POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

SYSTEM PERFORMANCE ANOMALY DETECTION USING TRACING DATA ANALYSIS

IMAN KOHYARNEJADFARD
Département de génie informatique et génie logiciel

Thèse présentée en vue de l'obtention du diplôme de *Philosophiæ Doctor*
Génie informatique

Avril 2022

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Cette thèse intitulée :

SYSTEM PERFORMANCE ANOMALY DETECTION USING TRACING DATA ANALYSIS

présentée par **Iman KOHYARNEJADFARD**
en vue de l'obtention du diplôme de *Philosophiæ Doctor*
a été dûment acceptée par le jury d'examen constitué de :

Gilles PESANT, président
Daniel ALOISE, membre et directeur de recherche
Michel DAGENAIS, membre et codirecteur de recherche
Mohammad Adnan HAMDAQA, membre
Naser EZZATI-JIVAN, membre externe

DEDICATION

To my ***Parents****, and my* ***Sister****, For supporting me all these years. I am thankful for having you in my life.*

To my lovely ***Parisa****, Who has always been a source of support and encouragement, and has given meaning and happiness to my life.*

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my research director, **Daniel ALOISE**, for giving me this chance to carry out my research under his supervision. Further, I would take this opportunity to thank him for his valuable insights, guidance, and significant contributions to directing and supervising this work.

I would also like to express my sincere gratitude to my research co-director, **Michel Dagenais**. I want to thank him for all his support and encouragement and for creating such a lovely lab.

Also, I express my thanks to **Naser Ezzati Jivan**, **Seyed Vahid Azhari**, **Mahsa Shakeri,** **Hani Nemati** and **Genevieve Bastien** for their help and constant support during my research.

Finally, thanks to all my colleagues who have assisted me throughout this work and helped me solve my problems.

RÉSUMÉ

Les progrès technologiques et l'augmentation de la puissance de calcul ont récemment conduit à l'émergence d'architectures logicielles complexes et à grande échelle. Les unités centrales de traitement conventionnelles sont maintenant soutenues par des unités de co-traitement pour accélérer différentes tâches. L'impact de ces améliorations peut être observé dans les systèmes distribués, les microservices, les appareils IdO (*internet of things* ou IoT en anglais) et les environnements infonuagiques qui sont devenus de plus en plus complexes à mesure qu'ils grandissent en termes d'échelle et de fonctionnalités. Dans de tels systèmes, une tâche simple engage de nombreux cœurs en parallèle, potentiellement sur plusieurs nœuds, et une même opération peut être servie de différentes manières par différents cœurs et nœuds physiques. De plus, plusieurs facteurs tels que leur distribution dans le réseau, l'utilisation de différentes technologies, leur courte durée de vie, les bogues logiciels, les pannes matérielles et les conflits de ressources rendent ces systèmes sujets à la montée de comportements anormaux. Le haut degré de complexité et la distribution inhérente des petits services compliquent la compréhension des performances de ces environnements. En outre, les outils de surveillance et d'analyse des performances disponibles présentent de nombreuses lacunes.

Différents outils de traçage et de surveillance des systèmes monolithiques et des systèmes distribués ont été explorés dans cette étude pour trouver un moyen d'extraire efficacement les informations de toutes les unités à tous les niveaux. Le suivi du système d'exploitation ou des applications utilisateur nécessite la capacité d'enregistrer chaque seconde des milliers d'événements de bas niveaux, ce qui impose une surcharge au système susceptible d'affecter les performances de l'application cible. Par conséquent, nous avons utilisé un outil de traçage léger appelé Linux Trace Toolkit Next Generation (LTTng) qui fournit un progiciel (*package* en anglais) de traçage à haut débit avec une faible surcharge pour le traçage corrélé du noyau Linux, des applications et des bibliothèques. Cependant, sans outils de diagnostic automatisé, les experts système doivent examiner une quantité massive de données de traçage de bas niveau pour déterminer la cause d'un problème de performances, ce qui prend beaucoup de temps et est fastidieux en pratique.

Dans cette thèse, divers aspects de la détection des anomalies de performance, y compris l'architecture de l'environnement cible et le type de données d'entraînement, ont été étudiés, et plusieurs approches ont été proposées pour réduire le temps de dépannage dans différents environnements. Ces approches guident les développeurs pour découvrir les problèmes de performances en mettant en évidence les parties inhabituelles des données de traçage. Les approches proposées fonctionnent en collectant des données de traçage, en extrayant les données appropriées dans Trace Compass, et enfin en envoyant les données extraites au module de détection.

Dans la première contribution de cette thèse, nous présentons une approche de détection d'anomalies pour la surveillance pratique de processus s'exécutant sur un système afin de détecter des vecteurs anormaux d'appels système. Les flux d'appels système sont divisés en courtes séquences à l'aide d'une stratégie de fenêtre glissante. Contrairement aux études précédentes, notre approche proposée considère la durée des appels système les plus importants comme faisant partie des vecteurs de caractéristiques. La durée d'un appel système dans une fenêtre agit comme la fréquence pondérée de cet appel système. De plus, nous utilisons une machine à vecteurs de support pour détecter les fenêtres anormales.

Notre deuxième contribution aborde le problème de la disponibilité des données étiquetées en proposant des techniques d'apprentissage en fonction du volume de données étiquetées. Une technique supervisée est introduite lorsqu'une grande quantité de données de formation étiquetées est disponible, alors qu'une technique non supervisée est préférée lorsque les données étiquetées ne sont pas disponibles. De plus, nous proposons un nouveau modèle d'apprentissage automatique semi-supervisé, qui bénéficie à la fois de techniques d'apprentissage supervisé et non supervisé, lorsque seules quelques données étiquetées sont disponibles.

Enfin, dans la dernière contribution de cette thèse, nous proposons une méthode basée sur le traitement des langues naturelles (*natural language processing* ou *NLP* en anglais) pour détecter les anomalies de performance dans les environnements de microservices, en plus de localiser les régressions entre les versions. La méthode proposée apprend une représentation des noms d'événements avec d'autres arguments pour remédier aux limitations d'autres méthodes qui n'utilisent pas d'arguments d'événement. Il bénéficie également du traçage distribué pour collecter des séquences d'événements qui se sont produits pendant les durées. De plus, cette méthode ne nécessite aucune connaissance préalable, ce qui facilite la collecte des données d'apprentissage.

ABSTRACT

Advances in technology and computing power have led to the emergence of complex and large-scale software architectures in recent years. The conventional central processing units are now getting support from co-processing units to speed up different tasks. The result of these improvements can be seen in distributed systems, Microservices, IoT devices, and cloud environments that have become increasingly complex as they grow in both scale and functionality. In such systems, a simple task involves many cores in parallel, possibly on multiple nodes, and also, a single operation can be served in different ways by different cores and physical nodes. Moreover, several factors, such as their distribution in the network, the use of different technologies, their short life, software bugs, hardware failures, and resource contentions, make these systems prone to the rise of anomalous system behaviors. The high degree of complexity and inherent distribution of small services makes understanding the performance of such environments challenging. Besides, available performance monitoring and analysis tools have many shortcomings.

Different tools for tracing and monitoring monolithic systems and distributed systems have been explored in this study to find a way to efficiently extract information from all units at all levels. Tracing the OS or user applications needs the ability to record thousands of low-level events per second, which imposes overhead to the system that may affect the performance of the target application. Hence, we employed a lightweight tracing tool called the Linux Trace Toolkit Next Generation (LTTng) that provides high throughput tracing package with a low overhead for correlated tracing of the Linux kernel, applications, and libraries. However, without an automated diagnostic tool, system experts have to examine a massive amount of low-level tracing data to determine the cause of a performance issue, which is really time-consuming and tedious in practice.

In this thesis, various aspects of performance anomaly detection, including the architecture of the target environment and the type of training data, have been investigated, and multiple approaches have been proposed to reduce troubleshooting time in different environments. These approaches guide developers to discover performance issues by highlighting unusual parts of the tracing data. The proposed approaches work by collecting tracing data, extracting the appropriate data in Trace Compass, and finally sending the extracted data to the detection module.

In the first contribution of this thesis, we present an anomaly detection approach for practical monitoring of processes running on a system to detect anomalous vectors of system calls. The system calls streams are split into short sequences using a sliding window strategy. Unlike previous studies, our proposed approach considers the duration of the most important system calls as part of the feature vectors. The duration of a system call in a window acts like the weighted frequency of that system call. In addition, we employ a Support Vector Machine to detect anomalous windows.

Our second contribution addresses the problem of the availability of labelled data by proposing learning techniques depending on the volume of labelled data. A supervised technique is introduced for situations where a large amount of labelled training data is available, whereas an unsupervised technique is preferred when labelled data is not available. Furthermore, we propose a novel semi-supervised machine learning model that benefits from both supervised and unsupervised learning techniques when only a few labelled data are available.

Finally, in the last contribution of this thesis, we propose an NLP-based method to detect performance anomalies in microservice environments, besides locating release-over-release regressions. The proposed method learns a representation of the event names along with other arguments to remedy the limitations of other methods that do not use event arguments. It also benefits from distributed tracing to collect sequences of events that happened during spans. Moreover, this method needs no prior knowledge, which facilitates the collection of training data.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ACRONYMS

LTTng	Linux Trace Toolkit Next Generation
OS	Operating System
VM	Virtual Machine
vCPU	Virtual CPU
pCPU	Physical CPU
FTP	File Transfer Protocol
OLTP	OnLine Transaction Processing
DoS	Denial-of-service
syscall	system call
CI	Continuous Integration
AI	Artificial Intelligence
ML	Machine Learning
NLP	Natural Language Processing
LSTM	Long Short Term Memory
SVM	Support Vector Machine
DBSCAN	Density-Based Spatial Clustering of Applications with Noise
2D	Two-dimensional space
MinPts	Minimum Points
Dist	Distance
ARI	Adjusted Rand Index
t-SNE	t-Distributed Stochastic Neighbor Embedding
Acc	Accuracy
Prec	Precision
Rec	Recall
RBF	Radial Basis Function
SIG	SIGmoid
POLY	POLYnomial

CHAPTER 1 INTRODUCTION

1.1 Motivation

In recent years, the computing infrastructure has significantly evolved, whereas complex systems have facilitated many complicated and large-scale tasks. For example, functional co-processing units accommodate conventional processing units to speed up particular tasks such as virtualization or complex machine learning computations. As a result of these advances, more sophisticated software architectures such as microservices have been launched, where small interconnected services present a complex service such as a web application. Consequently, a simple operation can involve multiple parallel cores being served in a few seconds or milliseconds.

These improvements in hardware and software have increased user expectations. Even when some parts of the system are brought down for maintenance, the users usually do not notice it. Hence, any performance fluctuations or increased latency may lead to user dissatisfaction and revenue losses. However, although considerable efforts have been made to ensure performance, the complexity and large scale of new systems make them fragile and prone to performance anomalies and failures. Different reasons such as misconfigurations, software bugs, hardware faults, network disconnection, aging phenomena of the systems, and extreme load injected by other applications into the system, may degrade the performance of an application or particular service.

Therefore, monitoring and analyzing the performance of programs to find any performance anomaly or degradation is extremely important, and any delay in identifying performance problems and troubleshooting can significantly increase the cost of fixing them. In addition, applications monitoring becomes more challenging by increasing the degree of automation and distribution.

Anomaly detection is vital in such a context because anomalies in data may translate to essential and critical information in many application domains [4]. In the data analysis context, anomalies are data that do not conform to the well-defined notation of normal behaviour [5]. Anomalies might appear in the data for various reasons, which are all challenging for the analyst to reveal. It should be noted that anomaly detection is different from noise detection and noise elimination, which refer to unwanted noise in the data. Besides, we should distinguish between anomaly detection and novelty detection, which is useful in identifying new unobserved patterns.

The main focus of this thesis is to promote the analysis of the performance of monolithic and microservice-based applications and enhance the detection of anomalous behaviors and analysis of the root causes of detected anomalies inside these applications. This includes the development of several algorithms to process the large volume of tracing data by taking advantage of machine learning technologies and open-source tools such as LTTng and Trace Compass.

The insights provided by our research are intended to help companies detect and avoid different kinds of anomalies. The system's behavior is defined by the sequence of events (such as system calls or states) obtained by open source tools. In this way, an extensive amount of information is accessible from the CPU, Memory, I/O, and network. Experiments indicate that the definition of normal behavior is stable during standard UNIX programs [6]. Notably, the horizon of this work is the development of an anomaly detection framework with negligible overhead and a minimum human intervention that can be applied in a variety of environments and ultimately convert tracing information into meaningful visualizations.

1.2 Research Questions

Given the issues presented earlier and through a feasibility analysis of different methods, five basic research questions are identified that have not been addressed in depth in this area:

- How to identify deviations in the performance of the system? Which deviations should be considered as anomalies?
- How to develop an anomaly detection method that is applicable to different environments and does not impose excessive overhead on the system?
- What are the best methods and tools for collecting information about a system? In order to characterize an application's execution status, low-level information is required.
- What are the most appropriate parameters and features for detecting anomalies?
- Which machine learning methods can be used in performance anomaly detection?

1.3 Research Objectives

Based on recent studies, there is a need to elaborate more automated techniques for performance anomaly detection and root cause analysis. Several anomaly detection techniques using supervised and unsupervised machine learning methods have been proposed in this thesis to obtain a highly accurate view of the performance anomalies while keeping the tracing overhead almost negligible.

The major aim of this research is to investigate how machine learning techniques can be adapted and applied to improve automated performance anomaly detection in execution flow. We refined our objective into four specific objectives as follow:

- To review the present methods and trends in this research area and identify open challenges.
- To study the behavior of systems using tracing data and machine learning.
- To develop a new algorithm for automatic performance anomaly detection with limited human intervention.
- To develop, implement and evaluate a framework that can automatically detect performance anomalies, and investigate the root cause of the anomalies in microservice environments.

1.4 Contributions

In line with the research objectives mentioned above, this dissertation offers the following main contributions in the field of performance anomaly detection:

- It presents a new approach in the field of anomaly detection, which relies on LTTng and Trace Compass. LTTng imposes a small overhead on the system and makes this approach usable in various environments, including microservices or monolithic applications.
- It proposes enhanced supervised, unsupervised, and semi-supervised techniques to find abnormal behaviors during the streams of system calls.
- It presents a new framework to find abnormal behaviors in microservice environments by employing different parameters of the events and NLP.
- It provides several meaningful visualizations in Trace Compass, such as a time chart, which allows the developers to visually analyze the abnormal parts of the execution flow.

1.5 Outline

This thesis is composed of eight chapters. Following this introduction, Chapter 2 provides some background information and a summary of available methods for performance analysis, feature extraction, machine learning, and anomaly detection. Chapter 3 presents our general methodology and describes the process of identifying problems, applicable cases, milestones, and final results in terms of articles. In the following, three research articles are presented in Chapters 4, 5, and 6.

Chapter 4 presents an anomaly detection approach for practical monitoring of processes running on a system to detect anomalous vectors of system calls. Our proposed approach computes the execution time of system calls in addition to the frequency of each individual call in a window. Finally, a multi-class support vector machine is applied to evaluate the system's performance and detect anomalous sequences.

In Chapter 5, we propose an anomaly detection framework that reduces troubleshooting time, besides guiding developers to discover performance problems by highlighting anomalous parts in trace data. This chapter addresses the problem of the availability of labelled data by proposing several learning techniques. A supervised method is introduced when large amounts of labelled training data are available. An unsupervised method has also been offered to be used in the absence of labelled data. Moreover, we propose a novel semi-supervised machine learning model within the proposed framework that benefits from both supervised and unsupervised learning techniques when only a few labelled data are available.

Chapter 6 proposes a natural language processing (NLP) based approach to detect performance anomalies besides locating release-over-release regressions in microservice environments. This approach benefits from distributed tracing data to collect sequences of events that happened during spans. Furthermore, this approach needs no prior knowledge, which facilitates the collection of training data.

Chapter 7 summarizes the findings, limitations, and main recommendations for future work. Finally, concluding remarks regarding this research are provided in chapter 8.

1.6 Publications

The chapters mentioned above are based on the published and submitted articles introduced in this section. The articles are as follows:

1. Iman Kohyarnejadfard, Daniel Aloise, Seyed Vahid Azhari, and Michel Dagenais. "Anomaly detection in microservice environments using distributed tracing data analysis and

NLP.", Submitted in Journal of Cloud Computing.

2. Iman Kohyarnejadfard, Daniel Aloise, Michel R. Dagenais and Mahsa Shakeri. "A framework for detecting system performance anomalies using tracing data analysis and Machine Learning.", Entropy 23, no. 8 (2021): 1011.

3. Iman Kohyarnejadfard, Daniel Aloise, and Mahsa Shakeri. "System performance anomaly detection using tracing data analysis." Proceedings of the 2019 5th International Conference on Computer and Technology Applications. 2019.

CHAPTER 2 LITERATURE REVIEW

Tracing is a popular way to analyze, debug, and monitor a system. It is a valuable technique for gaining information about a system while minimizing the monitoring influence, and different works have applied machine learning techniques to analyze this information and find performance anomalies. Several performance analysis tools and methodologies exist in different environments. So, it is essential to review the available state-of-the-art in performance analysis and anomaly detection tools and see how these tools and techniques have advanced. This chapter investigates the currently available tools, methods, and approaches developed in this area of research.

In the first section of this chapter, we examined the basic concepts related to anomaly detection. We also introduced various data collection methods as well as the machine learning techniques that can be used to analyze the collected data. Then the previous work and the progress made in the field of anomaly detection were investigated in Section 2.2. Finally, in Section 2.3, while reviewing the shortcomings of previous methods, we explore how our proposed methods differ from the previous related literature.

2.1 Basic concepts in anomaly detection

This section starts with some definitions and terminology related to tracing and anomaly detection in subsection 2.1.1. Background information on performance anomalies is studied in subsection 2.1.2. This subsection explains how abnormalities arise and affect the system. In subsection 2.1.3, we talk about the properties of an anomaly detection tool. Next, a brief review of the most popular data collection tools and technologies are presented in subsection 2.1.4. Moreover, the most valuable sources of information for anomaly detection, such as system calls, are reviewed in this subsection. This informs us about the various features that may be used to create a dataset. Following this, the most useful trace analysis tools are investigated in subsection 2.1.5. Then, we shall have a detailed look at some machine learning approaches in subsection 2.1.6.

2.1.1 Definitions and Terminology

In this subsection, we provide some of the most useful definitions and terms that are necessary to understand the rest of this dissertation.

Definitions related to Tracing:

- **Operating system:** A software that manages a computer's hardware is called Operating System (OS). It is also used as a basis for application programs and acts as an intermediary between the computer user and the computer hardware [7].
- **Kernel:** Kernel mode and user mode are two modes of operation in most computers: The operating system runs in kernel mode, which is also called supervisor mode. In kernel mode, it has full access to all the devices and can execute any instruction the machine is able to execute [8].
- **Tracepoint:** It is a statement located in the code of an application that provides a hook to invoke a probe. A tracepoint can be enabled or disabled dynamically [9].
- **Probe:** a prob is a function that is hooked to a tracepoint and is called at runtime if the tracepoint is enabled. It should be as small and fast as possible, to add low overhead to the system and is either implemented by the tracer or by the user [10].
- **Event:** The event is created by a tracer, when it encounters an active tracepoint at run-time, and contains information such as a timestamp.
- **System call:** A system call is a way for programs to communicate with the operating system. A system call is created when a computer program makes a request to the operating system's kernel, thus providing the services of the operating system to the user programs [11].
- **Timestamp:** The timestamp refers to the time of an event, upon encountering the tracepoint, which is stored as a log element or metadata.
- **Atomic Operation:** It has the property of being indivisible, which means that intermediate values or intermediate states are inaccessible to other operations. They usually need to be supported by the hardware or the operating system, and developers must take care of the atomicity [9].
- **Monolithic architecture:** It is the traditional approach to software development. In this architecture, the functions are encapsulated into a single application [12].
- **Microservices architecture:** It is a technique for developing software applications that has inherited principles and concepts from the Service-Oriented Architecture (SOA) style. A service-based application includes very small loosely coupled software services [13].

- **Virtualization:** This technology enables the physical machine to run one or more virtual machines at the same time.
- **Virtual Machine (VM):** It is a software implementation of a machine which can execute programs like a physical machine [14].
- **Cloud Computing:** Cloud computing is a large-scale distributed computing model which offers a pool of abstracted, virtualized, dynamically-scalable, managed computing power, storage, and platforms. It is often used to deliver services on-demand to customers over the Internet [15].
- **Node:** A node is an independent physical device in a network of other tools, with the ability of sending, receiving, or forwarding information.
- **Cluster of nodes:** It's a set of nodes that interact with each other, over a high bandwidth communication network.
- **Host:** There are several degrees of machine virtualization. Based on the definition of Cloud Computing, some approaches allow installing and running unmodified guest operating systems on a host OS [16].
- **Instrumentation:** Instrumentation is a code section that can be added to any part of a target system, either at run time or compile time, for collecting information about the applications. Instrumentation at run time is called dynamic binary instrumentation, and instrumentation at compile time is called static instrumentation. In static instrumentation, the source code of the application is needed, and an extra recompilation time imposed. Also, the applications need to be restarted. On the other hand, in dynamic instrumentation, the source code is not required, and the code is injected into the application at run-time [17].

Definitions related to Machine Learning:

- **Machine Learning:** It is the scientific study of algorithms and statistical models that computer systems utilize to do a specific task, without using explicit instructions, using patterns and inference instead. Machine learning algorithms construct a mathematical model based on sample data, to make decisions or predict something without being explicitly programmed to perform the task [18].
- **Anomalies:** In general, anomalies are data that do not fit in a standard form of data in that area [5]. Anomalies might appear in the data for several reasons.

- **Anomaly detection:** Anomaly detection is the process of identifying unusual or rare items, events, or observations which are significantly different from the majority of the data. Anomaly detection has become a hot issue, because it can provide valuable information at critical moments, which can help to prevent or take appropriate action [19].

- **Anomaly prediction:** Anomaly prediction is valuable for anticipating future anomalies in data. It is valuable for the defense against anomalies before they can have their adverse effect.

- **Training dataset:** A training dataset is a part of the dataset that is used for learning, and also to fit the different parameters of the method.

- **Validation dataset:** This part of the dataset is used to tune the hyperparameters such as the architecture of a classifier. A validation dataset is required to avoid over-fitting when the classification parameters need to be adjusted [20].

- **labelled data:** labelled data is a group of samples that have been tagged with one or more labels, for example 1 or 0.

- **Supervised and unsupervised learning:** Supervised learning algorithms are trained using labelled samples, such as an input where the desired output is known. In contrast, unsupervised learning algorithms are used when we don't have labels [21].

- **Classification:** Given an $n \times d$ training data matrix D and a class label value in $\{1, ..., k\}$, associated with each of the n rows in D, the classification consists of creating a training model M, which can be used to predict the class label of a d-dimensional record $Y \in D$ [22].

- **Clustering:** Given a set of data points, partition them into groups containing very similar data points [22].

- **Training:** Training is a process in which a machine learning (ML) algorithm is fed with sufficient training data to learn [21].

- **Validation:** After clustering or classification of the data has been determined, it is essential to evaluate its quality. This problem is referred to as validation. Different validation criteria exist for classification and clustering.

- **Feature engineering:** It is the process of using domain knowledge to extract features (characteristics, properties, attributes) from raw data [23].

2.1.2 Performance Anomaly Detection

Performance anomalies are the most significant obstacles for the system to perform confidently and predictably in enterprise applications. Many sources can cause anomalies, such as varying application load, application bugs, updates, and hardware failure. In a situation where the workload is the source of the anomaly, the application imposes continuous and more than expected workload intensity to the system. Faults in system resources and components may considerably affect application performance at a high cost [24]. In addition, software bugs, operator errors, hardware faults, and security violations may cause system failures.

The preliminary performance profiling of a process that reflects its typical behavior can be done using synthetic workloads or benchmarks. At a higher level, the performance of computer systems is delineated by measuring the duration for performing a given set of tasks, or the amount of system resources consumed within a time interval [25].

There exist many metrics for measuring the performance of a system. Latency and throughput are the most common ones. They are used to describe the operation state of a computer system. The elapsed time between the beginning of an operation and its completion is the latency, (e.g., the delay between when a user clicks to open a webpage and when the browser displays that webpage). Throughput is a measure of how many jobs a system can perform in a given amount of time (e.g., the number of user requests completed within a time interval). In addition, the resource utilization of an application indicates the amount of resources (e.g., number of CPUs, and the size of physical memory or disk) used by that application. The CPU utilization is the percentage of time during the CPU is executing a process, whereas the memory utilization is the amount of storage capacity dedicated to a particular process.

Figure 5.1(a) shows an example of the CPU utilization of a process during its lifetime. When an application is running normally, the CPU used by that application is conventional. Hence an expected maximum CPU utilization threshold can be defined for each application. In this case, if the CPU usage exceeds the threshold value, the process behavior is prone to the existence of an anomaly. Furthermore, as represented in Figure 5.1(b), during the anomalous execution of a process, the latency usually increases, while this curve has a relatively steady trend during normal behavior [26].

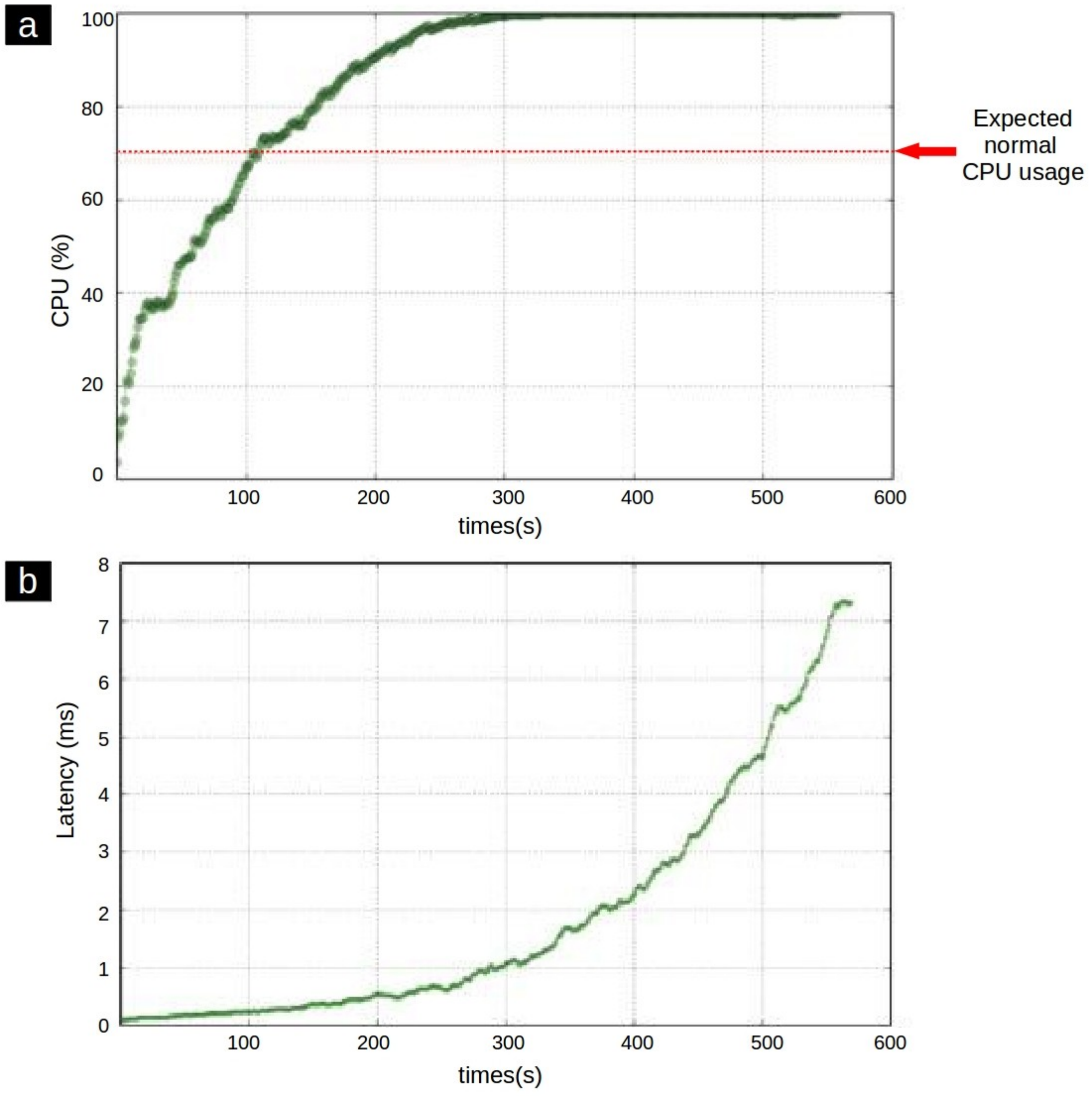


Figure 2.1 Above: the CPU usage of an application during the execution time. Below: an anomalous latency growth pattern [1].

2.1.3 Anomaly detection tools

Anomaly detection tools observe the target system's behavior to reach a decision about the status of the system. Figure 2.2 presents a general overview of such a tool. In the first step, the data collection module employs data-providing techniques and tools to record the system state and its changes. It gathers data and sends the data to the analysis module. Next, in the analysis module, the provided data is processed, and meaningful information is extracted to represent the performance of a target system. Finally, the analyzed data is sent to the visualization module, which generates views for system administrators. It may also trigger an alert. In the following, the characteristics of the desired tool are examined, and then each of these modules is described in detail.

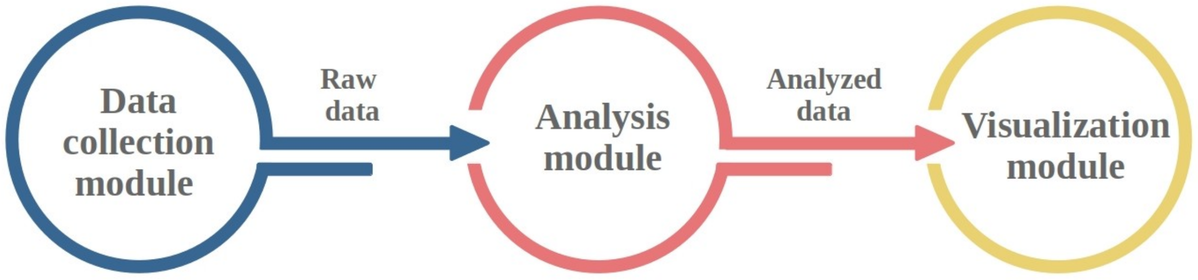


Figure 2.2 A general overview of a performance anomaly detection tool and its component modules.

Characteristics of the desired anomaly detection tool

Several tracing and analysis tools have been employed for monitoring the performance of Operating Systems, processes, and whole systems. These tools need to evolve to be able to capture the new types of anomalies in different domains. Some essential capabilities and features of a suitable analysis tool are described in this subsection as follows [27]:

- **Availability:** It is essential to ensure that the analysis tool is fully functional and serves the requests properly when it is needed. Availability can also be defined as the probability that a system has not failed when it needs to be used.
- **Generality:** It is crucial for an analysis tool to be free from restrictions and limitations, to adapt to each configuration, and provide reliable reports. Unfortunately, most tools often try to satisfy a specific need, and more general tools are needed in this area.
- **Open-source:** The computing infrastructure and software management are continuously changing and growing. Accordingly, it is imperative for the analysis tools to be adaptive. Closed-source tools cannot adapt to the rapid pace of technological changes. Fortunately, there are many open-source tracing, logging, and visualization tools available that can be employed to build an open-source anomaly detection tool.
- **Resolution:** The analysis tool needs to provide high-resolution results. Each event should have nanosecond precision.

- **Dynamic:** An analysis tool should be dynamic in the sense that it deals with a changing environment. Static analysis tools cope with a particular system, and do not have this ability.
- **Robust:** An analysis tool needs to cope with errors and new situations while continuing its operation. This is critical since a fault in one part has a devastating effect on the whole system.
- **Low overhead:** An appropriate analysis tool should add a low overhead to the system.

Performance analysis tools take as input a vast amount of data that accurately records system behaviour over time. Tracing data and logs, as the most popular information resources, have this ability, and they include an enumerated collection of events, sorted by timestamp. Various works make different uses of this structure, as the input of their performance analysis tool. Frequency of events, TF-IDF, graphs, and subsequences of events have been widely used [28–30]. Data collection and representation techniques are discussed in detail in this survey.

The way of reporting the anomalies is also an essential aspect of any anomaly detection technique. Two major types of outputs can be defined for such techniques. Some techniques assign an anomaly *score* to each sample in the test data, by considering the degree to which that sample is considered an anomaly. Hence, the output of the score-based techniques is a ranked list of anomalies. An analyst can investigate the top few anomalies or even can define a cutoff threshold to select the anomalies. Some other techniques assign a *label* (normal or anomalous) to each test sample. They provide binary labels or more than two labels. The analyst does not need to find any threshold, but can also investigate anomalous points directly.

2.1.4 Data collection module

The data collection module is the first building block of any performance analysis tool. It records the behavior of a target system for subsequent processing, using statistics or machine learning techniques to reach a decision about the system status. The following subsections elaborate more on the data-providing techniques and most useful tracing tools. We will also explain how the detailed information obtained from these tools can be used to analyze the system performance and build the appropriate database.

Most popular data providing techniques

The first step in detecting anomalies in any system is to provide the data shown in Figure 2.2 as data collector module. There are different technologies for collecting data, each with different applications. The first technology is **Logging** [31]. A log entry is a message from the operating system or application code indicating that an event occurred. Logs usually contain body, content, and time and are written as structured or unstructured data in log files. Logs are widely being used in statistical analysis researches [32]. However, despite the simplicity of this technique, logging is not efficient when the events are generated with high frequency.

Another traditional technique is **Debugging**. Debugging is a part of the process of software testing and refers to the process of locating and correcting code errors [33]. GDB is a common debugging tool, which allows one to check what is going on inside another program during its execution. However, this technique can not be used in this project because the purpose of this thesis is to find anomalies in programs that have been deployed.

Sampling is another technique that can be used to generate reports on system-level and application-level performance. Many processors have their own dedicated performance monitoring hardware. Performance analysis tools (e.g., OProfile) employ this hardware (or a timer-based substitute in cases where performance monitoring hardware is not present) to collect samples of performance data [34].

Finally, **Tracing** is another robust and efficient approach for debugging and reverse engineering complex systems. Many tracers have emerged across all software stack layers and even at the hardware level in recent years. Tracing is a high-speed system-wide fined grained logging technique. Traditional logging approaches suffer from two significant bottlenecks: 1) gathering low-level information is really difficult, 2) time accurate details about the system's behavior in real-time are hard to obtain. Logging is proper for a high-level analysis of less frequent events, such as user accesses, exceptional conditions, or database transactions. In general, logging is one of the applications of tracing. In tracing, a particular piece of code called tracepoint is inserted in the code. A tracepoint can be as small as a function call or be part of the standard kernel tracing infrastructure. The generated events are extremely low level and may occur more frequently. Tracing techniques can be categorized in several ways:

- **Hardware tracing vs. software tracing:** Tracing techniques can be divided into hardware and software tracing, based on the source of the events. The ability to record a complete trace of the instructions executed is available in modern CPUs. On the other hand, software tracing does not need any specific hardware, and is based on the tracepoints in the program code.

- ◦ **Static tracing vs. dynamic tracing:** Tracing can be classified as static or dynamic, depending on how tracepoints are added to the code. In static tracing, the program source code is modified, and recompilation is required. In contrast, tracepoints are added directly into running processes via dynamic instrumentation, with the help of debugging information generated during the compilation process.

- ◦ **kernel tracing vs. userspace tracing:** Tracing is divided into kernel and userspace tracing, depending on the tracing domain. The term userspace tracing is used when events are collected from userspace, and Kernel tracing refers to the events gathered from the OS kernel [35].

The advantages of tracing over other techniques led us to use it as the primary data providing technique during this project.

Most useful tracing tools

Tracing is a technique employed to find out what goes on in a running software system. The *tracer* is the software that records events in tracing files. These events are generated by specific instrumentation points located in the software source code. Tracers have the ability to trace the OS and user applications simultaneously. Recording low-level events with an occurrence frequency of thousands per second, helps to solve a wide range of problems. We study some of the most useful available tracing tools in this subsection.

