one suspicious set, and thus we use the maximal combination among all suspicious sets. For example, in Fig. 2, both $\{S_A, S_B\}$ and $\{S_A\}$ contain the root cause S_A, but S_A only affects traces containing $\{S_A, S_B\}$ rather than those containing either $\{S_A, S_B\}$ or $\{S_A\}$.

IV. Experiment

A. Study Data

We use two microservice systems as subjects, *i.e.*, a widely-used open-source microservice benchmark system (*Train-Ticket*) and a large Internet service provider's production microservice system.

1) Open-source Microservice System: Train-Ticket [22] is one of the largest open-source microservice systems, which has been widely used in the existing work [11], [22], [35], [36]. It contains 41 microservices. We deployed it with Kubernetes [37] on 7 physical machines, each of which has a 12-core 2.4GHz CPU, 12 GB RAM. Each service is deployed with multiple instances. We continuously ran a workload generator, which simulated the real-world user access pattern observed from our cooperating bank (see Section V).

Following the existing work [7], [11], [13], we constructed faults for *Train-Ticket* by fault injection. We adopted three fault types following the existing work [7], [11], [13], *i.e.*, application bugs, CPU exhausted, and network jam. Furthermore, to evaluate performance in various situations, we considered faults on three different levels of components, *i.e.*, microservice, container, and API. In total, we have 5 fault injection strategies, as summarized in Table I. To inject a fault of a specific type into the target of a specific level, we first chose a container/microservice/API randomly and then applied the corresponding injection strategy on it. Regarding multi-root-cause faults, we selected multiple containers/microservices/APIs of a specific level and injected faults of the corresponding type simultaneously. Each fault lasted for about 5 minutes. In total, we constructed 200 faults of 5 categories, along with 242,259 traces, 22,675 (9.36%) of which are affected by the faults. In particular, to sufficiently investigate whether *TraceRCA* can work for both single-root-cause faults and multi-root-cause faults (although the latter is rare in practice [22], [38]), we constructed 11 faults that have more than one root-cause microservices among the 200 faults. For ease of presentation, we call the *Train-Ticket* dataset $\mathcal{A}$.

2) Production Microservice System: This system is a real-world microservice system with 13 microservices in a large ISP with more than 50 million users (part of its whole system). In particular, developers provided us 22 faults of 5 categories (i.e., CPU exhaustion, memory exhaustion, host network error, container network error, and database failures), along with 1,136,825 traces, 17,041 (1.50%) of which are affected by the faults. We call the dataset from this production system $\mathcal{B}$.

In total, there are 222 faults, 1,379,084 traces, 39,728 (2.88%) of which are affected by the faults. The datasets have been published anonymously to promote future research[1]. Since our compared approaches contain supervised approaches, for each fault, we randomly selected 20% traces as the training set and the remaining traces as the test set. Note that there are the same fault types in the training and test sets and the same ratio of abnormal traces to guarantee the effectiveness of supervised approaches. All the unsupervised approaches only use the test set for each fault.

B. Overall Performance on Root Cause Localization

We used the following metrics to evaluate its effectiveness following the existing work [7], [11]:

- *Top-k accuracy (A@k)* refers to the probability that the root causes are included in the top-k results. We chose k=1, 2, 3.
- *Mean average rank (MAR)* refers to the mean of the average of all root-cause microservice ranks in each fault.
- *Mean first rank (MFR)* refers to the mean of the first root-cause microservice rank in each fault.

We compared *TraceRCA* with the state-of-the-art (SOTA) *trace-based unsupervised* approach **MicroScope (MS)** [7] and **TraceAnomaly (TA)** [13], and the SOTA *trace-based supervised* approach **MEPFL** [11]. Besides, we compared

[1] https://github.com/NetManAIOps/TraceRCA

TABLE I: Summary of fault injection strategies on Train-Ticket

Fault Type	Description	Level	#Cases
Application Bug	If there is an application bug, the responses from the faulty microservices can be incorrect. We use *Istio* [34] to randomly substitute some responses with wrong responses.	Microservice	58
CPU Exhausted	Due to configuration errors or bursting requests, CPU can be exhausted, which causes long latency, low throughput or no response.,We use *stress-ng*, a popular stress test tool, to exhaust CPU.	Microservice	59
Network Delay	When there is network jam, packet transmission requires more time, so latency of responses becomes longer. We use *Istio* to randomly delay requests.	Microservice, Container, API	59, 10, 14

