Università degli Studi di Milano - Bicocca

Department of Computer Science, Systems and Communication

Data Science Master's Degree

Unsupervised Machine Learning for Intrusion Detection Systems

Supervisor: Prof. Denaro Giovanni

Thesis dissertation of:
Gianmarco Russo
Matricola 887277

Academic Year 2022-2023

1 Acknowledgements

I would like to express my sincere gratitude to my supervisor, Professor Giovanni Denaro, for his invaluable guidance, support, and patience throughout my graduate studies. His insightful comments, constructive feedback, and unwavering encouragement have been instrumental in shaping this research work.

I am grateful to my friends and classmates for their camaraderie, discussions, and moral support. The exchange of ideas and intellectual conversations have enriched my understanding of the subject matter.

I would also like to acknowledge the assistance of two PhD students, Rahim Heydarov(PhD @USI) and Alessandro Tundo(PhD @Unimib) who provided technical support and helped me overcome various obstacles during my research work.

I would like to acknowledge all the individuals, too numerous to mention individually, who have provided assistance, advice, or encouragement at various stages of this thesis. Your contributions, big and small, have shaped this work and my academic journey.

Special mention to my girlfriend Mafalda, who supported me in the toughest moments with her unconditional support and positivity. Your smile brightened even the darkest of days.

Finally, I would like to express my heartfelt appreciation to my family for their unwavering support, understanding, and encouragement throughout my academic journey. Their love, encouragement, and faith in me have been a constant source of motivation and inspiration.

List of Figures

Index

Chapter 1

Introduction

1 Motivation

Cybercriminals exploit vulnerabilities in the source code of web applications for unauthorized access to databases and resources in web servers by leveraging different web-based attacks (e.g., Cross-Site Scripting and SQL Injection) to accomplish their malicious goals. Although security researchers have extensively investigated anomaly detection of web-based attacks, the cloud-native paradigm shift combined with the increasing usage of microservices – an architectural pattern that defines an application as a collection of independent services – introduces new challenges and opportunities. Users expect modern web applications to please their needs fast and reliably regardless of their device, geographical location, or time. Developers exploit the latest technology to meet users' expectations, and in this sense, the use of cloud services and scalable solutions combined with modular (microservices) and distributed architectures are the preferred solutions. With microservices becoming the fundamental blocks of modern web applications, studying the communications between those elements and the realtionship between services and nodes resources enables learning the workflow initiated from incoming traffic and, consequently, modeling an expected behavior for future runtime behaviour.

2 Problem Statement

This thesis explores anomaly detection of web-based attack on microservices based applications by modeling application performance metrics and service logs. The general idea is that a *normal activity* profile can be built upon the (simulated) normal activity on the web application and then the anomalies such as web attacks can be detected as *different behaviour* with respect to the normal activity. This task will be carried out by generating a dataset only containing normal activity and then train machine learning models to distinguish between the learnt behaviour and different behaviours.

3 Contribution

This thesis aims to design and develop an effective method to detect web-based attacks in a microservices architecture. At the moment, most research on anomaly detection of web-based attacks is targeting monolithic web applications without adequate consideration of the paradigm shift towards cloud computing. Moreover, the material available

doesn't experiment on realistic web applications but on simple applications used as proof of concept, mostly considering network traffic as training data. The application used as our testing environment is quite complex(40+ microservices) and has been deployed on a real, on premise kubernetes cluster.
The design and evalutation of machine learning models will be performed in an unsupervised manner, which is another point of contribution. This choice has been made to reflect the lack of labeled data in real intrusion detection scenarios, often having to deal with unlabelled or partially labelled data. The approach consists in combining **two** sources of information:

1. **Performance metrics**: cpu utilization, memory usage, network volume etc..
2. **Application logs**: messages produced by every application pod regarding its behaviour.

and combining them to try to detect cybersecurity attacks targeting the application.

The **contributions** can be summarized as follows:

- Deployment of a **complex microservice application**.
- Usage of **log** production as added **features**.
- Design of an **unsupervised approach** to the problem.

4 Structure of the thesis

The rest of the thesis is organized as follows:

1. **Chapter 2**: introduction of the topics discussed in this work, from intrusion detection systems to Kubernetes
2. **Chapter 3**: brief analysis of the existing literature on the subject.
3. **Chapter 4**: application deployment, monitoring and log collection solutions, dataset generation.
4. **Chapter 5**: malicious activity detection using unsupervised methods.
5. **Chapter 6**: conclusions and future developments.
6. **Bibliography**.

Chapter 2

Research Context

1 IDS

Intrusion Detection Systems (IDS) are a critical component in securing computer networks against potential cyber-attacks. An IDS monitors network traffic for suspicious or malicious activity and alerts security personnel to potential threats. IDS fall in two different categories:

- **Signature based IDS**
- **Machine Learning based IDS**

1.1 Signature Based IDS

Signature-based intrusion detection (IDS) is a technique used to detect and prevent malicious activities on a network by searching for specific patterns, or signatures, in network traffic[14]. This technique is based on pre-defined signatures of known attack patterns.
Signature-based IDS works by analyzing network traffic and comparing it to a database of known attack signatures. If the IDS detects traffic that matches a known signature, it can generate an alert, block the traffic, or take other actions based on the configuration of the system[12]. Signature-based IDS is often used to detect and prevent attacks such as viruses, worms, and other types of malware.
The **advantages** of signature-based IDS include:

- **Accuracy**: Signature-based IDS can be very accurate at detecting known attacks, as it is designed to identify specific patterns of malicious activity.
- **Speed**: Signature-based IDS can quickly detect known attacks, making it useful for identifying and stopping threats in real-time.
- **Easy to use**: Signature-based IDS is easy to use and does not require extensive knowledge of network security.