- **LTTng:** Linux Trace Toolkit next generation (LTTng) is a tool able to perform an extremely fast and very low-overhead kernel and userspace tracing [36]. Low overhead is a point that makes LTTng a good choice for online applications, so it is an appropriate choice for our anomaly detection project. The tracing technique used in LTTng implements a fast wait-free read-copy-update (RCU) buffer for storing data from tracepoint execution. Figure 2.3 presents the typical tracing process flow with LTTng. The components of LTTng, and how they interact with the application and the Linux kernel, are clearly explained in this figure. The session daemon is responsible for managing and controlling other components. At the beginning of the execution of the instrumented application, which contains the user's desired tracepoints, it registers itself to the session daemon. This is similar for kernel traces. After this registeration, the session daemon will manage all the tracing activity. Another essential part of LTTng, namely the consumer daemon, has the responsibility of handling the trace data coming from the applications. It exports the raw tracing data in the Common Trace Format (CTF), to be written on the disk. The CTF file extension format is a structured compact binary format, which is a good choice for further analysis by Babeltrace or Trace Compass. LTTng supports both static and dynamic tracing. Tracepoint can be added in both the source code of the kernel, and in user-space programs with UST. LTTng contains the LTTng kernel modules as well as the UST library. The LTTng kernel modules hold a set of probes to be attached to Linux kernel trace-points and entry and exit points of syscall functions.

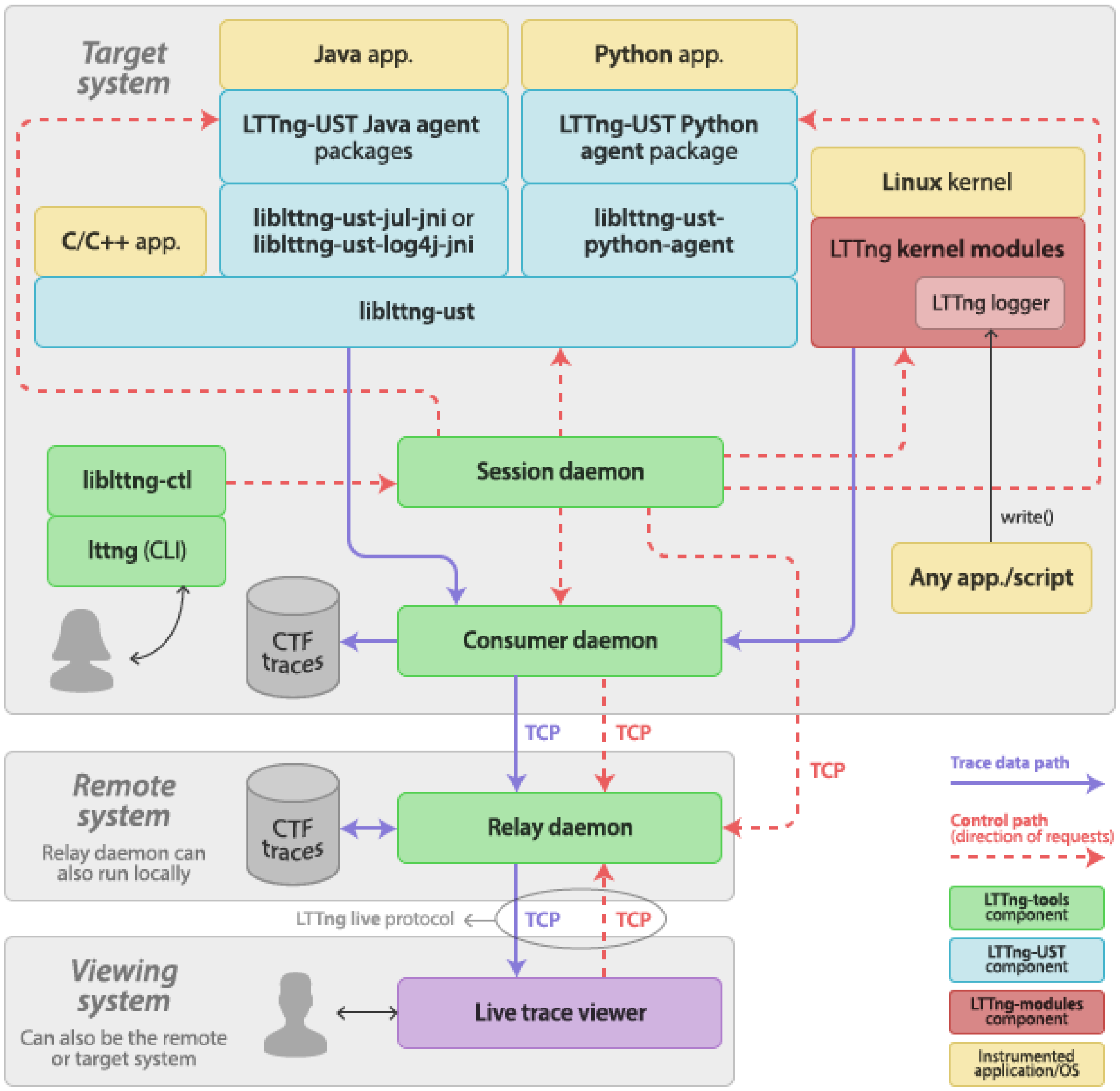


Figure 2.3 A tracing process flow with LTTng [2]

- **DTrace:** Sun Microsystems created this tool to do kernel tracing on its Solaris platform, but it was quickly ported to MacOS, Linux, and other platforms [37]. The DTrace tool can interpret user scripts and is able to load code into the Linux kernel for further execution and collecting the outputted data. DTrace is very flexible and dynamic but causes a larger overhead than LTTng, especially for multi-threaded applications [9].

- **eBPF:** eBPF is a subsystem in the Linux kernel in which a small bytecode interpreter can execute programs passed from the user space to the kernel. Such programs can be attached to tracepoints and KProbes using system calls, and they can output data to user space when executed, thanks to different mechanisms like pipes, VM register values, and eBPF maps [38].

- **Ftrace:** Ftrace is a tracer included in the Linux kernel which provides dynamic instrumentation, function tracing, system calls tracing, and so on. It presents a function graph in which the entry and exit of all functions at the kernel level are shown. It supports Kprobe for dynamic instrumentation. This method does not write events to the disk automatically and keeps them in memory. Moreover, the size of a payload is limited to the size of the page. A linked list is used to implement the ring buffer, and a buffer page can be read once it is full [39].

- **SystemTap:** SystemTap presents a free software infrastructure to simplify the collection of information about a running Linux system. It allows developers and administrators to write and reuse simple scripts to scrutinize the activities of a live Linux system. The basic idea behind SystemTap is attaching user-defined handlers to events. When any specified event occurs, the Linux kernel executes the handler, as if it were a quick subroutine, and then resumes. A handler is a series of script language statements which are designated to choose what should be done whenever the event occurs. This work includes typically extracting data from the event context, storing it into internal variables, or printing results [40].

- **sysdig:** This tool, like SystemTap, uses scripts to analyze Linux kernel events. Sysdig executes the scripts, or chisels in sysdig's jargon, in Lua while the system is being traced or afterward. The interface of sysdig and the curses-based csysdig tool is a command-line tool [41].

- **Distributed tracing tools:** Distributed tracing is a tracing method used to profile and monitor distributed applications, especially microservice-based applications. Distributed tracing helps find the location of failures and the factors that cause poor performance.

Over the past few years, the open-source community has developed several interesting distributed tracing tools and standards, the most important of which is OpenTracing. OpenTracing is the foundation for tools like Jaeger, Zipkin, and OpenCensus. OpenTracing consists of a set of standards and techniques that allows developers to add instrumentation to their application code using APIs that do not restrict them to any particular product or vendor. To this end, a coherent API specification for several programming languages and frameworks is provided. Spans are the primary building blocks in OpenTracing. In other words, a Trace can be thought of as a directed acyclic graph of Spans, where the edges between Spans are called References [42]. **Zipkin** [43] is an open-source tool that helps gather timing data needed to troubleshoot latency problems in microservice architectures [44,45]. **Jaeger** [46], inspired by Dapper and OpenZipkin, developed by Uber Technologies, is a popular tool that supports OpenTracing. This tool has been widely used for monitoring and troubleshooting microservices-based and distributed systems [47, 48]. It has instrumentation libraries in C++, C, Go, Java, Node, and Python. However, the high-level information that these tools provide is not always sufficient to characterize the execution status of the system, since they do not offer kernel events. In contrast, LTTng provides details of the program execution with higher resolution by presenting kernel and userspace events. LTTng imposes the least overhead on the system among other solutions.

Most useful information resources for anomaly detection

System-level performance metrics such as CPU utilization, RAM utilization, hard drive read rate, hard drive write rate, network device transmission rate, and network device receive rate are broadly used to discover performance anomalies or evaluate performance degradations. Besides, thread/process-level analysis such as call stack and execution flow analyses can pave the way for defining the anomaly detection model. In this subsection, we study the critical information resources that can help us in this project:

- **CPU utilization:** The CPU utilization refers to non-idle time, i.e. the time the CPU is not running the idle thread. The operating system kernel usually tracks this during context switch. This metric is as old as time-sharing systems.

- **Memory utilization:** The Memory Utilization metric is defined as an average utilization statistic derived from the percentage of available memory in use at a given time, averaged across the reporting interval.

- **Events:** Events are the most popular source of information and has been widely used in performance analysis. A trace contains a sequence of time-ordered events, saved in a trace file. We can record user application and operating system events at the same time. Events consist of well-defined fields such as name, timestamp, process ID, and so on. They open the possibility of resolving a wide range of problems, and performing application-specific analyses to produce reduced statistics and graphs useful to resolve a given issue. Many anomaly detection methods use events frequency, TF-IDF, execution graph, and sequence of events name, to create meaningful datasets for modeling system behavior, using machine learning methods [28, 49, 50].

- **System calls:** System calls are essential traceable events for determining abnormal behavior in a computer system. A system call is a way for programs to communicate with the operating system. System call traces generated by program executions are stable and consistent during the program's normal activities, so that they can be used to distinguish the abnormal operations from normal activities. System call streams are enormous and suitable to use in machine learning. A single process can produce thousands of system calls per second. Moreover, system call sequences can provide both momentary and temporal dynamics of process behavior.

- **Call stack:** We can obtain much information about the current status and the history (or the future, depending on how it is interpreted) of program execution from the call stack, especially in the form of return addresses. Therefore, the call stack of a program execution is an excellent information source for intrusion and anomaly detection [51].

- **Execution flow analysis:** Tracking processes and finding dependencies between them can show the cause of the process' waits. Different reasons can force a process to wait. A process can wait for a timer to fire. It may also wait for another process to wake it up, indicating that the process was waiting for another process to finish a task. The process can even wait for a device. The construction of the execution graph reveals the dependencies among the processes and different resources.

- **Critical path:** Given the list of participating tasks, the dependencies between them, and the time taken to complete each task, the critical path is determined by finding the longest sequence of tasks to complete the process. The length of the critical path is an estimate for the overall time to completion. The points on the critical path are ideal targets for optimization, because decreasing the time to complete these tasks decreases the length of the critical path and also overall time to completion. The critical-path analysis is a great technique for recognizing the critical bottlenecks in a complex system with multiple concurrent operations [52].

2.1.5 Analysis module: Most useful Analysis tools

Trace viewers and analyzers are specialized tools designed to read the trace files and perform various analyses to produce statistics and visualizations that help system experts solve problems more quickly.

Trace Compass

It is an Eclipse based analysis and visualization tool that provides various views such as a call graph or a timeline-based view for trace data generated by LTTng or other compatible tracers [53]. Trace Compass promotes the visualization and analysis of traces and logs from multiple sources. It facilitates diagnostic and monitoring operations of systems, from a simple device to an entire cloud. Trace Compass can take multiple traces and logs from various sources and formats, and join them into a single event stream that allows system-wide tracing. It is possible to correlate application, operating systems, virtual machine, and hardware traces to present the results together, delivering unprecedented insight into your entire system [54]. Using this tool gives us some benefits, such as a faster resolution of complex problems, easier system performance optimization, etc. It can be integrated into the Eclipse IDE or even used as a standalone application. Eclipse plug-ins facilitate the addition of new analysis and views. The EASE scripting feature also makes it very simple to write new scripts and analyses for developers [55].

Babeltrace

This open-source project produces a library with a C API, Python 3 bindings, as well as a handy command-line tool that makes it very easy for the user to view, convert, transform, and analyze traces. Moreover, Babeltrace is the reference parser implementation of the Common Trace Format (CTF), a popular trace format followed by various tracers such as LTTng. Using the Babeltrace library and its Python bindings, we can read and write CTF traces [56].

Traceshark

Traceshark is another trace visualization tool developed in C++. It supports Linux kernel traces containing Ftrace and Perf events and allows one to display some basic analyses, such as CPU status, frequency, and scheduling tasks [57].

Perfetto

It is a production-grade open-source performance instrumentation and trace analysis tool. It includes services and libraries for recording system-level and application-level traces. It also offers a library for analyzing traces using SQL and a web-based interface to visualize and explore multi-GB traces [58].

SvcTraceViewer

Windows Communication Foundation (WCF) Service Trace Viewer Tool helps analyze diagnostic traces that WCF generates. SvcTraceViewer provides an easy way to merge, visualize, and filter trace messages in the log in order to diagnose, repair, and verify WCF service issues [59].

2.1.6 Analysis module: Study on machine learning approaches for anomaly detection

This subsection introduces different supervised and unsupervised machine learning approaches recently used in anomaly detection tools and research projects.

Supervised techniques

The purpose of supervised learning is to construct a brief model of the distribution of class labels in term of predictor features. The trained classifier is then used to assign class labels to test samples where the value of the features are known, but the value of the class label is unknown [60]. So the supervised anomaly detection techniques operate in two phases. The training phase learns a classifier using the available labelled training data, and then in the test phase, a test sample is classified as normal or anomalous, using the trained classifier. Actually, the classifiers' goal is to distinguish between normal and anomalous classes that can be learned from the given feature space. Classification-based anomaly detection techniques are categorized into two-class and multi-class techniques. Multi-class classification based anomaly detection techniques assume that the training data includes multiple normal classes and multiple anomaly classes. Using these anomaly detection techniques, a classifier can distinguish between each standard class or each anomaly class [61].

- **Support Vector Machine:** Support Vector Machines (SVMs) [62, 63] have been employed for anomaly detection in many applications. SVM finds the hyperplane with the largest margin that classifies the training set samples into two classes. Then the unseen test samples are labelled by checking the sign of the hyperplane's function. Considering each sample X_i, for $i = 1, \ldots, n$ of the training data and its associated label y_i, SVM determines the optimal hyperplane by solving the following problem:

$$\min_{\omega,b} \frac{1}{2}\omega^T\omega + C\sum_{i=1}^{n}\xi_i \tag{2.1}$$

$$s.t.\ y_i\left(\omega^T\phi\left(X_i\right) + b\right) \geq 1 - \xi_i\ ,\ \xi_i \geq 0\ ,\ i = 1, ..., n \tag{2.2}$$

Where ω is a d-dimensional vector and ξ_i is a measure of the distance between the misclassified point and the separating hyperplane. The function $\phi\left(x_i\right)$ projects the original data sample x_i into a higher dimensional space and b is the bias. C controls the penalty associated with the training samples that lie on the wrong side of the decision boundary.

- **K-Nearest Neighbors (KNN):** The K-Nearest Neighbors (KNN) technique is a supervised machine learning algorithm for solving classification and regression problems that is simple to implement. KNN is a non-parametric algorithm since it does not make any assumption on the underlying data. Moreover, KNN is a lazy learning algorithm because it does not require a training phase, and it uses all the training data in the classification process. KNN is implemented by the following steps:
  - Load training and test data.
  - Choose the value of K.
  - For each data point in the test data:
    - Find the Euclidean distance to all training samples. Manhattan or Hamming distances can also be used instead of the Euclidean distance.
    - Store the distances in ascending order.
    - Choose the top K points from the sorted list (K nearest neighbors).
    - Label the test point based on the most frequent class present in the selected points.
  - End

- **Neural Networks:** Neural networks, also known as artificial neural networks (ANNs), are a subset of machine learning that can be used for clustering and classification. The name and structure of these networks are inspired by the human brain, and they mimic how biological neurons signal to each other. Neural Networks consist of multiple layers

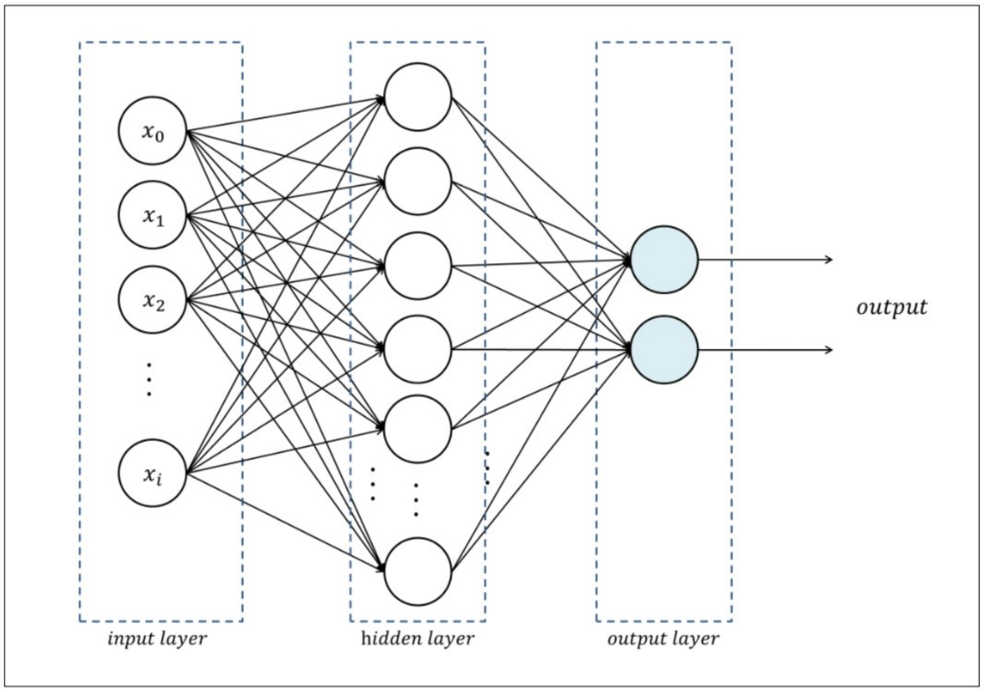


Figure 2.4 A simple Neural Network.

(two or more), where the first layer is the input layer, the last layer is the output layer, and some hidden layers are located in-between. The layers are connected by means of weights evaluated during the training phase.

Neural networks such as the feed-forward neural network, also known as the multilayer perceptron [18], are widely used to discover useful patterns or features that describe user behaviour on a system. They use the set of relevant features to build classifiers that can recognize anomalies and known intrusions, hopefully in real-time [64]. Figure 2.4 presents a simple neural network. The circles represent neurons, and each line represents a synapse. The inputs received by the synapses are multiplied by the weights.

- **Long-Short-Term-Memory (LSTM) Recurrent Neural Networks:** Short-term memory is a problem in RNNs (Recurrent Neural Networks). If one tries to predict something from a paragraph of text, RNNs may leave out important information at the beginning. LSTM was introduced as a solution to the short-term memory problem. It is one of the most popular techniques among several deep neural network techniques available, first introduced by S. Hochrieter J. Schmidhuber. LSTM networks are capable of learning order dependencies in sequence prediction problems [65]. This is a behaviour needed in complex problem domains like anomaly detection. Like recurrent neural networks, LSTM networks process the data passing on information as it propagates forward. However, the operations within the LSTM's cells are different. These operations allow the LSTM to keep or forget information.

The basis of LSTM is the cell states and various gates. The cell states act as the memory of the network, and they can carry relevant information throughout the processing of the sequence. So even information from the earlier time steps can lead to later time steps and reduce the effects of short-term memory. As the cell state continues its journey, information is added to the cell state or removed from it via gates. The gates are different neural networks that can learn which data in a sequence should be kept or discarded [66].

Unsupervised techniques

Clustering is an unsupervised method to group similar data instances (observations, data items, or feature vectors) into clusters [67]. The clustering problem has been addressed in many contexts, and by researchers in many disciplines. So it is useful as one of the steps in exploratory data analysis.

- **K-Means:** K-Means is an iterative clustering algorithm that groups samples based on their feature values into k different clusters. Data points that are assigned to the same cluster are supposed to have similar feature values. This algorithm aims to make the intra-cluster data points as similar as possible, while keeping the clusters as different as possible. K-Means assigns data points to a cluster, such that the sum of the squares of the Euclidean distances of data points and the cluster's centroid is at the minimum. The cluster's centroid is the arithmetic mean of all the data points which belong to that cluster. K-Means works as follows [68, 69]:
  - Determine the number of clusters K.
  - Shuffle the dataset and then randomly select K data points without replacement as the centroids.
  - Keep iterating until there is no change to the centroids:
    - Assign each data point to its closest centroid.
    - Compute the new centroids for the clusters by taking the arithmetic mean of all data points that belong to each cluster.

- **DBSCAN:** The use of K-Means clustering has some limitations. First, it requires the user to set the number of clusters a priori. Second, the presence of outliers have undeniable impact on this algorithm. Moreover, the shape of the underlying clusters is already defined implicitly by the similarity function in K-Means, and K-Means performs better when the clusters are spherical. Figure 2.5 reveals that the K-Means algorithm might not be able to effectively cluster such a dataset, since it has been designed to discover spherical clusters. In this case, density-based algorithms could be more beneficial.

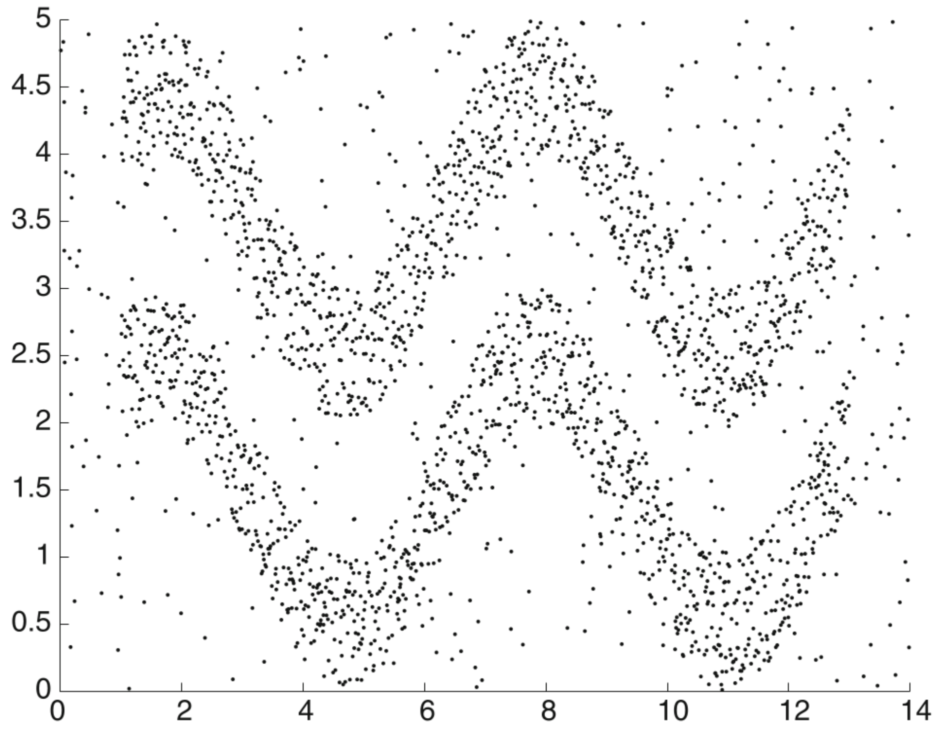


Figure 2.5 An arbitary shape dataset [3]

DBSCAN [70] is a density-based algorithm in which the individual data points in dense regions are used as building blocks after grouping them according to their density. The density of a point is the number of points that lie within a radius ϵ of that point, which can be obtained by the following formula:

$$N_\epsilon (p) = \{q \in D \mid dist\,(p, q) \leq \epsilon\} \tag{2.3}$$

DBSCAN classifies the data points into three categories of core, border, or outliers, based on parameters ϵ and $MinPts$. A point p is a core if at least $MinPts$ points are within the distance of ϵ (i.e., $N_\epsilon(p) \geq MinPts$). A data point is defined as a border point if $N_\epsilon(p)$ is less than $MinPts$, but it contains at least one core point within a radius ϵ. Otherwise, p is considered as noise and is assigned to the noise cluster.

This algorithm creates a connectivity graph in which core points are connected if they are within a distance ϵ from one another. Then, all connected components are identified, where these segments correspond to the clusters constructed on the core points. The border points are then assigned to the connected component with which it is best connected. The resulting groups are considered as final clusters, and noise points are reported as outliers. Thus, DBSCAN is able to cluster points into distinct categories, without setting the number of clusters in advance.

2.2 Related works

Many efforts have been made to improve anomaly detection tools and methods in recent years. However, to the best of our knowledge, each of these tools and methods has drawbacks. This section reviews most of the available academic and commercial approaches for performance anomaly detection.

Many sources may cause anomalies or performance degradation, such as application bugs, updates, software ageing phenomenon, and hardware failure. Many articles have attempted to discover or resolve performance degradations caused by each of these sources. As an example, software rejuvenation was introduced to prevent or at least delay ageing-related failures [71]. Software ageing has been demonstrated to affect many long-running systems, such as web servers, operating systems, and cloud applications. Ficco et al. [72] have also examined the effects of software ageing on the gradual increase in the failure rate or performance degradation of Apache Storm over time. A variety of bottleneck conditions, including system overload, and resource exhaustion, can also cause extended and intermittent system downtime. A number of global web services, including Yahoo Mail, Amazon Web Services, Google, LinkedIn, and Facebook, have recently suffered from such failures [73]. This problem has been addressed in the anomaly detection and bottleneck identification approach introduced in [73]. However, only a few examples of several possible sources of anomalies have been examined by researchers. Relying on the definition of anomaly, we believe that whatever the source of the anomaly is, it makes the execution flow different from the normal situation. Hence, it seems interesting to look at the problem from a more general point of view and try to find the deviations of the execution's flow, regardless of the source of the anomaly.

The first general aspect of an anomaly detection tool discussed in this section is the input. As mentioned earlier in this chapter, researchers consider different information such as resource utilization and tracing events as input to their analysis tool [74–76]. System resources include physical components such as the CPU, memory, disks, caches, and network. The resource utilization of an application is a metric that indicates the amount of capacity used, according to the available capacity. For example, memory utilization measures the amount of storage capacity consumed by a process or application. We categorize the approaches that use these metrics as metric-based approaches. In the metric-based approaches, the coarse system-level metrics are collected and treat the system as a black box [77,78]. Then, with the help of statistical machine learning methods, performance abnormalities are identified. However, these system metrics require precise program encapsulation. Ravichandiran et al. [79] proposed a traditional approach to identify system performance anomalies through analyzing the correlations in the resource usage data. They use application performance management (APM) tools that support various measures, to perform resource behavior analysis on microservices. Log-based approaches extract features from the logs to be used by machine learning and statistical techniques to detect abnormal behaviours. Wei Xu et al. parse logs to create composite features, and then analyze the features using machine learning to detect operational problems in large-scale data center services [80]. In [32], an unstructured log analysis technique for anomalies detection was proposed. In this technique, after converting log messages to log keys, a Finite State Automaton learns from training log sequences to provide the normal workflow for each system component. However, logging is more appropriate for high-level analysis of less frequent events, and gathering low-level information and time accurate details using logging is extremely difficult.

Tracing is another robust and efficient approach for reverse engineering and debugging of complex systems [73]. Many tracers across all software stack layers, and even at the hardware level, have emerged in the last years. Then, statistical methods, machine learning, or just simple statistics techniques are adopted to diagnose performance anomalies. PerfScope [81] is a tracing based performance bug inference tool to help the developer understand how a performance bug happened during a production run. Distributed tracing tools are another family of tracing tools that can be used in this field of research. Unlike the most traditional methods that monitor individual components of the architecture, distributed tracing is applied to complex distributed systems at the workflow level [82]. Tools like OpenCensus and OpenTracing [42] help to record the execution path of each microservice request. Jaeger [46], a popular tool that supports OpenTracing and developed by Uber, has been widely used to collect and store the service call data automatically. The method proposed in [47] uses Jaeger and dynamic instrumentation to collect execution traces across microservices. Next, this method calculates the anomaly degree of traces based on the tree edit distance, to find structural anomalies and then analyzes the difference between traces to determine the components responsible for the anomalies. Sage [48] is another machine learning-driven root cause analysis system for interactive cloud microservices. Sage benefits from Jaeger and an unsupervised ML model to capture the impact of dependencies between microservices and determine the root cause of unpredictable performance problems. In addition to Jaeger, its counterpart Zipkin [43] aids in gathering timing data needed to troubleshoot latency problems in microservice architectures. J.Cardoso et al. [44] proposed an anomaly detection method based on Zipkin, which uses a single modality of the data, with information about the trace structure to detect anomalies. Besides this work, the method proposed by Tao Wang et al. [45] characterizes Zipkin traces with calling trees and then learns trace patterns as baselines. It calculates the anomaly degree of the workflows impacted by faults in processing requests. It then locates the microservices causing anomalies by comparing current traces and learned baselines with tree edit distance. However, resource utilization measurements, logging and the mentioned tracing tools provide high-level information, which is not always sufficient to characterize the execution status of the system. Thus, tracing with LTTng is a fundamental part of our research. This open-source tool is implemented for achieving high throughput and includes multiple modules for Linux kernel and userspace tracing, thereby imposing low overhead to the operating system.

After examining data collection methods in various researches, the next step is how to characterize the executing software. Researchers have made much effort to improve anomaly detection by using different data representations and information resources. The use of system calls has led to dramatic advancements in anomaly detection techniques. Forrest et al. [6] showed that during the normal execution of a program, a consistent sequence of system calls is generated. In their method, all possible normal patterns of different lengths are collected to form the normal dataset. Then, different patterns in the new trace are compared with the normal dataset, and any deviation from the normal model is considered as an anomaly. The first weakness of this method is that finding all the patterns with different lengths is extremely time-consuming, because a short trace file contains thousands of events. Furthermore, the resulting database is massive. It is notably time-consuming to compare a new pattern to the entire normal dataset. Canzanese et al. characterized system call traces using a bag-of-n-grams model, which represents a system call trace as a vector of system call n-gram frequencies [76]. In this regard, Kolosnjaji et al. [29] attempted to apply deep learning to model the malware system call sequences for malware classification. They constructed a neural network based on convolutions, in order to obtain the most desirable features of n-grams. A well-known issue with N-gram-based approaches is sparsity. Like many statistical models, the N-gram model is significantly dependent on the training data. Besides, the performance of the N-gram model varies with the change in the value of N. In [28], the Linux kernel system calls are extracted to construct weighted directed graphs.

This method, in which the graph-based representation is used for anomaly detection, suffers from the high cost of obtaining such graphs. Finding related system calls, out of thousands of events, requires a high computational power. Tracing data or logs, as the most popular information resource in microservice environments, can be represented in the form of an enumerated collection of events, sorted by their timestamps [83]. Different works make different uses of this structure. In DeepLog [49], a deep neural network model is proposed to model an unstructured system log as a natural language sequence. In [84], by performing time-series-based forecasting, anomalies on cyclic resource usage patterns are detected. In the sequel, graph representations of the events are obtained from this data and employed to detect critical nodes and design anti-patterns proactively. The authors of [85] designed and developed a simplified MSA application and applied different graph algorithms, and then assessed their benefits in MSA analysis. In another article, Tao Wang et al. [86] organized the trace information, collected by the OpenTracing tool, to characterize processing requests workflow across multiple microservice instances as a calling tree. The proposed approach converts the given trace into the spans and detects performance anomalies using the model of normal key patterns.

Another aspect of an anomaly detection tool is choosing the appropriate statistical or machine learning method. The earliest efforts for anomaly detection had used statistical methods [87]. These works keep the activity of subjects and generate profiles to represent their behaviour. Profiles include measures such as activity intensity measures, categorical measures, and ordinal measures. An anomaly score is computed using an abnormality function, as events are processed. The detection system generates an alert if the anomaly score is greater than a certain threshold. In [88], CPU performance and network performance metrics in master-slave and nested-container models are compared, to provide a benchmark analysis guidance for system designers. These basic statistical models have some disadvantages. Defining proper thresholds, which can balance the likelihood of false positives and false negatives, is very difficult to set. Besides, most statistical anomaly detection techniques require the assumption of a quasi-stationary process. However, this cannot be assumed for most data processed by anomaly detection systems. Furthermore, these tools do not provide any details about the application execution flow. Wang and Battiti [74] proposed another statistical method in which the distance between a vector and its reconstruction onto the reduced PCA subspace represents whether the vector is normal or abnormal. This method is limited to pre-determined anomalies, and is not able to detect novel types of anomalies, besides suffering from the problem of defining thresholds.

In addition to statistical methods, several machine learning-based schemes have been applied to detect anomalies in systems. They work based on the establishment of a model that allows the patterns to be categorized [89]. Bayesian networks can encode probabilistic relationships among variables of interest, thereby predicting the consequences of an event in the system [90]. Ye and Borror presented a cyber-attack detection technique through anomaly detection and discussed the robustness of this model [75]. They used a Markov chain model to profile event transitions in the normal operating condition of a computer system. Achieving high performance in their technique, to model the sequential ordering of the events, depends considerably on the quality of the data. This is because the Markov Chain technique is not robust to outliers, and performs better when the amount of noise in data is low [91]. These models have better performance for small datasets.

Among other machine learning approaches, clustering algorithms can detect abnormal behaviour without prior knowledge. Many clustering algorithms, such as k-means, k-medoids, EM Clustering, and Outlier detection algorithms, have been employed for anomaly detection. In [92], the k-Means clustering algorithm with the accompaniment of different dimensionality reduction modules (PCA, ICA, GA, and PSO) was used to separate time intervals of the traffic data into normal and abnormal groups. Apart from clustering methods, classification based anomaly detection approaches like support vectors, Fuzzy Logic, and Neural Networks have been widely used in this area [93]. In [3], a fuzzy technique was proposed to extract abnormal patterns based on various statistical metrics, in which fuzzy logic rules are applied to classify data. However, in practice, the labelling process is highly complicated and even impossible sometimes. Recently, deep learning techniques which do not need labelled data have yielded promising results in different fields. Wang and Zhou [94] explained the potential of using deep learning techniques in side-channel signal analysis and cyber-attack detection. They exploited these signals to indicate the state of ongoing computational tasks without direct access to the device. They examined the application of deep learning methods to side-channel analysis in the classification of machine state and anomaly detection. Long Short-Term Memory (LSTM) neural networks is a deep learning method used in [95, 96] to detect anomalies in cloud infrastructures. Malhotra et al. [97] also presented a stacked LSTM model for anomaly detection in time series, where the network is trained on non-anomalous data. The drawback of these methods is that many details, including events arguments such as event type, tag, process name, etc., are ignored. Along with the aforementioned machine learning approaches, ensemble approaches are applicable in cases where a single model is incapable of distinguishing anomalies precisely [98].

2.3 Discussion

In this section, while reviewing the shortcomings of previous methods, we explore how our proposed methods differ from the previous related literature.

Most previous works have not provided a solution for data collection, and used pre-existing datasets [99–101]. Some other works used logging approaches, that cannot gather low-level information and time accurate details about the system's behaviour [31, 49]. Some popular tools (e.g. Jaeger and Zipkin) that support OpenTracing were also utilized in some papers. However, the high-level information that these tools provide is not always sufficient to characterize the execution status of the system. Besides, they do not offer kernel level events [44, 45, 47, 48]. Hence, we defined our data collection module using LTTng and Trace Compass. LTTng provides details of the program's execution with higher resolution by presenting kernel and userspace events. Tracing also enables us to deeply examine the execution flow using Trace Compass.

In addition to the detailed information that LTTng provides, it has several features that make our proposed data collection module applicable in most Linux-based environments without much change. Other tracing tools can not be used in different environments and applications, for example Jaeger and Zipkin, are only dedicated to monitoring and troubleshooting microservices and distributed systems [43, 46]. Chapters 4 and 5 show how our data collection module is configured to collect information from monolithic applications. Furthermore, the LTTng relay daemon used in Chapter 6 helps us trace distributed systems and cloud environments. LTTng as the main component of our data collection module can be installed quickly and without any special settings in different environments. This makes our data collection module a practical way to collect data in the real world.

Another strength of the methods proposed in this thesis is the way we used the collected data. The use of system call sequences [29, 102] and frequencies [103] is common in anomaly detection, but timing information is ignored in these methods. N-gram-based approaches are other well-known approaches that heavily depend on the training data, as in many statistical models. Furthermore, the performance of the N-gram model changes with the value of N [76]. To address the shortcomings of previous work, we introduced a new approach in Chapter 4, that uses the durations of system calls to generate feature vectors. The duration of a system call in a window acts like the weighted frequency of that system call. The results of Chapter 4 demonstrate that the introduced data collection module, feature selection method, and classification model provide a practical approach to detecting performance anomalies.

