Unsupervised Anomaly Detection for Traces through Planar Normalizing Flows

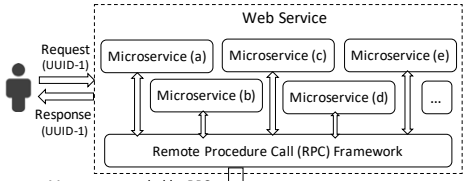
***Abstract*—**
Troubleshooting a microservice-based large software system is very challenging due to the large number of underlying microservices and the complex call relationships between them. This paper aims to detect any anomalies when they first appear in the microservice invocation traces (recorded by RPC tracing framework such as Google Dapper), to avoid eventually impacting the overall web service. Our core idea is to automatically learn the overall normal patterns of the traces during periodic offline training. Then in online anomaly detection, a new trace with a small likelihood (computed based on the learned normal pattern) is considered abnormal. With our novel trace representation and design of deep bayesian networks with planar normalizing flows, our unsupervised anomaly detection system, called *TraceAnomaly*, can accurately and robustly detect both trace response time and structure anomalies in a unified fashion. Our experiments using 46 different web services show that *TraceAnomaly* can achieve an average F-scores of $0.79 \sim 0.99$ with a fixed threshold; the detection precision of in real deployment during a week is at least 0.92.

I. Introduction

Recently, microservice architecture [1] has become more and more popular for large-scale software systems in web-based services. This architecture decouples a web service into multiple microservices (*e.g.*, 10∼100 microservices in company S studied in this paper), each with well-defined APIs. The top of Fig. 1 shows such a service. Each microservice can be individually upgraded, which enables more agile software development and deployment. The more frequent failures resulted from the frequent software updates, coupled with the complex microservice call relationships between the large number of microservices underneath a service, make the troubleshooting microservice-based service a daunting task.

This paper aims to take one important step in troubleshooting microservice-based web services: detect any anomalies when they first appear in the microservice invocation traces (recorded by Remote Procedure Call tracing framework such as Google Dapper [2]). There are two major types of anomalies: too long or too short a microservice response time, and missing/unexpected call messages, which can indicate software bugs, software/hardware failures, *etc.* Thus one would like to detect any trace anomalies as soon as they appear, so that they can be fixed before they significantly hurt the overall service.

Fig. 1 shows how the trace for one specific user request is collected by RPC tracing framework such as Google Dapper [2]. When a user request is received, a web service will generate a UUID (Universally Unique Identifier) that uniquely identifies all messages for this user request (*e.g.*, UUID-1 in Fig. 1). When microservice m calls microservice n, m sends a call message to n (denoted as *call(m→n)*). After the processing at n is completed, n sends the response message to m (denoted by *response(n→m)*). Both messages are routed between m and n via the RPC framework, thus the tracing mechanism can naturally record these messages with timestamps and UUIDs. All messages with the same UUIDs constitute one **microservice invocation trace** (**trace** for short hereinafter). The table in Fig. 1 shows an example trace with *UUID-1*.



Message order	UUID	Sending time of (m→n) at m (msec)	Receiving time of (m→n) at n (msec)	Message (m→n)
①	UUID-1	-	1519747202138	call(start→a)
②	UUID-1	1519747202144	**1519747202146**	call(a→b)
③	UUID-1	1519747202149	1519747202151	call(b→c)
④	UUID-1	1519747202149	1519747202150	call(b→d)
⑤	UUID-1	1519747202155	1519747202156	*response*(c→b)
⑥	UUID-1	1519747202159	1519747202160	call(d→e)
⑦	UUID-1	1519747202188	1519747202190	*response*(e→d)
⑧	UUID-1	1519747202194	1519747202196	*response*(d→b)
⑨	UUID-1	1519747202253	1519747202256	call(b→e)
⑩	UUID-1	1519747202323	1519747202324	*response*(e→b)
⑪	UUID-1	**1519747202355**	1519747202356	*response*(b→a)
⑫	UUID-1	1519747202360	-	*response* (a→end)

Fig. 1: An example trace with UUID-1. The first column is the message order based on the timestamps. The sending time of first call message (message ①) and the receiving time of last response message (message ⑫) are not collected, which are denoted by "-". The input parameters of user request are not available to this study due to privacy concerns.

Our goal of **trace anomaly detection** is the following: for a given trace, detect any anomalous message (unexpected microservice response time or an unexpected message) or the lack of an expected message. The two major challenges and our core tackling ideas are the following.

Challenge 1 and our core idea: detecting all types of trace anomalies for a service in a unified fashion. We have this challenge because: 1) response time and message relationship can change significantly without being anomalous. This mandates that response time modeling must consider the invocation path from the service entrance to a microservice. However, the number of individual paths is too large (*e.g.*, hundreds for one service, and there are hundreds of services in company S). This scale makes it infeasible to directly apply existing time series anomaly detection on path *individually* because these algorithms either require operators to manually

pick algorithm and tune parameters for each path [3], [4], [5], [6], [7], [8], [9], or need a dedicated learnt model for each call path [10], [11]. 2) for the same service, two traces might share the same structure (*i.e.*, nodes and edges. Fig. 2 is the structure of the trace in Fig. 1), and only the difference of response time can help distinguish between normal and abnormal traces (see an detailed example later in Fig. 5).

The above difficulties call for a unified detection approach for both response time and structure anomalies *at the trace level*, as opposed to *individual node, edge, or path level*. Our core idea is to treat each trace (*e.g.*, those in the table in Fig. 1) for a service as a training sample, and use machine learning to capture the overall patterns of the traces. To this end, we encode all the essential information (such as the response time information and call relationships) of a trace into a vector (called **trace vector** or **TV** hereinafter), a necessity required by most mature deep learning algorithms. The encoded information should not only represent the traces' whole patterns, but also can be easily interpreted by the operators. This is why we choose to handcraft the trace vector design with physical significance, as opposed to using network/graph representation learning (embedding) to automatically learn the vector which are not easy to interpret.

Challenge 2 and our core idea: designing an accurate, robust, unsupervised learning architecture that captures the characteristics of complex patterns, with reasonable training overhead. The complexity of the trace behavior are mainly due to the following three reasons. First, the call relationships are complex. All call relationships of a specific trace can form a graph (*e.g.*, Fig. 2 is the graph for the trace in Fig. 1). For a given service, the number of unique graphs resulted from various traces can be large (*e.g.*, 682 in a service of S). Second, the response time distribution is related to the microservice x and its invocation path from the entrance microservice (or *call path* hereinafter, see Fig. 3). For example, in Fig. 1 microservice e is invoked twice, with different response times (see Fig. 4) . Third, if there is an edge $m \rightarrow n$, then m's response time is impacted by n's since it has to wait for n's response. Depending on m's internal processing result, it might pass a different parameter to n, which results in a different response time of n.

With above complexity, we do not consider graph/network anomaly detection algorithms [12], [13], [14], [15], [16], [17] because they cannot work with our trace vector or deal with its complex behavior. Furthermore, because anomaly labels are infeasible to obtain in such a complex context with vast amount of data, we have to use unsupervised algorithm. Above requirements overall call for a high-capacity model, such as KDE, VAE, *etc.* Other models such as GMM and Diagnoal GMM can also be considered but their training time at high dimension might be too long, due to the large number of services and vast amount of trace data. The model should also be robust in that its performance is not sensitive to hyper-parameters. Our core idea to tackle this second challenge is to apply planar normalizing flows [18], which can use nonlinear mappings to increase the complexity of its latent variables, allowing the model to capture the complex patterns in a robust, accurate and unsupervised manner with reasonable training overhead.

In summary, for each service, our proposed system *TraceAnomaly* processes each trace (*e.g.*, the trace in Fig. 1) as a whole, constructs a trace vector that encodes both structure and response time (*e.g.*, the information in the table in Fig. 4), then learns the overall normal patterns during periodic offline training. After that, in online anomaly detection, for each new trace, a likelihood score is computed based on the learnt model, and a trace with a small likelihood is considered abnormal.

The contributions of this paper is summarized as follows:

- We propose a novel and the first method to construct feature vectors for traces, called Trace Vector, which efficiently encodes both the response time information and the call relationship information of traces.
- We propose *TraceAnomaly*, the first (to the best of our knowledge) unsupervised deep learning algorithm which can learn the complex trace patterns and accurately detect trace anomalies, with our novel trace representation and our design of deep Bayesian networks with planar normalizing flows.
- Our experiments using 46 different web services show that *TraceAnomaly* can achieve an average F-scores of $0.79 \sim 0.99$ with a fixed threshold; the detection precision of in 6 real service is at least 0.92 during real deployment for a week.

The rest of the paper is organized as follows. §II presents the concepts and overall design of *TraceAnomaly*. §III presents the design of our deep Bayesian network with Planar Normalizing Flows. §IV evaluates *TraceAnomaly* using historical traces and injected anomalies. §V shows the operational experience from real deployment. §VI reviews related work, and §VII concludes the paper.

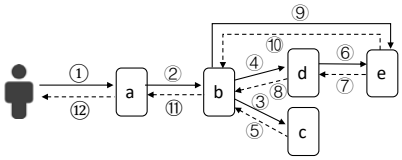


Fig. 2: Structure of the trace in Fig. 1, where circled numbers are message orders. The response messages are ignored to simplify the representation hereinafter.

