This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2821693, IEEE Transactions on Dependable and Secure Computing

Adaptive Performance Anomaly Detection in Distributed Systems Using Online SVMs

Javier Álvarez Cid-Fuentes, Claudia Szabo, and Katrina Falkner

Abstract—Performance anomaly detection is crucial for long running, large scale distributed systems. However, existing works focus on the detection of specific types of anomalies, rely on historical failure data, and cannot adapt to changes in system behavior at run time. In this work, we propose an adaptive framework for the detection and identification of complex anomalous behaviors, such as deadlocks and livelocks, in distributed systems without historical failure data. Our framework employs a two-step process involving two online SVM classifiers on periodically collected system metrics to identify at run time normal and anomalous behaviors such as deadlock, livelock, unwanted synchronization, and memory leaks. Our approach achieves over 0.70 F-score in detecting previously unseen anomalies and 0.78 F-score in identifying the type of known anomalies with a short delay after the anomalies appear, and with minimal expert intervention. Our experimental analysis uses system execution traces from our in-house distributed system with varied behaviors and a dataset by Yahoo!, and shows the benefits of our approach as well as future research challenges.

1 Introduction

Computer and communication systems are a pivotal part of current society to the point that some authors have coined the term *the network society* [1] to refer to their profound impact on present economic, social, political and cultural aspects. Most current software systems can be considered distributed systems, that is, *"systems in which components located on networked computers communicate and coordinate their actions by passing messages"* [2]. Since the global spread of the Internet, most computers communicate across some kind of network, and often perform actions in response to messages received from other computers. Even more, with the increasing popularity of the *Internet of Things* (IoT) [3], even a group of ordinary lamps may communicate with each other [4].

Due to this global interconnectivity, most distributed systems are also large-scale systems with hundreds, thousands or even millions of components. Examples are web services such as the widely used Google search engine [5], Facebook [6], or Amazon [7]; peer-to-peer (P2P) networks used for file sharing such as BitTorrent [8]; high performance computing (HPC) clusters; privacy oriented networks like TOR [9]; the internal networks of companies and universities; vast networks of small devices in the context of IoT; virtualized infrastructures on demand such as Amazon EC2 [10]; or large-scale systems from other sectors like finance, health, or the military.

Even though different distributed systems have different purposes and characteristics, dependability is crucial, and all systems need to ensure correct behavior, maximize efficiency and availability, protect the privacy of their users, and avoid malicious activities among other requirements. Failing to fulfill one or more of these requirements can cause economic losses [11], exposure of sensitive information [12] or, in extreme cases, loss of life [13]. For example, reports show that the cost of downtime in industry ranges from $100K to $540K per hour on average [14].

Threats to dependability range from software flaws to physical conditions (e.g., room temperature), to malicious attacks [15]. These threats can produce errors in the system, and errors end up causing failures [15]. The negative effects of system errors could be ameliorated with better monitoring tools that can quickly detect anomalies in the system [16]. In addition, it is important not only to detect the presence of an anomaly, but also to identify the anomaly type. This can potentially improve the dependability of the system as the root cause of problems can be more easily identified, without the need for the system expert to analyze large amounts of information [17].

Unfortunately, anomaly detection in large-scale systems is an arduous task with low efficiency [18]. On the one hand, analyzing a great number of components and interactions is difficult in terms of complexity and magnitude [19]. On the other hand, most of the anomalies that appear in running systems have not been seen before, and thus it is difficult to detect them, as historical failure data is unavailable [20], [21].

A popular method for anomaly detection in large-scale systems and networks is black box analysis, which consists of analyzing external system information, such as resource consumption, to find symptoms that can be indicative of the presence of an error [22]. However, existing approaches still have limitations that must be addressed in order to detect anomalies with increased accuracy in real world scenarios. Key issues include the need for historical failure data [23], not taking into account metric correlations [24], [25], [26], [27], not analyzing the system over a time period [27], and the lack of adaptiveness [28].

In this paper we present BARCA, a generic frame-

• *The authors are with the School of Computer Science at The University of Adelaide.*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2821693, IEEE Transactions on Dependable and Secure Computing

work for identifying anomalous behaviors in large-scale distributed systems. BARCA is designed to overcome the challenges faced by existing works by using a combination of several classifiers in a two-step process that allows the detection of previously unseen anomalies, as well as the identification of their type. BARCA builds and maintains a model of the system behavior at run time, and obtains relevant properties from this model that are used to classify the system behavior into normal or anomalous. This is done by means of several *support vector machines* (SVM) [29]. Since the behavioral model is built and updated at run time, BARCA does not require historical failure data and can adapt to changes in behavior. Moreover, BARCA does not depend on the architecture of the system under study, or makes any assumption on the characteristics of the anomalies, which means that can be employed in numerous scenarios and domains.

The rest of this paper is organized as follows: Section 2 reviews similar works in the area of anomaly detection in distributed systems, and analyzes their limitations. Section 3 describes our adaptive framework for anomaly detection in detail. Section 4 presents an extensive experimental analysis, and Section 5 concludes the paper.

2 Related Work

A common approach to anomaly detection in software systems is *program anomaly detection* [30], which consists of finding illegal control flows and anomalous system call sequences in program traces or execution logs. For example, Shu et al. [31] propose a method to detect co-occurrence and frequency anomalies in sequences of system calls using agglomerative clustering and one-class SVMs. Another example is DeepLog [32], a framework that employs a series of Long Short-Term Memory networks to find anomalies in long sequences of log entries. Some program anomaly detection approaches are extremely accurate, as they employ context-sensitive models [31], which are the most accurate detectors in theory [33]. However, program anomaly detection often requires code instrumentation, and can introduce a significant overhead on the system execution.

An alternative to program anomaly detection that does not require code instrumentation and introduces less overhead is *performance anomaly detection* [22]. Performance anomaly detection consists of finding anomalous deviations in the system performance metrics represented as temporal data. Despite being less accurate than program anomaly detection, performance anomaly detection is better suited for certain scenarios where access to the source code of the application is restricted. In this paper, we present a performance anomaly detection framework and thus in the remainder of this section we review relevant works in this particular area.

CloudPD [25] is an anomaly detection and remediation framework devised for cloud infrastructures. CloudPD's anomaly detection engine has two stages. In the first stage, an anomaly detector is used to find deviations in virtual machines' performance metrics based on a model of normal recent behavior. Metrics with significant deviations are further analyzed in the second stage where the anomaly is confirmed if there is a significant change in the correlation of the candidate metric and other metrics of the same virtual machine, or other virtual machines running the same application. Since CloudPD employs a model of normal behavior based on recent history, it is able to detect previously unseen anomalies and can adapt to changes in normal behavior.

Guan and Fu [23] present a method for identifying anomalies in cloud infrastructures. This method employs PCA to select the system metrics that are more correlated with failures. A Kalman filter [34] is then employed to estimate the value of the selected metrics at a time point. An anomaly is detected if the difference between the estimation and the actual value is greater than a predefined threshold. Guan and Fu's method can adapt to changes in behavior by updating the selected metrics when new information is obtained. However, since metric selection depends on previously seen failures, the method cannot detect novel anomalies that affect other system metrics.

O'Shea et al. [26] present a framework for the detection of anomalies in cloud platforms that is divided in three stages. In the first stage, historical data from the system under study can be preprocessed using various techniques (e.g., PCA). In the second stage, the processed data is employed to build a model of the normal behavior of the system using a wavelet transform and the Welford algorithm. In the third stage, the behavioral model is compared to data from the running system, and anomalies are detected when there is a significant deviation. This framework analyzes the system over a time period, and can be employed to detect different types of anomalies. However, it requires an extensive training phase and does not consider the correlation between the different system metrics.

Yu et al. [28] propose an anomaly detection framework for large-scale infrastructures. Yu et al. first group nodes together based on their geographical location and the network topology. Their framework then clusters together nodes of similar behavior within each group, and employs a two-phase majority voting algorithm to detect anomalous nodes in each cluster. The framework is able to detect previously unseen anomalies as long as these deviate from the behavior of the majority of the nodes. However, it is not clear if node clustering can be updated at run time if the behavior of the system changes.

Ibidunmoye et al. [27] propose an adaptive framework for the detection of anomalous measurements in metric streams. This framework is based on two detectors: a behavior-based detector and a prediction-based detector. The behavior-based detector analyzes a system metric using tumbling windows, and iteratively estimates the conditional density of the last measurement given the most recent window. The last measurement observed is then classified as anomalous if this conditional density estimate is lower than a particular threshold. The prediction-based detector analyzes metric streams using sliding windows, which are iteratively employed to make forecasts of the last measurement. If the difference between the forecasted and observed values is greater than a threshold, the measurement is considered anomalous. This framework does not require historical data as it is based on the most recent observations. However, the framework does not analyze the system over a time period and does not take into account metric correlations.

Even though there have been significant advances in

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2821693, IEEE Transactions on Dependable and Secure Computing

performance anomaly detection in distributed systems over the years, some challenges remain. We identify the following main issues in existing works:

Use of historical failure data: Some of the existing works [23] rely on historical failure data to detect anomalies. However, historical failure data is unavailable in most real world scenarios [21]. Moreover, obtaining samples of all the anomalies that can take place in a distributed system is impossible [35]. Thus, works that rely on historical failure data are limited to the detection of well-known anomalies, or anomalies that exhibit well-known patterns.

Time awareness: Many anomalies can only be detected when observed over a time period rather than at a time instant [19]. However, some works do not analyze the system over time [27]. This means that these works can only detect anomalies characterized by a single anomalous data point (i.e., point anomalies), and not other more complex anomalies such as periodic patterns.

Multi-metric: Some anomalies are characterized by an anomalous combination of multiple performance metrics. For example, high CPU utilization might be normal if the system workload is also high and abnormal otherwise [36], where the system workload can be characterized by the number of requests received per time unit. Another example are anomalies where different metrics that should be independent from each other end up being correlated [37]. Existing works [23], [25], [26], [27] that do not take into account metric correlations cannot detect these kind of anomalies.

