Performance Anomaly Detection through Sequence Alignment of System-Level Traces

Madeline Janecek
mj17th@brocku.ca
Brock University
St. Catharines, ON, Canada

Naser Ezzati-Jivan
nezzati@brocku.ca
Brock University
St. Catharines, ON, Canada

Abdelwahab Hamou-Lhadj
wahab.hamou-lhadj@concordia.ca
Concordia University
Montreal, QC, Canada

ABSTRACT

Identifying and diagnosing performance anomalies is essential for maintaining software quality, yet it can be a complex and time-consuming task. Low level kernel events have been used as an excellent data source to monitor performance, but raw trace data is often too large to easily conduct effective analyses. To address this shortcoming, in this paper, we propose a framework for uncovering performance problems using execution critical path data. A critical path is the longest execution sequence without wait delays, and it can provide valuable insight into a program's internal and external dependencies. Upon extracting this data, course grained anomaly detection techniques are employed to determine if a finer grained analysis is required. If this is the case, the critical paths of individual executions are grouped together with machine learning clustering to identify different execution types, and outlying anomalies are identified using performance indicators. Finally, multiple sequence alignment is used to pinpoint specific abnormalities in the identified anomalous executions, allowing for improved application performance diagnosis and overall program comprehension.

KEYWORDS

Anomaly Detection, Execution Tracing, Machine Learning, Multiple Sequence Alignment, Performance Analysis, Software Tracing

ACM Reference Format:
Madeline Janecek, Naser Ezzati-Jivan, and Abdelwahab Hamou-Lhadj. 2022. Performance Anomaly Detection through Sequence Alignment of System-Level Traces. In *30th International Conference on Program Comprehension (ICPC '22), May 16–17, 2022, Virtual Event, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3524610.3527898

1 INTRODUCTION

As a software system grows to be more complex, performance anomaly detection and analysis becomes increasingly difficult and time consuming. An application's execution flow, and by extension its performance, may be affected by a number of different factors, including software bugs, updates, improper resource management, and hardware failures. Furthermore, user expectations for performance are also increasing, which means that failure to detect and remedy the cause of performance anomalies can quickly lead to user dissatisfaction and possible financial losses. As such, there is a need for tools to assist developers in understanding their software's execution behaviour and locating specific locations of performance anomalies.

The collection and interpretation of system calls through execution tracing is a well established practice within general anomaly detection [18, 24]. As the fundamental interface between processes and the kernel, system calls can give a detailed picture of system execution. In general, kernel tracing is suitable for the analysis of runtime behaviors of any systems that run on the kernel. However, considering the number of events that need to be recorded, this method tends to introduce high collection overhead, negatively impacting performance and analysis accuracy. Furthermore, when considering the sheer size of the collected trace data, it can be difficult to properly and efficiently analyze the program's execution behaviour.

This led us to explore critical path analysis for performance anomaly detection. The critical path is defined as the longest execution sequence, and thus it is made up of the elements that contribute to the process's overall latency [32]. Metrics like CPU or disk usage, which are also common data sources for software performance evaluation [2], may characterize an execution's resource utilization, whereas critical path analysis can show when and why an execution waits for such resources. The critical path extraction algorithm has the added benefit of only requiring a few kernel events, making it a cheaper alternative to collecting system calls. In Section 4 we will show how collecting each of the approximately 600 kernel events has a much greater impact on overall performance than when only collecting the events that are necessary for critical path extraction. Accordingly, analysing a critical path is a cheaper alternative that still gives us an enhanced understanding of which elements would contribute to user-visible performance issues.

For instance, we had several containers made with Docker 10.10.7, each of which were made with Apache HTTP Server 2.4.51 images. One of these web servers took, on average, 10.9% longer to handle web requests. Examining the container's initial configuration showed that the anomalous performance could be attributed to the fact that the container was connected to a bridge network. Unless otherwise specified, Docker containers connect newly-started containers to an automatically generated bridge network. This meant that the requests handled by this Apache HTTP Server had to go through another level of virtualization through the docker daemon. The other containers, which were set to use Docker's `host` network mode, were using the host's native network stack directly, which reduced their isolation but lead to better overall performance. When examining the critical paths of requests from web servers with both

ICPC '22, May 16–17, 2022, Virtual Event, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9298-3/22/05…$15.00
https://doi.org/10.1145/3524610.3527898

configurations, we saw that there were differences in the paths' structure and duration. By automating the process of identifying these differences, we could significantly reduce the time and effort involved in troubleshooting.

In this paper, we therefore propose a critical path based performance anomaly detection framework which is essential for program comprehension and analysis. Firstly, trace data is collected using an open-source tracing tool to make up a normal and sample data set. As we use kernel tracing, this is a black box method that does not require additional instrumentation. The data is divided into individual executions using delineating trace events. The critical path of each execution is extracted and then used to generate two types of critical path vectors. The first type, critical path duration vectors, represent the execution's performance. The second type, critical path count vectors, are used to infer the execution's critical path structure. Anomaly detection techniques are employed to see if anomalous executions are present in the sample data. If this is the case, clustering techniques are used to identify the different execution types for fine grained anomaly localization. Multiple sequence alignment is then used to highlight specific anomalies in the identified executions.

The purpose of the proposed approach is to detect performance degradation through the analysis of kernel traces. One advantage of analyzing kernel traces is that we can detect performance problems in various layers of the software stack that run on the kernel including standalone applications, server systems, virtual machines, and so forth. There is a great deal of literature that covers the importance of kernel trace in runtime analysis of software systems. Our approach runs offline (i.e., post-mortem analysis), and the intent is to help software analysts to understand the root causes of performance degradation. This is not intended to automatically predict potential performance problems while the system is running. The approach can be used for in-house as well as field testing. It can also be applied by system operators to pinpoint the causes of performance degradation.

The main contributions of this work are as follows: First, we outline our solution for performance anomaly detection. Our method is entirely reliant on execution trace data, and as such can provide developers with a detailed analysis of performance without any knowledge requirements of the software's internals. Second, we use critical path analysis to improve upon methods that rely on system call data in accuracy and imposed overhead. Third, we propose multiple sequence alignment, a technique commonly used in the life sciences [11], as an effective method to specifically pinpoint differences in anomalous executions, thereby simplifying the software debugging process.

The remaining parts of this paper are as follows: Firstly, Section 2 presents prior related works. After that, Section 3 describes each step of the proposed framework in detail. Then, Section 4 discusses the experimental evaluation of our approach using two case studies. Finally, Section 5 concludes this paper and discusses plans for future work.

2 RELATED WORK

Several previous works have explored different methods for performance anomaly detection. The earliest of such works tend to focus on statistical profiling methods [28]. However, statistical approaches typically do not provide any insight as to the root cause of an identified anomaly, but instead work to just label data as normal or anomalous. Other machine learning based techniques using Markov models, neural networks, and so forth have also been examined in previous works [13]. While these approaches present several advantages over the statistical approaches, they are often much more computationally expensive.