However, there are also some **limitations** to signature-based IDS:

- **Limited coverage**: Signature-based IDS can only detect attacks that match known signatures. New and unknown attacks can easily bypass the system.
- **False positives**: Signature-based IDS can generate false positives if legitimate traffic matches a known signature.

- **Maintenance**: Signature-based IDS requires regular updates to the signature database to keep up with new attack patterns.

In summary, signature-based IDS is a useful technique for detecting known attacks and preventing them from causing damage on a network[17]. However, it should not be relied on as the only line of defense against malicious activity, and should be used in conjunction with other security measures to provide a comprehensive security solution.

1.2 Machine Learning IDS

Machine learning-based intrusion detection (IDS) is an approach that **uses machine learning algorithms to** analyze network traffic and **detect malicious activity based on patterns and anomalies in the data**. Unlike signature-based IDS, machine learning-based IDS does **not rely on pre-defined attack signatures**[13]. Instead, it uses machine learning algorithms to analyze large amounts of data and identify patterns that may indicate a security threat.

Machine learning-based IDS works by using algorithms to learn from historical network traffic data and develop a baseline of normal behavior. The system can then compare current network traffic to this baseline and identify anomalies that may indicate a security threat[10]. Machine learning-based IDS can also be used to detect new and unknown threats that have not been seen before.

The **advantages** of machine learning-based IDS include:

- **Improved coverage**: Machine learning-based IDS can detect unknown attacks and threats that signature-based IDS cannot detect.
- **Reduced false positives**: Machine learning-based IDS can reduce the number of false positives by learning to distinguish between normal and abnormal behavior on the network.
- **Scalability**: Machine learning-based IDS can be highly scalable and can analyze large amounts of data in real-time.

However, there are also some **limitations** to machine learning-based IDS:

- **Complexity**: Machine learning-based IDS can be complex to implement and maintain, as it requires expertise in machine learning algorithms and data analysis.
- **Data quality**: Machine learning-based IDS requires high-quality data to be effective. Poor quality data can result in inaccurate results and false positives.
- **Resource-intensive**: Machine learning-based IDS can be resource-intensive, requiring significant computing power and storage capacity to analyze large amounts of data.

In summary, machine learning-based IDS is a powerful approach to detecting security threats that can provide improved coverage and reduced false positives compared to signature-based IDS. However, it requires significant expertise and resources to implement and maintain effectively.

In recent years, machine learning-based IDS has become an active research area, with a growing number of studies and experiments exploring the use of different machine learning techniques for intrusion detection. However, while machine learning-based IDS shows

promise in improving the security of computer networks, there are also challenges associated with its implementation, such as the need for large and diverse training datasets, the risk of false positives and false negatives, and the potential for attackers to manipulate or evade the system.
Despite these challenges, machine learning-based IDS represents an exciting frontier in the field of cybersecurity, and continued research and development in this area will be crucial for improving the detection and prevention of cyber-attacks.

2 Microservices

Microservices is an **architectural approach** to building software applications by breaking them down into small, **independent services that communicate with each other** through APIs. Each microservice is designed to perform a specific business function and can be developed, deployed, and maintained independently. Microservices architecture **allows for greater flexibility, scalability, and resilience in application development**.[7]

Microservices architecture has become popular in recent years due to its ability to address the challenges associated with monolithic architectures, such as complexity, scalability, and maintenance. By breaking down an application into small, self-contained services, developers can focus on specific areas of the application without worrying about how their changes will affect other parts of the system. This allows for faster development, easier testing, and quicker deployment of new features.

Microservices can be developed using different programming languages and technologies, and can be deployed using containerization technologies such as Docker and Kubernetes. Each microservice can be scaled independently, allowing for greater flexibility and better utilization of resources.

While microservices architecture offers many benefits, it also comes with some challenges. **Managing a large number of microservices can be complex**, and ensuring communication between them is secure and reliable can be difficult. Additionally, testing and debugging distributed systems can be more challenging than monolithic systems.

Overall, microservices architecture is a powerful approach to building scalable, flexible, and resilient software applications. It has become a popular choice for many organizations looking to build modern, cloud-native applications.

2.1 Containerization

Containerization is not a required part of the microservices concept, but it is often **used as a technology to support microservices architecture**.

Microservices architecture focuses on breaking down a large monolithic application into smaller, independent services that can be developed, deployed, and managed separately. Each microservice typically performs a specific business function and communicates with other microservices via APIs.

Containerization is a technology that allows for the efficient and reliable deployment and management of software applications. **Containers are lightweight, portable, and self-contained environments that can run applications and their dependencies consistently across different environments[9]**. This makes it easier to move applications between development, testing, and production environments and simplifies the deployment process.

Using containers to deploy microservices can offer several benefits. Containers **allow microservices to be deployed independently**, which means that developers can make changes to one microservice without affecting others. Containers also **provide a consistent and isolated environment for each microservice**, making it easier to manage dependencies and ensuring that each microservice has the resources it needs to run efficiently. Additionally, containers can be easily scaled horizontally to handle increasing traffic or demand.