Adaptiveness: The behavior of a running system changes over time. Therefore, an effective anomaly detection framework needs to be able to adapt to changes in this behavior by dynamically incorporating new data into its statistical model. Some of the existing approaches do not implement this kind of adaptiveness [28].

Progressive changes: Some works [25], [27] detect anomalies when a significant difference between current monitored data and recent past data is found. These approaches might find difficulties in detecting anomalies that exhibit a progressive change in the monitored metrics, such as a trend [22], since the difference between current readings and the recent past might not be significantly high.

3 Behavior Identification Framework

In this section we describe BARCA, our framework for the identification of anomalies in distributed systems. Fig. 1 shows BARCA's main components and how they interact with each other. BARCA has been designed to overcome the main limitations in existing works (see Section 2) by not relying on historical failure data, analyzing system behavior over a time period, taking into account metric correlations, and adapting to changes in system behavior by means of an incremental process. This adaptiveness does increase the complexity of BARCA's behavioral model, as there is a trade-off between keeping old knowledge and incorporating new data. The trade-off incurred by these improvements is that the performance of BARCA is inherently dependent on the quality and quantity of user feedback.

BARCA is composed of three main components: *Behavior Extractor*, *Behavior Identifier*, and *Feedback Provider*. The *Behavior Extractor* periodically collects system performance metrics from a running distributed system and generates a *Behavior Instance* (BI) every time new data is available. BIs are representations of the system behavior in a time period. The *Behavior Identifier* classifies each BI as normal or anomalous using a statistical model called the *Behavioral Model* (BM). When a BI is classified as anomalous, the *Behavior Identifier* informs the *Feedback Provider*. The *Feedback Provider* then decides whether to alert the system administrator. When alerted, the system administrator informs the *Feedback Provider* about the actual state of the system (i.e., normal or anomalous). The *Feedback Provider* transmits this information to the *Behavior Identifier*, which can then update the BM to incorporate the new data.

3.1 Threat Model

Two main types of anomalous deviations in the system performance metrics are detected, namely, *collective anomalies* (i.e., anomalous sequences of metric readings [22]) and *correlation anomalies* (i.e., anomalous correlations between two or more system metrics). This means that BARCA can potentially detect any threat or error that produces these deviations on the monitored metrics. For example, a deadlock where CPU utilization suddenly drops produces a collective anomaly, and a memory leak where memory utilization increases while CPU utilization remains the same produces a correlation anomaly, assuming CPU and memory utilization are usually correlated. Thus, BARCA is not tied to specific threats or errors but to their impact on the system performance metrics that are monitored.

3.2 Behavior Extractor

The *Behavior Extractor* periodically collects system performance metrics from a running distributed system. These metrics may be *hardware metrics*, such as the CPU, memory, or the network utilization of the system nodes, *operating system metrics*, such as the number of running processes or threads in each node, or *software metrics*, such as the number of exchanged messages between applications. These metrics can be represented as time series, where each data point is the value of a certain metric at a particular time instant. Thus, given a time frame, the behavior of the system can be defined by a set of time series. This set of time series is what we call a *Behavior Instance* (BI).

Formally, given the set of all the available system metrics $M = \{m_i \mid 1 \leq i \leq n\}$, we denote r_{ij} as the reading for metric m_i at time instant j, and $s_{ij} = r_{ij}, r_{i(j+1)}, ..., r_{i(j+z)}$ as the sequence of z readings for metric m_i starting with r_{ij}. We then denote the BI at time instant j as the set $B_j = \{s_{ij} \mid 1 \leq i \leq n\}$, a collection of n time series of size z. Behavior Instances are thus created in a *sliding window* of constant size z, where z determines the granularity of the analysis. Every time a new set of readings $\{r_{i(j+z+1)} \mid 1 \leq i \leq n\}$ is obtained, a new BI is generated as $B_{j+1} = \{s_{i(j+1)} \mid 1 \leq i \leq n\}$, where $s_{i(j+1)} = r_{i(j+1)}, r_{i(j+2)}, ..., r_{i(j+z+1)}$.

The *Behavior Extractor* applies a feature extraction process to BIs after generating them. A feature is a function that derives some relevant property from a dataset. In this case, features extract relevant properties from the time

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2821693, IEEE Transactions on Dependable and Secure Computing

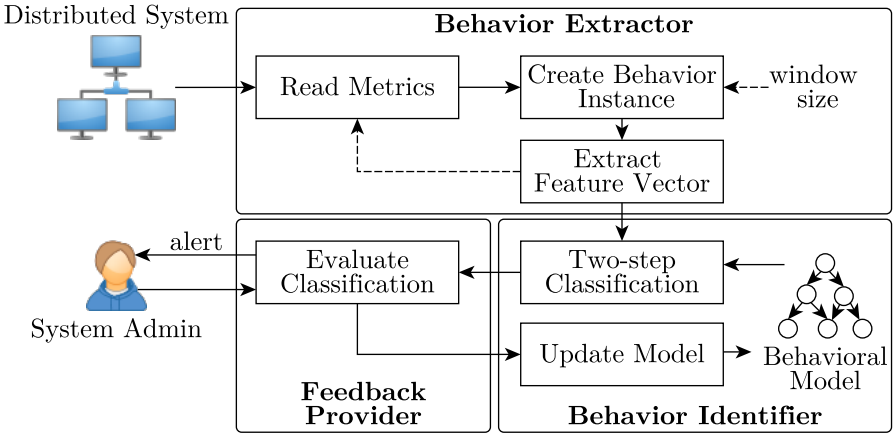


Fig. 1: BARCA's design.

series data such as mean, standard deviation, skewness, or kurtosis [38]. Computing these features serves two purposes. On the one hand, it provides a more robust way of comparing time series than straightforward differences between readings such as Euclidean distances [38]. On the other hand, it reduces the dimensionality of the data. After the feature extraction process, each BI is transformed into a *feature vector*. Feature vectors represent the behavior of the system during a time period in an N-dimensional space called the feature space. Detection of anomalous behaviors can then be done by detecting deviations in this feature space.

Formally, given a set of features $S_F = \{f_k \mid 1 \leq i \leq u\}$, where $f_k : \mathbb{R}^z \rightarrow \mathbb{R}^{d_k}$ is a feature that reduces the dimension of the data from z to d_k, we denote $v_{ij} = f_1(s_{ij}), f_2(s_{ij}), ..., f_k(s_{ij})$ as the sequence of all the features in S_F applied to the time series s_{ij}. We then define $F(B_j)$ as the concatenation of all the features applied to all the time series in B_j. Thus, the feature vector of B_j is $F(B_j) = v_{1j}, v_{2j}, ..., v_{nj} = x_j$, and $F : \mathbb{R}^{n \times z} \rightarrow \mathbb{R}^N$, where $N = \sum_{k=1}^{u} d_k$. This process is depicted in Fig. 2.

$$
\begin{array}{lllll}
\underbrace{\phantom{s_{1j}}}_{B_j} & & & & \underbrace{\phantom{v_{1j}}}_{x_j} \\
s_{1j} : r_{1j}, \ldots, r_{1(j+z)} & \rightarrow & f_1(s_{1j}), f_2(s_{1j}), \ldots, f_u(s_{1j}) & \rightarrow & v_{1j} \\
s_{2j} : r_{2j}, \ldots, r_{2(j+z)} & \rightarrow & f_1(s_{2j}), f_2(s_{2j}), \ldots, f_u(s_{2j}) & \rightarrow & v_{2j} \\
\vdots \quad\quad\quad\quad\quad\quad & & \vdots & & \vdots \\
s_{nj} : r_{nj}, \ldots, r_{n(j+z)} & \rightarrow & f_1(s_{nj}), f_2(s_{nj}), \ldots, f_u(s_{nj}) & \rightarrow & v_{nj}
\end{array}
$$

Fig. 2: Derivation process of feature vector x_j from Behavior Instance B_j and a set of features S_F. Each feature is applied to each sequence of readings in B_j, and the results are concatenated to form x_j.

Feature vectors capture the evolution of the system metrics over time and the relationships between these metrics. Therefore, deviations in the feature space can be caused by a change in one of the metrics, as well as by changes in the correlation of multiple metrics. Moreover, contextual information can be easily included in the feature vector by representing context as a system metric. For example, a high demand context could be represented as the number of client requests received. Since the feature vector takes into account metric correlations, different contexts are automatically represented as different regions in the feature space, and anomalies characterized by deviations in a specific context can be detected.

3.3 Behavior Identifier

The *Behavior Identifier* is in charge of building and maintaining the *Behavioral Model* (BM), of deciding whether a feature vector is anomalous, and of identifying the type of detected anomalies. This is done by means of several classifiers. A classifier is a supervised learning method to categorize sets of data [39]. Typically, a classifier is first trained with a set of labeled data, that is, data whose category is known, and then used to categorize unlabeled data afterwards. The process of categorizing unlabeled data is also known as label prediction. Building the training dataset requires a system expert to label historical data belonging to the considered categories. However, as mentioned before, in our case we assume that historical data is unavailable, and make use of online classifiers that can build and update the BM at run time. The accuracy of an online classifier increases with the number of predictions, as the statistical model converges to an optimal representation of the data distribution [40]. The disadvantage of using online classifiers is that, for a period of time at the beginning, the predictions can be less accurate. The advantages are that no previous labeled data is needed, and that the BM can easily adapt to changes in the definition of normal behavior. This is relevant as a precise definition of normal and anomalous behavior of a system is hard or even impossible to obtain before the actual execution of the system. In the following we describe the technical details of the *Behavior Identifier*.

