Detecting Cyber Security Attacks against a Microservices Application using Distributed Tracing

Stephen Jacob[a], Yuansong Qiao[b] and Brian Lee[c]
Department of Computer and Software Engineering, Athlone Institute of Technology, Dublin Rd., Athlone, Ireland

Keywords: Microservices, Cyber Security, Distributed Tracing, Anomaly Detection.

Abstract: Microservices are emerging as the dominant software design architecture for many different applications, and cyber attacks are targeting more software organisations every day. Newer techniques for detecting cyber intrusions against such applications are in high demand. Application functionality that is executed within a microservices application can be monitored and logged using distributed tracing. Distributed tracing is normally used for performance management of microservices applications. In this paper, we used distributed tracing for detecting cyber-security attacks. Each microservice call, or sequence of calls, executed in response to a request by an end user of the application is logged as a trace. Anomaly detection is a means of detecting irregular or unusual events or patterns in a data set that occur to a greater or a lesser degree than the majority of the data. In this paper, we present initial work that identifies anomalous distributions of traces. A frequency distribution of traces is obtained from normal data and traffic is identified as an anomaly candidate if it differs sufficiently from the base distribution. This approach is evaluated using a password guessing attack. In addition, we briefly discuss a NoSQL injection attack which we argue is difficult to detect using trace data.

1 INTRODUCTION

Many software applications such as those developed recently by Amazon, Twitter and Netflix are built using a microservices based architecture. Hackers will continue to target these state-of-the-art applications using newer and more sophisticated forms of cyber attacks. Consequently, software security personnel require similarly updated means of detecting cyber threats to their software applications.

One way to do this in a microservices-based application is to monitor the system's overall behaviour using distributed tracing and perform anomaly detection to discover anomalies or outliers in the system's overall functionality. A cyber attack deliberately targeting the app can create an irregular process activity within the application. By detecting irregular behaviour, cyber security personnel can be forewarned of ongoing cyber attacks.

In this paper, we use a distributed tracing system to monitor behaviour in microservices-based applications with a view to identify cyber security attacks. For our experiment, we generate normal microservice traffic and then execute cyber security attacks against the application. We aim to detect the attack by detecting an anomaly in the trace data. To the best of our knowledge, our work is the first attempt at using distributed tracing to detect cyber security attacks in an application with a microservices architecture.

A machine learning approach is applied to analyze the resulting traces. A user request generates a sequence of system calls which are logged by the distributed tracing system. Normal behaviour of the system over some interval of time will generate a number of these system call sequences. One approach to detect cyber security attacks is to identify a set of system call sequences that is in someway different from normal behaviour. Our approach, which is early work, is to compare the frequency distribution of unique system calls in the normal and attack data.

The structure of this paper is as follows: Section II will explore similar related works involving microservices applications, the use of distributed tracing to monitor an application's behaviour and the use of anomaly detection to detect irregular system calls in the overall application functionality. Section III will outline the relevant background information on the microservices architecture, distributed tracing

[a] https://orcid.org/0000-0003-2297-4343
[b] https://orcid.org/0000-0002-1543-1589
[c] https://orcid.org/0000-0002-8475-4074

Jacob, S., Qiao, Y. and Lee, B.
Detecting Cyber Security Attacks against a Microservices Application using Distributed Tracing.
DOI: 10.5220/0010308905880595
In *Proceedings of the 7th International Conference on Information Systems Security and Privacy (ICISSP 2021)*, pages 588-595
ISBN: 978-989-758-491-6

and anomaly detection. Section IV outlines an open-source microservices benchmark suite and describes the software tools said application is configured to use for our experiment. In Section V, we present two available end-to-end microservice applications from the benchmark suite, outline their general system behaviour and architectural design and two different possible forms of cyber attack we carried out for the experiment. Section VI presents the to-date report and current results obtained from the experiment. In Section VII, we outline the conclusions drawn from our experiment and possible further work to explore in the future.