TABLE II: Overall effectiveness comparison of root cause localization

Subject	Algorithm	A@1	⇑A@1	A@2	⇑A@2	A@3	⇑A@3	MAR	⇑MAR	MFR	⇑MFR
$\mathcal{A}$	TraceRCA	0.82	—	0.91	—	0.95	—	1.54	—	1.50	—
	MicroScope	0.55	**51.02%**	0.61	**49.28%**	0.70	**34.88%**	3.70	**58.49%**	3.55	**57.92%**
	MEPFL (RF)	0.92	-10.72%	0.97	-5.48%	0.98	-2.79%	1.40	-9.72%	1.37	-9.35%
	Random Walk	0.51	**61.93%**	0.84	**8.42%**	0.90	**5.34%**	2.32	**33.66%**	2.26	**33.87%**
	RCSF	0.50	**63.91%**	0.83	**10.31%**	0.90	**5.79%**	1.83	**16.27%**	1.77	**15.48%**
	TraceAnomaly	0.45	**81.22%**	0.56	**63.23%**	0.61	**56.38%**	4.65	**66.90%**	4.58	**67.28%**
$\mathcal{B}$	TraceRCA	0.88	—	1	—	1	—	1.12	—	1.12	—
	MicroScope	0.82	**7.32%**	0.88	**13.64%**	0.88	**13.64%**	2.12	**47.17%**	2.12	**47.17%**
	MEPFL (RF)	0.94	-6.38%	1	0.00%	1	0.00%	1.06	-5.66%	1.06	-5.66%
	Random Walk	0.82	**7.32%**	0.94	**6.38%**	1	0.00%	1.24	**9.68%**	1.24	**9.68%**
	RCSF	0.47	**87.23%**	1	0.00%	1	0.00%	1.53	**26.80%**	1.53	**26.80%**
	TraceAnomaly	0.68	**20.27%**	0.68	**33.47%**	0.77	**22.94%**	2.32	**33.57%**	2.32	**35.29%**

* A@k means top-k accuracy, and ⇑ means the improvement rate (%) of *TraceRCA* over compared approaches.

TABLE III: Comparison of root cause localization on faults of different levels on $\mathcal{A}$

Subject	Algorithm	A@1	⇑A@1	A@2	⇑A@2	A@3	⇑A@3	MAR	⇑MAR	MFR	⇑MFR
Microservice	TraceRCA	0.83	—	0.93	—	0.97	—	1.39	—	1.34	—
	MicroScope	0.56	**46.67%**	0.62	**49.49%**	0.70	**37.33%**	3.64	**61.77%**	3.47	**61.26%**
	MEPFL (RF)	0.94	-12.00%	0.97	-4.82%	0.97	-0.64%	1.42	**1.98%**	1.38	**2.71%**
	Random Walk	0.51	**61.96%**	0.86	**7.25%**	0.94	**3.00%**	1.97	**29.37%**	1.91	**29.51%**
	RCSF	0.52	**60.00%**	0.86	**7.64%**	0.93	**3.69%**	1.68	**16.98%**	1.60	**16.02%**
	TraceAnomaly	0.49	**70.85%**	0.59	**58.14%**	0.63	**53.11%**	4.42	**68.54%**	4.34	**69.13%**
Container	TraceRCA	0.80	—	0.80	—	0.80	—	3.80	0%	3.80	—
	MicroScope	0.20	**300.00%**	0.40	**100.00%**	0.40	**100.00%**	7.20	**47.22%**	7.20	**47.22%**
	MEPFL (RF)	0.80	0.00%	0.80	0.00%	1.00	-20.00%	1.40	-171.43%	1.40	-171.43%
	Random Walk	0.40	**100.00%**	0.60	**33.33%**	0.60	**33.33%**	8.40	**54.76%**	8.40	**54.76%**
	RCSF	0.40	**100.00%**	0.60	**33.33%**	0.60	**33.33%**	3.60	-5.56%	3.60	-5.56%
	TraceAnomaly	0.20	**300.00%**	0.30	**166.67%**	0.30	**166.67%**	7.10	**46.48%**	7.10	**46.48%**
API	TraceRCA	0.83	—	0.83	—	0.83	—	1.75	—	1.75	—
	MicroScope	0.58	**42.86%**	0.67	**64.29%**	0.92	-9.09%	2.00	**12.50%**	2.00	**12.50%**
	MEPFL (RF)	0.83	0.00%	1.00	-16.67%	1.00	-16.67%	1.17	-50.00%	1.17	-50.00%
	Random Walk	0.58	**42.86%**	0.75	**11.11%**	0.64	**29.63%**	2.33	**25.00%**	2.33	**25.00%**
	RCSF	0.42	**100.00%**	0.58	**42.86%**	0.67	**25.00%**	2.58	**32.26%**	2.58	**32.26%**
	TraceAnomaly	0.21	**287.33%**	0.36	**132.40%**	0.50	**66.00%**	5.86	**70.12%**	5.86	**70.12%**

TABLE IV: Comparison of root cause localization on multi-root-cause faults of $\mathcal{A}$