3.3.1 Support Vector Machines

The *Behavior Identifier* employs a type of classifier called *Support Vector Machines* (SVMs), which are a widely used mechanism for classifying data [29], [41]. Having a set of labeled vectors $L = \{(x_i, y_i) \mid 1 \leq i \leq l\}$, where $x_i \in \mathbb{R}^n$, and $y_i \in \{-1, +1\}$ represents two distinct categories, the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2821693, IEEE Transactions on Dependable and Secure Computing

training phase in an SVM consists of finding the optimal hyperplane that separates the two categories in an n-dimensional space. Finding this hyperplane requires minimizing $||w||^2$, where w is a vector normal to the hyperplane. This minimization can be expressed as the optimization problem

$$\begin{aligned} &\sum_{i=1}^{l} \alpha_i - \frac{1}{2} \sum_{i=1}^{l} \sum_{j=1}^{l} \alpha_i \alpha_j y_i y_j x_i \cdot x_j \\ &\text{subject to} \quad \alpha_i \geq 0 \quad \text{and} \quad \sum_{i=1}^{l} \alpha_i y_i = 0 \,, \end{aligned} \tag{1}$$

where α_i and α_j are Lagrange multipliers.

After solving the optimization problem, unlabeled vectors can be categorized according to their position in the n-dimensional space with respect to the hyperplane. The decision function for an unlabeled vector x_j becomes

$$f(x_j) = \operatorname{sgn}\left(\sum_{i=1}^{l} \alpha_i y_i x_i \cdot x_j - b \right) . \tag{2}$$

For not linearly separable categories in the n-dimensional space, a kernel trick can be used by replacing the product $x_i \cdot x_j$ in Eq. (1) by a kernel function $K(x_i, x_j) \equiv \phi(x_i) \cdot \phi(x_j)$, where $\phi : \mathbb{R}^n \to \mathbb{R}^m$ represents a transformation from the n-dimensional space to an m-dimensional space. This kernel function enables building the hyperplane in the m-dimensional space, where the two categories can be separated. A widely used kernel function is the *radial basis function* (RBF) kernel, which is defined as [42]

$$K(x,\ x') = \mathrm{e}^{-\gamma \|x - x'\|^2} \,, \tag{3}$$

where γ is a free parameter and $\|x - x'\|^2$ is the *squared Euclidean distance* between x and x'.

3.3.2 Gradient Descent

Solving Eq. (1) given a training dataset yields the optimal hyperplane or decision function. In our case, we are interested in building and updating this decision function at run time. To build an online support vector machine that modifies its decision function whenever new observations become available, the *gradient descent* method can be used. Gradient descent is an algorithm for finding the minimum of a function in an iterative manner. We employ a modified version of the SVM proposed by Kivinen et al. [43], where the hyperplane is obtained by minimizing the soft margin loss function

$$\ell_\rho(f(x),\ y) = \max(0,\ \rho - y f(x)) \,. \tag{4}$$

where $\rho \geq 0$ is a margin parameter that controls the amount of margin errors permitted. In this manner, we iteratively update the α coefficients using

$$\alpha_i = \begin{cases} -\eta \ell'(f_j(x_i),\ y_i) & \text{for} \quad i = j \\ (1 - \eta\lambda)\alpha_i & \text{for} \quad i < j \,, \end{cases} \tag{5}$$

which builds a classifier able to differentiate between two classes (i.e., a binary classifier). To build a one-class classifier, we employ the loss function

$$\ell_\rho(f(x)) = \max(0,\ \rho - f(x)) - \nu\rho \,, \tag{6}$$

where ρ defines the size of the decision function, and ν controls the frequency of anomalous classifications. In this case, the update rules are:

$$(\alpha_i, \alpha_j, \rho) = \begin{cases} ((1-\eta)\alpha_i,\ \eta\ ,\ \rho - \eta(1-\nu)) & \text{if } f(x) < \rho \\ ((1-\eta)\alpha_i,\ 0\ ,\ \rho + \eta\nu) & \text{otherwise} \,. \end{cases} \tag{7}$$

3.3.3 Data Normalization

Normalizing the vectors before classification is crucial to avoid bias towards a particular dimension [44] because SVMs rely on dot products between vectors. For example, the dot product between $x = [0.3,\ 100]$ and $x' = [0.1,\ 200]$ is biased towards the second dimension as it takes much larger values. For this reason, the *Behavior Identifier* normalizes each vector component to the $[0, 1]$ interval using a sigmoidal function. This type of function has the advantage that it is robust to the presence of extreme values in the data. The function is defined as

$$x'_i = \frac{1}{1 + \mathrm{e}^{\frac{x_i - \mu_i}{\sigma_i}}} \quad \text{for} \quad 1 \leq i \leq n \,, \tag{8}$$

where subindex i in this case denotes the ith component of a vector, x is the original vector, x' is the scaled vector, and μ and σ are the mean and standard deviation of the set of seen vectors. Since the *Behavior Identifier* receives vectors one at a time, μ and σ are computed also online. Initially, μ is set to the first received vector and $\sigma = [1, 1, ..., 1]$. Then, updates can be performed using

$$\begin{aligned} \mu'_i &= (1 - \frac{1}{t})\mu_i + \frac{1}{t} x_i \\ \sigma'_i &= (1 - \frac{1}{t})\sigma_i + \frac{1}{t}(x_i - \mu'_i)^2 \,, \end{aligned} \tag{9}$$

where x is the last received vector and t is the number of vectors processed so far.

3.3.4 Two-step Classification

The *Behavior Identifier* makes use of a combination of a one-class classifier and multiple binary classifiers to achieve two main goals: the one-class classifier is used to detect previously unseen anomalies without historical failure data, and a set of binary classifiers is used to identify the type of known anomalies. Algorithm 1 presents the pseudocode of the one-class classifier. Classes normal and anomalous are represented with -1 and $+1$ respectively.

The classifier contains two main methods: CLASSIFY and UPDATE_MODEL. CLASSIFY returns the predicted label (lines 7 and 9) of an unlabeled vector x_j (line 6). UPDATE_MODEL modifies the current decision function given a vector x_j of the normal class (-1). If the kernel evaluation is less than ρ (line 14), meaning that the current function misclassifies x_j, the vector is incorporated into the set of support vectors V (line 15) with $\alpha_j = \eta$ (line 12), and ρ is decreased (line 17). Here, instead of using $\rho = \rho - \eta(1-\nu)$ as defined in Eq. (7), we use $\rho = \rho - \rho\eta(1-\nu)$ to enforce

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2821693, IEEE Transactions on Dependable and Secure Computing

Algorithm 1 Online One-class Classifier

```
1: function INIT
2:     V ← ∅                                   ▷ Set of support vectors
3: end function
4:
5: function CLASSIFY(x_j: Vector)
6:     if ρ < Σ_{x_i∈V} α_i K(x_i, x_j) then
7:         return -1
8:     else
9:         return +1
10:    end if
11: end function
12:
13: function UPDATE_MODEL(x_j: Vector)
14:    if Σ_{x_i∈V} α_i K(x_i, x_j) < ρ then
15:        V ← V ∪ {x_j}
16:        α_j ← η
17:        ρ ← ρ − ρη(1 − ν)
18:    else
19:        ρ ← ρ + ρην
20:    end if
21:    for all α_i ≠ α_j do
22:        α_i ← (1 − η)α_i
23:    end for
24: end function
```

Algorithm 2 Online Binary Classifier

```
1: function INIT
2:     V ← ∅                                   ▷ Set of support vectors
3: end function
4:
5: function CLASSIFY(x_j: Vector)
6:     if Σ_{x_i∈V} α_i K(x_i, x_j) < 0 then
7:         return -1
8:     else
9:         return +1
10:    end if
11: end function
12:
13: function UPDATE_MODEL(x_j: Vector, y_j: int)
14:    if y_j Σ_{x_i∈V} α_i K(x_i, x_j) ≤ ρ then
15:        α_j ← ηy_j
16:        V ← V ∪ {(x_j, y_j)}
17:    end if
18:    for all α_i ≠ α_j do
19:        α_i ← (1 − λη)α_i
20:    end for
21: end function
```

$\rho > 0$, given that $\eta < 1$ and $\nu < 1$. Alternatively, ρ is increased if the current function correctly classifies x_j (line 19). This makes ρ to approach its optimal value close to the kernel evaluation, ensuring that the size of the sphere around the normal data is minimal. Finally, the α values are also updated (line 22).

Algorithm 2 shows the pseudocode of the binary classifier, which is very similar to the one-class classifier. The CLASSIFY method uses the decision function to label vectors (line 6), and the UPDATE_MODEL now makes use of y_j to decide whether to update the decision function based on the soft margin loss in Eq. (4) (line 14). If the current function misclassifies x_j, the vector is added to V (line 16) and $\alpha_j = \eta y_j$ according to Eq. (5) (line 15). Finally, the rest of the α coefficients are updated (line 19).

The binary classifier is designed to receive a similar amount of positive and negative vectors over time. Given an unbalanced dataset, the classifier is biased towards the most common class. This is obvious when $|V| = 1$, since the kernel evaluation in the CLASSIFY method (line 6) will always yield the sign of the only support vector in the set. Moreover, successive calls to the UPDATE_MODEL method with vectors of the same type will produce no actual changes in the model, as the update condition (line 14) will be false with high probability after a few updates, and new vectors of the same type will not be included into V. To avoid a biased model in a scenario where most of the vectors are negative because distributed systems run most of the time in normal conditions, we update the binary classifier in a balanced manner. Algorithm 3 shows the pseudocode of the balanced update process.