2 RELATED WORK

(Gan and Delimitrou, 2018) presented and characterized an end-to-end microservices application that implemented an extensible movie renting and streaming service. An advantage discovered using the movie service was being able used to measure the duration time between client API calls to the application and the computation of the service. The appeal and convenience of the microservices architecture was highlighted, particularly how the individual services can communicate with each other using remote procedure calls and how the framework can be used to analyse performance bottlenecks. No analysis was carried out to detect anomalies caused by security attacks.

(Nedelkoski et al., 2019) developed a deep learning approach to model distributed tracing and log network data with a focus on time-series based anomaly detection. The model was trained to learn the general behaviour of complex distributions of distinct data traces over a long period of time. An anomaly detection technique was developed using a probability-based error threshold setting. The anomaly detection models were used to classify data points as anomalies or not and provide descriptive analysis and results. Furthermore, a post-processing strategy was combined with the threshold setting to mitigate the occurrence of data points erroneously classified as anomalies, or false positives. Again, the techniques were applied exclusively to performance bottleneck and did not examine security attacks.

(Chandola et al., 2009) wrote a survey with the intention of providing a thorough overview of research on anomaly detection. One of the highlights provided by their survey is the observation that data points classified as outliers are only anomalous in regards to a context. Another highlight is that anomalies can be divided into different categories, each with key assumptions that differentiate outliers from normal, regular data. An expansive literature on various techniques that can be used for anomaly detection is also provided, as well as their advantages and disadvantages. Finally, this survey observes that anomaly detection is being utilized in more and more complex systems every day.

(Gan et al., 2019a) highlights the recent shift of monolithic architectures to the loosely coupled microservices-based framework for many software applications. The paper also presents an open-source benchmark suite known as DeathStarBench, comprised of multiple microservices-based applications. In the paper, DeathStarBench is used primarily to study the immensely complex architectural characteristics of a microservices application, observe these applications in real deployment by hundreds of users and carry out the identification of performance bottlenecks.

(Gan et al., 2019c) used a performance debugging system known as Seer which monitors temporal and spatial patterns of behaviour in general cloud applications, including microservices. The Seer system combines distributed tracing with low-level hardware monitoring to detect and diagnose Quality of Service (QoS) violations to avoid unwanted behaviour cascading through the application system. The advantages of microservices-based architectures simplifying correctness debugging, and Seer identifying application level bugs and indicating how to improve the microservices framework were highlighted to achieve optimal performance.

A survey (Toth and Chawla, 2018) was written to provide an overview and a more comprehensive understanding of the concept of group anomaly detection in contrast to pointwise anomaly detection. The study highlights the advantages of group deviation detection techniques including discovery of abnormal behaviour, mitigating risks and prevention of malicious activity in the fields of particles, physics and health care collusion. The survey also outlines the state-of-the-art techniques for carrying out group deviation detection as well as the frameworks and data structures.

3 BACKGROUND INFORMATION

In this section, we will describe the microservices software architecture and give an outline of the different technologies and anomaly detection techniques used in this paper.

3.1 Microservices

Microservices, or simply the microservices architecture (MSA), is a service-oriented software architectural design which divides the overall application into a collection of smaller inter-connected component services. A single microservice handles one business service of the application's total functionality, e.g. database queries or message posting.

Microservices-based applications share a common cross-service API allowing different microservices to communicate with each other. A single microservice has a well-defined interface that can be called in response to a user's request and subsequently communicates with other microservices using either a RESTful API or remote procedure calls (RPC) (Sun et al., 2015) (Nagothu et al., 2018). In a distributed application, a single microservice will operate alongside other microservices but can be developed, deployed and scaled independently. Another advantage of the MSA is that the design supports programming language and framework heterogeneity (Gan and Delimitrou, 2018). Therefore, microservices are quickly becoming a newer platform trend for cloud-native applications such as Twitter, Amazon and Netflix (Gan et al., 2019b).