In Chapter 5, we addressed the problem of the availability of labelled data. Collecting labelled data is very difficult due to the nature of the anomaly detection problem and requires a specialist who has knowledge of machine learning and performance analysis. Previous work in the literature does not offer a solution that covers a variety of scenarios, including when labelled data is available, when labelled data is not available, and when a part of the dataset is labelled [29, 69, 70, 94, 100, 104]. We also addressed the problem of the sparsity of the dataset in this chapter. Other approaches, such as n-gram-based approaches [29], suffer from this problem. To solve the sparsity problem in the dataset, we proposed a feature selection method in which the less important system calls are removed from the feature vectors. The results show that this significantly increases the efficiency of the anomaly detection model.

Chapter 6 proposes an anomaly detection approach for microservice. This approach uses LTTng for data collection, which provides more details than Jaeger and Zipkin used in other works [44,45,47,48]. To detect anomalies in microservices, because the interactions between services have a graph structure, researchers either directly analyze the graphs or convert them into sequences. However, the graph-based approaches require high computational power [28] and the sequence-based approaches ignore events parameters [6, 29, 49, 84, 90]. Event parameters such as process name, message, and event type contain beneficial details that increase detection quality. Chapter 6 presents our solution for solving the problem of high computational power needed by graph-based approaches. We also use events parameters to improve anomaly detection in microservices.

CHAPTER 3 RESEARCH METHODOLOGY

As discussed in Chapters 1 and 2, this work focuses on performance anomaly detection using tracing data and machine learning. This chapter presents an overall view of our methodology and describes the process of identifying issues, applicable cases, milestones, and final results in terms of articles. Given the nature of this research, the progress of the project can be visualized in the form of three major threads, as illustrated in Figure 3.1. In the first part of our research, we applied a supervised technique to find the abnormal windows of system calls. Our feature vectors consisted of the execution time of system calls, in addition to the frequency of each individual system call. Then, we proposed a new approach that addresses the problem of the availability of labelled data by offering several learning techniques. We suggested supervised, unsupervised, and semi-supervised techniques, considering the amount of available labelled training data. Afterward, we proposed a natural language processing (NLP) based approach to detect performance anomalies, besides locating release-over-release regressions in microservice environments. These works are described briefly in the following sub-sections. Next, the three articles resulting from this research are presented in Chapters 4, 5, and 6, respectively, in chronological order of their submissions.

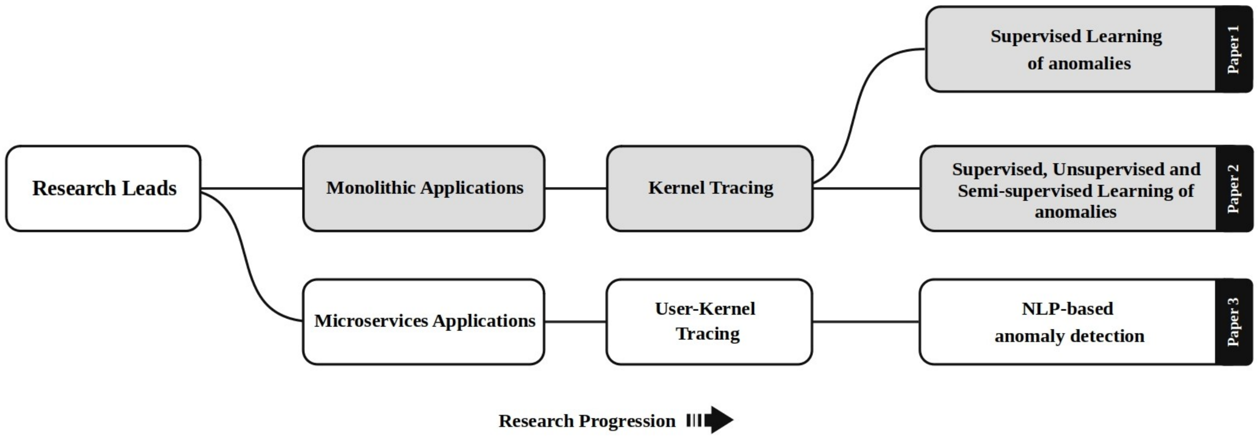

Research Leads
Monolithic Applications
Kernel Tracing
Supervised Learning of anomalies
Paper 1
Supervised, Unsupervised and Semi-supervised Learning of anomalies
Paper 2
Microservices Applications
User-Kernel Tracing
NLP-based anomaly detection
Paper 3
Research Progression

Figure 3.1 Milestones and research progress.

3.1 System performance anomaly detection using tracing data analysis

Chapter 4 presents an anomaly detection approach for practical monitoring of processes running on a system to detect abnormal vectors of system calls. The idea behind this work is that the system call trace obtained from an abnormal process is highly different from processes running under normal conditions. Our proposed approach employs the Linux Trace Toolkit (LTTng) to monitor the processes running on a system and extracts the streams of system calls. Then a sliding window is applied to continuously extract short system call sequences. Our proposed approach computes the execution time of system calls in addition to the frequency of each system call in subsequences. In other words, we define a compact representation for each subsequence that yields two separate feature vectors containing the frequency and duration of the system calls inside the current sliding window. Thus, our methodology can handle large and varying volumes of data. The length of feature vectors is equal to the total number of Linux system calls. Finally, a multi-class support vector machine approach is applied to evaluate the system's performance and detect abnormal subsequences. A comprehensive experimental study on a real dataset collected using LTTng demonstrates that our proposed approach is able to distinguish normal subsequences from anomalous ones with CPU or memory related problems.

3.2 A framework for detecting system performance anomalies using tracing data analysis

The article presented in Chapter 5 is motivated by one of the most critical problems in machine learning: the availability of labelled data. Data labeling is a complex and time-consuming process that requires a highly knowledgeable expert in the field. Like the previous article, subsequences of system calls are sent to the machine learning module that reveals anomalous subsequences. However, this time, we followed three distinct approaches depending on the amount of available labelled data. In the case of supervised learning, the Fisher score feature selection, along with a correlation filtering strategy are applied to determine the best subset of features in the dataset. Once the top-ranked features are selected, we employ a multi-class Support Vector Machine model. When labelled data is not available, our proposal uses the DBSCAN algorithm to group feature vectors into different categories in terms of performance. Finally, we proposed a novel semi-supervised machine learning model that benefits from both supervised and unsupervised learning techniques, when only a few labelled data are available.

3.3 Anomaly detection in microservice environments using distributed tracing data analysis and NLP

Chapter 6 investigates the use of NLP to find abnormal behaviors in microservice-based environments. Several factors such as the distribution of microservices in the network, the use of different technologies, and their short life make microservices prone to the occurrence of abnormal behaviors. This chapter proposes a natural language processing (NLP) based approach to detect performance anomalies in spans during a given trace. One of the benefits of this approach is that the whole system needs no prior knowledge, which facilitates data collection. We developed a handcrafted data extraction module in Trace Compass to construct the spans and sub spans using the request/response events tag. This module is also responsible for converting each span into a sequence of events. Then our LSTM-based model learns the possible patterns of events along with their arguments (e.g., event type, tag, and process name). Using events arguments, and learning this level of detail, sets our model apart from the others found in the literature.

CHAPTER 4 ARTICLE 1: SYSTEM PERFORMANCE ANOMALY DETECTION USING TRACING DATA ANALYSIS

Authors: Iman Kohyarnejadfard, Daniel Aloise, Mahsa Shakeri
Department of Computer and Software Engineering, Ecole Polytechnique de Montreal, Montreal, H3T 1J4, Canada

Published in: Association for Computing Machinery, New York, NY, United States.

Reference as: I. Kohyarnejadfard, D. Aloise, and M. Shakeri, "System performance anomaly detection using tracing data analysis," in Proceedings of the 2019 5th International Conference on Computer and Technology Applications, ICCTA 2019, (New York, NY, USA), pp. 169–173, ACM, 2019.

4.1 Abstract

In recent years, distributed systems have become increasingly complex as they grow in both scale and functionality. Such complexity makes these systems prone to performance anomalies. Efficient anomaly detection frameworks enable rapid recovery mechanisms to increase the system's reliability. In this paper, we present an anomaly detection approach for practical monitoring of processes running on a system to detect anomalous vectors of system calls. Our proposed methodology employs a Linux tracing toolkit (LTTng) to monitor the processes running on a system and extracts the streams of system calls. The system calls streams are split into short sequences using a sliding window strategy. Unlike previous studies, our proposed approach computes the execution time of system calls in addition to the frequency of each individual call in a window. Finally, a multi-class support vector machine approach is applied to evaluate the performance of the system and detect the anomalous sequences. A comprehensive experimental study on a real dataset collected using LTTng demonstrates that our proposed method is able to distinguish normal sequences from anomalous ones with CPU or memory related problems.

Keywords: Anomaly detection, Machine learning, Performance evaluation, Data mining, Time series, Linux Tracing

4.2 Introduction

Recently, the usage of distributed systems, like cloud computing infrastructures, enterprise data centers, and massive data processing systems, are rapidly increasing. The complexity of these systems makes them prone to performance anomalies. System performance degradation could be caused by various reasons such as excessive load of an application on resources or system misconfiguration. It is a tedious task for human administrators to manually monitor the execution status of the systems. Therefore, it is imperative to develop automatic anomaly detection approaches with a minimum human intervention.

An anomaly is a change in system performance that does not fit with the expected normal behavior. It is usually difficult to distinguish between normal and abnormal system status. Moreover, modeling the normal behavior of a system that enfolds every possible normal situations can be very complex. In this regard, behavioral analysis techniques could be used to monitor performance of the processes running on a system. These approaches use the characteristics of the executing software to identify potential anomalies in a system. One such technique is system call analysis, in which abnormal behaviors are identified by system call traces. System calls are requests for services, such as memory and filesystem access, that a process makes of the operating system. System calls can represent low-level interactions between a process and the kernel in the system.

Some studies apply time-delay embedding (tide) which records normal executions of applications using look-ahead pairs [6]. At test time, any deviation from the normal model is considered as anomaly. Since a single process could produce massive amount of system calls per second, these approaches do not scale well. Therefore, other studies split the system calls streams into short sequences and extract features over a fixed time frame window [105]. One approach extracts the histogram of the system call types for anomaly detection [103]. Kang et al. [104] utilize a bag of system calls for the intrusion detection and describe the misuses with the standard machine learning techniques. Other approaches rely on computing the frequencies of short sequences (n-grams) of system calls over a fixed time window [76].

In general, the studies that are only based on modeling the normal behavior, have the disadvantages of missing some normal system call patterns. Therefore, in situations where enough anomalous patterns are available, including the anomalous samples in the learning procedure might be useful. In addition, the available performance anomaly detection approaches do not consider the latency of a system call. A process can be blocked in a system call for different reasons, like high overload on resources. Thus, considering the duration of system calls could improve the accuracy of the anomaly detection on a system.

In this paper, we propose a performance evaluation approach which uses short sequences of system calls along with their duration to identify a system's status, as normal or anomalous. In this work, we separately evaluate the performance of each individual computer system as a sub-component of the distributed architecture. Information about the individual system's status can guide the administrators in monitoring the performance of the whole distributed framework. In our method, each system is instrumented with a lightweight kernel tracing tool called the Linux Trace Toolkit Next Generation (LTTng) [106]. The system calls trace logs are converted to short sequences using sliding window. Then, the system call tracing data are represented by the frequency of individual calls and their corresponding execution time. Finally, a supervised machine learning based approach is applied on the extracted features to discriminate between normal and anomalous system call sequences.

The rest of the paper is organized as follows. In subsection 4.3 our proposed system anomaly detection approach is presented. Subsection 4.4 provides the experimental results followed by the conclusions in subsection 4.5.

4.3 Methodology

The overview of the proposed anomaly detection method is presented in Figure 4.1. The proposed framework monitors the performance of a system by recording the stream of system calls produced by its processes. System calls are the fundamental interface between a process and the Linux kernel, which can be extracted using a Linux API. In our proposed approach, the system call stream is converted to a group of short sequences and then analyzed to detect the anomalous behavior of the system. An anomalous system call sequence may correspond to the following scenarios:

- The system is running a CPU intensive process which causes an insufficient CPU allocation problem
- The system is running a Memory intensive process which leads to an insufficient memory allocation problem.

The above problems could be modeled through a multi-class classification approach. Our proposed framework consists of multiple components: trace data extraction, pre-processing and normalization, feature selection, and anomaly detection. These steps are defined in more details in the following sections.

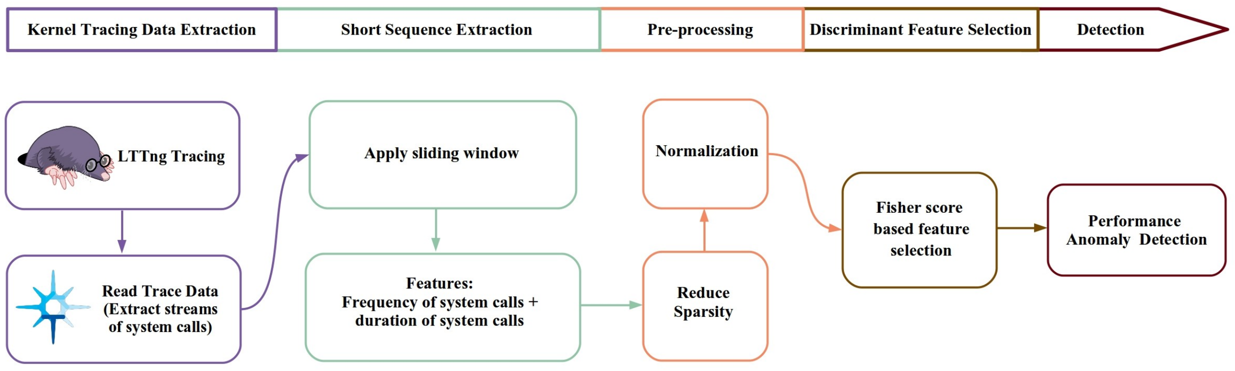

Kernel Tracing Data Extraction
Short Sequence Extraction
Pre-processing
Discriminant Feature Selection
Detection
LTTng Tracing
Read Trace Data (Extract streams of system calls)
Apply sliding window
Features: Frequency of system calls + duration of system calls
Normalization
Reduce Sparsity
Fisher score based feature selection
Performance Anomaly Detection

Figure 4.1 The overview of the proposed framework.

4.3.1 Kernel tracing data extraction

The Linux Trace Toolkit next generation (LTTng) [106] is used to gather information about the running processes on a system. LTTng is a powerful and lightweight open source Linux tracing tool which provides detailed information on the kernel and user-space executions. LTTng tool is used to collect system calls issued by the monitored processes and Trace Compass open source software [54] is utilized to read the LTTng trace logs into a dictionary of events. In this event dictionary, each system call entry consists of a timestamp, process ID, and other run-time information related to the running processes.

4.3.2 Short sequence extraction and pre-processing

In this work, we assign an index to each system call as shown in Figure 4.2. Once the stream of system calls data is extracted, the processes other than the one under study (e.g., MySQL) is filtered out by using the process ID field. Then, the system calls index, timestamp, and their corresponding execution times are listed for all threads of the selected process (see Figure 4.3).

In the next step, a sliding window is applied to continuously extract the short system call sequences. For each short sequence, two separate feature vectors are defined: 1) frequency of system calls x_{freq} and 2) duration of system calls x_{dur} where

- x_{freq} is a vector of size k, one per system call type. Each vector element counts the number of calls issued during the window time frame.
- x_{dur} is a vector of size k, one per system call type. Each vector component represents the duration of each system call in the window time frame.

Here, k is the total number of system calls that are included in the feature vector. In general around 318 different system calls exist in Linux system. However, depending on the type of the application running on the system, some system calls may not occur at all. This would result in a very sparse dataset. Therefore, here, we reduce the sparsity of the collected data by removing all unhappened system calls from the feature vectors.

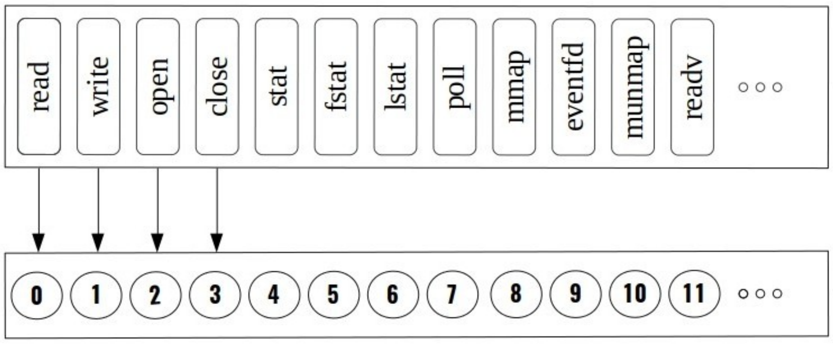


Figure 4.2 Each system call is assigned to a unique number.

As a normalization pre-processing step, data standardization is applied on the dataset. This process rescales the features in a way that they have the properties of a standard normal distribution with mean of 0 and standard deviation of 1.

4.3.3 Discriminant feature selection

In this step, Fisher score feature selection method [18] is applied to determine the most discriminative subset of features in the dataset. This algorithm computes a score for each feature and then selects the desired number of features according to their scores. The larger the Fisher score, the greater the discriminatory power of the attribute.

Given a dataset $\{(x_i, y_i)\}_{i=1}^{n}$, where $x_i \in \mathbb{R}^k$ is the input sequence vector and $y_i \in \{1, 2, ..., l\}$ is its corresponding class label, we aim to identify the most informative feature subset of size m. The most discriminative subset of features is determined in two steps. First, the Fisher score FS_j for the feature j is computed as follows:

$$FS_j = \frac{\sum_{c=1}^{l} n_c \left(\mu_c^j - \mu_j\right)^2}{\left(\sigma^j\right)^2} \tag{4.1}$$

where n_c represents the fraction of records in class c, μ_j and $\sigma^j = \sum_{c=1}^{l} n_c \left(\sigma_c^j\right)^2$ are the mean and the standard deviations of the entire dataset corresponding to the j-th feature, respectively. Here, μ_c^j and σ_c^j denote the mean and the standard deviations of the c-th class corresponding to the feature j.

Once the Fisher score for all features are computed, the top-m ranked features with high scores are selected as the most discriminative ones.

4.3.4 System performance anomaly detection

Once the system call feature vectors are collected per sequence, the monitoring framework categorizes the system performance into $l = 3$ separate classes: CPU problem, memory problem, and normal. Here, we apply multi-class support vector machine classification algorithm [63].

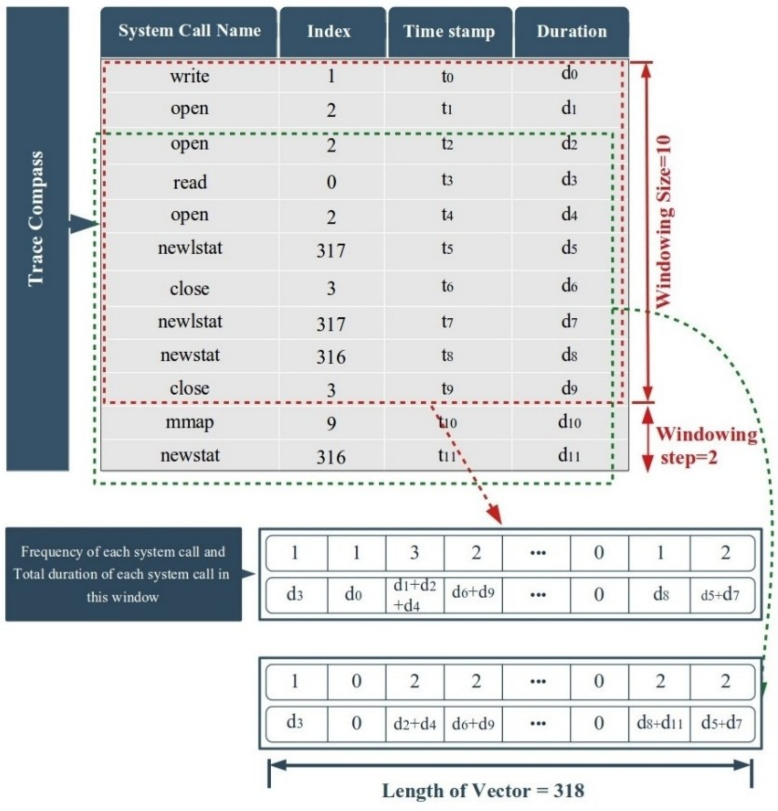


Figure 4.3 The summary of the extracted information from the stream of system calls.

Given the training data $\{(x_i, y_i)\}_{i=1}^n$ in the m dimensional space, the goal is to find the decision boundaries that can separate each training data vector of one specific class from that of others. We apply one-versus-one multi-class strategy which trains all possible pairwise classifiers. A test example is labelled to the class with the most votes.

For each data point x_i each binary hyperplane can be defined as following:

$$\omega^T \phi(x) + b \tag{4.2}$$

The term ω is normal to the hyperplane and b is the bias. The function $\phi(x)$, represents the projected input data x into a non-linear high-dimensional space. The support vector technique requires to solve the following optimization problem:

$$\min_{\omega,b,\varepsilon} \frac{1}{2}\omega^T\omega + C\sum_{i=1}^{n}\varepsilon_i \tag{4.3}$$

under the following constraints:
$y_i \left(\omega^T\phi(x_i) + b\right) \geq 1 - \varepsilon_i$
$\varepsilon_i \geq 0, i = 1, ..., n$

Here, $y_i \in \{1, -1\}^n$ where 1 is the positive and -1 is the negative class. The constant C controls the magnitude of the penalty associated with the training samples that lie on the wrong side of the decision boundary. The radial basis function (RBF) of $\phi(x) = e^{\gamma\|(x-x_i)\|^2}$ is applied to map the data into the non-linear high-dimensional space. The term γ is the parameter controlling the width of the Gaussian kernel. The accuracy of the classification is dependent on the value of the parameters C and γ.

4.4 Experiments

We evaluate our proposed anomaly detection approach on a real system performance anomaly dataset generated based on different faults. First, our experimental setup and dataset generation is described in subsection 4.4.1. Then, the result of the performance anomaly detection approach is presented in subsection 4.4.2.

4.4.1 Setup and dataset generation

We conducted our experiments on a group of virtual machines (VMs), in order to better manage system resource allocation. The host machine had an Intel Core i7 4 GHz $\times$ 8 CPU and 32 GB of memory. The VMs had different CPU cores and memory allocations depending on the workload simulation. We use the open source MySQL synthetic benchmarks tool, Sysbench, with oltp (Online Transaction Processing) test in complex mode. In this work, different faults are simulated on the VMs to generate the performance anomaly dataset. Examples of the simulated anomalies on the VMs are listed bellow:

- *CPU problem*: Limiting the amount of CPU resources allocated to a VM. (e.g., 1 CPU core, while running 8 threads of MySQL).
- *Memory problem*: Limiting the amount of memory resources assigned to a VM. (e.g., 256 MB memory, while the MySQL table is of size 6 GB).

In total, the generated dataset includes 18k normal and anomalous samples. Moreover, the classes (normal, CPU problem, and memory problem) contain the same number of samples which leads to a balanced dataset.

4.4.2 Results

In this section, we evaluate the performance of the proposed anomaly detection approach. The proposed method records a stream of system calls and then extracts the short sequences using a sliding window strategy. The window size is chosen as 10k with the overlapping size of 100. In the next step, duration and frequency based features are extracted from the sequences of system calls. After applying pre-processing and Fisher score method a multi-class SVM approach is applied to distinguish between normal and anomalous system.

To validate the accuracy of the proposed methodology, a 10-fold cross-validation strategy is used which randomly partitions our dataset into 10 equal size subsets. Therefore, the classifier is trained on 90% of the samples and the remaining data is split into two folds as validation and test set. The whole process is repeated 10 times for an unbiased evaluation.

As mentioned previously, Fisher scores are computed on the vector of system calls to determine the most informative features. Figure 4.4 and 4.5 illustrate the Fisher values computed for each system call in both frequency and duration based feature spaces. As expected, some system calls have high Fisher scores and therefore play an effective role in discrimination between classes. To better show the impact of computing the Fisher scores, Figure 4.6 shows the accuracy of anomaly detection approach on the validation set using different numbers of selected features. This experiment reveals that $m = 22$ and $m = 6$ top score system calls should be selected for the frequency and duration based approaches, respectively.

As mentioned in subection 4.3.4, we apply a multi-class SVM with RBF kernel on our dataset to classify the input sequences into three classes: normal, CPU problem, and memory prob-

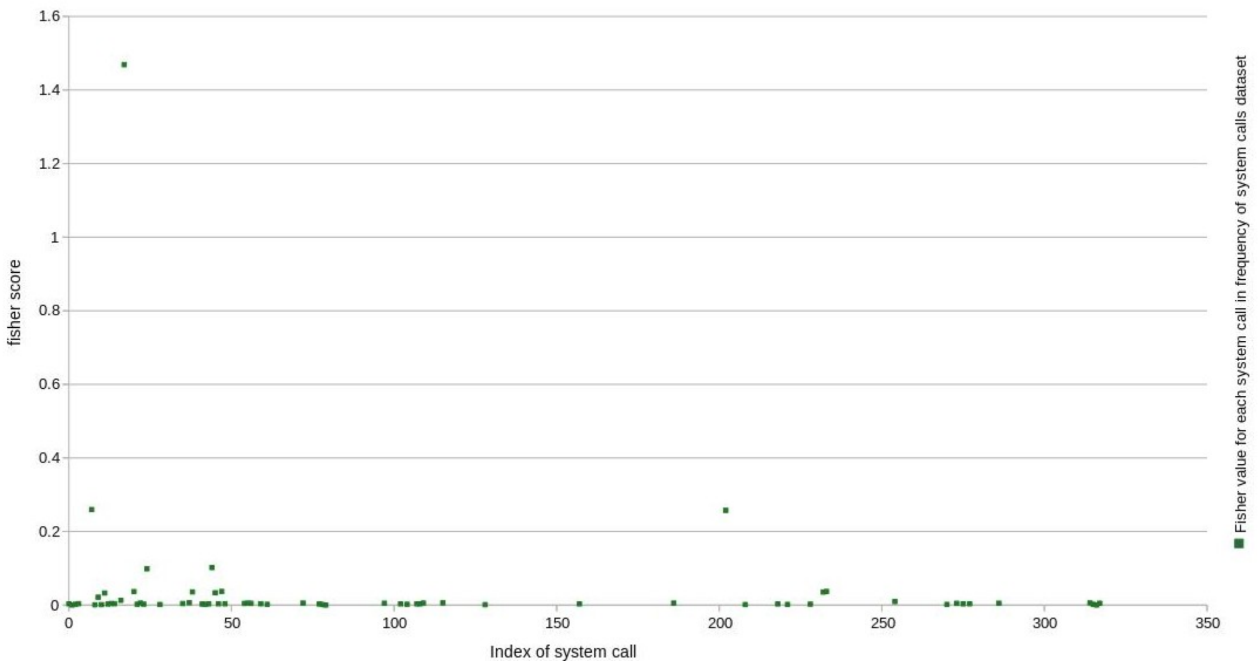


Figure 4.4 The Fisher score for each system call in frequency-based approach.

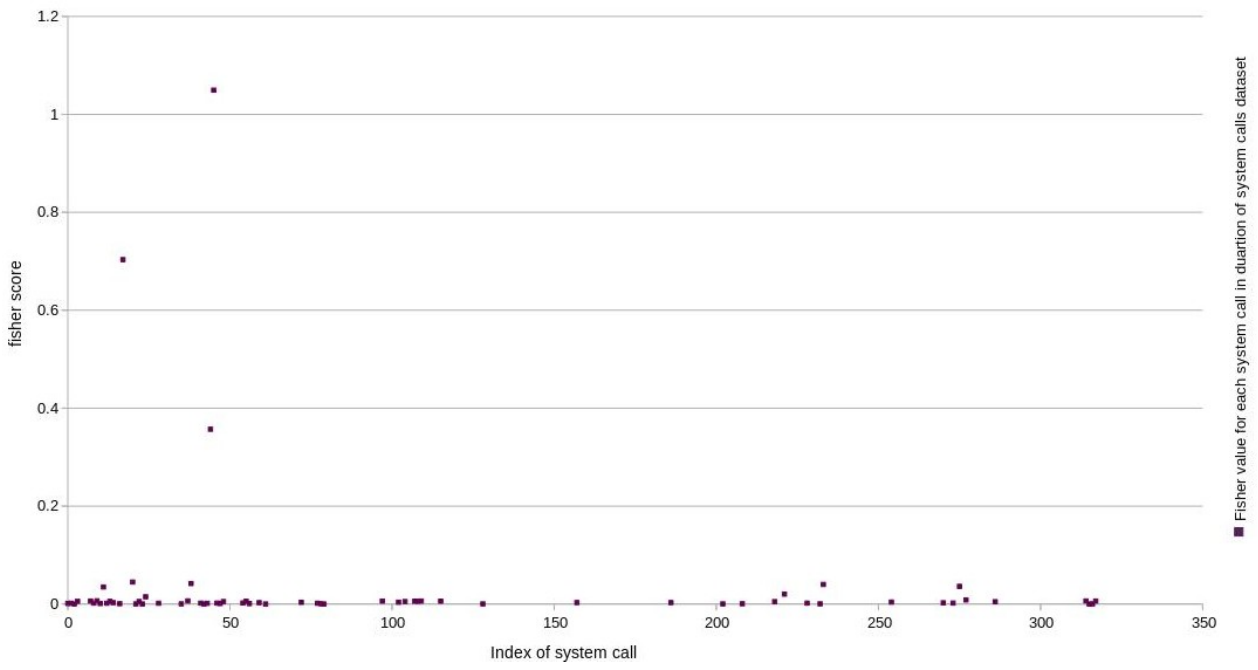


Figure 4.5 The Fisher score for each system call in duration-based approach.

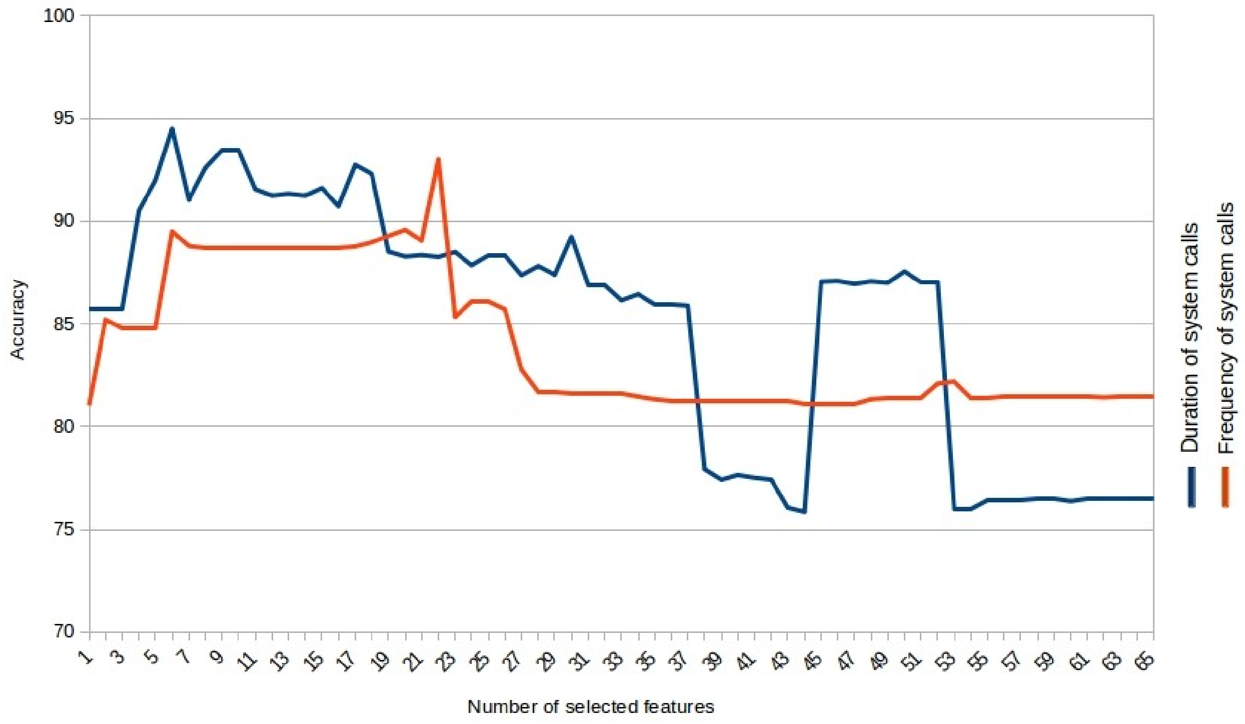


Figure 4.6 Anomaly detection accuracy versus different number of top-ranked features.

lem. The accuracy of the classification method can be optimized by varying the two parameters, C and γ. The parameter C is the regularization term which controls the maximum penalty imposed on the margin for the miss-classified points. The regularization parameter C trades off between the training error and margin maximization. The term γ which is involved in the RBF Kernel, is a distance measure that defines the training points' influence on the hyper-plane. In order to optimize these parameters, a grid search algorithm is performed. Figures 4.7 and 4.8 show the average accuracy of the validation set using different combination of parameters. The pair with the highest accuracy is chosen as final optimal one. According to these analysis, the pairs $(C = 10000, \gamma = 0.1)$ and $(C = 100000, \gamma = 0.1)$ are selected for the frequency and duration based approaches, respectively .

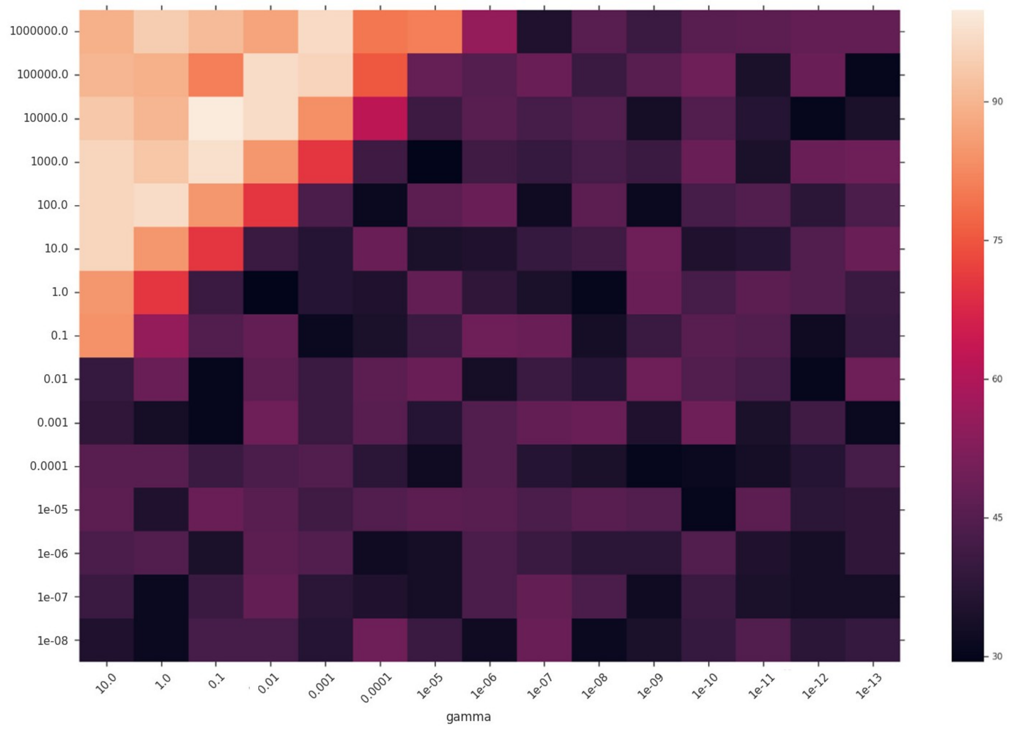


Figure 4.7 Heat map of the duration-based anomaly detection accuracy using different parameters γ and C.

Given these optimal parameters, our proposed anomaly detection method are evaluated on an unseen test dataset. The result of the three class classification accuracy for 10 multiple runs is presented in Figure 4.9. This shows that both frequency and duration of system calls can accurately perform multi-class anomaly detection. However, duration-based approach produced more accurate detection compared to the frequency-based approach.