II. TV CONSTRUCTION AND *TraceAnomaly* OVERVIEW

This section presents how to extract call paths and response times from the traces needed for trace vector TV, and how to construct TV. Then the overall *TraceAnomaly* design is presented.

A. Extracting call paths and response times from a trace

To address Challenge 1 summarized in § I, our design goal for trace vector is that both the response time pattern and structure pattern of traces for a given service can be learned from the trace vector data.

Sending time of (m→n) at m (msec)	Message (m→n)
-	call(start → a)
1519747202144	call(a→b)
1519747202149	call(b→**c**)
1519747202149	call(b→**d**)
1519747202159	call(d→e)
1519747202253	call(b→e)

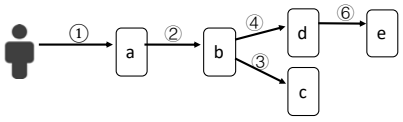
Call messages of a trace sorted by the sending time.

Microservice x	Call path of microservice x (x, (call path))
a	(a, (start→a))
b	(b, (start→a, a→b))
c	(c, (start→a, a→b, b→c))
d	(d, (start→a, a→b, b→c, b→d))
e	(e, (start→a, a→b, b→c, b→d, d→e))
e	(e, (start→a, a→b, b→c, b→d, d→e, b→e))

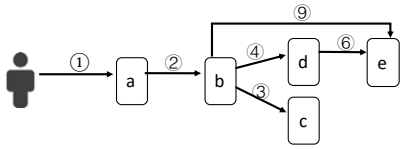
Extracted call paths

Fig. 3: Call path extraction

Microservice x	Call path of microservice x (x, call path)	Response time of (x, call path) (msec)
a	(a, (start→a))	222
b	(b, (start→a, a→b))	**209**
c	(c, (start→a, a→b, b→c))	4
d	(d, (start→a, a→b, b→c, b→d))	44
e	**(e, (start→a, a→b, b→c, b→d, d→e))**	**28**
e	**(e, (start→a, a→b, b→c, b→d, d→e, b→e))**	**67**



Call path (e, (start→a, a→b, b→c, b→d, d→e))

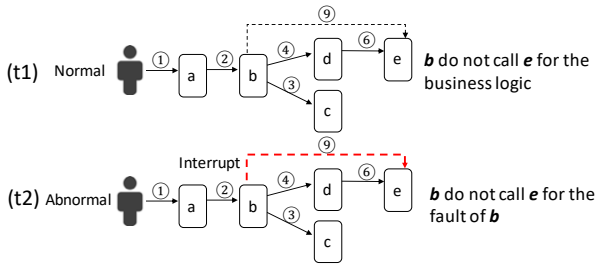


Call path (e, (start→a, a→b, b→c, b→d, d→e, b→e))

Fig. 4: Response time of microservices' call paths in Fig. 1. Call path of microservice x is the call message sequence sorted by sending time before x is called.

We first elaborate the challenges faced by above design goal. One service's trace response time and message relationship can change even under normal executions. 1) Two traces with different entrance input parameters (unavailable to this study due to privacy concerns) may result in different traces with different call messages and message response times. This is due to *code logic* difference. 2) Two traces with the same entrance input parameters might have different message latency or different call message due to the run time conditions. This is due to *runtime variation*. 3) The same microservice can be called multiple times and have different response times in the same trace, *e.g.*, in Fig. 1 microservice x is invoked twice, with different response times (⑥ ⑦ vs. that with ⑩ and ⑪). This is also due to code logic.

Roughly speaking, the input parameters of microservice x heavily impact x's internal execution logic and its call to other microservices, thus its response time (which is also impacted by runtime conditions and its downtream microservices' response times). Thus, ideally, if we can obtain the input parameters of x, we can assume the response time of x with given input parameters follow some patterns. Unfortunately, those input parameters are not recorded by the RPC tracing framework, thus unavailable to *TraceAnomaly*. Instead, our idea is to use *call path* of x to approximate x's input parameters.



Microservice x	Call path of microservice x (x, (call path))	(t1) Response time of (x, call path) (msec)	(t2) Response time of (x, call path) (msec)
a	(a, (start→a))	222	**702**
b	(b, (start→a, a→b))	209	**609**
c	(c, (start→a, a→b, b→c))	4	**14**
d	(d, (start→a, a→b, b→c, b→d))	44	**204**
e	(e, (start→a, a→b, b→c, b→d, d→e))	28	**128**

Fig. 5: Trace t1 and trace t2 have the same structure, but t1 is normal while t2 is abnormal which costs more time than expected. The interrupt failure of microservice b caused the anomalies in the response times of trace t2.

*A **call path** of microservice x in a trace, denoted as (x, call path), is the sequence of call messages (sorted by sending time) before x is called.* Fig. 3 shows the process of how to extract the call path for each microservice in a trace. First, all call messages of a trace are sorted by their sending time. If some call messages have the same sending time, then these call messages could be sorted by the IDs of callee n (to ensure the uniqueness of the order). As shown in the left half of Fig. 3, the sending time of call($b \rightarrow c$) and call(b→d) are the same, then the two call messages are sorted by the alphabetical order (c is ranked ahead of d). Second, for each call message (m→n), extract the call message sequence from the sorted call messages. The call message sequence starts from the first call message and end at the current call message (m→n), which represents the call path of the callee microservice n. Fig. 3 shows all the extracted call paths of the trace in Fig. 1.

By encoding all the call messages in the same trace into a call path that happened before x is called, we hope that the call path has captured "what has happened" in the trace which potentially had impacted on x's response time. We thus use (x, call path) as the *dimension* in our trace vector TV, and the value of this dimension is the corresponding response time. The response time rt of microservice (x, call path) can be computed by the receiving time of call message rct and the sending time of corresponding response message srt: $rt = srt - rct$. For example, in Fig. 4, the response time of microservice b (*209ms*) is computed by the receiving time of message ② ((*1519747202146* in Fig. 1) and the sending time of message ⑪ (*1519747202355* in Fig. 1). Note that the last two rows of the table Fig. 4 show that the different response times of microservice e's two different call paths.

Above encoding approach enables *TraceAnomaly* to have some patterns to learn from each dimension of TV, which, according to above design and analysis, should have some "normal patterns". But how do we detect structure anomalies (*e.g.*, missing, unexpected call messages)? Ideally, for a trace (*e.g.*, Fig. 1), we would want to capture its entire trace structure

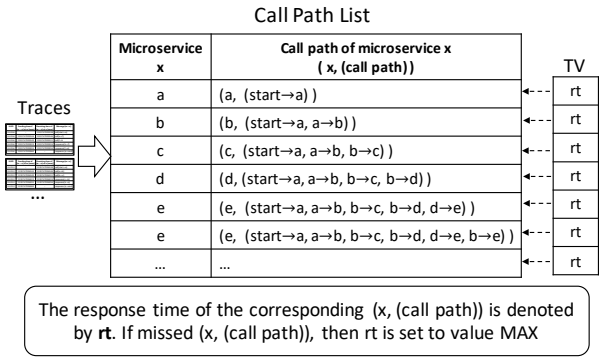


Fig. 6: Process of TV construction.

(*e.g.*, Fig. 2), including its message sequence (*i.e.*, the circled number on the edges). Fortunately, the call path information encoded in the TV as the dimension ID already includes all the call messages (the middle column of the table in Fig. 4, which include not only the "topology" (node and edge) of Fig. 2, but also the order how this topology is grown (the circled number on the edges). In summary, the trace structure is naturally encoded in the dimension ID of the trace vector TV, which can be used by machine learning to learn normal patterns and detect anomalies, for which we do not need additional design.

Note that, for the same service, two traces might share the same "structure" (*i.e.*, nodes and edges), and only the difference of response time can help distinguish between normal and abnormal traces. As shown in Fig. 5, the structures of trace $t1$ and trace $t2$ are the same, while $t1$ is normal because its input parameters (unavailable to us due to privacy concerns) to this service demands such as a structure, and $t2$ is abnormal because its input parameter demands the edge ⑨ which in the trace does not happen due to some fault at microservice b during runtime. Thus, previous log anomaly detection approaches [19], [20], [21], [22], [23], [21] focused on inferring the (at their study time) unavailable UUID and detect structure anomalies, thus cannot distinguish between $t1$ and $t2$. Luckily, most of the time the response time anomaly coincides with the structure anomaly, as shown in the last column of the table in Fig. 5, and TV encoding can naturally deal with this by declaring this trace as an anomaly. This confirms that our design choice of *developing a unified trace anomaly detection for both response time and structure* is a promising one.

B. Trace Vector Construction

We now introduce the details of encoding traces as trace vectors (TVs). We choose to handcraft the vector design based on our understanding of trace (see § II-A), as opposed to applying representation learning, so that the resulting vector has physical significance.

Before the training time, all call paths of the traces for a service are extracted via the call path extraction method shown in Fig. 3. The set of all unique call paths for a service form its *call path list*. Given the large amount of training data of traces, we assume that all normal call paths (or call relationships) are covered by the call path list that we constructed during training time. Thus, before each training, we have a fixed TV dimension, each of which corresponds to a unique callpath from the call path list.

