ABSTRACT

LIN, YUHANG. Self-Supervised Distributed Machine Learning for Robust Containerized Systems. (Under the direction of Xiaohui Gu).

Containers are widely embraced in production computing environments due to their efficiency and minimal isolation overhead. However, these applications are vulnerable to both security attacks and software bugs. While machine learning techniques have been explored to detect such issues, they are not without limitations. A critical concern for developers lies in identifying the specific root cause functions responsible for attacks or bugs. As cloud systems, including intricate microservice setups, gain traction, debugging performance bugs becomes even more intricate. Existing solutions often fall short in capturing robust causal relationships within monitored service data, resulting in inaccurate root cause prioritization. Additionally, these solutions frequently fail to pinpoint the precise function-level origin, leaving developers with the challenging task of pinpointing the responsible function. This dissertation addresses the challenges encompassing attack detection, root cause analysis at the service and function levels, with the ultimate aim of establishing resilient containerized systems.

Firstly, we address security attack detection for short-lived dynamic containers using CDL, a classified distributed learning framework. CDL integrates online application classification and anomaly detection to tackle the challenge of limited training data for dynamic containers across diverse applications. Evaluation on 33 real-world vulnerability attacks demonstrates CDL significantly reduces false positives, improving detection rates.

Secondly, we introduce SHIL, a self-supervised hybrid learning solution for reducing false alarms without manual labeling. SHIL combines unsupervised and supervised methods, achieving higher accuracy while cutting false alarms by 39-91% across 41 security attacks in various applications.

Lastly, we delve into function-level root cause analysis for containerized systems. FCA, an information theory-based approach, identifies causal relationships among system metrics and context spans to pinpoint root cause services and functions. FCA’s effectiveness is confirmed through an evaluation over ten performance bugs across three benchmark microservice applications.
Self-Supervised Distributed Machine Learning for Robust Containerized Systems

by
Yuhang Lin

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina
2023

APPROVED BY:

Min Chi  Guoliang Jin

Huiyang Zhou  Xiaohui Gu
Chair of Advisory Committee
Yuhang Lin received his Bachelor’s degree from Zhejiang Normal University in 2016 and his Master’s degree from the University of Vermont in 2018. At North Carolina State University, he worked under the direction of Dr. Xiaohui (Helen) Gu. His research focuses on automatically detecting and diagnosing security and performance bugs in container systems. He interned at Meta (formerly named Facebook) in the summers of 2021 and 2022.
ACKNOWLEDGEMENTS

I am truly grateful to many individuals whose unwavering support and encouragement have been instrumental in the completion of this dissertation. I would like to take a moment to express my heartfelt appreciation to each of them.

First and foremost, I extend my profound gratitude to my advisor, Dr. Xiaohui (Helen) Gu. Her profound knowledge and guidance have enriched my academic journey, and it has been a distinct honor to collaborate with her during my tenure at this university. The memories of our time working together will forever hold a special place in my heart.

I am also deeply indebted to my esteemed committee members: Dr. Guoliang Jin, Dr. Min Chi, and Dr. Huiyang Zhou. Their invaluable insights and suggestions have significantly contributed to the refinement of my dissertation.

Next, I wish to express my heartfelt thanks to several current and former labmates: Dr. Jingzhu He and Fogo Tunde-Onadele. Their unwavering friendship and support have been both precious and indispensable.

Furthermore, I would like to extend my gratitude to my dear friends at our university, namely Yao Meng, Ming Liang, Junkai Liang, and Muh-Jang Chen. Their enduring friendship and encouragement have been a constant source of inspiration.

I am also grateful for the generous funding provided by the National Security Agency and Cisco, which made my research endeavors possible.

Lastly, but certainly not least, I want to express my deepest appreciation to my parents, family members, and friends for their boundless love and unwavering support throughout my academic journey. Your presence in my life has been a source of strength and motivation.
TABLE OF CONTENTS

List of Tables ........................................................................................................... vi
List of Figures ........................................................................................................... vii

Chapter 1 INTRODUCTION ......................................................................................... 1
  1.1 Motivations ........................................................................................................ 1
  1.2 Summary of the State of the Art ....................................................................... 2
  1.3 Summary of Contributions ................................................................................ 2

Chapter 2 CDL: CLASSIFIED DISTRIBUTED LEARNING FOR DETECTING SECURITY
  ATTACKS IN CONTAINERIZED APPLICATIONS ................................................... 4
  2.1 Introduction ....................................................................................................... 4
  2.2 System Design .................................................................................................. 6
    2.2.1 System Overview ....................................................................................... 6
    2.2.2 System Call Feature Extraction ............................................................... 7
    2.2.3 Application Classification ....................................................................... 9
    2.2.4 System Call Data Grouping .................................................................. 10
    2.2.5 Classified Learning and Anomaly Detection .......................................... 11
  2.3 Experimental Evaluation ................................................................................ 12
    2.3.1 Evaluation Methodology ....................................................................... 12
    2.3.2 Results Analysis ..................................................................................... 17
    2.3.3 Case Study ............................................................................................... 22
  2.4 Summary .......................................................................................................... 24

Chapter 3 SHIL: SELF-SUPERVISED HYBRID LEARNING FOR SECURITY ATTACK
  DETECTION IN CONTAINERIZED APPLICATIONS ............................................. 26
  3.1 Introduction ..................................................................................................... 26
  3.2 System Design ................................................................................................ 28
    3.2.1 Unsupervised Anomaly Detection ......................................................... 30
    3.2.2 Hybrid Alert Validation ....................................................................... 31
    3.2.3 Self-supervised Model Creation ........................................................... 33
  3.3 Experimental Evaluation ................................................................................ 36
    3.3.1 Evaluation Methodology ...................................................................... 36
    3.3.2 Results Analysis .................................................................................... 40
    3.3.3 Case Study ............................................................................................. 42
  3.4 Summary .......................................................................................................... 43

Chapter 4 FCA: FULL-STACK CAUSAL ANALYSIS FOR MICROSERVICE PERFOR-
  MANCE DEBUGGING ......................................................................................... 44
  4.1 Introduction ..................................................................................................... 44
    4.1.1 Motivating Example .............................................................................. 46
    4.1.2 Contribution ........................................................................................... 46
  4.2 System Design ................................................................................................ 47
LIST OF TABLES

Table 2.1 A frequency vector sample for the Elasticsearch application (CVE-2015-1427). An attack is triggered at $t = 1586738324176$. CDL raises alarms from $t = 1586738324476$. .......................................................... 8

Table 2.2 List of explored real-world vulnerabilities. ........................................... 13

Table 2.3 Classification results among different application classifiers. ............. 17

Table 2.4 Detection results of all CVE examined among different models. .......... 18

Table 2.5 Summary of detection results among different models. ..................... 18

Table 2.6 Detection results among different models with 99 percentile of training reconstruction error. .............................................................. 18

Table 2.7 CDL system run time measurements. .................................................. 22

Table 3.1 A frequency vector sample for the OpenSSH application (CVE-2016-6515). An attack is triggered at $t = 1586496672891$. Anomaly detection raises alarms from $t = 1586496673091$ to $t = 1586496673191$. The entries with asterisks (*) are detected outliers. ........................................... 31

Table 3.2 List of explored real-world vulnerabilities. ........................................... 37

Table 3.3 Comparison with alternative approaches. ........................................... 39

Table 3.4 System run time measurements of different learning methods. Each sample represents the frequency vector of all system calls produced within 100 ms. .............................................................. 42

Table 4.1 Summary of bugs that we use in our experiment. .............................. 59

Table 4.2 Summary of root cause service results. .............................................. 59

Table 4.3 Summary of root cause function results. “X” means failure to detect the root cause function. .............................................................. 59

Table 4.4 Overhead measurements of FCA analysis components, given a sampling interval of 30s. .............................................................. 63
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>System overview of CDL.</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>An example of random forest operation.</td>
<td>9</td>
</tr>
<tr>
<td>2.3</td>
<td>System call data grouping in the CDL system.</td>
<td>10</td>
</tr>
<tr>
<td>2.4</td>
<td>Classified learning and anomaly detection of CDL.</td>
<td>11</td>
</tr>
<tr>
<td>2.5</td>
<td>Architecture of the autoencoder.</td>
<td>11</td>
</tr>
<tr>
<td>2.6</td>
<td>Classification results among different application classifiers.</td>
<td>15</td>
</tr>
<tr>
<td>2.7</td>
<td>Detection results among different models with 99.9 percentile of training reconstruction error.</td>
<td>19</td>
</tr>
<tr>
<td>2.8</td>
<td>Detection results among different models with 99 percentile of training reconstruction error.</td>
<td>20</td>
</tr>
<tr>
<td>2.9</td>
<td>Comparisons of lead time among different models.</td>
<td>21</td>
</tr>
<tr>
<td>3.1</td>
<td>System overview of SHIL.</td>
<td>29</td>
</tr>
<tr>
<td>3.2</td>
<td>Self-supervised model creation.</td>
<td>32</td>
</tr>
<tr>
<td>3.3</td>
<td>An example of outlier detection with the isolation forest model. The leaf nodes with the shortest heights from their root nodes tend to be outliers.</td>
<td>33</td>
</tr>
<tr>
<td>3.4</td>
<td>Model comparisons among unsupervised autoencoder using 95% and 99% thresholds, supervised RF, supervised CNN, combined, and SHIL.</td>
<td>40</td>
</tr>
<tr>
<td>4.1</td>
<td>Due to data corruption, an issue arises within the I/O function. When <code>istream.ignore</code> returns 0 and 0 is used as the stride, this sequence of events triggers a situation where the ReviewStorage service becomes unresponsive and begins to hang.</td>
<td>45</td>
</tr>
<tr>
<td>4.2</td>
<td>The system overview of FCA.</td>
<td>47</td>
</tr>
<tr>
<td>4.3</td>
<td>We consider the span end time to determine the average span duration for each window.</td>
<td>49</td>
</tr>
<tr>
<td>4.4</td>
<td>The causality score improves with anomaly detection preprocessing as observed during an infinite loop bug example in the Online Boutique application.</td>
<td>51</td>
</tr>
<tr>
<td>4.5</td>
<td>An example of the anomaly alignment algorithm. We require a minimum number of consecutive anomalous samples to avoid misalignment due to noise.</td>
<td>51</td>
</tr>
<tr>
<td>4.6</td>
<td>An example of service dependency graph extracted from Jaeger. The direction of each edge is from the caller to the callee, with the number of call count on the edge.</td>
<td>52</td>
</tr>
<tr>
<td>4.7</td>
<td>A comparison of the root cause service CPU metric data with the root cause function execution time for the infinite loop bug injected into the Online Boutique application.</td>
<td>53</td>
</tr>
<tr>
<td>4.8</td>
<td>Correlation values of the top candidate services using different algorithms. Every bar with a cross (X) on its top is the root cause service.</td>
<td>56</td>
</tr>
<tr>
<td>4.9</td>
<td>Correlation values of the top candidate functions using different algorithms. Every bar with a cross (X) on its top is the root cause function.</td>
<td>60</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

1.1 Motivations

Container-based distributed system platforms have gained tremendous popularity in many real world applications because of their cost-effectiveness and accessibility. However, it is challenging to provide a robust environment for containerized systems.

On one hand, containers open up new attack surfaces to malicious attackers. Recent studies (Sulinski (2017); Shu et al. (2017); Kwon and Lee (2020)) have shown that containers are vulnerable to various security attacks and attackers are consistently developing new attacks. Therefore we should develop an automatic online system that can detect unseen security attacks on containers while achieving a good balance between a high detection rate and a low false positive rate.

On the other hand, as containerized applications develop and become increasingly complex, debugging distributed performance bugs becomes a key challenge (Balalaie et al. (2016); Jamshidi et al. (2018)). The increased complexity makes it more difficult to understand the dependencies among different system components and pinpoint root cause of the bug. By knowing the root cause function, software developers can directly address that function and fix the bug in much less time. Thus we need to provide an automatic system to analyze the root cause function of the performance bugs in containerized applications.
1.2 Summary of the State of the Art

Security attack detection can be achieved by either a signature-driven approach Li (2004); Liao et al. (2013) or an anomaly detection approach Forrest et al. (1996); Geng and Jia (2009). Previous work Anthi et al. (2019); Lin et al. (2020); Tunde-Onadele et al. (2020); Idhammad et al. (2018) has proposed to apply different machine learning methods including supervised learning, unsupervised learning, and semi-supervised learning, to achieve effective security attack detection.

Existing microservice approaches often localize the root cause at the service level Wu et al. (2020); Ma et al. (2020); Lin et al. (2018a); Meng et al. (2020), which leaves more debugging effort to the developer. Moreover, they generally perform correlation analysis methods (e.g. Pearson or partial correlation), which only identify linear dependence relationships in monitored data. Other microservice-based work that address non-linear dependence with Granger causality Wang et al. (2021) use neural network models that require more overhead such as larger training data costs. It is valuable to capture general dependencies to accurately discover the root cause.

1.3 Summary of Contributions

In this dissertation, we make the following contributions:

• We have presented CDL, a new classified distributed learning framework that aims at achieving practical and efficient security attack detection for containerized applications. CDL integrates online application classification and application-specific anomaly detection models to overcome the challenges of lacking sufficient training data for individual short-lived containers. We have implemented a prototype of CDL and conducted experiments over 33 real world vulnerability exploits in 24 commonly used applications. Our results show that CDL can reduce false positive rates from over 12% to 0.24% compared to traditional learning methods without aggregating training data from different containers and increase the true positive rate from 45% to 74% compared to simple training data aggregation without performing application classifications. CDL supports real time security attack detection, which makes it practical for production computing environments.

• We have presented SHIL, a new self-supervised hybrid learning system for more efficiently detecting security attacks in container-based computing environments. SHIL identifies anomaly detection boundary cases as most likely false alarms and combines unsupervised and supervised machine learning methods to filter out majority of the false
alarms without missing most of the true attacks. For practical deployment of supervised learning models, SHIL adopts a self-supervised learning approach to labelling training data automatically using outlier detection over a window of recent measurement samples when attack alerts are first raised. Our experimental results with real world security attacks including the recent high-impacting Log4j attack show that SHIL can significantly reduce false alarms by up to 91% while maintaining similar detection rates compared to existing pure-supervised or pure-unsupervised methods. SHIL is light-weight and does not require manual data labelling, which makes it practical for security attack detection in container-based production environments.

• We have presented FCA, a new causal analysis framework for distributed performance bugs in microservice architectures. FCA finds the root cause of the service and function by analyzing system metrics and spans, which carry information about request timing, service dependencies, and function execution. It quantifies causal relationships with information theory measures, time-series alignment techniques, and service dependency graphs. We have developed a prototype of FCA and evaluate it over ten performance bugs in three benchmark microservice applications. The results show that FCA successfully discovers the root cause in 100% of bugs.

This dissertation is organized as follows. Chapter 2 describes a new classified distributed learning framework for security attack detection in containerized applications. Chapter 3 presents a self-supervised hybrid learning system for more efficient security attack detection in container-based environments. Chapter 4 presents a causal analysis framework for identifying root causes of performance bugs in microservice architectures. Chapter 5 discusses the related work. Finally, Chapter 6 concludes this dissertation.
CHAPTER 2

CDL: CLASSIFIED DISTRIBUTED LEARNING FOR DETECTING SECURITY ATTACKS IN CONTAINERIZED APPLICATIONS

2.1 Introduction

Container technology is widely adopted in today’s distributed computing environments for its efficiency and low overhead of isolation. However, recent studies Sulinski (2017); Shu et al. (2017) have shown that containers are prone to various security attacks, which has become one of the top concerns for users to fully adopt container technology Bettini (2015). Indeed, previous study Shu et al. (2017) reveals an alarming degree of vulnerability exposure and spread in the official Docker Hub container repository.

Security attack detection can be achieved by either a signature-driven approach Li (2004); Liao et al. (2013) or an anomaly detection approach Forrest et al. (1996); Geng and Jia (2009). In this work, we focus on studying the latter approach for dynamically detecting both known and unknown attacks. Container-based distributed systems bring both new challenges and opportunities for security attack detection. On one hand, containerized applications are often ephemeral, which typically run for a short period of time before the container is stopped for sav-
ing resources because restarting a container is typically fast and incurs low cost Carter (2018); Datadog (2018). As a result, it is challenging for the anomaly detection system to collect sufficient training data to build a reliable normal behavior model. On the other hand, containerized applications are highly replicated both spatially and temporally. The user often spawns a large number of containers from the same container image to achieve concurrent processing of a large workload. The same container might be restarted at different time to process periodically repeating requests. Thus, the inherent redundancy of the container environment presents new opportunities for the anomaly detection model to leverage the power of distributed learning which can aggregate related training data from a large number of distributed containers to create a robust normal behavior model.

In this chapter, we present a new classified distributed learning (CDL) framework for achieving efficient security attack detection in containerized applications. CDL implements a distributed learning framework to overcome the challenge of insufficient training data in ephemeral container environments. Different from the traditional learning scheme which derives an independent normal behavior model for each container from a limited set of training data obtained during its short lifetime, CDL aims at building robust normal behavior models by performing distributed learning over aggregated training data from a group of distributed containers. However, different applications can have very distinct normal behaviors. So it is insufficient to simply aggregate all training data from all containers without considering the specific applications running inside different containers. To address the challenge, we propose to incorporate application-based classification into the distributed learning framework to create more precise normal behavior model for different applications.

CDL adopts a system call driven anomaly detection approach using out-of-box container monitoring tools Sysdig (2016) to achieve non-intrusive, low-cost security attack detection. Specifically, we continuously collect system call traces produced by each container and extract feature vectors such as the invocation frequencies of different system call types within each sampling interval (e.g., 0.1 second). We then feed the feature vectors into anomaly detection models to detect different attacks. In this work, we choose autoencoder neural network Thompson et al. (2002) as our anomaly detection model because of its computation efficiency and high accuracy.

Specifically, CDL consists of three integrated components: 1) application classifiers which categorize different system call vectors into their corresponding application groups for creating precise normal behavior model for each application; 2) data assemblers which collect system call feature vectors from different containers and group system call feature vectors based on the application classification results (e.g., Apache Tomcat versus OpenSSL), and 3) classified learning which build normal behavior models for different applications and perform attack
detection using application-specific models. This chapter makes the following contributions:

- We propose a new classified distributed learning framework to achieve efficient security attack detection for containerized applications.

- We present efficient application classification and anomaly detection schemes using light-weight, black-box online learning methods.