System call data is a common data source for performance anomaly detection techniques. Tracing based methods have also been utilized for different kinds of software analyses, including performance monitoring [23], intrusion detection [21, 24], root cause detection [4], and program comprehension [6, 29]. Tracing tools are capable of providing a fine grained picture of software execution while imposing minimal overhead, thereby allowing for detailed analyses. They do this by recording different runtime events at predefined tracepoints. Resource utilization metrics like CPU, memory, disk, and file usages may also be derived from kernel traces using methods described by Giraldeau *et al.* in [15]. The resulting trace file may contain millions of events, and as such machine learning algorithms like LSTM [26], ELM [7] and classifiers [36] are often used in trace data analysis.

One method to interpret trace data is pairwise sequence alignment, which is a technique originally used in life sciences to find common subsequences in RNA, DNA, and proteins [11]. Other works have explored applying these alignments methods to automatically compare of trace data [34, 35], which when done manually is a time consuming and error-prone task. One shortcoming of these existing studies is that they only compare two traces and highlight their differences. This can be problematic as it can be challenging to find a single trace that perfectly exemplifies a program's normal performance, especially considering that performance variations may occur without being anomalous.

To address this issue, we explored the use of multiple sequence alignment techniques. Multiple sequence alignment is the process of aligning several sequences, and consequently it is more computationally complex than pairwise alignment. There are several multiple sequence alignment methods made for biological sequence analysis, such as CLUSTALW [33], T-Coffee [27], and Progressive POA [16]. In [11], Edgar presents MUltiple Sequence Comparison by Log- Expectation (MUSCLE) as the ideal multiple sequence alignment method in both space and time complexity. Thus, we have chosen to use the MUSCLE algorithm within our framework. This allows us to compare anomalous traces to a group of several normal traces, giving a more thorough analysis than pairwise sequence alignment.

Clustering techniques are also common in trace data analysis. For instance in [22], Kohyarnejadfard *et al.* utilize DBSCAN clustering to discover anomalous subsequences of system calls. In [30], Rhee *et al.* generate system call distribution vectors and use a hierarchical clustering approach to find groups of trace data resulting from the same user code. In our performance anomaly detection framework, we use machine learning clustering to discover different execution types, however instead of using system call data we turn to critical path analysis.

Critical path analysis has already been established as an effective tool in general program performance analysis. It is often used to

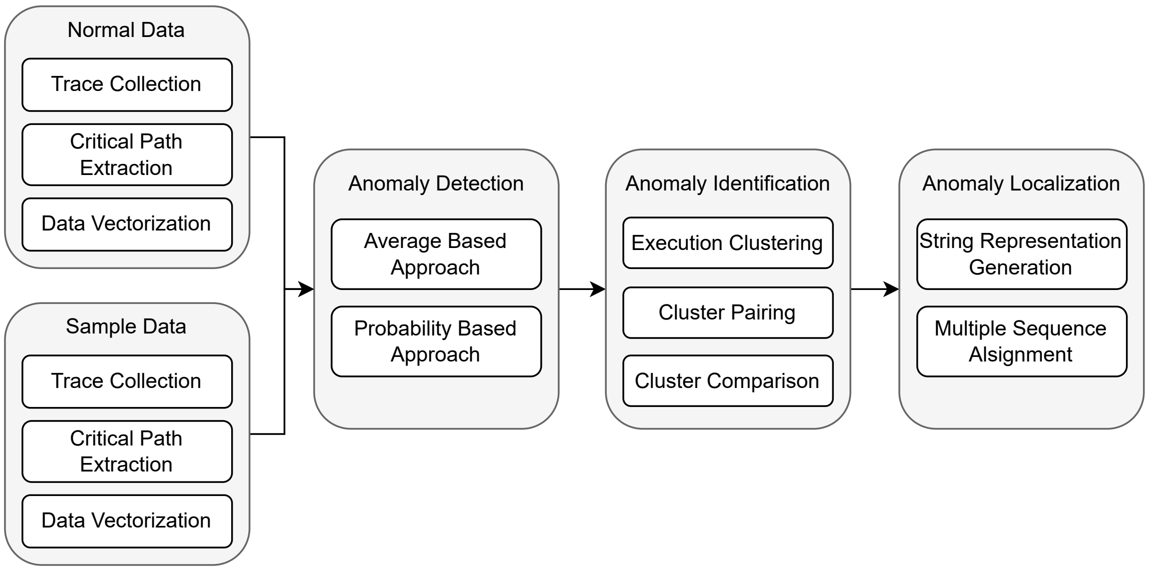


Figure 1: The proposed framework.

help developers interpret raw trace data, as deciphering possibly millions of trace events is often non-trivial task. For instance, Hendriks *et al.* [17] discuss how they can compare the critical paths of different executions to identify performance bottlenecks. In [31], Sambasivan *et al.* present Spectroscope, which is a tool used to extract critical paths to diagnose the cause of response-time mutations within distributed systems. This is presented as a powerful solution for identifying performance changes between two system versions or time periods. In [14], Giraldeau *et al.* discuss how critical path analysis can improve execution latency and diagnose the root cause of performance problems.

In this paper, we work to combine the advantages seen with critical path analysis with multiple sequence alignment techniques. This results in a thorough model of normal execution that can be used to point developers to specific differences between the software's normal execution and anomalies identified using machine learning clustering.

3 METHODOLOGY

The process of detecting and investigating a software's performance anomalies can be complicated and time consuming. To assist developers with this tedious task, we propose a performance anomaly detection framework that operates using execution trace data. This framework requires two different sets of trace data as input, which will be referred to as the normal and sample data sets. The normal data set is used to model the software's typical performance. The sample data set is then compared to the normal data set to check for any anomalies.

An overview of the proposed framework is shown in Figure 1. First, trace data is collected to generate the two data sets. Delineating events are then used to break up the trace data into individual executions. Each execution's critical path is then extracted and vectorized. Average based and probability based methods are employed to determine if there are anomalies present in the sample data set. If this is the case, we conduct a fine grained analysis to identify specific anomalies. Clustering methods are employed to identity the different execution types present in both the normal and sample data. Clusters from the sample data set are compared to their most similar clusters in the normal data set to identity outliers, and multiple sequence alignment is used to pinpoint the differences.

3.1 Trace Collection

Linux Tracing Toolkit: next generation (LTTng) [9] is a lightweight open source tracing tool that is capable of extracting detailed information from multiple software layers, while imposing minimal overhead. Within this context, LTTng is used to record different events from the kernel-level, which are then used to provide valuable insights as to what occurs within a system at runtime. Raw trace data consists of a sequence of timestamped events taken from several concurrently running threads. At each event various metadata fields known as contexts are recorded. These contexts can be the timestamp, process ID (PID), event types, and so forth. The Linux kernel has tracepoints pre-inserted, and therefore in a Linux system these kernel-level tracepoints do not require any additional instrumentation.