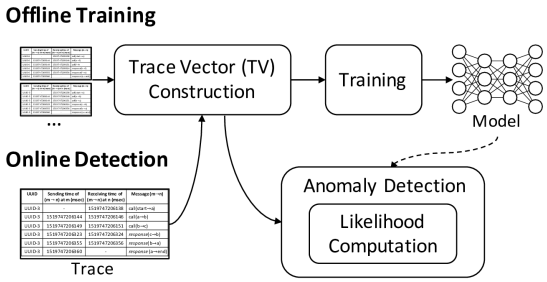


Fig. 7: Architecture of TraceAnomaly

During training time of a service, for each trace, we extract each call path and its corresponding response time rt, using the methods presented in § II-A , and then record them into a TV, illustrated in Fig. 6. If a specific trace does not have *(x, call path)*, then the corresponding dimensions are set to value *MAX*. The TV construction for a trace during online detection is the same as above. We assume that a previously unseen call path during detection is an anomaly.

C. TraceAnomaly Overall Architecture

Fig. 7 shows the architecture of *TraceAnomaly*. Offline training is conducted periodically (*e.g.*, every week or day). All traces of a web service are used to train a model offline. Each trace is encoded as a TV using method shown in § II-B. All traces are then fed into an unsupervised learning algorithm (see § III) to learn the normal patterns from the TVs, and generate a model. During online detection, each new trace is encoded into a TV, and the trained model outputs a likelihood for each TV. If the likelihood of a TV is lower than a threshold, *TraceAnomaly* declares it as an anomaly. If a trace contains a previously unseen path, it is also considered as an anomaly. § III presents the design of training and detection, and explain how Challenge 2 mentioned in § I is addressed.

Anomalies in the training set. We are trying to learn the normal patterns from the training dataset, but there can be anomalies in the training dataset. However, our observation and assumption is that there are very few anomalies in the traces. We train our deep Bayesian network by stochastic gradient descent (SGD), which can naturally tolerate rare anomalies. It updates the model parameters according to mini-batch samples of the input data, thus only captures the most significant patterns of the data distribution. We believe such rare anomalies would not affect the final performance of our model too much because of the properties of SGD. The only problem is that such anomalies might cause our training process to be unstable, but we successfully developed techniques to address this issue. See § III-C for more details.

III. Algorithm Design

In this section, we first introduce some algorithms used in this paper. We then present the architecture of our model. Next, we introduce the details of training model. Finally, we show how to detect anomalies by the trained model.

A. Background of Bayesian Network, Variational Inference, SGVB, Importance Sampling and Planar Normalizing Flows

Bayesian networks are probabilistic models, which can be represented by *directed acyclic graphs* (*e.g.*, our model Fig. 8a). A *node* on a Bayesian network represents a random variable, while a *link* between a pair of nodes indicates there is potentially a conditional dependence, ready to be learnt. Deep Bayesian networks, a special type of Bayesian networks, model such links by neural networks. An example of the deep Bayesian networks is the *variational auto-encoder* (VAE) [24].

In complex Bayesian networks, especially for deep Bayesian networks, it is often computational intractable to compute some posterior distributions (typically along the opposite directions of links, *e.g.*, $p_\theta(\mathbf{z}|\mathbf{x})$ in Fig. 8a). These posteriors are often required in both training and testing. In such cases, *variational inference* techniques are often adopted, to approximate the intractable posterior by another network (*e.g.*, $q_\phi(\mathbf{z}|\mathbf{x})$ to approximate $p_\theta(\mathbf{z}|\mathbf{x})$ in Fig. 8a).

SGVB [24] is a variational inference algorithm. It uses the *reparameterization trick* to rewrite $\mathbf{z}$ into $\mathbf{z} = \mathbf{z}(\boldsymbol{\xi})$, where $\boldsymbol{\xi} \sim q(\boldsymbol{\xi})$ is a random variable independent of ϕ, and $\mathbf{z}(\cdot)$ is a continuous, differentiable mapping. With this trick, it then trains $p_\theta(\mathbf{x}, \mathbf{z})$ (*i.e.*, the "main" model, which we call *the generative net*) and $q_\phi(\mathbf{z}|\mathbf{x})$ (*the variational net*) simultaneously by *maximizing* just one loss $\mathcal{L}(\theta, \phi)$:

$$
\begin{aligned}
&\mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}\left[\log p_\theta(\mathbf{x})\right] \\
\geq &\mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}\left[\log p_\theta(\mathbf{x})\right] - \mathrm{KL}\left[q_\phi(\mathbf{z}|\mathbf{x}) \| p_\theta(\mathbf{z}|\mathbf{x})\right] \\
= &\mathbb{E}_{\mathbf{x}\sim p_{\text{data}}, \mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}\left[\log p_\theta(\mathbf{x}) + \log p_\theta(\mathbf{z}|\mathbf{x}) - \log q_\phi(\mathbf{z}|\mathbf{x})\right] \\
= &\mathbb{E}_{\mathbf{x}\sim p_{\text{data}}, \mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}\left[\log p_\theta(\mathbf{x}|\mathbf{z}) + \log p_\theta(\mathbf{z}) - \log q_\phi(\mathbf{z}|\mathbf{x})\right] \\
= &\mathbb{E}_{\mathbf{x}\sim p_{\text{data}}, \boldsymbol{\xi}\sim q(\boldsymbol{\xi})}[\log p_\theta(\mathbf{x}|\mathbf{z}(\boldsymbol{\xi})) + \log p_\theta(\mathbf{z}(\boldsymbol{\xi})) \\
&\qquad - \log q_\phi(\mathbf{z}(\boldsymbol{\xi})|\mathbf{x})] \\
= &\mathcal{L}(\theta, \phi)
\end{aligned}
$$

Monte Carlo integration[25] can be used to compute the above expectation, as shown below, where $\mathbf{x}_{(l)}$ are mini-batch samples, and $\boldsymbol{\xi}_{(l)}$ are samples corresponding to each $\mathbf{x}_{(l)}$:

$$
\begin{aligned}
\mathcal{L}(\theta, \phi) \approx \frac{1}{L}\sum_{l=1}^{L}\Big[&\log p_\theta(\mathbf{x}_{(l)}|\mathbf{z}(\boldsymbol{\xi}_{(l)})) + \log p_\theta(\mathbf{z}(\boldsymbol{\xi}_{(l)})) \\
&- \log q_\phi(\mathbf{z}(\boldsymbol{\xi}_{(l)})|\mathbf{x}_{(l)})\Big]
\end{aligned}
$$

It also requires to compute expectations when deriving useful outputs from a trained Bayesian network. For example, in Fig. 8a, $p_\theta(\mathbf{x}) = \mathbb{E}_{\mathbf{z}\sim p_\theta(\mathbf{z})}\left[p_\theta(\mathbf{x}|\mathbf{z})\right]$, which can be approximated by $\frac{1}{L_z}\sum_{l=1}^{L_z} p_\theta(\mathbf{x}|\mathbf{z}_{(l)})$, where $\mathbf{z}_{(l)}$ are samples from $p_\theta(\mathbf{z})$. Although this estimator is unbiased, it typically has large variance. By obtaining samples from an alternative *proposal distribution* $\hat{p}(\mathbf{z})$, the variance could be reduced. This is the *importance sampling* technique[25]. The optimal *proposal distribution* in this case is $\hat{p}(\mathbf{z}) = p(\mathbf{z}|\mathbf{x})$[1], whereas in practice, $q(\mathbf{z}|\mathbf{x})$ is simply used to obtain the $\mathbf{z}_{(l)}$ samples, giving that $p_\theta(\mathbf{x}) \approx \frac{1}{L_z}\sum_{l=1}^{L_z} p_\theta(\mathbf{x}|\mathbf{z}_{(l)})\, p_\theta(\mathbf{z}_{(l)})/q_\phi(\mathbf{z}_{(l)}|\mathbf{x})$.

[1]To compute $\mathbb{E}_{p(x)}\left[f(x)\right]$, the optimal *proposal distribution* is $\hat{p}(x) = |f(x)|\, p(x)/\int |f(x')|\, p(x')\, \mathrm{d}x'$[25], thus $\hat{p}(\mathbf{z}) = p(\mathbf{z}|\mathbf{x})$ for the above case.

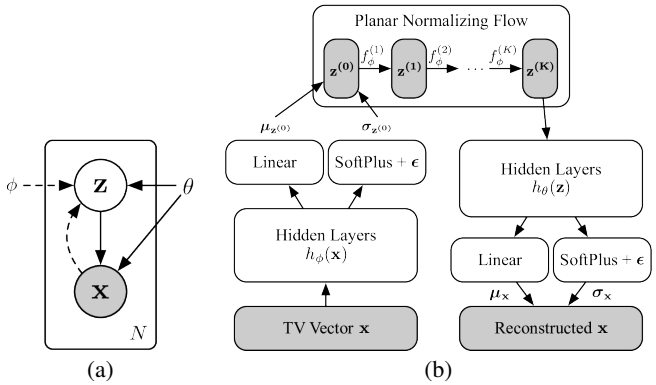


Fig. 8: The architecture of our model: (a) in the form of Bayesian network; (b) in the form of neural network.