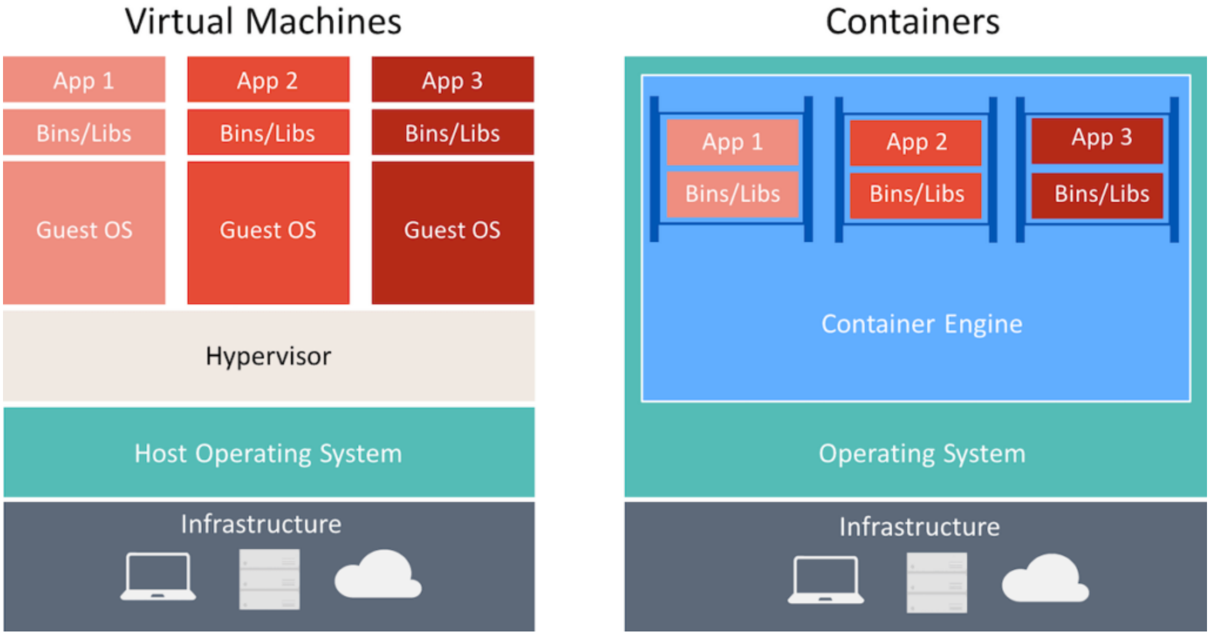


Figure 2.1: Virtual machines vs Containers

2.2 Kubernetes

Kubernetes is an open-source **container orchestration platform that automates the deployment, scaling, and management of containerized applications**[2]. Originally developed by Google, Kubernetes is now maintained by the Cloud Native Computing Foundation (CNCF) and has become a popular tool for deploying and managing containerized applications.

Kubernetes **allows developers to abstract away the underlying infrastructure and focus on building and deploying applications using container technology**. It provides a declarative API for defining the desired state of an application, and then manages the deployment and scaling of containers to meet that desired state. Kubernetes can also perform rolling updates and rollbacks, ensuring that applications are always available and up-to-date.

One of the key features of Kubernetes is its ability to scale applications dynamically based on demand. Kubernetes can automatically scale the number of containers running an application up or down based on CPU usage, memory usage, or other metrics. This allows applications to scale quickly and efficiently, and ensures that resources are utilized optimally.

Kubernetes also provides many features for managing the health and availability of applications. It can perform automatic failover in the event of a container or node failure, and can also perform self-healing by restarting containers that have failed. Kubernetes also provides built-in load balancing, allowing traffic to be routed to the most appropriate container.

Kubernetes can be used with a variety of container runtimes, including Docker and CRI-O, and can be run on-premises or in the cloud. It integrates with many popular tools and platforms, including monitoring and logging tools, and has a large and active community that contributes to its development and maintenance.

Overall, Kubernetes is a powerful tool for managing containerized applications at scale. It provides a flexible and reliable platform for deploying and managing modern cloud-native applications.

Kubernetes Resources

Here are the most common and used Kubernetes resources[4]:

- **Pod**: A pod is the smallest deployable unit in Kubernetes and represents a single instance of a running process in a cluster.
- **Deployment**: A deployment manages a set of replicas of a pod and allows you to scale the number of replicas up or down, roll out updates, and roll back to previous versions.
- **Service**: A service provides a stable IP address and DNS name for a set of pods, allowing other pods and services to communicate with them.
- **ConfigMap**: A ConfigMap allows you to store configuration data as key-value pairs and inject that configuration into pods as environment variables or as files in a volume.
- **Secret**: A Secret allows you to store sensitive data, such as passwords, keys, and tokens, and inject that data into pods as environment variables or as files in a volume.
- **Volume**: A volume provides a way to store data persistently or share data between containers in a pod.
- **Namespace**: A namespace provides a way to partition resources in a cluster and organize them by teams, projects, or environments.

These are just some of the resources that Kubernetes provides. There are many more resources available, and you can even create custom resources to extend Kubernetes functionality.