In another experiment, the average classification accuracy of the proposed RBF-SVM anomaly detection framework is compared to SVM with other kernels (Sigmoid and polynomial). Ta-

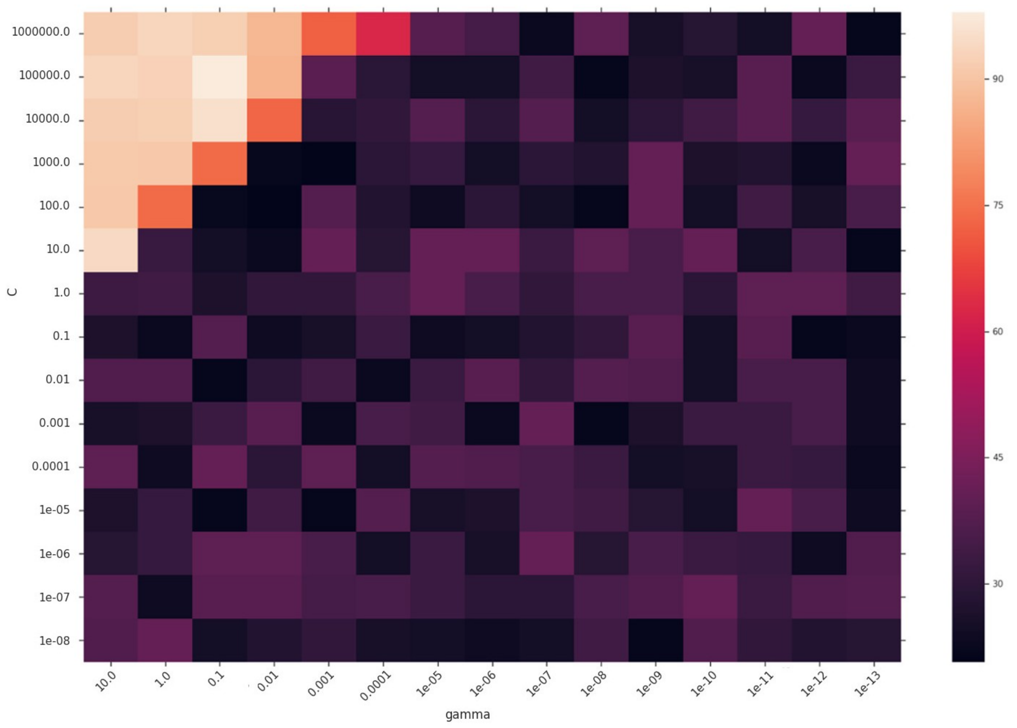


Figure 4.8 Heat map of the frequency-based anomaly detection accuracy using different parameters γ and C.

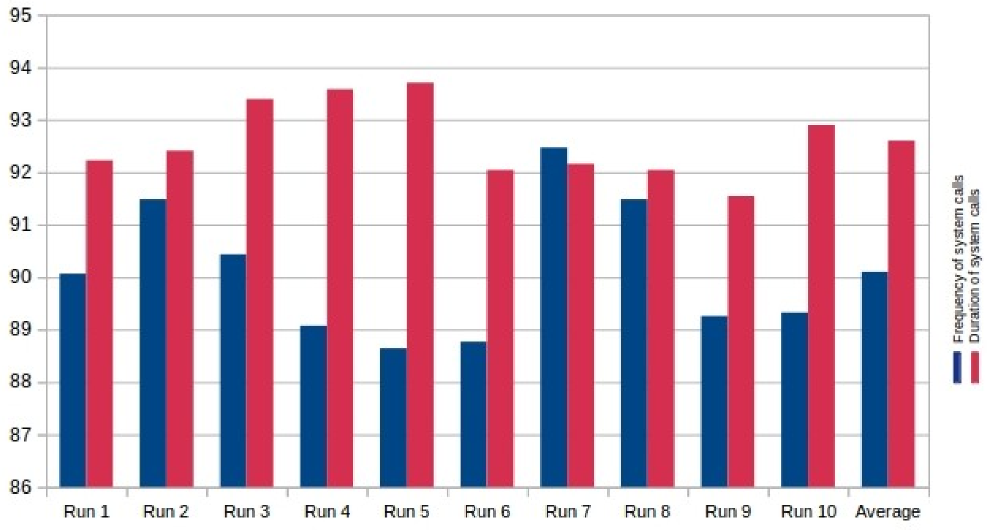


Figure 4.9 Accuracy of the proposed approach on multiple runs.

Table 4.1 The performance of the proposed RBF based anomaly detection approach compared to the Sigmoid (SIG) and polynomial (POLY) based methods. The performances are reported on both duration and frequency of system calls.

	Duration			**Frequency**		
	Acc	**Prec**	**Rec**	**Acc**	**Prec**	**Rec**
RBF	**0.925**	**0.848**	**0.806**	**0.900**	0.857	**0.911**
SIG	0.906	0.796	0.660	0.886	0.884	0.879
POLY	0.911	0.810	0.674	0.882	**0.913**	0.744

ble 4.1 reports the average accuracy, precision, and recall for these approaches. It should be noted that in the field of anomaly detection, recall usually plays a more effective role than precision. These results show that the proposed method based on RBF kernel outperforms the two other approaches.

4.5 Conclusions

In this paper, a system performance anomaly detection approach is proposed. The proposed method records the stream of system calls using the Linux kernel tracing. Then, the short sequences of system calls are extracted and two feature vectors of duration and frequency are created. Fisher Score method is applied to select the most discriminative features and a three-class SVM algorithm is employed to distinguish among systems with normal behavior, CPU shortage, and memory shortage. Experiments showed that our proposed method was able to produce promising results. Future work includes incorporating other system call parameters as features.

CHAPTER 5 ARTICLE 2: A FRAMEWORK FOR DETECTING SYSTEM PERFORMANCE ANOMALIES USING TRACING DATA ANALYSIS

Authors: Iman Kohyarnejadfard, Daniel Aloise, Michel R. Dagenais and Mahsa Shakeri
Department of Computer and Software Engineering, Ecole Polytechnique de Montreal, Montreal, H3T 1J4, Canada

Published in: Entropy

Reference as: Kohyarnejadfard, Iman, Daniel Aloise, Michel R. Dagenais, and Mahsa Shakeri. "A Framework for Detecting System Performance Anomalies Using Tracing Data Analysis." Entropy 23, no. 8 (2021): 1011.

5.1 Abstract

Advances in technology and computing power have led to the emergence of complex and large-scale software architectures in recent years. However, they are prone to performance anomalies due to various reasons, including software bugs, hardware failures, and resource contentions. Performance metrics represent the average load on the system and do not help discover the cause of the problem if abnormal behavior occurs during software execution. Consequently, system experts have to examine a massive amount of low-level tracing data to determine the cause of a performance issue. In this work, we propose an anomaly detection framework that reduces troubleshooting time, besides guiding developers to discover performance problems by highlighting anomalous parts in trace data. Our framework works by collecting streams of system calls during the execution of a process using the Linux Trace Toolkit Next Generation(LTTng), sending them to a machine learning module that reveals anomalous subsequences of system calls based on their execution times and frequency. Extensive experiments on real datasets from two different applications (e.g., MySQL and Chrome), for varying scenarios in terms of available labelled data, demonstrate the effectiveness of our approach to distinguish normal sequences from abnormal ones.

Keywords: Anomaly detection, Machine learning, Performance evaluation, Operating System, Tracing

5.2 Introduction

In recent years, computing infrastructure has significantly evolved, whereas complex systems have facilitated many complicated and large-scale tasks. For example, functional co-processing units accommodate conventional processing units to speed up particular tasks such as virtualization or complex machine learning computations. Consequently, a simple operation can involve multiple parallel cores, being served in a few seconds or milliseconds. These improvements have increased the expectation level of the users, so that any performance fluctuations or increased latency may lead to user dissatisfaction and financial loss. Different reasons such as software bugs, misconfigurations, network disconnection, hardware faults, aging phenomena of the systems, or even extreme load injected by other applications into the system, may degrade the performance of a particular service or application. Hence, monitoring and analyzing the performance of applications to find any performance anomaly or degradation is of particular importance. Indeed, any delay in detecting performance problems and troubleshooting can significantly increase the cost to fix them.

Performance anomaly detection refers to the problem of finding exceptional patterns in execution flow that do not conform to the expected normal behavior. Many sources may cause performance anomalies, such as application bugs, updates, software aging phenomenon, and hardware failure. It should be noted that performance anomalies are different from high resource consumption. An application might be inherently CPU or I/O intensive without being categorized as anomalous. However, imposing a continuous and more than expected average workload intensity on the system can be a sign of an anomaly. Relying on the definition of performance anomaly detection, we believe that whatever the source of the anomaly is, it makes the execution's flow different from the normal situation. Consequently, it seems interesting to look at the problem from a more general point of view and try to find the deviations of the execution's flow, regardless of the source of the anomaly. In case of any abnormal behavior during software execution, system developers or experts need information that not only locates that behavior but also provides details of the execution at the time the anomaly occurs. The performance metrics provided by tools such as top, etc., can represent the average load on the system. However, they do not help detect anomalies since a live threshold subject to the current system state would be needed to distinguish whether the application's behavior is normal or abnormal, which is practically impossible. Even if such thresholds were available, these tools would not provide any details about the application's execution flow. Therefore, system experts often employ logs and low-level tracing tools to define efficient strategies to find anomalies as well as their causes. Tracing is an effective way of gaining information on a system for further analysis and debugging of the underlying system, while minimizing monitoring influence [107]. However, it is an exhausting responsibility for human administrators to manually examine a massive amount of low-level tracing data and monitor the execution status of an application [108]. Hence, an accurate anomaly detection framework with minimum human intervention is in order.

Tracing data can offer detailed information about the execution of applications and processes. System calls are essential traceable events that contain valuable information about the program's flow. They represent low-level interactions between a process and the kernel in the system. Processes must interact with the operating system for each request, such as opening a file, writing into the registry, or opening a network connection, which is done through system calls. A system call trace provides an ordered sequence of system calls that a process performs during its execution. The definition of normal behavior is stable for standard UNIX process [6]. When a process is anomalous, its system call trace is extremely different from the process running under normal conditions [102, 109]. Our goal in anomaly detection is to find sets of system calls that are not likely to happen together in normal situations.

This work proposes a general anomaly detection framework to process the large volume of tracing data by taking advantage of machine learning technologies and open-source tools (i.e., LTTng and Trace Compass). Its main contributions are the following: **First**, unlike many other methods that use performance metrics or unstructured logs, we employed LTTng for data collection, which provides a system software package for correlated tracing of the Linux kernel, applications, and libraries [110]. LTTng provides high-resolution details of the program's execution by presenting kernel and userspace events related to the moment anomalies occur. **Second**, this article has addressed the problem of availability of labelled data by proposing learning techniques depending on their volume. Consequently, when a large amount of labelled training data is available, a supervised method is introduced, whereas an unsupervised method is preferred when labelled data is not available. Moreover, we propose a novel semi-supervised machine learning model within proposed framework that benefits from both supervised and unsupervised learning techniques when only a few labelled data are available. It should be noted that all proposed learning methods use the same data structure. **Third**, this is the first time that the durations of the most important system calls are used to make feature vectors. The duration of a system call in a window acts like the weighted frequency of that system call. Further, using the most important system calls instead of the whole set of system calls is novel, and it is shown to improve detection performance. **Fourth**, the proposed anomaly detection framework reduces troubleshooting time and directs the developer or troubleshooter to discover the problem by highlighting the anomalous parts of the trace. It helps developers look at just a few small windows instead of the whole trace that contains millions of events. Using the proposed anomaly detection framework alongside Trace Compass gives the developers a deep understanding of what happened at the time of the anomaly. It enables developers to use many preexisting scripts and views in Trace Compass for further analyzing the anomaly detection output.

The rest of the paper is organized as follows. In Section 5.3, related studies are presented. In Section 5.4, we describe the details of the performance anomalies in processes. In Section 5.5, we introduce our automatic integrated anomaly detection framework. Section 5.6 discusses the algorithm for kernel tracing and data extraction. Preprocessing of the extracted data is explained in Section 5.7. Then, the feature selection strategy along with supervised, unsupervised, and semi-supervised anomaly detection methods are proposed in Section 5.8. Section 5.9 provides the experimental results from two different applications (i.e., MySQL and Chrome), followed by the conclusions in Section 5.10.

5.3 Previous Work

In this section, the available techniques for performance anomaly detection are reviewed. The earliest efforts consisted of statistical methods [87]. These works keep the activity of subjects and generate profiles to represent their behavior. Profiles include measures such as activity intensity measure, audit record distribution measure, categorical measures, and ordinal measure. As events are processed, an anomaly score is computed using an abnormality function and profiles. If the anomaly score is higher than a certain threshold, the detection system generates an alert. Statistical models have some disadvantages. Defining proper thresholds which can balance the likelihood of false positives and false negatives is very difficult to set. Moreover, most of the statistical anomaly detection techniques require the assumption of a quasi-stationary process. However, this cannot be assumed for most data processed by anomaly detection systems. Wang and Battiti [74] proposed a method in which the distance between a vector and its reconstruction onto the reduced PCA subspace represents whether the vector is normal or abnormal. This method is limited to pre-determined anomalies and is not able to detect novel types of anomalies, besides suffering from the problem of defining thresholds.

In addition to these methods, several machine learning-based schemes have been applied to detect anomalies in systems. They work based on the establishment of a model that allows the patterns to be categorized [89]. Bayesian networks can encode probabilistic relationships among variables of interest, thereby predicting the consequences of an event in the system [90]. Ye and Borror presented a cyber-attack detection technique through anomaly detection using a Markov chain [75]. Achieving high performance in these techniques depends on the quality of the data. This is because the Markov Chain technique is not robust to outliers and performs better when the amount of noise in data is low [91]. Besides, these models have better performance for small datasets. Among other approaches, clustering algorithms can detect abnormal behavior without prior knowledge. Many clustering algorithms, such as k-means, k-medoids, EM Clustering, and outlier detection algorithms, have been employed for anomaly detection. In [92], the k-Means clustering algorithm with the accompaniment of different dimensionality reduction modules (PCA, ICA, GA, and PSO) was used to separate time intervals of the traffic data into normal and anomalous groups. However, none of these works have mentioned how to collect the data. These works are limited to clustering preexisting datasets and do not provide a solution for real-world usage. Apart from clustering methods, classification-based anomaly detection approaches like support vectors, Fuzzy Logic, and Neural Networks have been widely used in this area [93]. In [3], a fuzzy technique is proposed to extract abnormal patterns based on various statistical metrics in which fuzzy logic rules are applied to classify data [3]. Statistical metrics cannot be used to find the root cause of the anomaly after detecting an anomaly because these metrics do not provide details of the execution flow.

One imperative point in system performance analysis is how to characterize the executing software. In this regard, behavioral analysis techniques can be used to automatically monitor the performance of the processes running on a system. Some other studies have used system calls to characterize software behavior. Forrest et al. [6] showed that during the normal execution of a program, a consistent sequence of system calls is generated. In their method, all possible normal patterns of different lengths are collected to form the normal dataset. Then different patterns in the new trace are compared with the normal dataset, and any deviation from the normal model is considered an anomaly. The first weakness of this method is that finding all the patterns with different lengths is extremely time-consuming because a short tracing file includes thousands of events. Furthermore, the resulting database is massive. It is notably time-consuming to compare a new pattern to the entire normal dataset.

The use of system calls has led to a dramatic improvement in anomaly detection techniques. Canzanese et al. characterized system call traces using a bag-of-n-grams model, which represents a system call trace as a vector of system call n-gram frequencies [76]. In this regard, Kolosnjaji et al. [29] attempted to apply deep learning to model the malware system call sequences for malware classification. They constructed a neural network based on convolutions in order to obtain the most desirable features of n-grams. A well-known issue with N-gram-based approaches is sparsity. The N-gram model, like many statistical models, is significantly dependent on the training data. Besides, the performance of the N-gram model varies with the change in the value of N. In [28], Strace is utilized for collecting logs, and then the Linux kernel system calls are extracted to construct weighted directed graphs. This method in which the graph-based representation is used for anomaly detection suffers from the high cost of obtaining such graphs. Finding related system calls out of thousands of events requires extremely high computational power.

Many sources may cause anomalies or performance degradation, such as application bugs, updates, software aging phenomenon, and hardware failure. Various articles have tried to discover or solve performance degradations resulting from each of these sources. For example, software rejuvenation was introduced to prevent or at least delay aging-related failures [71]. Software aging has been demonstrated to affect many long-running systems, such as web servers, operating systems, and cloud applications. Ficco et al. have examined the effects of software aging on the gradual increase in the failure rate or performance degradation of Apache Storm over time [72]. Apache Storm is an open-source distributed real-time computational system for processing data streams. These systems may be affected by software aging because they usually run for a very long time. In their work, the measures related to the system resources usage and the user-perceived performance are collected by vmstat utility and by reading from Storm logs details about emitted requests and their responses. This information is employed to discover evidence of software aging. However, software aging is just one of several sources of anomalies. Relying on the definition of anomaly, we believe that whatever the source of the anomaly is, it makes the execution's flow different from the normal situation. Hence, it seems interesting to look at the problem from a more general point of view and try to find the deviations of the execution's flow, regardless of the source of the anomaly. In addition, LTTng can gather kernel events as well as the userspace events without imposing much overhead to the system. LTTng has several features that make it usable for most Linux-based environments. For instance, the LTTng relay daemon enables us to trace distributed systems.

Our work distinguishes from the previous related literature since:

- Unlike most previous works, which did not provide a solution for data collection, we defined our data collection module using LTTng and Trace Compass. Using various LTTng features makes our proposed framework applicable in most Linux-based environments without much change. For instance, the LTTng relay daemon enables us to trace distributed systems and cloud environments. In addition, no special settings are employed while collecting the data. We used tracing in our proposed framework because tracing enables us to examine the execution flow using tools such as Trace Compass.

- Statistical metrics can not be used to find the cause of the anomaly after detecting an anomaly. Compared to statistical techniques, our proposed framework has no assumption and is not dependent on the existence of any threshold. This fact and the way we use the system calls increase the generality of our method and make it usable for any application and environment.

- Achieving high performance using Bayesian networks and Markov chain techniques depends on the quality of the data. These techniques are not robust to outliers and perform better when the amount of noise in data is low. Besides, these models have better performance for small datasets. These problems were solved in our work by carefully choosing the learning method so that the presence of noise or new data points does not cause much change in the model and works appropriately for large data.
- Many of the available performance anomaly detection approaches use supervised methods, which require labelled data. However, labelled data is not always available. While proposing an unsupervised approach is desirable, it is a great challenge to achieve high accuracy by means of an unsupervised method. We have provided a package of supervised, unsupervised, and semi-supervised methods that can be used according to the volume of available labelled data. All these three methods use the same data structure, and no special settings are employed while collecting the data. After training the model in our proposed method, the detection is done very quickly and without high computational cost.
- Unlike methods that compare a pattern to all normal patterns in a database to determine if it is abnormal, our method is not limited to a primary database. Finally, presenting events that occurred during the anomaly helps the developer not spend much time examining the entire events in a trace or log file in order to discover the anomaly's cause.

5.4 Performance Anomaly in Processes

Performance anomalies are the most significant obstacles to the system to perform confidently and predictably in enterprise applications. Many sources can cause anomalies, such as varying application load, application bugs, updates, and hardware failure. In a situation where the workload is the source of the anomaly, the application imposes continuous and more than expected average workload intensity to the system. Faults in system resources and components may considerably affect application performance at a high cost [24]. In addition, software bugs, operator errors, hardware faults, and security violations may cause system failures.

The preliminary performance profiling of a process that reflects its typical behavior can be done using synthetic workloads or benchmarks. At a higher level, the performance of computer systems is delineated by measuring the duration of performing a given set of tasks or the amount of system resources consumed within a time interval [25].

There exist many metrics for measuring the performance of a system. Latency and throughput are the most used ones. They are used to describe the operation state of a computer system. The time that passes between the beginning of an operation and its completion is the latency, (e.g., the delay between when a user clicks to open a webpage and when the browser displays that webpage). Throughput is a measure of how many jobs a system can perform in a given amount of time (e.g., the number of users' requests completed within a time interval). In addition, resource utilization of an application indicates the amount of resources (e.g., number of CPUs, and the size of physical memory or disk) used by that application. The CPU utilization is the percentage of time in which the CPU is executing a process whereas the memory utilization is the amount of storage capacity dedicated to a particular process.

Figure 5.1(a) shows an example of the CPU utilization of a process during its lifetime. When an application is running normally, the CPU used by that application is conventional. Hence an expected maximum CPU utilization threshold can be defined for each application. In this case, if the CPU usage exceeds the threshold value, the process behavior is prone to the existence of an anomaly. Furthermore, as represented in Figure 5.1b, during the anomalous running of a process, the latency is usually increased while this curve has a relatively steady trend during normal behavior [26].

From another perspective, data anomalies can be defined in various forms. Two principal forms of anomalies are point anomalies and collective anomalies. Point anomalies are data points that are different from normal data. For example, consider a situation where data is generated from different data distributions, each one defining a cluster. In this case, data points which do not seem to have been generated by the data distributions are considered as point anomalies. While searching this type of anomalies, performance metrics such as CPU utilization or throughput can be used to determine if abnormal behavior has occurred at a particular timestamp. In the case of collective anomalies, we cannot detect individual data points as anomalies by themselves; however, their collective occurrence together may be an anomalous behavior. In this method, instead of detecting an anomaly at a particular timestamp, the system's behavior during a sequence of events is investigated. Due to the use of tracing data and the fact that a single event obtained from tracing does not contain enough information to detect an anomaly, we search mostly for collective anomalies in this work. Besides, looking at the sequence of events provides insightful information about the system behaviors over a period of time, which is essential for analyzing the root cause of an

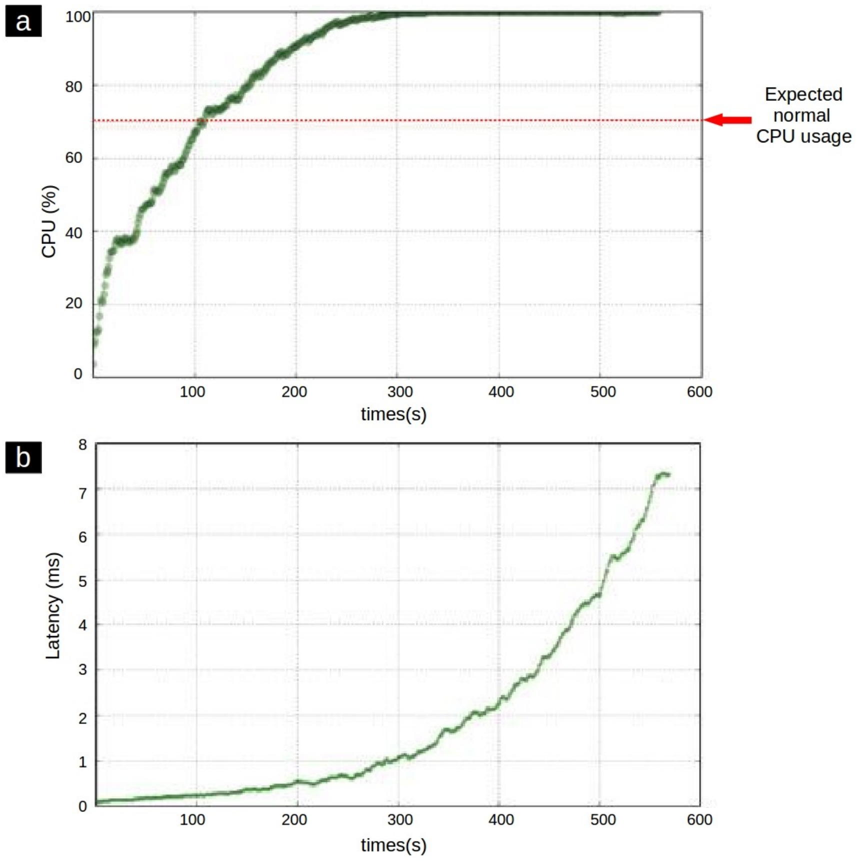


Figure 5.1 (**a**) CPU usage of an application during the time. (**b**) an anomalous latency growth pattern [1].

anomaly. Finally, by targeting the collective anomalies, our framework can even handle rare system call invocation paths. For example, a user never opens FTP connections on Chrome, but one day decides to do so. This will lead to a significantly different invocation path. In practice, observing a new system call is not a reason for an anomaly to occur, and we cannot consider a subsequence of events to be abnormal only due to the presence of a new system call. The subsequence in which the rare system call occurs is considered abnormal not only because of that system call but also because of the effects the system call has on the surrounding system calls. This system call may also be ignored during the feature selection process, in which case its effect is still present in the subsequence.

Anomalies can be defined from the user experience aspect, and in many situations, anomalies happen on the server-side, but their effect can be realized on the user side. Moreover, a physical or virtual node is often not dedicated to a unique particular service. So, latency or throughput in a sampling period cannot help to find anomalies in a program execution while several programs are running on the node. In this case, separating the normal and the abnormal behavior is very difficult, and the result depends on the hardware. Furthermore, the latency or throughput does not contain execution details, while the sequence of events such as system calls reveals many details about program execution.

5.5 The Automatic Integrated Anomaly Detection Framework

In this work, we propose an automatic anomaly detection framework to process the large volume of tracing data by taking advantage of machine learning technologies. The system architecture of the proposed framework is shown in Figure 5.2. The generality of the framework is extremely important, and it must be capable of working along with any program or system with different settings.

As illustrated in Figure 5.2, the entire framework is divided into several modules. First, kernel tracing is done to gather the system calls information during the execution of a program. We employ LTTng (Linux Trace Toolkit Next Generation) in this module, a system software package for correlated tracing of the Linux kernel, applications, and libraries. The raw tracing data is fed into the data extraction module that processes it with a windowing method that will be introduced later in this paper. Data transformation and feature extraction is done in the Trace Compass application. This module is responsible for preparing data for the detection module in both the training and detection phases. This data contains feature vectors extracted from the tracing data. When the model is trained, the data extraction module sends new feature vectors to the detection module at detection time. We discuss each module in the following sections.

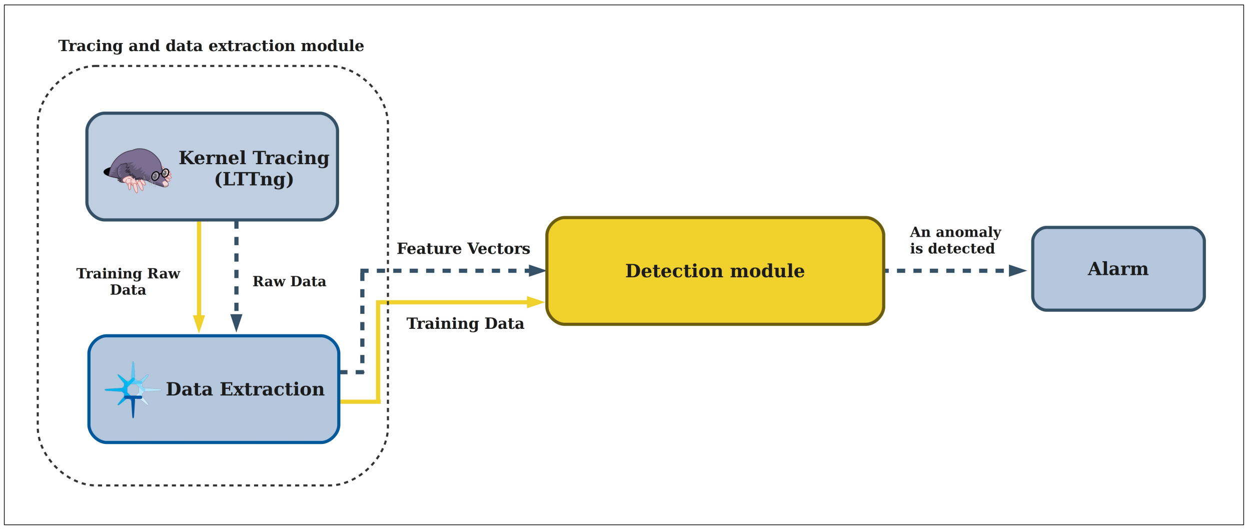


Figure 5.2 The system architecture of the proposed framework.

Our anomaly detection framework's design aims to provide high accuracy and time efficiency in analyzing tracing data and detect anomalous performance behaviors in large scale systems. By investigating the required framework specifications it seems challenging to apply an anomaly detection framework in practice because of two issues. The first is that continuously collecting system calls for machine learning methods is computationally expensive and of needs much storage space. Furthermore, the machine learning model itself takes a long time to train. To address the latter issue, we assume that once the model has been trained for an application, there is no need to retrain it, and periodic updates are enough. However, continuously collecting system calls is still needed. Hence, we propose using LTTng-rotate for increasing data collection efficiency, thus reducing the size of the tracing file.

5.6 Kernel Tracing and Data Extraction

In this section, the data extraction technique is explained in which two data sets are created by tracing the underlying system kernel by means of a sliding windowing technique. We use our data collection instead of using many existing systems calls data sets. Our data extraction technique allows us to collect our own fields (name, index, and especially duration) and trace any program. Furthermore, we can consider the time of data collection in the overall process because the data collection is not free as it is considered in many existing works, and finally, the collaborations required to handle a request can be considered, which is not the case in the existing data sets.

Tracing is a popular technique to analyze, debug, and monitor processes running on a system. Furthermore, it is an efficient way of gaining information on a system while minimizing the monitoring influence. The instrumentation of the traced application provides as output timestamp-matched events and intuition on the execution of various parts of a system. Thus, the precision of the monitored events is equal to the internal clock of the device.

Traces are massive data which can be fed into a machine learning framework. Fortunately, several standard tools and tracing methodologies exist in different environments. Here, in order to analyze the behavior of each process and find out the performance status, each system is equipped with a lightweight tracing tool called the Linux Trace Toolkit Next Generation(LTTng) [110]. It is implemented for high throughput tracing and includes multiple modules for Linux kernel and userspace tracing with minimal cost. Tracing the OS or user applications requires the ability to record thousands of low-level events per second which imposes some overhead to the system that may affect the performance of the target application. Hence, LTTng is a proper tool to be used in our experiment as we would like a tracer to have a low overhead on the monitored systems. Figure 5.3 represents the process of collecting kernel events in a trace file and transferring it into the Trace Analysis module. As illustrated in this figure, the userspace application sends requests to the Linux kernel using system calls which are recorded by LTTng Tracer on .ctf trace files. In the sequel, the trace file is fed into the trace analysis module to create the dataset and perform more investigation.

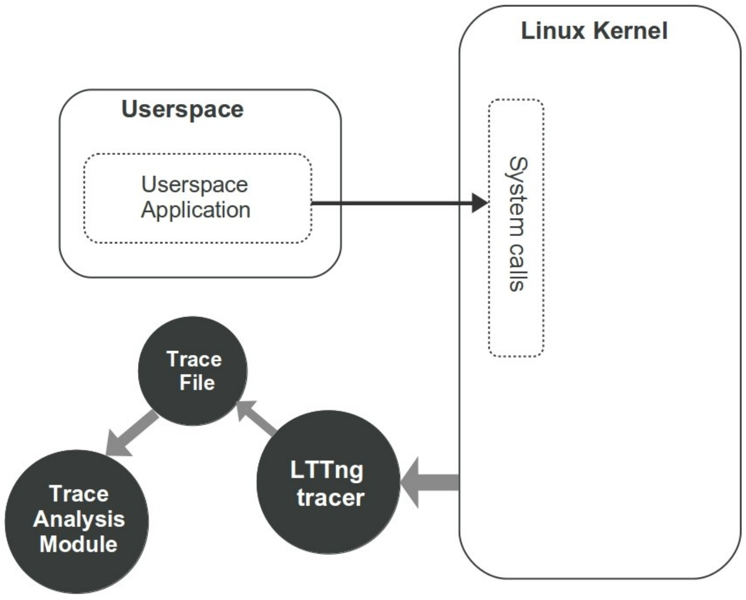


Figure 5.3 Data extraction steps using kernel tracing.

We implemented the trace analyzer module within the Trace Compass open source tool [54], with visualization mechanisms to promote the analysis of the system performance anomalies with different perspectives. Actually, the LTTng tool is applied to collect system calls originated by the monitored processes, and Trace Compass is employed to read the LTTng trace files and to produce a sequence of events with all their associated information (e.g., system call name, timestamp, and duration). Our methodology focuses on system calls, so while the Trace Compass code is reading the trace file, it only collects system calls and skips other events. In the obtained dictionary, each system call entry contains a timestamp, process ID, and some additional run-time information associated with that system call, which is depicted in Figure 5.4.

First, the processes other than the one under study (e.g., MySQL and Chrome) are filtered out considering the process ID field. Then, instead of working with system call names, an index is assigned to each system call. The system calls indices, the corresponding execution times, and other related information are listed for all threads of the target process. Since a single process can produce a huge amount of system calls, considering all system calls at once is not practical in real applications. Therefore, a sliding window is used to continuously extract data from subsequences of system calls. For each subsequence, we define a compact representation that yields two separate feature vectors containing the frequency ($x_{frequency}$) and the duration ($x_{duration}$) of the system calls inside the current sliding window. Thus, our methodology can handle large and varying volumes of data. Since we monitor 318 different system calls of the Linux operating system, each feature vector has 318 dimensions, one per system call type. This feature extraction strategy is shown in Figure 5.4.

The pseudocode for extracting the feature vectors is represented in Algorithm 1. The algorithm receives the windowing size α, the windowing step β, a Trace τ which contains a sequence of events, and the target process m as input. Some factors must be considered when selecting α and β values. Windows must contain sufficient information about the status of the system over a period of time. In one hand, choosing a small amount as the length of the window reduces the useful information volume of the subsequence and increases the number of subsequences. Furthermore, it may even increase sparsity. In the other hand, if a large α value is selected, it is likely that the screened subsequences contain both normal and abnormal events. Moreover, small β values make the subsequences very similar, and larger values also ignore many possible subsequences. There is no need to worry about calculating these values. It can be a manual trial-and-error process to performed at training-validation time. It does not impose much computation cost to the whole framework.

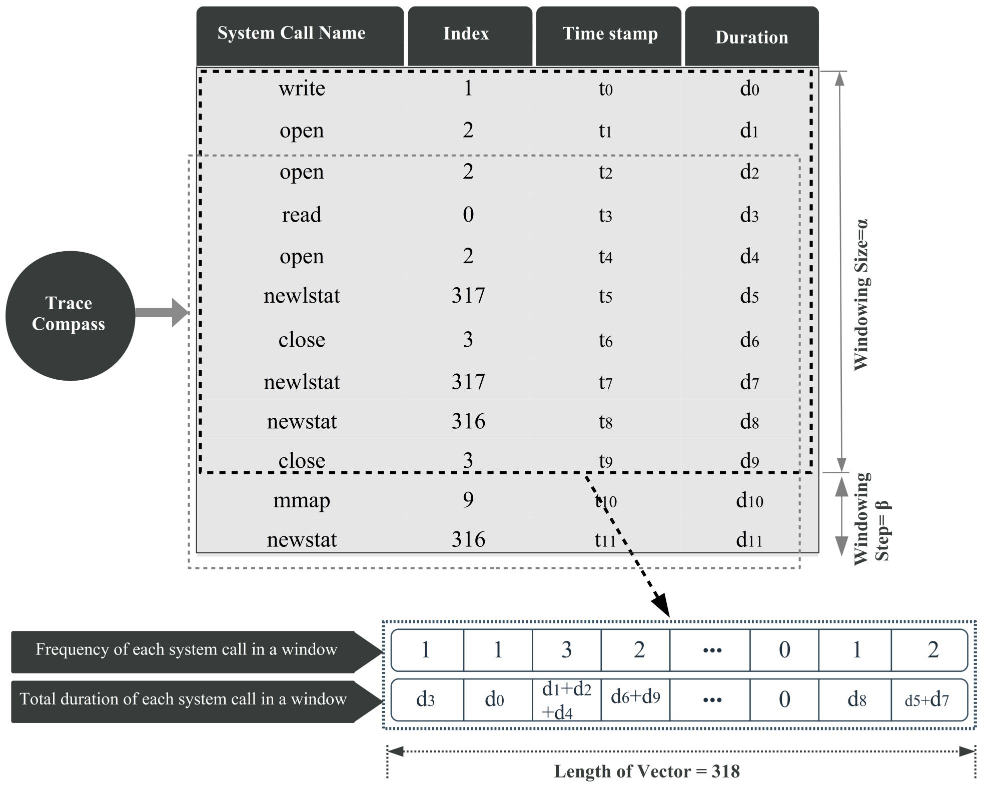


Figure 5.4 Reading trace file and extracting vectors using windowing method.