3.2 Distributed Tracing

In the field of software engineering, distributed tracing is the process of monitoring, profiling and logging the execution path through a cloud-native application at runtime in response to a user's request. A user's request typically results in behaviour that can span across multiple services in the application, resulting in a distributed trace, a detailed record of the execution path through the application.

A **distributed trace** itself is represented as the sequential set of spans, each sharing a traceID. A **span** is represented as a single application event, or service call executed in response to a user's request. Other fields, or characteristics in a span include the name of the executed call, the timestamp and the duration of the call. The span can also contain meta information including an executed HTTP URL and the response code to that HTTP call.

Service tracing and system logging are vital to understanding the behaviour of user's requests that access and propagate through an application. Due to the complexity of a cloud system domain, distributed tracing is well suited and commonly used for performance monitoring of microservices-based applications.

3.3 Anomaly Detection

Anomaly detection is the process of detecting irregular instances or occurrences within a data series. These anomalous instances do not conform to the general behaviour of the data. Anomalous data can indicate an error is present. Anomalous instances, or outliers, can occur in a variety of system data applications including fraud detection for health insurance, finances, cyber attack intrusion and military surveillance for enemy activity.

Anomaly detection can be carried out in two different ways: supervised and unsupervised anomaly detection (Chandola et al., 2009). In supervised anomaly detection, both the general data and irregular data series are categorized and labeled. The labeled data is then trained by a machine learning algorithm to learn the general behavior of the data and subsequently detect anomalous data. The data will be trained whether it contains anomalies or not. By contrast, unsupervised anomaly detection does not require labeled training data. It is implied that the normal instances are far more frequent than anomalous instances and the trained model is robust to such anomalies. Anomaly detection is generally used in unsupervised databases, which do not use labels and lack structure.

There is another classification of anomaly detection techniques into pointwise anomaly detection and group anomaly detection (GAD) (Toth and Chawla, 2018). The more recognizable form is pointwise anomaly detection which detects individual instances in a data set that are anomalous. Pointwise anomaly detection is not useful for the detection of anomalous behaviour of groups of instances. Group-based anomaly detection is an emerging form of detection which detects an outlier group of instances in the data.

4 EXPERIMENT

In this section, we examine the use of distributed traces for cyber security anomaly detection using the open-source microservices benchmark suite DeathStarBench (Gan et al., 2019a).

First we give an overview of DeathStarBench and the associated distributed tracing technologies. Then we analyze two well-known cyber security attacks, a password guess attack and NoSQL Injection. In particular, we examine if it is possible to detect these attacks using microservices distributed tracing data.

We see that, while in the case of the password attack, we can detect a group anomaly by looking at the distribution of unique call sequences over a short period of time while it is quite difficult to detect a NoSQL Injection attack in the case where no changes have been made to the microservice implementations.

4.1 DeathStarBench

The microservice-based application executed for this report was **DeathStarBench**, an open-source benchmark suite comprised of several end-to-end applications including a **social networking** application where registered users compose posts and follow other users, a **media** application where users can post movie reviews and a **hotel reservation** service (Gan et al., 2019a). The social network application was chosen for examination in this work.

4.1.1 Docker

Docker is a Platform as a Service (PaaS) tool designed to create, deploy and run applications in a virtual environment. Individual application functionality and their respective source code, dependencies and libraries are isolated and compressed into files called Docker **images**. These image files are then used as templates to build lightweight executable packages called **containers**.

In the DeathStarBench suite, every individual microservice is run as a Docker container. A tool called *docker-compose* is used to create, configure and start the microservices that are part of the application.

4.1.2 Thrift

Thrift is binary communication protocol and interface definition language developed by Apache software Foundation. The Thrift language provides support for client-server RPCs. In the DeathStarBench suite, the inter process communication between the different microservices is handled by Thrift and the interfaces for the microservices are written in the Thrift language.