Algorithm 3 Balanced Update Process

```
1: function INIT
2:     P ← ∅                                   ▷ Set of positive vectors
3:     N ← ∅                                   ▷ Set of negative vectors
4: end function
5:
6: function BALANCED_UPDATE(x_j: Vector, y_j: int)
7:     if |V| = 0 then
8:         UPDATE_MODEL(x_j, y_j)
9:     else if y_j < 0 then
10:        if |P| > 0 then
11:            UPDATE_MODEL(x_j, y_j)
12:            x_p ← GET_RANDOM(P)
13:            UPDATE_MODEL(x_p, +1)
14:        else
15:            N ← N ∪ x_j
16:        end if
17:    else
18:        if |N| > 0 then
19:            UPDATE_MODEL(x_j, y_j)
20:            x_n ← GET_RANDOM(N)
21:            UPDATE_MODEL(x_n, −1)
22:        else
23:            P ← P ∪ x_j
24:        end if
25:    end if
26: end function
```

If the set of support vectors is empty (line 7), the model is updated normally (line 8). Otherwise, if x_j is negative and there are positive vectors stored in P (lines 9 and 10), the model is updated with x_j (line 11) and a random vector from P (lines 12 and 13). If x_j is negative but P is empty (line 14), instead of updating the model, x_j is stored in N (line 15). The process when x_j is positive is the same but swapping N and P (lines 18 to 24). This avoids successive updates with vectors of the same type by storing them to be

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2821693, IEEE Transactions on Dependable and Secure Computing

used in the future, and ensures that the model is updated in a balanced manner.

As mentioned before, the *Behavior Identifier* combines several classifiers to detect previously unseen anomalies and to identify the type of known anomalies. We call this two-step classification because *Behavior Identifier* decides whether a vector is anomalous in a first step and, in case it is, the *Behavior Identifier* identifies its type in a second step. This two-step process is done by means of a one-class classifier (OC), a binary classifier (BC), and a multi-class classifier (MC). The MC is in turn composed of multiple binary classifiers organized in a tree-like *directed acyclic graph* (DAG) [45].

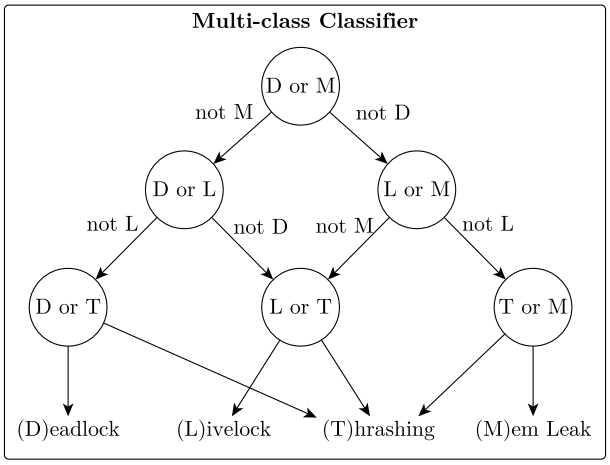


Fig. 3: DAG of binary classifiers to form a multi-class classifier.

Fig. 3 shows an MC able to identify four types of anomalies: deadlock, livelock, thrashing, and memory leak. Each node in the graph represents a binary classifier, and each of these binary classifiers discards one type of anomaly until the final result is obtained. For example, the node at the top of the graph decides if a vector is a memory leak or a deadlock. The classification process continues to the left if the vector *is not* a memory leak, and to the right if the vector *is not* a deadlock. When a new type of anomaly is found, the DAG is extended with a new level. There are $c-1$ levels given c types of anomalies, and each level has one extra node than the previous level. Thus, the total number of nodes for c types of anomalies is $\frac{c(c-1)}{2}$, and the number of classifications needed to obtain the final result is $c-1$. Each binary classifier in the MC maintains its own decision function, updating it online when new data becomes available. Given an anomalous vector of type y_a, the MC updates the decision functions of the nodes that compare y_a against other types. This means a total of $c-1$ updates.

Fig. 4 depicts the two-step process that combines the OC, BC, and the MC. In the first step, the OC and the BC classify vectors as normal or anomalous. The OC serves to detect previously unseen anomalies while the BC is used to reinforce the detection of already known anomalies. The outputs of the OC and the BC are combined to produce a final prediction for step one. A simple approach is to combine the outputs using a logical *or*, and more advanced techniques could use weights based on the reliability of each classifier. The process finishes if the vector is classified as normal in this first step. If the vector is anomalous, the process moves on to step two, where the MC identifies the type of the anomaly. The total number of evaluations is therefore 2 for normal vectors and $c+1$ for anomalous vectors.

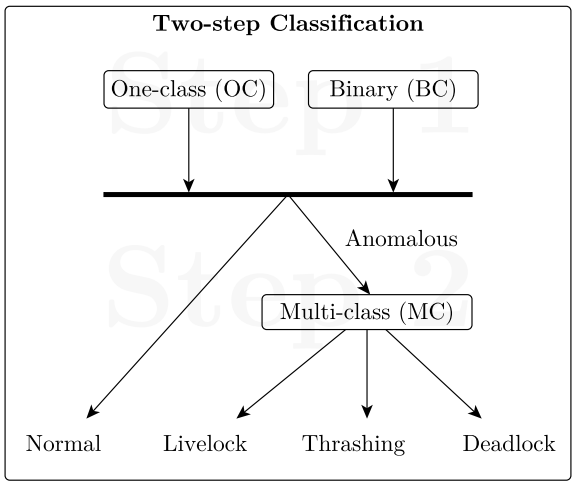


Fig. 4: Two-step classification process. Multi-class classification is provided by means of a directed acyclic graph of binary classifiers.

3.3.5 Model Update

The *Behavior Identifier* transmits the result of the two-step classification to the *Feedback Provider*. The *Feedback Provider* then decides whether to alert the system administrator. If the system administrator is alerted and feedback is received, the *Behavior Identifier* updates the *Behavioral Model*. The BM is defined by the decision functions of the multiple classifiers that are involved in the two-step classification process. Given a vector x_j with known type y_j, the OC's decision function is updated if $y_j = -1$, (i.e., x_j is normal), and the MC is updated if x_j is anomalous. The BC is updated in any case as it is used to differentiate between normal vectors and known anomalies. Each of the classifiers normalizes x_j before updating the decision function, and also updates the normalizing function using Eq. (9). Note that each of the classifiers maintains its own normalizing function.

3.4 Feedback Provider

The *Feedback Provider* decides when to alert the system administrator based on the result of the two-step classification process. The *Feedback Provider* behavior is modeled as a *finite state machine* (FSM), which is shown in Fig. 5.

In this FSM, the $pred$ variable represents predictions received from the *Behavior Identifier*, the fb variable represents feedback received from the system administrator, and the $norm$ and $anom$ variables store the number of consecutive normal and anomalous predictions respectively. A negative value in these variables means normal behavior, whereas positive values represent different types of anomalies. The *Feedback Provider* starts in the *Train* state. This state is completely optional and is used to train BARCA if some

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2821693, IEEE Transactions on Dependable and Secure Computing

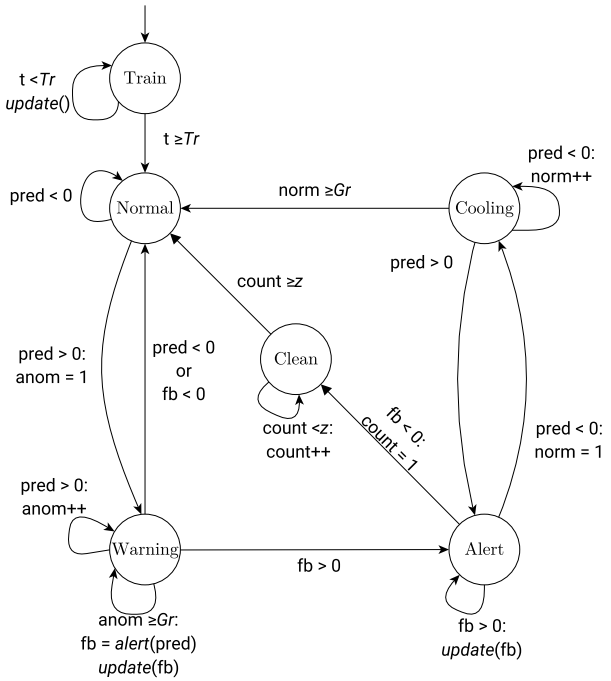


Fig. 5: *Feedback Provider* behavior modeled as a FSM.

labeled data is available, or if it can be guaranteed that the system will run normally for a certain amount of time at the beginning. The parameter Tr controls the duration of this initial training phase, and the state is avoided if $Tr = 0$. After this, the *Feedback Provider* moves on to the *Normal* state, and remains in it as long as the predictions received from the *Behavior Identifier* are normal. The *Feedback Provider* moves to the *Warning* state if it receives an anomalous prediction. The *Feedback Provider* remains in the *Warning* state as long as the number of consecutive anomalous predictions (variable $anom$) is less than a predefined number, represented by the parameter Gr. If a normal prediction is received, the *Feedback Provider* goes back to the *Normal* state. The *Feedback Provider* raises an alert if it receives more than Gr consecutive anomalous predictions while in the *Warning* state. In this manner, Gr helps to reduce the number of false alerts caused by noisy readings. When an alert is raised, the *Feedback Provider* requests feedback to the system administrator. The $update()$ method transmits this feedback to the *Behavior Identifier*, which can then update the *Behavioral Model* accordingly. The *Feedback Provider* moves back to the *Normal* state if the actual behavior of the system is normal (i.e., $fb < 0$). Otherwise, the *Feedback Provider* moves to the the *Alert* state. The *Feedback Provider* remains in the *Alert* state as long as the system administrator confirms the presence of the anomaly. If no feedback is given, the *Feedback Provider* returns to the *Normal* state through the *Cooling* state if Gr consecutive normal predictions are received. Conversely, if the system administrator confirms the end of the anomaly, the *Feedback Provider* moves to the *Clean* state. The *Feedback Provider* remains in this state for z observations (i.e., the sliding window size), to ensure that no anomalous readings remain in the BIs once the *Feedback Provider* returns to the *Normal* state.

4 Experimental Analysis

In this section we evaluate BARCA's ability to detect and identify various types of errors that generate collective and correlation anomalies, as described in Section 3.1. Towards this, we employ data from two sources: our in-house synthetic distributed system (SDS) [46], and the Yahoo! Webscope labeled anomaly detection dataset [47].