Algorithm 1 Feature extraction procedure

```
Input: Trace $\tau$, Process $m$, $\alpha$, $\beta$
Output: $D$, $F$
1: $D \leftarrow \emptyset$, $F \leftarrow \emptyset$
2: $SP = \{e \in \tau |\ type(e) = systemcall\ and\ process(e) = m\}$
3: $W \leftarrow$ MakeSubsequences($SP$, $\alpha$, $\beta$)
4: for all $i \in \{1, 2, ..., |W|\}$ do
5:     $FV \leftarrow \sum_{j=1}^{\alpha} R_{i,j}$
6:     $DV \leftarrow \sum_{j=1}^{\alpha} S_{i,j}$
7:     $F \leftarrow F \cup (FV)$
8:     $D \leftarrow D \cup (DV)$
9: end for
```

Algorithm 1 first obtains the set of frequency based feature vectors (F) and the set of duration based feature vectors (D). At the beginning of the algorithm, the system calls belonging to the process m are extracted from the total events in the trace file and a set SP is built (line 2). The function $type(e)$ determines if the event e is a system call or not. Then in line 3 the function MakeSubsequences() obtains all possible subsequences in SP by considering the widowing size α and the windowing step β. For each subsequence W_i two data structures of size ($\alpha \times 318$) are built: R_i and S_i. Let $R_{i,j}$ be a (1×318) one-hot vector which corresponds to the j-th system call done in the i-th subsequence. In this vector, the k-th cell where k=index($w_{i,j}$) is equal to one. The vector FV is calculated by the sum of all the one-hot vectors $R_{i,j}$ for all $j \in \{1, 2, ..., \alpha\}$. In a similar way, $S_{i,j}$ represents a (1×318) one-hot vector which corresponds to the j-th system call done in the i-th subsequence.. In this vector, all the cells have zero value except the k-th cell where k=index($w_{i,j}$). The value of this cell is equal to the duration of that system call. The vector DV is computed by the sum of all the one-hot vectors $S_{i,j}$ for all $j \in \{1, 2, ..., \alpha\}$. Finally, D and F provide a set of duration vectors and a set of frequency vectors, correspondingly (lines 7 and 8).

5.7 Preprocessing of the Extracted Data

Data preprocessing is an essential stage for the success of any machine learning model. In almost all knowledge discovery tasks, the data preprocessing step takes the major part of the overall development effort, even more than the data mining task [111].

5.7.1 Problem of Sparsity

Each subsequence of events present in a window is represented by a frequency (duration) vector with the size of total number of system calls (i.e., 318). Naturally, most of the values in each vector will be zero due to the large number of system calls. Besides, a specific process utilizes special system calls during its execution. In other words, some columns of the data sample will consist of zero values. This characteristic dramatically impacts calculating sample similarities. Moreover, it is hard to understand the relationships between different feature vectors when the training set is not large enough in the presence of sparsity [112]. Thus, in this paper, we reduce the sparsity of the collected data by eliminating all unused features related to system calls that never occur during the execution of the monitored process.

5.7.2 Data Normalization

Data normalization is a fundamental phase of data preprocessing. Data normalization is employed to reduce the dominating effect of some attributes measured in different scales. Here, data standardization is applied on the dataset as a normalization preprocessing step. Let, $\Gamma = \{X_1, X_2, ..., X_n\}$ denote the d-dimensional data set. Thus, Γ is a $n \times d$ matrix:

$$\Gamma = \begin{bmatrix} x_{11} & ... & x_{1d} \\ ... & ... & ... \\ x_{n1} & ... & x_{nd} \end{bmatrix} \tag{5.1}$$

Given a dataset Γ, the Z-score standardization formula is defined as:

$$x_{ij} = Z\left(x_{ij}\right) = \frac{x_{ij} - \mu_j}{\sigma_j}, \tag{5.2}$$

where μ_j and σ_j are, respectively, the samples mean and the standard deviation of the jth attribute. This method rescales the features in a way that they have a standard normal distribution with mean of 0 and standard deviation of 1.

5.8 Performance Anomaly Detection

In this section, first, we assume that enough labelled training samples are available. Thus, we propose a supervised monitoring framework that classifies the system performance into three separated classes: normal, CPU issue, and Memory issue. Although a supervised approach could usually produce acceptable detection results, it requires enough labelled data. Since providing labelled data for the whole data distribution is not always possible, we propose to use an unsupervised approach in Section 5.8.2. The unsupervised approach does not require any labelled data and clusters the input data into separate categories, which could represent different groups of normal, CPU issue, or Memory issue. However, unsupervised approaches usually present worse classification performance than supervised methods in practice given that no priori information is exploited. Therefore, in order to introduce a from of supervision into the unsupervised approach and improve the detection performance, we propose a semi-supervised approach in Section 5.8.3. In this method, we assume that a subset of data is labelled and can be used to guide the feature selection procedure. In this way, the benefits of the supervised and unsupervised learning strategy are combined into a semi-supervised anomaly detection approach.

5.8.1 Supervised Performance Anomaly Detection

Once the system call feature vectors are collected per subsequence, the purpose of the anomaly detection algorithm becomes to train a model with normal and abnormal data from the provided labelled training dataset. Later, the task would be to determine whether a test sample vector belongs to a normal or abnormal behavior. Here, we describe a supervised monitoring framework that classifies the system performance into three separate classes. If a vector has a normal behavior, it will be assigned to the first category. The second class is defined as a CPU issue or, in other words, insufficient CPU allocation problem, which may happen when the system is running a CPU intensive process. Finally, the vectors extracted from a system running a memory-intensive process are assigned to the third category. This class indicates an insufficient memory allocation issue.

Iterative Feature Selection

Feature selection is the process of finding the most discriminative subset of features for performing classification. In the case of supervised learning, this selection is performed based on the available labelled data. A proper feature selection can improve the learning accuracy and reduce the learning time. Here, the Fisher score along with a correlation filtering strategy [18] are applied to determine the best subset of features in the dataset. In this algorithm, a subset of features are found in a way that the distances between samples in different classes become as large as possible, while the distances between data points in the same class stay as small as possible. The Fisher Score FS_j, for $j = 1, \ldots, d$, can be calculated as follows:

$$FS_j = \frac{\sum_{c=1}^{k} n_c \left(\mu_{jc} - \mu_j\right)^2}{\sum_{c=1}^{k} n_c \sigma_{jc}^2}, \tag{5.3}$$

where n_c is the number of samples in class c, for $c = 1, \ldots, k$ (number of classes), and μ_{jc} corresponds to the average value of feature j restricted to the samples in class c. Further, σ_{jc}^2 is the variance of feature j for samples in class c.

The computed Fisher scores of each feature are sorted in non-increasing order and scanned iteratively to select the ℓ features that have low correlation together. A feature is selected to compose the list of ℓ features if its pairwise correlation with one of the features already selected in superior to a given threshold. This procedure continues iteratively until ℓ features are selected. Here, the correlation between two features j_1 and j_2 is computed as follows:

$$Cov(j_1, j_2) = \frac{\sum_i^n (x_{ij_1} - \mu_{j_1})(x_{ij_2} - \mu_{j_2})}{n-1} \tag{5.4}$$

Supervised Multi-Class Anomaly Detection

Once the top-ranked features are selected, we employ a multi-class support vector machine (SVM) [63] classification model. We choose SVM considering its generalization ability and its successful utilization in different pattern recognition applications, such as anomaly detection tasks [113]. SVM finds the hyperplane with the largest margin that classifies the training set samples into two classes. Then the unseen test samples are labelled by checking the sign of the hyperplane's function.

Considering each sample X_i, for $i = 1, \ldots, n$ of the training data and its associated label y_i, SVM finds the optimal hyperplan by solving the following problem:

$$\min_{\omega, d} \frac{1}{2} \omega^T \omega + C \sum_{i=1}^{n} \xi_i \tag{5.5}$$

$$s.t.\ y_i \left(\omega^T \phi\left(X_i\right) + c\right) \geq 1 - \xi_i\ ,\ \xi_i \geq 0\ ,\ i = 1, ..., n \tag{5.6}$$

where ω is d-dimensional vector and ξ_i is a measure of the distance between the misclassified point and the separating hyperplane. The function $\phi\left(x_i\right)$ projects the original data sample x_i into a higher dimensional space and d is the bias. C controls the penalty associated with the training samples that lie on the wrong side of the decision boundary. The radial basis function (RBF) of $\phi(x) = e^{\gamma \|(x - x_i)\|^2}$ is applied to map the data into the non-linear high-dimensional space. The term γ is a parameter that controls the width of the Gaussian kernel. The accuracy of the classification is then dependent on the value of the parameters C and γ.

In this work, we generalize the binary classification model by means of a one-versus-one approach. In this approach, one classifier per pair of classes is built. In our case, it fits three classifiers for three possible pairs of classes: (1) samples with memory issues from the samples with CPU issues, (2) samples with memory issues from the normal samples, (3) samples with CPU issues from the normal samples. The class which received the most votes is selected at prediction time. In the case that two classes have an equal number of votes, it selects the class with the highest aggregate classification confidence by summing over the pairwise classification confidence levels computed by the underlying binary classifiers.

5.8.2 Unsupervised Learning of the Performance Anomalies

Most current anomaly detection systems use labelled training data. As mentioned before, producing this kind of training data is usually expensive. Besides, the definition of normal and anomalous behaviours may change over time. To address these problems, we propose to use an unsupervised system call based anomaly detection scheme. This technique segments unlabelled data vectors into distinct clusters. The proposed unsupervised approach should be able to categorize previously unseen types of anomalies. A wide variety of models for cluster analysis exists; however, the initial choices are usually representative-based algorithms such as K-Means, which directly uses the distances between the data points to cluster a dataset. Another clustering approach based on data density used in this work is DBSCAN which can group clusters of varied complex shapes. In the following, we briefly describe the K-Means and the $DBSCAN$ algorithms.

K-Means Clustering

K-Means is a clustering algorithm that groups samples based on their feature values into k different clusters. Data samples which are assigned to the same cluster are supposed to have similar feature values. In this clustering technique, the sum of the squares of the Euclidean distances of data points to their closest representatives is used as an objective function [68,69]:

$$Dist\left(X_i, X_j\right) = \|X_i - X_j\|_2^2 \tag{5.7}$$

where $X_i = (x_{i1}, ..., x_{id})$ and $X_j = (x_{j1}, ..., x_{jd})$ are two input vectors with d features and $\|\cdot\|_p$ represents the $L_p - norm$. K-Means begins by initializing the k centroids using a straightforward heuristic like random sampling from the dataset and then refines the centroids in the following steps until stability is reached:

- Assign each vector to the closest centroid using the similarity function (Equation 5.7)
- Determine the optimal centroid for each cluster C_j

Dbscan Clustering

The use of K-Means clustering has some limitations. First, it requires the user to set the number of clusters a priori. Second, the presence of outliers has an undeniable impact on K-means. Besides, K-means works better for spherical clusters considering the Euclidean space as the underlying data space. To further reveal this point, consider the clusters represented in Figure 5.5. These plots depict the frequency-based vectors extracted from a chrome process use case along with their real labels. Since the original data has more than 120 attributes, two separate dimensionality reduction approaches were applied to better visualize the data. In Figure 5.5a we present data obtained with the t-distributed Stochastic Neighbor Embedding (t-SNE) [114] while Figure 5.5b presents the data projected in the plane by means of PCA [115]. Both figures reveal that the K-means algorithm is not appropriate to correctly cluster the illustrated dataset. Here there are three clusters of arbitrary shape in the data, and thus density-based algorithms are preferable.

Hence, our proposal uses the DBSCAN algorithm [70], in which the individual data points in dense regions are used as building blocks after grouping them according to their density.

DBSCAN algorithm requires two parameters. The first parameter is ϵ, which defines the neighborhood around a data point. Two points are considered as neighbors if the distance between them is lower or equal to ϵ. If the ϵ value is chosen too small, then a large part of the data will be considered as outliers. On the other hand, if it is chosen very large then the clusters will merge, and the majority of the data points will be in the same cluster. The second parameter is $MinPts$, which indicates the minimum number of neighbors (data points) within ϵ radius. The density of a point is the number of points that lie within a radius ϵ of that point which can be obtained by the following formula :

$$N_\epsilon(X_i) = \{X_j \in Dataset \mid Dist(X_i, X_j) \leq \epsilon\} \tag{5.8}$$

DBSCAN classifies the data points into three categories of core, border, and outliers based on the hyperparameters ϵ and $MinPts$. A point is a core one if it has more than $MinPts$ points within ϵ, and a border point is a point that has fewer than MinPts within ϵ, but it is in the neighborhood of a core point. A point that is not a core point or border point is considered as an outlier. Also, three terms required for understanding the DBSCAN algorithm: (1) point A is "directly density reachable" from point B if A is within distance ϵ from core point B. (2) A point A is "density reachable" from B if there is a set of core points leading from B to A. (3) Two points A and B are "density connected" if there is a core point C, such that both A and B are density reachable from C. A density-based cluster is defined as a group of density connected points. By considering these definitions, DBSCAN algorithm can be described in the following steps:

- For each point x_i, compute the distance between x_i and the other points. Finds all neighbor points within distance ϵ of the starting point x_i. Each point, with a neighbor count greater than or equal to $MinPts$, is marked as core point or visited.
- For each core point, if it is not already assigned to a cluster, create a new cluster. Find all its density connected points recursively and assign them to the same cluster as the core point.
- Iterate through the remaining unvisited points in the dataset.

Those points that do not belong to any cluster are considered as outliers. DBSCAN is able to cluster points into distinct categories without setting the number of clusters.

5.8.3 Semi-Supervised Learning of the Performance Anomalies

Although unsupervised approaches allow one to tackle a massive amount of unlabelled data, they might present worse classification performance than supervised learning methods in practice due to the lack of knowledge about the application itself. In this sense, feature selection can improve the performance of these methods to a great extent. The primary purpose of feature selection is to remove the attributes that do not cluster well which is specially useful for distance-based clustering due to the curse of dimensionality [116]. In unsupervised problems, feature selection is usually more complicated since external validation criteria (such as labels in the underlying data) are not available. Nevertheless, if we have the label of some of the data points, supervised feature extraction methods help discover subsets of features that maximize the underlying clustering tendency. As mentioned before, we benefit from labelled data in this project. Therefore, a variety of supervised criteria can be used, such as the Fisher score. The Fisher score, discussed in Section 5.8.1, measures the ratio of the intercluster variance to the intracluster variance on any attribute. Our proposed semi-supervised learning method selects the most discriminative features from a small set

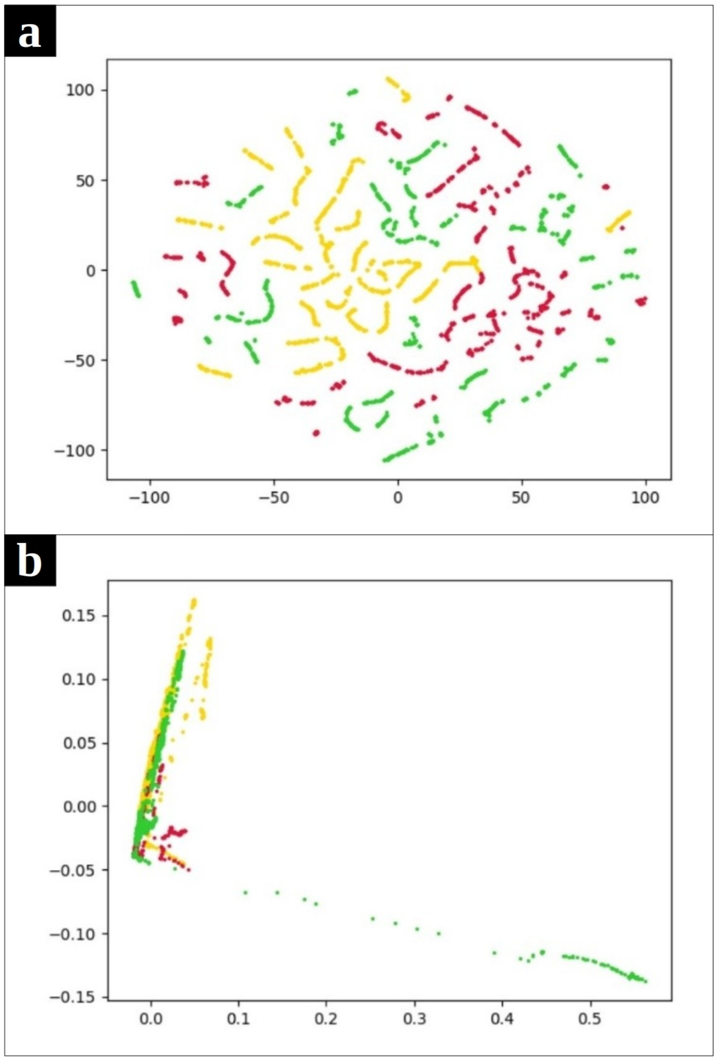


Figure 5.5 Frequency-based samples extracted from Chrome process. Red, yellow and green points refer to normal, CPU problems, and memory problems, respectively. (**a**) uses t-SNE and (**b**) utilizes PCA to map data points onto 2D subspaces.

of labelled data by means of the iterative selection method of Section 5.8.1. In the sequel, the DBSCAN clustering algorithm is applied to group the remaining data into the sought number of classes.

Figure 5.6 summarizes the proposed anomaly detection technique. The kernel tracing data extraction module, which utilized LTTng, Trace Compass, and our windowing method, has the duty of generating vectors. Then in the preprocessing module, some refinements on data are done, and the vectors of more informative features are obtained. Finally, DBSCAN clustering is applied to the obtained dataset.

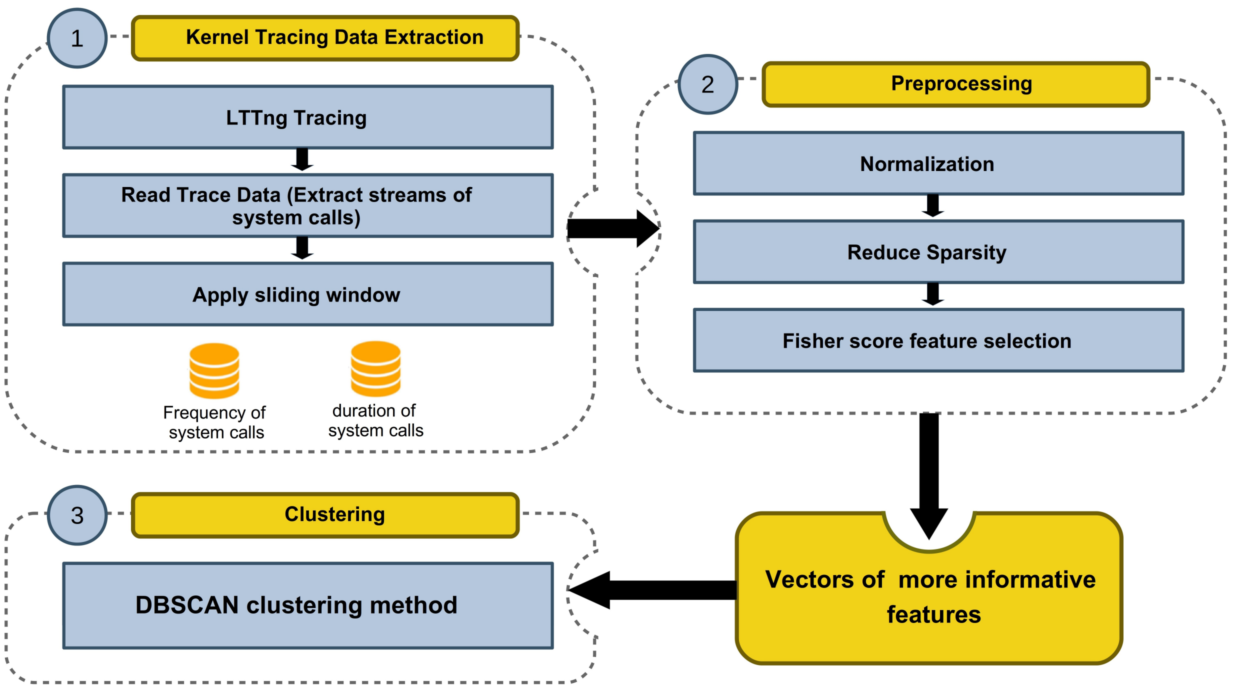

1
Kernel Tracing Data Extraction
LTTng Tracing
Read Trace Data (Extract streams of system calls)
Apply sliding window
Frequency of system calls
duration of system calls
2
Preprocessing
Normalization
Reduce Sparsity
Fisher score feature selection
Vectors of more informative features
3
Clustering
DBSCAN clustering method

Figure 5.6 The architecture of the proposed Semi-supervised framework.

5.9 Evaluation

We evaluated both proposed supervised and semi-supervised anomaly detection approaches on two real system performance anomaly datasets generated based on different faults from Mysql and Chrome applications. Our experimental setup and dataset generation is explained in Section 5.9.1. Then, we analyzed a practical use-case in Section 5.9.2. Finally, the results of the performance anomaly detection approaches are examined in Section 5.9.3.

5.9.1 Setup and Dataset Generation

Our experiments were performed on a group of virtual machines (VMs) allowing us to better manage system resource allocation. The host machine had an Intel Core i7 4 GHz $\times$ 8 CPU and 32 GB of memory. The VMs were equipped with different number of CPU cores and memory allocations depending on the workload simulation, running Linux Kernel version 4.15.0. As the first use case, we used the open-source MySQL synthetic benchmark tool, Sysbench 0.4.12, with OLTP test in complex mode. In order to generate the performance anomaly dataset for MySQL processes, different faults are simulated on the VMs. For example, to create a CPU issue, CPU resources allocated to a VM are limited (e.g., one CPU core, while running eight threads of MySQL). Likewise, a memory issue is created by limiting the amount of memory resources assigned to a VM (e.g., 256 MB memory, while the MySQL table is of size 6 GB). The second use case regards tracing Chrome processes. The $ChromeUnderStress$ 1.0 chrome extension is used to open, close, and refresh many light and heavy pages in Chrome with configurable speed. Faults are simulated by running this Chrome extension on the VMs with different amount of CPU and memory resources. The traces are collected using LTTng 2.10.5. The generated datasets include three classes: normal, CPU issue, and memory issue. Moreover, both MySQL and Chrome datasets are made to contain the same number of samples (i.e., 6000) from each class. We injected faults into the system for each use case using the tools we introduced. However, for other applications injecting faults is possible using two scenarios. The first scenario is injecting faults as intentional software bugs into the code. In this case, we can pause the code for n milliseconds and then continue, calculate π with m bits of precision, or other scenarios. In the second scenario of fault injection, the target is the system in which the code runs using a workload generator tool designed to subject the system to a configurable measure of CPU, memory, I/O, disk, and network stress such as Stress or Stress-ng [117] and PUMBA [118]. Besides, we must keep in mind that whether with a label or without a label, the data collection step is such that all system calls in Linux are considered. We have presented a straightforward method based on kernel tracing using LTTng, which is very light and easy to install in the system to gather all system calls information. The most informative system calls are selected in the next step, the data extraction module. Therefore, the operator does not need to know how useful each system call is, as this will be done automatically later by the framework.

5.9.2 Analysis of Practical Use-Cases

In this experiment, we analyzed the performance vulnerability due to resource Denial-of-Service (DoS) attacks. The goal of DoS attacks is to disrupt fair access to system resources.

We aim to identify a class of DoS attacks in which an application consumes most of the resources so that no other useful work can be done. Thus, it maliciously destroys, for example, the memory-related performance of other applications using shared resources.

In our test scenarios, we investigate the effect of such attack on the performance of Mysql. The machine on which the Mysql is executed is made subject to attacks in few short time intervals. In order to simulate such an attack, the Stress tool has been used to keep the system's resources in an intentionally induced state of overload or deadlock so that the system is unable to perform any other work. In another test, we simulated attacks on compression programs (zip bombs) that can involve highly recursive compressed files for which their decompression result in an uncontrolled consumption of CPU time and file descriptors. Our proposed detection scheme proves to be effective in locating the windows in which the attacks actually take place.

Similar to the data collection phase, the trace file is read by our script in Trace Compass. Then, the whole set of system calls are formatted into windows, which are in turn analyzed by the detection module which highlights the anomalous ones. Our proposed anomaly detection framework represents the output of the detection module in a Trace Compass time chart view. Figures 5.7a,c demonstrate the effectiveness of our proposed method in locating the attacks that have been simulated by Stress and zip bombs. In these time charts, the normal and anomalous windows are illustrated in green and red colors, respectively. The proposed framework helps system experts to focus at just a few small windows instead of the whole trace that may include millions of events. The resulting time charts can be zoomed in and zoomed out in specific areas (Figure 5.7b).

More detailed data can be computed from the trace as the user zooms in using the mouse wheel or right-clicking and dragging in the time scale. The time axis in the time chart is aligned with other views that support automatic time axis alignment. The other capability of our framework is its events editor view (Figure 5.8a), which presents the events in a tabular format. Filtering or searching of events in the table can be done by entering matching conditions in one or multiple columns in the header row. As can be seen in Figure 5.8, in addition to the original events fields, a new field has been added to each event. The field category determines whether the event belonged to a normal or abnormal window. Finally, the statistics view (Figure 5.8b) is provided to display the various event counters. Time synchronization is enabled between the time chart view, events editor, and statistics view. When a filter is applied in the events table, the non-matching ticks are removed from the Time Chart view (and vice versa) [54]. Moreover, the currently selected time range's event type distribution is shown by selecting a time range in the time chart. Figure 5.8b shows the statistics view of the selected anomalous area detected by our tool. The distribution of events in the selected area led us to identify that the attack created by the zip bombs caused the system not to respond to Mysql requests appropriately during this period. The implementation of this visualization module which can be run using the scripting plugin in Trace Compass, is available on Github `https://github.com/kohyar/syscall_anomaly_tracecompass_visualization.git` (accessed on 28 July 2021).

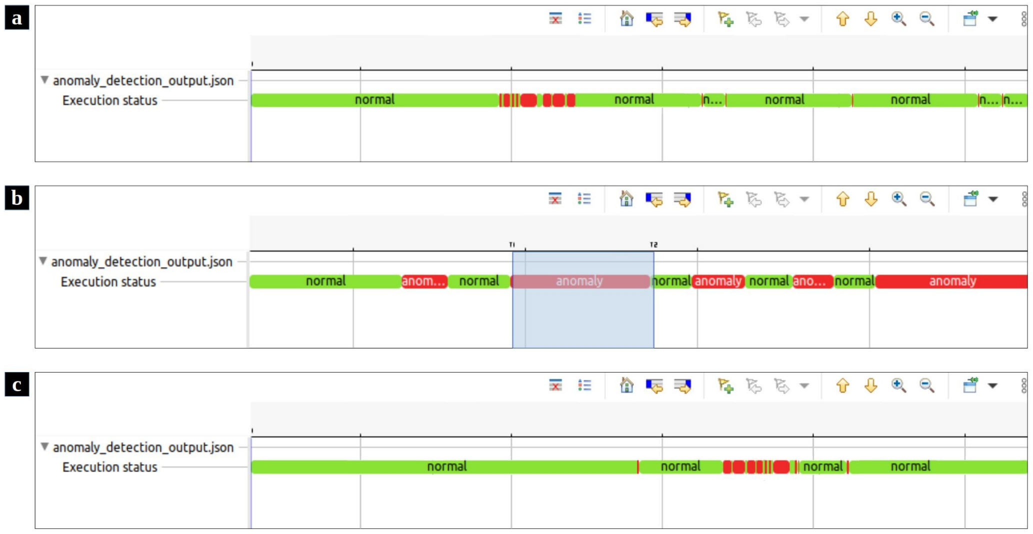
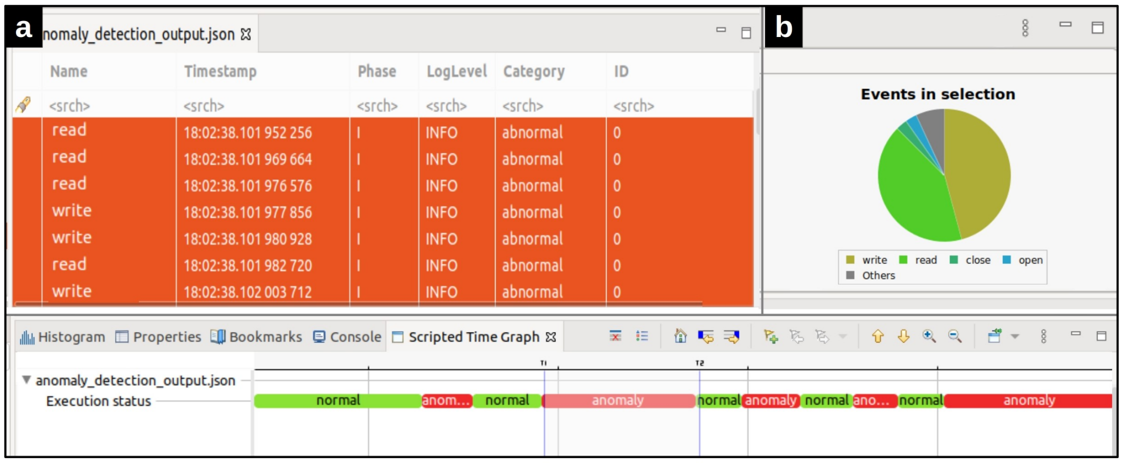


Figure 5.7 The visualized results of the test scenarios in Trace Compass time charts. (**a**) The visualized anomaly detection output where zip bombs simulated DoS attack. (**b**) the time chart provides the ability to zoom in and zoom out a specific area. (**c**) The visualized anomaly detection output where DoS attack was simulated by Stress.

5.9.3 Results

In this section, we evaluate the performance of the proposed anomaly detection approaches with respect to two different extracted feature spaces, one based on the duration and another based on the frequency of system calls. We deploy MySQL and Chrome processes on VMs and extract system calls from tracing the Linux kernel events to construct the feature vectors. In all experiments, the window size is $\alpha = 10^4$ with $\beta = 10^2$ of overlapping. At first, we conduct an experimental study on the supervised method described in Section 5.8.1. Then, the experimental results of the semi-supervised method are reported.



Name	Timestamp	Phase	LogLevel	Category	ID
<srch>	<srch>	<srch>	<srch>	<srch>	<srch>
read	18:02:38.101 952 256	I	INFO	abnormal	0
read	18:02:38.101 969 664	I	INFO	abnormal	0
read	18:02:38.101 976 576	I	INFO	abnormal	0
write	18:02:38.101 977 856	I	INFO	abnormal	0
write	18:02:38.101 980 928	I	INFO	abnormal	0
read	18:02:38.101 982 720	I	INFO	abnormal	0
write	18:02:38.102 003 712	I	INFO	abnormal	0

Figure 5.8 Different features our framework has offered. (**a**) The events editor table for the selected anomalous area, (**b**) The statistics chart for the selected anomalous area.

Experimental Results of the Supervised Method

To tune the hyperparameters of our supervised model, 10-fold cross-validation strategy is used. One fold is used as validation and the union of other folds as training data. This process is repeated ten times for an unbiased evaluation. Fisher scores of the system calls are calculated in each run over the training set. As expected, results show that in both frequency and duration based feature spaces, some system calls have high Fisher scores, and therefore, play a more important role in separating the classes. Figure 5.9 shows the accuracy of the supervised anomaly detection approach during 10-fold cross-validation by varying the number ℓ of selected features. The experiment on MySQL processes reveals that $\ell = 17$ and $\ell = 8$ should be selected for the frequency and duration feature space, respectively. The same experiment on Chrome processes shows that the best number of features is $\ell = 103$ regarding frequency-based features and $\ell = 112$ regarding the duration-based features.

As mentioned before, we employ a multi-class SVM with Radial Basis Function (RBF) kernel on our dataset to classify the input sequences into three classes: normal, CPU issue, and memory issue. The accuracy of the classification method depends on the value of two hyperparameters, C and γ, of the Radial Basis Function kernel SVM. Intuitively, the gamma parameter determines how far the influence of a single training example reaches, with low values meaning 'far' and high values meaning 'close'. The gamma parameters can be seen as the inverse of the radius of influence of samples selected by the model as support vectors [119]. The parameter C is the regularization term, which controls the penalty forced on the margin for the misclassified data points. In order to optimize these hyperparameters, a grid search

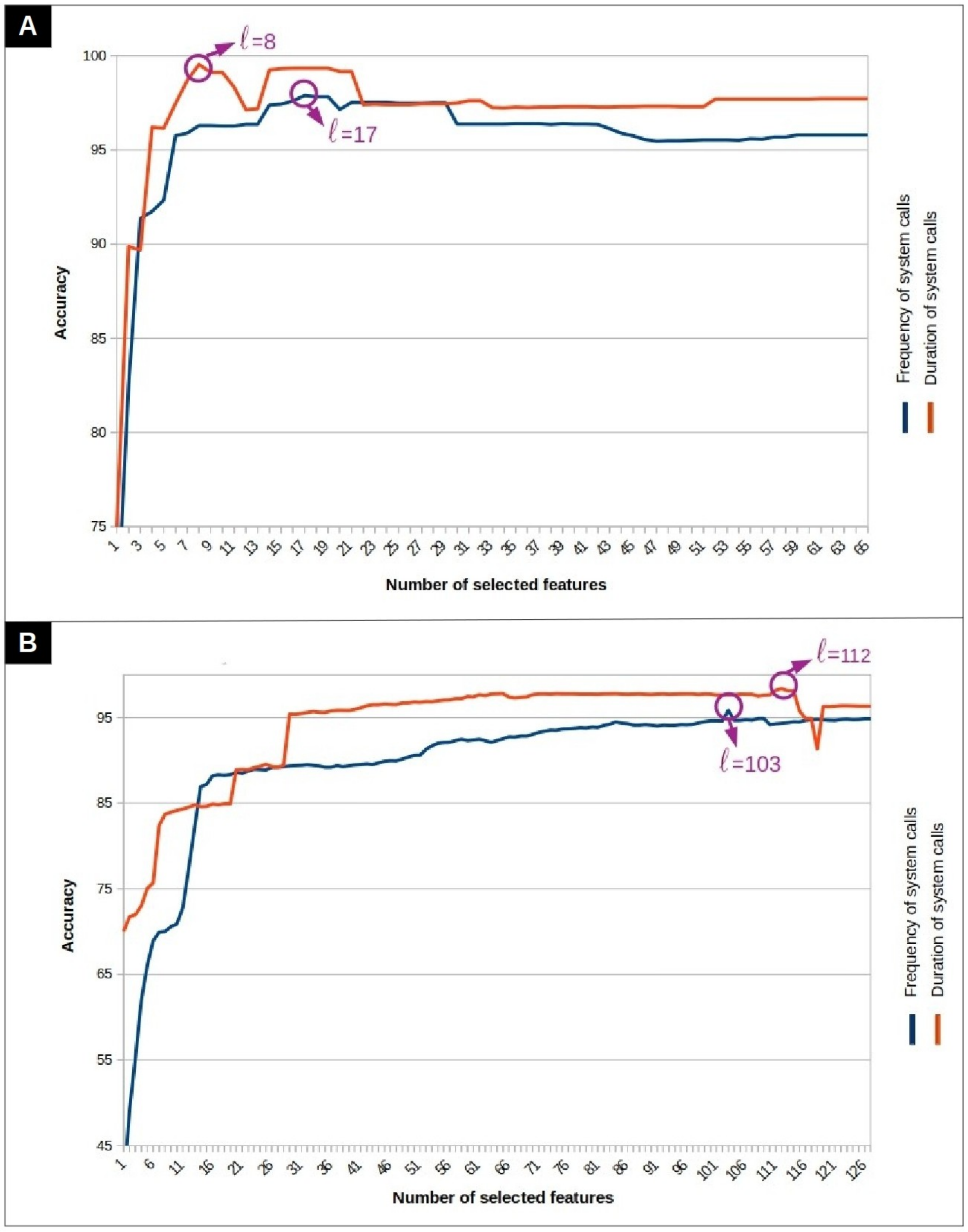


Figure 5.9 SVM-based anomaly detection accuracy versus the different number of top-ranked features. (**a**) Mysql dataset (**b**) Chrome dataset.

algorithm is performed. Figures 5.10 and 5.11 depict the effect of using different combination of parameters on the average accuracy over the validation set. According to Figure 5.10, the pairs ($C = 10^4, \gamma = 1$) and ($C = 10^5, \gamma = 1$) yield the best SVM performance for the MySQL data set in the frequency and duration feature spaces, respectively. Likewise, we observe in Figure 5.11 that the pair of values ($C = 10^3, \gamma = 10$) and ($C = 10^5, \gamma = 10$) are the best for SVM on the Chrome dataset for the frequency and the duration feature spaces, respectively.

After optimizing the SVM hyperparameters, we evaluate our proposed supervised anomaly detection method on unseen test data. The accuracy, precision, and recall of the proposed RBF-SVM anomaly detection framework is reported in Table 5.1. These results show that both frequency and duration of system calls are useful features to perform multi-class anomaly detection, being SVM able to obtain good classification metrics by using either of them.