- We have implemented a prototype of CDL and evaluated it over 33 recent real critical vulnerabilities with high CVSS scores in 24 commonly used server applications.

Our results show that CDL can successfully detect 31 out of 33 attacks with a low false positive rate (0.24% on average). In contrast, the traditional learning scheme without aggregating any training data incurs orders of magnitude higher false positive rate (12.74% on average) because of lacking sufficient training data. We also compare CDL with conventional distributed learning methods which aggregate all training data without considering behavior variations among different applications. The resulting model can only detect 20 out of 33 tested attacks due to unsorted training data. Moreover, CDL can detect two critical attack types (i.e., return a shell and execute arbitrary code, consume excessive CPU) 15-18 seconds before attacks succeed. CDL is lightweight and efficient, which can finish online anomaly detection for each extracted system call feature vector sampled at 100 millisecond intervals within a few milliseconds.

The rest of the chapter is structured as follows. Section 2.2 presents CDL design in detail. Section 2.3 describes our experimental evaluation methodology and our experimental results. Section 2.4 concludes this chapter.

## 2.2 System Design

In this section, we present the design of the CDL system. We first provide an overview about the CDL system. We then describe each CDL component in detail.

### 2.2.1 System Overview

CDL implements a classified distributed learning framework shown by Figure 2.1. CDL leverages an open-source container monitoring tool called Sysdig (Sysdig, 2016) to collect system call traces from the outside of the containers. We leverage system calls for their ability to reveal security attacks with low cost. CDL performs continuous analysis over collected system call traces to achieve online attack detection, which consists of four major steps: 1) system call feature extraction, 2) application classification, 3) system call data grouping, and 4) classified
Figure 2.1: System overview of CDL.

Specifically, the system call feature extraction component performs continuous system call data pre-processing to extract useful features for attack detection. For example, we compute the frequency count of each system call type within a certain sampling interval to categorize the interactions between the application and the kernel. Next, the application classifier identifies groupings of containerized applications and versions (e.g., ActiveMQ v1.0, Bash v3.0) based on their system call composition and frequency. We currently leverage the random forest model to perform application composition classification. Such an ensemble model offers good accuracy by minimizing the over-fitting errors Ho (1995). Next, the system call grouping module assembles the feature data from different containers into groups based on their application tags. Finally, the grouped data are fed into an unsupervised neural network to perform classified learning or detection. During classified training, we create separate models for each application class. Those models are then used to perform anomaly detection over the application containers within their respective categories.

2.2.2 System Call Feature Extraction

To monitor containerized applications, we trace the system calls they produce using the open source tool Sysdig Sysdig (2016). The tracing tool accesses the host kernel to provide helpful information about operating system (OS) level events such as file read/write and synchronization operations. Sysdig is also able to filter the events by containers. Thus, the result is a
Table 2.1: A frequency vector sample for the *Elasticsearch* application (CVE-2015-1427). An attack is triggered at $t = 1586738324176$. CDL raises alarms from $t = 1586738324476$.

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>System call</th>
<th>futex</th>
<th>lseek</th>
<th>read</th>
<th>stat</th>
</tr>
</thead>
<tbody>
<tr>
<td>1586738323776</td>
<td></td>
<td>8</td>
<td>0</td>
<td>361</td>
<td>0</td>
</tr>
<tr>
<td>1586738323876</td>
<td></td>
<td>6</td>
<td>0</td>
<td>349</td>
<td>0</td>
</tr>
<tr>
<td>1586738323976</td>
<td></td>
<td>8</td>
<td>0</td>
<td>344</td>
<td>0</td>
</tr>
<tr>
<td>1586738324076</td>
<td></td>
<td>6</td>
<td>0</td>
<td>309</td>
<td>0</td>
</tr>
<tr>
<td>1586738324176 (attack starts)</td>
<td></td>
<td>12</td>
<td>0</td>
<td>297</td>
<td>0</td>
</tr>
<tr>
<td>1586738324276</td>
<td></td>
<td>6</td>
<td>0</td>
<td>344</td>
<td>0</td>
</tr>
<tr>
<td>1586738324376</td>
<td></td>
<td>10</td>
<td>0</td>
<td>383</td>
<td>0</td>
</tr>
<tr>
<td>1586738324476 (attack detected by CDL)</td>
<td></td>
<td>8</td>
<td>0</td>
<td>451</td>
<td>0</td>
</tr>
<tr>
<td>1586738324576</td>
<td></td>
<td>8</td>
<td>6</td>
<td>375</td>
<td>3</td>
</tr>
<tr>
<td>1586738324676</td>
<td></td>
<td>370</td>
<td>64</td>
<td>434</td>
<td>32</td>
</tr>
<tr>
<td>1586738324776</td>
<td></td>
<td>118</td>
<td>193</td>
<td>625</td>
<td>94</td>
</tr>
<tr>
<td>1586738325876 (attack completes)</td>
<td></td>
<td>24</td>
<td>76</td>
<td>378</td>
<td>35</td>
</tr>
</tbody>
</table>

detailed capture of the system calls made by a specified container.

To achieve anomaly detection, we process the raw system call trace into a stream of frequency vectors, that is, for each sampling point, we calculate the occurrences of each system call type and record it as a frequency vector feature for detecting security attacks. To handle the diversity of system call types produced by different applications, we expand the frequency vector to the same dimension by including all existing Linux system call types. Table 2.1 shows an example of a partial frequency vector for an attack exploiting the vulnerability CVE-2015-1427 in the Elasticsearch application. Each timestamp $t$ is associated with a vector $V = [f_1, f_2, f_3, ..., f_k]$ where $f_i (i \in [1, k])$ is the occurrence count for system call type $s_i$ during the last sampling period (100 milliseconds) starting at time $t$. For example, during the time period $[t, t + 100)$ milliseconds with $t = 1586738323776$ (shown by the first row in Table 2.1), the *futex* system call occurs 8 times, whereas *read* is called 361 times. The attack starts at $t = 1586738324176$. We highlight those system calls with abnormal frequency changes after the attack starts. We notice significant frequency increases in *futex*, *lseek*, *read* and *stat* system calls. CDL starts to raise an alarm at $t = 1586738324476$ during an initial increase in the number of *read* calls. In the next sample, the containerized application starts to invoke the *stat* call. However, by time $t = 1586738324676$, the frequency of the *futex* call is over an order of magnitude higher than that before the attack starts.

It is also noteworthy that our extracted frequency vector trace is orders of magnitude smaller than the original system call trace in data size. For example, in our experiments, the average raw system call trace has an average size of 1.2 GiB while the extracted feature vector trace is only 4.8 MiB on average. During the following application classification and classified training/learning.
steps, we only need to process feature vectors without transmitting large raw system call traces over networks.

2.2.3 Application Classification

In order to create precise normal behavior models for different applications, it is important to distinguish different applications. However, we need to tackle a set of challenges to achieve the goal in production container environments. First, we cannot rely on human inputs to manually label each container since containers can be highly dynamic and a production system often consists of tens of thousands of containers. Second, we have to avoid intrusive monitoring tools which can bring large overhead to light-weight containers. Third, our application classification schemes need to be workload insensitive since the containers of the same application might process different workloads at different time. To this end, we leverage the random forest learning scheme Ho (1995) to achieve the goal shown by Figure 2.2. During our experiments, we observe that system call feature vectors extracted by our system call preprocessing scheme provide sufficient and workload insensitive patterns for us to distinguish different applications.

The random forest classifier uses a number of decision trees to improve accuracy while combating over-fitting. Each decision tree receives only a portion of the training data. Each
decision tree makes local optimal splitting decisions among a random subset of the features when splitting nodes. As a result, the trained decision trees are usually quite different from one another. The output of the random forest classifier then uses the majority voting result among individual decision trees. For example, in Figure 2.2, the random forest model consists of three decision trees. Given a system call frequency vector, the first decision tree classifies the input data is from the ActiveMQ application; the second and the third decision trees classify the input data is from the Bash application. The random forest model will output Bash as the final application classification result.

### 2.2.4 System Call Data Grouping

The system call data grouping component handles the assembly of data from multiple containers into different application classes assigned by the application classifier. System call data grouping operates in conjunction with the application classifier as shown by Figure 2.3. Once the containers of the same application have been identified by the application classifier, their data are aggregated by concatenation, that is, the frequency vector traces of different containers described in section 2.2.2 are appended to one another for model training or attack detection. As all feature vectors are aligned with the same dimension, it is easy to concatenate the classified data from distributed containers.

Specifically, each application class has its own file with the feature vector data obtained from multiple containers of the same class. Each feature vector is appended to the file corresponding to the group predicted by the application classifier. The data grouping module performs periodical data segmentation and send the newest data segment to different application models for classified training or attack detection, which will be described next.
2.2.5 Classified Learning and Anomaly Detection

Our classified learning scheme creates and maintains a model ensemble consisting of different models for different application classes shown by Figure 2.4. In this work, we chose autoencoder neural networks for unsupervised model training and anomaly detection to meet our goal of achieving online security attack detection for containerized applications.

The autoencoder network consists of encode and decode regions shown by Figure 2.5. Input data are compressed in the encode region and reconstructed in the decode region. The model replicates its input data after compressing the input data through intermediate neural network layers. During the training phase, a model is built to minimize the difference between its input and output. When the autoencoder model is sufficiently trained, the model is able to produce an output data with small reconstruction error compared with the normal input data. We can then use the reconstruction error to implement anomaly detection. Specifically, when the autoencoder model produces an output with high reconstruction error, we can infer an abnormal input data is detected.

Our neural network consists of four layers of neurons with sigmoid activation in addition
to the input and output layers. Each input data trains the autoencoder model for 10 iterations. We choose to use a small neural network with just four layers and a small number of training iterations in order to achieve online training for the container environment. We implement the autoencoder with Tensorflow.

During training, our objective is to minimize mean squared error (MSE). We perform backpropagation with the Tensorflow implementation of the root mean square propagation (RMSProp) optimizer, whose steps are executed with the following equations. Weight \( w \) is updated according to the formula:

\[
    w_t = w_{t-1} - \alpha \frac{g_t}{\sqrt{rms_t} + \epsilon}
\]

where \( \alpha \) corresponds to learning rate, \( L \) is MSE loss, \( g \) is the gradient of the loss with respect to the weight \( \frac{dL}{dw} \), \( \rho \) and \( \epsilon \) are small Tensorflow default constants, and the root mean square of the gradient \( rms \) is given by:

\[
    rms_t = \rho \times rms_{t-1} + (1 - \rho) \times g_t^2
\]

To determine proper error threshold for anomaly detection, we adopt a statistical approach. Assuming majority of input data are normal data during the training phase, we collect all the reconstruction errors produced by the training data and compute a high percentile value (e.g., 99.9 percentile) as the threshold. We refer to this percentile value as the value of the \textit{training reconstruction error}. This percentile value selection represents the tradeoff between detection rate and false alarm rate. Intuitively, smaller threshold yields more detections but also more false alarms. We conduct experiments to illustrate such tradeoffs in Section 2.3.

2.3 Experimental Evaluation

In this section, we present our experimental evaluation. We implement our prototype system and evaluate it on Amazon EC2 t3.large instances with two 2.5 GHz vCPUs and 8 GB memory running Ubuntu 16.04 64-bit.

2.3.1 Evaluation Methodology

In this subsection, we give details about our evaluation methodology as well as the alternative approaches that we compare our results against.
We investigate 33 recent real-world vulnerabilities documented in the Common Vulnerabilities and Exposures (CVE) database. These selected vulnerabilities appear in 24 open source software, varying from back-end to front-end applications, which covers different types of containerized applications commonly used in practice Carter (2018); Datadog (2018). All the studied vulnerabilities and their attributes are summarized in Table 2.2.

### Experiment setup

As mentioned in Section 2.2, we collect system call data for each vulnerability in multiple containers for training and testing. We design different workload intensity levels that represent...
the normal operation of each application. To emulate real world workload variations in dynamic containers, we change the request rate by multiplying a scale factor for each new container. Each container receives a multiple (i.e. 1x, 2x, 4x and 8x.) of the workload. We use Apache Jmeter to deliver different kinds of workload. For example, we send HTTP GET requests to the web application containers with the PHP vulnerability (CVE-2012-1823). Whereas, application containers providing Network Time Protocol (NTP) services are sent current time requests. While containers are under the appropriate workload, we exploit their security vulnerabilities to investigate our attack detection model. We test four containers for each vulnerability. For each container, the experiments last a total time of seven minutes except in cases where the attack crashes the application. The experiment procedure is as follows. First, we run the application under normal workload conditions for four minutes. Thereafter, we trigger the attack and allow it to run until it succeeds. Finally, we exit the attack after it completes where applicable. Thus, with a sample time of 0.1 seconds, each container contributes about 2400 normal samples. The first minute of each container data is used for training the application classifier, the next two minutes are used for training autoencoder models, and the last four minutes are used for testing the trained models.

Application classification setup

We consider each unique application and version combination in Table 2.2 as a distinct application class. There are two instances, corresponding to CVE-2015-8103 and CVE-2017-12149 vulnerabilities, that share the same application and version number so we consider them to be the same application. We compile a list of 555 system calls that are used by current Linux kernels (as of version 4.19) as the frequency vector. This is done to cover all system calls encountered both in our experiments and in unobserved application environments CDL would operate on. For any new application, we first expand its frequency vector dimension to 555 by inserting zero values at the positions of system calls that are not used during the application run-time. The expanded data are then fed into a random forest classifier.

We use the scikit-learn implementation of random forest scikit learn (2019). In our experiment, every random forest classifier uses 200 decision tree classifiers with no specified maximum depth. To produce a stable output, we use a random state of zero. For each application, we train a random forest classifier using the first minute of each container instance of that application.
Autoencoder neural network setup

CDL adopts a small four layer neural network for each of its individual models. The first two layers form the encoder part of autoencoder while the third and fourth layers form the decoder as depicted in Figure 2.5. Our autoencoder prototype, implemented with Tensorflow, consists of 278 neurons in the first and fourth hidden layers and 70 in the second and third layers. We
set learning rate to 0.001 and utilize the root mean square propagation (RMSProp) optimizer to minimize the mean squared error (MSE) loss function. Once we train an autoencoder model, we run anomaly detection on its training data to analyze its reconstruction errors for choosing a threshold. We select the values corresponding to the 99.9 percentile of those errors as the default training reconstruction error for detecting anomalies.

**Alternative approaches**

We evaluate our classified learning approach against two commonly used existing training methods.

**The sampling method:** Without aggregating training data from multiple containers, each model is trained with a subset of data samples. We call this traditional learning method the sampling method. The drawback of this approach is that the captured portions of historical training data may not adequately represent all normal application behaviors.

**The monolithic method:** The monolithic approach combines data from all containers without distinguishing different applications. This approach assumes that all data would improve the model which may not necessarily be true. For instance, distinct applications that experience differing trends may interfere with one another during the training. This monolithic scenario represents the other extreme where an excessive amount of data is utilized blindly.

**Evaluation metrics**

We use true positive rate (TPR) and false positive rate (FPR) metrics to compare the detection results among different approaches. The calculation for these metrics are given by the following equations where TP denotes the number of samples that the detector correctly identifies while FP denotes the number of samples that the detector falsely identifies and TN denotes the number of the samples the detector correctly rejects while FN denotes the number of the samples the detector falsely rejects.

\[
TPR = \frac{TP}{TP + FN} \tag{2.3}
\]

\[
FPR = \frac{FP}{FP + TN} \tag{2.4}
\]

We also measure the time from when the attack is first detected to the time when the attack succeeds. We use call this time duration the *lead time*. The longer the lead time is, the more likely the attack can be stopped before it compromises the application.
Table 2.3: Classification results among different application classifiers.

<table>
<thead>
<tr>
<th>Application Classifier</th>
<th>TPR</th>
<th>FPR</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVM</td>
<td>49.44%</td>
<td>1.60%</td>
</tr>
<tr>
<td>Extremely Randomized Trees</td>
<td>89.84%</td>
<td>0.33%</td>
</tr>
<tr>
<td>CDL</td>
<td>91.03%</td>
<td>0.29%</td>
</tr>
</tbody>
</table>

2.3.2 Results Analysis

Application classification

We compare the random forest classifier used by CDL with extremely randomized trees (ER Trees) and support vector machine (SVM) classifiers. Similar to the random forest case, we use scikit-learn implementations of both extremely randomized trees and SVM with a random state of zero. Table 2.3 gives the overall true positive rate and false positive rate results of each classifier. In addition, we present the classifier performance across the different threat impact categories in Figure 2.6. Each classifier has to accurately classify system call data from various applications collected under different workload conditions.

SVM, which has only one classifier instance, finds this task difficult. It also generates the lowest true positive rate among all attack categories. In contrast, both random forest and extremely randomized trees are ensemble models which make use of the voting classification results of multiple decision trees. Thus, both CDL and extremely randomized trees achieve much higher true positive rate and much lower false positive rate than SVM. CDL achieves higher true positive rate and lower false positive rate than extremely randomized trees on average. CDL attains the highest true positive rate among all threat impact categories.

Classified detection

Table 2.4 shows the detection results over all vulnerability attacks used in our experiments. The results show that CDL can successfully detect 31 out of 33 attacks while the monolithic method can only detect 20 out of 33 attacks due to conflicting training data. Although the sampling method can detect 32 out of 33 attacks, it produces orders of magnitude higher false positives than CDL, shown by Table 2.5. We also varied the percentile threshold in the autoencoder model and show the results. The results using 99 percentile of training reconstruction error are listed in Table 2.6. We observe that CDL can achieve both high true positive rate and low false positive rate.

Figure 2.7 and Figure 2.8 illustrate the detection results across all attack threat impact categories. CDL models achieve the highest true positive rate in detecting “execute arbitrary
Table 2.4: Detection results of all CVE examined among different models.