Subject	Algorithm	A@1	⇑A@1	A@2	⇑A@2	A@3	⇑A@3	MAR	⇑MAR	MFR	⇑MFR
multi-root-cause cases on $\mathcal{A}$	TraceRCA	0.45	—	0.82	—	0.95	—	1.77	—	1.09	—
	MicroScope	0.27	**66.67%**	0.27	**200.00%**	0.41	**133.33%**	5.18	**65.79%**	2.73	**60.00%**
	MEPFL (RF)	0.45	0.00%	0.95	-14.29%	0.95	0.00%	1.64	-8.33%	1.09	0.00%
	Random Walk	0.41	**11.11%**	0.64	**28.57%**	0.82	**16.67%**	2.27	**22.00%**	1.36	**20.00%**
	RCSF	0.23	**100.00%**	0.50	**63.64%**	0.73	**31.25%**	2.82	**37.10%**	1.73	**36.84%**
	TraceAnomaly	0.50	**-10.00%**	0.73	**12.75%**	0.82	**16.11%**	2.50	**29.20%**	1.00	**-9.00%**

TraceRCA with two SOTA *invocation-based unsupervised* approaches, **RCSF** [6] and **Random Walk (RW)** [5], [8]–[10], [12]. As MS, TA, and MEPFL localize root causes trace-by-trace, the final results are voted by all abnormal traces. We ran MS based on our anomaly detection approach because they did not describe theirs. We adapted MEPFL on our collected features and selected **RF** (random forest) since RF performs well, runs fast compared with KNN (k nearest neighbor), and easy to train compared with MLP (multi-layer perceptron) [11]. We used RW with self and backward edges following the previous work [5], [8]–[10], [12] while using the anomaly severity of each microservice pair as weights because these previous approaches based on RW either only handles single metric ([5], [8], [10]) or relies on user participation ([9], [12]), which makes it unrealistic to use correlations with the alerting metric as weights. Original TraceAnomaly [13] utilizes only one metric (latency), which poorly performs since we inject faults in multiple metrics, and thus we applied TraceAnomaly on all metrics as *TraceRCA*.

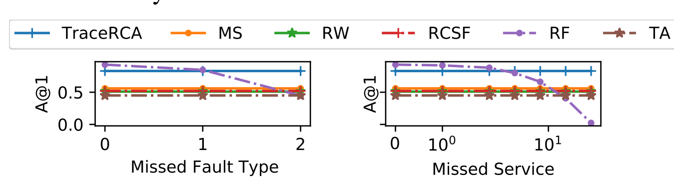


Fig. 5: Influence of the coverage of faults types and microservices in training data on $\mathcal{A}$ (shared y-axis).

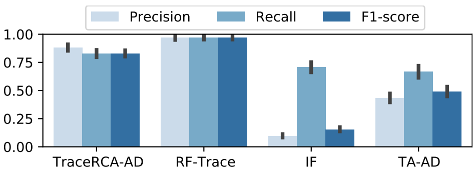


Fig. 6: Trace anomaly detection comparison

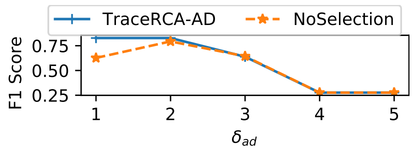


Fig. 7: Influence of feature selection and δ_{ad} on less significant anomalies

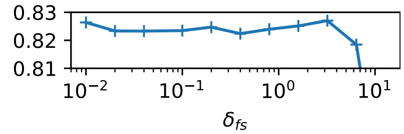


Fig. 8: Influence of δ_{fs} on $\mathcal{A}$

Table II compares the overall effectiveness. From this table, *TraceRCA* outperforms all the compared unsupervised approaches on both $\mathcal{A}$ and $\mathcal{B}$. The top-1 accuracy of *TraceRCA* outperforms the compared unsupervised approaches by 51.02%∼81.22% on $\mathcal{A}$. Most approaches perform well on $\mathcal{B}$ due to its smaller number of microservices and less complicated dependency among microservices. Nevertheless, the top-1 accuracy of *TraceRCA* still outperforms the compared unsupervised approaches by 7.32%∼87.23% on $\mathcal{B}$. On average across all the faults, the top-1 accuracy of *TraceRCA* achieves 83%, and the MAR achieves 1.50. *TraceRCA* outperforms the compared unsupervised approaches by 44.8%∼66.1% in top-1 accuracy, and 17.2%∼78.7% in MAR by further calculation. The unsupervised trace-based approaches, MicroScope and TraceAnomaly, assume a fixed pattern of anomalies propagation while *TraceRCA* dynamically infers it, and thus they are less robust. Besides, TraceAnomaly performs poorly on anomaly detection since it is initially designed for latency only (see Section IV-C), which harms its performance a lot.