Experimental Results of the Semi-Supervised Method

Following the experiment setting mentioned before, we conduct clustering experiments using K-Means algorithm and DBSCAN to evaluate the performance of unsupervised and semi-supervised performance anomaly detection. For the case of clustering, ARI (Adjusted Rand Score) is used to measure the performance [120]. ARI computes a similarity measure between two clustering solutions by considering all pairs of samples and counting pairs that are assigned in the same or different clusters in the predicted and true clusterings.

We study in Table 5.2 how the iterative feature selection method of Section 5.8.3 impacts the performance of K-Means. This table shows that the ARI of the K-Means clustering method for the frequency-based dataset by selecting the 17 and 103 features with the highest Fisher scores (i.e., $\ell = 17$ and $\ell = 103$) is 0.003 and 0.128 on MySQL and Chrome, respectively. On the other hand, for the duration-based dataset, using $\ell = 8$ leads to the ARI of 0.038 for MySQL samples, and the ARI of 0.018 is obtained for the Chrome samples by selecting $\ell = 112$. The values of ℓ used are the same of the previous section obtained with the supervised model.

To better explain the output of K-Means clustering, Figure 5.12 presents the result of this

Table 5.1 The performance of the proposed supervised anomaly detection approach.

	Number of features	**Accuracy**	**Precision**	**Recall**
MySQL Process	Frequency (ℓ=17)	0.928	0.989	0.968
	Duration (ℓ=8)	0.937	0.988	0.978
Chrome Process	Frequency (ℓ=103)	0.951	0.990	0.994
	Duration (ℓ=112)	0.959	0.991	0.985

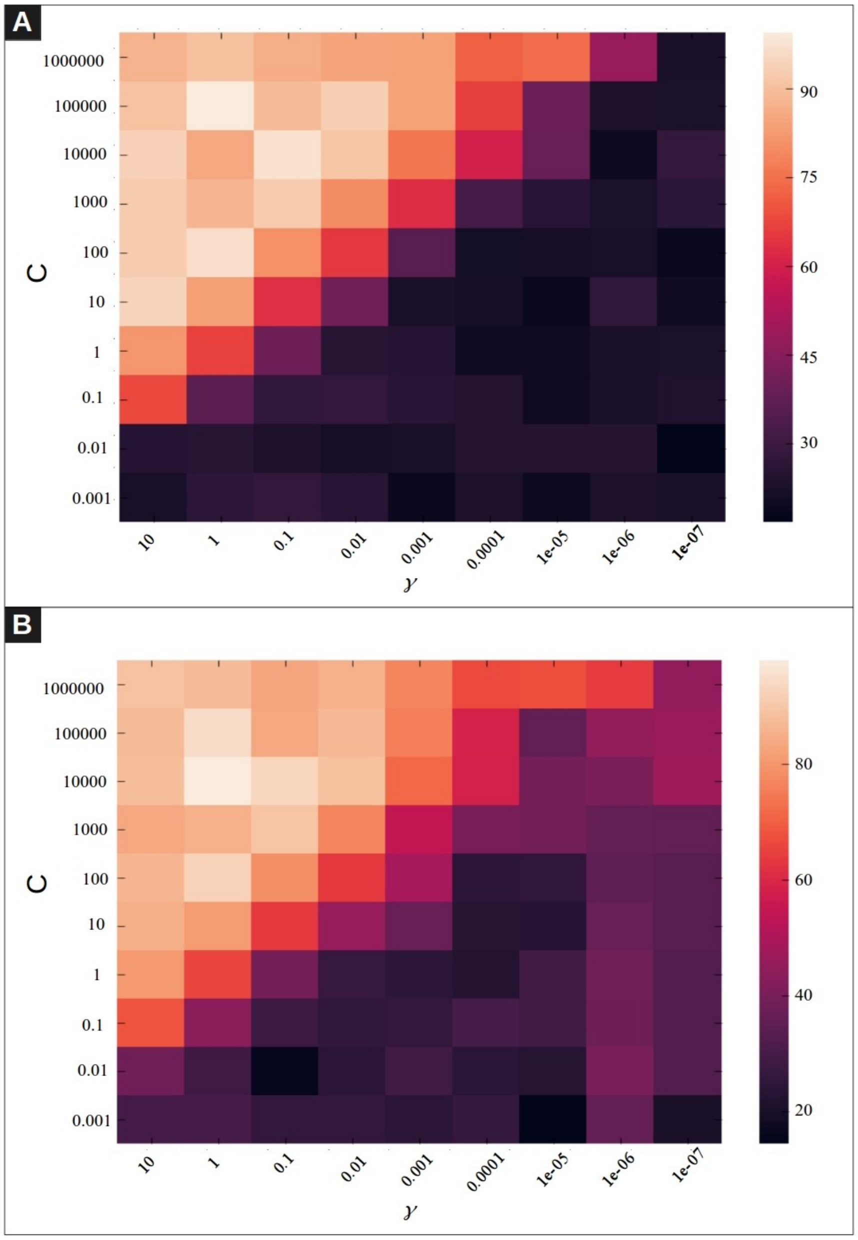
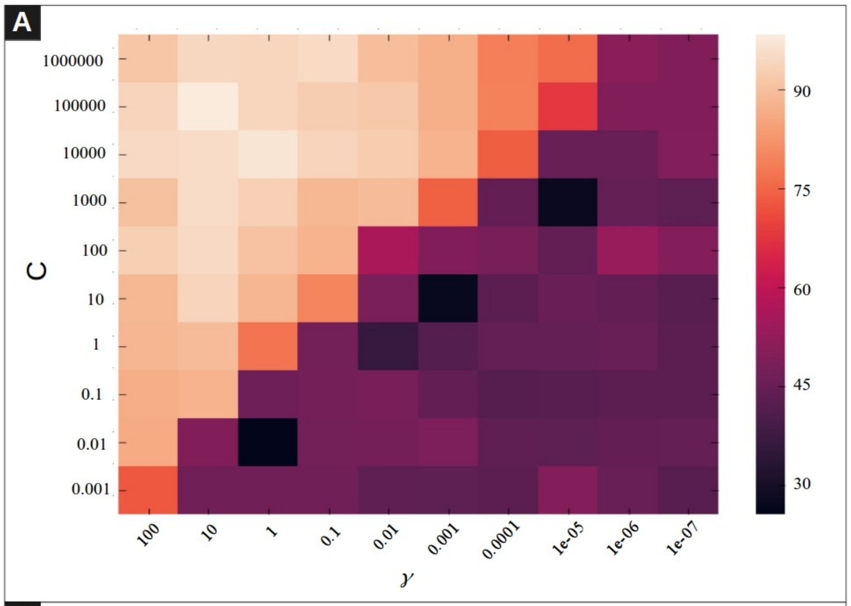


Figure 5.10 Heat map of the frequency-based and duration-based supervised anomaly detection accuracy using different parameters γ and C for Mysql dataset. (**a**) The heat map for frequency feature space, (**b**) The heat map for duration feature space.



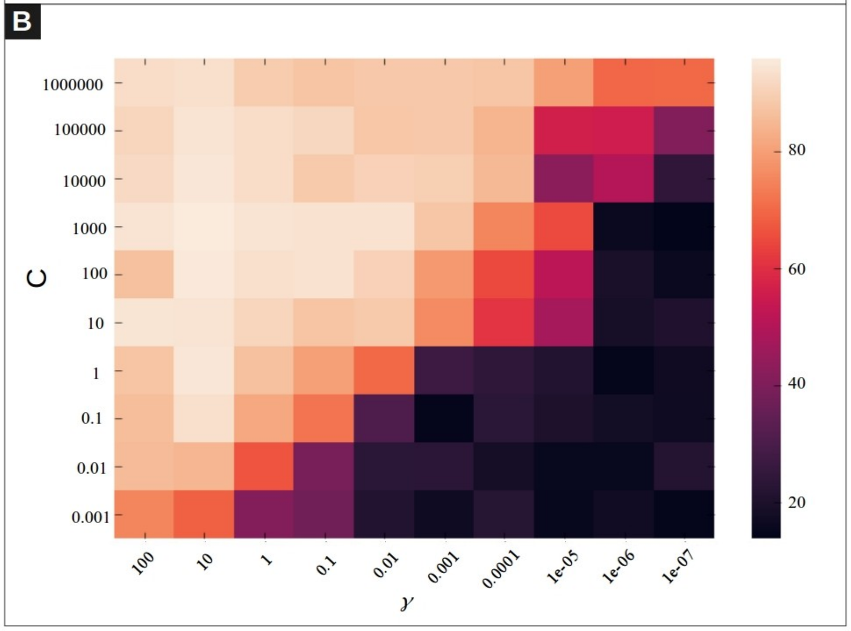


Figure 5.11 Heat map of the frequency-based and duration-based supervised anomaly detection accuracy using different parameters γ and C for Chrome dataset. (**a**) The heat map for frequency feature space, (**b**) The heat map for duration feature space.

Table 5.2 Validation of K-Means based semi-supervised technique on original features versus where the Fisher score feature selection method is applied.

	Frequency-Based Data set		**Duration-Based Data set**	
MySQL Process	Original Features	0.000	Original Features	0.000
	Fisher Score (ℓ=17)	0.003	Fisher Score (ℓ=8)	0.038
Chrome Process	Original Features	0.084	Original Features	0.001
	Fisher Score (ℓ=103)	0.128	Fisher Score (ℓ=112)	0.018

method visually. In general, these results reveal that the K-Means framework does not perform well in both duration-based and frequency-based feature spaces. This comes from the fact that the distributions of data samples in the different clusters do not have a spherical shape. In the next experiment, we analyze the performance of the DBSCAN algorithm.

The clustering results using the DBSCAN algorithm on the original feature space are shown in Table 5.3 for both MySQL and Chrome datasets. The parameter ϵ determines the maximum distance between two samples for one to be considered as in the neighborhood of the other. The performance of the DBSCAN method is evaluated by varying ϵ, thus obtaining different number of clusters.

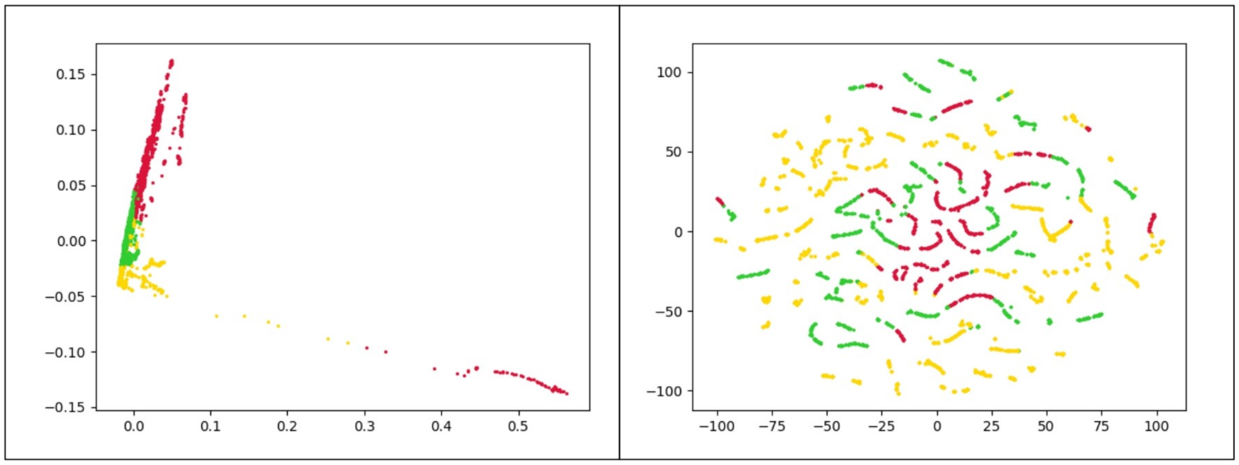


Figure 5.12 The visual result of K-Means clustering after choosing $\ell = 103$ features with the highest fisher score on frequency-based data set for the Chrome process; each color refers to a cluster. The **left plot** uses PCA, and the **right plot** utilizes t-SNE to map data points onto 2D subspaces.

The comparison results of DBSCAN are shown in Table 5.3. By examining the ARI using values of ℓ obtained in the supervised model, the DBSCAN yields an ARI of 0.874 by selecting

$\ell = 17$ on frequency-based data set for MySQL process for which three large and five small clusters are detected. Similarly, on frequency-based data set for the Chrome process, the DBSCAN leads to an ARI of 0.823 when 103 features are selected based on the highest Fisher scores. Three large and six small clusters are obtained in this experiment. The results show that three much larger clusters are obtained in both use cases, and each of these clusters ideally contains one type of data we introduced before (normal data, memory problems, and CPU problem). The comparison of Figure 5.5, which shows the data points with the actual labels, and Figure 5.13, which illustrates the data points with labels obtained from the semi-supervised method, confirm this claim. From this table, it is clear that the performance of DBSCAN is superior to that of K-means. Moreover, the classification performance of DBSCAN clustering largely benefits from the supervised feature selection procedure. Finally, Table 5.3 displays that the proposed semi-supervised anomaly detection on the frequency-based feature space shows better ARI than the duration-based space for the mentioned processes. Interestingly, the evidence from this study intimates that by selecting the most discriminative features, the number of identified clusters by DBSCAN is decreased. This finding highlights the role of the mentioned feature selection method for mitigating the effects of the curse of dimensionality and overfitting.

To better understand the output of DBSCAN clustering model, Figure 5.13 displays the result of this model visually on frequency-based data set for the Chrome process. In the first plot, two principal components of PCA are used, and similarly, the second plot utilizes t-SNE [114] to map data points onto 2D subspaces.

5.10 Conclusions

In this paper, a framework for monitoring of processes and detecting performance anomalies was proposed. The framework is able to distinguish normal behavior, CPU shortage, and memory shortage in monitored traced systems. The proposed methodology works based on recording the stream of system calls using the Linux kernel tracing. From that, short sequences of system calls are extracted, and two feature vectors of duration and frequency

Table 5.3 Validation of DBSCAN based semi-supervised technique on original features versus where the Fisher score feature selection method is applied.

	Frequency-Based Data set	**ARI**	**Number of Clusters**	**Duration-Based Data set**	**ARI**	**Number of Clusters**
MySQL Process	Original Features($\epsilon = 10^{-3}$)	0.281	17	Original Features($\epsilon = 10^{-3}$)	0.278	18
	Fisher Score ($\ell = 17$ and $\epsilon = 10^{-3}$)	0.874	8	Fisher Score ($\ell = 8$ and $\epsilon = 10^{-3}$)	0.855	8
Chrome Process	Original Features($\epsilon = 5 \times 10^{-4}$)	0.254	21	Original Features($\epsilon = 10^{-3}$)	0.127	27
	Fisher Score ($\ell = 103$ and $\epsilon = 5 \times 10^{-4}$)	0.823	9	Fisher Score ($\ell = 112$ and $\epsilon = 10^{-3}$)	0.701	11

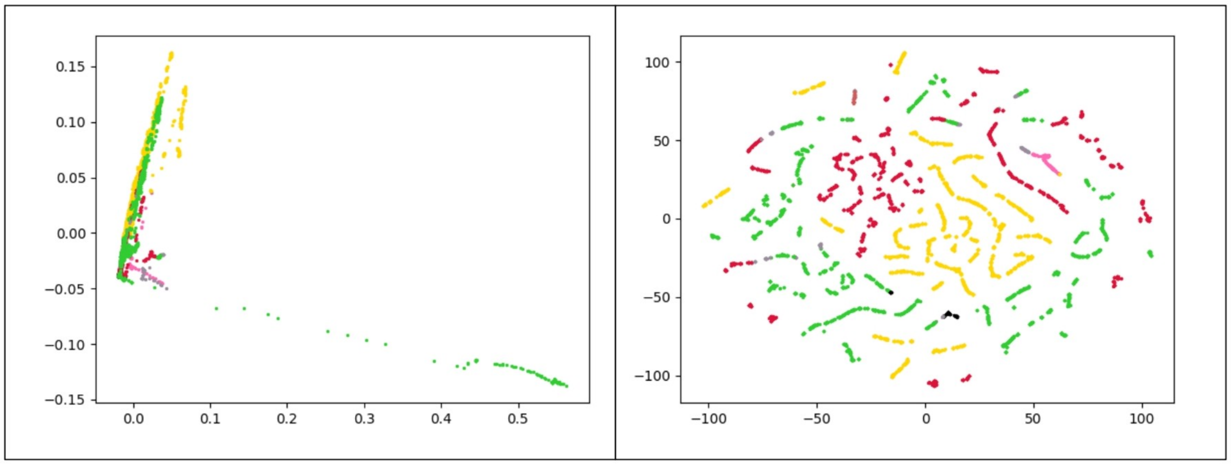


Figure 5.13 The visual result of DBSCAN clustering with $\epsilon = 5 \times 10^{-4}$ after choosing $\ell = 103$ features with the highest fisher score on frequency-based data set for the Chrome process; each color refers to a cluster. The **left plot** uses PCA, and the **right plot** utilizes t-SNE to map data points onto 2D subspaces.

are created to be exploited by machine learning techniques. The way we defined the data collection module makes this framework general enough to work with any specific application. Collecting system calls can be simply done on any system. Also, no special settings are used in the data collection module. Then, the extracted feature vectors are exploited by supervised, unsupervised, and semi-supervised techniques depending on the volume of available labelled data. In the supervised case, Fisher Score was applied to select the most discriminative features, and a three-class SVM algorithm was employed to detect classes. The classification performance of the method is very good, with accuracy never below 0.92. The performance of unsupervised clustering methods (i.e., K-means and DBSCAN) was also evaluated for the case when no prior knowledge is used. Our experiments revealed that the performance of $DBSCAN$ is superior to that of K-means but not as good as that of the proposed supervised approach. Our research underlined the importance of supervised feature selection procedure (Fisher score feature selection), which is used in the proposed semi-supervised approach. Our experiments revealed that the supervised selection of features is able to boost considerably the performance of unsupervised clustering algorithms, with ARI measures as good as 0.874 regarding partition agreement. Taken together, these findings suggest that our framework is an effective tool for automated anomaly detection from traced system calls. The proposed framework along with other works done by our team will be integrated as an open-source Trace Compass extension. In the future, we will explore the performance anomalies in microservice systems using tracing data and Machine Learning. Furthermore, it would be interesting to investigate other learning models for detection of anomalies to achieve better detection performance.

CHAPTER 6 ARTICLE 3: ANOMALY DETECTION IN MICROSERVICE ENVIRONMENTS USING DISTRIBUTED TRACING DATA ANALYSIS AND NLP

Authors: Iman Kohyarnejadfard, Daniel Aloise, Seyed Vahid Azhari, Michel Dagenais
Department of Computer and Software Engineering, Ecole Polytechnique de Montreal, Montreal, H3T 1J4, Canada

Submitted in: Journal of Cloud Computing

6.1 Abstract

In recent years DevOps and agile approaches like Continuous Integration and microservice architectures have become extremely popular given the increasing need for flexible and scalable solutions. However, several factors such as their distribution in the network, the use of different technologies, their short life, etc. make microservices prone to the occurrence of anomalous system behaviours. In addition, due to the high degree of complexity of small services, it is difficult to adequately monitor the security and behavior of microservice environments. In this work, we propose a natural language processing (NLP)-based approach to detect performance anomalies in spans during a given trace, besides locating release-over-release regressions. Notably, the whole system needs no prior knowledge, which facilitates the collection of training data. Our proposed approach benefits from distributed tracing data to collect sequences of events that happened during spans. Extensive experiments on real datasets demonstrate that the proposed method achieved 0.9759 F_score. The results also reveal that in addition to the ability to detect anomalies and release-over-release regressions, our proposed approach speeds up root cause analysis by means of implemented visualization tools in Trace Compass.

Keywords: Performance monitoring, Anomaly detection, Tracing, Microservices, Machine learning, NLP, LSTM

6.2 Introduction

Nowadays, computing infrastructure has significantly evolved with complex systems facilitating many complicated and large-scale tasks in distributed environments and cloud infrastructures.The *microservice* architecture has emerged as a result of this development. Microservices are small services that are interconnected among many others to present a complex service such as a web application [121]. They provide greater scalability, making possible the distribution of an application over multiple physical or virtual systems. In addition, the microservice architecture improves productivity by decomposing applications into smaller services that are easier to manage and faster to develop. Unlike the monolithic architecture, if one microservice fails the others continue to work.

These improvements have increased user expectations in a way that any performance anomaly may lead to user dissatisfaction and revenue losses. Even when several services are brought down for maintenance, the users usually notice it. Although significant efforts have been made to ensure the quality of microservices, the complexity and large scale of these systems make them fragile and prone to performance anomalies and failures [122]. Besides, performance monitoring and tracing of these applications become more challenging by increasing the degree of automation and distribution. For example, each service can be developed using its own language or technology while still communicating with other services. Moreover, unlike monolithic applications in which dedicated teams work on discrete functions such as UI or database, microservices employ cross-functional teams to handle an application's entire life cycle using a continuous delivery model [121, 123]. Nonetheless, dynamic services makes monitoring more difficult. Even if a tracer can record all the execution details, it is still hard to detect the source of the problem inside the trace files.

Different reasons may cause performance anomalies in microservice environments [124]. Any problem in a service, such as a network disconnection or hard disk failure, may cause the microservice system to crash. Misconfigurations or extreme load by a service can also affect the whole system. Changes in one service may influence other dependent services' workload and may result in response time degradation. Moreover, the agile nature of microservice environments yields multiple services updates per day, and several versions of the application may be deployed in a short amount of time. As such, several methods may change in a new update, which affect the response time behavior of services and may lead to many false alarms from monitoring tools [124, 125].

The way we trace such environments and collect data is of particular importance. A microservice-based application consists of tens, hundreds, or thousands of services running across many hosts. Consequently, it is not possible to rely on an individual trace. Distributed tracing provides a view of a request's life as it travels across multiple nodes and services communicating over various protocols [126]. It enables to follow the spans and events that occur in different nodes. A *span* is the primary building block in distributed tracing and represents an individual unit of work done in a distributed system. Besides, many sub-spans may be generated during the spans lifetime, in which tens of userspace and kernel events occur in a particular order. The proposed diagnostic approach works based on collecting sequences of events during spans using the Linux Trace Toolkit Next Generation (LTTng) [110], sending them to the detection module, and eventually analyzing the outputs of the model in Trace Compass. LTTng provides a system software package for correlated tracing of the Linux kernel, applications, and libraries [110].

In this paper, we propose a general framework to find anomalies as well as release-over-release regressions in microservice environments by taking advantage of NLP and open-source tools (i.e., LTTng and Trace Compass). In general, anomaly detection and localization is the process of finding patterns in data that deviate from normal behavior [19] and literally, it is different from noise detection and noise elimination, which refer to unwanted noise in the data. Anomalies in data may happen in various forms, such as point anomalies and collective anomalies, two principal forms of anomalies. Methods that work based on detecting point anomalies and also metric-based algorithms cannot always identify the root cause of anomalies. A single data point (event or metric), regardless of the data points that occurred around it, does not include enough information to determine whether an anomaly happened in complex systems such as microservices. We usually state that an anomaly has occurred when the program's execution has not been normal during a time interval that includes many events. We look for abnormal event patterns or collective anomalies in these intervals. Only a limited number of events can be the result of an action. Therefore, just a few of the possible events can appear as the next event in the sequence of observed events [101]. Similar to words in natural language processing, events as elements of a sequence follow specific patterns and grammar rules. We used this idea in our anomaly detection framework and applied a general NLP-based strategy to distinguish normal and abnormal patterns in the sequence of events. In this way, we avoid creating labelled datasets for supervised learning. Finally, besides locating anomalies, our proposed framework also allows analysts to zoom in the detected anomalous part of the trace to discover the root cause of the problem.

The main contributions of our work can be summarized as follows:

- Unlike many other methods that use OpenTracing, our anomaly detection framework employs LTTng to perform distributed tracing. OpenTracing is a vendor-agnostic API to help developers easily instrument tracing into their code [42]. A trace in OpenTracing is a directed acyclic graph of spans, and it provides only relationships across microservices. In contrast, LTTng provides details of the program's execution with higher resolution by presenting kernel and userspace events.

- We developed a handcrafted data extraction module in Trace Compass to construct the spans using the request/response events tag. Moreover, the hierarchical structure of these tags helps us to extract subspans. This module is also responsible for converting each span into a sequence of events.

- Our LSTM-based model, designed for post-analysis of traces, learns the normal patterns of events along with their arguments (e.g., event type, tag, and process name). Further, this model is trained to predict the next event's arguments in addition to the event's name. Learning and predicting at this level of detail sets our model apart from the others found in the literature.

- Our framework makes it possible to examine the system behavior from both the system and service perspectives, which gives the troubleshooter a deep understanding of what happens at the time of an anomaly. Our framework's visualizations considerably reduce troubleshooting time by highlighting the anomalous parts of the trace and directing the debugger to the most relevant problem sites of interest. Without such visualizations, manually tracking the performance of systems within low-level tracing data, possibly including thousands of events from different spans, is indeed a very exhausting task.

- In addition to anomaly detection, our framework can be applied to identify release-over-release regressions. Finding potential regressions from one release to another is extremely valuable, and conventional performance tests cannot reveal sufficient regressions. Many subtle changes in spans or sequence of events signify a regression that can be captured using our framework.

The rest of the paper is organized as follows. In Section 6.3, related studies are presented. In Section 6.4, we introduce our automatic integrated anomaly detection framework for microservice environments. Section 6.5 provides the experimental results followed by the conclusions in Section 6.6.

6.3 Previous Work

In traditional approaches, application performance management (APM) tools that support various measures are utilized to perform resource behavior analysis on microservices [79]. Tracing is another robust and efficient approach for reverse engineering and debugging of complex systems [73]. Many tracers across all software stack layers, and even at the hardware level, have emerged in the last years. Distributed tracing, unlike the most traditional methods that only monitor individual components of the architecture, is applied to complex distributed systems at the workflow level [82]. Tools like OpenCensus and OpenTracing [42] help to record the execution path of each microservice request. Jaeger [46], a popular tool that supports

OpenTracing and developed by Uber, has been widely used to automatically collect and store the service call data [47, 48]. Its counterpart Zipkin [43] aids in gathering timing data needed to troubleshoot latency problems in microservice architectures [44, 45]. However, the high-level information that these tools provide is not always sufficient to characterize the execution status of the system since they do not offer kernel events. Thus, tracing with LTTng is a fundamental part of our anomaly detection framework. This open-source tool is implemented for achieving high throughput and includes multiple modules for Linux kernel and userspace tracing, thereby imposing low overhead to the operating system. Besides, this tool can work with a variety of environments, such as monolithic applications, microservices, and IoT devices [127].

The earliest efforts for anomaly detection had used statistical methods [87] where an anomaly score was calculated using a function of abnormality to show the behavior of the application. In [88], CPU performance and network performance metrics in master-slave and nested-container models are compared to provide a benchmark analysis guidance for system designers. However, a live threshold is required given the system's current state to determine whether the program behavior is normal or abnormal, which is practically impossible to set in real-time. Furthermore, these tools do not provide any details about the application's execution flow. Several machine learning-based schemes have also been applied to detect anomalies in microservice systems in addition to statistical and metric-based methods. Hierarchical Hidden Markov Models (HHMM) are adopted in [128] to learn a model based on different monitored metrics such as CPU, Memory, and Network to locate anomalous behaviors. Besides, many clustering algorithms, such as k-means, k-medoids, EM clustering, and outlier detection algorithms, have been employed for anomaly detection in microservice environments [92, 99, 100]. The main problem with such methods is that they are usually difficult to interpret. Supervised methods such as SVM, Fuzzy Logic, and Neural Networks, which use labelled data, were proposed in [93, 129, 130]. Adel Abusitta used SVM to detect DoS attacks in virtualized clouds under a changing environment [129]. In [3], a fuzzy technique was proposed to extract abnormal patterns based on various statistical metrics in which fuzzy logic rules are applied to classify data. However, in practice, the labeling process is highly complicated, and even impossible sometimes. Recently, deep learning techniques which do not need labelled data have yielded promising results. Nedelkoski et al. [95] and [96] propose anomaly detection methods for large cloud infrastructures using long short-term memory (LSTM) neural networks [97] with data from distributed tracing technologies. In [97], a stacked LSTM network model was presented for anomaly detection in time series where the network was trained on non-anomalous data. The drawback of these methods is that many details, including events arguments such as event type, tag, process name, and return value are ignored.

Furthermore, the researchers have made much effort to improve anomaly detection by using different data representations and information resources. Tracing data or log, as the most popular information resource, can be represented in the form of an enumerated collection of events sorted by their timestamps [83]. Different works make different uses of this structure. In DeepLog [49], a deep neural network model is proposed to model an unstructured system log as a natural language sequence. In [84], by performing time-series-based forecasting, anomalies on cyclic resource usage patterns are detected. In the sequel, graph representations of the events are obtained from this data and employed to detect critical nodes and design anti-patterns proactively. The authors of [85] designed and developed a simplified MSA application and applied different graph algorithms, and then assessed their benefits in MSA analysis. In another article, Tao Wang et al. [86] organized the trace information collected by the OpenTracing tool to characterize processing requests workflow across multiple microservice instances as a calling tree. The proposed approach converts the given trace into the spans and detects performance anomalies using the model of normal key patterns.

Some points distinguish our work from previous related literature. Fistly, unlike traditional approaches where application performance management tools that support various metrics (e.g., CPU and memory utilization) are utilized to perform resource behavior analysis on microservices, our work's main source of information is tracing data. Compared to these approaches, our proposed framework is not dependent on the existence of any threshold. Moreover, the metrics used by these approaches do not help to find the cause of the anomaly after detecting it. Tracing provides considerable details about the application's execution flow and about what exactly happened at the time an anomaly occurred. Secondly, most previous works that make use of tracing data employ OpenTracing-based tools such as Jaeger or Zipkin to perform distributed tracing. Nevertheless, the high-level information that these tools provide about microservices interaction is not always sufficient to characterize the execution status of the system. Our proposed framework employs the LTTng open-source tool, which imposes low overhead on the operating system and presents low-level kernel and userspace tracing. Thirdly, while clustering approaches are difficult to interpret, the main drawback of supervised methods is that they require labelled data. The process of labeling data points in terms of performance status is highly complicated and sometimes even impossible. In addition, to collect labelled data related to an application, an very specialized professional is needed. We propose in this work an unsupervised method to learn normal execution patterns. Collecting normal data is pretty easy and can be done automatically without any supervision. Fourthly, Deep learning-based and NLP-based approaches ignore events arguments in their modeling. Event arguments such as process name, message, and event type contain beneficial details that increase detection quality [131]. We use these arguments in the training of our model. Then, in the prediction phase, our model predicts the name of the next event as well as its arguments. Finally, previous works from the literature, such as DeepLog, have not presented any solution to analyze the model's output. However, using Trace Compass in our approach enables us to develop analysis scripts and use many preexisting scripts and visualizations to examine the model's output more deeply.

6.4 ANOMALY DETECTION FRAMEWORK

In this section, we introduce an NLP-based anomaly detection framework for post-analysis of LTTng traces. It is designed to help developers to efficiently find the root causes of abnormal behaviors in microservice environments. We aim to provide a general framework applicable to microservice-based applications with different settings.

Figure 6.1 presents the architecture of our approach along with its three main modules, i.e., the *tracing module*, the *data extraction module*, and the *analysis module*. We discuss this architecture in detail in the next subsections.

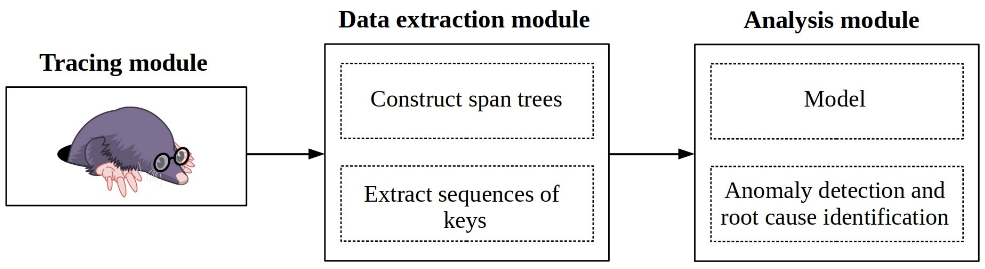


Figure 6.1 The architecture of our proposed anomaly detection method for microservice environments.

6.4.1 Tracing module

Tracing is an efficient way of gaining information on a system for further analysis and debugging, thus minimizing the monitoring influence. Distributed tracing is derived from traditional tracing so as to be employed within distributed systems. Distributed tracing technologies provide a view of the communication among microservices [126] Microservices mostly use Representational State Transfer (REST) as a usual way to communicate with other microservices.

We aim to provide a general anomaly detection framework that can be easily applicable for any microservice-based application in practice and subsequently lead to the discovery of the cause of the identified anomalies. To evaluate our framework, we describe how to analyze an application and prepare its associated dataset, instead of using pre-existing available datasets which do not inherently contain information needed to extract spans and their associated sequence of events.

We created our dataset by tracing a distributed software available in Ciena Corporation. Many new releases of this software are provided by the developers of this company every day, so that traces are collected from different releases to compose the dataset. We denote the set of all traces collected from different releases as $\Gamma = \left\{T_1, T_2, ..., T_n\right\}$, where n indicates the number of collected traces.

Figure 6.2 illustrates the structure of our tracing modules that make use of the LTTng open-source tool. As presented in this figure, LTTng is deployed on each node to send the tracing data to the manager. The running LTTng-relayd daemon on the manager collects the tracing data received from the nodes. Later, Trace Compass integrates the traces obtained from different nodes to form a Trace $T_i = \left\{e_1, e_2, ..., e_{g(T_i)}\right\}$, where $g(T_i)$ is the number of events associated to T_i. Actually, T_i is represented as an enumerated collection of events sorted by their timestamps.

During the execution of a microservice application, many tasks or spans, such as opening a web page, are performed. In fact, a trace can be divided into a set of spans, where each span consists of a sequence of events that are invoked in a specific order to perform the desired task. It should be noted that spans can not be directly retrieved using LTTng. In the sequel, we will discuss in detail how to extract spans from tracing data.

6.4.2 Data extraction module

We implemented the data extraction module within the Trace Compass open-source tool, which offers scripting capability [55] and visualization mechanisms to promote our analysis. LTTng generates a CTF (Common Trace Format) file for every node in the microservice environment. The CTF format is a file format optimized for the production and analyses of big tracing data [110]. After generating the CTF files, Trace Compass is used to read these files and integrates them into trace T_i, where i indicates the index of this trace in Γ. The result of this process is an enumerated collection of events sorted by their timestamps.

An event is composed of well-defined fields that are common to all events, such as name, timestamp, and process ID. However, the delivered sequence of time-ordered events does not provide the spans that reflect separate tasks. In order to extract spans and their subspans, Γ is scanned with respect to the tag of request/response events. Other events are then

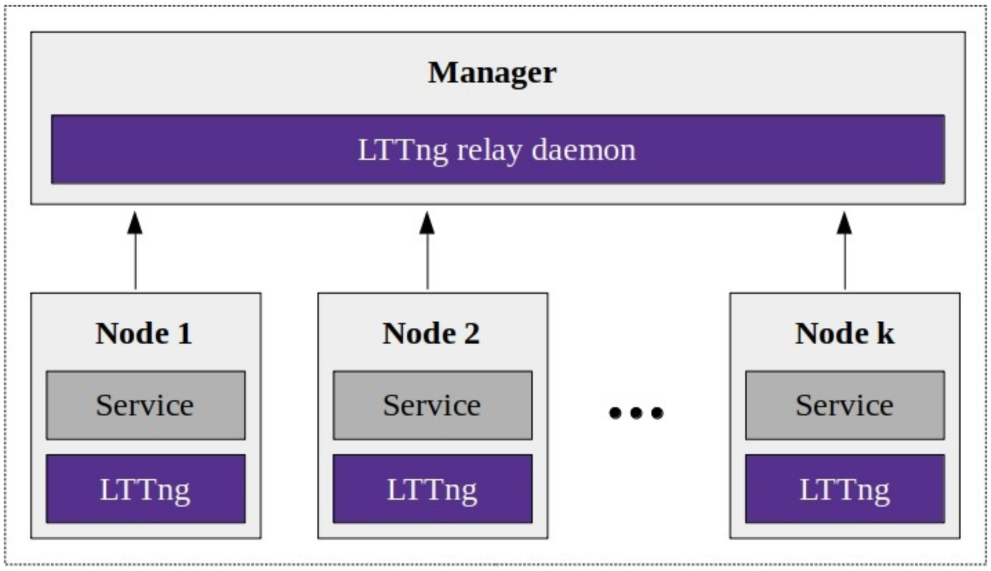


Figure 6.2 The overview of our distributed tracing module.

processed, so as that each event is assigned to the span it belongs. In our framework, events are stored by means of their associated *keys* composed by the name of the event and its arguments.