In this work, we use the term *execution* to describe any tasks or behaviour of interest. Our methods are meant to be applicable for several different scenarios. Thus, an execution could be a function call, the compilation of an application, a view rendering, and so on. In order to conduct a meaningful analysis, the raw trace data must first be divided into individual executions. To do this, delineating events are also collected to indicate the start and end of an execution. For some types of executions these can be kernel-level events, however in some cases collecting data from the userspace

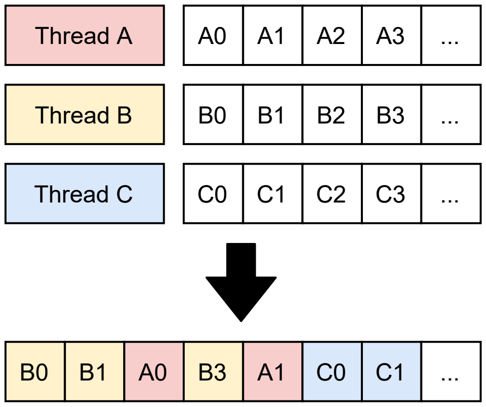


Figure 2: A visualization of an interleaved events compared to the individual threads' true ordering

level might be required. As an example, when examining trace data collected from an Apache web server where an execution is a web request, the userspace events `request_start` and `request_exit` may be recorded to indicate when an execution starts and ends, respectively [12].

We trace several executions to make up two distinct data sets: the normal data set and the sample data set. The normal data set is collected to model the software when it is exhibiting normal performance behaviour. This process is then repeated to generate the sample data set. The sample data set might be collected from the software running in a new environment, a new version of the software, and so forth. Making a comparison between the two data sets' performance will allow us to check if any performance impacting anomalies have been introduced, and if this is the case help to identify the root cause.

3.2 Critical Path Extraction

Upon collecting trace data for both the normal and sample data sets, we extract the critical path of each execution. An execution's critical path is defined as its longest execution sequence [32], and thus it shows which elements have contributed to the execution's overall latency. The benefits of examining critical paths for the purpose of performance analysis are threefold. Firstly, a critical path gives a compact yet comprehensive representation of all the interactions between a thread and other threads running simultaneously within the system. Secondly, simply combining the trace events of multiple threads into interleaved sequences can break the true per-thread ordering result in an inaccurate representation of program execution, as shown in Figure 2. Hence extracting and analysing the critical path gives a more accurate representation of execution events. Finally, to extract an execution's critical path we only need to collect a small number of trace events, which imposes far less overhead than enabling all kernel events while tracing. The required kernel level events are as follows [14]:

- `sched_switch` indicates that a new thread has replaced a different thread that had been previously running on a CPU.
- `sched_wakeup` indicates that a previously blocked thread is now ready to run.
- `irq_handler_entry` indicates that the hardware interrupt handler has started its execution.
- `irq_handler_exit` indicates that the hardware interrupt handler has ended its execution.
- `hrtimer_expire_entry` indicates when a high resolution timer interrupt has started its execution.
- `hrtimer_expire_exit` indicates when a high resolution timer interrupt has completed execution.
- `softirq_entry` indicates that the software interrupt handler has started its execution.
- `softirq_exit` indicates that the software interrupt handler has ended its execution.

To determine an execution's critical path, we modify an algorithm that was first introduced by Giraldeau *et al.* [14]. Throughout the tracing period a thread may be in one of several execution states. There are two possible running states: running in system call mode and running in usermode. These two running states differ in the level of privileges required for their execution, with system call mode being able to execute kernel functions. Additionally there are several blocked states: blocked for disk, blocked for futex, blocked for another task (i.e., process or thread), blocked for interrupt handling, and so forth. We determine the various execution states of each thread, and they then use these states to generate a specialized directed acyclic graph.

A visualization of an execution's critical path in the form of a graph is shown in Figure 3. This graph, which is formally referred to as an execution graph, is representative of the inter-thread interactions and dependencies. The critical path is then extracted from an execution graph by recursively substituting each blocking edge with the corresponding waking thread.

3.3 Data Vectorization

The length of different critical path sequences may vary, so we generate vector representations to create simplified and standardized representation of the various critical paths. This allows for more straightforward comparisons between executions for the purpose of anomaly detection. It also allows us to use machine learning clustering algorithms, as they typically expect fixed-sized vectors as input.

In this work we generate two types of vectors for each execution in both the normal and sample data sets. The first of these two vectors are *critical path count* vectors. With this representation, one vector represents a single execution, and each of its components represents the number of times the execution's critical path entered the corresponding state. For each execution, we also generate a *critical path duration* vector. With this format, each vector component represents the total time an execution's critical path spent in the corresponding execution state. These vector representations are similar to the system call sequence vectorization methods previously explored in [10, 19, 22]. A simplified example to illustrate the vectorization process for both representation methods is shown in Figure 4.

The critical path count vectors are representative of their executions' critical path structure, and by extension, its execution behaviour. It is reasonable to assume that when given the same

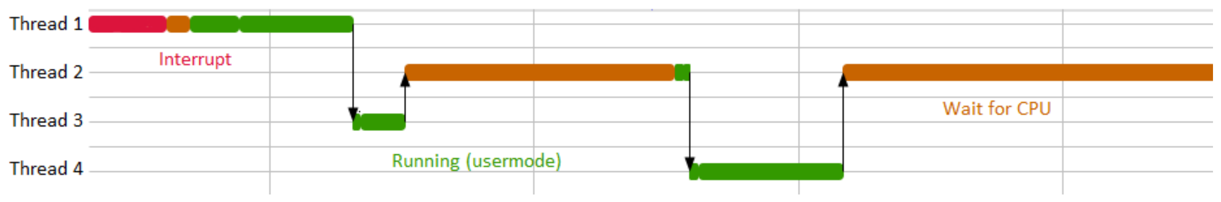
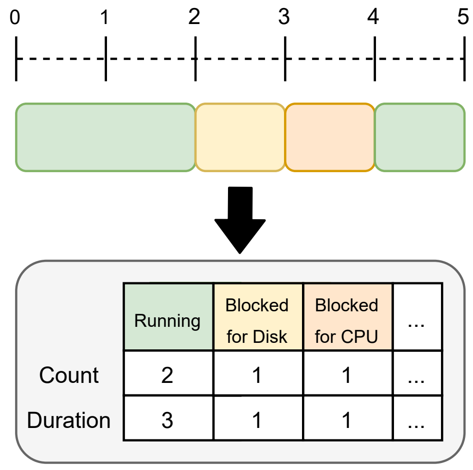


Figure 3: An execution's critical path shown as an execution graph.