Table III and Table IV present the results of root cause localization on different levels of faults and multi-root-cause faults in $\mathcal{A}$, respectively. From Table III, *TraceRCA* performs better than all unsupervised baselines in most cases across all the three levels. For example, the top-1 accuracy of *TraceRCA* achieves 0.80∼0.83 and outperforms other unsupervised approaches by 42.86%∼300% across all levels, and the MAR outperforms by 12.50%∼70.12%. From Table IV, *TraceRCA* largely outperforms all the compared unsupervised approaches w.r.t most metrics on multi-root-cause faults. For example, the top-2 accuracy of *TraceRCA* achieves 0.82 and outperforms other unsupervised approaches by 12.75%∼200%, and the MAR outperforms by 22.00%∼65.79%.

Conclusion 1 *TraceRCA significantly outperforms the SOTA unsupervised approaches in all the three studied levels of faults and both single-root-cause and multi-root-cause faults.*

Compared with the SOTA supervised approach (RF), *TraceRCA* is inferior but not too much. On average across all the faults on $\mathcal{A}$ and $\mathcal{B}$, *TraceRCA* underperforms RF by 10.3% in top-1 accuracy and 9.4% in MAR. The inferiority of *TraceRCA* is expected since supervised approaches have much more knowledge than unsupervised ones in general. Besides, by analyzing failed cases, we find that the intermediate step of anomaly detection may incur noise, which also affects the overall effectiveness (see Fig. 10). However, *supervised approaches heavily rely on high coverage of fault types and microservices in training data*. To demonstrate it, we constructed modified training sets by removing faults of a specific type or microservice from original datasets. It simulated the cases that in training data, there are missing fault types (*e.g.*, new fault types) or microservices (*e.g.*, microservices in which it is hard to inject faults). From Fig. 5, the effectiveness of *RF* indeed degrades quickly as the number of missed fault types and microservices increasing, while *unsupervised* approaches perform stably and outperform *RF* eventually. In practice, it is hard to guarantee high-quality training data, and thus our unsupervised approach *TraceRCA* is more practical and stable.

Conclusion 2 *TraceRCA performs almost as well as the SOTA supervised approach, but the latter relies on training data with high coverage of all fault types and microservices.*

C. Effectiveness of Our Trace Anomaly Detection Method

Following the existing work [11], we used three widely-used metrics, *i.e.*, precision, recall, and F1-score, to evaluate the effectiveness of the trace anomaly detection method in *TraceRCA* (denoted as *TraceRCA*-AD). We compared *TraceRCA*-AD with the SOTA *supervised* approach MEPFL (the same model as that for root cause localization can also be used for trace anomaly detection and achieves the SOTA performance [11]), and the SOTA *unsupervised trace-based* approach TraceAnomaly (the part of anomaly detection, denoted as TA-AD), and a widely-used *unsupervised invocation-based* method IF (isolation forest [39]). For MEPFL, we also used RF (denoted as RF-trace) as the representative due to the same reason in Section IV-B. For IF, we treated all feature values of an invocation as a multi-dimensional sample and applied IF on all the samples of each microservice pair to detect abnormal invocations, based on which we detected abnormal traces following Section III-A2.

As shown in Fig. 6, both *TraceRCA*-AD and *RF-Trace* achieve over 0.8 in F1-score. Note that our unsupervised method, *TraceRCA*-AD, is competitive with the supervised method, *RF-Trace*, while IF and TraceAnomaly performs poorly. It is because our datasets contain multiple features, and some are noisy and misleading. Thus the unsupervised approaches are hard to perform well without feature selection.

Although the supervised method, *i.e.*, *RF-Trace*, achieves the best effectiveness, it relies on high coverage of fault types and microservices in training data. In Fig. 9, we investigated the effectiveness of *RF-Trace* with different numbers of missed fault types and microservices in training data (following the experiment process of Fig. 5). The F1-score of *RF-Trace* degrades very quickly as the number of missed fault types or microservices increases, while the F1-score of unsupervised methods tends to be stable and stay competitive in this process. It is hard to guarantee high-quality training data in practice, and thus *TraceRCA*-AD is more practical and stable.

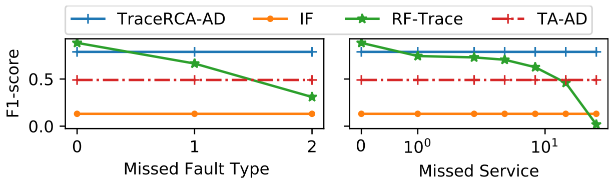


Fig. 9: Influence of coverage of fault types and microservices in training data on anomaly detection (shared y-axis).

As adaptive feature selection is an important step in *TraceRCA*-AD, we then investigated its contribution by comparing *TraceRCA*-AD and *TraceRCA*-AD without feature selection (denoted as $NoSelection$). As shown in Fig. 7, when δ_{ad} (the threshold of anomaly severities) is high, both have low F1-scores since they have many false negatives. But when δ_{ad} is low, feature selection helps reduce false positives, and thus, *TraceRCA*-AD has a higher F1-score. Therefore, adaptive feature selection is helpful to make *TraceRCA* more practical.