4.1.3 Jaeger

Jaeger is an open-source distributed tracing system that traces a client's request execution path through the application. Jaeger is comprised of three different components with its own function: *jaeger-agent*, a network daemon that listens for executed span or service calls sent over a User Datagram Protocol (UDP), *jaeger-collector* where the span data is stored and the *jaeger-query* which queries the jaeger-collector, retrieves the resulting traces and serves as a JavaScript UI for those traces.

4.1.4 ElasticSearch

In our experiment with DeathStarBench, we reconfigured the Jaeger-collector component to work with ElasticSearch, a storage backend for JSON documents. The ElasticSearch API was primarily used to download the data in bulk.

The architecture of the microservices application including the Docker Host or container, and the Jaeger components configured with the ElasticSearch storage backend are displayed in Figure 1.

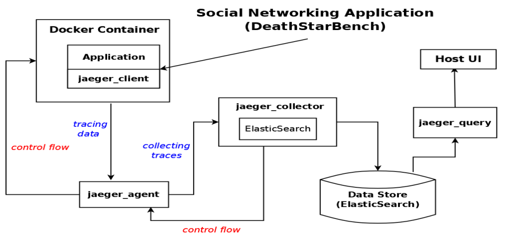

Figure 1: Application Structure for Social Network.

4.2 Password Guessing Attack

A password guess attack is a type of remote to local (R2L) cyber attack. An intruder attempts to access unauthorized information from a local machine through a remote machine (Dhanabal and Shantharajah, 2015). The hacker tries multiple times to log into an application by continuously attempting any combination of usernames and passwords until they find one that works. Therefore, a password attack would be comprised of multiple incorrect logins within a short span of time.

The social networking application from the DeathStarBench microservices suite supports a user login operation, so we selected it for our experiment. We used a password guessing attack, resulting in multiple incorrect logins within a short period of time, as an example of anomalous user activity. This irregular activity would correspond to a group anomaly whereas one or two incorrect logins in regular trace data would be normal user activity. The method to detect this group anomaly is to detect when the frequency distribution of traces is different from the frequency distribution of normal data.

4.2.1 Frequency Analysis

A user request to the application generates a trace which is a sequence of spans. Each event span is identified by a combination of the service name and operation name.

We defined a distance metric between two sets of traces, T_1 and T_2. Let $s_1, s_2 \ldots s_n$ be the set of all unique sequences in T_1 and T_2. The frequency distributions f(T_1) and f(T_2) are defined as:

$$f(T_1) = (s_1, f_1^1) + (s_2, f_2^1) + ... + (s_n, f_n^1) \quad (1)$$

$$f(T_2) = (s_1, f_1^2) + (s_2, f_2^2) + ... + (s_n f_n^2) \quad (2)$$

In the equations above, f_i^j is the frequency of s_i in T_j. The difference in the frequency distributions $d(T_1, T_2)$ is defined using the Euclidean distance metric:

$$d(T_1, T_2) = \sqrt{(f_1^1 - f_1^2)^2 + ... + (f_n^1 - f_n^2)^2} \quad (3)$$

4.2.2 User Requests and System Calls

We carry out a simple experiment with two different types of requests: **composePost**, where a user uploads a posts consisting of media content such as text, tags and links, and **userLogin** where a user logs in.

When a post is uploaded to the application, the posts are stored in a MongoDB database and cached in a Memcached service. The following are some of the microservice calls that are executed during a **composePost** request: *UploadMedia, UploadText, UploadUniqueID, UploadURLs, UploadUserMentions, StorePost, WriteUserTimelines, MongoInsertPost and MmcSetPost*. For **composePost** requests, there are in total 27 different microservice calls executed and 1308 different microservice call sequences.