Since in variational inference, $q_\phi(\mathbf{z}|\mathbf{x})$ is used to approximate the true posterior $p_\theta(\mathbf{z}|\mathbf{x})$, a too simple family for $q_\phi(\mathbf{z}|\mathbf{x})$ would cause the network to *underfit*. This is exactly the case for the VAE, where they assume $q_\phi(\mathbf{z}|\mathbf{x})$ to be diagonal Gaussian. One solution is to first sample $\mathbf{z}^{(0)}$ from $q^{(0)}(\mathbf{z}^{(0)}|\mathbf{x})$, then transform $\mathbf{z}^{(0)}$ to a series of more complicated distributions, by using $\mathbf{z}^{(k)} = f^{(k)}(\mathbf{z}^{(k-1)})$, where $f^{(k)},\ k = 1 \ldots K$ are *invertible mappings*. In addition, the *Jacobian determinant* of $f^{(k)}$ must be computational tractable, so as to compute $q^{(k)}(\mathbf{z}^{(k)}|\mathbf{x})$. We use *planar mappings*[2] as $f^{(k)}$ in this paper, which is proposed by [18]. We call the normalizing flows constructed from such mappings, the *planar normalizing flows* (*Planar NF* for short).

B. Model Architecture

The architecture of our model is demonstrated in Fig. 8. $\mathbf{x}$ denotes the TV vectors, while $\mathbf{z}$ and $\mathbf{z}^{(k)}$ are latent variables.

In Fig. 8a, the dashed lines denote the *variational net*, *i.e.*, $q_\phi(\mathbf{z}|\mathbf{x})$, while the solid lines denote the *generative net*, *i.e.*, $p_\theta(\mathbf{x}, \mathbf{z})$. The planar normalizing flow is omitted in Fig. 8a, this is because: (1) $\mathbf{z}^{(0)}$ is chosen to be a diagonal Gaussian in our model, so the *reparameterization trick* can be applied on $\mathbf{z}^{(0)}$; and (2) the planar mappings $f^{(k)}$ are deterministic, sufficing the condition of the reparameterization trick, while the log likelihood $q^{(k)}(\mathbf{z}^{(k)}|\mathbf{x})$ can be derived directly according to $q^{(k-1)}(\mathbf{z}^{(k-1)}|\mathbf{x})$ and the Jaccobian determinant of $f^{(k)}$. These facts allow us to treat the whole planar normalizing flow as "hidden layers" of the reparamaterized variational net, omitting $\mathbf{z}^{(0)}$ to $\mathbf{z}^{(K-1)}$, just taking $\mathbf{z}^{(K)}$ as the only latent variable $\mathbf{z}$, resulting in a much simpler training loss (see § III-C) and detection output (see § III-D).

Fig. 8b is the detailed network structure of our model, described in the following two paragraphs.

The TV vectors $\mathbf{x}$ are fed into the *variational net*, passing through hidden layers $h_\phi(\mathbf{x})$, to obtain hidden features. These features are then used to derive the mean $\boldsymbol{\mu}_{\mathbf{z}^{(0)}}$ and the standard deviation $\boldsymbol{\sigma}_{\mathbf{z}^{(0)}}$ of $\mathbf{z}^{(0)}$: $\boldsymbol{\mu}_{\mathbf{z}^{(0)}} = \mathbf{W}_{\boldsymbol{\mu}_{\mathbf{z}^{(0)}}} h_\phi(\mathbf{x}) + \mathbf{b}_{\boldsymbol{\mu}_{\mathbf{z}^{(0)}}}$, and $\boldsymbol{\sigma}_{\mathbf{z}^{(0)}} = \mathrm{SoftPlus}\left(\mathbf{W}_{\boldsymbol{\sigma}_{\mathbf{z}^{(0)}}} h_\phi(\mathbf{x}) + \mathbf{b}_{\boldsymbol{\sigma}_{\mathbf{z}^{(0)}}}\right) + \boldsymbol{\epsilon}$, where

[2]The *planar mappings* proposed by [18] are $f(\mathbf{z}) = \mathbf{z} + \mathbf{u}\tanh(\mathbf{w}^\top \mathbf{z} + b)$, with some constraints on $\mathbf{u}$ to ensure that f is invertible.

$\mathrm{SoftPlus}(\mathbf{a}) = \log(\mathbf{1} + \exp(\mathbf{a}))$, applied on each element of $\mathbf{a}$. $\mathbf{W}_{\boldsymbol{\mu}_{\mathbf{z}^{(0)}}}$, $\mathbf{b}_{\boldsymbol{\mu}_{\mathbf{z}^{(0)}}}$, $\mathbf{W}_{\boldsymbol{\sigma}_{\mathbf{z}^{(0)}}}$ and $\mathbf{b}_{\boldsymbol{\sigma}_{\mathbf{z}^{(0)}}}$ are network parameters to be learnt. $\boldsymbol{\epsilon}$ is a small constant vector, chosen as the minimum value for $\boldsymbol{\sigma}_{\mathbf{z}^{(0)}}$, which can help avoid numerical issues in training (see § III-C), as is adopted in [11]. $\mathbf{z}^{(0)}$ is sampled from $\mathcal{N}\left(\boldsymbol{\mu}_{\mathbf{z}^{(0)}}, \boldsymbol{\sigma}^2_{\mathbf{z}^{(0)}}\mathbf{I}\right)$, then passed through the planar normalizing flow of length K, to obtain $\mathbf{z}^{(K)}$.

The output $\mathbf{z}^{(K)}$ of the planar normalizing flow is used as the latent variable $\mathbf{z}$, and fed into the *generative net*, passing through the hidden layers $h_\theta(\mathbf{z})$, to again obtain hidden features. These features are then used to derive the mean $\boldsymbol{\mu}_\mathbf{x}$ and the standard deviation $\boldsymbol{\sigma}_\mathbf{x}$ of $\mathbf{x}$, just like for $\mathbf{z}^{(0)}$. The reconstructed $\mathbf{x}$ are then sampled from $\mathcal{N}\left(\boldsymbol{\mu}_\mathbf{x}, {\boldsymbol{\sigma}_\mathbf{x}}^2\mathbf{I}\right)$.

We choose to use the planar normalizing flow, because as a deep Bayesian network, it is designed to capture complicated data patterns. Also, it is shown to work better [18] than the VAEs. This can help us tackle the 2nd challenge.

C. Training

According to § III-A and § III-B, it is straightforward to train our model with SGVB, by maximizing $\mathcal{L}(\phi, \theta)$ ((1)) on mini-batches, where $\boldsymbol{\xi}_{(l)}$ are samples from $\mathcal{N}(\mathbf{0}, \mathbf{I})$, and $\mathbf{z}^{(K)}(\boldsymbol{\xi}_{(l)}) = (f^{(K)}_\phi \circ f^{(K-1)}_\phi \circ \cdots \circ f^{(1)}_\phi)(\boldsymbol{\xi}_{(l)} \cdot \boldsymbol{\sigma}_{\mathbf{z}^{(0)}} + \boldsymbol{\mu}_{\mathbf{z}^{(0)}})$. We add subscripts to $f^{(k)}_\phi$ and $q^{(k)}_\phi$, to emphasize the fact that we treat them as part of the variational net. The exact formula of $f^{(k)}_\phi$ and $q^{(k)}_\phi$ can be found in [18].

$$\mathcal{L}(\phi, \theta) \approx \frac{1}{L}\sum_{l=1}^{L}\Big[\log p_\theta(\mathbf{x}_{(l)}|\mathbf{z}^{(K)}(\boldsymbol{\xi}_{(l)})) + \log p_\theta(\mathbf{z}^{(K)}(\boldsymbol{\xi}_{(l)})) - \log q^{(K)}_\phi(\mathbf{z}^{(K)}(\boldsymbol{\xi}_{(l)})|\mathbf{x}_{(l)})\Big] \quad (1)$$

We observed that during training, sometimes $\mathcal{L}(\phi, \theta)$ may bump into very high values, causing severe perturbation on the model parameters. As a result, the model may suddenly get "dragged" out of a local minimal, as if it was initialized to a new random state, causing the training procedure to "restart". We suspect it was because of the terms $-(\boldsymbol{\mu}_{\mathbf{z}^{(0)}} - \mathbf{z}^{(0)})^2/\boldsymbol{\sigma}^2_{\mathbf{z}^{(0)}}$ and $-(\boldsymbol{\mu}_\mathbf{x} - \mathbf{x})^2/{\boldsymbol{\sigma}_\mathbf{x}}^2$ in $\mathcal{L}(\phi, \theta)$, explained as follows. One possible situation is that, due to the TV construction method (§ II), when $\mathbf{x}$ is high-dimensional, many dimensions of each $\mathbf{x}$ would simply be MAX, all having a same value. Therefore, the estimation of $\boldsymbol{\sigma}_{\mathbf{z}^{(0)}}$ and $\boldsymbol{\sigma}_\mathbf{x}$ for a given $\mathbf{x}$ may get very close to zero at some dimensions. When values of such dimensions of $\mathbf{x}$ do not match its estimated $\boldsymbol{\mu}_{\mathbf{z}^{(0)}}$ or $\boldsymbol{\mu}_\mathbf{x}$ very well (which is likely to happen when the model has not fully converged yet), the loss just "explodes". Another possible scenario is that, some of the training data might be actually abnormal, but mistakenly treated as normal data (as discussed in § II-C). This will result in huge difference between the estimated $\boldsymbol{\mu}_\mathbf{x}$ and the input $\mathbf{x}$, again causing the loss to "blow out".