Kubernetes Cluster

The main components of a Kubernetes cluster[19] are:

- **Master node(s)**: The master node is the control plane of the cluster, responsible for managing the overall state of the system. It includes several components, such as the API server, etcd, scheduler, and controller manager.

- **Worker node(s)**: The worker node is the worker machine that runs the containers for the applications. It includes the kubelet, which communicates with the API server to receive instructions for running containers, and the container runtime, such as Docker.
- **etcd**: etcd is a distributed key-value store that stores the configuration and state information of the cluster. It is used by the API server, controller manager, and scheduler to coordinate cluster state
- **API server**: The API server is the front-end for the Kubernetes control plane. It exposes the Kubernetes API, which enables administrators to interact with the cluster and manage its resources.
- **Kubelet**: The kubelet is an agent that runs on each worker node and is responsible for managing the containers running on the node. It communicates with the API server to receive instructions for running containers and ensures that the containers are running as expected.
- **Container runtime**: The container runtime is the software that runs the containers on the worker nodes. Docker is a commonly used container runtime, but other runtimes such as CRI-O and containerd are also supported by Kubernetes.
- **Scheduler**: The scheduler is responsible for determining which worker node a new pod should be scheduled on based on resource availability and other factors.
- **Controller manager**: The controller manager is responsible for managing the different controllers in the cluster, such as the replication controller, which ensures that the desired number of replicas of a pod are running at all times.

There are also other optional components that can be added to Kubernetes cluster, such as a load balancer or a monitoring system, depending on the needs of the application.

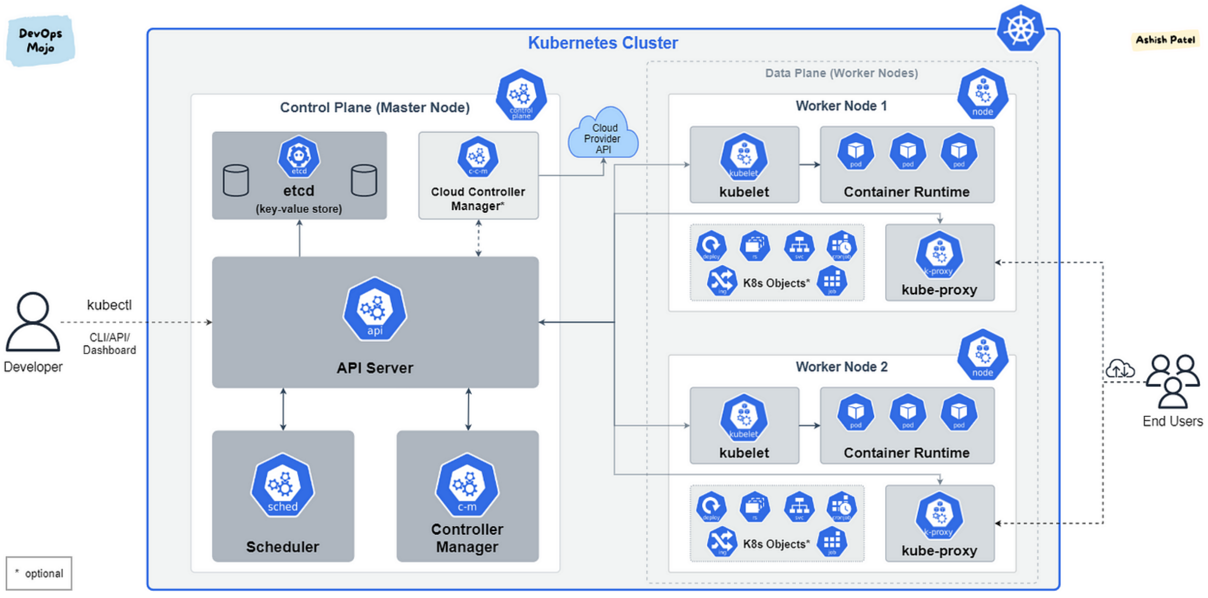


Figure 2.2: Kubernetes cluster components

Chapter 3

State of the art

Different datasets and research articles regarding machine learning based IDS have been published in recent years. This section aims to summarize the contribution and the value added by the following studies, while highlighting what is still open in research.

1 Research Overview

Research in this field has already explored machine learning and deep learning algorithms for several intrusion detection use cases:

- *Alom et al*[3](2017) presents an in-depth analysis of various unsupervised deep learning architectures applied to network intrusion detection, including stacked autoencoders, variational autoencoders, and GANs. They discuss the advantages and challenges associated with each approach, emphasizing the importance of data preprocessing, model selection, and hyperparameter tuning to achieve optimal results.

- *Soheily-Khah et al*[16](2018) addresses the problem of intrusion detection in network systems using a hybrid approach that combines supervised and unsupervised machine learning techniques. The authors conduct a case study on the ISCX dataset to evaluate the effectiveness of their proposed approach.

- *Chiba et al*[6](2019) focuses on developing an intelligent approach for building an Intrusion Detection System (IDS) specifically designed for cloud environments. The authors propose a deep neural network (DNN) as the core component of their IDS architecture and combine it with multiple machine learning algorithms to enhance its performance.

- *Elsayed et al*[8](2020) discusses the evaluation of the proposed LSTM-based autoencoder on real-world network datasets. The authors compare its performance with traditional anomaly detection methods, such as Support Vector Machines (SVM) and Random Forests. Evaluation metrics, including accuracy, precision, recall, and F1-score, are used to assess the model's effectiveness in detecting network anomalies.