4.1 Synthetic Distributed System

The SDS consists of a distributed system where multiple Clients share access to several Databases. Clients can freely read information from the Databases, but write access is granted using a distributed mutual exclusion protocol [48]. Thus, access to each Database is controlled by a subset of Clients. Clients wanting to write to a Database must request votes to the Database controllers, and can only perform the write after achieving a majority of their votes.

The behavior of the SDS is complex enough to allow the introduction of several anomalous behaviors. Specifically, we model four types of anomalies: deadlock, livelock, unwanted synchronization, and memory leak. In deadlock mode Client votes are uniformly distributed among all Clients, and the SDS stalls due to a lack of consensus. In livelock mode the SDS progress stalls because Clients continuously change their vote. Unwanted synchronization occurs when Clients continuously access the same Databases in the same order, and memory leak is generated by forcing a particular Client to allocate an increasing amount of memory over time. Anomalies can be activated and deactivated at run time by modifying a variable, allowing the SDS to resume normal operation after a period of time.

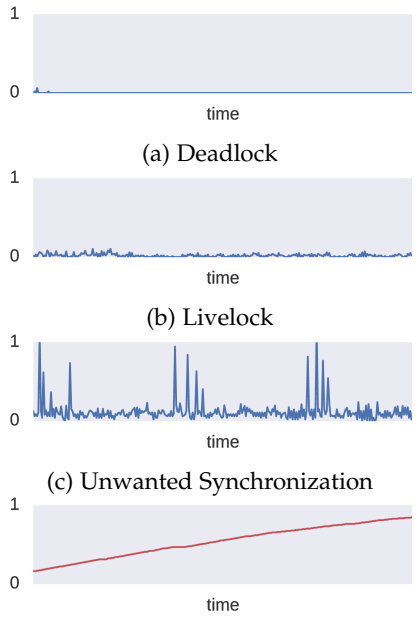
We run the SDS on our testbed, consisting of five nodes, each composed of two Intel Xeon Dual Core 2.33GHz processors, 4GB of memory, and 73GB of disk space, running CentOS 6.5. We collect 60 hardware metrics every second, including CPU, network, memory and disk utilization of each node. Figure 6 shows how the different anomalies affect these system metrics. Deadlock and livelock cause a downward shift in network utilization. Unwanted synchronization generates evenly spaced spikes in network utilization, and the memory leak creates an upward trend in memory utilization. Deadlock, livelock, and unwanted synchronization are system-wide anomalies that have an impact on 10 of the 60 metrics, whereas memory leak only affects one metric.

We employ two datasets with data from the SDS[1]. The first dataset, referred to as SDS-Dataset-1, consists of SDS monitoring data from 10 runs per anomaly type of 100 minutes long, where the same 10 minute anomaly is injected twice (at minutes 30 and 80).

The second dataset, referred to as SDS-Dataset-2, consists of monitoring data from several SDS runs with four anomalies per run. We run the SDS for 200 minutes and inject four 10 minute anomalies at minutes 20, 70, 120, and 170. Let $t1$, $t2$, $t3$, and $t4$ be the type of the four anomalies, the first two anomalies are of different types in all runs (i.e., $t1 \neq t2$). Then, we perform 10 runs where $t1 = t3$ and $t2 = t4$, and 10 runs where $t1 = t4$ and $t2 = t3$. We do this

1. Available at https://github.com/javicid/anomalies.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2821693, IEEE Transactions on Dependable and Secure Computing



(a) Deadlock

(b) Livelock

(c) Unwanted Synchronization

(d) Memory Leak

Fig. 6: SDS metrics under different anomalous behaviors. (a), (b) and (c) represent network utilization, while (d) represents memory utilization. Values are normalized.

for every combination of anomalies, obtaining 120 runs in total. For example, given deadlock and livelock, we perform 10 runs where the order of the anomalies is deadlock-livelock-deadlock-livelock; and 10 runs where the order of the anomalies is deadlock-livelock-livelock-deadlock.

Both SDS datasets include monitoring data from the 60 performance metrics collected. However, the SDS-Dataset-1 does not include network utilization in the case of memory leak as the high variance in this metric produces biased results.

4.2 Yahoo! Dataset

The Yahoo! Webscope labeled anomaly detection dataset [47] is composed of real and synthetic time series. The real time series come from several Yahoo! services, whereas the synthetic data consists of 100 time series with varying trend, noise, and seasonality. We run our experiments using the synthetic data because the real time series do not contain collective anomalies.

Each of the 100 synthetic time series is composed of 1,681 data points, and have anomalies at random timestamps. We split these time series in smaller sections, and group these sections together to simulate the behavior of a distributed system from which several performance metrics are collected. We generate 1,320 groups of 65 time series of 150 data points long. Within each group, a random number (from 1 to 25) of time series contain an anomaly from timestamp 100 to the end of the time series. Each group of time series can be seen as the behavior of a system from which 65 performance metrics are collected. The first 100 data points are considered the normal behavior of the system, and the remaining 50 data points are considered anomalous. In the rest of this section, we will refer to this dataset as Yahoo-Dataset.

4.3 Evaluation Metrics

BARCA's objective is to alert the system administrator as soon as possible after anomalies appear, but also to produce the minimum possible number of false alarms, as inspecting them is very time consuming. The typical evaluation metrics employed in anomaly detection [49], which are true positive rate (TPR) and false positive rate (FPR) with respect to the vector classifications are not very informative in our case. For example, in a run of 50 minutes long from which 20 seconds are anomalous and 2980 seconds are normal, generating 10 false positives yields a FPR of $\frac{2970}{2980} \approx 0.003$, which could be interpreted as a good result. However, investigating 10 false alarms in 50 minutes is unacceptable in real world scenarios. Thus, we employ TPR (or $Recall$) and $Precision$, and define them with respect to the number of anomalies instead of the classified vectors, that is,

$$Recall = \frac{\text{detected anomalies}}{\text{total number of anomalies}}$$
$$Precision = \frac{\text{detected anomalies}}{\text{total number of alerts}}, \tag{10}$$

where we consider an anomaly as detected if at least one alert is raised while the anomaly is active. $Recall$ evaluates the ability of BARCA to detect anomalies, whereas $Precision$ evaluates the absence of false positives. Both metrics range from 0 to 1, being 1 the optimal value. Note that multiple alerts during the same anomaly affect $Precision$ negatively. In this manner we penalize random alerts and ensure that the anomaly is being truly detected. Additionally, in some experiments we also provide the detection time (DT), which is the time from the appearance of an anomaly until it is detected, and the F-score (F_1), which is the harmonic mean of $Recall$ and $Precision$:

$$F_1 = 2 \cdot \frac{Recall \cdot Precision}{Recall + Precision}. \tag{11}$$

Apart from these metrics, we redefine $Recall$ and $Precision$ in respect to the number of correct identifications to evaluate BARCA's performance in identifying the anomaly types. That is, given an anomaly of type t, we define

$$Recall_t = \frac{\text{times } t \text{ is correctly identified}}{\text{times } t \text{ is detected}}$$
$$Precision_t = \frac{\text{times } t \text{ is correctly identified}}{\text{times an anomaly is identified as } t}. \tag{12}$$

These two metrics and the corresponding F-score (i.e., their harmonic mean) give an idea of how well an anomaly type is identified, and how often an anomaly type is confused with others.

4.4 Features

As described in Section 3.2, the *Behavior Extractor* builds feature vectors from the system metrics seen as time series. There is a large amount of features that can be extracted from time series [50]. For example, statistical features such as mean or standard deviation represent certain properties of the probability distribution of the values in the series. In

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2821693, IEEE Transactions on Dependable and Secure Computing

addition, applying certain transformations to a time series can also be used to obtain new knowledge. For example, the autocorrelation function [51] reveals periodic patterns.

Other techniques that can be used to extract features from time series are frequency domain transformations such as the discrete Fourier transform (DFT) or the discrete wavelet transform (DWT) [52]; regression models [53]; time series analysis models such as the autoregressive integrated moving average model (ARIMA) [54], [51]; and chaotic analysis functions such as the Lyapunov exponent [55].

In our experiments, we analyze BARCA's performance using 17 features, summarized in Table 1, in the detection and identification of the different anomalies in the SDS and Yahoo! datasets. FH and FL are the 25% higher and lower frequencies of the discrete Fourier transform respectively, the linear regression model is obtained by fitting a model of the form $y = \beta_0 + \beta_1 x + \epsilon$ to the time series [53], and the Lyapunov exponent is a measure of how chaotic a time series is [55].