In order to train a model which is able to detect performance anomalies as well as release-over-release degradations, a massive training dataset is required to cover as many normal patterns of keys as possible. Actually, the training data Γ correspond to entries of traces obtained from the execution of previous stable releases of an application. Figure 6.3 summarizes how to create such a dataset. After collecting n different traces, each of them is processed, so as that all the spans associated with each trace are individually extracted from Γ. Next, for each span, its sequence of events is collected and stored in S_i, for $i = 1, \ldots, m$. In our framework, each sequence S_i is represented by its corresponding keys $\kappa_i^1, \ldots, \kappa_i^{h(S_i)}$, where κ_i^k represents the k-th key in the sequence S_i, and $h(S_i)$ indicates the length of sequence S_i.

Extract spans

In the following, we describe how spans are extracted from an LTTng trace. LTTng uses tracepoints designed to hook mechanisms that can be activated at runtime to record information about programs execution. Tracepoints are placed inside the code by developers or debuggers to extract useful information without the need of knowing the code in-depth. Hence, we can expect to encounter different event types in trace data, indicating the beginning or the end of a span, or any other operation.

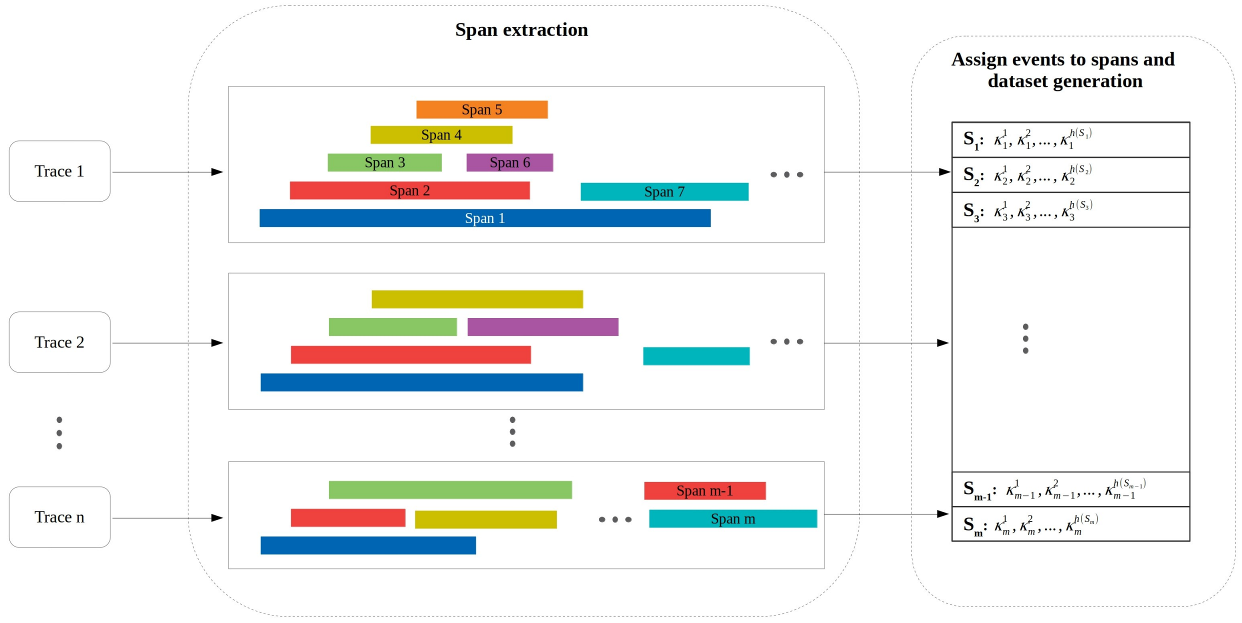

Span extraction
Trace 1
Trace 2
Trace n
Span 5
Span 4
Span 3
Span 6
Span 2
Span 7
Span 1
Span m-1
Span m
Assign events to spans and dataset generation
S_1: $\kappa_1^1, \kappa_1^2, \ldots, \kappa_1^{h(S_1)}$
S_2: $\kappa_2^1, \kappa_2^2, \ldots, \kappa_2^{h(S_2)}$
S_3: $\kappa_3^1, \kappa_3^2, \ldots, \kappa_3^{h(S_3)}$
S_{m-1}: $\kappa_{m-1}^1, \kappa_{m-1}^2, \ldots, \kappa_{m-1}^{h(S_{m-1})}$
S_m: $\kappa_m^1, \kappa_m^2, \ldots, \kappa_m^{h(S_m)}$

Figure 6.3 Illustration of the process for creating the training dataset from multiple traces.

Requests and responses are the two types of events we consider for extracting spans. Each span starts with a request and ends with a response. In addition, the request and response associated with a span possess the same tag. For example, a request with tag 00 indicates the start of a span, whereas a response with the same tag marks the end of that span. Moreover, many sub-spans may be generated during a span's lifetime since a service may communicate with other services to answer a demand. Similar to spans, sub-spans are created with a request and a response that share the same tag. Besides, the parent's tag of each sub-span is embedded in the children's tag. For example, 00/01 indicates a sub-span whose parent is represented by the 00 tag. As shown in Figure 6.4, each span and its sub-spans form a tree. Yet, each span can be displayed as a sequence of requests and responses sorted by their timestamp. In the example of Figure 6.4, this sequence would be $S = \{Req, Req, Resp, Req, Resp, Resp\}$.

Construct sequences of keys

In addition to requests and responses, many other userspace and kernel events happen during each span. After collecting all spans, all events in Γ are processed, and events are assigned to the span to which they belong. The appropriate span for each event is found by comparing the event's arguments (e.g., TID and PID) with the arguments of the events that have been assigned to the spans. Once the appropriated span is identified, the event is placed

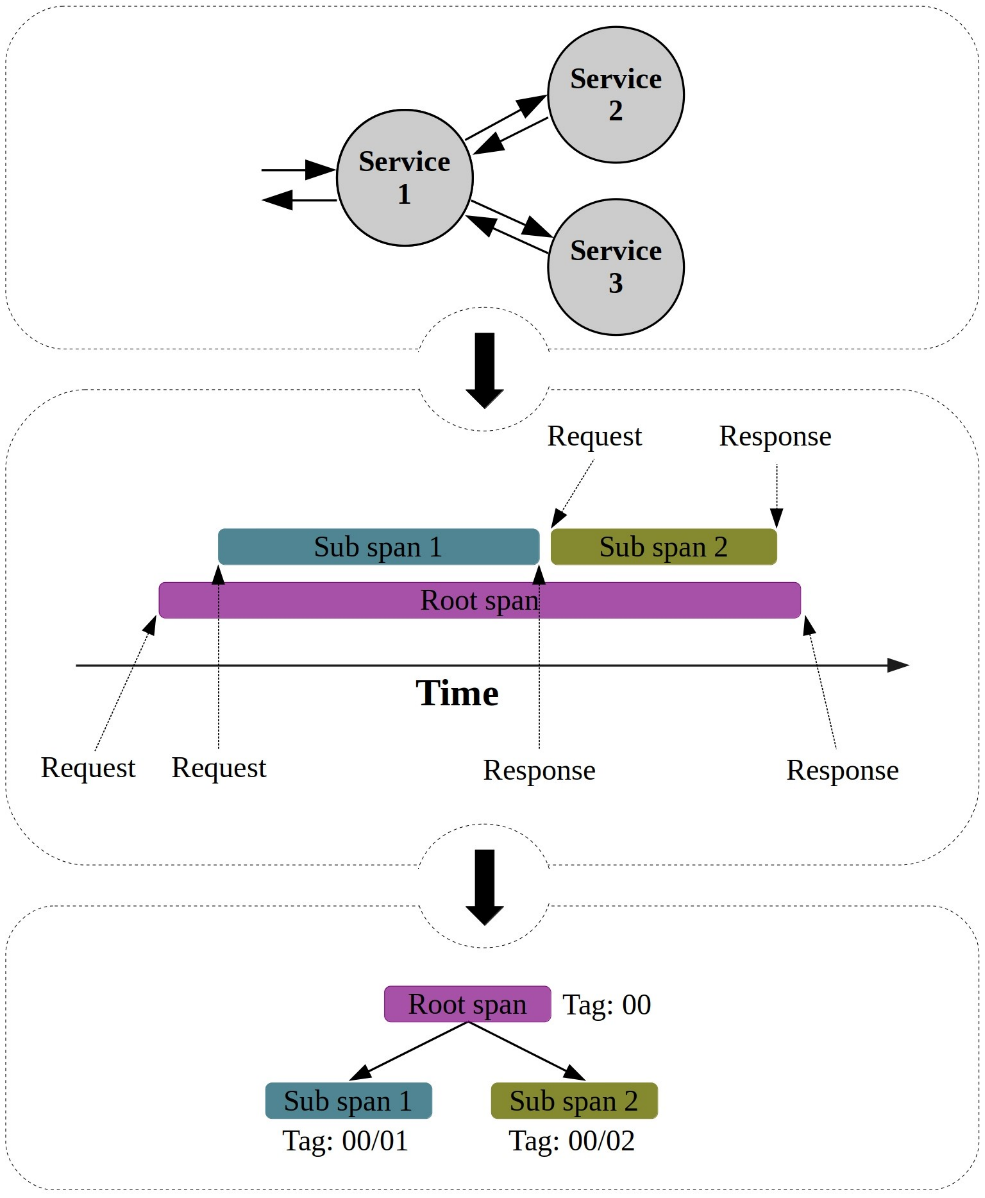


Figure 6.4 The structure of a span and its sub-spans in a distributed trace.

in the sequence according to its timestamp. In the example of Figure 6.4, if an event that happened right after the first request is encountered, the resulting sequence becomes $S = \{Req, Event, Req, Resp, Req, Resp, Resp\}$. This process is repeated for all events so as that a set of sequences is obtained, in which each sequence refers to a span.

The previous paragraph explained how sequences are extracted from a trace. However, it is not yet stated how the arguments of events are used. Whenever a specific tracepoint is encountered at runtime, an event is produced with its arguments such as a name, timestamp, and possibly many others. Event arguments such as process name, message, and event type might contain important information to increase detection quality.

The scope of this work is limited to the arguments that are common to all events. In our experimental traces, event name, process name (Procname), Thread ID (TID), Process ID (PID), timestamp, message, and event type are present in most of the events. We divided the events into two categories: 1) requests/responses, and 2) other events. Table 6.1 lists the arguments we selected for each category of events. The key for requests and responses is created using the name, type, tag, and procname arguments. Event type specifies whether the event is a request or a response, and tag specifies the span or sub-span to which the event belongs. The second category of general events uses the event name, procname, and message arguments to compose the keys. Thus, the resulting keys are all textual strings, where $V = \{v_1, v_2, ..., v_d\}$ denotes the set of all possible unique keys.

Extending our framework to a new argument, albeit simple, may require a much larger dataset depending on the number of values that argument may have. To illustrate, Let us suppose we use only one argument to create keys, and that this argument has β_1 different values. In this case, only β_1 unique keys are created ($d = \beta_1$). If another argument with β_2 different values is then used, $\beta_1 \times \beta_2$ unique keys are obtained ($d = \beta_1 \times \beta_2$). Thus, each time a new argument with β_i different values is considered, the number of unique keys increases β_i times.

Table 6.1 The categories of events and the arguments used by our framework.

Event category	**Argument**	**Type**
Request/Response	Event name	string
	Event type	string
	Tag	string
	Procname	string
Others	Event name	string
	Procname	string
	Message	string

6.4.3 Analysis module

In microservice environments as well as in our experimental application, events are expected to occur in a particular order. Actually, the keys in the sequences obtained by the data extraction module follow specific patterns and grammar rules similar to the ones found in natural languages. It should be noted that the patterns in the data are too complex to be identified using formal languages. Hence, only a few possible keys can appear as the next key in a sequence following a specific set of keys. The training dataset in our experiments includes normal sequences of keys obtained from previous stable releases of the application. In this section, we review the machine learning model we propose to distinguish normal patterns from abnormal ones. We adopted the LSTM network to model this sequence to word problem given its success for modeling text prediction and other similar natural language processing similar tasks. This model learns the probable keys at the moment t according to the previously observed sequences of keys. Later in the detection phase, the model determines which events in a sequence do not conform to normal patterns.

We modeled the anomaly detection problem on our sequences of keys as a multi-class classification problem for which the input length α is fixed. Remark that the sequences obtained by the data extraction module are of different lengths. Multiple sub-sequences of fixed size are hence obtained by considering a window of size α over the larger sequences. It should be noted that sequences smaller than α are very rare in our dataset. In our experimental environment, such sequences are related to small operations that are often not prone to performance anomalies. Consequently, they are simply ignored by the analysis module. Let $V = \{v_1, v_2, ..., v_d\}$ be the set of all possible unique keys, for which each key v_i defines a class. From a sequence of size $h(S_i)$, $h(S_i) - \alpha$ subsequences are analyzed. Thus, for each sequence S_i, the input of the model is denoted by $X_i^j = \kappa_i^j, \kappa_i^{(j+1)}, ..., \kappa_i^{(j+\alpha-1)}$ and the output is expressed by $Y_i^j = \kappa_i^{(j+\alpha)}$, where $j \in 1, ..., h(S_i) - \alpha$. The sequences show a part of a task's execution path in which keys happen in a particular order. Hence, for each X_i^j, Y_i^j can only take a few of the d possible keys from V and is dependent on the sequence X_i^j that appeared before Y_i^j. In other words, the input of the model is a sequence of α recent keys, and the output is a probability distribution over the d keys from V, expressing the probability that the following key in the sequence is $v_r \in V$. Eventually, a model of the conditional probability distribution $Prob(\kappa_i^{j+\alpha} = v_r | \left\{\kappa_i^j, \kappa_i^{(j+1)}, ..., \kappa_i^{(j+\alpha-1)}\right\}), v_r \in V$ is made after the training. Figure 6.5 shows an overview of the described anomaly detection model.

An LSTM network is employed to learn a probability distribution
$Prob(\kappa_i^{j+\alpha} = v_r | \left\{\kappa_i^j, \kappa_i^{(j+1)}, ..., \kappa_i^{(j+\alpha-1)}\right\})$ that maximizes the probability of the training sequences. The architecture of this LSTM network is shown in Figure 6.6. Each layer contains

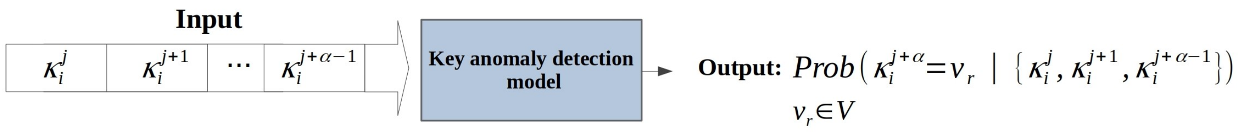

Input
κ_i^j
κ_i^{j+1}
$\cdots$
$\kappa_i^{j+\alpha-1}$
Key anomaly detection model
Output: $\underset{v_r \in V}{Prob}(\kappa_i^{j+\alpha} = v_r \mid \{\kappa_i^j, \kappa_i^{j+1}, \kappa_i^{j+\alpha-1}\})$

Figure 6.5 The overview of our anomaly detection model.

α LSTM blocks, where each block process a key of the input sequence. LSTM blocks have a cell state vector C and a hidden vector H. Both values are moved to the next block to initialize its state. The values of input κ_i^q and H_i^{q-1}, for $q \in \{j, j+1, ..., j+\alpha-1\}$, determine how the current input and the previous output affect that state. They indicate how much of C_i^{q-1} (the previous cell state) holds in the state C_i^q. They also influence the construction of the output H_i^q. Our deep LSTM neural network architecture includes two hidden layers in which the hidden state of the previous layer is used as the input of each corresponding LSTM block in the next layer.

During training, appropriate weights are assigned to input so that the final output of the LSTM provides the desired key. The categorical cross-entropy loss function [132] is used as the loss function for the designed multi-classification task. Then, a standard multinomial logistic function is applied to translate the last hidden state into a probability distribution $Prob(\kappa_i^{j+\alpha} = v_r | \left\{\kappa_i^j, \kappa_i^{(j+1)}, ..., \kappa_i^{(j+\alpha-1)}\right\}, v_r \in V)$.

In the detection phase, the trained model is used to analyze unseen tracing data. This trace can be obtained from an old or a new release of the software. Like what was done to collect the training data, spans are extracted and then converted into sequences of different lengths. Therefore, from a sequence of size $h(S_i)$, $h(S_i) - \alpha$ subsequences are obtained, and $h(S_i) - \alpha$ probability distributions are predicted. The model predicts the probability distribution $Prob(\kappa_i^{j+\alpha} | \left\{\kappa_i^j, \kappa_i^{(j+1)}, ..., \kappa_i^{(j+\alpha-1)}\right\}) = \{v_1 : p_1, v_2 : p_2, ..., v_d : p_d\}$, where p_j describes the probability of v_j to appear as the next key value. Then, $\kappa_i^{j+\alpha}$ is marked as an unexpected key if the probability of the real seen value of $\kappa_i^{j+\alpha}$ is less than the confidence threshold of 0.5. We chose a threshold of 0.5 because the traditional default threshold for interpreting probabilities to class labels is 0.5.

6.5 EVALUATION

In the following, we evaluate the proposed technique by analyzing a microservice-based application. First, the experimental setup and dataset generation are explained in subsection 6.5.1. Then, in subsection 6.5.2 we evaluate the performance of our model. Subsection 6.5.3 an-

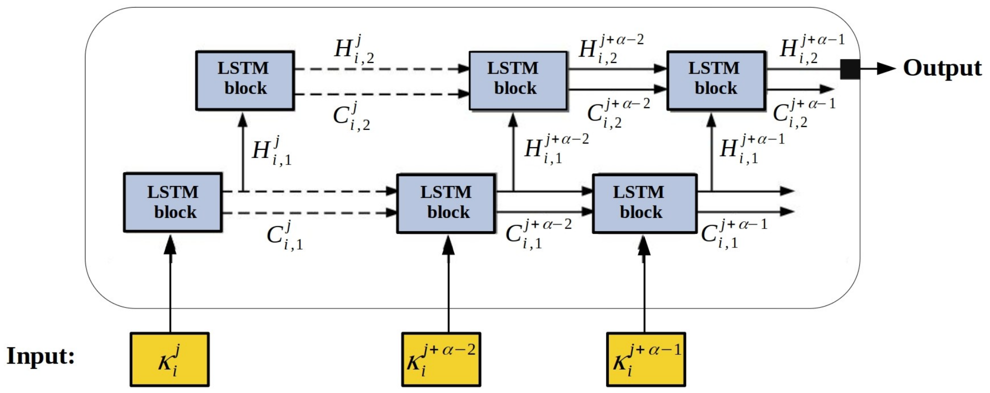


Figure 6.6 The architecture of the LSTM network we used in our anomaly detection framework.

alyzes some practical use-cases and examines the success of our framework in locating the anomalies we injected into the system through various simulated scenarios. Finally, in subsection 6.5.4, we explain how the scripting feature of Trace Compass as well as different views can assist experts to find the root cause of anomalies.

6.5.1 Experimental setup and dataset generation

We deployed the target microservice environment (developed by Ciena Co.) on a virtualized platform with two nodes, each equipped with two cores Intel Core Processor (Broadwell, IBRS), 4 Gb of RAM. An Oracle Linux server was installed on both nodes. Moreover, LTTng was employed on each of them to send the tracing data to the manager. The manager VM benefits from the LTTng-relayd daemon, which is responsible for receiving trace data from remote LTTng daemons.

In order to create the training data, 12 traces with duration between 5 to 10 minutes were obtained from the previous stable releases of the studied software. After removing incomplete spans, a total of 61709 spans were extracted. The dictionary of unique keys collected from the training data contains 4028 unique keys.

Our data collection module has been implemented using python and the Trace Compass Scripting feature [55]. Furthermore, we employed PyTorch to implement the LSTM network[1]. Finally, the model was trained on a server with two Intel(R) Xeon(R) Bronze 3104 1.70GHz CPUs and NVIDIA TITAN V graphic card.

[1]The implementations of the data collection module and the detection module are available in `https://github.com/kohyar/LTTng_LSTM_Anomaly_Detection`.

6.5.2 Evaluation of the anomaly detection framework

As mentioned earlier, we modelled the anomaly detection problem on our sequences of keys as a multi-class classification problem. Thus, there exist 4028 different classes in the training dataset, each one associated to a key. As such, we employed multi-class evaluation metrics to evaluate our model. Precision, recall, and f_score are the metrics we used to evaluate our model. Unlike binary classification, these metrics are obtained for each class separately in a multi-class classification problem. For an individual class C_i, the values of $precision_i$, $recall_i$, and f_score_i are computed as follows:

$$precision_i = \frac{TP_i}{TP_i + FP_i} \tag{6.1}$$

$$recall_i = \frac{TP_i}{TP_i + FN_i} \tag{6.2}$$

$$f\_score_i = \frac{2 \times precision_i \times recall_i}{precision_i + recall_i} \tag{6.3}$$

The *precision*, *recall* and *F_score* for the overall multi-classification problem is then computed by averaging (1), (2) and (3) for the set of classes C_i, with $i = 1, \ldots, 4028$ [133].

As described in section 6.4.3, the sequences obtained from spans are of different lengths. Therefore $h(S_i) - \alpha$ sub-sequences can be obtained by taking a window of size α over a sequence of size $h(S_i)$. To tune the hyperparameter α, we measured the F_score and training time for $\alpha \in \{8, 9, .., 30\}$. The minimum length of sequences in the training data is 8. Also, the results (Figure 6.7) show that for α values greater than 19, the performance decreases. We tried to choose a value for α that would lead to a highly effective model in a reasonable training time. Figure 6.7 presents the *F_score* of the model as well as *F_score/training time* by varying the value of α. According to the results obtained on the training data, $\alpha = 17$ achieves the best classification results, being used in our experiments hereafter.

We evaluated the quality of our model through 10-fold cross-validation. In 10-fold cross-validation, the dataset is divided into ten subsets of approximately equal size. One of the subsets is reserved for testing, while the remaining subsets are used for training. This process is repeated 10 times, and the results are averaged over each one of 10 different tested subsets.

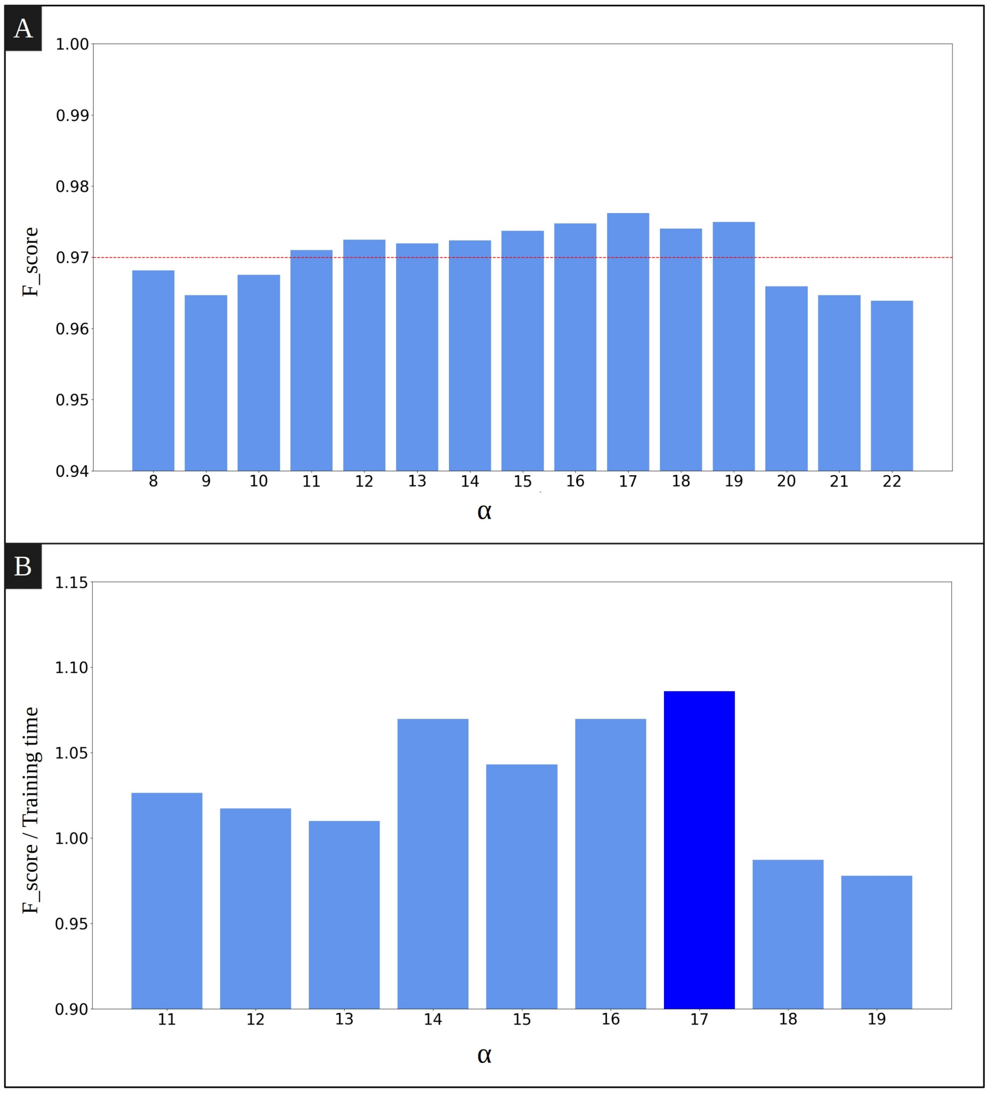


Figure 6.7 A) F_score of the model by varying α. The dotted line indicates that the F_score of the model for α values between 11 and 19 is greater than 0.97. B) $F\_score/Training\ time$ for different values of α. Only the values for which the F_score is greater than 0.97 are shown in this figure.

Results of evaluating our model with 10-fold cross-validation are listed in Table 6.2. It should be noted that our dataset contains approximately 5 million sequences.

Table 6.2 Results of evaluating our model with 10-fold cross- validation.

Precision	0.9774
Recall	0.9760
F__score	0.9759

6.5.3 Analysis of practical use-cases

In this subsection, a newer release of the application was investigated to evaluate the model on detecting possible performance degradations and anomalies. For this purpose, we examined three different scenarios. In the first scenario, the regular execution of the application, i.e., without any anomaly injection, was analyzed to determine where and why the new release did not follow the normal patterns learned by the model. In the other two scenarios, we investigated the performance vulnerability of the new release when an external factor disrupts fair access to system resources. To simulate such attacks, a significant CPU load on the multi-core nodes was generated as the second scenario by continuously compressing and decompressing a stream of random data (zip bombs). Finally, in the third scenario, disk stress was injected into the nodes by creating a file and then using a loop to copy it repeatedly.

Table 6.3 The number of detected unexpected keys along with the total number of predictions made by our model for three scenarios.

	Number of detected unexpected keys	Number of predictions
Test data (Scenario 1)	32043	518503
Test data (Scenario 2)	65507	489593
Test data (Scenario 3)	66978	453781

After collecting the tracing data for each of the mentioned scenarios, the data collection module extracted the spans from these tracing files and created the sequences of keys. For

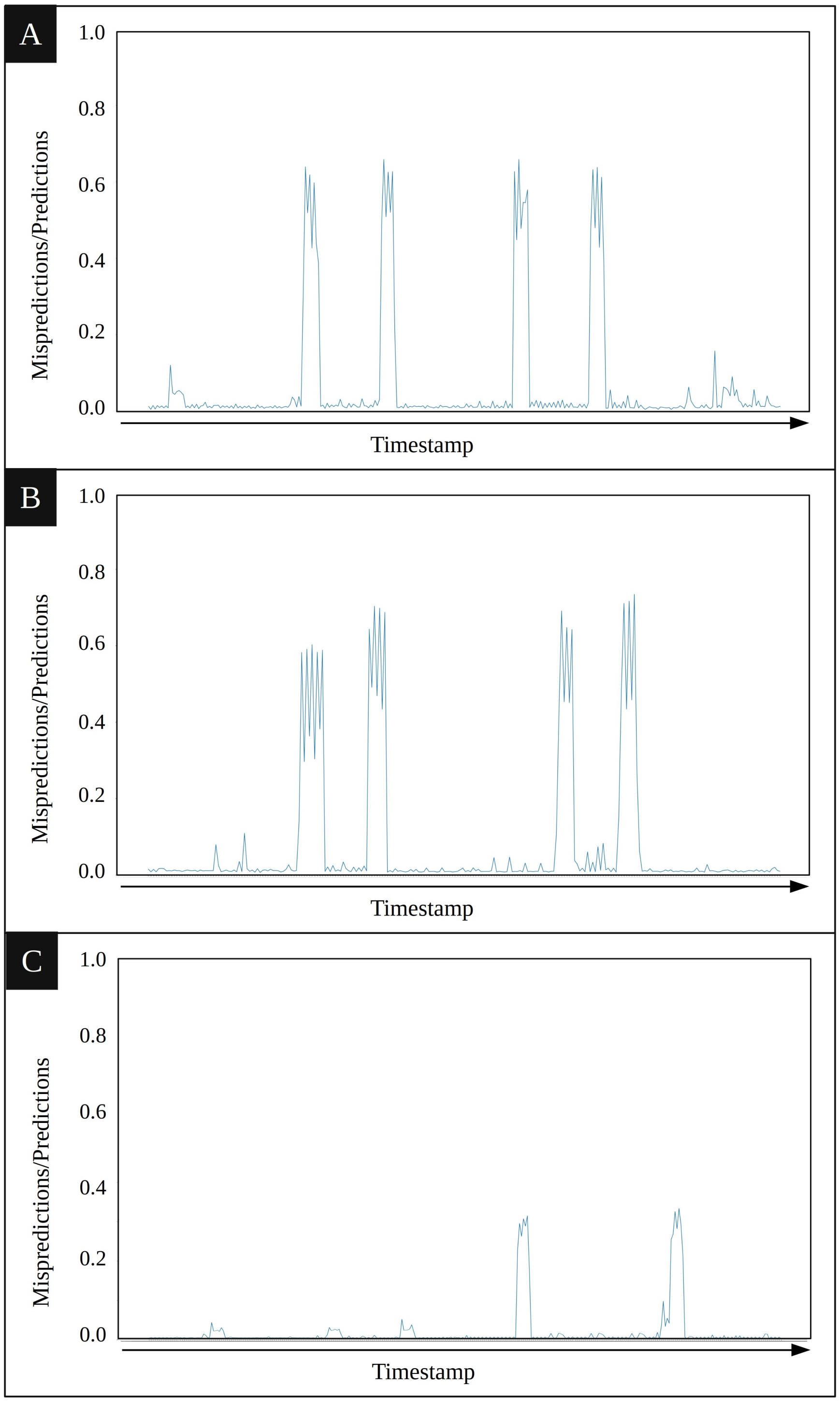


Figure 6.8 This Figure depicts the Likelihood of detecting unexpected keys over the traces obtained from the three mentioned scenarios. A) In this scenario, CPU-related anomalies were injected into the system. B) In this scenario, disk-related anomalies were injected into the system. C) In this scenario, a new release of the application without injecting anomaly was investigated.

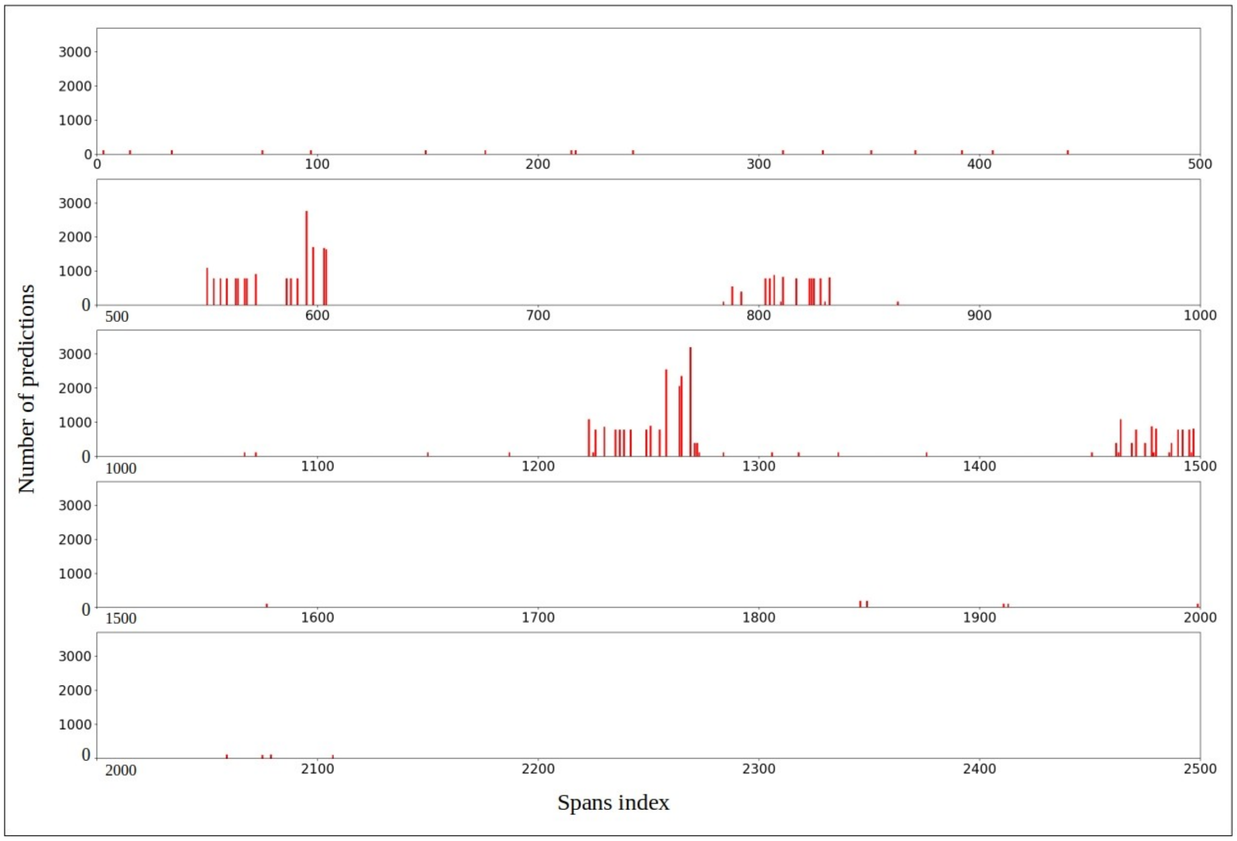


Figure 6.9 The anomalous spans that appeared during the trace of the second scenario where each span has been drawn with a bar.

all input subsequences, the model determines whether the key that appeared in the sequence right after the input subsequence is probable to happen or not. The model marks that key as an *unexpected key* if it predicts that the probability of that key in the sequence is lower than the confidence threshold of 0.5.

Table 6.3 reports the number of detected unexpected keys along with the total number of predictions made by our model for three scenarios. As expected, the number of detected unexpected keys in the first scenario is less than in the other two other scenarios, where CPU and disk stress were injected into the system. The injected load in the second and third scenarios has made the application behave much differently. The first scenario reveals how the changes applied to the application by developers in a newer release may affect the execution's path of the application.

Our proposed framework, however, does not signal keys as anomalous as soon as an unexpected key is detected. It also takes into consideration the frequency of unexpected keys over the monitored period of time. Once a high frequency of unexpected keys is identified, that sequence of keys is highlighted for further investigation by developers or system experts. This is intended to reduce troubleshooting time, as the developers can examine few specific intervals instead of looking at large amounts of tracing data, which might include thousands of system events. As we show next, the output of the model can be examined from two different perspectives.

System-based anomaly detection

In system-based anomaly detection, the entire execution is examined regardless of the span to which each unexpected key belongs. To illustrate, let us consider traces of 5 minutes divided into small time intervals of 1 second. The chart displayed in Figure 6.8 shows the rate of detecting unexpected keys, i.e., number of detected unexpected keys divided by the total amount of predictions, computed in each of the monitored intervals for the three tested scenarios. They reveal the intervals in which more unexpected keys have detected, and are hence, more likely to represent anomalies. This view helps developers to focus only on the areas prone to anomalies. The peaks in Figures 6.8(a) and 6.8(b) correspond exactly to the moments the anomalies were injected into the system, being correctly discovered by our framework. Figure 6.8(c) includes a smaller number of peaks with lower heights. The observed peaks indicate the moments in which the new release did not follow the normal behavior of the previous ones.

Service-based anomaly detection

In service-based anomaly detection, we detect anomalous spans. Unlike system-based anomaly detection, in which we examine the entire execution, service-based anomaly detection identifies spans with a high rate of unexpected keys. The rate of unexpected keys for each span correspond to the rate of unexpected keys in the sequence associated with that span.