- *Verkeren et al.*[18](2021) presents an in-depth analysis of various unsupervised machine learning techniques that can be applied to intrusion detection. These techniques include clustering algorithms, anomaly detection methods, and generative models. The paper discusses the advantages, limitations, and considerations associated with each technique, emphasizing the importance of selecting an appropriate method based on the specific requirements and characteristics of the intrusion detection system.The authors compare the performance of different techniques in terms of their ability to detect known attacks, detect unknown attacks, and minimize false positives. Evaluation metrics such as accuracy, precision, recall, and F1-score are used to assess the effectiveness of the models.

- *Kilincer et al.*[11](2021) presents a comparative study of various machine learning algorithms, including decision trees, support vector machines (SVM), random forests, naive Bayes, k-nearest neighbors (k-NN), and deep learning models for intrusion detection. The authors evaluate the algorithms using different performance metrics, such as accuracy, precision, recall, and F1-score, to assess their effectiveness in detecting intrusions.
- *Chen et al.*(2020)[5] presents the results of applying the CNN-based intrusion detection system to real-world network traffic datasets. The authors compare the performance of their proposed system with traditional machine learning methods and other deep learning architectures. Evaluation metrics such as accuracy, precision, recall, and F1-score are used to assess the system's effectiveness in detecting network intrusions. The results demonstrate that the CNN-based intrusion detection system outperforms traditional methods and achieves competitive performance compared to other deep learning architectures.
- *Sethi et al.*(2020)[15] provides a detailed explanation of the architecture and design of the DRL-based intrusion detection system. The system consists of an agent that interacts with the cloud environment, observes the state of the system, and takes actions to detect and mitigate intrusions. The results demonstrate that the DRL-based intrusion detection system outperforms traditional methods and achieves competitive performance compared to other machine learning approaches. The adaptive and learning capabilities of the system contribute to its ability to detect and respond to complex and evolving intrusions in cloud environments.

1.1 Limitations

As we can clearly deduct by the aforementioned research papers it seems that most of the possibilities model-wise have been explored and evaluated. What really seems to be lacking in research (to the author's knowledge at the time of writing this piece of work, at least) is the focus on distributed systems and in particular microservice based architecture. Most of the research results mentioned **don't target specific applications** but **network topologies** in general, evaluating **attacks targeting physical machines instead of services**.

2 Available IDS datasets

The most popolar and most used IDS dataset can be summarized in the following table:
All of these datasets offer similar features, mostly information about packets traveling through the network like:

Dataset	Attack Types	Labels	Attributes	Network Environment
KDD-99	DoS, Probing Attacks, R2L, U2R	Yes	41	Conventional Network
NSL-KDD	DoS, Probing Attacks, R2L, U2R	Yes	41	Conventional Network
Kyoto Dataset 2006	Not specified, various attacks to honeypots	Yes	24	Conventional Network
ISCX2012	HTTP DoS, DDoS, SSH bruteforce, infiltration(through metasploit)	Yes	Not Known	Conventional Network
CIC-IDS2017	Botnet, Dos, DDoS, XSS, SQLi, SSH bruteforce, portscan	Yes	83	Conventional Network
CIC-IDS2018	Botnet, Dos, DDoS, XSS, SQLi, SSH bruteforce, portscan	Yes	83	AWS Platform
InSDN	Botnet, Dos, DDoS, web attacks, password bruteforce, portscan	Yes	83	SDN Network

Table 3.1: Overview of existing datasets

- Source and destination IP addresses
- Protocol
- Port number
- Headers
- Payload length

As shown in the dataset overview table all these dataset have common characteristics like labeled instances, homogenous network topologies and attack types.
This thesis wants to focus the research on the generation of an IDS dataset based on a microservice application, with application level attacks, and then use it to train and test machine learning algorithms in the detection of these attacks.
Moreover, these datasets are mainly focused on network communication. That is why our approach wants to tackle the problem using different data, as introduced in the first chapter, like performance metrics and aggregate application logs[1] instead of network communication.
The above datasets are also labeled, allowing for the usage and evaluation of supervised machine learning models, whereas this approach wants to focus on unsupervised models.

Chapter 4

Testbed Setup and Dataset Generation

As anticipated in the first chapter, the contributions of this thesis can be summarized in **three** points:

1. Deployment of a **complex microservice application**.
2. Usage of **log** production as added **features**.
3. Design of an **unsupervised approach** to the problem.

This section illustrates point one and two:

- **Section 1** describes the deployment of the microservice benchmark application.
- **Section 2** reports how the proposed dataset has been generated and collected.
- **Section 3** summarizes the characteristics of the aforementioned dataset.
- **Section 4** focuses on test data and on the attacks that have been performed.