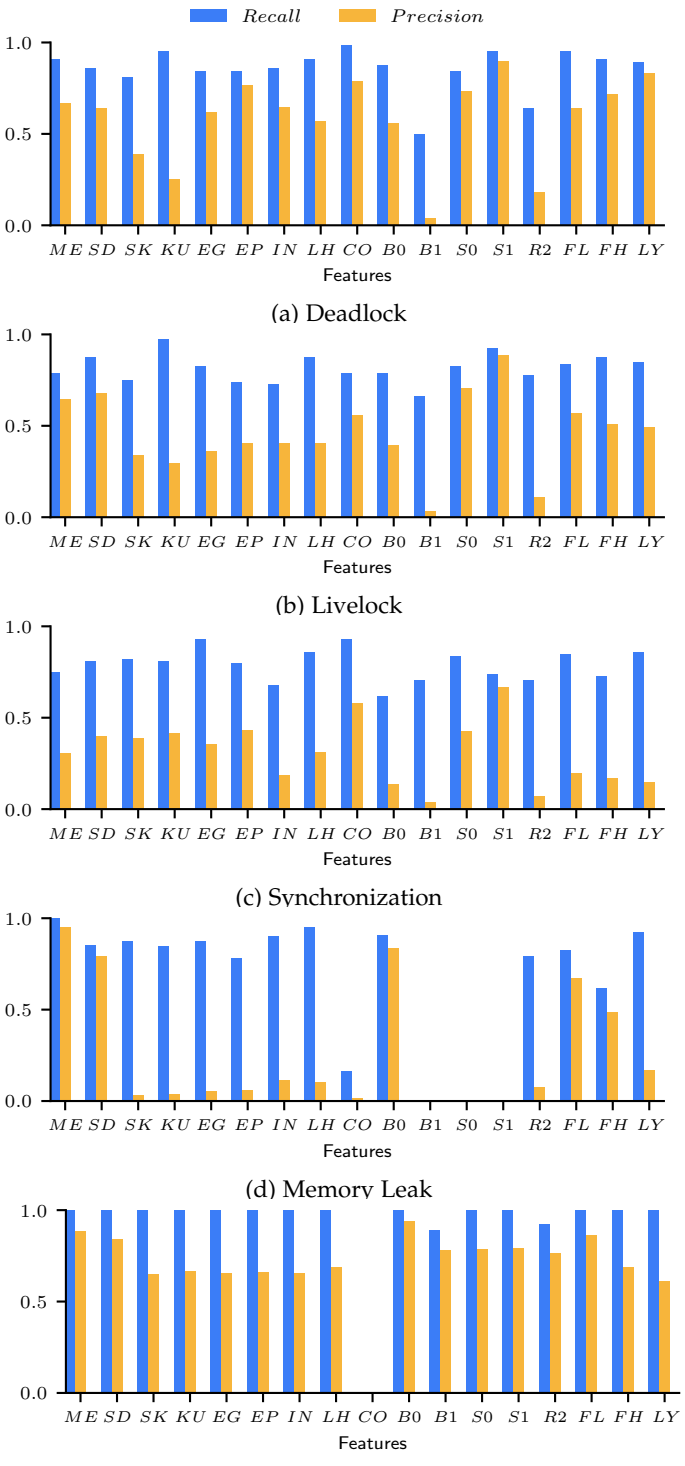
TABLE 1: Features used in the experiments.

	Feature	Code
First order	Mean	ME
	Standard Deviation	SD
	Skewness	SK
	Kurtosis	KU
Second order [38]	Energy	EG
	Entropy	EP
	Correlation	CO
	Inertia	IN
	Local Homogeneity	LH
Fourier transform	Highest frequencies	FH
	Lowest frequencies	FL
Chaos theory	Lyapunov exponent [55]	LY
Linear regression [53]	β_0 (y-intercept)	B0
	β_1 (slope)	B1
	Std. error of β_0	S0
	Std. error of β_1	S1
	Coefficient of determination	R2

4.5 Anomaly Detection

In a first set of experiments, we evaluate BARCA's ability to detect previously unseen anomalies. Fig. 7 shows mean $Recall$ and $Precision$ obtained in the SDS-Dataset-1 and the Yahoo-Dataset when employing different features to build the feature vectors. With the SDS data we use $Gr = 20$, $Tr = 400$, and windows of 32, 64, 128 and 256 seconds (see Section 3). With the Yahoo! data we employ $Gr = 10$, $Tr = 50$, and windows of 8, 16 and 32 data points. Results are averaged over all window sizes to get a better idea on how well different features perform regardless of the window employed.

Deadlock is best detected using $S1$ and LY. $S1$ obtains a $Recall$ and $Precision$ of 0.95 and 0.90, whereas LY obtains 0.89 and 0.83. In the case of livelock, also $S1$ obtains the best results with 0.92 $Recall$ and 0.89 $Precision$. For synchronization, CO works best with 0.93 $Recall$ and 0.58 $Precision$. The memory leak is best detected with ME and $B0$. ME obtains 1.00 $Recall$ and 0.95 $Precision$, and $B0$ obtains 0.91 $Recall$ and 0.83 $Precision$.



(a) Deadlock

(b) Livelock

(c) Synchronization

(d) Memory Leak

(e) Yahoo!

Fig. 7: Mean $Recall$ and $Precision$ obtained with the different individual features.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2821693, IEEE Transactions on Dependable and Secure Computing

In the case of the Yahoo! anomalies, also ME and $B0$ obtain the highest performance. ME achieves 1.00 $Recall$ and 0.88 $Precision$, and $B0$ achieves 1.00 $Recall$ and 0.94 $Precision$. $S1$ and CO are in general better at dealing with high variance in the system metrics, like network utilization in the deadlock, livelock, and synchronization cases. Conversely, $B0$ and ME work better in the cases where there is not such a great variance in the metrics, like in the memory leak and Yahoo! anomalies.

The features that obtain the highest overall performance are ME and SD. This is because these two features represent the underlying distribution of values in the system metrics very well. Regarding FL and FH, both features obtain similar results, with FL being slightly superior in some cases. Apart from CO, second order features like EG, IN or LH do not achieve good performance, and the worst overall features are $B1$ and $R2$. It is clear that there is a trade-off between detection accuracy and the types of anomalies that can be detected. For example, livelock can be detected with very high accuracy using $S1$, but ME and SD provide better detection across different types of anomalies.

We are also interested in knowing if a combination of multiple features can improve detection accuracy across different types of anomalies. Since evaluating every feature subset is unfeasible, we perform a simulated annealing search [56], and find that the set of features with the best overall performance is $S_1 = \{CO, ME, B0, S1\}$. Table 2 shows the mean F-score obtained for this set compared to the F-score of the best individual feature (ME). The set S_1 sacrifices some performance in detecting the memory leak to achieve an overall mean improvement of 15% over ME. While features $S1$ and CO reinforce each other in the detection of deadlock, livelock and synchronization, ME and $B0$ enable the detection of the memory leak.

TABLE 2: Mean detection F-score of the set of features $S_1 = \{CO, ME, B0, S1\}$ versus ME.

Anomaly	ME	S_1	Improvement
Deadlock	0.74	0.91	22%
Livelock	0.69	0.79	14%
Synchronization	0.41	0.65	58%
Mem. Leak	0.97	0.81	-16%
Yahoo!	0.92	0.90	-2%
Mean	0.75	0.81	15%

Finally, we test the set $\{CO, ME, B0, S1\}$ in SDS-Dataset-2. Table 3 shows $Recall$, $Precision$ and detection time averaged over 120 runs with different anomalies. $Precision$ is computed overall since false positives cannot be associated with a particular anomaly in a run containing anomalies of different types. We use $Tr = 400$, $Gr = 20$, and a window of 128 seconds.

It can be seen how deadlock, livelock and synchronization are correctly detected with high accuracy. The drop in the detection of the memory leak is due to the introduction of network utilization in the analysis of this anomaly (which is not included in the SDS-Dataset-1). Network utilization has so much variance that it masks the trend in memory usage generated by the memory leak. A solution to this problem would be to have dedicated classifiers for certain system metrics, the use of different features with different metrics, or the use of a combination of multiple classifiers with different settings (e.g., several window sizes).

The slight drop in $Recall$ in synchronization when compared to previous experiments can be explained by the fact that the classifier parameters that enable the detection of one anomaly with high accuracy might not allow the detection of other anomalies. In other words, being able to detect multiple anomalies in the same run comes at the cost of overall accuracy. Even so, most anomalies are detected in less than 200 seconds (or observations) with high $Precision$ (0.86). This means that expert intervention is minimal as, on average, BARCA generated 0.33 false alerts per run, which is approximately 1 false alert every 10 hours.

4.6 Anomaly Identification

In a second set of experiments, we evaluate BARCA's ability to identify the type of known anomalies. In these experiments, we employ the SDS-Dataset-2 using the first 100 minutes of execution to train BARCA. Since the SDS-Dataset-2 contains four anomalies, this means that BARCA is trained with the first two anomalies, and then used to detect and identify the last two. We set $Gr = 20$, $Tr = 6000$, and $z = 128$ (see Section 3). Fig. 8 shows mean $Precision_t$ and $Recall_t$ per anomaly type using different features.

In this case, the results are much more consistent across the different types of behavior, with LH outperforming the rest of the features in all cases. It seems that second order features are more useful for identifying the type of the anomalies than for detecting them, whereas the opposite happens to the linear regression features. This can be caused by similarities across the different anomalous behaviors, such as low $S0$ and $S1$, that make them indistinguishable when using these features.

Next, we run a simulated annealing search to know if the identification performance can be improved by combining multiple features. We find that the optimal set is $S_2 = \{LH, IN\}$, with a slightly better performance than LH. Other sets found in the simulated annealing process, such as $S_3 = \{LH, IN, EP\}$, can also outperform LH in certain cases, but using more than these three features always reduces performance. This is because LH outperforms the rest of the features in all cases. Including IN or EP can reinforce the identification of anomalies to a certain degree, but adding more features beyond that only hinders performance. Table 4 shows the a comparison of the the F-score obtained by LH, S_2, and S_3.

Finally, Table 5 shows the confusion matrix obtained when using S_2 as the feature set. Given two anomalies of types t and t', the element $C_{tt'}$ of the confusion matrix C is the ratio between the number of times that t has been classified as t' and the number of times that t has been detected. In a perfect scenario, $C_{tt'}$ would be 1 for $t = t'$ and 0 for $t \neq t'$.

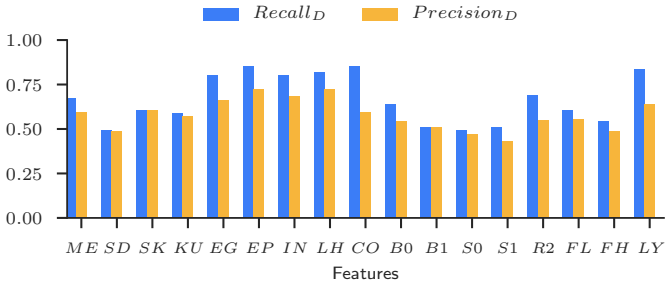
4.7 Discussion

Our experiments show that BARCA is able to detect and identify several types of anomalous behaviors that generate collective and correlation anomalies in the system performance metrics. With the adequate choice of features,

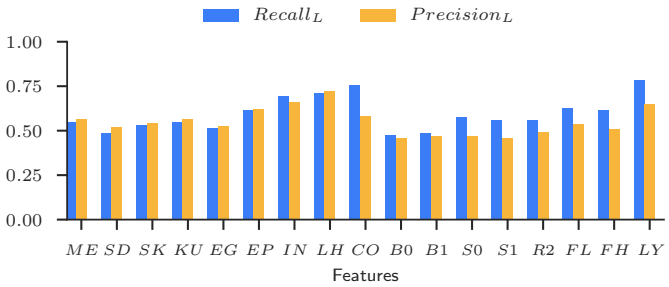
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2821693, IEEE Transactions on Dependable and Secure Computing

TABLE 3: Average $Recall$, $Precision$ and DT using the set of features $\{CO, ME, B0, S1\}$ in SDS-Dataset-2 ($Tr = 400$, $Gr = 20$, $z = 128$).