	Running	Blocked for Disk	Blocked for CPU	...
Count	2	1	1	...
Duration	3	1	1	...

Figure 4: An example of how a critical path sequence is converted to critical path count vectors and critical path duration vectors.

workload, executions of the same type should have similar performance [3]. A single piece of software may exhibit a wide range of execution behaviours with varying performance expectations, so we compare the critical path count vectors to determine if executions are similar enough to justify their comparison. Clustering these vectors eliminates the need for manual analysis of each execution, which can be time consuming and requires deep knowledge of the program's internals. As such, the count vectors alone are not necessarily used to measure performance, but instead they are used to identify which executions are comparable.

While we use the critical path count vectors to show an execution's structure, we use its critical path duration vector to represent of the execution's performance. Counting events or system calls is commonly used in other works to evaluate performance, however because critical path states are a more generalized or abstracted representation of that data, the count vectors alone would not be adequate to measure performance. For instance, a long abnormal blocked state would not be distinguishable from a shorter normal blocked state using critical path counts. That's why we also have to consider the states' durations. These duration vectors are representative of the execution's performance, as the critical paths indicate what contributed to a given execution's overall runtime. User visible performance problems will result in longer state durations, which would then be reflected in the critical path duration vectors.

Algorithm 1: Anomaly Identification Algorithm

```
for all $s \in execution\_states$ do
    $a1 = normal\_data$.getAverage($s$);
    $std = normal\_data$.getStandardDev($s$);
    $a2 = sample\_data$.getAverage($s$);
    if $a2 > a1 + t1 * std$ then
        flagAnomaly($s$);
    end
    $x = |\{e | e > a1, e \in normal\_data\}|$
    $p1 = x / |normal\_data|$
    $y = |\{e | e > a2, e \in sample\_data\}|$
    $p2 = y / |sample\_data|$
    if $p2 > p1 * t2$ then
        flagAnomaly($s$);
    end
end
```

3.4 Anomaly Detection

After we have generated the vectors for our two data sets, we examine their critical path state duration vectors to determine if the sample data set contains anomalies. At this stage, we are conducting a preliminary course grained analysis to determine if a closer look at the executions is even necessary. For each critical path state *s*, we look for evidence of performance abnormalities using an average based and probability based approach, both of which are shown in Algorithm 1.

With the average based approach, we are checking if there was a noticeable increase in time spent in one or more of the execution states. For instance, if the sample executions spent more time waiting for CPU than normal, this would be detected using the average based approach. For each state we first compute the average time the executions within the normal data set spent in the state *a1*, as well as the standard deviation *std*. We also compute the average time that executions from the sample data set spent in the same state *a2*. If the sample average is greater than the normal average added to the standard deviation multiplied by a predefined value *t1*, then it is said that the sample data contains performance anomalies.

With the probability based approach, we first compute the percentage of executions in the normal and sample data sets that spent longer in a given state than the normal data set's average. These two values will be referred to as *p1* and *p2*. If *p2* exceeds *p1* multiplied by another predefined value *t2*, this would also indicate that the sample data set has anomalous executions. The average based approach checks if executions exceed a threshold for normal execution, whereas the probability based approach checks if the

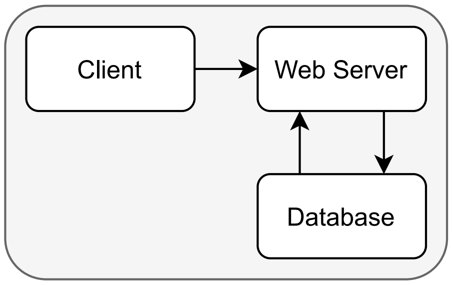


Figure 5: A web server architecture.

sample executions are consistently higher than normal while still being within this threshold, which can be another indication of performance abnormalities.

If after examining each of the execution states no abnormalities are found, then the software is said to be free of performance anomalies and the process is complete. Otherwise, the executions are examined more closely following the steps described in subsequent sections.

3.5 Anomaly Identification

In order to conduct fine grained anomaly detection, we must first determine which normal set executions are comparable to the sample executions. A single program may exhibit a wide range of critical paths with different performance expectations. For example, consider a web server with the architecture shown in Figure 5 where an execution is defined as a web request. A read request and a write request could both be without any abnormalities while still having drastically different critical paths.

There are additional factors that may result in normal variations between the two executions' critical path structure or their overall performance, including but not limited to the number of concurrent users, availability of system resources, and input size. For instance, one of these factors could be the frequency of disk accesses. To reduce this time consuming operation, many systems implement disk caching. If all the data is already found in the cache then no disk operations are required, which results in faster processing times. In the case where some of the required data is not found in the cache, the request will be blocked to recover any missing information. There may also be executions where the main thread must wait for the another thread to complete a task before it can complete its execution. All of these scenarios are from the same system, yet they result in different internal executions, hence different critical paths and processing times (as shown in Figure 6).

As some executions are expected to perform differently without being anomalous, we can not use a single definition of normal performance to identify anomalies. An alternative method that we utilize in this work is to examine the execution types separately. Grouping the executions based on their critical path similarity allows us to identify and separately analyze each execution type. This is more precise than an overgeneralized monolithic model of normal execution, and allows us to find executions that have suboptimal performance for their assigned group. It will ultimately be up to the user to ensure that the normal data properly and

Algorithm 2: Cluster Pairing Algorithm

```
for all $c1 \in sample\_clusters$ do
    $t = 0$;
    $pair$ = null;
    for all $c2 \in normal\_clusters$ do
        $temp = normal\_clusters$.clone();
        $temp$.combine($c1$, $c2$);
        $s = temp$.getSilhouetteScore();
        if $s > t$ then
            $t = s$;
            $pair = c2$;
        end
    end
    pair($c1$, $pair$);
end
```

comprehensively exhibits the system's expected behaviours, but the users would not have to concern themselves with manually sorting or grouping the data by execution type.

We find these execution groups within our normal and sample data set using machine learning clustering. We argue that the critical path count vectors are indicative of the executions' critical path structure, and thus clustering these vectors will automatically find the different execution types. Ordering Points to Identify the Clustering Structure (OPTICS) [1] is a machine learning clustering algorithms that is capable of identifying arbitrarily shaped clusters. Most well-known clustering algorithms require highly influential input parameters that are not easily determined. For instance, with the popular K-Means clustering algorithm [25] the number of clusters must be predetermined. OPTICS is presented as an alternative clustering algorithm that only requires that the minimum number of points needed to create a cluster be specified beforehand. OPTICS works by creating an ordered list representing the input data's density-based clustering structure. Samples within this list are positioned in such a way that they are next to their closest neighbour. Clusters may then be extracted from this list when observing the points' reachability distances.