To deal with this problem, we adopt three techniques: (1) use $\boldsymbol{\epsilon}$ to control the minimum values of $\boldsymbol{\sigma}_{\mathbf{z}^{(0)}}$ and $\boldsymbol{\sigma}_\mathbf{x}$ (as [11]); (2) use *gradient clipping* to avoid updating the network parameters too vastly in a single mini-batch; and (3) adopt *early-stopping* during training. However, the amount of our training data is very limited (§ IV-A), such that we cannot afford the cost of splitting out a dedicated validation set. We thus adopt a not common strategy: we use the whole training set to train the model, and after every a few epochs, we use the whole training set to compute $\log p_\theta(\mathbf{x})$ ((2)). We memorize the model parameters which produce the largest $\log p_\theta(\mathbf{x})$, during the whole training procedure.

D. Anomaly Detection

To detect whether or not a TV vector is anomaly, we use its log-likelihood against the model, *i.e.*, $\log p_\theta(\mathbf{x})$. According to § III-A, it can be computed by importance sampling (for simplicity, we omit the superscript for $q^{(K)}_\phi$ and $\mathbf{z}^{(K)}$):

$$\log p_\theta(\mathbf{x}) \approx \log \frac{1}{L_z}\sum_{l=1}^{L_z}\left[\frac{p_\theta(\mathbf{x}|\mathbf{z}_{(l)})\, p_\theta(\mathbf{z}_{(l)})}{q_\phi(\mathbf{z}_{(l)}|\mathbf{x})}\right]$$

For numerical stability, we compute it by:

$$\log p_\theta(\mathbf{x}) \approx \mathrm{LogMeanExp}^{L_z}_{l=1}\Big[\log p_\theta(\mathbf{x}|\mathbf{z}_{(l)}) + \log p_\theta(\mathbf{z}_{(l)}) - \log q_\phi(\mathbf{z}_{(l)}|\mathbf{x})\Big] \quad (2)$$

where for $a_1, \ldots, a_L$ and $a_{max} = \max(a_1, \ldots, a_L)$:

$$\mathrm{LogMeanExp}^{L}_{l=1}(a_l) = a_{max} + \log \frac{1}{L}\sum_{l=1}^{L}\exp(a_l - a_{max})$$

There has been several work using Bayesian networks to do anomaly detection tasks. [26] proposes to use the *reconstruction probability* $\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{z})]$ of VAE to detect anomaly images. [27] uses the reconstruction probability of VRNN [28] to detect anomalies on time series. [11] also adopts the reconstruction probability of VAE, to detect anomalies on KPIs (a special type of time series). In addition, [11] analyzes why this "reconstruction probability", which is actually not a well-defined probability, can work under its situation, and develops several techniques to enhance the model performance when using the reconstruction probability. The experiments of [11] also suggests that these techniques are essential for the reconstruction probability to work.

In this paper, we prefer $\log p_\theta(\mathbf{x})$ to the reconstruction probability, because in our situation, it is hard to develop all the necessary techniques similar to that of [11], so as to ensure the reconstruction probability can work well. We guess the true log-likelihood $\log p_\theta(\mathbf{x})$ is more reliable. The performance comparison between $\log p_\theta(\mathbf{x})$ and the reconstruction probability can be found in § IV-C, which supports our choice.

IV. Evaluation

In this section, historical traces from 46 web services and injected anomalies are used to evaluate the performance of *TraceAnomaly*.

A. Dataset

Our datasets come from a large Internet company S which serves tens of millions of users. The web services in this company adopt the microservices architecture, and the traces are collected by RPC framework.

There are hundreds of different web services in company S. We obtained 46 datasets from 46 services in S. Table I

	min	25th percentile	50th percentile	75th percentile	max
#microservice	3	3	5	11	71
#call	2	2	4	11	103
#TV dimension	2	2	5	41	1120
#structure	2	2	3	12	199

TABLE I: The statistics of the 46 datasets, including the number of different microservices of a dataset (#microservice), the number of different calls of a dataset (#call), the TV dimension of a dataset (#TV dimension), and the number of different trace structures of a dataset (#structure).

shows the statistics of these datasets. We can see that the 46 datasets cover both complex services (*e.g.*, the one with 1120 TV dimensions) and simple services (*e.g.*, the one with 2 TV dimensions). For limited privilege, we only managed to obtain around 40,000 samples of normal trace data for each service.

There are tens of millions of traces every day in company S. However, due to the limited storage space, the traces are kept for only 30 days, so the abnormal traces from further history failures cannot be retrieved. Also, the services of company S are well-maintained. As a result, we only managed to find very few abnormal traces for each service. Due to these reasons, we decide to inject anomalies into normal traces, so as to obtain abnormal data for evaluation.

We observed that the majority of the anomalies are response time anomalies and call relationship interruption anomalies, and these anomalies can be injected in reasonable ways. We cooperated with experienced operators in company S, to design the strategy of injecting these two types of anomalies. We introduce such strategy as follows.

- *Response time anomalies.* We randomly choose a dimension of a trace's TV to inject response time anomaly. The dimension's value is set to a greater value which is k times of the mean of the corresponding normal value, k is randomly chosen from [100, 1000] which based on the few real response time anomalies and the operators' experience. Finally, other dimensions' value of the TV are recalculated according to the call relationships.
- *Call relationship interruption anomalies.* We randomly choose a dimension of a trace's TV to inject interruption anomaly. The dimension's value is set to value MAX to indicate interruption of a microservice. For the call relationships, other related calls will be also interrupted. Therefore, the value of the dimensions corresponding to the related calls are also set to value MAX.

Therefore, for each dataset, we randomly split out 11000 trace data as test set, while the remaining data are used as training set. We further randomly split the test set into three parts, with $(10000, 500, 500)$ trace data. We inject response time anomalies into each trace of the 2nd part, and we inject interruption anomalies into the 3rd part. However, the 3rd part is not always entirely used in testing. According to the observation from Table II, we know that response time anomalies appear more frequently than interruption anomalies, but we do not know the exact ratio. We thus choose to do separated experiments using different subsets (5, 50 and 500 data subset) of the 3rd test set. Due to the page limitation, we only report the results of the $(10000, 500, 50)$ combination. The results of other two cases share the same trend.

B. Experiment Setup

Because our 46 datasets have different number of TV dimensions, it is not appropriate to use $\mathbf{z}$ with fixed dimensions. At the same time, a TV usually contains many 0 or MAX values, which implies that the TVs should lie in a relatively low dimensional manifold, compared to TV. We thus empirically set the relationship between the dimensiona of TV (denoted as $\mathrm{Dim}(\mathbf{x})$) and the dimension of $\mathbf{z}$ (denoted as $\mathrm{Dim}(\mathbf{z})$) to be: $\mathrm{Dim}(\mathbf{z}) = 5 \times 2^{\lfloor \log_{10}(\mathrm{Dim}(\mathbf{x})) \rfloor}$.

The hidden layers of $q_\phi(\mathbf{z}|\mathbf{x})$ and $p_\theta(\mathbf{x}|\mathbf{z})$, *i.e.*, $h_\phi(\mathbf{x})$ and $h_\theta(\mathbf{z})$, are chosen to be two LeakyReLU layers, each with 100 units. $\boldsymbol{\epsilon}$ of the std layers is set to 10^{-4}. We did not carry out exhaustive search on the architecture of hidden layers.

Other hyper-parameters are also chosen empirically. We use 500 as the sampling number L_z of importance sampling. We use 200 as the batch size L for training, and run for 1000 epochs. We use Adam optimizer. The learning rate λ is first set to 10^{-3}, then adjusted to 5×10^{-4} after 200 epochs, 10^{-4} after 400 epochs, 10^{-5} after 550 epochs, and finally 10^{-6} after 800 epochs. We also apply L2 regularization to the hidden layers, with a coefficient 10^{-4}. We clip the gradients by norm, with 10.0 as the limit. We set value MAX in TV to be 600000.

C. Anomaly Detection

Our core idea is to first learn the patterns of normal traces by an unsupervised learning algorithm, then apply the learnt model to detect anomalies. We choose to use planar NF in our framework (as introduced in § III), however, some other unsupervised algorithms can also serve as a drop-in replacement to planar NF in our scenario. We shall introduce these algorithms, before heading towards the evaluation results.

Gaussian Mixture Model (GMM) [29]. GMM is a probabilistic model, assuming that the distribution of data vector $\mathbf{x}$ is a mixture of Gaussian, *i.e.*, $p_{GMM}(\mathbf{x}) = \sum_{i=1}^{n_c} \pi_i \mathcal{N}(\boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i)$, where $\sum \pi_i = 1$. n_c is the number of Gaussian kernels. The more complex the pattern of $\mathbf{x}$ is, the more Gaussian kernels are needed. However, the larger n_c is, the longer time it would cost to train a GMM model, while too many kernels may even cause *overfitting*. Therefore, we use cross-validation in training to choose the optimal number of kernels for each dataset. To speed up the training, we pick up values for n_c only from the Fibonacci sequence $(1, 2, 3, 5, 8, \ldots, 144)$. We use the log density (*i.e.*, $\log p_{GMM}(\mathbf{x})$) as the detection output of GMM.