Conclusion 3 *The anomaly detection method in TraceRCA performs almost as well as the SOTA supervised method but is more practical. Also, adaptive feature selection in TraceRCA is helpful to ensure good performance.*

As mentioned in Section IV-B, the intermediate step of trace anomaly detection affects the effectiveness of *TraceRCA*. Here, we investigated its influence by randomly reversing the anomaly detection results with probability $noise\ ratio$, which simulates ineffective anomaly detection. The results are shown in Fig. 10. The supervised approach, *i.e.*, *RF-Trace*, does not rely on anomaly detection, and thus their results are not affected. When the noise ratio is less than 4%, the performance of *TraceRCA* does not degrade obviously. When the noise ratio goes larger than 8%, as well as all other unsupervised approaches, the performance of *TraceRCA* degrades but keeps outperforming others. That further demonstrates the contribution of our trace anomaly detection method. However, a more advanced anomaly detection approach is still required for further improving *TraceRCA*, which is not the target of this paper but can be regarded as our future work. Besides, *TraceRCA* is not able to detect structurally abnormal traces, which refers to traces that have invocations corresponding to unexpected microservice pairs while these invocations' monitoring metrics are expected. According to our interviews with several domain engineers, such structural anomalies are much less prevalent but may be related to severe issues like attacks. We keep this as part of our future work.

Conclusion 4 *The effectiveness of all unsupervised approaches, including TraceRCA, relies on the effectiveness of anomaly detection, but TraceRCA is more insensitive and consistently outperforms the SOTA unsupervised approaches.*

D. Impact of Main Parameters

We investigated the impact of main parameters (*i.e.*, $\delta_{spt}, k, \delta_{ad}$ and δ_{fs}). In Fig. 11 and Fig. 12, the top-1 accuracy and MAR of *TraceRCA* keep high while δ_{spt} (microservice sets with supports greater than it are frequent, see Section III-B) and k (the number of microservice sets taken for microservice ranking in Section III-B) change in a large range. As shown in Fig. 7, even if δ_{ad} (invocations with anomaly severities greater than it are abnormal, see Section III-A) is lower than the best threshold (*i.e.*, 2 in Fig. 7), with the help of our adaptive feature selection, the F1-score of *TraceRCA*-AD does not degrade. In Fig. 8, the F1-score of *TraceRCA*-AD keeps good performance even if δ_{fs} (features whose distributions change greater than it are useful, see Section III-A) varies.

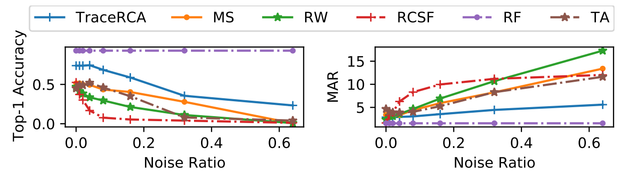


Fig. 10: Influence of ineffective anomaly detection

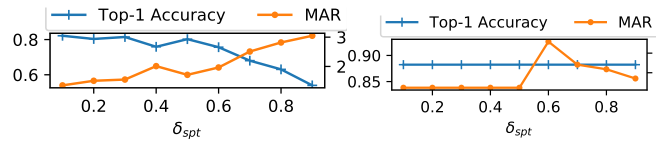


Fig. 11: Influence of δ_{spt} on $\mathcal{A}$ (left) and $\mathcal{B}$ (right)

Conclusion 5 *For each of the main parameters, TraceRCA is insensitive to it in a large range.*

E. Scalability

Here, we investigated the efficiency of *TraceRCA*. Our experiments were conducted on a server with 12 cores and 64G RAM. *TraceRCA* and all baselines are implemented with Python. Fig. 13 shows how many traces each approach can handle per second per core. *TraceRCA* is not the fastest one, but it is efficient enough. For a system with 100,000 traces per minute (a typical number from the large production system studied in Section V), *TraceRCA* takes only about 60 seconds to localize the root cause for a 5-minute fault. Although *TraceRCA* performs feature selection and trace anomaly detection on different microservice pairs separately, the overall time complexity is only related to the number of traces. In Fig. 14a, we present the relative running time (compared with the median running time without trace sampling) with different trace sampling proportions, by which we found that the running time is almost linear to the number of traces.