Once the OPTICS clustering has identified the different execution types within the normal and sample data sets, we determine which normal clusters are the most similar to each of the sample clusters. We do this following the steps outlined in Algorithm 2. We take a sample data cluster *c1* and check the silhouette score (1) after combining it with each of the clusters *c2* from the normal data set. The silhouette score is a metric that measures the consistency of a cluster, and as such can be used to see which groups should have comparable performance metrics. The sample data cluster will be paired with the normal cluster that results in the highest silhouette score. This process is repeated for each of the clusters in the sample data set.

$$s_i = \frac{out_i - in_i}{max(in_i, out_i)} \tag{1}$$

We then use the most similar normal cluster to model normal execution for our sample group. We identify outliers using a similar method to the average based anomaly detection approach discussed

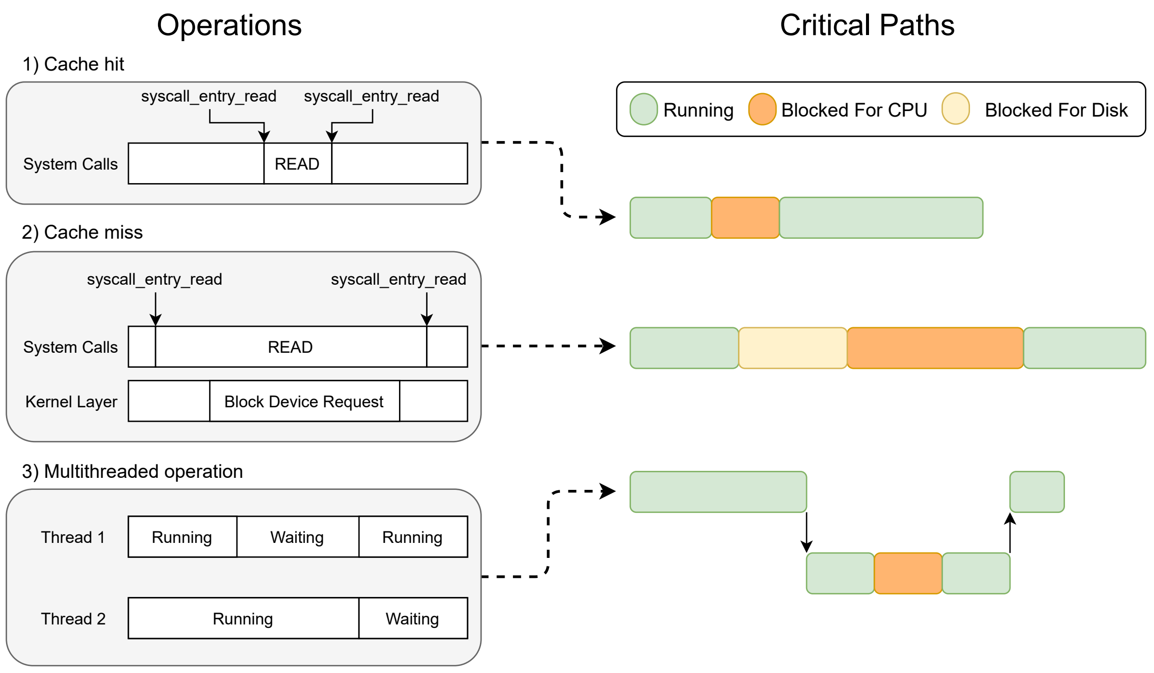


Figure 6: A visualization of the different critical paths found with different operations.

in Section 3.4. The average and standard deviation for each execution state in the normal group is calculated. Any executions within the monitored group with a state duration that exceeds the normal average plus the standard deviation multiplied by a threshold are marked as anomalies. Any anomalies will go though the fault localization process discussed in Section 3.6 to help developers identify the root cause of the performance anomaly.

3.6 Anomaly Localization

Once we have determined which executions are exhibiting anomalous behaviour, we set out to find the specific differences between the anomalies and normal executions. To do this we employ multiple sequence alignment techniques, which are commonly used in the life sciences to compare biological sequences like DNA, RNA, and proteins [11]. There exist studies that explore the use of sequence alignment for the purpose of trace comparison (e.g., [34, 35]), but these studies focus on single executions only, making it challenging to distinguish between anomalous behaviour and normal variations from the selected normal executions. We overcome this problem by proposing the use of multiple sequence alignments, which address this issue by comparing the anomalous execution to a group of normal executions.

Multiple sequence alignment is also an improvement on other trace comparison methods that simply point out the difference between two sequences. Even within a group of similar executions, it is highly unlikely to find two critical path sequences that are identical. Aligning several executions can help to find comparable regions, thereby allowing us to point out anomalous performance as well as improbable sections of the critical path sequence.

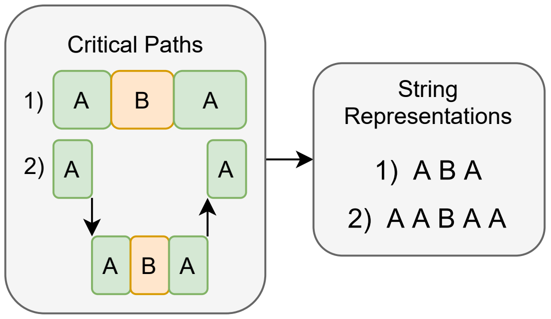


Figure 7: The conversion of different execution critical paths to their string representations.

This process starts by generating text representations of the anomalous execution and the corresponding group normal of executions from the normal data set. Each of the execution states is assigned a character, and a string made up of those characters is made for each execution's critical path. A visual representation of this process is shown in Figure 7. The amount of time the execution spent in each of the states making up the critical path sequence is also recorded.

The generated critical path sequences are then aligned so that similar regions of execution can be compared to highlight abnormalities. There are many existing algorithms for multiple sequence alignment. The one used in our proposed methodology is MUltiple Sequence Comparison by Log- Expectation (MUSCLE) [11], which is presented as an improvement in accuracy and computational complexity when compared to other well known multiple sequence

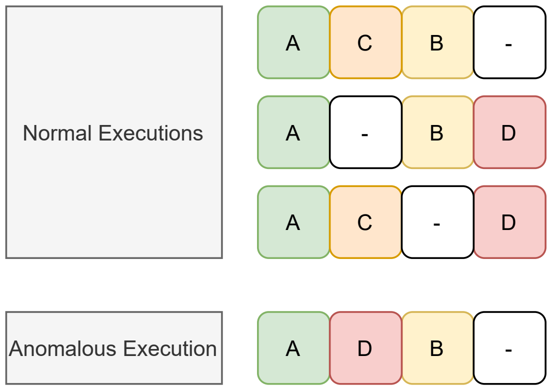


Figure 8: A visualization of the critical path sequences of four executions after undergoing multiple sequence alignment. Different states correspond to different characters, gaps are denoted by '-'.

alignment algorithms. MUSCLE builds a progressive alignment by assigning sequences to leaf nodes of a binary tree, and then aligning the children of each node. It then takes this tree and works to improve and refine the initial alignments. This results in the input sequences having gaps inserted so that similar execution states are aligned with each other.