<table>
<thead>
<tr>
<th>Threat Impact</th>
<th>CVE ID</th>
<th>Detected</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Sampling</td>
<td>Monolithic</td>
<td>CDL</td>
<td></td>
</tr>
<tr>
<td>Return a shell and execute arbitrary code</td>
<td>CVE-2012-1823</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CVE-2014-3120</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CVE-2015-1427</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CVE-2015-2208</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CVE-2015-3306</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CVE-2015-8103</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CVE-2016-3088</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CVE-2016-9920</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CVE-2016-10033</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CVE-2017-7494</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CVE-2017-8291</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CVE-2017-11610</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CVE-2017-12149</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CVE-2017-12613</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Execute arbitrary code</td>
<td>CVE-2014-6271</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CVE-2015-8562</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CVE-2016-3714</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CVE-2017-5638</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CVE-2017-12794</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CVE-2018-16509</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CVE-2018-19473</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CVE-2019-6116</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Disclose credential information</td>
<td>CVE-2014-0160</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CVE-2015-5531</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CVE-2017-7229</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CVE-2017-8917</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CVE-2018-15473</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CVE-2020-1938</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Consume excessive CPU</td>
<td>CVE-2014-0050</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CVE-2016-6515</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Crash the application</td>
<td>CVE-2015-5477</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CVE-2016-7434</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Escalate privilege level</td>
<td>CVE-2017-12635</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Total successful detection</td>
<td></td>
<td>32</td>
<td>20</td>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.5: Summary of detection results among different models.

<table>
<thead>
<tr>
<th>Model</th>
<th>TPR</th>
<th>FPR</th>
<th>Lead time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sampling</td>
<td>89.39%</td>
<td>12.74%</td>
<td>12.16s</td>
</tr>
<tr>
<td>Monolithic</td>
<td>44.70%</td>
<td>0.08%</td>
<td>5.77s</td>
</tr>
<tr>
<td>CDL</td>
<td>74.24%</td>
<td>0.24%</td>
<td>9.23s</td>
</tr>
</tbody>
</table>

Table 2.6: Detection results among different models with 99 percentile of training reconstruction error.

<table>
<thead>
<tr>
<th>Model</th>
<th>TPR</th>
<th>FPR</th>
<th>Lead time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sampling</td>
<td>90.15%</td>
<td>27.88%</td>
<td>12.77s</td>
</tr>
<tr>
<td>Monolithic</td>
<td>52.27%</td>
<td>0.93%</td>
<td>6.01s</td>
</tr>
<tr>
<td>CDL</td>
<td>81.06%</td>
<td>1.07%</td>
<td>10.91s</td>
</tr>
</tbody>
</table>

code" attacks and the lowest false positive rate in detecting “disclose credential information"
and “consume excessive CPU” attacks. Due to insufficient training, sampling models usually do not have enough knowledge of normal behavior. Thus, they under-fit the data and have high true positive rate and high false positive rate at the same time. The monolithic model is trained using all the training data from all the different applications. Thus, it over-fits the
data and has low true positive rate and low false positive rate. CDL achieves good trade-off between true positive rate and false positive rate. In terms of true positive rate, CDL performs well in three categories: “return a shell and execute arbitrary code”, “execute arbitrary code” and “escalate privilege level”, but less accurately in the remaining three categories: “disclose
credential information”, “consume excessive CPU” and “crash the application”. We discuss in the next section, Section 2.3.3, our observation that attacks in those three categories usually result in significant changes to the top frequent system calls, while the rest three categories usually happen suddenly with little deviation in system calls.

The lead time comparison between two different percentile of training reconstruction error
values can be found in Figure 2.9 as well as in Table 2.5 and Table 2.6. Higher percentile of training reconstruction error generally leads to lower lead time. However, the lead time result of the “escalate privilege level” category is not sensitive to the percentile of training reconstruction error used. CDL models have much larger lead time than the monolithic models in almost all cases except in the “execute arbitrary code” attacks. Sampling models generally have the largest lead time among all models. Note, however, that sampling models have much higher false positive rate which makes them unsuitable to apply in practice.

**System Run Time Analysis**

Lastly, we evaluate the run time for different operations of the CDL system. Table 2.7 shows the time per sample for key steps in our system. Classifier training is the most time consuming step and it needs about 135.35 ms to train each sample. Autoencoder training takes about 2.53 ms for training each sample. Application classification takes about 3.80 ms to complete and attack detection by autoencoder model takes about 3.50 ms for each sample. Recall that each sample covers a sampling period of 100 ms, so CDL can detect attacks in real time. Overall, the run time analysis show that CDL is lightweight and applicable for detecting attacks in real time under real world settings.

**2.3.3 Case Study**

In this subsection, we discuss an example in detail to show how CDL's identification results help to better detect attacks. Thereafter, we highlight an attack from each of the other categories.

**Execute arbitrary code**

We investigate an attack to the Joomla CVE-2015-8562 exposure. The attack takes advantage of a vulnerability in the HTTP User-Agent field of a request that allows PHP object injection. In particular, the attack we trigger causes a deserialization error. We study the alarms in the containers detected by CDL but not by sampling and monolithic models by examining the
average system call frequencies in the normal period to compare with that of the attack period. In all containers, the top frequent system calls during the alarm period are the following: \texttt{fstat}, \texttt{lstat}, \texttt{access}, \texttt{close} and \texttt{open}. These are also the system calls with the most-changed average counts between the normal and the attack period. \texttt{Fstat} and \texttt{lstat} are responsible for retrieving file status information like file owner ID, file size or time of last file access. \texttt{Access} checks whether the calling process can access a specified file. \texttt{Open} readies a requested file, while \texttt{close} discontinues the use of a file descriptor. These calls are very relevant to this Joomla attack that accesses cookies secured in a MySQL database. Attempts are also made to convert the file content from its serial format into data structures in memory.

\textbf{Return a shell and execute arbitrary code}

Here, we highlight the CVE-2016-9920 vulnerability of the Roundcube mail application. This exposure allows custom parameters to be accepted in the mail fields of the application. This allows an attacker to insert commands that execute and can return a shell. The top system calls made by the container while under attack are \texttt{read}, \texttt{stat}, \texttt{close}, \texttt{open}, \texttt{write} and \texttt{mmap}. The attack triggers Roundcube to save a PHP shell command into a file that will be run. Thus, \texttt{stat} obtains file information that \texttt{write} can use to record the crafted command. When the file contents are invoked to run, they are mapped into memory with \texttt{mmap}. Finally, \texttt{close} and \texttt{open} calls are made to use file descriptors as needed.

\textbf{Disclose credential information}

Next, we discuss the Heartbleed bug (CVE-2014-0160) of the OpenSSL library. The bug allows a malicious user to access protected information outside an assigned memory buffer when receiving SSL Heartbeat responses. We record the following top system calls during the attack: \texttt{gettimeofday}, \texttt{stat}, \texttt{poll}, \texttt{writev}, \texttt{close}, and \texttt{open}. The attack incorporates a timeout mechanism while waiting for a server response to confirm whether the server is vulnerable. This is responsible for the \texttt{gettimeofday} calls, which significantly outnumber the other system calls. \texttt{Stat} and \texttt{writev} calls retrieve file information and write from multiple buffers respectively, to prepare Heartbeat response messages. Meanwhile, \texttt{poll} waits for the above files to ready their I/O data. \texttt{Open} and \texttt{close} calls are also notably involved in managing the life cycles of the files used.

\textbf{Consume excessive CPU}

In CVE-2016-6515, OpenSSH before version 7.3 does not limit the length for passwords, which can result in a denial-of-service attack (DoS attack) by a long string. The top frequent system calls during the alarm period are: \texttt{close}, \texttt{read}, \texttt{mmap}, \texttt{open}, \texttt{mprotect} and \texttt{fstat}. Those system
calls are also the top frequent system calls before triggering the attack, but we observe the frequency of each system calls increase by at most 5 times. Processing password information is sensitive. Thus, we deduce that `mmap` maps related disk contents into memory while `mprotect` ensures proper access restrictions. The remaining system calls get related file information, read the data and then close the file descriptor when done. The infinite loop triggered by the attack would likely cause these tasks to occur repeatedly.

**Crash the application**

According to CVE-2016-7434, NTP suffers from a null pointer reference which could lead to crashing. Because of the nature of crashing, the attack period usually lasts less than 0.3 second. The top frequent system calls during the alarm period are: `gettid`, `rt_sigprocmask`, `read`, `write`, `clock_gettime` and `recvmsg`. We believe that the attack happens in such a short time, that there is not a significant change in system call composition. Nevertheless, the NTP attack is triggered upon receiving a malicious most recently used list (mrulist) query. `Gettid` may be used to obtain the thread ID of the request delivered by `recvmsg` over the socket connection. The mrulist then needs to be read and processed by `read` and `write` calls. Meanwhile, `clock_gettime` would correspond to expected NTP workload to acquire time information.

**Escalate privilege level**

We analyze the CVE-2017-12635 Couchdb vulnerability. Because of different ways of parsing JSON objects by JavaScript and Erlang, this vulnerability can be used to gain access to create an administrator account. The top frequent system calls during the alarm period are: `close`, `epoll_wait`, `sched_yield`, `futex`, and `switch`. We notice that the appearance of system call `close` is significantly larger than that of any other system calls. The reason is that the attack changes the behavior of the application, so that it deletes file descriptors more often. The attack is administered with multiple HTTP requests with the close option set in the `Connection` field. The `close` system call will be useful for closing unneeded files related to those packets.

### 2.4 Summary

In this chapter, we have presented CDL, a new classified distributed learning framework that aims at achieving practical and efficient security attack detection for containerized applications. CDL integrates online application classification and application-specific anomaly detection models to overcome the challenges of lacking sufficient training data for individual short-lived containers. We have implemented a prototype of CDL and conducted experiments over 33 real
world vulnerability exploits in 24 commonly used applications. Our results show that CDL can reduce false positive rates from over 12% to 0.24% compared to traditional learning methods without aggregating training data from different containers and increase the true positive rate from 45% to 74% compared to simple training data aggregation without performing application classifications. CDL supports real time security attack detection, which makes it practical for production computing environments.
3.1 Introduction

Container-based distributed system platforms have gained tremendous popularity in many real world applications because of their cost-effectiveness and accessibility. However, containers also open new attack surfaces to malicious attackers. Recent studies Sulinski (2017); Shu et al. (2017); Kwon and Lee (2020) have shown that containers are vulnerable to various security attacks. For example, Tesla suffered a cryptojacking attack in February 2018 RedLock (2018). The hackers infiltrated Tesla’s Kubernetes console to steal sensitive data such as telemetry. Besides data exposure, the hackers performed crypto mining with low resource intensity to evade detection. Furthermore, Apache Log4j recently disclosed a vulnerability in December 2021 that seriously affected distributed systems worldwide, including container clusters of Standards and Technology (2021). Java software systems extensively use the open-source Log4j logging
framework, exposing them to remote code execution attacks. The Google Cloud open source insights team estimated that over 35,000 artifacts in the Maven Central repository are vulnerable, four times that of the average vulnerability Wetter and Ringland (2021). In just four days after the disclosure, Check Point Research reported over 800,000 global attacks to the vulnerability Team (2021).

Traditional intrusion detection systems Lin et al. (2018b) typically employ rule-based approaches, which however cannot adapt to highly dynamic container environments and often miss detecting emergent attack behaviors that have not been captured by existing detection rules. Previous work Anthi et al. (2019); Lin et al. (2020); Tunde-Onadele et al. (2020); Idhammad et al. (2018) has proposed to apply different machine learning methods including supervised learning, unsupervised learning, and semi-supervised learning, to achieve effective security attack detection. Supervised learning typically achieves higher detection accuracy than unsupervised learning. However, it is difficult to collect sufficient high quality labelled training data in highly ephemeral container-based computing environments. Anomaly detection methods based on unsupervised learning are much easier to be deployed in real world dynamic computing environments, which do not require any labelled training data. However, due to the lack of labelled training data, anomaly detection methods often suffer from high false alarms. Previous work also proposed semi-supervised learning methods Idhammad et al. (2018) for security attack detection, which start with a trained supervised learning model and use unsupervised methods to augment the supervised model by providing labels to unlabelled data. However, the semi-supervised approach still requires labelled training data in order to train the initial supervised model.

In this chapter, we present a new self-supervised hybrid learning (SHIL) system for performing adaptive online security attack detection in dynamic containerized applications. SHIL aims at improving security detection accuracy without requiring any labelled training data which are particularly difficult to obtain in ephemeral container-based systems. In contrast to semi-supervised learning, which starts from supervised models, SHIL starts with unsupervised models and employs supervised learning methods for cross validation purposes only. The rationale behind our approach is based on the observation that most false alarms occur when the measurement sample is within close vicinity of the anomaly detection threshold, which is called a boundary case. Cross-validations using multiple different learning methods over boundary cases can effectively filter out false alarms without missing true attack anomalies.

SHIL leverages non-intrusive, light-weight system call monitoring tools Sysdig (2016) to achieve practical security attack detection for containers. SHIL consists of three key modules: 1) unsupervised anomaly detection, which leverages autoencoder neural networks An and Cho (2015) to achieve fast attack detection without requiring any labelled training data; 2) hybrid...
**alert validation**, which identifies boundary case anomalies and performs false alarm filtering by cross validating the anomalies using the supervised learning method random forest tree Ho (1995); and 3) **self-supervised model creation**, which performs outlier detection using isolation forest Liu et al. (2008) over a window of recent system call data upon an attack alert raised by either SHIL or a third party attack detection tool. The outlier detection allows SHIL to generate training labels automatically to achieve self-supervised learning.

Specifically, this chapter makes the following contributions:

- We propose a new self-supervised hybrid machine learning approach to achieving effective security attack detection with few false alarms.
- We describe a self-supervised model creation method which leverages both the anomalies produced by SHIL and an outlier detection method to produce training labels automatically for the supervised learning method.
- We implement and evaluate a prototype of SHIL over 41 recent real critical vulnerabilities with high CVSS scores including the high impact Apache Log4j vulnerability, in 28 commonly used server applications.

Our experimental results show that SHIL can reduce false alarms by 39% to 91% compared to existing unsupervised or supervised learning methods while achieving higher or similar detection rates. SHIL is light-weight, which makes it practical for large-scale container based computing environments.

The rest of the chapter is structured as follows. Section 3.2 presents the system design in detail. Section 3.3 describes our experimental evaluation methodology and our experimental results. Section 3.4 concludes this chapter.

### 3.2 System Design

In this section, we present the design of the SHIL system. We first provide a system overview. Next, we describe each component in detail.

SHIL takes a self-supervised hybrid machine learning approach to security attack detection. As highlighted in Figure 3.1, the system consists of four key integrated components: 1) **system call pre-processing**, 2) **unsupervised anomaly detection**, 3) **hybrid alert validation**, and 4) **self-supervised model creation**.

SHIL leverages light-weight system call monitoring tool Sysdig (2016) to detect security attacks. Previous intrusion detection work have revealed that the program executes a locally consistent set of system call sequences during normal operations and attacks often exhibit
significant changes in system call invocations Forrest et al. (1996); Liu et al. (2018). The system call pre-processing module extracts a system call frequency vector feature from raw system call traces. Each system call frequency vector denotes the number of invocations for all system call types (e.g., `sys_read`) within the sampling interval (e.g., 100 milliseconds). Table 3.1 shows examples of frequency vectors from an OpenSSH 7.2p2 container during an exploit of the CVE-2016-6515 vulnerability. The attack starts at timestamp 1586496672891 and ends at timestamp 1586496674291. During the attack, we observe the frequency of the `execve` and `lstat` system calls increase when the attack starts being detected until the attack completes. Meanwhile, the frequencies of both `access` and `mmap` calls sharply increase during the same period. Notice that our self-supervised hybrid learning approach can be applied to any system call features (e.g., n-grams Forrest et al. (1996)) adopted by the intrusion detection system. In this chapter, we choose to use the system call frequency vector features for both low cost training and real-time attack detection.

The unsupervised anomaly detection component detects anomalies in the system call frequency vector data using an autoencoder neural network. The trained model computes the difference between an input vector and its reconstruction of the vector into a value called reconstruction error. The model compares the reconstruction error against a pre-defined percentile value (e.g., 99 percentile value of all reconstruction errors) to detect anomalies. Section 3.2.1 will provide details about this module.
The hybrid alert validation component checks whether the detected anomaly is a boundary case (e.g., within a small deviation from the normal value) and invokes the supervised model to perform cross validation if it is considered to be a boundary case. The goal is to filter out most false alarms produced by the boundary cases and only raise attack alerts when both unsupervised and supervised models confirm the alert. Section 3.2.2 will provide details about this module.

To achieve supervised learning without requiring manual data labelling, SHIL adopts a self-supervised learning approach using the self-supervised model creation method. Upon an attack alert, SHIL creates a supervised attack detection model such as random forest using a window of system call frequency vector data before and after the attack is detected. Instead of relying on manual data labelling, SHIL automatically creates training data labels by performing outlier detection using isolation forest models Liu et al. (2008). Section 3.2.3 will provide details about this module.

### 3.2.1 Unsupervised Anomaly Detection

In contrast to previous semi-supervised learning methods which start from supervised models, SHIL starts from unsupervised anomaly detection models. We choose our design based on two key rationales: 1) unsupervised anomaly detection does not require labelled training data as it relies on recognizing deviations from normal behaviors; and 2) unsupervised anomaly detection has the ability to detect unknown attacks. So SHIL can inherit all the advantages of the unsupervised learning methods by only involving supervised models for performing cross validations on boundary cases.

Our unsupervised anomaly detection leverages autoencoder neural networks. The autoencoder neural network is an artificial neural network model with a symmetric structure, capable of reconstructing its input as the output. The network consists of two major sections: the encoder and the decoder. The encoder reduces the frequency vectors into increasingly lower dimensions from the input layer to the narrowest hidden layer at the autoencoder center. Conversely, the decoder reconstructs the low dimension vector representation from the hidden layer to the output layer. We train each autoencoder repeatedly with certain number of iterations (e.g., 10 iterations) to allow it to rapidly adjust its weights under various system call activity.

During detection, the difference between the input and output of the autoencoder model is referred to as the reconstruction error. Anomalous frequency vector samples that deviate from the model of normal samples are likely to be reconstructed poorly, resulting in more substantial error degrees than others. Therefore, we implement the anomaly detection by comparing the
Table 3.1: A frequency vector sample for the OpenSSH application (CVE-2016-6515). An attack is triggered at $t = 1586496672891$. Anomaly detection raises alarms from $t = 1586496673091$ to $t = 1586496673191$. The entries with asterisks (*) are detected outliers.