In large production systems, *trace sampling* is widely used to reduce system load since the number of traces can be

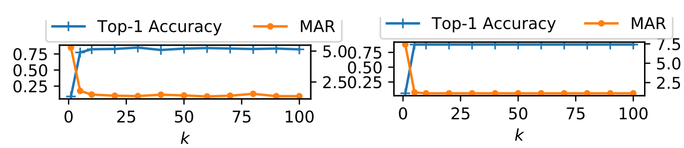
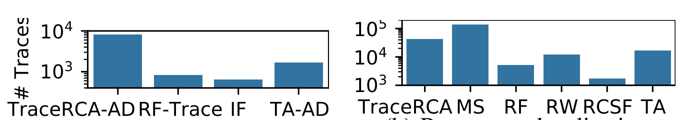


Fig. 12: Influence of k on $\mathcal{A}$ (left) and $\mathcal{B}$ (right)



(a) Trace anomaly detection (b) Root cause localization

Fig. 13: Comparison of efficiency

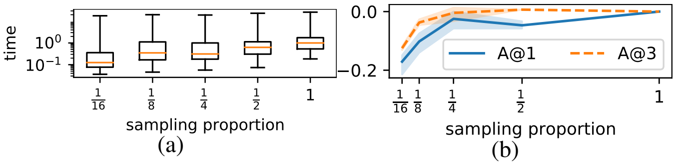


Fig. 14: Efficiency improvement and performance degradation of *TraceRCA* with trace sampling.

extremely large. *TraceRCA* is able to achieve relatively good performance with trace sampling. In Fig. 14b, we plot the performance degradation with different sampling proportions, where the band refers to the standard deviation (we repeated the experiment three times). With only $\frac{1}{16}$ of all traces, *TraceRCA* can achieve about 80∼90% of the best performance.

Conclusion 6 *TraceRCA has good scalability for practice.*

V. Deployment and Learned Lessons

TraceRCA has been successfully deployed in a production *service-oriented* system containing over 80 services of a large commercial bank. Since it is not a microservice system, the number of services seems small. However, it is large-scale w.r.t. the number of traces (over 100,000 traces per minute). We used the parameters of *TraceRCA* described in Section III. Based on the operators' feedback, *TraceRCA* helps them accurately and efficiently localize root-cause microservices in practice many times and saves much effort. In this section, we share some learned lessons from the deployment.

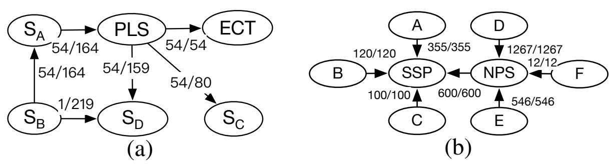


Fig. 15: Two real-world faults. Irrelevant services are omitted. The numbers on each microservice pair represent the numbers of abnormal/total traces passing through it.

First, traces are really necessary for root cause localization. Invocation indicates the relationship between only adjacent microservices, while trace provides the relationship among all microservices on the same trace. For example, in Fig. 15a, *PLS* either calls S_C or S_D directly, or calls *ECT* before calling them. A fault happened in *ECT*, and it affected the invocations of *PLS*→*ECT* and the following ones of *PLS*→S_C or *PLS*→S_D. Without traces, the relationship among *ECT*, S_C and S_D is uncertain, and thus we can hardly infer which downstream microservice of *PLS* is the root cause. But with traces, *TraceRCA* can easily observe that all abnormal traces intersect at *ECT*. In Fig. 15b, all microservice pairs are abnormal, and root-cause microservices are *SSP* and *NPS*. Without traces, the anomaly in *NPS* is highly likely to be considered as caused by *SSP*, since *NPS* relies on *SSP* and the invocations of *NPS*→*SSP* are all abnormal. With traces, *TraceRCA* can easily confirm that *SSP* and *NPS* are two independent root causes because most traces containing *NPS* do not contain *SSP*.

Second, abnormal metrics vary on different microservices even for one fault. For example, if a microservice encounters resource exhaustion, its response rate goes down. For its upstream microservices, the response latency will increase if they have to wait until timeout. The response latency may also keep steady or decrease if the faulty service refuses connection immediately, and in such cases, the success rate will decrease. Thus, a multi-metric anomaly detection approach is necessary.

Third, the interpretability of a root cause localization approach is important for operators to accept its results and take action. On the one hand, following a wrong localization result makes mitigation take a longer time. On the other hand, different microservices are usually in the charge of different operators or different departments, so the result of an approach affects the distribution of responsibility among operators or departments. In most cases, *TraceRCA*'s results can be understood well by displaying abnormal traces with invocations' normality and highlighting their intersection.