Diag GMM. Diag GMM is a variant of GMM, where each covariance matrix Σ_i is chosen to be diagonal. This would greatly speed up training ($O(n_c \mathrm{Dim}^2(\mathbf{x}))$ for a standard GMM, and $O(n_c \mathrm{Dim}(\mathbf{x}))$ for a diag GMM), with a potential cost of downgrading the model performance.

Kernel Density Estimation (KDE) [30]. KDE is a non-parametric method to learn the data distribution, which approximates the ground-truth distribution by placing a "kernel" on each training data, and estimate the density of a test data by summing up densities from all kernels. We use Gaussian

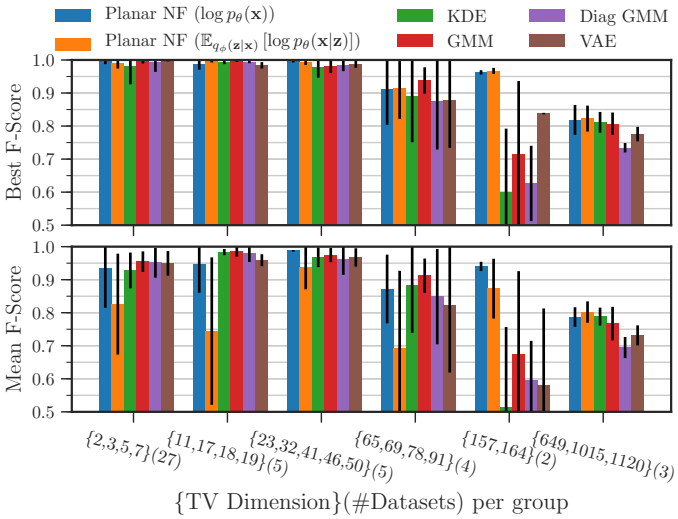


Fig. 9: Performance of the algorithms on the $(10000, 500, 50)$ test set combination, the other combinations share the same trend. 46 dataset are grouped into 6 groups according to their TV dimension. Different color bars indicate the performances of different algorithms on the grouped datasets. x-axis labels are the dimensions of TV and the number of datasets in each group. The top figure presents the mean and variance of best F-scores in each group. The bottom figure demonstrates the mean and variance of F-scores with respect to various enumerated thresholds within a particular range.

kernels, with a fixed *bandwidth*. The precise value of the bandwidth for each dataset is determined by cross-validation. The set of possible bandwidth is $\{10^{-2+0.4\times j} | j = 0, 1, \ldots, 10\}$. We use the log density as the detection output of KDE.

VAE [24]. VAE is a deep Bayesian network, which is the building block of Planar NF. Our chosen algorithm, Planar NF, differs from a VAE in the only way that it applies planar normalizing flow on the posterior $q_\phi(\mathbf{z}|\mathbf{x})$ of a VAE. Just as Planar NF, we use $\log p_\theta(\mathbf{x})$ as the detection output of VAE.

Planar NF with Reconstruction Probability. As introduced in § III-D, this combination uses our trained Planar NF model, with $\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{z})]$ as its detection output .

Next, we introduce the metrics used in our experiments: *F-score* and *best F-score*. All the algorithms introduced here can output one anomaly score (akka, the detection output) for each test trace. This score indicates the likelihood of a trace being *normal*. A threshold is then needed to determine whether or not an alert should be triggered: if the score is smaller than the threshold, then it is likely an abnormal trace. Given a particular threshold, *precision* and *recall* of the alerts can be computed accordingly, as well as *F-score*, the harmonic mean of precision and recall. To compare the performance of algorithms without a threshold, we also enumerate all possible thresholds, obtaining all F-scores, and pick the *best F-score*. The best F-score indicates the best possible performance of a model on a particular test set, given an optimal threshold.

The top figure of Fig. 9 demonstrates the means and variances of *best F-scores*, within each dataset group. Since all the algorithms have a very high best F-score on the first 3 groups, we omit them and concentrate on the later 3 groups. We see that Planar NF (with both $\log p_\theta(\mathbf{x})$ and $\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{z})]$ as outputs) consistently outperforms KDE, Diag GMM and VAE. The fact that Planar NF works better than VAE on these groups (with high dimensional $\mathbf{x}$, thus is more complicated) supports our hypothesis that high capacity models are required for learning the patterns of TV data. The best F-score of Planar NF is larger than 0.9 on the 4th and the 5th group, only dropping to 0.8 on the last group. Although the Planar NF is slightly worse than GMM on the 4th group, it is better than GMM on the 5th and the 6th group. In particular, it has a huge advantage over GMM on the 5th group, where the GMM has a large variance in best F-score, suggesting the GMM is much less stable than Planar NF. Although this exception of GMM might be fixed with some additional tricks, it is out of the scope of our work. We thus claim that Planar NF is better than other algorithms, in the sense of best F-score.

However, the *best F-score* only indicate the best possible performance of a model in theory. In reality, a threshold is always needed for detection. It is a common practice to empirically choose a static threshold for each dataset, or even a global fixed threshold for all datasets. We thus want to evaluate the average performance of the algorithms under various thresholds, within a reasonable range. In order to determine this particular range, we need to examine the distribution of thresholds corresponding to the best F-scores (*best threshold* for short hereinafter). Since the TV dimensions are different among our datasets, and since the overall value range of all detection outputs scale linearly according to the dimension (*e.g.*, $\log p_\theta(\mathbf{x}) = \sum_{i=1}^{\mathrm{Dim}(\mathbf{x})} \log p_\theta(x_i)$; other outputs have similar relationship), we plot the CDF of Best Threshold / $\mathrm{Dim}(\mathbf{x})$ as Fig. 10. We see that most of the mass is concentrated within the range $[-20, 0]$. We thus enumerate $\delta \in [-20, 0]$ by an equal interval of 0.1, leading to 201 different δ values, and use $\delta \cdot \mathrm{Dim}(\mathbf{x})$ as the thresholds for all algorithms on each dataset. We then obtain the average F-score of the algorithms on all datasets, with respect to all the chosen thresholds, which is plotted as the bottom figure of Fig. 9. It is clear that although the two outputs of Planar NF have comparable best F-scores, in most cases, the reconstruction probability works worse than $\log p_\theta(\mathbf{x})$ with enumerated fixed thresholds. This suggests that in practice, it is more difficult to choose a suitable threshold for Planar NF coupled with $\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{z})]$ than with $\log p_\theta(\mathbf{x})$, which rules out the reconstruction probability. All the other algorithms still share the same trend in this mean F-score evaluation as the best F-score evaluation. So we claim that, Planar NF with $\log p_\theta(\mathbf{x})$ is the best algorithm, in the sense of average F-score with enumerated thresholds. Another reason to prefer Planar NF to some other candidates is the time cost for training. The training time of all the algorithms under different TV dimensions is plotted as Fig. 11. We can see that as the dimension of $\mathbf{x}$ increases, GMM and KDE require increasingly larger training time. As a contrary, the Planar NF uses nearly a constant training time on different datasets, and is much lower than GMM and KDE on high dimensional datasets. This relatively

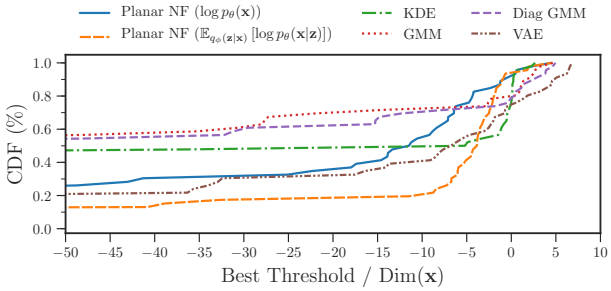


Fig. 10: Distribution of Best Threshold / $\mathrm{Dim}(\mathbf{x})$ on $(10000, 500, 50)$ test set combination. Most of the mass is concentrated within the range $[-20, 0]$. Notice although some algorithms may have a long tail on the left, they may also have a relatively large safe zone for choosing a good threshold, leading to a good average F-score when thresholds are chosen from the range $[-20 \cdot \mathrm{Dim}(\mathbf{x}), 0]$. See Fig. 9 and Fig. 12.