1 Benchmark Application

The application chosen for our experiment is a microservices application called *Train Ticket*, developed by Fudan University. The application offers the functionalities of a train ticket purchase website. It is composed of **41 microservices** and can be deployed with **Kubernetes**, with the additions of monitoring and distributed tracing solutions if needed.
The exagons represent the services, with links defining the interaction between them.
On the left we have some utility services:

- **NACOS** is an open-source dynamic service discovery, configuration, and service management platform. It stands for "Naming and Configuration Server." NACOS provides a centralized solution for service registration, service discovery, dynamic configuration, and service health management in a distributed system or microservices architecture.
- **Sentinel** is responsible for flow control: it refers to the ability to limit the number of requests or transactions allowed to access a specific resource or service within a certain time frame. It helps prevent the system from being overwhelmed by excessive traffic, ensuring stability, and protecting against performance degradation or failures

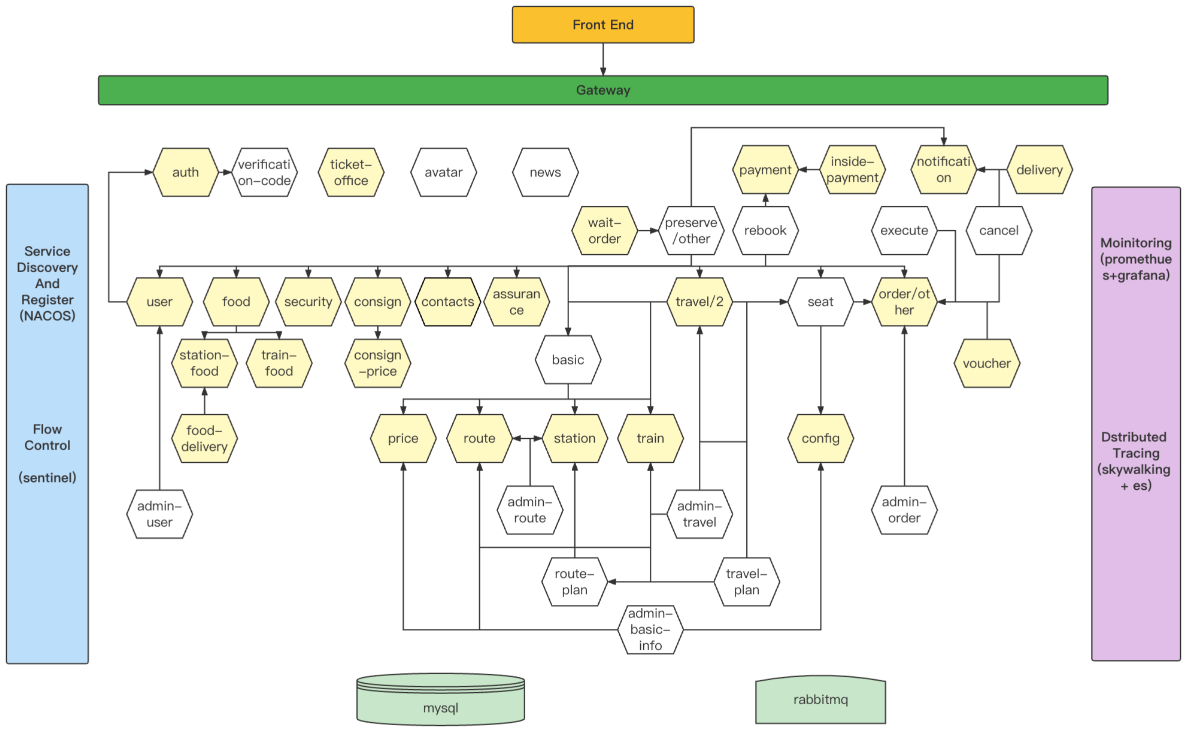


Figure 4.1: Train ticket architecture

While on the right we have the monitoring and tracing solutions:

- **Prometheus** is an open-source monitoring and alerting toolkit that is widely used for monitoring systems and services in a distributed environment.

- **Apache SkyWalking** is an open-source observability and application performance monitoring (APM) system designed for distributed systems and microservices architectures. It helps developers monitor, trace, and diagnose the performance of their applications and services.

At the bottom we have the storage section:

- **MySQL** is a popular open-source relational database management system (RDBMS) that is widely used for managing and storing structured data.

- **RabbitMQ** is a message broker software that facilitates asynchronous communication and messaging between applications and services. It acts as a middleman or mediator, enabling different components of a distributed system to exchange messages in a decoupled manner.

1.1 Deployment

Testbed

The application has been deployed in an on-premise environment, composed of 4 virtual machines. Each node has the following **specifications**:

1. 8-core CPU,
2. 16GB RAM,
3. 120GB HDD,
4. Ubuntu Server 20.04.

The repository guide suggests at least four nodes with the same configuration. In this experiment **one node** has been set as **master node** while the remaining **three** were set as **worker nodes**. The master node is solely a master and does not also function as a worker node.