For the **userLogin** requests, there are 4 microservice calls and 3 different sequences. The four calls used are *Login, MmcGetLogin, MongoFindUser, MmcSetLogin*. A **userLogin** request can be satisfied from the cache if the appropriate user object is in Memcached. If not, a call needs to be made to the MongoDB database to get the user information. The microservice call MmcGetLogin checks if the user's credentials have been cached in Memcached. MongoFindUser looks for the registered user in MongoDb. MmcSetLogin caches user credentials in Memcached.

The following are valid sequences of microservice calls executed during a **userLogin** request.

- (Login -> MmcGetLogin)
- (Login -> MmcGetLogin -> MongoFindUser)
- (Login -> MmcGetLogin -> MongoFindUser -> MmcSetLogin)

The first sequence of calls can be executed for both a correct and an incorrect login, depending on whether a correct or incorrect password is supplied. The second sequence corresponds to an incorrect login as the provided password is not correct and the user data is not cached to Memcached. The third sequence corresponds to a correct login as matching user credentials have been found in MongoDb and are stored in Memcached.

4.2.3 Definition of Training (Normal) Data

Normal application requests are composed primarily of **composePost** requests and correct **userLogin** requests, along with a small number of incorrect **userLogin** requests. In normal application traffic, only a small number of **userLogin** requests would be incorrect, caused by users entering the wrong credentials by mistake. An experiment carried out by (Brostoff and Sasse, 2003) showed that only 10% of all logins were incorrect.

The requests for **composePost** were sent to the social networking application using a HTTP workload traffic generator. This returned a total of 1856 distributed traces. A number of **userLogin** requests were also sent and returned 592 correct and 102 incorrect login traces. These HTTP requests generated 2550 traces in total with a vocabulary of 32 span event types. 2000 of these were set aside as a training (normal) data and the distribution of the three different requests for this normal data set is displayed in Table 1 below.

Table 1: Normal Data Set.

User Requests	Number
composePosts	1526
correct userLogin	420
incorrect userLogin	54
Total	2000

4.2.4 Validation Data

Validation data, also normal data, consists of 300 **composePosts**, 165 correct and 35 incorrect **userLogin** requests giving a total of 500 requests. (Note that this is a higher proportion of incorrect logins than documented in (Brostoff and Sasse, 2003)), but this makes the anomaly detection harder, not easier.) The corresponding validating traces were divided into 10 subsets each of size 50. These subsets were created in order to estimate the mean and standard deviation of the distances (to the normal training data) of other normal data samples.

4.2.5 Attack Data

A third data set was created to simulate a password guessing attack against the application. This anomalous data set has many more incorrect **userLogin** traces than correct ones. The distribution of the different traces in this anomalous data set is displayed in Table 2 below.

Table 2: Anomalous Data Set.

User Requests	Number
composePosts	30
correct userLogin	7
incorrect userLogin	13
Total	50

4.2.6 Experiment

Our experiment with the data sets is outlined as follows. Using the Eq. 1 defined above, we calculated the frequency distribution $f(T_0)$ for all existing traces in the normal training data set. We then applied Equation 2 to the 10 validation data sets to calculate the frequency distribution for each: $f(T_1)$, $f(T_2)$, . . . $f(T_{10})$. Using the distance metric defined in Eq. 3, we calculated the difference in distributions between $f(T_0)$ and each of the validation data distributions $f(T_1)$, $f(T_2)$, ... $f(T_{10})$ and these difference values are displayed in Figure 2 below.

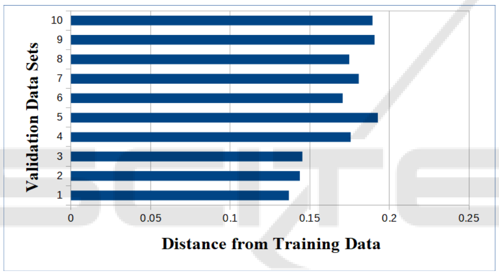

Figure 2: Distance of Validation Data from Training Data.