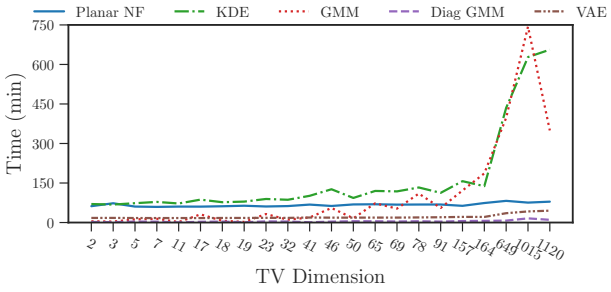


Fig. 11: Training time of different algorithms under different TV dimensions. This includes the cross-validation time for KDE, GMM and Diag GMM. If multiple datasets share the same dimension, the average training time is plotted. Since the two variants of Planar NF only differs in testing, we only plot one line for Planar NF.

constant training time is caused by the following reason. The Planar NF is trained on one GPU per dataset, so the main bottleneck is its depth (about 20 planar mappings) instead of its width (the dimension of $\mathbf{x}$). The total training time on all 46 datasets is 49 GPU hours for Planar NF. In large companies like S, there are around 100 services to be detected, so it would cost about 107 GPU hours to train Planar NF on all datasets. This is a reasonable cost. Because of the above reasons, we claim that Planar NF with $\log p_\theta(\mathbf{x})$ is the best algorithm among all the candidates.

D. Analysis

To examine the internal mechanisms of our model and its competitors, we plot heat maps of the trained models on one of our datasets, as Fig. 12 shows. The TV of the dataset are 2-d. We choose this dataset, because it has relatively the most complicated pattern among all the 2-d datasets, as shown in Fig. 12a (which is plotted by directly drawing each TV as a 20% transparent point on the figure, with the first dimension as x, and the second dimension as y). The other figures in Fig. 12 are plotted as follows. First, we pick 500 values along each of the x- and y-axis, with equal intervals, and obtain 250,000 points from the Cartesian product of x- and y-values. These points are used as the input $\mathbf{x}$ for the models. We then

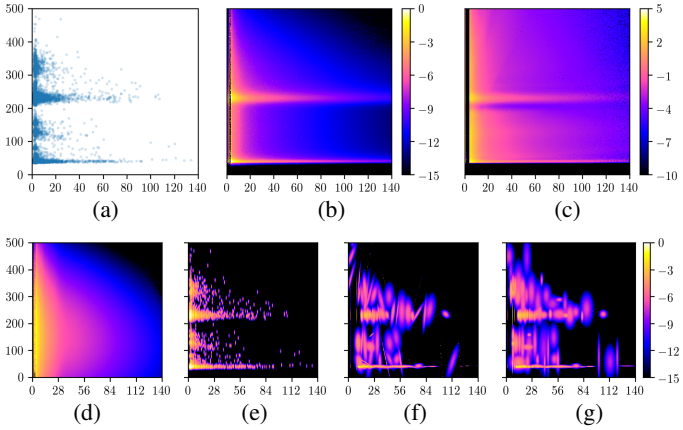


Fig. 12: Heat maps of: (a) $\log p(\mathbf{x})$ of data; (b) $\log p_\theta(\mathbf{x})$ of Planar NF; (c) $\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{z}|\mathbf{x})]$ of Planar NF; (d) $\log p_\theta(\mathbf{x})$ of VAE; (e) log density of KDE; (f) log density of GMM; and (g) log density of Diag GMM. Notice in (c) the upper and lower limit is different from others.

compute the score ($\log p_\theta(\mathbf{x})$, $\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{z}|\mathbf{x})]$, and log density, respectively) of each $\mathbf{x}$ corresponding to each model. Finally, we plot these scores as the heat maps of Fig. 12.

Fig. 12 suggests that our chosen model, Planar NF (Fig. 12b), fits the data pattern much better than other models. The VAE (Fig. 12d) fails to capture the details. The KDE (Fig. 12e) puts one Gaussian kernel at each training point (the bandwidth of each kernel is around 0.02512, selected by cross validation), which might be insufficient to cover all the test points in high-dimensional spaces. The GMM and Diag GMM (Figs 12f and 12g) tries hard to fit the data pattern with a fixed number of Gaussian kernels (144 kernels in both figures, determined by cross validation), however, is still far from the true pattern.

Fig. 12 also explains why the threshold is easier to tune when using $\log p(\mathbf{x})$ than using $\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{z}|\mathbf{x})]$. The upper and lower limit of Fig. 12c is adjusted, such that the left area is roughly as same bright as Fig. 12b. However, the right top corner and the right bottom area of Fig. 12c is far brighter than Fig. 12b. It is clear that those right areas should be dark, since barely any training data present there. As a result, the "safe zone" of choosing a proper threshold for $\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{z}|\mathbf{x})]$ is fairly narrow, making it difficult to tune the threshold. This is why, in average, Planar NF performs badly when using $\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{z}|\mathbf{x})]$ coupled with fixed thresholds (Fig. 9), which confirms our choice to use $\log p_\theta(\mathbf{x})$.

V. Operational Experience

We have deployed *TraceAnomaly* on six services (different from those in § IV) in company S's production system for weeks. In the deployment, the model is retrained retrained every day using the traces of last seven days. We now evaluate *TraceAnomaly*'s performance during one of these weeks and study one case in detail.

Since the number of traces is huge, it is impossible to label all traces. Therefore we focus on precision and let the operators check all the abnormal traces detected by *TraceAnomaly*.

service	#TV-dimension	#P	#TP	#TP-RT	#TP-call relation	Precision
s1	26	73	70	55	15	0.96
s2	21	187	181	108	73	0.97
s3	14	351	323	320	3	0.92
s4	4	3	3	2	1	1.0
s5	4	13	12	1	11	0.92
s6	2	4	4	3	1	1.0

TABLE II: Performance of *TraceAnomaly* on six real web services during a week. P=detected anomalies; TP=confirmed anomalies; TP-RT=confirmed response time anomalies; RT-call relationship=confirmed call relationship anomalies. Precision=$\frac{\text{\# confirmed anomaly}}{\text{\# detected anomaly}}$.

Table II shows the results. We can see that *TraceAnomaly* achieves a precision of at least 0.92.

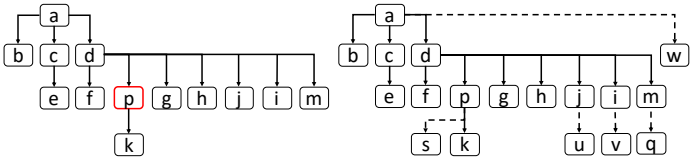
The false positives detected by *TraceAnomaly* are mainly due to the execution of very rare code logic. Such executions appear once in a few weeks thus were not the in the training trace set of one week. The trace-off between the precision with the training data length is left for our future work.

Case Study. Fig. 13a shows the details of one of confirmed abnormal traces in service s1 (Table II). Fig. 13b shows the structure of the abnormal trace. The root cause of the abnormal trace is that microservice p had some times out when executing some internal logic, which result in microservices s, u, v, q, w not being called. As a result, although the response time of microservice p is more than expected, but some microservices' response time decreased because microservice s, u, v, q, w were not called. Therefore, there is no significant difference in the total execution time for $s1$ for this abnormal trace (1269 msec) compared with a normal trace (1244 msec) of $s1$. Thus the anomaly solely based on $s1$'s total execution time will not be able to catch this anomaly. On the other hand, coincidently, some normal logic of in $s1$ cause some of its traces to have the same structure with the abnormal trace (Fig. 13b). Therefore, this abnormal trace can not be detected either by the trace structure. *TraceAnomaly* can successfully detected the anomaly because that *TraceAnomaly* can extract each call path and its response time, then the long time cost of microservice p can be reflected on the trace's TV, which can be detected by *TraceAnomaly*.

Sending time of (m→n) at m (msec)	Receiving time of (m→n) at n (msec)	Message (m→n)
-	1534780807651	call(start→a)
1534780807653	1534780807655	call(a→b)
1534780807795	1534780807796	*response*(b→a)
1534780807797	1534780807798	call(a→c)
1534780807808	1534780807809	call(c→e)
1534780807812	1534780807813	*response*(e→c)
1534780807817	1534780807818	*response*(c→a)
1534780807820	1534780807823	call(a→d)
1534780807829	1534780807831	call(d→f)
1534780807835	1534780807837	*response*(f→d)
1534780807856	1534780807857	**call(d→p)**
1534780807858	1534780807861	call(d→g)
1534780807858	1534780807860	call(d→h)
1534780807864	1534780807865	*response*(h→d)

Sending time of (m→n) at m (msec)	Receiving time of (m→n) at n (msec)	Message (m→n)
1534780807909	1534780807911	*response*(g→d)
1534780808840	1534780808842	call(p→k)
1534780808844	1534780808846	*response*(k→p)
1534780808856	**1534780808858**	***response*(p→d)**
1534780808890	1534780808892	call(d→j)
1534780808890	1534780808892	call(d→l)
1534780808890	1534780808892	call(d→m)
1534780808904	1534780808905	*response* (i→d)
1534780808904	1534780808905	*response* (l→d)
1534780808909	1534780808911	*response* (m→d)
1534780808915	1534780808917	*response* (d→a)
1534780808920	-	*response* (a→end)

(a)



(b) (c)

Fig. 13: (a) abnormal trace which takes 1269 msec; (b) structure of the abnormal trace, which coincidently is the same some other normal trace; and (c) the normal structure that this trace should have, which normally takes 1244 msec.

VI. Related Work

This section briefly reviews the related work on anomaly detection of traces, logs, time series, and graphs.