VI. Related Work

Recently, a great deal of effort has been devoted to localizing root-cause microservices. There are also many approaches on root cause localization for *service-oriented system*, *component based system*, and *cloud native system*. The underlying intuitions of them are similar to that of microservice systems, and thus, most such approaches can also be applied to root-cause microservice localization. Many approaches [5], [8]–[10], [12] are based on *random walk*. The intuition is that if microservices are visited in a sequence by picking up the next one with the highest probability of causal anomaly among all neighbors, the more visits of a microservice, the more it explains the anomalies of all microservices [5]. To address the problem of naive random walk, second-order random walk [8], self and backward edges [5], [9], [10], and combination of multi metrics [9], [12] are proposed. RCSF [6] mines *frequent sequential patterns* [32] as root causes directly among all call paths (thus, it is not trace-based) from abnormal services to the alerting service. Some approaches [11], [40] utilize historical faults or injected faults to build a *supervised* algorithm. The intuition is that similar faults have the same root causes. However, historical faults are inadequate, and fault injection in production systems is impractical due to 1) the cost of maintaining a benchmark with similar architecture and scale 2) and the limited types of fault injection. *Unsupervised trace-based approaches* [7], [13] utilize traces to improve fault diagnosis. They detect abnormal traces in an unsupervised manner and infer the root-cause microservice based on a mined causal graph or dependency graph.

Spectrum-based fault localization (SBFL) is popular and useful in program debugging [14]–[19]. A typical SBFL collects coverage information for program elements (*e.g.*, statements and methods) while running test cases and then employs a predefined scoring function to compute the suspicious scores for program elements. The intuition is that a program element covered by less passed tests and more failed tests is more likely to be the root cause, which is similar to our insight (see Section III). But SBFL uses user-defined test cases and need not anomaly detection, and SBFL directly mines suspicious program elements while *TraceRCA* mines suspicious microservice sets for robustness and ranks microservices at the further step.

VII. Conclusion

In this paper, we propose *TraceRCA*, a practical root-cause microservice localization approach via trace analysis, which is composed of trace anomaly detection, suspicious microservice set mining, and microservice ranking. The key insight of TraceRCA is that a microservice with more abnormal traces and less normal traces passing though it is more likely to be the root cause. Based on a widely-used open-source microservice benchmark and a production system, we conduct the largest experimental studies in the field, and the results show *TraceRCA* can localize the root cause accurately and efficiently. We also share learned lessons from our deployment in a large commercial bank.

VIII. Acknowledgement

This work was partially supported by the National Key R&D Program of China under Grant No.2019YFE0105500, the State Key Program of National Natural Science of China under Grant 62072264, and the Beijing National Research Center for Information Science and Technology (BNRist) key projects.

References

[1] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov, "Microservices: The journey so far and challenges ahead," *IEEE Software*, vol. 35, no. 3, pp. 24–35, 2018.

[2] "Why You Can't Talk About Microservices Without Mentioning Netflix," https://smartbear.com/blog/develop/why-you-cant-talk-about-microservices-without-ment/, 2019, [Online; accessed 14-Octorber-2019].

[3] M. Ranney, "What i wish i had known before scaling uber to 1000 services," in *GOTO Chicago 2016*, 2016.

[4] "Amazon's one hour of downtime on Prime Day may have cost it up to $100 million in lost sales," https://www.businessinsider.com/amazon-prime-day-website-issues-cost-it-millions-in-lost-sales-2018-7, 2019, [Online; accessed 04-November-2019].

[5] M. Kim, R. Sumbaly, and S. Shah, "Root cause detection in a service-oriented architecture," *ACM SIGMETRICS Performance Evaluation Review*, vol. 41, no. 1, pp. 93–104, 2013.

[6] K. Wang, C. Fung, C. Ding, P. Pei, S. Huang, Z. Luan, and D. Qian, "A methodology for root-cause analysis in component based systems," in *IWQoS 2015*. IEEE, 2015, pp. 243–248.

[7] J. Lin, P. Chen, and Z. Zheng, "Microscope: Pinpoint performance issues with causal graphs in microservice environments," in *ICSOC 2018*, 2018.

[8] P. Wang, J. Xu, M. Ma, W. Lin, D. Pan, Y. Wang, and P. Chen, "Cloudranger: root cause identification for cloud native systems," in *CCGRID 2018*. IEEE Press, 2018, pp. 492–502.

[9] M. Ma, W. Lin, D. Pan, and P. Wang, "Ms-rank: Multi-metric and self-adaptive root cause diagnosis for microservice applications," in *ICWS 2019*. IEEE, 2019, pp. 60–67.

[10] J. Weng, J. H. Wang, J. Yang, and Y. Yang, "Root cause analysis of anomalies of multitier services in public clouds," *IEEE/ACM Transactions on Networking*, vol. 26, no. 4, pp. 1646–1659, 2018.

[11] 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 *ESEC/FSE 2019*.

[13] 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 *ISSRE 2020*, pp. 48–58.

[12] M. Ma, J. Xu, Y. Wang, P. Chen, Z. Zhang, and P. Wang, "AutoMAP: Diagnose Your Microservice-based Web Applications Automatically," in *WWW 2020*.