At each point in the resulting aligned sequences, the probability that a normal execution would be in a given execution state is computed. The average time that an execution would stay in that state is also computed for each step. Abnormalities in the anomalous execution's structure as well as notable differences in how long components within the critical path sequence took can then be presented as specific anomalies.

For instance, take the simplified example of multiple sequence alignment shown in Figure 8. A developer would be shown that the anomalous execution unexpectedly entered the D state at the second point of the aligned execution sequence. Furthermore, the developer would also be notified if the anomalous execution spent significantly longer in the A or B states than what was average for the normal executions. This significantly narrows down the search space for the developer. Instead of having to examine the anomalous execution in its entirety, they could begin by examining where the abnormalities occur.

4 EVALUATION AND DISCUSSION

To test the effectiveness of our proposed approach, we examined and evaluated two separate case studies. In the first study, we examined two versions of a userspace application to ensure that our framework is capable of identifying anomalies caused by software bugs. In the second case study, we examined containerized applications to see if the approach could identify anomalies caused by an application's environment. These experiments were all conducted on a virtual machines created with Oracle VM VirtualBox 6.1.18 that were allocated two CPUs. The guest operating system used was Ubuntu 20.04.1 with Linux kernel 5.8.0, and traces were recorded using LTTng 2.11.2.

Table 1: Tracing Overhead

Benchmark	Duration (ms)	Overhead
No Tracing	1076.50	0%
Full Kernel Tracing	1329.26	+23.479%
Minimal Tracing	1128.08	+4.79145%

4.1 Userspace Application

For this case study we traced several executions of a test application written using Python 3.8.5. The application was specifically written to exhibit a range of execution types, including read operations, write operations, etc. After collecting a normal data set, a new version of the application with intentionally added software bugs was created and traced to make up a sample data set. As we chose to evaluate an application that we implemented, we were able to determine if the identified root causes of performance problems were real with a reasonable level of certainty. Throughout both tracing periods, we also recorded when the different execution types occurred so that we could be manually label the extracted critical paths.

The normal data was made up of about 100 executions, whereas the sample data set consisted of 10 executions taken from the new version of the software. The executions within the sample data set took on average 2-3 seconds longer than those within the normal data set. When clustering the executions using their critical path count vectors, we said that a minimum number of 5 points were required to make a cluster.

4.1.1 Data Collection Overhead. To examine the overhead that is imposed with execution tracing, we observed runtime performance with three distinct tracing configurations:

- No tracing: To provide a base case, an execution's performance was recorded without the use of any execution tracing.
- Full Kernel Tracing: To provide a point of comparison, we perform system tracing with all kernel tracepoints enabled. This configuration is not necessary for the proposed methodology, and is only meant to show the worst case scenario.
- Minimal Kernel Tracing: The enabled tracepoints are limited to only those required for the proposed methodology. A list of the necessary tracepoints is provided in Section 3.

The results taken from the average of 10 executions are shown in Table 1. These results show that trace data collection using full kernel tracing imposes a 23.48% overhead, however this can be significantly improved by limiting the trace events to only what is necessary for defining and extracting critical paths of execution. The configuration required for the proposed approach only imposes a 4.79% overhead. Therefore, the performance impact of our anomaly detection framework is negligible.

4.1.2 Execution Clustering. To ensure that critical path count vector clustering can be used to distinguish between different execution types, we used OPTICS clustering with the generated normal data set. The resulting clusters are shown in Figure 9. To evaluate the clustering, an execution was said to be properly assigned to a cluster

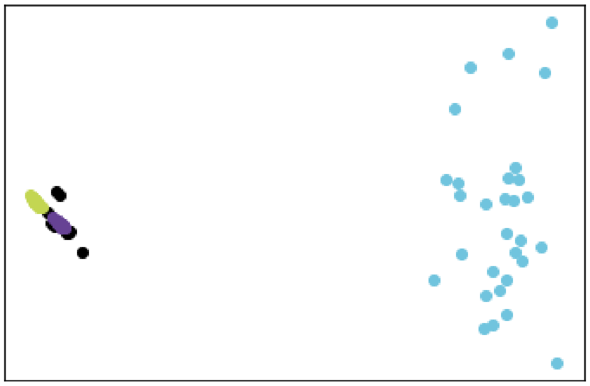


Figure 9: A visualization of the critical path count vector clustering. Vector dimensionality was reduced using PCA.

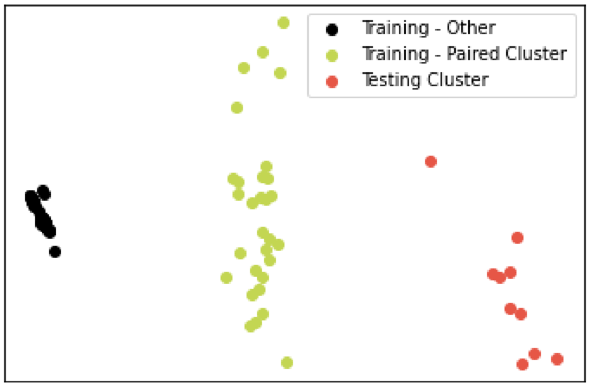


Figure 10: A visualization of the critical path count vector clustering. A sample cluster (red) and its paired normal cluster (green) is shown with the remaining vectors within the normal data set (black). Vector dimensionality was reduced using PCA.

if the majority of executions within that cluster were of the same type. Using this definition, we saw that the clustering method put 86.67% of the executions into an appropriate cluster. Accordingly, the proposed clustering method is shown to be a suitable technique for discerning between different execution types without the need for manual analysis or knowledge of the program's internal structure.

4.1.3 Cluster Pairing. To test if our cluster pairing technique would properly assign sample clusters to normal clusters made up of executions with the same execution type, we first used OPTICS clustering to identify the different groups present in the sample data set. We then used the silhouette score to pair each cluster with one of the normal clusters. We saw that despite the differences in critical path structure due to the injected faults, the highest silhouette score was achieved when a sample cluster was paired with a normal cluster of the same execution type. A visualization of the normal data with a sample data cluster is shown in Figure 10.

4.1.4 Anomaly Detection. Using the cluster pairings, we applied the outlier detection methods to determine which executions exhibited anomalous performance. When using the anomaly detection techniques we said that values greater than 1.5 standard deviations above the average were said to be anomalous. We then took those identified anomalies and tested the multiple sequence alignment method. Based on our results, we saw that the multiple sequence alignment method was capable of distinguishing between normal and anomalous states within the critical path sequences. Take for example the execution show in Figure 11. When manually reviewing the critical paths that made up the normal data set, we saw that it was typical for executions of that type to occasionally be in the blocked state for an extended period of time. Our methods did not simply mark all blocked states as anomalous. Instead, it was able to correctly identify several smaller blocked states as anomalies. This shows that our methodology goes beyond simply identifying long blocked periods. Thus multiple sequence alignment allows us to more accurately pinpoint abnormalities than we would looking at runtime alone.