In the final stage of our experiment, we applied Equation 2 to return the frequency distribution of traces for the anomalous data set denoted as $f(T_{11})$.

4.2.7 Results

We calculated the mean and the standard deviation for the distance values between the frequency distributions of the normal data and each of the ten validating data sets. The mean was 0.1701 and the standard deviation was 0.0208. We set a threshold of two standard deviations above the mean, that is 0.2117, for detecting attack data. We find that the distance of the attack data from the normal data was 0.2171 which is above this threshold. Assuming a normal distribution, we would expect 97.5% of normal validation data to be below this threshold. All ten validating data sets are below this threshold as we would expect. In a real world scenario, a hundred percent prediction like this would of course be unlikely.

4.3 NoSQL Injection Attack

A NoSQL Injection attack is an attack where the hacker gets a software system to behave in a way it never intended to by injecting code that accesses the database into a data field. (Belmar, 2019). For example, an application could allow access to information on a single user with a provided user's ID. A hacker could instead get the app to return all users' data from the database.

In the case of an application using MongoDB, this requires the execution of database query with a WHERE clause. The attacker enters JavaScript code that ends up being executed in the WHERE clause. For example, if the user enters '|| 'a' == 'a' for a username, this can result in the query returning data on all users. In order to make an application vulnerable to such an attack, it is necessary to remove all checks on user input format and ensure that the query is implemented using a WHERE clause. This would not be a recommended way to access a NoSQL database.

To detect this using distributed tracing data, either the call sequence would need to change for the NoSQL Injection request or the duration of the system service calls would need to increase. The call sequence could only be changed if the hacker could change a number of the service implementations. Also notice that the duration of the MongoDb database call is unlikely to change even if a request for a single user object or for all user objects is executed. In the second case, the call would return a resulting database object called a **ResultSet** of fixed batch size and execution time will not be appreciably longer that a request for a single user.

In addition, even if it was possible to obtain a **ResultSet** of User objects from the database, it would not be possible to move them between services due to the well-defined interfaces between the microservices. A **ReadUserInfo** method is constrained to return only a single JSON User object.

The conclusion is that even if it were possible to read a number of users from the database it would not be possible to detect the attack using distributed tracing logs.

5 CONCLUSIONS

This report explored our use of the distributed tracing, in particular the Jaeger implementation, to monitor and log user requests to a microservices-based application. We applied anomaly detection to detect cyber security attacks in the generated log traces. We see that it is possible to use distributed tracing to detect a password guessing cyber security attack. Distributed tracing is typically used to detect performance issues in microservices applications but to the best of our knowledge, distributed tracing has not previously been used to detect cyber security attacks.

In particular, we detected a simulated password guess attack against our application using the generated distributed traces. Due to the fact that a password guessing attack can only be detected by examining a number of requests the technique can be categorized as group-based anomaly detection. We calculated the distribution of normal application request traffic, and compared this distribution to that of the anomalous data. The frequency distribution for the password attack is further from the normal data than the normal validation data sets and using the mean and standard deviation of frequency distance of the validating data sets, the distance from normal data is greater than two standard deviations above the mean. This value is a candidate for an anomaly detection threshold.

We also determined that it is not feasible to detect certain types of cyber-security attacks against a microservices-based application using this approach. We argued that it is not possible to detect a type of NoSQL Injection attack which results in multiple objects being returned from a NoSQL database instead of a single object. This would not result in any substantial changes to the distributed logging data and hence would not be detectable.

6 FURTHER WORK

At the moment, our work is preliminary and only represents the behaviour of a microservice application using sequences of microservice calls. We plan to use call graphs instead of sequences of calls to represent behaviour. Call graphs would be comprised of *nodes* which correspond to microservices, and *edges* corresponding to the calls between the microservices. Graph-related approaches have previously been used to model microservices (Aubet et al., 2018) and detect anomalous performance issues in such applications (Le et al., 2011).