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>System Call Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>access</td>
</tr>
<tr>
<td>1586496672491</td>
<td>0</td>
</tr>
<tr>
<td>1586496672591</td>
<td>0</td>
</tr>
<tr>
<td>1586496672691</td>
<td>0</td>
</tr>
<tr>
<td>1586496672791</td>
<td>0</td>
</tr>
<tr>
<td>1586496672891 (attack starts)</td>
<td>0</td>
</tr>
<tr>
<td>1586496672991</td>
<td>79</td>
</tr>
<tr>
<td>1586496673091 (attack detected)</td>
<td>136</td>
</tr>
<tr>
<td>1586496673191*</td>
<td>268</td>
</tr>
<tr>
<td>1586496673291*</td>
<td>209</td>
</tr>
<tr>
<td>1586496673391</td>
<td>164</td>
</tr>
<tr>
<td>1586496673491</td>
<td>70</td>
</tr>
<tr>
<td>1586496673591</td>
<td>190</td>
</tr>
<tr>
<td>1586496673691</td>
<td>76</td>
</tr>
<tr>
<td>1586496673791*</td>
<td>129</td>
</tr>
<tr>
<td>1586496673891</td>
<td>130</td>
</tr>
<tr>
<td>1586496673991</td>
<td>82</td>
</tr>
<tr>
<td>1586496674091*</td>
<td>159</td>
</tr>
<tr>
<td>1586496674191</td>
<td>79</td>
</tr>
<tr>
<td>1586496674291 (attack succeeds)</td>
<td>91</td>
</tr>
<tr>
<td>1586496674391</td>
<td>192</td>
</tr>
<tr>
<td>1586496674491</td>
<td>50</td>
</tr>
<tr>
<td>1586496674591</td>
<td>0</td>
</tr>
<tr>
<td>1586496674691</td>
<td>0</td>
</tr>
</tbody>
</table>

reconstruction error of the current sample with a pre-defined reconstruction error threshold. Specifically, we select a certain percentile value (e.g. 99 percentile) of the reconstruction errors during the training phase as the threshold of our anomaly detection model. The rationale is that the majority of the training data are normal data which have small reconstruction errors.

We use a model ensemble approach that trains a separate model for each group of containers that run the same application with the same version. For example, we create an anomaly detection model to monitor a group of containers running JBoss 6.1.0. The application specific models provide higher accuracy over monolithic models trained over different applications because of fewer conflicting training data Lin et al. (2020).

### 3.2.2 Hybrid Alert Validation

The anomaly detection percentile threshold typically controls the trade-off between the detection rate and false positive rate. If we want the anomaly detection model detects more attacks, we need to configure relatively lower percentile threshold in order to capture more anomalous
behaviors. However, lower percentile threshold inevitably produces more false alarms since the anomaly detection model is more likely to detect rogue anomalies caused by dynamic container-based computing environments and transient workload fluctuations. During our experiments, we observe that majority of those false alarms occur in boundary cases where the reconstruction error is above the anomaly detection threshold within a small range. For example, the error range between 100% and 110% of the threshold contains over 50% of the false alarms. Thus, we propose to identify those boundary cases and perform cross-validations over those boundary cases using a self-supervised model trained by a supervised learning method.

Recall that the unsupervised anomaly detection calculates the reconstruction error of each frequency vector that is continuously compared against an error threshold to make an anomaly decision. We define the error range above the threshold that contains the majority of false alarms as the boundary case. SHIL then feeds the boundary case anomaly into a pre-trained supervised model for cross validation. If the supervised model does not classify the measurement sample as anomalous, SHIL deems it to be a false alarm from the unsupervised anomaly detection model and drops the alert. Notice that SHIL only applies the cross validation over the boundary cases so that SHIL can avoid dropping alerts about unknown attacks that are often missed by supervised models.

Figure 3.2: Self-supervised model creation.
3.2.3 Self-supervised Model Creation

Supervised models can typically achieve high detection accuracy when sufficient high quality labelled training data are available. However, it is often difficult to obtain labelled training data in production environments, especially in highly dynamic container-based computing environments. Moreover, for dynamic advanced attacks, it is extremely challenging if not totally impossible to accurately label each measurement sample as normal or abnormal at a fine-grained time scale (e.g., every 10 milliseconds) required by supervised learning models. When an attack is first started, it may not manifest in the system call frequency vector changes immediately. For example, in Table 3.1, the attack is triggered at timestamp 1586496672891, however there is not much change until timestamp 1586496672991. After that we can see the
increase of access, execve, lstat and mmap system call frequencies. In addition, the attack may have a lasting impact resulting in non-deterministic behaviors (i.e., normal, abnormal, or mixed activities) after the attack succeeds.

To address the challenges of lacking high-quality labelled training data, we propose a self-supervised model training approach to achieving automatic data labelling, as shown in Figure 3.2. Intuitively, measurement samples outside the attack period represent normal fluctuating behaviors in a dynamic application and measurement samples collected during the attack period represent abnormal behaviors incurred by the attack. However, advanced attacks might not exhibit abnormal behaviors constantly throughout the attack period as shown in our experiments. If we label all measurement samples during the attack period as anomaly data, it is highly possible to create a model that tends to raise many false alarms. To address the challenge, we introduce an outlier detection process to preprocess the training data.

Specifically, our self-supervised model training consists of the following major steps. First, we perform outlier detection over all measurement data during the attack period identified by our unsupervised learning methods or other third-party attack detection tools. Second, we perform a similarity check between each outlier detected during the attack period with all the normal measurement samples collected during a small window preceding the attack detection time. In our experiments, we employed Manhattan distance between two frequency vectors to measure the similarity. We then further filter those outliers which resemble the normal execution behavior. The rationale behind our approach is to capture the true attack behavior while minimizing the false positive likelihood. Thus, we only label those true outliers as anomalous in the supervised model training. Those filtered outliers which resemble normal execution data will be relabelled into “normal” to reinforce the normal behavior training.

During our experimental study, we observe that only a small portion of samples during the attack period contain significant changes in system call frequencies, which are often detected as outliers. Thus, our approach provides fine-grained labelling instead of assuming all measurement samples during the attack period as abnormal. Our experiments show that such a fine-grained data labeling approach can effectively induce high quality supervised learning models.

We employ the isolation forest Liu et al. (2008) outlier detection method to generate fine-grained labels. The isolation forest consists of an ensemble of decision trees. The goal of each decision tree is to isolate each input vector from the others. Each tree is split on a random value in the possible range of a randomly selected system call type (e.g., read) until all vectors are separated. Since outliers are uncommon and numerically different from normal samples, it takes fewer decisions to distinguish them from others. Thus, outliers are found closer to the root of the isolation trees.
Figure 3.3 illustrates the outlier detection of a simple frequency vector sample on a constructed isolation forest. The input vector is isolated in each tree by following the decision tree path consisting of the highlighted links. For instance, in the left-most tree, the vector is separated with a single split of the $\text{read} > 1$ node. Whereas, the tree in the center of the forest separates the vector with two decision nodes: $\text{read} > 3$ and $\text{write} < 3$.

To determine whether the vector is an outlier, an anomaly score is computed as follows.

$$s(x, n) = 2^{\frac{E(h(x))}{c(n)}}$$

(3.1)

Given a set of $n$ instances, the anomaly score $s(x, n)$ of an instance $x$ in the isolation forest is defined by Equation (3.1), where $h(x)$ is the path length of $x$ from the root, $E(h(x))$ is the average of $h(x)$ from the collection of isolation trees and $c(n)$ is the average path length of unsuccessful search in an isolation tree. In a binary search tree, $c(n)$ is estimated by the average height from the root to its leaf nodes Liu et al. (2008). When $s$ is close to 1, it is nearly certain that $x$ is an outlier. Whereas, when $s$ is close to 0, $x$ is highly likely to be a normal sample. In Figure 3.3, the heights $h(x)$ of the vector sample in each tree from left to right are one, two, and one, respectively, while the average height of the leaf nodes $c(n)$ is 2.25. The sample generates an anomaly score of 0.66. If we set the outlier detection threshold as 0.5, this input vector is labelled as an outlier.

For example, Table 3.1 shows the labelling results produced by our outlier detection method. Those entries marked with asterisk (*) indicate detected outliers which will be labelled as anomaly in the supervised model’s training data. The measurement samples before the attack detection time and all non-outlier measurements are labelled as normal training data. We can see that only those outliers within the attack period show abnormal behavior while those non-outlier samples still exhibit similar behaviors to those measurements during the normal execution period.

After the outlier detection module produces labelled training data, it feeds the labelled training data into a supervised learning model. During our experiments, we choose the random forest (RF) learning method as our supervised learning method Ho (1995) for its simplicity and effectiveness. The RF model is an ensemble of many decision trees, which makes its final classification based on the majority classification of its constituents. During training, each tree chooses a split value from a random subset of its features to optimize its anomaly classification decision. These characteristics make the RF model resilient to noises. For a single tree, the fraction of abnormal samples in the output leaf node yields a probability value. The overall prediction probability is the the average prediction probability over all the decision trees. When the prediction probability is above a pre-defined threshold such as 60%, the frequency vector...
is classified as abnormal.

### 3.3 Experimental Evaluation

In this section, we first describe our evaluation methodology. Next, we compare SHIL with a set of alternative schemes. We implement a prototype of SHIL and evaluate it on a desktop with four 3.4 GHz cores and 8 GB memory running Ubuntu 18.04 64-bit.

#### 3.3.1 Evaluation Methodology

**Real-world vulnerabilities.** We evaluate 41 vulnerabilities from 28 applications listed in the common vulnerabilities and exposures (CVE) database, which include many applications commonly used in production environments Carter (2019); Newcomb (2021). Table 3.2 shows the complete list of the CVEs, including the common vulnerability scoring system (CVSS) score v2.0, application name, and version of each entry. We focus on CVEs in recent years, including the recent high-impacting Log4j CVE (CVE-2021-44228). We classify the vulnerabilities into six categories according to the threat impact of the attack. The threat impact categories comprise attacks that 1) return a shell and execute arbitrary code, 2) execute arbitrary code, 3) disclose credential information, 4) consume excessive CPU, 5) crash the application, and 6) escalate privilege level.

**Experiment Setup.** For each vulnerability, we set up four Docker containers under Ubuntu 16.04 LTS. We use Apache JMeter to deliver a different multiple of workload to each of the four containers, i.e. 1x, 2x, 4x and 8x. We design suitable workload for each vulnerability according to the kind of traffic the main containerized application accepts. For example, we simulate traffic using HTTP GET requests to the containers of CVE-2020-1938 because the application, Apache Tomcat, is a web server software. Once the container starts running, we use Sysdig to collect seven minutes of its system call activity, unless the container exits early due to an attack (such as the attack to CVE-2015-5477 that crashes the BIND application). The short duration gives enough samples for our models and aligns with the ephemeral nature of containers.

Each experiment is conducted as follows. First, we let the container run for four minutes under the appropriate normal workload. Next, we trigger the attack using open source exploit code around the start of the fifth minute and let the attack continue running until it succeeds. Meanwhile, the exploit program logs the attack triggering time and attack success time due to our modification of the original program from exploit databases. Lastly, we stop the attack where applicable. After finishing the experiment, we process the collected system calls into system call frequency vectors using a sampling rate of 100 milliseconds.
### Table 3.2: List of explored real-world vulnerabilities.

<table>
<thead>
<tr>
<th>Threat Impact</th>
<th>CVE ID</th>
<th>CVSS Score</th>
<th>Application</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return a shell and execute arbitrary code</td>
<td>CVE-2012-1823</td>
<td>7.5</td>
<td>PHP</td>
<td>5.4.1</td>
</tr>
<tr>
<td></td>
<td>CVE-2014-3120</td>
<td>6.8</td>
<td>Elasticsearch</td>
<td>1.1.1</td>
</tr>
<tr>
<td></td>
<td>CVE-2015-1427</td>
<td>7.5</td>
<td>Elasticsearch</td>
<td>1.4.2</td>
</tr>
<tr>
<td></td>
<td>CVE-2015-2208</td>
<td>7.5</td>
<td>phpMoAdmin</td>
<td>1.1.2</td>
</tr>
<tr>
<td></td>
<td>CVE-2015-3306</td>
<td>10.0</td>
<td>ProFTPD</td>
<td>1.3.5</td>
</tr>
<tr>
<td></td>
<td>CVE-2015-8103</td>
<td>7.5</td>
<td>JBoss</td>
<td>6.1.0</td>
</tr>
<tr>
<td></td>
<td>CVE-2016-3088</td>
<td>7.5</td>
<td>Apache ActiveMQ</td>
<td>5.11.1</td>
</tr>
<tr>
<td></td>
<td>CVE-2016-9920</td>
<td>6.0</td>
<td>Roundcube</td>
<td>1.2.2</td>
</tr>
<tr>
<td></td>
<td>CVE-2016-10033</td>
<td>7.5</td>
<td>PHPMailer</td>
<td>5.2.16</td>
</tr>
<tr>
<td></td>
<td>CVE-2017-7494</td>
<td>10.0</td>
<td>Samba</td>
<td>4.5.9</td>
</tr>
<tr>
<td></td>
<td>CVE-2017-8291</td>
<td>6.8</td>
<td>Ghostscript</td>
<td>9.2.1</td>
</tr>
<tr>
<td></td>
<td>CVE-2017-11610</td>
<td>9.0</td>
<td>Supervisor</td>
<td>3.3.2</td>
</tr>
<tr>
<td></td>
<td>CVE-2017-12149</td>
<td>7.5</td>
<td>JBoss</td>
<td>6.1.0</td>
</tr>
<tr>
<td></td>
<td>CVE-2017-12615</td>
<td>6.8</td>
<td>Apache Tomcat</td>
<td>8.5.19</td>
</tr>
<tr>
<td>Execute arbitrary code</td>
<td>CVE-2014-6271</td>
<td>10.0</td>
<td>Bash</td>
<td>4.2.37</td>
</tr>
<tr>
<td></td>
<td>CVE-2015-8562</td>
<td>7.5</td>
<td>Joomla</td>
<td>3.4.2</td>
</tr>
<tr>
<td></td>
<td>CVE-2016-3714</td>
<td>10.0</td>
<td>ImageMagick</td>
<td>6.7.9</td>
</tr>
<tr>
<td></td>
<td>CVE-2017-5638</td>
<td>10.0</td>
<td>Apache Struts 2</td>
<td>2.5.0</td>
</tr>
<tr>
<td></td>
<td>CVE-2017-12794</td>
<td>4.3</td>
<td>Django</td>
<td>1.11.4</td>
</tr>
<tr>
<td></td>
<td>CVE-2018-11776</td>
<td>9.3</td>
<td>Apache Struts 2</td>
<td>2.3.34</td>
</tr>
<tr>
<td></td>
<td>CVE-2018-16509</td>
<td>9.3</td>
<td>Ghostscript</td>
<td>9.23</td>
</tr>
<tr>
<td></td>
<td>CVE-2018-19475</td>
<td>6.8</td>
<td>Ghostscript</td>
<td>9.25</td>
</tr>
<tr>
<td></td>
<td>CVE-2019-6116</td>
<td>6.8</td>
<td>Ghostscript</td>
<td>9.26</td>
</tr>
<tr>
<td></td>
<td>CVE-2019-5420</td>
<td>7.5</td>
<td>Rails</td>
<td>5.2.2</td>
</tr>
<tr>
<td></td>
<td>CVE-2020-17530</td>
<td>7.5</td>
<td>Apache Struts 2</td>
<td>2.5.25</td>
</tr>
<tr>
<td></td>
<td>CVE-2021-44228</td>
<td>9.3</td>
<td>Apache Solr</td>
<td>8.11.0</td>
</tr>
<tr>
<td></td>
<td>(log4j)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Disclose credential information</td>
<td>CVE-2014-0160</td>
<td>5.0</td>
<td>OpenSSL</td>
<td>1.0.1e</td>
</tr>
<tr>
<td></td>
<td>CVE-2015-5531</td>
<td>5.0</td>
<td>Elasticsearch</td>
<td>1.6.0</td>
</tr>
<tr>
<td></td>
<td>CVE-2017-7529</td>
<td>5.0</td>
<td>Nginx</td>
<td>1.13.2-1</td>
</tr>
<tr>
<td></td>
<td>CVE-2017-8917</td>
<td>7.5</td>
<td>Joomla</td>
<td>3.7.0</td>
</tr>
<tr>
<td></td>
<td>CVE-2018-15473</td>
<td>5.0</td>
<td>OpenSSH</td>
<td>7.7p1</td>
</tr>
<tr>
<td></td>
<td>CVE-2020-1938</td>
<td>7.5</td>
<td>Apache Tomcat</td>
<td>9.0.30</td>
</tr>
<tr>
<td></td>
<td>CVE-2021-28164</td>
<td>5.0</td>
<td>Jetty</td>
<td>9.4.37</td>
</tr>
<tr>
<td></td>
<td>CVE-2021-28169</td>
<td>5.0</td>
<td>Jetty</td>
<td>9.4.40</td>
</tr>
<tr>
<td></td>
<td>CVE-2021-34429</td>
<td>5.0</td>
<td>Jetty</td>
<td>9.4.40</td>
</tr>
<tr>
<td></td>
<td>CVE-2021-41773</td>
<td>4.3</td>
<td>Apache HTTP Server</td>
<td>2.4.49</td>
</tr>
<tr>
<td>Consume excessive CPU</td>
<td>CVE-2014-0050</td>
<td>7.5</td>
<td>Apache CommonsFileUpload</td>
<td>1.3.1</td>
</tr>
<tr>
<td></td>
<td>CVE-2016-6515</td>
<td>7.8</td>
<td>OpenSSH</td>
<td>7.2p2</td>
</tr>
<tr>
<td>Crash the application</td>
<td>CVE-2015-5477</td>
<td>7.8</td>
<td>BIND</td>
<td>9</td>
</tr>
<tr>
<td>Escalate privilege level</td>
<td>CVE-2016-7434</td>
<td>5.0</td>
<td>NTP</td>
<td>1.4.2.8</td>
</tr>
<tr>
<td></td>
<td>CVE-2017-12635</td>
<td>10.0</td>
<td>CouchDB</td>
<td>2.1.0</td>
</tr>
</tbody>
</table>

**SHIL Prototype Implementation.** We use TensorFlow to build an autoencoder (AE) model using four hidden layers with 278 neurons in the first and fourth hidden layers and 70 neurons in the second and third hidden layers. For our autoencoder model, we measure the reconstruction error using root mean square error (RMSE) as defined in Equation (3.2), where \( N \) is the number of samples, \( y_i \) is the value of input, and \( \hat{y}_i \) is the value of the output.
\[
RMSE = \sqrt{\frac{\sum_{i=1}^{N}(\hat{y}_i - y_i)^2}{N}}
\]  

(3.2)

We adopt the scikit-learn implementation of isolation forest and set the contamination threshold to be 0.5 after experimentation. Each container has its own attack and workload characteristics, thus each container has its own isolation forest model that fits and predicts the outliers within its alerted attack period.