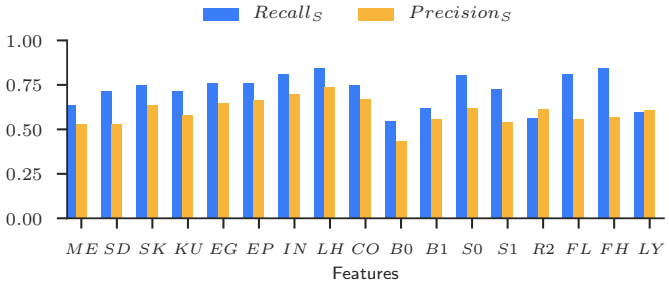
	Recall		*Precision*		DT (s)	
	Mean	Std. Dev.	Mean	Std. Dev.	Mean	Std. Dev.
Deadlock	0.99	0.11	0.86	0.19	208	54
Livelock	0.92	0.27			215	59
Synchronization	0.62	0.48			158	43
Memory Leak	0.02	0.15			291	3



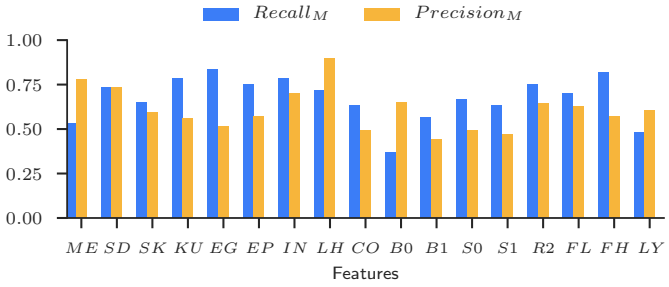
(a) Deadlock



(b) Livelock



(c) Synchronization



(d) Memory Leak

Fig. 8: Mean $Recall_t$ and $Precision_t$ obtained with the different individual features.

TABLE 4: Mean identification F-score of the set of features $S_2 = \{LH, IN\}$ versus $S_3 = \{LH, IN, EP\}$ and LH.

Anomaly	LH	S_3	S_2	Improvement
Deadlock	0.77	0.80	0.82	6.1%
Livelock	0.71	0.75	0.75	4.4%
Synchronization	0.79	0.76	0.76	-3.7%
Mem. Leak	0.80	0.76	0.79	-1.3%
Mean	0.77	0.77	0.78	1.4%

TABLE 5: Confusion matrix obtained using the feature set $\{LH, IN\}$ and a window of 128 seconds.

	D	L	S	M
Deadlock	**0.77**	0.09	0.09	0.06
Livelock	0.07	**0.69**	0.22	0.02
Synch.	0.04	0.02	**0.88**	0.05
Mem. Leak	0.06	0.00	0.06	**0.88**

BARCA can detect anomalies that represent as little as $\sim 2\%$ of the feature vector, as shown in the case of the memory leak in SDS-Dataset-1 (Section 4.5). The results presented in this paper cannot be directly compared to the results reported by other works because BARCA implements a much broader approach that is not tailored to a specific system or anomaly type (see Section 2). Nevertheless, the experiments validate our adaptive anomaly identification framework, and prove that complex anomalies can be detected without historical data, and without prior system information, by building a model of the normal behavior of the system that includes the evolution of the system over time and takes into account metric correlations. BARCA is the only framework that tackles the detection and identification of anomalous behaviors from a generic perspective, that can be employed in numerous scenarios without previous system knowledge.

We have seen that, in our behavior characterization process, an adequate selection of features is essential, and that a combination of various features is typically better than using a single one. One of the best features for anomaly detection in many cases is $B0$. The reason for this can be that the y-intercept serves a as measure of trend or the distribution of values depending on the case. Nevertheless, different features capture different anomalies, and even though combining multiple features improves overall performance, it comes at the cost of some loss in the detection of certain anomalies. Moreover, features that are useful to detect anomalies differ from features that are useful to identify their type. The use of multiple behavior representations,

employing different sets of features, can improve overall detection and identification rates in scenarios with many anomalies types.

The poor results obtained in the memory leak case (see Table 3) suggest that high variance in certain system metrics can negatively affect the accuracy in detecting anomalies that impact on other metrics. This is because the classifiers employed are biased towards dimensions with high variance. Solutions to this problem can be the use of other types of classifiers, applying noise reduction techniques to smooth certain metrics, or employing multiple behavioral representations that analyze different system metrics.

The major experimental findings that we obtain are: i) complex anomalous behaviors can be detected without prior system knowledge by analyzing the system performance metrics over a time period; ii) using more features when characterizing system behavior can help in the detection of a larger variety of anomalies with some accuracy loss; iii) CO, ME, $B0$, and $S1$ are the best features for anomaly detection, and LH, IN and other second order features work better for anomaly identification; and iv) including all system metrics in the same behavioral model might have a negative impact on accuracy when some metrics exhibit high variance.

As explained in Section 3, BARCA relies on user feedback to build its behavioral model. This is an advantage, as BARCA is employed without historical data and adapts dynamically to changes in system behavior. However, relying on user feedback can be a drawback in scenarios where feedback is difficult to obtain, contains mistakes, or cannot be provided in time. A way to reduce the impact of user feedback on BARCA's accuracy is to provide an initial training phase where the system is guaranteed to be running normally. With this training phase, the number of false alerts can be reduced to a minimum as shown in Section 4.5, where BARCA obtained 0.86 average $Precision$, and generated 0.33 false alerts per run on average.

Despite the promising results obtained, BARCA's accuracy relies on an appropriate choice of parameters, which can be difficult in certain scenarios. Through our experiments, we have seen that windows of 128 observations or more generally produce better results as longer windows act as a smoothing function on the system metrics. Even though Tr should be set as high as possible, we have obtained good detection results with $Tr \approx 400$ [2], our experiments have also shown that Gr values between 10 and 50 work best. A key issue remains the choice of classifier parameters (see Section 3.3), as this is a less intuitive decision. Thus, a mechanism to tune the classifier parameters at run time would be greatly beneficial for BARCA.

5 Conclusions

This paper presents BARCA, an adaptive framework for anomaly identification without historical data. Contrary to existing approaches, BARCA can detect previously unseen anomalies, analyzes system behavior over time taking into account metric correlations, and dynamically adapts to changes in system behavior. We have proven BARCA's ability to detect and identify anomalous behaviors using data from our in-house synthetic distributed system and a dataset from Yahoo!.

Our results show that BARCA is able to detect different types of anomalous behaviors with 0.65 mean $Recall$ and over 0.80 mean $Precision$, reaching over 0.90 F-score for certain anomaly types. BARCA requires minimal expert intervention as it generates 0.33 false alerts per run on average. Moreover, BARCA is also able to identify the type of complex anomalous behaviors with over 0.80 mean F-score when employing the adequate features to represent system behavior. We found that first order features and linear regression features are useful to detect previously unseen anomalies, whereas second order features are more useful to identify the type of known anomalies.

In the future, we will experiment with a real world dataset to further validate BARCA, while adding other improvements, such as using a different behavioral representation for detection and identification, combinations of multiple classifiers, or finding a practical way of choosing the optimal classifier parameters in an online setting.

2. Additional experiments can be found in https://github.com/javicid/anomalies/blob/master/parametric_study.pdf

References

[1] M. Castells, *The Rise of the Network Society*. Wiley, 2011.
[2] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design*, 4th ed. Addison-Wesley, 2005.
[3] L. Atzori, A. Iera, and G. Morabito, "The Internet of Things: A Survey," *Computer Networks*, vol. 54, no. 15, pp. 2787–2805, 2010.
[4] E. Ronen, C. OFlynn, A. Shamir, and A.-O. Weingarten, "IoT Goes Nuclear: Creating a ZigBee Chain Reaction," Cryptology ePrint Archive, 2016, http://eprint.iacr.org/2016/1047.
[5] "Google Search," http://www.google.com.
[6] "Facebook, Inc." http://www.facebook.com.
[7] "Amazon.com, Inc." http://www.amazon.com.
[8] B. Cohen, "The BitTorrent Protocol Specification," 2008.
[9] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The Second-Generation Onion Router," in *Proceedings of the 13th USENIX Security Symposium*, 2004.
[10] Amazon Elastic Compute Cloud, http://aws.amazon.com/ec2/.
[11] "The Cost of Downtime," http://blogs.gartner.com/andrew-lerner/2014/07/16/the-cost-of-downtime/.
[12] C. Cachin and M. Schunter, "A Cloud You Can Trust," *IEEE Spectrum*, vol. 48, no. 12, pp. 28–51, 2011.
[13] W. E. Wong, D. Vidroha, A. Surampudi, H. Kim, and M. F. Siok, "Recent Catastrophic Accidents: Investigating How Software Was Responsible," in *Proceedings of the 4th International Conference on Secure Software Integration and Reliability Improvement*, 2010, pp. 14–22.
[14] M. Farshchi, J.-G. Schneider, I. Weber, and J. Grundy, "Experience Report: Anomaly Detection of Cloud Application Operations Using Log and Cloud Metric Correlation Analysis," in *Proceedings of the 26th International Symposium on Software Reliability Engineering*, 2015, pp. 24–34.
[15] A. Avižienis, J. C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.
[16] T. Wang, J. Wei, W. Zhang, H. Zhong, and T. Huang, "Workload-Aware Anomaly Detection for Web Applications," *Journal of Systems and Software*, vol. 89, pp. 19–32, 2014.
[17] H. Malik, H. Hemmati, and A. E. Hassan, "Automatic Detection of Performance Deviations in the Load Testing of Large Scale Systems," in *Proceedings of the 35th International Conference on Software Engineering*, 2013, pp. 1012–1021.
[18] Z. Lan, Z. Zheng, and Y. Li, "Toward Automated Anomaly Identification in Large-Scale Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 2, pp. 174–187, 2010.
[19] I. Laguna, S. Mitra, F. A. Arshad, N. Theera-Ampornpunt, Z. Zhu, S. Bagchi, S. P. Midkiff, M. Kistler, and A. Gheith, "Automatic Problem Localization via Multi-Dimensional Metric Profiling," in *Proceedings of the 32nd International Symposium on Reliable Distributed Systems*, 2013, pp. 121–132.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2821693, IEEE Transactions on Dependable and Secure Computing

[20] Q. Guan, Z. Zhang, and S. Fu, "Proactive Failure Management by Integrated Unsupervised and Semi-Supervised Learning for Dependable Cloud Systems," in *Proceedings of the 6th International Conference on Availability, Reliability and Security*, 2011.