The Euclidean distance metric in Eq. 3 takes no account of the order in which sequences occur. To address this limitation, we intend to train a neural network to learn the normal behaviour of the sequences of call graphs. A Long Short Term Memory (LSTM) deep learning network model is suited to modeling sequential data and identifying long-term dependencies in the sequences. Our LSTM model would be used to assign a probability value to each sequence of call graphs. An anomaly would be triggered if a sequence of call graphs was found to have a lower probability than most sequences. Recent work has demonstrated that LSTM neural networks can learn the behaviour of time-series data and subsequently detect anomalous data (Malhotra et al., 2015) (Nedelkoski et al., 2019). Finally, we will also examine the possibility of strategic attacks designed to circumvent the anomaly detection mechanism and examine ways to prevent these types of attacks.

ACKNOWLEDGEMENTS

This publication has emanated from research conducted with the financial support of Athlone Institute of Technology under its President's Seed Fund (2020) and Science Foundation Ireland (SFI) under Grant Number SFI 16/RC/3918, co-funded by the European Regional Development Fund.

REFERENCES

Aubet, F.-X., Pahl, M.-O., Liebald, S., and Norouzian, M. R. (2018). Graph-based anomaly detection for iot microservices. *Measurements*, 120(140):160.

Belmar, C. (2019). A nosql injection primer (with mongo).

Brostoff, S. and Sasse, M. A. (2003). "ten strikes and you're out": Increasing the number of login attempts can improve password usability. *Human-Computer Interation, Security*.

Chandola, V., Banerjee, A., and Kumar, V. (2009). Anomaly detection: A survey. *ACM computing surveys (CSUR)*, 41(3):1–58.

Dhanabal, L. and Shantharajah, S. (2015). A study on nsl-kdd dataset for intrusion detection system based on classification algorithms. *International Journal of Advanced Research in Computer and Communication Engineering*, 4(6):446–452.

Gan, Y. and Delimitrou, C. (2018). The architectural implications of cloud microservices. *IEEE Computer Architecture Letters*, 17(2):155–158.

Gan, Y., Zhang, Y., Cheng, D., Shetty, A., Rathi, P., Katarki, N., Bruno, A., Hu, J., Ritchken, B., Jackson, B., et al. (2019a). An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18.

Gan, Y., Zhang, Y., Hu, K., Cheng, D., He, Y., Pancholi, M., and Delimitrou, C. (2019b). Leveraging deep learning to improve performance predictability in cloud microservices with seer. *ACM SIGOPS Operating Systems Review*, 53(1):34–39.

Gan, Y., Zhang, Y., Hu, K., Cheng, D., He, Y., Pancholi, M., and Delimitrou, C. (2019c). Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 19–33.

Le, D. Q., Jeong, T., Roman, H. E., and Hong, J. W.-K. (2011). Traffic dispersion graph based anomaly detection. In *Proceedings of the Second Symposium on Information and Communication Technology*, pages 36–41.

Malhotra, P., Vig, L., Shroff, G., and Agarwal, P. (2015). Long short term memory networks for anomaly detection in time series. In *Proceedings*, volume 89, pages 89–94. Presses universitaires de Louvain.

Nagothu, D., Xu, R., Nikouei, S. Y., and Chen, Y. (2018). A microservice-enabled architecture for smart surveillance using blockchain technology. In *2018 IEEE International Smart Cities Conference (ISC2)*, pages 1–4. IEEE.

Nedelkoski, S., Cardoso, J. S., and Kao, O. (2019). Anomaly detection and classification using distributed tracing and deep learning. In *CCGRID*, pages 241–250.

Sun, Y., Nanda, S., and Jaeger, T. (2015). Security-as-a-service for microservices-based cloud applications. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 50–57. IEEE.

Toth, E. and Chawla, S. (2018). Group deviation detection methods: A survey. *ACM Computing Surveys (CSUR)*, 51(4):1–38.