The outliers detected may contain some data points similar to the normal period. We then calculate the pairwise Manhattan distances between outliers and normal data points. If the smallest Manhattan distance between an outlier and a normal data point is less than the similarity threshold, we remove this outlier. After several experiments, we choose 5 as the similarity threshold.

We apply the scikit-learn implementation of random forest to build our supervised model. Specifically, we use 100 decision trees with no specified maximum depth. We observe that adding more decision trees does not improve the precision but consumes more CPU and memory resources.

**Alternative approaches.** To evaluate the efficacy of SHIL, we compare SHIL with several alternative real-time, light-weight security attack detection methods Lin et al. (2020); Tunde-Onadele et al. (2020) that have been proposed for container systems. We also implement pure-supervised or pure-unsupervised learning methods for evaluating the efficacy of our hybrid learning approaches.

- **Classified Distributed Learning (CDL) Lin et al. (2020):** We run CDL using 95 and 99 percentile anomaly detection thresholds. CDL uses the same autoencoder model as used in SHIL.

- **Self-patch Tunde-Onadele et al. (2020):** To make a fair comparison, we only compare our work with the attack detection part of the self-patch. Self-patch uses an autoencoder model with four hidden layers to detect attacks to containers. There are 256 neurons in the first and fourth hidden layers, and 128 neurons in the second and third hidden layers. Self-patch applies 99 percentile anomaly detection threshold only.

- **Supervised (RF):** We use a pure supervised random forest model. The hyperparameters used in the random forest model are the same as the random forest model used in SHIL.

- **Supervised (CNN):** We use the convolution neural network (CNN) learning method, which has been recently applied to cybersecurity Vinayakumar et al. (2017). We use Keras, with Tensorflow as the backend, to implement this model. This model consists of two
Table 3.3: Comparison with alternative approaches.

<table>
<thead>
<tr>
<th>Model</th>
<th>Detection rate</th>
<th>FPR</th>
<th>Lead time</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDL-95%</td>
<td>92.07%</td>
<td>6.57%</td>
<td>10.29s</td>
</tr>
<tr>
<td>CDL-99%</td>
<td>81.10%</td>
<td>1.40%</td>
<td>9.17s</td>
</tr>
<tr>
<td>Self-patch</td>
<td>45.73%</td>
<td>1.41%</td>
<td>7.99s</td>
</tr>
<tr>
<td>Supervised (RF)</td>
<td>77.44%</td>
<td>9.55%</td>
<td>10.24s</td>
</tr>
<tr>
<td>Supervised (CNN)</td>
<td>70.73%</td>
<td>4.63%</td>
<td>7.22s</td>
</tr>
<tr>
<td>SHIL 120% Boundary</td>
<td>89.02%</td>
<td>2.04%</td>
<td>9.51s</td>
</tr>
<tr>
<td>SHIL 200% Boundary</td>
<td>84.15%</td>
<td>0.85%</td>
<td>8.48s</td>
</tr>
</tbody>
</table>

1-D convolution layers with the rectified linear activation function (ReLU) and a kernel size of 5. The reason for choosing 1-D convolution layer is that the layer moves along one dimension, which makes it applicable for time-series data. The kernel size represents how many features are considered every time the kernel moves across a vector sample. The first CNN layer has eight filters while the second layer has four filters. The CNN then includes a flatten layer to flatten the output from the second convolution layer. Finally, the network has a dense layer of sigmoid activation function neurons to predict the probability of the input frequency being abnormal. Our CNN model is trained for 35 iterations using the Adam optimizer with a learning rate of 0.001. To train a more robust model, we shuffle the training set during training.

**Evaluation metrics.** We define $DC$ to be the number of containers that are under attack and correctly identified by the detector, and $MC$ to be the number of containers that are under attack and incorrectly missed by the detector. We use false positive (FP) to denote the number of measurement samples the detector falsely identifies, and true negative (TN) to be the number of samples the detector correctly rejects. Using the above definitions, the detection rate and false positive rate (FPR) are given by equations (3.3) and (3.4), respectively.

$$detection\ rate = \frac{DC}{DC + MC} \quad (3.3)$$

$$FPR = \frac{FP}{FP + TN} \quad (3.4)$$

We introduce *lead time* to denote the duration from the time the first alert is raised by the detection system to the time the attack is successful if no action is triggered before then.
3.3.2 Results Analysis

In this section, we present the analysis of the experiment results. Table 3.3 compares the detection results among CDL Lin et al. (2020) using 95 percentile (CDL-95%) and 99 percentile (CDL-99%) anomaly detection thresholds, Self-patch Tunde-Onadele et al. (2020), the pure supervised RF and CNN models, and our SHIL approach using 120% (SHIL-120%) and 200% boundary case threshold (SHIL-200%).

For CDL, increasing the anomaly detection threshold from 95 percentile to 99 percentile reduces the false positive rate from 6.57% to 1.40%. However, we also see a big drop in detection rate from 92.07% to 81.10%. Self-patch has a similar FPR as CDL-99%, but its detection
rate is only 45.73%. In contrast, SHIL-120% model can reduce false positive rate significantly by 69.0% while maintaining similar detection rate and lead time compared to pure unsupervised model CDL-95%. By increasing the boundary cases to a higher threshold, SHIL-200% can achieve a higher detection rate while reducing the false positive rate by 39.3%, compared to the pure unsupervised model CDL-99%. By having a self-supervised model to effectively filter the false positives, SHIL can adopt an unsupervised model with lower anomaly detection thresholds, thus achieving a higher detection rate and lower false positive rate at the same time.

The pure supervised models perform poorly which suffers from both low detection rate and high false alarms due to low quality labelled training data Johnson and Khoshgoftaar (2019). When comparing with the pure supervised RF and CNN models, SHIL-200% can reduce the false positive rate by 91.1% and 81.6%, respectively with higher detection rate and similar lead time.

Figure 3.4a shows the false positive rate of different models under different attack impact categories. The false positive rate of the pure supervised RF model is the highest in all categories except the “Crash the application” and “Escalate privilege level" categories. Furthermore, pure supervised models, and our SHIL models all achieve zero false positive rate in the “Crash the application” category. CDL-95% model has the highest false positive rate in “Crash the application” and “Escalate privilege level" categories and the second highest false positive rate in the remaining categories. The false positive rate of CDL-99% and Self-patch model is generally low, but its false positive rate in “Crash the application" category is 4.23% and 2.40% respectively. SHIL achieves consistently low false positive rate in all categories. Among all threat impact categories, the false positive rate of SHIL is below 1% in 31 attacks.

Figure 3.4b compares the detection rate of different schemes. SHIL achieves over 87.5% detection rate, except in “Disclose credential information” and “Crash the application" categories. In the “Execute arbitrary code" category, SHIL achieves perfect detection rate. The pure supervised CNN model can only detect 12.5% attack belonging to the “Crash the application” vulnerabilities while pure supervised RF model is only able to detect 25% of attacked containers. The reason is that the attacks in this category happen in such a short time that the pure supervised models cannot get enough training samples. However, the pure unsupervised model can detect some of the attacks in this category that show high reconstruction errors. SHIL uses the detection from the unsupervised model and does not always need to validate with the supervised model because the reconstruction errors from the detected samples are usually much higher than the boundary case. Thus, the detection rate of SHIL is still higher than those of the supervised models.

Next, we compare the lead time of different models shown in Figure 3.4c. Overall, the lead time is similar among all different models except pure supervised CNN model having a small
Table 3.4: System run time measurements of different learning methods. Each sample represents the frequency vector of all system calls produced within 100 ms.

<table>
<thead>
<tr>
<th>System Modules</th>
<th>Execution Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDL training</td>
<td>7.75 ± 0.40 (5000 samples)</td>
</tr>
<tr>
<td>Self-patch training</td>
<td>8.20 ± 0.04 (5000 samples)</td>
</tr>
<tr>
<td>SHIL training</td>
<td>8.42 ± 0.42 (5000 samples)</td>
</tr>
<tr>
<td>CDL detection</td>
<td>7.30 ± 0.10 per sample</td>
</tr>
<tr>
<td>Self-patch detection</td>
<td>0.0001 ± 0.00 per sample</td>
</tr>
<tr>
<td>SHIL detection</td>
<td>7.65 ± 0.11 per sample</td>
</tr>
</tbody>
</table>

lead time in the “Return a shell and execute arbitrary code” category. All six models achieve a large lead time in the “Return a shell and execute arbitrary code” category. CDL-95% model achieves the largest lead time in the “Return a shell and execute arbitrary code”, “Disclose credential information”, and “Escalate privilege level” categories, while the supervised random forest achieves the largest lead time in the remaining categories. It is not surprising to see no lead time in the “Crash the application” category by all the models as the attacks suddenly terminate the applications.

**System run time measurements.** Table 3.4 compares the training and testing time of CDL, Self-patch, and SHIL. SHIL takes slightly more time during training and testing due to more components in the system. However, considering each sample represents the frequency vector of all system calls produced within 100 ms, SHIL is light-weight and practical for real time security attack detection in large-scale container-based environments.

### 3.3.3 Case Study

In this subsection, we analyze one representative attack from the top three categories to understand how SHIL identifies the attack with lower false positive rate compared with the unsupervised model.

**Return a shell and execute arbitrary code.** CVE-2017-12615 occurs when the attacker uploads a malicious JSP file which contains the attack code to Apache Tomcat. Tomcat prohibits users from uploading and executing files with the suffix “.jsp” to prevent malicious operations. However, the attacker can bypass the rule by uploading a JSP file with the suffix “.jsp ” containing a trailing space. After that, Tomcat re-formats the file name by removing any trailing spaces and then executes it. Compared with the unsupervised model, SHIL improves the false positive rate by 88.05% without decreasing the detection rate. The 103 false positives removed by SHIL across four containers exhibit similar patterns. The false positive samples contain the `stat` call, which has a higher frequency compared with other normal run samples. The unsupervised model identifies them as anomalies. We observe that `stat` calls occur periodically and are
generated by the workload as the normal period exhibits a similar periodical pattern. SHIL's self-supervised model can identify the periodical pattern and filter out those false positives.

**Execute arbitrary code.** CVE-2021-44228 is an Apache Log4j vulnerability that allows an attacker to execute arbitrary expressions input via the Java naming and directory interface (JNDI) service. The attacker can send a log message with special syntax to perform a JNDI lookup of a malicious lightweight directory access protocol (LDAP) server resource. The vulnerable application using Log4j will parse the message, connect to the attacker's server, and execute the payload it receives. Compared with the unsupervised model, SHIL reduces false positive rate by 87% with no reduction in detection rate. We observe the false positive samples exhibit periodical patterns, which are generated by the dynamic periodical workload and have similar patterns to the normal execution. SHIL successfully filters out those false alarms. After the attack is triggered, we observe increasing occurrences of several system calls including `clone`, `connect`, `execve`, `fcntl` and `mmap`.

**Disclose credential information.** CVE-2018-15473 is an Open-SSH vulnerability that allows the attacker to steal credential information because of no limit on the maximum attempts of inputting user names and passwords. The attacker finds a valid username using a brute-force method and then cracks the passwords in a similar way.

SHIL filters out 83 false positives, reducing FPR by 92.21% without hurting the detection rate. Since the sampling interval is small, we observe certain system calls (accept, stat, close, `fstat`, `read` and `mmap`) do not have average distributions across normal run samples, causing further false positives.

### 3.4 Summary

In this chapter, we have presented SHIL, a new self-supervised hybrid learning system for more efficiently detecting security attacks in container-based computing environments. SHIL identifies anomaly detection boundary cases as most likely false alarms and combines unsupervised and supervised machine learning methods to filter out majority of the false alarms without missing most of the true attacks. For practical deployment of supervised learning models, SHIL adopts a self-supervised learning approach to labelling training data automatically using outlier detection over a window of recent measurement samples when attack alerts are first raised. Our experimental results with real world security attacks including the recent high-impacting Log4j attack show that SHIL can significantly reduce false alarms by up to 91% while maintaining similar detection rates compared to existing pure-supervised or pure-unsupervised methods. SHIL is light-weight and does not require manual data labelling, which makes it practical for security attack detection in container-based production environments.
4.1 Introduction

Microservice architectures have become a popular method for deploying applications in production cloud environments. Microservices divide applications into modular services that are responsible for different functionality and can efficiently scale to user demands. However, as microservice applications develop and become increasingly complex, debugging distributed performance bugs becomes a key challenge Balalaie et al. (2016); Jamshidi et al. (2018). The increased complexity makes it more difficult to understand the dependencies among different system components and pinpoint the root cause of a performance bug manifested in the production environment. For example, Microsoft suffered a recent outage in its microservices-supported 365 online web applications due to the performance issues in its caching infrastructure Kharpal (2023); Microsoft (2022). Twitter also recently experienced microservice issues that caused the platform to turn off its SMS two-factor authentication Brown (2022). An automatic root cause analysis tool can help developers save time and resource costs of manual debugging.
class ReviewStorageHandler : public ReviewStorageServiceIf
{
    void ReviewStorageHandler::StoreReview(
        int64_t req_id,
        const Review &review,
        const std::map<std::string, std::string> &carrier)
    {
        std::istream in = ...;
        long len = ...;
        while (len > 0) {
            long ret = in.ignore(len); // in is corrupted
            if (ret < 0) { // ret = 0
                throw std::runtime_error("IOException");
            }
            len -= ret;
        }
    }

Figure 4.1: Due to data corruption, an issue arises within the I/O function. When `istream.ignore` returns 0 and 0 is used as the stride, this sequence of events triggers a situation where the ReviewStorage service becomes unresponsive and begins to hang.

By knowing the root cause function, software developers can directly jump into that function and fix the bug in much shorter time.

Much work (e.g., Mace et al. (2015); Kobayashi et al. (2017); He et al. (2022)) has been done to debug performance problems in distributed systems. However, highly dynamic microservice applications bring new challenges to this notoriously difficult problem. First, microservices often share a common physical host infrastructure for saving resources. As a result, many dependent or independent application pods/containers are co-located together, which makes the root cause analysis particularly challenging. Previous work has proposed various correlation methods (e.g., Pearson Pearson (1920)) to localize problematic microservice component Wu et al. (2020); Ma et al. (2020); Lin et al. (2018a); Meng et al. (2020). However, those correlation approach often suffer from high false positives, which sometimes incur more debugging work to the developer. Furthermore, previous work often cannot localize to the buggy function level, which still leaves most debugging work to the developer. Previous work also proposed to leverage Granger causality methods He et al. (2022) to pinpoint root cause functions. However, they often do not consider the dynamic dependency relationships among different service components.
4.1.1 Motivating Example

We demonstrate the performance bug challenges with a data corruption bug injected into the Media Microservice application Gan et al. (2019). Normally, the user uses the Nginx web server to compose movie reviews, which calls services such as the compose review service. The compose review service collects some information (e.g. user ID, movie ID, review content, etc) from the Nginx web server and then stores the review in the backend via the review storage service before returning. However, after triggering a data corruption bug shown in Figure 4.1 in the review storage service, the user finds that the application no longer responds.

Moreover, since execution hangs, the Kubernetes logs do not provide any useful information and may lose log data after some time when containers restart. The span messages from the application performance management (APM) tool Authors (2023d,c) also only report a container restart alert which can occur frequently in microservice environments. So developers do not have any useful clues for pinpointing this particular data corruption hang bug that is triggered in the Media microservice application. The developer would also observe the hanging symptom in multiple services such as the compose review service besides the Nginx web server when the response time change triggers a service level objective (SLO) violation. Thus, the developer needs an effective debugging tool to help pinpoint the root cause buggy function in the review storage service.

4.1.2 Contribution

In this chapter, we present FCA, a new dependency-aware full stack causal analysis framework, which leverages both infrastructure metrics (e.g., CPU usage, memory usage, network traffic) and application performance traces to achieve fine-grained performance debugging in distributed microservice applications. When a performance bug (e.g., SLO violation, software hang) is detected, FCA aims at pinpointing the root cause service and the buggy function among a large number of service and function candidates in a few minutes.

To achieve this goal, our approach consists of two steps to drill down the buggy function. First, we perform anomaly detection over both infrastructure metrics and application performance traces and employ the causal inference across those detected anomalies to narrow down the root cause service. Since microservice systems often consist of many dependent or in-dependent services, we incorporate the service dependency in our causal analysis to surface the root cause service more effectively. After localizing the faulty service, our second step is to analyze the function execution trace of the faulty service to localize the buggy function using a similar causal analysis process as the first step.

The key element that decides the effectiveness of FCA is a robust causal inference method.
Previous approach has been focusing on linear correlation methods such as Pearson (1920), which however cannot properly handle non-linear correlations that are common in machine data. We employ an information theory based causal analysis to cross-examine the execution time and resource consumption of the faulty service. Specifically, we use mutual information (MI) (Lizier 2014), which observes Granger causality, to evaluate the causal probability cross monitoring data streams. However, the MI scheme is sensitive to data noises and time lags. To better handle noisy machine data, we introduce anomaly detection and anomaly alignment to achieve robust cross-stack causal inference. Specifically, this paper makes the following major contributions:

• We introduce a new dependency-aware full-stack causal analysis system to achieve fine-grained performance debugging to the buggy function level.

• We present a robust cross-stack causal inference algorithm to surface the root cause service and function with high accuracy.

• We have implemented a prototype of FCA and evaluated using a set of real world performance bugs injected in a set of microservice benchmark applications. Our results show that FCA can accurately pinpoint the root cause service and function in all buggy runs with much higher accuracy than other alternative solutions.

The rest of the chapter is structured as follows. Section 4.2 presents the system design in detail. Section 4.3 describes our experimental evaluation methodology and our experimental results. Section 4.4 concludes this chapter.

4.2 System Design

In this section, we present the design of our system. We first provide a system overview. Next, we describe each component in detail.
4.2.1 System Overview

FCA is a dependency-aware full stack causal analysis framework, which can automatically pinpoint root cause service and function for microservice applications, illustrated by Figure 4.2. FCA consists of three key components: 1) anomaly detection over infrastructure metric data and application performance trace data, 2) service dependency graph extraction from application trace data, and 3) cross-stack causal analysis for pinpointing root cause services and functions.