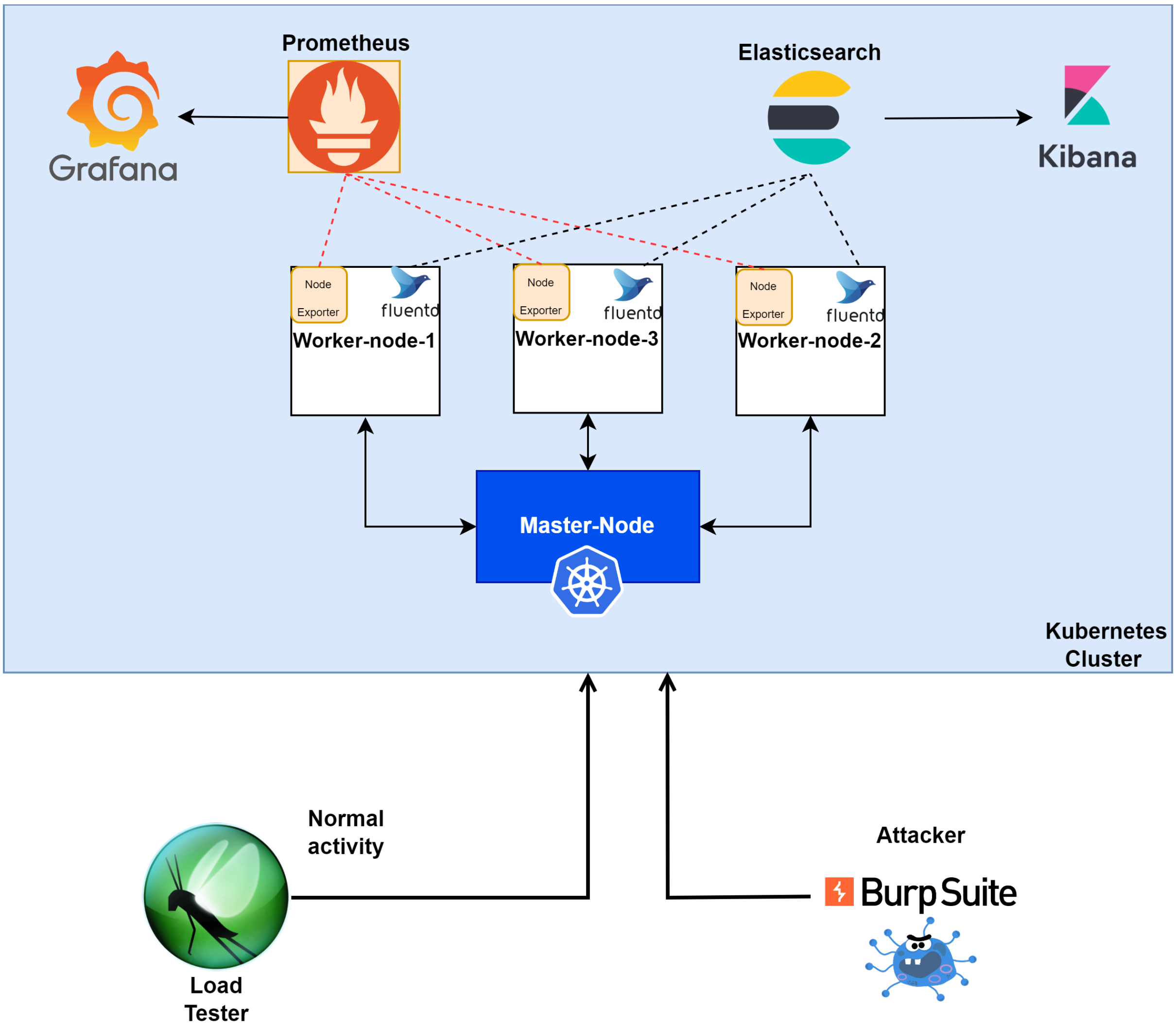


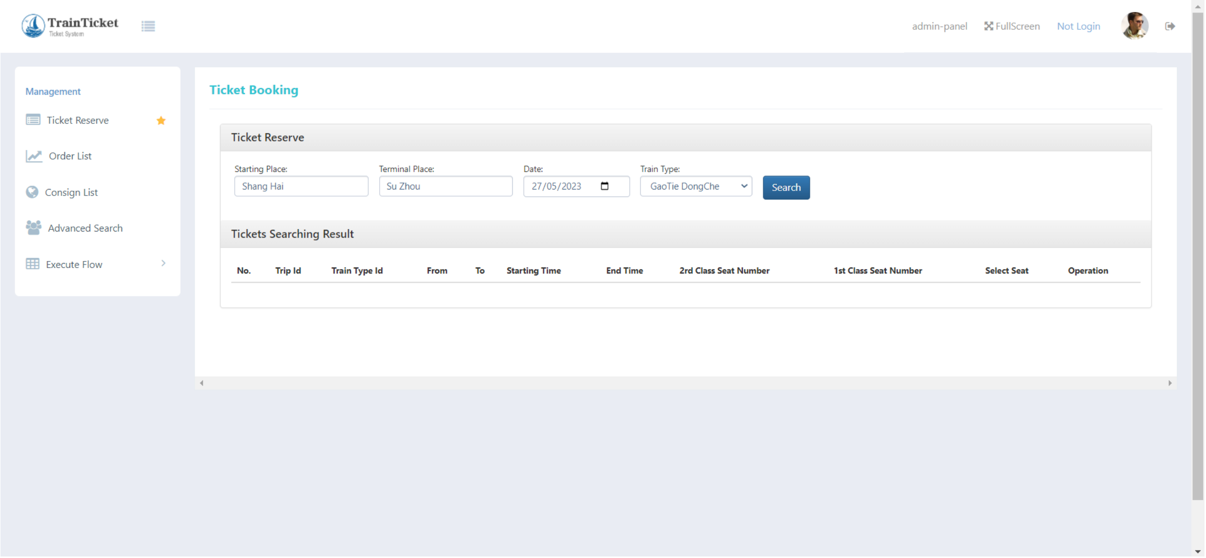
Figure 4.2: Testbed complete architecture

We are going to see all the cluster components(app deployment, prometheus and elastic search) in the following sections.