A span for which the rate of unexpected keys is greater than 0.5 is marked as an anomalous span. The chart of Figure 6.9 depicts the anomalous spans detected by our framework during the test trace obtained from the second scenario. Spans are numbered according to their start time and are shown with a red bar. In Figure 6.9, the x-axis shows the spans index and, and the y-axis indicates the rate of unexpected keys. From this figure, we observe that many anomalous spans have been detected when anomalies were injected in the system. This view enables developers to filter a trace based on the anomalous spans tag, that merits further investigation.

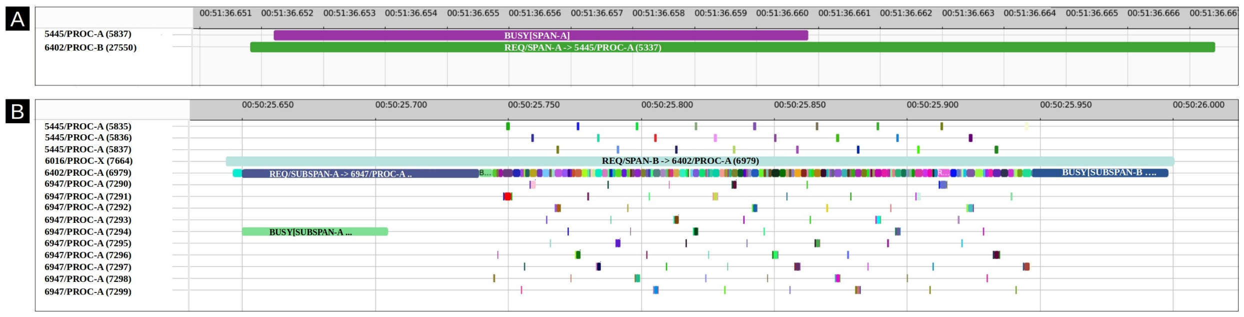


Figure 6.10 This figure presents the time chart generated by our script in Trace Compass, which helped to find the cause of anomalies in our test traces (due to Ciena's security rules, we have changed the original names of the processes in these screenshots). A) A sample of a normal span. B) A sample of an anomalous span where the PROC-X is the caused of the problem.

6.5.4 Root cause analysis

Using Trace Compass in our framework provides the developers an in-depth perception of what happens during a trace, especially in the presence of anomalies. Trace Compass is already used by many companies in the field of performance analysis. Developers can benefit from many scripts and views developed by these companies in Trace Compass for further analysis of their traces. Thus, we have converted the output of our anomaly detection model to Google's Trace Event format to be able to investigate the root causes of the identified anomalies. Our output in this format contains a set of events, each of which equivalent to an event in the original trace. However, three new fields have been added to each event. Field category determines whether the event identified as unexpected. In addition, each event keeps the tag of the span in which it is located, and finally, another field shows if the related span is abnormal or not.

To understand the cause of the anomalies in the introduced test traces, we provided a script that separates all the processes in the trace thereby displaying them with different colors in a time chart like illustrated in Figure 6.10. This time chart can be zoomed in and out in particular areas. Furthermore, the time axis in this time chart is aligned with other views and tables that support automatic time axis alignment, such as the editor view that presents the events in a tabular format or the statistics view that displays the various event counters. More detailed data can be computed from the trace as the user zooms in the time chart or filters events in the editor table.

Figure 6.10(a) shows the structure of a sample normal span. In Figure 6.10(b), the trace was filtered based on the tag of one of the anomalous spans. Interestingly, the provided Trace Compass script could successfully find the cause of a latency issue in the target application that has been detected by our anomaly detection model and led us to the process that caused this problem. This process was present in many other abnormal spans as well. These results demonstrate the effectiveness of our proposed method in locating anomalies and finding their root cause.

6.6 CONCLUSION

Microservice environments provide diverse and enormous services, which necessarily require a stable performance. In this research, a general-purpose NLP-based anomaly detection framework was presented for detecting abnormal behaviors and release-over-release regressions in microservice environments. It works based on recording streams of events using distributed tracing, sending them to the data extraction module so as to create sequences of keys, which are finally analyzed using a deep LSTM model. The model learns a representation of the event names along with their arguments.

The proposed framework is general enough to work with any specific application since no particular assumptions and settings are used in the data collection module. Besides, the whole system needs no prior knowledge. Our framework is also projected to help in the root cause analysis of system issues. The root cause analysis is performed through various plots and scripts that we have provided in Trace Compass. Extensive experiments on real datasets confirm the effectiveness of our approach. Taken together, these findings suggest that our framework is an effective tool that to reduce troubleshooting time by directing the developer to the most relevant problem sites of interest. In the future, we will examine the impact of employing kernel tracing and other arguments of events on the proposed approach. Furthermore, it would be interesting to investigate other NLP techniques to improve detection performance.

CHAPTER 7 GENERAL DISCUSSION

In this thesis, we studied and introduced novel analytical techniques and tools to improve performance anomaly detection and reduce troubleshooting time in systems. In section 7.1, we revisit our objectives and discuss how our research helped achieve those. The broad impact of our work in the field of performance analysis and industry is addressed in Chapter 7.2. Finally, section 7.3 discusses the limitations of our research and suggests potential research projects for future researchers.

7.1 Summary of works and revisiting milestones

Our milestones in this research were discussed earlier in Chapter 3. In this section, we further discuss how our research helped achieve these milestones.

Motivated by the lack of automated performance anomaly detection tools that use low-level tracing data and impose low overhead into the system, Chapter 4 introduces a supervised approach to locate abnormal behaviors during a trace file collected from the execution of an application. This approach monitors the execution of a process by recording its stream of system calls by means of an open-source Linux tracing tool (i.e., LTTng), which is able to provide accurate, detailed information on the kernel and user-space executions. Moreover, we employed the MySQL synthetic benchmarks tool, Sysbench, with oltp test in complex mode to generate the dataset. Finally, we demonstrated the power of the proposed technique by evaluating the model on a dataset of 18k normal and anomalous samples.

On the one hand, preparing a suitable dataset is a significant part of any data science project. On the other hand, collecting labelled data is very difficult due to the nature of the anomaly detection problem. Collecting labelled data is probably not attractive to companies because it is costly and requires a specialist who has a thorough knowledge of the system. For this reason, in Chapter 5, we examined the issue of labelled data in more detail. At first, we improved the performance of our supervised anomaly detection approach for different use cases based on well-known applications like MySQL and Chrome. Then, the performance of unsupervised clustering methods such as K-means and DBSCAN, for cases where labelled data is not available, was evaluated. Our experiments revealed that the performance of DBSCAN was superior to that of K-means, but not as good as that of the proposed supervised approach. Hence, in the next step, we added some labelled data points to the unlabelled dataset. In this case, we applied a supervised feature extraction method, which help discover subsets of features that maximize the underlying clustering tendency. The proposed semi-supervised anomaly detection method demonstrates how this supervision significantly boosts the performance of unsupervised clustering algorithms. Taken together, we proposed a robust framework that offers solutions for different scenarios. In addition, the plots and visualizations provided improve performance analysis by domain experts.

In the continuation of this research (Chapter 6), we examined the performance of microservice-based applications. These complex applications require stable performance, so that users are always satisfied with the provided service. The way we trace microservice-based applications is different, because they consist of services running across multiple hosts. Consequently, it is not possible to rely on an individual trace, and the traces must be collected from multiple nodes. In this work, we have introduced a data collection module in which we used distributed tracing, responsible for recording the requests progress as they travel across multiple nodes and services, communicating over various protocols. In addition, this module is responsible for extracting the spans, which are the primary building blocks in distributed tracing and represent individual units of work. Similar to words in natural languages, events in a span follow specific patterns and grammar rules. We used this idea in our anomaly detection framework and applied an NLP-based strategy to distinguish normal and abnormal event patterns during the spans. Notably, our LSTM model learns the normal patterns of events along with their arguments (e.g., event type, tag, and process name). Moreover, the proposed anomaly detection framework offers some visualizations in Trace Compass, which considerably reduce the troubleshooting time by highlighting the anomalous parts of the trace.

7.2 Research Impact

Complex monolithic and distributed applications are pervasively used in industry, and their performance is of great importance. However, current monitoring tools do not provide enough information to troubleshoot potential performance degradations in these applications. Moreover, finding the root cause of the problems in logging and tracing data, in which hundreds of events occur every second, is an exhausting and time-consuming task. Thus, there is a need for low-overhead tools that can extract meaningful information to solve problems during the execution of the applications. In this research project, several techniques have been proposed to address various shortcomings in the field of performance anomaly detection. We have improved the use of userspace and kernel events by introducing various feature extraction methods that employ events' arguments, frequency, and duration. We have also addressed the problem of the availability of labelled data by introducing supervised, unsupervised, semi-supervised, as well as NLP-based detection techniques. Most importantly, our research covers a variety of software architectures and environments, including monolithic and microservices, and makes troubleshooting these environments faster by providing interactive visualization tools based on Trace Compass.

7.3 Recommendations for Future Research

The techniques presented in Chapters 4, 5, and 6 use various parameters of events as features, such as frequency, duration, and name. Based on our research experience, we recommend that future researchers improve our feature selection mechanism to incorporate other parameters as well. It should be noted that we will often need larger datasets by adding new features. Furthermore, it would be interesting to examine other supervised and unsupervised learning models to achieve better performance in identifying anomalies. We have applied SVM, K-Means, DBSCAN, and LSTM to find abnormal behaviors during this research. In the articles presented during this dissertation, we created some datasets containing normal and abnormal data points through the scenarios described earlier. One suggestion for future work is to enhance these datasets to include other types of anomalies. The proposed detection techniques are expected to be able to find new anomalies, as these techniques are not limited to specific anomalies.

7.4 Limitations

The provided techniques in Chapters 4, and 5 are limited to the system calls name and duration. Some other trace-based methods use insightful details about system calls, such as the events parameters and their interactions. Although these works can identify some specific types of anomalies which are detectable through analysing the parameters, they don't consider the time of data collection in the overall process. However, the data collection is not free and we considered this important point in the proposed approaches in the mentioned chapters. Regarding the approach introduced in Chapter 6, it should be noted that extending this approach to a new argument, albeit simple, may require a much larger dataset.

CHAPTER 8 CONCLUSION

In recent years, the computing infrastructure has significantly evolved, and complex systems have facilitated many complex and large-scale tasks. As a result of this development, distributed environments and cloud infrastructures have emerged, allowing users to access a collection of resources from anywhere and anytime. Moreover, various software architectures, including microservices, have been formed to facilitate the software development process.

These advances have increased the users expectations, and any performance fluctuations may lead to user dissatisfaction and financial loss. Unfortunately, the complexity of new infrastructures and architectures makes them prone to functional anomalies. Hence, monitoring and analyzing the performance of applications, to find any performance degradation, is of particular importance. However, performance analysis of these systems with hundreds of hosts and VMs, is extremely complex.

Tracing data provides information about the execution of applications and processes. However, tracing tools produce a massive amount of low-level raw data, and it is a tedious task for human administrators to manually monitor the execution status of the systems and determine the cause of a performance issue. In this regard, behavioral analysis techniques could be used to automatically analyze the system and reduce troubleshooting time by directing the administrators to the most relevant problem sites.

In our first contribution, the proposed approach records the stream of system calls using the Linux kernel tracing. Unlike previous works, this approach acquires the execution time of system calls in addition to their frequency. Thus, two feature vectors of duration and frequency are created for each extracted sequence of system calls. In order to optimize the SVM model, the Fisher Score feature selection method is applied to select the most discriminative features. Through the promising results obtained from multiple experiments, we demonstrated that the approach we devised could effectively distinguish normal sequences from abnormal ones.

In the second part of the work, we addressed the problem of the availability of labelled data in anomaly detection by proposing a handcrafted data collection module and a multifunctional learning module. In addition, the way we defined the data collection module, which does not use specific settings, makes this framework general enough to work with different applications. This framework introduces supervised and unsupervised techniques that are most compatible with this kind of dataset. Our experiments also revealed that the supervised selection of features could significantly boost the performance of unsupervised learning models, leading us to propose a semi-supervised approach.

In the last part of the work, we proposed a general-purpose NLP-based anomaly detection framework for detecting abnormal behaviors and release-over-release regressions in microservice environments. It records streams of events using distributed tracing and sends them to the data extraction module so as to create sequences of keys. Subsequently, these sequences are analyzed using a deep LSTM model. The LSTM model learns a representation of the event names along with their arguments. Notably, the whole system does not require prior knowledge such as labelled data.

All in all, the findings suggest that our proposed techniques tools, and visualizations effectively reduce troubleshooting time. We recommend that future studies investigate other arguments of events to improve our feature selection mechanism. Future research could also explore other state-of-the-art machine learning techniques to improve anomaly detection performance.

REFERENCES

[1] O. Ibidunmoye, F. Hernández-Rodriguez, and E. Elmroth, “Performance anomaly detection and bottleneck identification,” *ACM Computing Surveys (CSUR)*, vol. 48, no. 1, p. 4, 2015.

[2] “Lttng documentation,” https://lttng.org/docs/v2.10/, accessed: 2019-10-23.

[3] S. Agrawal and J. Agrawal, “Survey on anomaly detection using data mining techniques,” *Procedia Computer Science*, vol. 60, pp. 708–713, 2015.

[4] A. J. Hoglund, K. Hatonen, and A. S. Sorvari, “A computer host-based user anomaly detection system using the self-organizing map,” in *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*, vol. 5, July 2000, pp. 411–416 vol.5.

[5] V. Chandola, A. Banerjee, and V. Kumar, “Anomaly detection: A survey,” *ACM Comput. Surv.*, vol. 41, no. 3, pp. 15:1–15:58, Jul. 2009. [Online]. Available: http://doi.acm.org/10.1145/1541880.1541882

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

[7] A. Silberschatz, G. Gagne, and P. B. Galvin, *Operating system concepts.* Wiley, 2018.

[8] A. S. Tanenbaum and H. Bos, *Modern operating systems.* Pearson, 2015.

[9] M. Gebai and M. R. Dagenais, “Survey and analysis of kernel and userspace tracers on linux: Design, implementation, and overhead,” *ACM Comput. Surv.*, vol. 51, no. 2, pp. 26:1–26:33, Mar. 2018. [Online]. Available: http://doi.acm.org/10.1145/3158644

[10] Y. Wang, K. B. Kent, and G. Johnson, “Improving j9 virtual machine with lttng for efficient and effective tracing,” *Software: Practice and Experience*, vol. 45, no. 7, pp. 973–987, 2015.

[11] A. S. Tanenbaum and H. Bos, *Modern operating systems.* Pearson, 2015.

[12] L. De Lauretis, “From monolithic architecture to microservices architecture,” in *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW).* IEEE, 2019, pp. 93–96.

[13] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, “Microservices: yesterday, today, and tomorrow,” *Present and ulterior software engineering*, pp. 195–216, 2017.

[14] Oracle, “Introduction to virtualization,” http://docs.oracle.com/cd/E20065_01/doc.30/e18549/intro.htm.

[15] I. T. Foster, Y. Zhao, I. Raicu, and S. Lu, “Cloud computing and grid computing 360-degree compared,” *CoRR*, vol. abs/0901.0131, 2009. [Online]. Available: http://arxiv.org/abs/0901.0131

[16] B. Quétier, V. Neri, and F. Cappello, “Scalability comparison of four host virtualization tools,” *Journal of Grid Computing*, vol. 5, no. 1, pp. 83–98, Mar 2007. [Online]. Available: https://doi.org/10.1007/s10723-006-9052-6

[17] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, “Pin: building customized program analysis tools with dynamic instrumentation,” in *Acm sigplan notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.

[18] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics).* Berlin, Heidelberg: Springer-Verlag, 2006.

[19] V. Chandola, A. Banerjee, and V. Kumar, “Anomaly detection: A survey,” *ACM computing surveys (CSUR)*, vol. 41, no. 3, pp. 1–58, 2009.

[20] B. D. Ripley and N. Hjort, *Pattern recognition and neural networks.* Cambridge university press, 1996.

[21] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An introduction to statistical learning.* Springer, 2013, vol. 112.

[22] C. C. Aggarwal, *Data Mining: The Textbook.* Springer Publishing Company, Incorporated, 2015.

[23] A. Ng, “Machine learning and ai via brain simulations,” *Accessed: May*, vol. 3, p. 2018, 2013.

[24] J. K. Muppala, S. P. Woolet, and K. S. Trivedi, "Real-time systems performance in the presence of failures," *Computer*, vol. 24, no. 5, pp. 37–47, 1991.

[25] B. Gregg, *Systems performance: enterprise and the cloud.* Pearson Education, 2013.

[26] L. Cherkasova, K. Ozonat, N. Mi, J. Symons, and E. Smirni, "Automated anomaly detection and performance modeling of enterprise applications," *ACM Transactions on Computer Systems (TOCS)*, vol. 27, no. 3, p. 6, 2009.

[27] Z. He, F. Peters, T. Menzies, and Y. Yang, "Learning from open-source projects: An empirical study on defect prediction," in *2013 ACM/IEEE international symposium on empirical software engineering and measurement.* IEEE, 2013, pp. 45–54.

[28] S. Hou, A. Saas, L. Chen, and Y. Ye, "Deep4maldroid: A deep learning framework for android malware detection based on linux kernel system call graphs," in *2016 IEEE/WIC/ACM International Conference on Web Intelligence Workshops (WIW)*, Oct 2016, pp. 104–111.

[29] B. Kolosnjaji, A. Zarras, G. Webster, and C. Eckert, "Deep learning for classification of malware system call sequences," pp. 137–149, 2016.

[30] F. Maggi, M. Matteucci, and S. Zanero, "Detecting intrusions through system call sequence and argument analysis," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 4, pp. 381–395, Oct 2010.

[31] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen, "Revirt: Enabling intrusion analysis through virtual-machine logging and replay," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 211–224, 2002.

[32] Q. Fu, J.-G. Lou, Y. Wang, and J. Li, "Execution anomaly detection in distributed systems through unstructured log analysis," in *2009 ninth IEEE international conference on data mining.* IEEE, 2009, pp. 149–158.

[33] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing.* John Wiley & Sons, 2011.

[34] "Oprofile." [Online]. Available: https://docs.fedoraproject.org/en-US/fedora/rawhide/system-administrators-guide/monitoring-and-automation/OProfile/

[35] "Lttng v2.13 - lttng documentation." [Online]. Available: https://lttng.org/docs/v2.13/#doc-what-is-tracing

[36] M. Desnoyers, "Low-impact operating system tracing," Ph.D. dissertation, École Polytechnique de Montréal, 2009.

[37] R. McDougall, J. Mauro, and B. Gregg, *Solaris (TM) Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris (Solaris Series).* Prentice Hall PTR, 2006.

[38] "Implementing openstate with ebpf," https://qmonnet.github.io/whirl-offload/2017/02/11/implementing-openstate-with-ebpf/.

[39] S. Rostedt, "ftrace - function tracer," https://www.kernel.org/doc/Documentation/trace/ftrace.txt.

[40] F. C. Eigler and R. Hat, "Problem solving with systemtap," pp. 261–268, 2006.

[41] J. S. Sanz, "Tracking down application bottlenecks with tracers," https://sysdig.com/blog/tracking-down-application-bottlenecks-with-tracers/.

[42] OpenTracing, "Vendor-neutral apis and instrumentation for distributed tracing," accessed: 2021-04-09. [Online]. Available: https://opentracing.io/

[43] Zipkin, "A distributed tracing system," accessed: 2021-04-10. [Online]. Available: https://zipkin.io/

[44] S. Nedelkoski, J. Cardoso, and O. Kao, "Anomaly detection from system tracing data using multimodal deep learning," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD).* IEEE, 2019, pp. 179–186.

[45] T. Wang, W. Zhang, J. Xu, and Z. Gu, "Workflow-aware automatic fault diagnosis for microservice-based applications with statistics," *IEEE Transactions on Network and Service Management*, vol. 17, no. 4, pp. 2350–2363, 2020.

[46] Jaeger, "Open source, end-to-end distributed tracing," accessed: 2021-04-10. [Online]. Available: https://www.jaegertracing.io/

[47] L. Meng, F. Ji, Y. Sun, and T. Wang, "Detecting anomalies in microservices with execution trace comparison," *Future Generation Computer Systems*, vol. 116, pp. 291–301, 2021.

[48] Y. Gan, M. Liang, S. Dev, D. Lo, and C. Delimitrou, "Sage: Using unsupervised learning for scalable performance debugging in microservices," *arXiv preprint arXiv:2101.00267*, 2021.

[49] 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*, 2017, pp. 1285–1298.

[50] J. Wang, Y. Tang, S. He, C. Zhao, P. K. Sharma, O. Alfarraj, and A. Tolba, "Logevent2vec: Logevent-to-vector based anomaly detection for large-scale logs in internet of things," *Sensors*, vol. 20, no. 9, p. 2451, 2020.

[51] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and Weibo Gong, "Anomaly detection using call stack information," in *2003 Symposium on Security and Privacy, 2003.*, May 2003, pp. 62–75.

[52] A. Desai, K. Rajan, and K. Vaswani, "Critical path based performance models for distributed queries," 2012.

[53] D. Nadeau, N. Ezzati-Jivan, and M. R. Dagenais, "Efficient large-scale heterogeneous debugging using dynamic tracing," *Journal of Systems Architecture*, 2019.

[54] E. T. Compass, "Trace compass," 2017, accessed: 2021-12-30. [Online]. Available: https://projects.eclipse.org/projects/tools.tracecompass

[55] "tracecompass-ease-scripting," accessed: 2021-05-10. [Online]. Available: https://archive.eclipse.org/tracecompass.incubator/doc/org.eclipse.tracecompass.incubator.scripting.doc.user/User-Guide.html

[56] "Babeltrace," https://diamon.org/babeltrace/, accessed: 2019-09-1.

[57] "traceshark: This is a tool for linux kernel ftrace and perf events visualization," accessed: 2022-3-4. [Online]. Available: https://github.com/cunctator/traceshark

[58] "System profiling, app tracing and trace analysis - perfetto tracing docs," accessed: 2022-3-4. [Online]. Available: https://perfetto.dev/docs/

[59] "Service trace viewer tool (svctraceviewer.exe) - wcf," accessed: 2022-3-4. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/framework/wcf/service-trace-viewer-tool-svctraceviewer-exe

[60] S. B. Kotsiantis, I. Zaharakis, and P. Pintelas, "Supervised machine learning: A review of classification techniques," *Emerging artificial intelligence applications in computer engineering*, vol. 160, pp. 3–24, 2007.

[61] C. De Stefano, C. Sansone, and M. Vento, “To reject or not to reject: that is the question-an answer in case of neural classifiers,” *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 30, no. 1, pp. 84–94, 2000.

[62] U. Kressel, “Advances in kernel methods, chapter Pairwise classification and support vector machines,” 1999.

[63] U. H.-G. Kre, “Advances in kernel methods,” B. Schölkopf, C. J. C. Burges, and A. J. Smola, Eds. Cambridge, MA, USA: MIT Press, 1999, ch. Pairwise Classification and Support Vector Machines, pp. 255–268. [Online]. Available: http://dl.acm.org/citation.cfm?id=299094.299108

[64] S. Mukkamala, G. Janoski, and A. Sung, “Intrusion detection using neural networks and support vector machines,” in *Proceedings of the 2002 International Joint Conference on Neural Networks. IJCNN'02 (Cat. No.02CH37290)*, vol. 2, May 2002, pp. 1702–1707 vol.2.

[65] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[66] X. Yuan, C. Li, and X. Li, “Deepdefense: identifying ddos attack via deep learning,” in *2017 IEEE International Conference on Smart Computing (SMARTCOMP)*. IEEE, 2017, pp. 1–8.

[67] A. K. Jain, M. N. Murty, and P. J. Flynn, “Data clustering: A review,” *ACM Comput. Surv.*, vol. 31, no. 3, pp. 264–323, Sep. 1999. [Online]. Available: http://doi.acm.org/10.1145/331499.331504

[68] S. P. Lloyd, “Least squares quantization in pcm,” *IEEE Transactions on Information Theory*, vol. 28, pp. 129–137, 1982.

[69] C. Aytekin, X. Ni, F. Cricri, and E. Aksu, “Clustering and unsupervised anomaly detection with l 2 normalized deep auto-encoder representations,” pp. 1–6, 2018.

[70] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, “A density-based algorithm for discovering clusters a density-based algorithm for discovering clusters in large spatial databases with noise,” in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, ser. KDD'96. AAAI Press, 1996, pp. 226–231. [Online]. Available: http://dl.acm.org/citation.cfm?id=3001460.3001507

[71] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton, "Software rejuvenation: Analysis, module and applications," in *Twenty-fifth international symposium on fault-tolerant computing. Digest of papers.* IEEE, 1995, pp. 381–390.

[72] M. Ficco, R. Pietrantuono, and S. Russo, "Aging-related performance anomalies in the apache storm stream processing system," *Future Generation Computer Systems*, vol. 86, pp. 975–994, 2018.

[73] O. Ibidunmoye, F. Hernández-Rodriguez, and E. Elmroth, "Performance anomaly detection and bottleneck identification," *ACM Computing Surveys (CSUR)*, vol. 48, no. 1, pp. 1–35, 2015.

[74] W. Wang and R. Battiti, "Identifying intrusions in computer networks with principal component analysis," pp. 8 pp.–279, 2006.

[75] N. Ye, Y. Zhang, and C. M. Borror, "Robustness of the Markov-chain model for cyber-attack detection," *IEEE Transactions on Reliability*, vol. 53, no. 1, pp. 116–123, 2004.

[76] R. Canzanese, S. Mancoridis, and M. Kam, "System call-based detection of malicious processes," in *2015 IEEE International Conference on Software Quality, Reliability and Security*, Aug 2015, pp. 119–124.

[77] D. J. Dean, H. Nguyen, and X. Gu, "Ubl: Unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems," in *Proceedings of the 9th international conference on Autonomic computing*, 2012, pp. 191–200.

[78] D. J. Dean, P. Wang, X. Gu, W. Enck, and G. Jin, "Automatic server hang bug diagnosis: Feasible reality or pipe dream?" in *2015 IEEE International Conference on Autonomic Computing.* IEEE, 2015, pp. 127–132.

[79] R. Ravichandiran, H. Bannazadeh, and A. Leon-Garcia, "Anomaly detection using resource behaviour analysis for autoscaling systems," in *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft).* IEEE, 2018, pp. 192–196.

[80] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 117–132.

[81] D. J. Dean, H. Nguyen, X. Gu, H. Zhang, J. Rhee, N. Arora, and G. Jiang, "Perfscope: Practical online server performance bug inference in production cloud computing infrastructures," in *Proceedings of the ACM Symposium on Cloud Computing*, 2014, pp. 1–13.

[82] Y. Shkuro, *Mastering Distributed Tracing: Analyzing performance in microservices and complex systems.* Packt Publishing Ltd, 2019.

[83] 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*, 2015, pp. 1939–1947.

[84] R. Ravichandiran, H. Bannazadeh, and A. Leon-Garcia, “Anomaly detection using resource behaviour analysis for autoscaling systems,” in *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft).* IEEE, 2018, pp. 192–196.

[85] E. Gaidels and M. Kirikova, “Service dependency graph analysis in microservice architecture,” in *International Conference on Business Informatics Research.* Springer, 2020, pp. 128–139.

[86] T. Wang, W. Zhang, J. Xu, and Z. Gu, “Workflow-aware automatic fault diagnosis for microservice-based applications with statistics,” *IEEE Transactions on Network and Service Management*, vol. 17, no. 4, pp. 2350–2363, 2020.

[87] A. Patcha and J.-M. Park, “An overview of anomaly detection techniques: Existing solutions and latest technological trends,” *Computer networks*, vol. 51, no. 12, pp. 3448–3470, 2007.

[88] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar, and M. Steinder, “Performance evaluation of microservices architectures using containers,” in *2015 IEEE 14th International Symposium on Network Computing and Applications.* IEEE, 2015, pp. 27–34.

[89] P. Garcia-Teodoro, J. Diaz-Verdejo, G. Maciá-Fernández, and E. Vázquez, “Anomaly-based network intrusion detection: Techniques, systems and challenges,” *computers security*, vol. 28, no. 1-2, pp. 18–28, 2009.

[90] N. Ye, “Probabilistic networks with undirected links for anomaly detection,” 2000.

[91] I. L. MacDonald and W. Zucchini, *Hidden Markov and other models for discrete-valued time series.* CRC Press, 1997, vol. 110.

[92] W. G. Syarif I., Prugel-Bennett A., “Data mining approaches for network intrusion detection from dimensionality reduction to misuse and anomaly detection,” *Journal of Information Technology Review*, vol. 3, no. 2, pp. 70–83, 2012.

[93] N. Kaur, "Survey paper on data mining techniques of intrusion detection," *International Journal of Science, Engineering and Technology Research*, vol. 2, no. 4, pp. 799–804, 2013.

[94] X. Wang, Q. Zhou, J. Harer, G. Brown, S. Qiu, Z. Dou, J. Wang, A. Hinton, C. A. Gonzalez, and P. Chin, "Deep learning-based classification and anomaly detection of side-channel signals," p. 1063006, 2018.

[95] S. Nedelkoski, J. Cardoso, and O. Kao, "Anomaly detection from system tracing data using multimodal deep learning," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 2019, pp. 179–186.

[96] S. Ji, W. Wu, and Y. Pu, "Multi-indicators prediction in microservice using granger causality test and attention lstm," in *2020 IEEE World Congress on Services (SERVICES)*. IEEE, 2020, pp. 77–82.

[97] P. Malhotra, L. Vig, G. Shroff, and P. Agarwal, "Long short term memory networks for anomaly detection in time series," in *Proceedings*, vol. 89. Presses universitaires de Louvain, 2015, pp. 89–94.

[98] M. Bacher., M. Bacher., I. Ben-Gal., and E. Shmueli., "An information theory subspace analysis approach with application to anomaly detection ensembles," in *Proceedings of the 9th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management - Volume 1: KDIR,*, INSTICC. SciTePress, 2017, pp. 27–39.

[99] P. Animesh and M. Jung, "Network anomaly detection with incomplete audit data," *Elsevier Science*, pp. 5–35, 2007.

[100] R. Ranjan and G. Sahoo, "A new clustering approach for anomaly intrusion detection," *arXiv preprint arXiv:1404.2772*, 2014.

[101] J. Ohlsson, "Anomaly detection in microservice infrastructures," 2018.

[102] S. M. Varghese and K. P. Jacob, "Anomaly detection using system call sequence sets," *Journal of software*, vol. 2, no. 6, pp. 14–21, 2007.

[103] A. Liu, C. Martin, T. Hetherington, and S. Matzner, "A comparison of system call feature representations for insider threat detection," in *Proceedings from the Sixth Annual IEEE SMC Information Assurance Workshop*, June 2005, pp. 340–347.

[104] D.-K. Kang, D. Fuller, and V. Honavar, “Learning classifiers for misuse and anomaly detection using a bag of system calls representation,” in *Proceedings from the Sixth Annual IEEE SMC Information Assurance Workshop*, June 2005, pp. 118–125.

[105] Z. Xu, X. Yu, Y. Feng, J. Hu, Z. Tari, and F. Han, “A multi-module anomaly detection scheme based on system call prediction,” in *2013 IEEE 8th Conference on Industrial Electronics and Applications (ICIEA)*, June 2013, pp. 1376–1381.

[106] M. Desnoyers and M. Dagenais, “The lttng tracer : A low impact performance and behavior monitor for gnu / linux,” in *OLS Ottawa Linux Symposium*, 2006.

[107] C. Yuan, N. Lao, J.-R. Wen, J. Li, Z. Zhang, Y.-M. Wang, and W.-Y. Ma, “Automated known problem diagnosis with event traces,” pp. 375–388, 2006.

[108] D. Liu, Y. Zhao, H. Xu, Y. Sun, D. Pei, J. Luo, X. Jing, and M. Feng, “Opprentice: Towards practical and automatic anomaly detection through machine learning,” in *Proceedings of the 2015 Internet Measurement Conference*, ser. IMC ’15. New York, NY, USA: ACM, 2015, pp. 211–224. [Online]. Available: http://doi.acm.org/10.1145/2815675.2815679

[109] S. Suratkar, F. Kazi, R. Gaikwad, A. Shete, R. Kabra, and S. Khirsagar, “Multi hidden markov models for improved anomaly detection using system call analysis,” in *2019 IEEE Bombay Section Signature Conference (IBSSC)*. IEEE, 2019, pp. 1–6.

[110] M. Desnoyers and M. Dagenais, “The lttng tracer : A low impact performance and behavior monitor for gnu / linux,” in *OLS Ottawa Linux Symposium*, 2006.

[111] J. Cai, J. Luo, S. Wang, and S. Yang, “Feature selection in machine learning: A new perspective,” *Neurocomputing*, vol. 300, pp. 70–79, 2018.

[112] L. Sayfullina, “Reducing Sparsity in Sentiment Analysis Data using Novel Dimensionality Reduction Approaches,” 2014.

[113] J. Alvarez Cid-Fuentes, C. Szabo, and K. Falkner, “Adaptive performance anomaly detection in distributed systems using online svms,” *IEEE Transactions on Dependable and Secure Computing*, pp. 1–1, 2018.

[114] A. Campbell, K. Caudle, and R. C. Hoover, “Examining Intermediate Data Reduction Algorithms for use with t-SNE,” pp. 36–42, 2019.

[115] S. Wold, K. Esbensen, and P. Geladi, “Principal component analysis,” *Chemometrics and intelligent laboratory systems*, vol. 2, no. 1-3, pp. 37–52, 1987.

[116] C. C. Aggarwal, *Data Mining: The Textbook.* Springer Publishing Company, Incorporated, 2015.

[117] C. I. King, "Stress-ng," *URL: http://kernel. ubuntu. com/git/cking/stressng. git/(visited on 28/03/2018)*, 2017.

[118] A. ledenev, "Pumba-chaos testing and network emulation tool for docker," 2019, accessed: 2020-08-23. [Online]. Available: https://github.com/alexei-led/pumba

[119] "Rbf svm parameters," https://scikit-learn.org/stable/auto_examples/svm/plot_rbf_parameters.html, accessed: 2019-10-21.

[120] W. M. Rand, "Objective criteria for the evaluation of clustering methods," *Journal of the American Statistical association*, vol. 66, no. 336, pp. 846–850, 1971.

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

[122] Y. Gan, Y. Zhang, K. Hu, D. Cheng, Y. He, M. Pancholi, and C. Delimitrou, "Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices," in *Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems*, 2019, pp. 19–33.

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

[124] L. Wu, J. Tordsson, E. Elmroth, and O. Kao, "Microrca: Root cause localization of performance issues in microservices," in *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium.* IEEE, 2020, pp. 1–9.

[125] M. Solé, V. Muntés-Mulero, A. I. Rana, and G. Estrada, "Survey on models and techniques for root-cause analysis," *arXiv preprint arXiv:1701.08546*, 2017.

[126] Ú. Erlingsson, M. Peinado, S. Peter, M. Budiu, and G. Mainar-Ruiz, "Fay: Extensible distributed tracing from kernels to clusters," *ACM Transactions on Computer Systems (TOCS)*, vol. 30, no. 4, pp. 1–35, 2012.

[127] R. Gassais, N. Ezzati-Jivan, J. M. Fernandez, D. Aloise, and M. R. Dagenais, "Multi-level host-based intrusion detection system for internet of things," *Journal of Cloud Computing*, vol. 9, no. 1, pp. 1–16, 2020.

[128] A. Samir and C. Pahl, "Dla: Detecting and localizing anomalies in containerized microservice architectures using markov models," in *2019 7th International Conference on Future Internet of Things and Cloud (FiCloud).* IEEE, 2019, pp. 205–213.

[129] A. Abusitta, M. Bellaiche, and M. Dagenais, “An svm-based framework for detecting dos attacks in virtualized clouds under changing environment,” *Journal of Cloud Computing*, vol. 7, no. 1, pp. 1–18, 2018.

[130] M. F. Elrawy, A. I. Awad, and H. F. Hamed, “Intrusion detection systems for iot-based smart environments: a survey,” *Journal of Cloud Computing*, vol. 7, no. 1, pp. 1–20, 2018.

[131] Q. Fournier, D. Aloise, S. V. Azhari, and F. Tetreault, “On improving deep learning trace analysis with system call arguments,” in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 120–130.

[132] Z. Zhang and M. R. Sabuncu, “Generalized cross entropy loss for training deep neural networks with noisy labels,” in *32nd Conference on Neural Information Processing Systems (NeurIPS)*, 2018.

[133] M. Sokolova and G. Lapalme, “A systematic analysis of performance measures for classification tasks,” *Information processing & management*, vol. 45, no. 4, pp. 427–437, 2009.