Upon detecting a performance problem (e.g., SLO violations, software hang), FCA performs analysis over a recent time window of infrastructure metric data (e.g., CPU usage, memory usage, network traffic) and application performance trace data such as distributed request context data also known as span. We obtain metric data via API calls to open-source monitoring tools Prometheus Authors (2023b) and NodeExporter Prometheus (2023). The system metrics data are also accompanied by useful name-space information which can be employed for the root cause localization. Meanwhile, we get request context data with the open-source tools Jaeger Authors (2023c) and OpenTelemetry Authors (2023d) in the form of span units that encapsulate the operations performed as user requests are processed through the application. The span offers helpful information including function calls and timing details.

The anomaly detection module leverages machine learning anomaly detection methods to identify focus points in the aforementioned input signals that will improve the effectiveness of the causal analysis. Accordingly, the module transforms each of the time series data to a piece-wise function, where each data-point holds one value of the binary anomaly detection model decision.

The dependency graph extraction acquires dependencies among the microservice nodes to inform the causal analysis. This component traverses the dependency graph constructed by Jaeger based on the spans collected. Specifically, we extract information such as node parent-child relationships and their call counts from the underlying graph to help localize the root cause.

The causal analysis component performs statistical analysis of the system metric and span data to find the root cause service and function. We use the information theory approach of mutual information (MI) Lizier (2014) to understand the dependence relationships among the processed CPU utilization across the cluster nodes and the span of the faulty node. First, we leverage the namespace and microservice dependency information to select a set of relevant service nodes as candidates for further analysis. Second, we compute the MI scores using the aligned data of the candidates and rank them with an algorithm to output the highest ranking node. Third, we inspect the operations of the root cause service node to pinpoint the root cause function. We repeat the score computation and ranking steps, comparing the span data of the operations with that of the faulty node.
4.2.2 Microservice Data Collection

**Infrastructure Metrics Collection.** We periodically (i.e. every 30 seconds) record different types of system metrics, including CPU utilization percentage, active memory usage, network received, network transmitted, disk read, disk write, and restart count of each pod running in the Kubernetes cluster. We adopt non-overlapping moving windows when aggregating the system metrics into time series samples.

**Request Context Data Collection.** We record the response time history every second, along with the failures and exceptions of the responses. Using the same windows as the system metrics, we also store the spans collected from Jaeger, which include the timing information (start time and duration), operation name, etc.

A span is a fundamental component that signifies a discrete unit of work or operation within a system. A trace consists of multiple spans from different microservices. By querying Jaeger with a certain start and end time, it returns all the traces that overlap this timing window. We then preprocess the returned traces by examining all the spans inside of each trace. By checking the operation names and service name, we ignore those spans for health checking because they are typically short and not related to the current requests from the users. For each service, we only focus on the spans belonging to that service, while ignoring the spans from other services. We also ignore those spans with start time later than the current window end time, and spans with end time earlier than the current window start time. For example, in Figure 4.3, we use span B and span C only when calculating the average span duration for the window. For both input metric and span signals, we apply a moving window to smooth the time series of both metric and span data. In addition, we extract the function trace, which Jaeger outputs based on the spans.

Recall that each trace contains multiple spans from different services. For each service, we only focus on the spans belonging to that service, while ignoring the spans from other services.

Figure 4.3: We consider the span end time to determine the average span duration for each window.
By checking the operation names and service name, we ignore those spans for health checking because they are typically short and not related to the current requests from the users. For accurate correlation analysis, we use the same timing window as the system metrics when processing the average span duration. The spans will not be finished until all the services in the call chain are finished. Figure 4.4a shows an example average span for hang bugs. We can see the decreasing trend at time 10. This is because the spans of the checkout service do not finish until the root cause service (email service) is restarted by Kubernetes. Thus the earlier the span starts, the longer the span becomes. This is the reason why we introduce the anomaly detection to accommodate the limitation of the span collection of hang bugs.

### 4.2.3 Anomaly Detection

Microservice environments deal with the challenge of rapidly changing workloads. As such, the system metrics and spans are also dynamic time series data. However, the fluctuations over an extended period of time is a problem for our causal analysis computations. We observe that the noise can lead to misleading MI calculation results.

The Figure 4.4 shows an example CPU usage and span duration excerpt during an infinite loop performance bug exploit period. The subfigure Figure 4.4b shows the MI scores using the raw data while the subfigure Figure 4.4c shows that after performing anomaly detection preprocessing. Before, the score of the productcatalog service is close to that of the root cause emailservice despite the productcatalog series not influencing the span duration symptom. Whereas, after preprocessing, the values are further apart and emailservice is more clearly the root cause. The anomaly detection emphasizes the data during the narrower window of faulty activity.

We employ an unsupervised percentile approach for anomaly detection. In the recent time series window, we observe the data-points with values above a high percentile threshold and those with values below a low percentile threshold. We label these outlier points as abnormal data points and the others as normal. The thresholds can be set by experts depending on the history and the metric type. Specific implementation details of the anomaly detection model(s) are discussed in section 4.3.

### 4.2.4 Anomaly Alignment for Causal Analysis

We observe that time shifts between the faulty node and candidate node signals largely affect the resulting MI values, which leads to inaccurate ranking. We employ an *anomaly alignment algorithm* to address this issue.
(a) The symptom service (checkout service) span duration time series.

(b) Without anomaly detection and alignment, the email service (root cause) score is 0.55 while the productcatalog service score is 0.42.

(c) With anomaly detection and alignment, the email service (root cause) score is 1.00 while the productcatalog service score is 0.015.

Figure 4.4: The causality score improves with anomaly detection preprocessing as observed during an infinite loop bug example in the Online Boutique application.

We align the faulty node and the candidate node signals around the start of anomalous activity trim them to their overlapping duration to achieve more meaningful MI scores and improve the root cause ranking. The algorithm finally returns the signals to compute the causal score between the faulty node and the candidate node signals.

After obtaining the anomaly detection preprocessed time series, the algorithm operates as
Figure 4.5: An example of the anomaly alignment algorithm. We require a minimum number of consecutive anomalous samples to avoid misalignment due to noise.

follows.

1. Find the first set of continuous anomalies greater than the minimum required number of consecutive anomalies.
2. If no continuous anomalies are found, stop the alignment.
3. Data preprocessing:
   (a) Add the normal window size of data points (e.g. ten) before the first anomaly.
   (b) Add the abnormal window size of data points (e.g. five) after the first anomaly.
4. Align the first anomaly between the two time series.
5. Ensure that the data overlaps with equal length:
   (a) Compare the number of data points before the first anomaly, remove the extra points at the beginning of the longer data.
   (b) Compare the number of data points after the first anomaly, remove the extra points at the end of the longer data.
6. Return the aligned data of both time series.

Figure 4.5 is an example to demonstrate our anomaly alignment algorithm. "1" means the sample is an anomaly, while "0" means it is not an anomaly. We can see from the figure above that the anomalies from the original time series A and B do not align. There is one anomaly at the index 0 of the time series A. Similarly, at index 2 of the time series B, there is an anomaly. However, we do not align these two anomalies since they can result from random noise. In addition, aligning by these indices would misalign the true anomalies later in the series. We
Figure 4.6: An example of service dependency graph extracted from Jaeger. The direction of each edge is from the caller to the callee, with the number of call count on the edge.

set a minimum required number of consecutive anomalies to be three in this case. After we detect at least three consecutive anomalies in both time series, we align the first anomaly of that consecutive anomalies, namely the anomaly at index 5 of time series A with the anomaly at index 6 of time series B. We then add the same number of data points between the series before and after the first anomaly separately.

We considered approaches like *dynamic time warping* (DTW) Bellman and Kalaba (1959) to address the alignment issues but it resulted in some problems. It adds data values around the shifted signal, which can actually distort the original time series and yield erroneous mutual information results.

### 4.2.5 Service Dependency Graph Extraction

It is important to know the dependency relationships among the microservices currently running in the system as it helps us understand existing causal relationships. We use Jaeger to extract the service dependency graph. Figure 4.6 depicts an example service dependency graph. The graph consists of several directed edges from caller to callee. The number at each
edge represents the call count. Jaeger constructs the graph in five broad steps (Jaeger, 2023). Overall, it reads spans within a time-period, groups spans by trace ID, finds parent-child relationships in the span references, compiles the edge counts, and stores the output in a database.

We extract this constructed service dependency graph and save it as a local JSON file with Jaeger’s internal API. During causal analysis, we load the graph from the file and run search algorithms to retrieve information about the relevant nodes.

### 4.2.6 Causal Analysis

With access to the pre-processed system metrics and spans as well as the service dependency graph, we apply the causal analysis to discover the root cause service and function of the fault.

Our goal is to understand the relationship between time series data so we consider causal analysis methods that can measure how one time series causes another to happen. We investigated similarity. However, similarity measures describe how similar two datasets are. In our case, our focus is not analyzing how similar two datasets are but to understand the relationship between them.

In this chapter, we make two assumptions. First, we assume that there exists correlation between the root cause service/function and SLO violations. This is because the root cause service and function will impact the functionality of the dependent services and eventually lead to the SLO violations. Second, we assume that benign services and functions that exhibit anomalies are not dependent on the symptomatic service. Benign services and functions can exhibit anomalies due to various reasons, such as hardware failures or network issues. However, if the services or functions are benign, they will not negatively impact the symptom service, thus they have no dependency relationship with the symptom service.

**Root Cause Service Analysis**

First, we find the root cause service that contributes to the fault.

For our system, we use the response time change point of the fault node as the performance anomaly alert. Nevertheless, note that the focus of our work is not this initial anomaly detection. When the anomaly alert occurs, we obtain a recent window of the metric and span input data. The window includes a *normal* period before the alert and an *abnormal* window afterwards. The goal is to select a large enough period to observe the differences between the normal and abnormal data periods. The abnormal period also needs sufficient data to encompass the performance bug symptom impact. Moreover, we allow enough data samples for our
preprocessing methods, that is, the anomaly detection model. For our experiments, we collect 24 samples before the alert and four after the alert. We describe the details in section 4.3.

Thereafter, we begin the following overall causal analysis service procedure.

For a given fault node A:

1. With the dependency graph, consider the child nodes of node A, descendants of node A, and node A itself as the set of candidates.
2. Calculate the pairwise correlation of each metric of each candidate with the average span duration of node A.
3. If there is a tie, rank the nodes that have no child (leaf nodes) first.
4. If there is still a tie, rank the nodes by the descending order of calling count (the number on the edges) that is closest to node A.
5. If there is still a tie, rank the node first by the earliest change point time.

We begin by querying the service dependency graph for candidate nodes. Starting from the faulty node, we add child nodes at each generation to the set of candidates until we reach a maximum depth. Next, we compute the causality score for each faulty node and candidate node pair so that we can rank the candidates. Each pair consists of the average span duration of the faulty node and the CPU utilization of a candidates. Recall that we employ an anomaly alignment algorithm, described in subsection 4.2.3, to align the signal pair to compare.

**Mutual Information:** We apply Granger causality by using mutual information (MI) to identify the root cause. For variables \( X_1 \) and \( X_2 \), if previous values of \( X_1 \) give information to predict future values of \( X_2 \) than just previous values of \( X_2 \), then \( X_1 \) "Granger-causes" \( X_2 \). MI calculates how much information a variable provides about another. For two random variables \( X_1 \) and \( X_2 \), the MI formula is outlined in Equation 4.1.

\[
MI(X_1; X_2) = \sum_{x_1 \in X_1} \sum_{x_2 \in X_2} p(x_1, x_2) \log \frac{p(x_1, x_2)}{p(x_1)p(x_2)}
\]

where \( p(x_1) \) and \( p(x_2) \) are the marginal probability mass functions of \( X_1 \) and \( X_2 \), respectively, and \( p(x, y) \) is the joint probability mass function of \( X \) and \( Y \).

We considered the cointegration method Engle and Granger (1987). However, the approach has stationary requirements that is not realistic for real-time microservice monitoring metrics.

We also considered the cross-correlation Bracewell and Kahn (1966). Cross-correlation is typically used for searching for a known feature in a long time series data. However, in our case, the feature is unknown and we need to compare the full length of all time series data. Thus cross-correlation is not applicable in our scenario.
The root cause service (emailService) CPU usage metric data.

The root cause function (SendOrderConfirmation) execution time data.

Figure 4.7: A comparison of the root cause service CPU metric data with the root cause function execution time for the infinite loop bug injected into the Online Boutique application.

**Tie Breaking:** In some cases where there are multiple candidates sharing the same correlation coefficient value, we introduce the tie breaking algorithm to rank them using the information from the service dependency graph that we collect earlier. We break ties in order of leaf node status, call count, and earliest change point time heuristics. A leaf service node is dependent on no other service so it is likely the cause compared to another node that depends on another service. In addition, the higher the call count to a node, the more likely it is to cause an issue as there are more instances of dependence other nodes have to it. Finally, we consider the node with the earliest change point time in the system metric (i.e. CPU utilization), based on the expectation that the root cause change occurs first.

**Root Cause Function Analysis**

After determining the root cause microservice, we further investigate the root cause function within it.

We analyze which function contributes to the system metric issue of the root cause service node now that we know the root cause service node metric contributes to the span duration
symptoms of the faulty service. We essentially repeat the causal analysis steps but compare the execution time of the functions within the root cause service to the CPU utilization of root cause service node. The execution time series data is extracted from the span data.

For example, we inject the infinite loop shown in Figure 4.1 to the EmailService node of the Online Boutique application (discussed in subsection 4.3.1). The example Figure 4.7 captures the CPU utilization of the EmailService and the execution time of the rootcause SendOrderConfirmation function during an infinite loop test. Note that during the increase in CPU utilization of the EmailService, the execution time of the SendOrderConfirmation function drops to zero. When the execution is stuck in the infinite loop, the function is unable to communicate its span so no execution is collected during the time period.

In the root cause function analysis, we break ties by prioritizing the function with the earliest change point in execution time. We no longer use the service dependency graph since we now deduce the root cause service location. The overall algorithm is summarized below.

For a given root cause service node A:

1. Consider the recently executing functions within the root cause service node A as the set of candidates.
2. Calculate the pairwise correlation of the execution time series of each candidate with the system metric of node A.
3. If there is a tie, rank the node first by the earliest change point time.

4.3 Experimental Evaluation

In this section, we first describe our evaluation methodology. Next, we compare our system with a set of alternative schemes. We implement a prototype of our system and evaluate it on a desktop with 16 3.6 GHz cores and 32 GB memory running Ubuntu 20.04 64-bit.

4.3.1 Evaluation Methodology

**Evaluation microservice applications** We evaluate our system on three microservice applications.

- **Online Boutique**: Google Cloud developed Online Boutique as a microservice demo application. It includes 10 microservices, such as frontend, chart, checkout, shipping, etc. to simulate an e-commerce website that allows people to browse, add items to their cart, and checkout the items from their cart. On every listing page, it will recommend several items related to the current item for the users to shop next.
• **Social Network**: The SAIL group at Cornell University developed Social Network Gan et al. (2019) as a microservice based broadcast-style social network, which consists of 27 unique microservices. It supports multiple actions for the users such as following, unfollowing, and blocking other users. After users create posts with texts, images, and links, the posts will be broadcasted to all of their followers.

• **Media Service**: The SAIL group at Cornell University also developed Media Service Gan et al. (2019) with 33 unique microservices. This allows users to browse movie information online, as well as rent, review, rate, and watch movies.

**Experiment Setup** We use OpenTelemetry Authors (2023d) to analyze the span duration of microservices. We use Jaeger Authors (2023c) to manage the traces, and then use the internal API to retrieve the service dependency graph. We employ Prometheus Authors (2023b) to store and query the system metrics of pods, including CPU utilization ratio, active memory usage, network traffic transmitted in bytes, network traffic received in bytes, disk write in bytes, and disk read in bytes, using non-overlapping moving window of 30 seconds. Please note that the system metrics may not exhibit change immediately after the fault. In some cases, for example, the changes in CPU utilization ratio needs time to accumulate enough CPU usage seconds.

We use Locust Authors (2023a) to deliver random workload using 10 simulated workers, which share the same list of tasks for simulating real usage of the microservice application, such as visiting the listing pages, adding items to the shopping cart, checking out the items in the shopping cart, etc. After finishing a certain task, each worker has a random wait time before performing the next task. At each second, Locust records the average response time of the communication from the workers. Upon detecting the change point of the response time, we continue to collect the system metrics for another two minutes, and we query Jaeger and save all the traces in the window of from twelve minutes before the change point to two minutes after the change point for later average span duration calculation.

For each performance bug, we repeat the experiment for three times and report the average and standard deviation.

### 4.3.2 Alternative Methods

**Alternative correlation analysis** We evaluated our system with different alternative correlation analysis tools.

• **Pearson correlation coefficient** Pearson (1920): The Pearson correlation coefficient is a measure of the linear relationship between two continuous variables. It ranges from -1 (perfect negative correlation) to 1 (perfect positive correlation), with 0 indicating no
Symptom

Timeout

MediaMicroservices

UploadMovieReview in the movie review service does not release the allocated memory.

Data corruption

SocialNetwork

The corrupted data stream affects the loop stride, causing the system to hang.

Timeout

The post storage service encounters an infinite loop when storing new posts.

Timeout

GetFollowers in the social graph service does not release the allocated memory.

Data corruption

SocialNetwork

The corrupted data stream returns an error code, causing a system slowdown.

Timeout

GetFollowers in the social graph service does not release the allocated memory.

Timeout

Infinite loop

The corrupted data stream affects the loop stride, causing the system to hang.

Timeout

The post storage service encounters an infinite loop when storing new posts.

Data corruption

SocialNetwork

The corrupted data stream affects the loop stride, causing the system to hang.

Timeout

The post storage service encounters an infinite loop when storing new posts.

Data corruption

OnlineBoutique

Memory leak

Timeout is missing in the user mention service, leading to system hanging.

Timeout

The payment service when charging the credit card.

Data corruption

SocialNetwork

The corrupted data stream affects the loop stride, causing the system to hang.

SocialNetwork

Timeout

The post storage service encounters an infinite loop when storing new posts.

UploadUserReview in the social graph service does not release the allocated memory.

Timeout

The post storage service encounters an infinite loop when storing new posts.

Timeout

The email service encounters an infinite loop when sending the confirmation email.

Timeout

MediaMicroservices

UploadMovieReview in the movie review service does not release the allocated memory.

Data corruption

SocialNetwork

The corrupted data stream returns an error code, causing a system slowdown.

SocialNetwork

Timeout

The payment service when charging the credit card.

Table 4.1: Summary of bugs that we use in our experiment.