4.2 Container Resource Provisioning

In recent years, container-based virtualization has been gaining popularity for both individual and industrial uses. From the kernel perspective, trace events from a process running within a container are no different from those running on the host [20]. To perform container aware tracing we just need to ensure that certain information is collected during tracing. Each container has an unique namespace. Thus by enabling the namespace ID (`pid_ns`) context, as well as the thread ID (`tid`) and virtual thread ID (`vtid`) contexts, one can identify which events belong to a specific container. By using tools like `lsns` [1], we can identify the namespaces of each container running on the host [8]. Thus, the proposed methodology is capable of finding anomalous containers and providing insight as to the root cause.

In this case study, we set up several container with Docker 20.10.7. and Ubuntu 20.04 images. Within these experiments, an execution was defined as one call of a bash script that retrieved files using GNU Wget. As containers share their host's resources, container products often allow users to specify restrictions on resources like CPU and memory. This is to help to mitigate the risk of performance impacting resource contention that is often seen with running too many containers concurrently [5]. However, it can be difficult to determine the optimal value of these parameters, and miscalculations may severely impact the container's overall performance. The normal data was collected from a container that was given no such resource constraints, and as such it was permitted to used as much of any resource as the host's kernel scheduler allowed. Trace data collected from containers with different resource constraints made up the sample data set.

4.2.1 Execution Clustering. Even when examining a simple program, we saw that clustering the normal data set executions by their critical path similarity still resulted in several distinct groups. A visualization of these groups is shown in Figure 13. This goes to show the importance of the critical path clustering. We saw that

[1]https://man7.org/linux/man-pages/man8/lsns.8.html

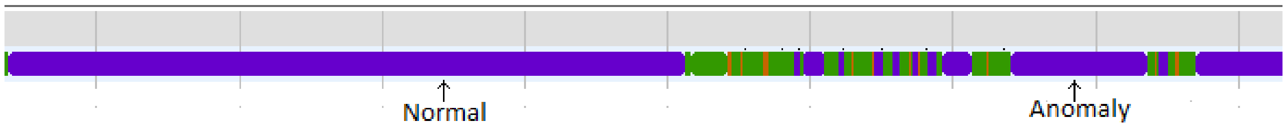


Figure 11: An anomalous execution's critical path, where the blocked state is represented by the colour purple.

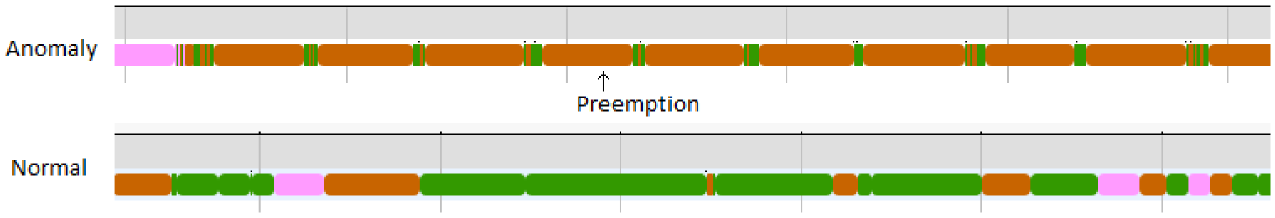


Figure 12: Two execution critical paths, one taken from an execution in the normal data set (bottom) and one of an identified anomaly (top).

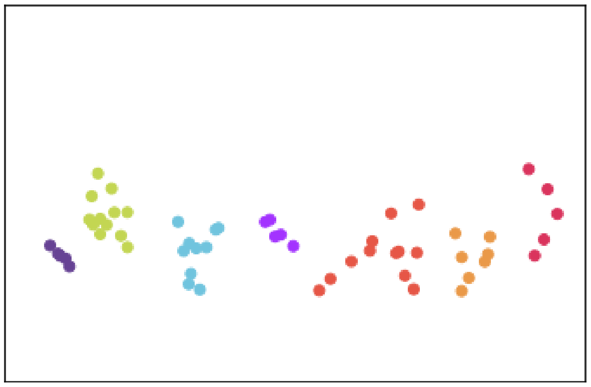


Figure 13: A visualization of the critical path count vector clustering. Vector dimensionality was reduced using PCA.

average runtime of one group could differ from another group's average runtime by up to 164.4%. With such a range of performance expectations, generating one model of normal performance would be too imprecise to effectively search for true anomalies within the sample data.

4.2.2 Anomaly Detection. Using our anomaly detection framework, we found several suboptimal executions that came from one particular container. The critical path of one of these anomalous executions compared to an execution from the normal data set is shown in Figure 12. Using the multiple sequence alignment techniques, we found several short preempted states in the anomaly's critical path that were not typical for its execution type. Further manual investigation of these states revealed that when the execution was preempted, it was not replaced on the CPU. The container from which this anomalous execution was taken from had its CPU usage limited to 10% of what was allocated to containers from which the normal data was collected. Upon discovering that the container's performance was suffering due to its configurations, we were able to grant it access to more resources to fix the problem.

5 CONCLUSIONS AND FUTURE WORK

Customer expectations and software complexity are both rapidly increasingly, and consequently so is the demand for tools to assist developers with precise performance anomaly detection. In this paper, we presented a trace based performance anomaly detection framework using critical path analysis. LTTng is first used to collect execution runtime data, which is then separated into individual executions. Each execution's critical path is extracted and used to create critical path count vectors as well as critical path duration vectors. This process is used to generate a sample and normal data set. If anomaly detection techniques indicate that anomalous executions are present in the sample data, then OPTICS clustering is used to discover the different execution types without the need for manual intervention. Sample data clusters are compared to their most similar normal data clusters, allowing for a more precise analysis. Any identified anomalies undergo multiple sequence alignment to find specific anomalies within their critical path. To demonstrate the usefulness of this framework, we discussed two experimental use cases using userspace and containerized applications.

As for future work, we plan to expand upon our current method to automatically pinpoint the root cause of any identified anomalies. To do this plan on experimenting with call stack tracing. Additionally, we plan to investigate the use of more advanced deep learning techniques for performance anomaly detection.

REFERENCES

[1] Ankerst, M., Breunig, M. M., Kriegel, H.-P., and Sander, J. OPTICS: Ordering points to identify the clustering structure. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 1999), SIGMOD '99, Association for Computing Machinery, p. 49–60.

[2] Aslanpour, M. S., Gill, S. S., and Toosi, A. N. Performance evaluation metrics for cloud, fog and edge computing: A review, taxonomy, benchmarks and standards for future research. *Internet of Things 12* (2020), 100273.