[21] Y. Tan, "Online Performance Anomaly Prediction and Prevention for Complex Distributed Systems," Ph.D. dissertation, North Carolina State University, 2012.

[22] O. Ibidunmoye, F. Hernández-Rodriguez, and E. Elmroth, "Performance Anomaly Detection and Bottleneck Identification," *ACM Computing Surveys*, vol. 48, no. 1, pp. 4:1–4:35, 2015.

[23] Q. Guan and S. Fu, "Adaptive Anomaly Identification by Exploring Metric Subspace in Cloud Computing Infrastructures," in *Proceedings of the 32nd International Symposium on Reliable Distributed Systems*, 2013, pp. 205–214.

[24] C. Wang, K. Viswanathan, L. Choudur, V. Talwar, W. Satterfield, and K. Schwan, "Statistical Techniques for Online Anomaly Detection in Data Centers," in *Proceedings of the 12th IFIP/IEEE International Symposium on Integrated Network Management*, 2011, pp. 385–392.

[25] B. Sharma, P. Jayachandran, A. Verma, and C. R. Das, "CloudPD: Problem Determination and Diagnosis in Shared Dynamic Clouds," in *Proceedings of the 43rd International Conference on Dependable Systems and Networks*, 2013, pp. 1–12.

[26] D. O'Shea, V. C. Emeakaroha, J. Pendlebury, N. Cafferkey, J. P. Morrison, and T. Lynn, "A Wavelet-inspired Anomaly Detection Framework for Cloud Platforms," in *Proceedings of the 6th International Conference on Cloud Computing and Services Science*, 2016, pp. 106–117.

[27] O. Ibidunmoye, A. R. Rezaie, and E. Elmroth, "Adaptive Anomaly Detection in Performance Metric Streams," *IEEE Transactions on Network and Service Management*, vol. PP, no. 99, pp. 1–1, 2017.

[28] L. Yu and Z. Lan, "A Scalable, Non-Parametric Method for Detecting Performance Anomaly in Large Scale Computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 7, pp. 1902–1914, 2016.

[29] C. Cortes and V. Vapnik, "Support-Vector Networks," *Machine Learning*, vol. 20, no. 3, pp. 273–297, 1995.

[30] D. Yao, X. Shu, L. Cheng, and S. J. Stolfo, *Anomaly Detection as a Service: Challenges, Advances, and Opportunities*. Morgan & Claypool, 2017.

[31] X. Shu, D. Yao, and N. Ramakrishnan, "Unearthing Stealthy Program Attacks Buried in Extremely Long Execution Paths," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 401–413.

[32] M. Du, F. Li, G. Zheng, and V. Srikumar, "DeepLog: Anomaly Detection and Diagnosis from System Logs Through Deep Learning," in *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1285–1298.

[33] X. Shu, D. Yao, and B. G. Ryder, "A Formal Framework for Program Anomaly Detection," in *Proceedings of the 18th International Symposium on Recent Advances in Intrusion Detection*, 2015, pp. 270–292.

[34] R. E. Kalman, "A New Approach to Linear Filtering and Prediction Problems," *Journal of Basic Engineering*, pp. 35–45, 1960.

[35] S. D. Gribble, "Robustness in Complex Systems," in *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, 2001, pp. 21–26.

[36] M. Gabel, K. Sato, D. Keren, and S. Matsuoka, "Latent Fault Detection with Unbalanced Workloads," Lawrence Livermore National Laboratory, Tech. Rep., 2014.

[37] J. C. Mogul, "Emergent (Mis)Behavior vs. Complex Software Systems," in *Proceedings of the 1st ACM SIGOPS European Conference on Computer Systems*, 2006, pp. 293–304.

[38] R. J. Alcock and Y. Manolopoulos, "Time-Series Similarity Queries Employing a Feature-Based Approach," in *Proceedings of the 7th Panhellenic Conference on Informatics*, 1999, pp. 27–29.

[39] M. Mohri, A. Rostamizadeh, and A. Talwalkar, *Foundations of Machine Learning*. MIT Press, 2012.

[40] S. Shalev-Shwartz, Y. Singer, N. Srebro, and A. Cotter, "Pegasos: Primal Estimated Sub-Gradient Solver for SVM," *Mathematical Programming*, vol. 127, no. 1, pp. 3–30, 2011.

[41] C. J. Burges, "A Tutorial on Support Vector Machines for Pattern Recognition," *Data Mining and Knowledge Discovery*, vol. 2, no. 2, pp. 121–167, 1998.

[42] Vert, J. P. and Tsuda, K. and Schölkopf, B., "A Primer on Kernel Methods," in *Kernel Methods in Computational Biology*. MIT Press, 2004, ch. 2, pp. 35–70.

[43] J. Kivinen, A. Smola, and R. Williamson, "Online Learning with Kernels," *Transactions on Signal Processing*, vol. 52, no. 8, pp. 2165–2176, 2004.

[44] C.-W. Hsu, C.-C. Chang, and C.-J. Lin, "A Practical Guide to Support Vector Classification," 2003.

[45] J. C. Platt, N. Cristianini, and J. Shawe-Taylor, "Large Margin DAGs for Multiclass Classification," in *Proceedings of the 12th Conference in Advances in Neural Information Processing Systems*, 1999, pp. 547–553.

[46] J. Álvarez Cid-Fuentes, C. Szabo, and K. Falkner, "Online Behavior Identification in Distributed Systems," in *Proceedings of the 34th Symposium on Reliable Distributed Systems*, 2015, pp. 202–211.

[47] "Yahoo! Webscope dataset ydata-labeled-time-series-anomalies-v1_0." [Online]. Available: https://webscope.sandbox.yahoo.com

[48] G. Ricart and A. K. Agrawala, "An Optimal Algorithm for Mutual Exclusion in Computer Networks," *Communications of the ACM*, vol. 24, no. 1, pp. 9–17, 1981.

[49] Y. Tan, X. Gu, and H. Wang, "Adaptive System Anomaly Prediction for Large-Scale Hosting Infrastructures," in *Proceedings of the 29th Symposium on Principles of Distributed Computing*, 2010, pp. 173–182.

[50] B. D. Fulcher and N. S. Jones, "Highly Comparative Feature-Based Time-Series Classification," *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 12, pp. 3026–3037, 2014.

[51] W. W. S. Wei, *Time Series Analysis*. Addison-Wesley, 1994.

[52] Y.-L. Wu, D. Agrawal, and A. El Abbadi, "A Comparison of DFT and DWT Based Similarity Search in Time-Series Databases," in *Proceedings of the 9th International Conference on Information and Knowledge Management*, 2000, pp. 488–495.

[53] R. J. Freund, W. J. Wilson, and P. Sa, *Regression Analysis*. Academic Press, 2006.

[54] C. S. Hood and C. Ji, "Proactive Network-Fault Detection," *IEEE Transactions on Reliability*, vol. 46, no. 3, pp. 333–341, 1997.

[55] X. Wang, K. Smith, and R. Hyndman, "Characteristic-Based Clustering for Time Series Data," *Data Mining and Knowledge Discovery*, vol. 13, no. 3, pp. 335–364, 2006.

[56] V. Černý, "Thermodynamical Approach to the Traveling Salesman Problem: An Efficient Simulation Algorithm," *Journal of Optimization Theory and Applications*, vol. 45, no. 1, pp. 41–51, 1985.

Javier Álvarez Cid-Fuentes is a researcher at the Workflows and Distributed Computing group of the Barcelona Supercomputing Center. His research interests include anomaly detection in distributed systems, as well as parallel programming models for distributed infrastructures. Álvarez Cid-Fuentes received a PhD in computer science from the University of Adelaide in 2018. His email address is javier.alvarez@bsc.es.

Claudia Szabo is a Senior Lecturer at the School of Computer Science at The University of Adelaide in Australia and leads the Complex Systems Program within the Centre for Distributed and Intelligent Technologies (CDIT). Her research interests are in complex systems, in particular in identifying and predicting unexpected behaviours. Her email address is claudia.szabo@adelaide.edu.au.

Katrina Falkner is Head of the School of Computer Science in the Faculty of Engineering, Computer and Mathematical Sciences, and leads the Computer Science Education Research Group (CSER), and the Modelling & Analysis Program within the Centre for Distributed and Intelligent Technologies (CDIT). Professor Falkner has extensive experience in industry consultation, including work with DST Group, NICTA, Google US, Google Australia & New Zealand, the Commonwealth Department of Education and Training, and Telstra Foundation, crossing both the areas of Computer Science Education (within CSER) and Distributed Systems and Modelling (within CDIT).