<table>
<thead>
<tr>
<th>Bug ID</th>
<th>Symptom</th>
<th>Microservice</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Infinite loop</td>
<td>OnlineBoutique</td>
<td>The email service encounters an infinite loop when sending the confirmation email.</td>
</tr>
<tr>
<td>2</td>
<td>Memory leak</td>
<td>SocialNetwork</td>
<td>GetFollowers in the social graph service does not release the allocated memory.</td>
</tr>
<tr>
<td>3</td>
<td>Memory leak</td>
<td>MediaMicroservices</td>
<td>UploadMovieReview in the movie review service does not release the allocated memory.</td>
</tr>
<tr>
<td>4</td>
<td>Infinite loop</td>
<td>MediaMicroservices</td>
<td>The user review service fails in an infinite loop when uploading new user reviews.</td>
</tr>
<tr>
<td>5</td>
<td>Infinite loop</td>
<td>SocialNetwork</td>
<td>The post storage service encounters an infinite loop when storing new posts.</td>
</tr>
<tr>
<td>6</td>
<td>Data corruption</td>
<td>SocialNetwork</td>
<td>The corrupted data stream returns an error code, causing a system slowdown.</td>
</tr>
<tr>
<td>7</td>
<td>Data corruption</td>
<td>MediaMicroservices</td>
<td>The corrupted data stream affects the loop stride, causing the system to hang.</td>
</tr>
<tr>
<td>8</td>
<td>Timeout</td>
<td>SocialNetwork</td>
<td>Timeout is missing in the user mention service, leading to system hanging.</td>
</tr>
<tr>
<td>9</td>
<td>Timeout</td>
<td>MediaMicroservices</td>
<td>Timeout is set incorrectly in the rate system, causing system hanging.</td>
</tr>
<tr>
<td>10</td>
<td>Timeout</td>
<td>OnlineBoutique</td>
<td>Timeout is missing in the payment service when charging the credit card.</td>
</tr>
</tbody>
</table>

Table 4.2: Summary of root cause service results.

<table>
<thead>
<tr>
<th>Bug ID</th>
<th>Number of Candidate Services</th>
<th>Root Cause Service</th>
<th>MI</th>
<th>Pearson</th>
<th>Spearman</th>
<th>Kendall</th>
<th>Microscope</th>
<th>MI+AD+Align</th>
<th>MI+AD+Align +Namespace</th>
<th>FCA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Rank FP</td>
<td>Rank FP</td>
<td>Rank FP</td>
<td>Rank FP</td>
<td>Rank FP</td>
<td>Rank FP</td>
<td>Rank FP</td>
</tr>
<tr>
<td>1</td>
<td>47</td>
<td>Email service</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>60</td>
<td>Social graph service</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>66</td>
<td>User review service</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>64</td>
<td>Post storage service</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>64</td>
<td>User mention service</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>66</td>
<td>Billing service</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>66</td>
<td>Payment service</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.3: Summary of root cause function results. “X” means failure to detect the root cause function.

<table>
<thead>
<tr>
<th>Bug ID</th>
<th>Number of Candidate Functions</th>
<th>Root Cause Function</th>
<th>MI</th>
<th>Pearson</th>
<th>Spearman</th>
<th>Kendall</th>
<th>Microscope</th>
<th>MI+AD+Align</th>
<th>MI+AD+Align +Namespace</th>
<th>FCA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Rank FP</td>
<td>Rank FP</td>
<td>Rank FP</td>
<td>Rank FP</td>
<td>Rank FP</td>
<td>Rank FP</td>
<td>Rank FP</td>
</tr>
<tr>
<td>1</td>
<td>75</td>
<td>SendOrderConfirmation</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>641</td>
<td>GetFollowers</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>3</td>
<td>644</td>
<td>UploadMovieReview</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>4</td>
<td>66</td>
<td>WritePost</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>5</td>
<td>61</td>
<td>ComposeUserMentions</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>6</td>
<td>61</td>
<td>ComposeUserMentions</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>7</td>
<td>64</td>
<td>StoreReview</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>8</td>
<td>64</td>
<td>ComposeUserMentions</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>9</td>
<td>64</td>
<td>UploadRating</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>10</td>
<td>75</td>
<td>Change</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

The formula for the Pearson correlation coefficient (r) is: 

\[ r = \frac{\sum(x - \bar{x})(y - \bar{y})}{\sqrt{\sum(x - \bar{x})^2 \sum(y - \bar{y})^2}} \]

, where \( x \) and \( y \) are individual data points, \( \bar{x} \) and \( \bar{y} \) are the means of the respective variables, and the summation is over all data points.

• **Spearman's rank correlation coefficient**

Spearman (1961): Spearman's rank correlation coefficient is a non-parametric measure of the strength and direction of association between two ranked variables. It is based on the ranks of the data rather than the actual values. The formula for Spearman's rank correlation coefficient \( \rho \) is: 

\[ \rho = 1 - \frac{6 \sum d_i^2}{n(n^2-1)} \]

, where \( d_i \) is the difference between the ranks of corresponding values in the two variables, and \( n \) is the number of observations.

• **Kendall rank correlation coefficient**

Abdi (2007): Kendall rank correlation coefficient, also known as Kendall's tau (\( \tau \)), is a non-parametric measure of the strength and direction
of association between two ranked variables. It is based on the number of concordant and discordant pairs in the data. The formula for Kendall’s tau is: $\tau = \frac{(n_c - n_d)}{\sqrt{(n_0 - n_1)(n_0 - n_2)}}$, where $n_c$ is the number of concordant pairs, $n_d$ is the number of discordant pairs, $n_0 = n(n - 1)/2$, $n_1$ is the number of pairs tied for the first variable, and $n_2$ is the number of pairs tied for the second variable.

**Alternative algorithms** We evaluated our system with three different alternative algorithms: Microscope Lin et al. (2018a), MI + AD + Align, and MI + AD + Align + Namespace to evaluate the necessity of our system design.

- **Microscope Lin et al. (2018a):** Microscope is designed to detect the root cause service by using Pearson and service dependency information. Please note that in the original paper, Microscope cannot detect the root cause function. However, we still extend its idea of using Pearson and service dependency information to detect the root cause function for fair comparison.

- **MI + AD + Align:** On top of mutual information (MI), we add the anomaly detection
and anomaly alignment, and it becomes MI + AD + Align. It does not have namespace or service dependency information of the microservice application to be evaluated. It considers all microservice running in the same system to be equal.

- **MI + AD + Align + Namespace**: On top of MI + AD + Align, we further include the namespace information and it becomes MI + AD + Align + Namespace. It ranks the microservices running in the same namespace as the faulty microservice higher than the other microservices running in other namespaces. It does not include any information about the service dependency.

**Evaluation metrics** In our evaluation, *Rank* refers to the rank of the actual root cause service or function given by the analysis method in question. We also define the number of false positives (FP) as the services/functions ranked above or tied with the actual root cause service/function.

### 4.3.3 Result Analysis

In this section, we analyze the results of the experiment. We first analyze the experiment result of the infinite loop in the Online Boutique application, then we analyze the experiment result of all ten performance bugs.

**Online Boutique application**

First we examine the results of the online boutique application. For one of the background microservices, the recommendation service, we induce a CPU hog at the same time as the root cause injection. This presents a case where another service exhibits coinciding symptoms but is not actually the underlying performance issue. Thus, we expect an effective system to identify the recommendation service as problematic but not as the root cause (the email service).

**Root Cause Service Analysis** Figure 4.8 depicts the correlation values under eight different algorithms: MI, Pearson, Spearman, Kendall, Microscope, MI + anomaly detection + alignment (MI + AD + Align), MI + anomaly detection + alignment + namespace information (MI + AD + Align + Namespace), and our FCA algorithm. We pick four representative services in the bar chart: the email service (the root cause service), the recommendation service (the service with CPU hog), kube-proxy (a Kubernetes component), and the checkout service (a service in the Online Boutique application).

The root cause service, email service, ranks the first for four different algorithms: MI, MI + AD + Align, MI + AD + Align + Namespace, and FCA. The results show that those five algorithms can successfully identify the root cause service, while the rest four algorithms,
Pearson, Spearman, Kendall, and Microscope fail to do so. Nevertheless, only FCA shows the large difference of correlation value between the email service and the second rank service, which indicates strong confidence in the root cause service.

Three algorithms, MI, MI + AD + Align, MI + AD + Align + Namespace, ranks the recommendation service as the second, while Pearson ranks the same service as the first, which meets our expectation of the high ranking of the recommendation service due to the CPU hog symptom. However, because the recommendation service is not ranked as one of the priority services according to subsection 4.2.6, it is reset to 0 in Microscope and FCA.

Kube-proxy, a Kubernetes component, is a network proxy running in each node. As there are a lot of network traffic during the experiment, we expect this service to be active throughout the time. Both Spearman and Kendall rank it as the first. We can also observe the high standard deviation with the MI + AD + Align. However, because this service is not in the same namespace as the Online Boutique, it is reset to be 0 in the last three algorithms, Microscope, MI + AD + Align + Namespace, and FCA.

The checkout service is the caller of the email service. At the end of the checkout process, the checkout service will call the email service to send a confirmation email to the buyer. We expect an intelligent algorithm not be confused by this service. The last three algorithms, MI + AD + Align, MI + AD + Align + Namespace, and FCA, are all able to assign it with a very low correlation value to indicate the low probability of being the root cause service.

**Root Cause Function Analysis** The root cause function for the infinite loop bug in Online Boutique is SendOrderConfirmation. In the source code of the email service, there are a total of eight functions, but only four functions were active during the entire experiment. Figure 4.9 shows the result of the root cause function analysis after three runs, assuming each algorithm successfully detects the root cause service. With Pearson, Spearman, Kendall, and Microscope, the root cause function is not listed as the first candidate. MI, MI + AD + Align, MI + AD + Align + Namespace, and our FCA successfully determine the root cause function as the first candidate. The result of the last three algorithms, MI + AD + Align, MI + AD + Align + Namespace, and FCA share the same result, because there is no namespace or dependency information inside the root causer service. Thus they are essentially the same when determining the root cause function. The difference between the first rank and the second rank is also the largest with our FCA approach, which again strengthens our belief in the result of the root cause function.

Table 4.1 summarizes all ten performance bugs evaluated in the experiment. We evaluate four different types of bugs, including infinite loop, memory leak, data corruption, and timeout. Each type is evaluated in at least two different microservice applications.

Table 4.2 shows the summary of the root cause service results. Each row includes the ranking of the root cause service and the number of false positives for each algorithm. As we can see
Table 4.4: Overhead measurements of FCA analysis components, given a sampling interval of 30s.

<table>
<thead>
<tr>
<th>FCA Module</th>
<th>Execution Time (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service anomaly detection</td>
<td>2.28 ± 0.03 per sample</td>
</tr>
<tr>
<td>Service anomaly alignment</td>
<td>0.12 ± 0.02 per sample</td>
</tr>
<tr>
<td>Root cause service analysis total</td>
<td>31.19 ± 0.60 per sample</td>
</tr>
<tr>
<td>Function anomaly detection</td>
<td>2.24 ± 0.14 per sample</td>
</tr>
<tr>
<td>Function anomaly alignment</td>
<td>0.14 ± 0.04 per sample</td>
</tr>
<tr>
<td>Root cause function analysis total</td>
<td>41.64 ± 1.09 per sample</td>
</tr>
</tbody>
</table>

from Table 4.2, MI, Pearson, Spearman, Kendall, and Microscope all suffer from high number of false positive and often fail to rank the root cause service as the first service. In contrast, MI + AD + Align, and MI + AD + Align + Namespace have much fewer false positive and they both successfully detects the root cause service in all ten bugs. With the help of the namespace information, MI + AD + Align + Namespace is able to reduce the number of false positive. FCA is the only one that achieves a perfect result of ranking the root cause service as the first service, while having no false positive.

Table 4.3 shows the summary of the root cause function results. If an algorithm fails to detect the root cause service in the previous step, it is unable to proceed to detect the root cause function in the final step. In this case, we use "X" to indicate the failure of detecting the root cause function. We can see MI fails to detect five bugs, Pearson fails to detect seven bugs, Spearman and Kendall both fail to detect eight bugs, and Microscope fails to detect four bugs. In contrast, MI + AD + Align, MI + AD + Align + Namespace, and our FCA algorithm can all successfully detect the root cause function. The results of those three algorithms are the same because inside of a specific microservice, there is no namespace or service dependency information.

**Overhead Analysis**

FCA is fast and suitable for running in real time as shown in Table 4.4. Moreover, FCA is lightweight and only requires a limited amount of memory to run. The average and maximum memory usage are 123.92 MB and 235 MB, respectively.

4.4 **Summary**

In this chapter, we present FCA, a new dependency-aware full stack causal analysis framework for achieving fine-grained performance debugging in microservice architectures. FCA com-
bines anomaly detection, information theory based causal analysis, and dependency graphs to pinpoint root cause services and functions when a performance problem is detected. We have developed a prototype of FCA and evaluate it over ten performance bugs in three benchmark microservice applications. The results show that FCA successfully identifies the root cause services and functions in all of the tested bugs with much fewer false positives than other alternative solutions. FCA is lightweight, which makes it practical for production systems.
5.1 Container Vulnerability

The security of containers has attracted the attention of a lot of researchers in recent years. Zerouali et al. (2019) find that among 7,380 studied official and community Docker images, every release is vulnerable. Docker image vulnerability analysis (DIVA) Shu et al. (2017) is a scalable framework for discovering, downloading and analyzing both official and community Docker images. DIVA shows that both official and community Docker images contain more than 180 vulnerabilities on average. Tunde-Onadele et al. (2019) compare multiple detection schemes and suggest that dynamic detection outperforms static vulnerability scanning for containers. By combing static and dynamic detection schemes, the detection rate can be further improved. Martin et al. (2018) identify, in the different components of the Docker ecosystem, several vulnerabilities and detailed real world exploitation scenarios. They also propose possible fixes and discuss the adoption of Docker by platform-as-a-service (PaaS) providers. Lin et al. (2018b) study 11 exploits that can successfully bypass the isolation provided by the container to achieve privilege escalation. The authors then propose a defense mechanism to defeat those identified privilege escalation exploits. These studies emphasize the current vulnerable state of the container environment. Thus, there is a strong need for effective methods of detecting security attacks with a system...
5.2 Anomaly Detection

Abed et al. Abed et al. (2015) apply the bag of system calls technique to detect anomalies in containers. This processes a system call trace into vectors in intervals of the same total count. Yolacan et al. Yolacan et al. (2014) propose a process trace clustering approach using multi-hidden Markov models (HMM) to detect system call anomalies. Maggi et al. Maggi et al. (2008) combine clustering and a behavioral Markov model to build an unsupervised host-based intrusion detection system based on system call arguments and sequences analysis. Geng et al. Geng and Jia (2009) improve the efficiency of the sequence time delay embedding (STIDE) algorithm by only considering system call sequences that contain axis system calls. These axis system calls could more effectively represent the characteristics of normal behaviors with low overhead. Deep learning models have been recently explored for anomaly detection. Greenhouse Lee et al. (2018) is designed as a zero-positive machine learning system which does not require any anomalous sample using long short-term memory (LSTM) method. Malhotra et al. Malhotra et al. (2015) present stacked LSTM networks for detecting anomalies in several time series datasets. Taylor et al. Taylor et al. (2016) apply LSTM to detect anomalies in a car’s controller area network (CAN) with low false alarm rate for catching possible intrusion to CAN. Sakurada et al. Sakurada and Yairi (2014) propose to use autoencoders with nonlinear dimensionality reduction for general anomaly detection.

In comparison to existing anomaly detection schemes, CDL focuses on addressing special challenges of insufficient training data in container environments. CDL proposes a new classified distributed learning framework which is orthogonal to specific machine learning algorithms used for anomaly detection. Although CDL currently employs the autoencoder anomaly detection algorithm, it can be easily applied to other anomaly detection algorithms.

5.3 Federated Learning

Konečný et al. Konečný et al. (2016) propose a decentralized approach to learning a shared centralized model, called federated learning. This is executed by aggregating local-computed updates from a large number of clients over an unreliable network. Later, Lin et al. Lin et al. (2017) design deep gradient compression (DGC) to reduce the communication bandwidth of federated learning. Yao et al. Yao et al. (2019) further improve federated learning by aggregating features from both the local and global models to achieve higher accuracy with less
communication cost. Sozinov et al. (2018) compare centralized training with federated learning and show that federated learning can achieve acceptable accuracy similar to centralized learning. CDL implements distributed learning using aggregated training data in a similar way as federated learning. However, CDL incorporates application classification into distributed learning to overcome the challenge of detecting security attacks in dynamic container systems.

5.4 Distributed Machine Learning

Hashdoop Fontugne et al. (2014) improves the detection accuracy of network traffic anomaly detectors on Hadoop. They achieve this by carefully splitting network traffic such that the sampled traffic maintains its original structure. Song et al. (2017) provide a parallel k-medoids clustering algorithm for high accuracy and efficiency. Chen et al. (2018) provide a robust model training system which is orders of magnitude faster than alternate median-based approaches. Gopal et al. (2013) achieve an order of magnitude decrease in training time through a parallel calculation of the likelihood function in logistic models. Petuum Xing et al. (2015) provides a unified parallel optimization framework to help machine learning (ML) programs run faster. Similar to the above approaches, CDL promotes the distributed learning approach. However, CDL differs from the above work by focusing on improving the accuracy of security attack detection using input data from similar applications to create lightweight application-specific models with low training cost.

5.5 Container vulnerability detection

Previous work has been done in detecting vulnerabilities within containerized environments. Lin et al. (2018b) studied 11 privilege escalation exploits and proposed a defense mechanism to defeat privilege escalation attacks. Self-Patch Tunde-Onadele et al. (2020) combined light-weight dynamic attack detection and targeted patching to achieve effective security protection for containerized applications. CDL Lin et al. (2020) was a classified distributed learning framework for containerized applications. Lindvärn and Lundqvist (2021) proposed using isolation forest for anomaly detection to achieve good detection rate and a relatively low false positive rate for 22 attacks. DIVA Shu et al. (2017) performed vulnerability detection on Docker Hub images. Similarly, DIVDS Kwon and Lee (2020) diagnosed Docker images when they are uploaded or downloaded from Docker image repositories. SHIL complements the existing work by providing a new efficient security attack detection mechanism by combining supervised and unsupervised learning methods.
5.6 Semi-supervised learning based intrusion detection

Previous work has been done in adopting a semi-supervised learning model to generate or label data for intrusion detection. Rathore et al. Rathore and Park (2018) proposed a decentralized fog-based attack detection framework which uses the semi-supervised fuzzy c-means (ESFCM) algorithm with an extreme learning machine (ELM) to detect attacks that occurred on internet of things (IoT) devices. Idhammad et al. Idhammad et al. (2018) introduced a semi-supervised learning approach for DDoS detection based on network entropy estimation, co-clustering, information gain ratio, and extra trees. Zimba et al. Zimba et al. (2020) proposed a semi-supervised algorithm based on shared nearest neighbour (SNN) clustering to detect advanced persistent threat (APT) attacks. Khonde et al. Khonde and Ulagamuthalvi (2019) described an ensemble-based semi-supervised learning approach for a distributed intrusion detection system. Compared with the existing work which started from supervised learning models, SHIL uses unsupervised models as the main decision-making modules and only employs supervised models on-demand for boundary cases to filter out potential false alarms.