[14] M. Zhang, Y. Li, X. Li, L. Chen, Y. Zhang, L. Zhang, and S. Khurshid, "An Empirical Study of Boosting Spectrum-based Fault Localization via PageRank," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[15] J. Jiang, R. Wang, Y. Xiong, X. Chen, and L. Zhang, "Combining Spectrum-Based Fault Localization and Statistical Debugging: An Empirical Study," in *ASE 2019*.

[16] J. Xuan, M. Martinez, F. DeMarco, M. Clément, S. L. Marcote, T. Durieux, D. L. Berre, and M. Monperrus, "Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs," *TSE*, vol. 43, no. 1, pp. 34–55, Jan. 2017.

[17] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," in *ICSE 2017*.

[18] Q. Xin and S. P. Reiss, "Leveraging syntax-related code for automated program repair," in *ASE 2017*.

[19] X. B. D. Le, D. Lo, and C. L. Goues, "History Driven Program Repair," in *SANER 2016*.

[20] F. Ahmed, J. Erman, Z. Ge, A. X. Liu, J. Wang, and H. Yan, "Detecting and localizing end-to-end performance degradation for cellular data services based on tcp loss ratio and round trip time," *TON*, vol. 25, no. 6, pp. 3709–3722, 2017.

[21] F. Lin, K. Muzumdar, N. P. Laptev, M.-V. Curelea, S. Lee, and S. Sankar, "Fast Dimensional Analysis for Root Cause Investigation in a Large-Scale Service Environment," *SIGMETRICS 2020*.

[22] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, "Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study," *TSE*, 2018.

[23] S. Newman, *Building microservices: designing fine-grained systems*. " O'Reilly Media, Inc.", 2015.

[24] X. Guo, X. Peng, H. Wang, W. Li, H. Jiang, D. Ding, T. Xie, and L. Su, "Graph-based trace analysis for microservice architecture understanding and problem diagnosis," in *ESEC/FSE 2020*.

[25] S.-B. Lee, D. Pei, M. Hajiaghayi, I. Pefkianakis, S. Lu, H. Yan, Z. Ge, J. Yates, and M. Kosseifi, "Threshold compression for 3G scalable monitoring," in *INFOCOM 2012*, pp. 1350–1358.

[26] J. Xu, Y. Wang, P. Chen, and P. Wang, "Lightweight and Adaptive Service API Performance Monitoring in Highly Dynamic Cloud Environment," in *SCC 2017*, pp. 35–43.

[27] H. Jayathilaka, C. Krintz, and R. Wolski, "Performance Monitoring and Root Cause Analysis for Cloud-hosted Web Applications," in *WWW 2017*.

[28] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang, "Towards highly reliable enterprise network services via inference of multi-level dependencies," *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 4, pp. 13–24, Aug. 2007.

[29] Y. Chen, R. Mahajan, B. Sridharan, and Z.-L. Zhang, "A provider-side view of web search response time," vol. 43, no. 4, pp. 243–254.

[30] H. Xu, W. Chen, N. Zhao, Z. Li, J. Bu, Z. Li, Y. Liu, Y. Zhao, D. Pei, Y. Feng *et al.*, "Unsupervised anomaly detection via variational auto-encoder for seasonal kpis in web applications," in *WWW 2018*.

[31] "GoogleCloudPlatform/microservices-demo," Google Cloud Platform, Feb. 2021. [Online]. Available: https://github.com/GoogleCloudPlatform/microservices-demo

[32] J. Han, J. Pei, and M. Kamber, *Data mining: concepts and techniques*. Elsevier, 2011.

[33] L. Geng and H. J. Hamilton, "Interestingness measures for data mining: A survey," *ACM Computing Surveys*, vol. 38, no. 3, pp. 9–es, Sep. 2006.

[34] "Istio," https://istio.io/, 2019, [Online; accessed 03-Octorber-2019].

[35] X. Hou, J. Liu, C. Li, and M. Guo, "Unleashing the scalability potential of power-constrained data center in the microservice era," in *ICPP 2019*.

[36] X. Zhou, X. Peng, T. Xie, J. Sun, W. Li, C. Ji, and D. Ding, "Delta debugging microservice systems," in *ASE 2018*.

[37] "Kubernetes," https://kubernetes.io/, 2019, [Online; accessed 03-Octorber-2019].

[38] Q. Lin, J.-G. Lou, H. Zhang, and D. Zhang, "idice: problem identification for emerging issues," in *ICSE 2016*. IEEE, 2016, pp. 214–224.

[39] F. T. Liu, K. M. Ting, and Z.-H. Zhou, "Isolation forest," in *ICDM 2008*. IEEE, 2008, pp. 413–422.

[40] C. Pham, L. Wang, B. C. Tak, S. Baset, C. Tang, Z. Kalbarczyk, and R. K. Iyer, "Failure diagnosis for distributed systems using targeted fault injection," *TPDS*, vol. 28, no. 2, pp. 503–516, 2016.