[3] Ates, E., Sturmann, L., Toslali, M., Krieger, O., Megginson, R., Coskun, A. K., and Sambasivan, R. R. An automated, cross-layer instrumentation framework for diagnosing performance problems in distributed applications. In *Proceedings*

of the ACM Symposium on Cloud Computing (New York, NY, USA, 2019), SoCC '19, Association for Computing Machinery, p. 165–170.

[4] Biancheri, C., Ezzati-Jivan, N., and Dagenais, M. R. Multilayer virtualized systems analysis with kernel tracing. In *2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)* (2016), pp. 1–6.

[5] Cai, L., Qi, Y., Wei, W., and Li, J. Improving resource usages of containers through auto-tuning container resource parameters. *IEEE Access PP* (07 2019), 1–1.

[6] Cornelissen, B., Zaidman, A., Deursen, A., Moonen, L., and Koschke, R. A systematic survey of program comprehension through dynamic analysis. *Software Engineering, IEEE Transactions on 35* (11 2009), 684 – 702.

[7] Creech, G., and Hu, J. A semantic approach to host-based intrusion detection systems using contiguousand discontiguous system call patterns. *IEEE Transactions on Computers 63*, 4 (2014), 807–819.

[8] DENYS, P.-F., Dagenais, M. R., and Pepin, M. Advanced tracing methods for container messaging systems analysis.

[9] Desnoyers, M., and Dagenais, M. LTTng: Tracing across execution layers, from the hypervisor to user-space. In *Linux Symposium* (2008), p. 101.

[10] Dymshits, M., Myara, B., and Tolpin, D. Process monitoring on sequences of system call count vectors. pp. 1–5.

[11] Edgar, R. MUSCLE: A multiple sequence alignment method with reduced time and space complexity. *BMC bioinformatics 5* (09 2004), 113.

[12] Ezzati-Jivan, N., Fournier, Q., Dagenais, M. R., and Hamou-Lhadj, A. DepGraph: Localizing performance bottlenecks in multi-core applications using waiting dependency graphs and software tracing. In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)* (2020), pp. 149–159.

[13] García-Teodoro, P., Díaz-Verdejo, J., Maciá-Fernández, G., and Vázquez, E. Anomaly-based network intrusion detection: Techniques, systems and challenges. *Computers Security 28*, 1 (2009), 18–28.

[14] Giraldeau, F., and Dagenais, M. Wait analysis of distributed systems using kernel tracing. *IEEE Transactions on Parallel and Distributed Systems 27*, 8 (2016), 2450–2461.

[15] Giraldeau, F., Desfossez, J., Goulet, D., Desnoyers, M., and Dagenais, M. R. Recovering system metrics from kernel trace. In *Linux Symposium 2011* (June 2011).

[16] Grasso, C. S., and Lee, C. J. Combining partial order alignment and progressive multiple sequence alignment increases alignment speed and scalability to very large alignment problems. *Bioinformatics 20 10* (2004), 1546–56.

[17] Hendriks, M., Verriet, J., Basten, T., Theelen, B., Brassé, M., and Somers, L. Analyzing execution traces: critical-path analysis and distance analysis. *International Journal on Software Tools for Technology Transfer 19* (08 2017).

[18] Islam, M. S., Khreich, W., and Hamou-Lhadj, A. Anomaly detection techniques based on kappa-pruned ensembles. *IEEE Transactions on Reliability 67*, 1 (2018), 212–229.

[19] Jana, I., and Oprea, A. AppMine: Behavioral analytics for web application vulnerability detection. pp. 69–80.

[20] Janecek, M., Ezzati-Jivan, N., and Azhari, S. V. Container workload characterization through host system tracing. In *2021 IEEE International Conference on Cloud Engineering (IC2E)* (2021), pp. 9–19.

[21] Khreich, W., Khosravifar, B., Hamou-Lhadj, A., and Talhi, C. An anomaly detection system based on variable n-gram features and one-class svm. *Information and Software Technology 91* (2017), 186–197.

[22] Kohyarnejadfard, I., Aloise, D., Dagenais, M., and Shakeri, M. A framework for detecting system performance anomalies using tracing data analysis. *Entropy 23* (08 2021).

[23] Kohyarnejadfard, I., Shakeri, M., and Aloise, D. System performance anomaly detection using tracing data analysis. In *Proceedings of the 2019 5th International Conference on Computer and Technology Applications* (New York, NY, USA, 2019), ICCTA 2019, Association for Computing Machinery, p. 169–173.

[24] Liu, M., Xue, Z., Xu, X., Zhong, C., and Chen, J. Host-based intrusion detection system with system calls: Review and future trends.

[25] McQueen, J. Some methods for classification and analysis of multivariate observations. *Computer and Chemistry 4* (01 1967), 257–272.

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

[27] Notredame, C., Higgins, D., and Heringa, J. Notredame, c., higgins, d. g. heringa, j. t-coffee: A novel method for fast and accurate multiple sequence alignment. j. mol. biol. 302, 205-217. *Journal of Molecular Biology 302* (10 2000), 205–217.

[28] Patcha, A., and Park, J.-M. An overview of anomaly detection techniques: Existing solutions and latest technological trends. *Computer Networks 51*, 12 (2007), 3448–3470.

[29] Pirzadeh, H., Shanian, S., Hamou-Lhadj, Abdelwahab, A. L., and Sharifee, A. An anomaly detection system based on variable n-gram features and one-class svm. *Science of Computer Programming 78*, 8.

[30] Rhee, J., Zhang, H., Arora, N., Jiang, G., and Yoshihira, K. Software system performance debugging with kernel events feature guidance. In *2014 IEEE Network Operations and Management Symposium (NOMS)* (2014), pp. 1–5.

[31] Sambasivan, R., Zheng, A., Rosa, M., Krevat, E., Whitman, S., Stroucken, M., Wang, W., Xu, L., and Ganger, G. Diagnosing performance changes by comparing request flows.

[32] Schulz, M. Extracting critical path graphs from MPI applications. In *2005 IEEE International Conference on Cluster Computing* (2005), pp. 1–10.

[33] Thompson, J., Higgins, D., and Gibson, T. Clustal w: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic acids research 22*, 22 (November 1994), 4673—4680.

[34] Weber, M., Brendel, R., and Brunst, H. Trace file comparison with a hierarchical sequence alignment algorithm. In *2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications* (2012), pp. 247–254.

[35] Weber, M., Mohror, K., Schulz, M., Supinski, B., Brunst, H., and Nagel, W. Alignment-based metrics for trace comparison. vol. 8097, pp. 29–40.

[36] Zhang, X., Niyaz, Q., Jahan, F., and Sun, W. Early detection of host-based intrusions in Linux environment. In *2020 IEEE International Conference on Electro Information Technology (EIT)* (2020), pp. 475–479.