5.7 Supervised learning based intrusion detection

Previous work has been done in applying supervised learning methods to intrusion detection. Anthi et al. Anthi et al. (2019) described a three-layered system using supervised learning for intrusion detection for smart home IoT devices. DTB-IDS Moon et al. (2017) presented a decision-tree-based anomaly detection method to detect APT attacks. Aksu et al. Aksu et al. (2018) applied the Fisher Score algorithm to selecting features and fed the features into support vector machine (SVM), k-nearest neighbour (k-NN) and decision tree (DT) algorithms for intrusion detection. Hosseini et al. Hosseini and Azizi (2019) performed incremental learning with supervised models to detect distributed denial of service (DDoS) attacks. Compared with the supervised learning methods, SHIL does not required labelled training data and only uses supervised models for false alarm filtering.

5.8 Unsupervised learning based anomaly detection

Previous work has been done in applying unsupervised learning methods to intrusion detection. Unicorn Han et al. (2020) used K-medoids and data provenance analysis to detect Advanced Persistent Threats (APTs). Scholkopf et al. Schölkopf et al. (2000) proposed a one-class support vector machine (SVM) and defined a frontier as a threshold for outlier detection. The isolation forest Liu et al. (2008) isolated anomalies from normal data by randomly selecting
a feature and a value in the possible range to split data points. Khan et al. (2019) proposed a hybrid intrusion detection system by combining multiple unsupervised learning methods. In comparison, SHIL leverages unsupervised learning methods to detect a set of candidate anomalies and uses supervised learning to filter out likely false alarms produced by unsupervised learning methods using boundary case thresholds.

Nevertheless, intrusion detection systems (IDS), including SHIL, have limitations. Rosenberg and Gudes (2017) show that attackers can use a camouflage algorithm to mislead machine learning classifiers such as decision trees and random forest. If attackers gain partial information about the model training set and features, they can modify their attacks to exhibit benign patterns. We may alleviate such attacks with measures such as training updates to the SHIL anomaly detection model. Furthermore, Shu et al. (2015) present a unified framework for any program anomaly detection method and prove that there is a theoretical accuracy limit. The authors note that system call based anomaly detection is limited by a lack of knowledge of program internal information such as call stack activity. We can complement SHIL with security tools that leverage such program internal context.

5.9 Microservice performance root cause analysis

Microservice approaches in recent research literature conduct various analyses on customized dependency graphs to provide a ranked list of possible anomalous root cause services. MicroRCA Wu et al. (2020) uses a personalized PageRank algorithm to reason about its attributed graph based on metrics monitored at the application and system levels. It begins its analysis in response to anomaly alerts from the unsupervised clustering algorithm named balanced iterative reducing and clustering using hierarchies (BIRCH). MicroRCA uses Pearson correlation as the similarity measure for their anomaly score which captures linear dependence. Whereas, we use mutual information (MI) to learn general dependence. MonitorRank Kim et al. (2013) performs a modified random walk on its call graph constructed from time series metrics and API pairs (each referred to as a sensor). The walk is based on a pseudo-anomaly clustering of sensors showing similar patterns to the front end sensor. Although MonitorRank outputs root cause communication relevant API, we rank with a finer granularity by ranking the root cause functions. MonitorRank also uses correlation, which is limited to linear dependence relationships. AutoMap Ma et al. (2020) uses a heuristic investigation algorithm that also involves a random walk on its behavior graph of service correlations over multiple types of metrics. It defines operations that allows the behavior graph to choose the suitable metric to diagnose a given scenario. AutoMap only uses a variety of metrics and assumes no prior known calling microservice topology. However, their analysis is based on Pearson correlation which is unable
to capture non-linear dependence.

Wang et al. (2021) leverage Granger causality models on only monitored log data of a microservice-based application to infer the impact of dependencies between microservices. They evaluate the performance of the state-of-the-art linear and nonlinear (i.e., neural) Granger causality methods on both synthetic data and real-world error log data from a publicly available benchmark microservice system. The neural network models capture Granger causality and can identify long-range dependencies. However, it requires much larger data for training than our work does. Microscope Lin et al. (2018a) calculates the Pearson correlation coefficient between the SLO metrics data of the front end faulty service and that of each candidate service obtained from its causality graph. The directed acyclic graph (DAG) combines separate communicating service dependency graphs from network analysis and non-communicating service dependency graphs from a parallelized PC-algorithm. Microscope mainly collect request latency metrics and network connection information for their dependency graph while we gather spans. Nevertheless, they are also limited to Pearson correlation as opposed to our use of MI. MicroCause Meng et al. (2020) combines a simple yet effective path condition time series (PCTS) algorithm to capture the sequential relationship of time series data, and a novel temporal cause oriented random walk (TCORW) method integrating the causal relationship, temporal order, and priority information of monitoring data. The PCTS algorithm builds on top of the improved PC-algorithm. The cause oriented random walk involves partial correlation to locate the root cause. While partial correlation tries to remove the effect of other co-related variables, it still indicates linear dependence relationships between variables.

Our system, FCA, not only provides the root cause service, but also provides the root cause function, which is not localized by the above related work. We generally use stronger predictability measures with MI than those that rely on correlation.

### 5.10 Causality analysis

In addition, other approaches leverage causality techniques to identify root causes of performance bugs that occur within a single machine. PerfSig He et al. (2022) is a multi-modality performance bug signature extraction tool which can identify principal anomaly patterns and root cause functions for performance bugs. It uses mutual information to capture the non-linear causal relationship between system metrics and function calls, and system metrics and system logs. In contrast, we focus on performance bugs in the microservice context that affect multiple nodes so we consider service dependencies first to find the root cause service. Moreover, during causal analysis, we use techniques like anomaly alignment to improve our mutual information analysis. PerfSig extracts principal patterns with low pass filtering,
time series discord mining, and fast fourier transform (FFT) techniques while we only require anomaly detection. Argus Weng et al. (2021) is a casual tracking tool for debugging performance anomalies in macOS. It introduces strong edges to represent causal connections, and weak edges to represent *likely not causal* connections in a trace graph. It then uses beam search to select the most likely causal path. Argus is able to identify the root cause API that is not exposed in some run-time cases by comparing the normal and abnormal executions using subgraph comparison. Argus builds its graph based on low level information like system calls and thread scheduling events. It uses the authors’ knowledge of communication messages among events to establish strong or weak relationships in the graph. Our work is an information theory based approach using less-intrusive system metrics and spans which make it suitable for microservices. Pivot Tracing Mace et al. (2015) is a monitoring framework for distributed systems that combines dynamic instrumentation with a novel happened-before join operator to allow tracking information across component or machine boundaries. Pivot Tracing lets users define arbitrary metrics at one point of the system during run-time, while being able to select, filter, and group them by events meaningful at other parts of the system, even when crossing the boundaries. Pivot Tracing implements a metadata propagation mechanism to better understand execution order over various processes, machine, and application boundaries. It also depends on an expert to determine code tracepoints and queries for debugging an issue. However, we avoid additional instrumentation overhead to existing microservice monitoring tools as we investigate causal relationships with Granger causality over system metrics and service dependencies in spans. Qiu et al. Qiu et al. (2020) propose an approach for mining causality and diagnosing the root cause using knowledge graph technology and a causal search algorithm. Their causality graph uses the PC algorithm involving conditional independence and partial correlation steps while their causal search algorithm also uses Pearson correlation. However, these correlation approaches do not address non-linear dependence relationships, which our work considers with MI. CloudCanary Zhai et al. (2020) conducts real-time audits on service updates to diagnose the root causes of correlated failure risks and then provides improvement plans to increase reliability. They reduce structural reliability auditing tasks to a Boolean satisfiability problem. In contrast, we do not take the same approach of reducing our problem to a Boolean satisfiability problem. CloudCanary relies on a fault graph, while our system uses a service dependency graph based on spans.

Kobayashi et al. Kobayashi et al. (2017) propose a causal-inference-based method for extracting pinpoint system failures and identifying their causes from network syslog data. The method reconstructs causality of network events from a set of time series of events. Kobayashi et al. build their causal graph of logged network events also using a PC algorithm variation. Although, they use the Fisher-Z test which is a transform of Pearson correlation, they also
use the G-squared test which can handle both linear and nonlinear dependence Wang et al. (2017). However, they focus on outputting network events and types which the developer need to would further examine to fix, compared to our work which directly provides the root cause function. AutoRoot Jing et al. (2021) is a fast and accurate multi-dimensional root cause localization algorithm, which uses an adaptive density clustering to improve the accuracy and an effective filtering mechanism to reduce the search time. AutoRoot actually predicts KPI values and performs computation to quantify the deviation as anomaly scores before outputting the root cause cluster. Instead, we use information theory methods that compare the existing metrics without predicting future data values. Further, we retrieve finer-grained root cause functions for effective debugging.

Compared to the above causal analysis research, FCA takes a lightweight, non-intrusive approach that finds root causes at the function-level. FCA leverages Granger causality with the mutual information measure to also identify non-linear dependence relationships and applies techniques like anomaly alignment to enhance the debugging results.

### 5.11 Distributed performance bug detection

Moreover, previous work present solutions specifically designed for distributed performance bugs. By analyzing the system execution on small-scale workload execution, PCatch Li et al. (2018) can automatically predict performance cascading bugs (PCbugs) that cause the system slowdown and even threaten the system availability. To accomplish this, PCatch consists of three key functions. It uses program analysis to identify code regions whose execution time can potentially increase dramatically with the workload size, adapts the traditional happens-before model to analyze the software resource contention and performance dependency relationship, and performs dynamic tracking to identify the slowdown propagation job scope. PCatch was evaluated using several distributed systems including Cassandra, Hadoop MapReduce, HBase, and HDFS. As opposed to PCatch, our analysis uses mutual information to measure causal dependence relationships and does not require code analysis. IASO Panda et al. (2019) is a peer-based, non-intrusive framework that addresses the fail-slow fault where a hardware or software can keep functioning with a much slower performance than expected. IASO converts the timeout signals into a stable and accurate fail-slow metric that it uses to successfully isolate a slow node within minutes. IASO uses low-cost timeout information to create a degradation score that considers the peers of similar services. Unlike FCA, it does not involve causation metrics and only finds the root cause at the service node level. CauseInfer Chen et al. (2014) is a black-box cause inference system that provides hints for distributed systems bugs without instrumentation. CauseInfer can automatically construct a causality graph and infer the causes
of performance problems along the causal paths in the graph. CauseInfer use the G-squared based measure along with the PC algorithm to build their causality graph. It then implements cumulative sum and z-score methods to calculate the anomaly scores. However, we consider mutual information to obtain causal scores and service call dependencies to improve the ranking. CauseInfer also uses TCP request latency while we use spans and system metrics like CPU utilization. Saha et al. Saha and Hoi (2022) process expert service incident reports expressed as problem review board (PRB) data by building an AI pipeline for root cause analysis. They build a causal knowledge graph with neural natural language processing (NLP) methods by extracting relevant history of symptom, root-cause, and resolution information. Thereafter, they infer root causes and fixes for current incidents. Saha et al. primarily support root cause analysis with an extensive record of previous incident reports from experts. Whereas, we establish causation using an information theory techniques over recent time series data.

Scalability bugs Leesatapornwongsa et al. (2017) are cluster-scale dependent latent bugs, which typically surface in large-scale deployments. ScaleCheck Stuardo et al. (2019) detects scalability bugs in large storage systems. ScaleCheck employs program analysis to find potential causes of scalability bugs, and a series of colocation techniques, for testing implementation code at real scales but on a commodity PC. ScaleCheck has been tested in several large-scale storage systems including Cassandra, HDFS, Riak, and Voldemort. ScaleCheck takes a program analysis and test simulation approach while we examine application-agnostic system metrics.

Consistency-Guided Fault Injection (CoFI) Chen et al. (2020) injects network partitions to expose partition bugs that are more likely to occur when the cloud systems are in inconsistent states. When detecting violations, CoFI injects network partitions to prevent the cloud systems from recovering back to consistent states, then checks if the cloud system still behaves correctly at the inconsistent states. CoFI is a low level approach that looks into program communication points to understand inconsistent network states among nodes. Whereas, we use higher-level statistical approaches to analyze system metrics and span time series. Arvalus and its variant D-Arvalus Scheinert et al. (2021) are neural graph transformation methods that model system components as nodes and their dependencies and placement as edges to improve the identification and localization of anomalies. Given a series of metric KPIs, those methods predict the most likely system state and perform localization when an anomaly is detected. Arvalus takes a graph convolutional neural network approach, using metrics and node meta-information to extract weights for their node dependency graph instead of constructing a causality graph score. DISTALYZER Nagaraj et al. (2012) can automatically help developers investigate performance issues in distributed systems through the use of logs. DISTALYZER can infer the strongest correlations between the system components and performance. In contrast, our system does not rely on application logs but leverages system metrics and span time series.
DISTALYZER also needs two separate sets of logs: one with good performance and the other with bad performance which we do not require. CrashTuner Lu et al. (2019) is designed for automatically detecting crash recovery bugs in crash-recovery-related mechanisms via fault injection. The authors observe that a crashed node accessing meta-info variables that reference high-level system state information often triggers crash-recovery bugs. Hence, CrashTuner identifies the crash points by automatically inferring meta-info variables via a log-based static program analysis. Unlike CrashTuner, we focus on monitoring system metrics and spans to locate the root cause service without the need to interact with the application at the code level.

The distributed detection approaches tend to focus on specific performance bugs with low-level mechanisms. FCA adopts an information-theory-based approach to detect the root cause function of widespread distributed performance bugs.
The contributions of this dissertation are a new classified distributed learning framework for security attack detection in containerized applications, a self-supervised hybrid learning system for more efficient security attack detection in container-based environments, and a causal analysis framework for identifying root causes of performance bugs in microservice architectures. Specifically, we make the following contributions:

• We have presented CDL, a new classified distributed learning framework that aims at achieving practical and efficient security attack detection for containerized applications. CDL integrates online application classification and application-specific anomaly detection models to overcome the challenges of lacking sufficient training data for individual short-lived containers. We have implemented a prototype of CDL and conducted experiments over 33 real world vulnerability exploits in 24 commonly used applications. Our results show that CDL can reduce false positive rates from over 12% to 0.24% compared to traditional learning methods without aggregating training data from different containers and increase the true positive rate from 45% to 74% compared to simple training data aggregation without performing application classifications. CDL supports real time security attack detection, which makes it practical for production computing environments.

• We have presented SHIL, a new self-supervised hybrid learning system for more effi-
ciently detecting security attacks in container-based computing environments. SHIL identifies anomaly detection boundary cases as most likely false alarms and combines unsupervised and supervised machine learning methods to filter out majority of the false alarms without missing most of the true attacks. For practical deployment of supervised learning models, SHIL adopts a self-supervised learning approach to labelling training data automatically using outlier detection over a window of recent measurement samples when attack alerts are first raised. Our experimental results with real world security attacks including the recent high-impacting Log4j attack show that SHIL can significantly reduce false alarms by up to 91% while maintaining similar detection rates compared to existing pure-supervised or pure-unsupervised methods. SHIL is light-weight and does not require manual data labelling, which makes it practical for security attack detection in container-based production environments.

- We have presented FCA, a new causal analysis framework for distributed performance bugs in microservice architectures. FCA finds the root cause of the service and function by analyzing system metrics and spans, which carry information about request timing, service dependencies, and function execution. It quantifies causal relationships with information theory measures, time-series alignment techniques, and service dependency graphs. We have developed a prototype of FCA and evaluate it over ten performance bugs in three benchmark microservice applications. The results show that FCA successfully discovers the root cause in 100% of bugs.
REFERENCES


Authors, L. (2023a). Locust - a modern load testing framework.


Authors, T. J. (2023c). Jaeger: open source, end-to-end distributed tracing.

Authors, T. O. (2023d). Opentelemetry.


Kharpal, A. (2023). Microsoft users hit with global cloud outage that impacted products like teams and outlook.


Microsoft (2022). Modernizing microsoft 365 with a move to windows containers on azure kubernetes service (aks).


Prometheus (2023). Monitoring linux host metrics with the node exporter: Prometheus.


IEEE.

containerized applications. In 2020 IEEE International Conference on Autonomic Computing
and Self-Organizing Systems (ACSOS), pages 21–27. IEEE.

network for network intrusion detection. In 2017 International Conference on Advances in
Computing, Communications and Informatics (ICACCI), pages 1222–1228. IEEE.

structure on cloud application microservices using granger causality models. In 2021 IEEE
14th International Conference on Cloud Computing (CLOUD), pages 558–565. IEEE.

ern desktop applications with annotated causal tracing. In 2021 USENIX Annual Technical
Conference (USENIX ATC 21), pages 193–207.


and Management Symposium, pages 1–9. IEEE.

Petuum: A new platform for distributed machine learning on big data. IEEE Transactions on

Yao, X., Huang, T., Wu, C., Zhang, R., and Sun, L. (2019). Towards faster and better feder-
ated learning: A feature fusion approach. In 2019 IEEE International Conference on Image
Processing (ICIP), pages 175–179. IEEE.

hmm's. In 2014 IEEE Eighth International Conference on Software Security and Reliability-
Companion, pages 25–30. IEEE.

Zerouali, A., Mens, T., Robles, G., and Gonzalez-Barahona, J. M. (2019). On the relation be-
tween outdated docker containers, severity vulnerabilities, and bugs. In 2019 IEEE 26th
International Conference on Software Analysis, Evolution and Reengineering (SANER), pages
491–501. IEEE.

before you change: Preventing correlated failures in service updates. In 17th USENIX Sym-
posium on Networked Systems Design and Implementation (NSDI 20), pages 575–589.