Anomaly detection in traces and logs. There are some previous works that focused on the detection of the structural anomalies in traces and logs. [31], [32], [33] use clustering methods. [22] mines control flow graph to detect anomalies in execution flows. [19], [34] build automata structure to detect anomalies within n-gram connection. [35] uses short system calls sequences as normal behavior's features based on n-grams. Above approaches focus on detecting structural anomalies, and cannot deal with the case in Fig. 5 where an abnormal trace can have the same structure as a normal trace. There are some works that mine more than just structures, but they are not applicable to our problem either. [36] mines static relationship between different logs' parameters. [21] builds Finite State Automaton and detect anomalies in time intervals sequence of each state transition, and [23] uses Long Short-Term Memory (LSTM) networks to detect each parameter separately. These methods cannot be used in our problem because they cannot capture the complex dynamic correlation between response time of different microservices in our traces (see § II-A), which is affected by input parameters and runtime conditions.

Traditional statistical algorithms such as [3], [4], [5], [6], [7], [8], [9], [37], [38], supervised algorithms such as [10], [39], and unsupervised algorithm [11] for **time series anomaly detection** all require a dedicated model for each time series. Thus they are not applicable to our trace anomaly detection problem, due to large number of call paths whose response time anomalies need to be detected.

Graph anomaly detection algorithms [12], [13], [14], [15], [16], [17] encode a node or sub-graph in a graph into vectors, and detect anomalies according to these vectors. Such vectors lack clear physical significance. Thus it is not guaranteed that the anomalies we want to detect can be reflected in the constructed vectors.

VII. Conclusion

This paper presents *TraceAnomaly*, an unsupervised anomaly detection approach that can automatically learn the overall normal patterns of the traces offline, and detect anomalies via computing the likelihood based on the learned normal pattern. *TraceAnomaly* has a novel trace vector encoding and deep variational bayesian networks with planar normalizing flows. Our experiments using 46 different web services show that *TraceAnomaly* can achieve an average F-scores of 0.79 $\sim$ 0.99 with a fixed threshold. *TraceAnomaly* has been deployed

on six real web service for a week, and the precision of detected anomalies can achieve above 0.92.

REFERENCES

[1] J. Thönes, "Microservices," *IEEE software*, vol. 32, no. 1, pp. 116–116, 2015.

[2] B. H. Sigelman, L. A. Barroso *et al.*, "Dapper, a large-scale distributed systems tracing infrastructure," Technical report, Google, Inc, Tech. Rep., 2010.

[3] S.-B. Lee, D. Pei *et al.*, "Threshold compression for 3g scalable monitoring," in *INFOCOM, 2012 Proceedings IEEE*. IEEE, 2012, pp. 1350–1358.

[4] B. Krishnamurthy, S. Sen, Y. Zhang, and Y. Chen, "Sketch-based change detection: methods, evaluation, and applications," in *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*. ACM, 2003, pp. 234–247.

[5] A. H. Yaacob, I. K. Tan *et al.*, "Arima based network anomaly detection," in *Communication Software and Networks, 2010. ICCSN'10. Second International Conference on*. IEEE, 2010, pp. 205–209.

[6] F. Knorn and D. J. Leith, "Adaptive kalman filtering for anomaly detection in software appliances," in *INFOCOM Workshops 2008, IEEE*. IEEE, 2008, pp. 1–6.

[7] H. Yan, A. Flavel *et al.*, "Argus: End-to-end service anomaly detection and localization from an isp's point of view," in *INFOCOM, 2012 Proceedings IEEE*. IEEE, 2012, pp. 2756–2760.

[8] Y. Chen, R. Mahajan *et al.*, "A provider-side view of web search response time," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, ser. SIGCOMM '13. New York, NY, USA: ACM, 2013, pp. 243–254. [Online]. Available: http://doi.acm.org/10.1145/2486001.2486035

[9] A. Mahimkar, Z. Ge *et al.*, "Rapid detection of maintenance induced changes in service performance," in *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*. ACM, 2011, p. 13.

[10] D. Liu, Y. Zhao *et al.*, "Opprentice: Towards practical and automatic anomaly detection through machine learning," in *Proceedings of the 2015 ACM Conference on Internet Measurement Conference*, ser. IMC '15. New York, NY, USA: ACM, 2015, pp. 211–224.

[11] 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 *Proceedings of the 2018 World Wide Web Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2018, pp. 187–196.

[12] L. B. Holder and D. J. Cook, "Graph-based data mining," in *Encyclopedia of data warehousing and mining*. IGI Global, 2005, pp. 540–545.

[13] C. Liu, X. Yan, H. Yu, J. Han, and P. S. Yu, "Mining behavior graphs for "backtrace" of noncrashing bugs," in *Proceedings of the 2005 SIAM International Conference on Data Mining*. SIAM, 2005, pp. 286–297.

[14] W. Eberle and L. Holder, "Discovering structural anomalies in graph-based data," in *Data Mining Workshops, 2007. ICDM Workshops 2007. Seventh IEEE International Conference on*. IEEE, 2007, pp. 393–398.

[15] C. C. Noble and D. J. Cook, "Graph-based anomaly detection," in *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2003, pp. 631–636.

[16] D. Chakrabarti, "Autopart: Parameter-free graph partitioning and outlier detection," in *European Conference on Principles of Data Mining and Knowledge Discovery*. Springer, 2004, pp. 112–124.

[17] L. Akoglu, M. McGlohon, and C. Faloutsos, "Oddball: Spotting anomalies in weighted graphs," in *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 2010, pp. 410–421.

[18] D. Rezende and S. Mohamed, "Variational Inference with Normalizing Flows," in *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, 2015, pp. 1530–1538.

[19] G. Jiang, H. Chen, C. Ungureanu, and K. Yoshihira, "Multiresolution abnormal trace detection using varied-length n-grams and automata," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 37, no. 1, pp. 86–97, 2007.

[20] J.-G. Lou, Q. Lin, R. Ding, Q. Fu, D. Zhang, and T. Xie, "Software analytics for incident management of online services: An experience report," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2013, pp. 475–485.

[21] Q. Fu, J.-G. Lou, Y. Wang, and J. Li, "Execution anomaly detection in distributed systems through unstructured log analysis," in *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*. IEEE, 2009, pp. 149–158.

[22] A. Nandi, A. Mandal, S. Atreja, G. B. Dasgupta, and S. Bhattacharya, "Anomaly detection using program control flow graph mining from execution logs," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2016, pp. 215–224.

[23] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 1285–1298.

[24] D. P. Kingma and M. Welling, "Auto-Encoding Variational Bayes," in *Proceedings of the International Conference on Learning Representations*, 2014.

[25] C. P. Robert and G. Casella, *Monte Carlo Statistical Methods (Springer Texts in Statistics)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[26] J. An and S. Cho, "Variational autoencoder based anomaly detection using reconstruction probability," SNU Data Mining Center, Tech. Rep., 2015.

[27] M. Sölch, J. Bayer, M. Ludersdorfer, and P. van der Smagt, "Variational inference for on-line anomaly detection in high-dimensional time series," *International Conference on Machine Laerning Anomaly detection Workshop*, 2016.

[28] J. Chung, K. Kastner, L. Dinh, K. Goel, A. C. Courville, and Y. Bengio, "A recurrent latent variable model for sequential data," in *Advances in neural information processing systems*, 2015, pp. 2980–2988.

[29] D. Reynolds, "Gaussian mixture models," *Encyclopedia of biometrics*, pp. 827–832, 2015.

[30] V. A. Epanechnikov, "Non-parametric estimation of a multivariate probability density," *Theory of Probability & Its Applications*, vol. 14, no. 1, pp. 153–158, 1969.

[31] R. Vaarandi, "A data clustering algorithm for mining patterns from event logs," in *IP Operations & Management, 2003.(IPOM 2003). 3rd IEEE Workshop on*. IEEE, 2003, pp. 119–126.

[32] E. Manzoor, S. M. Milajerdi, and L. Akoglu, "Fast memory-efficient anomaly detection in streaming heterogeneous graphs," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2016, pp. 1035–1044.

[33] W. Dickinson, D. Leon, and A. Podgurski, "Finding failures by cluster analysis of execution profiles," in *Proceedings of the 23rd international conference on Software engineering*. IEEE Computer Society, 2001, pp. 339–348.

[34] C. C. Michael and A. Ghosh, "Simple, state-based approaches to program-based anomaly detection," *ACM Transactions on Information and System Security (TISSEC)*, vol. 5, no. 3, pp. 203–237, 2002.

[35] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for unix processes," in *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on*. IEEE, 1996, pp. 120–128.

[36] J.-G. Lou, Q. Fu, S. Yang, Y. Xu, and J. Li, "Mining invariants from console logs for system problem detection." in *USENIX Annual Technical Conference*, 2010, pp. 23–25.

[37] B. Pincombe, "Anomaly detection in time series of graphs using arma processes," *Asor Bulletin*, vol. 24, no. 4, p. 2, 2005.

[38] W. Lu and A. A. Ghorbani, "Network anomaly detection based on wavelet analysis," *EURASIP Journal on Advances in Signal Processing*, vol. 2009, p. 4, 2009.

[39] N. Laptev, S. Amizadeh, and I. Flint, "Generic and scalable framework for automated time-series anomaly detection," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2015, pp. 1939–1947.