Train Ticket Deployment

As highlighted in the repository guide there two main deployment options: one using **Docker**(through the compose command) and one with **Kubernetes**. All the yaml

configuration files are provided and ready to use if the requirements in the guide are satisfied(pre-existing k8s cluster, helm and local Persistent Volume(PV) support).
The first step before proceding with the deployment is to have a kubernetes cluster up and running: to simplify the cluster setup a tool called *Kubeadm* has been used.
Once the node are set up all we need to do is to deploy the application using the make file provided (with the options of adding monitoring and distributed tracing solutions if wanted) on the master node. This node will be in charge of distributing the services across the three worker nodes.



Ticket Booking

Ticket Reserve

Starting Place: Shang Hai
Terminal Place: Su Zhou
Date: 27/05/2023
Train Type: GaoTie DongChe

Tickets Searching Result

No.	Trip Id	Train Type Id	From	To	Starting Time	End Time	2rd Class Seat Number	1st Class Seat Number	Select Seat	Operation

Figure 4.3: Train ticket homepage

2 Dataset Generation

2.1 Prometheus Monitoring

To be available to collect data we need to setup a monitoring solution. In this scenario Prometheus has been used to scrape and collect performance metrics node-wise, mostly hardware, with the likes of:

- CPU utilization
- Disk I/O operations
- Memory allocation
- Network volume traffic

Prometheus is an open-source monitoring and alerting system that is widely used in the industry to **monitor the health and performance of applications and infrastructure**. It was originally developed by SoundCloud and is now maintained by the Cloud Native Computing Foundation (CNCF).

Prometheus is designed to collect time-series data from a variety of sources, including applications, services, and systems. It stores this data in a highly efficient, compressed format that allows for fast and efficient querying and analysis. Prometheus also includes a powerful query language called PromQL, which allows users to explore and visualize their data in real-time.

Some of the key features of Prometheus include:

- **Multi-dimensional data model**: Prometheus uses a powerful multi-dimensional data model, which allows users to analyze and visualize their data across multiple dimensions, including time, hostname, and application.
- **Flexible querying**: Prometheus includes a flexible query language called PromQL, which allows users to create complex queries and visualizations of their data.
- **Alerting**: Prometheus includes a robust alerting system, which allows users to define rules for detecting and alerting on abnormal behavior or performance issues.
- **Service discovery**: Prometheus includes a powerful service discovery mechanism, which can automatically discover and monitor new services as they are added to a system.
- **Exporters**: Prometheus supports a variety of exporters, which allow users to monitor third-party systems and applications, such as databases, message brokers, and load balancers.

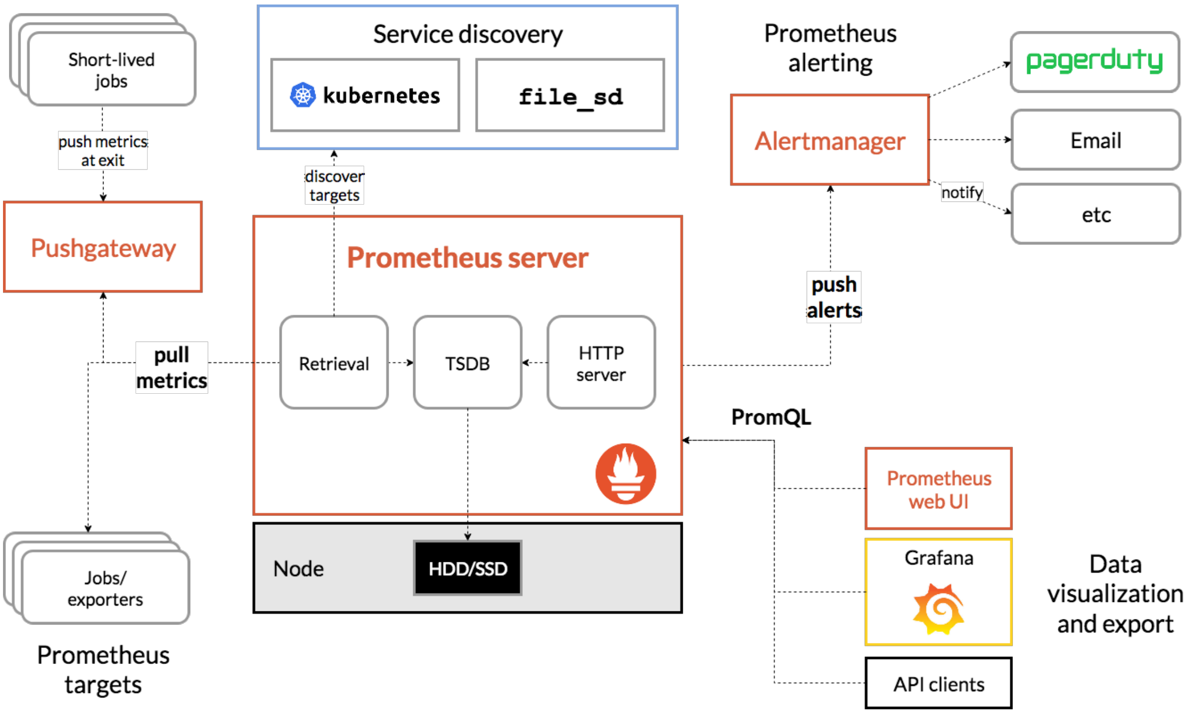
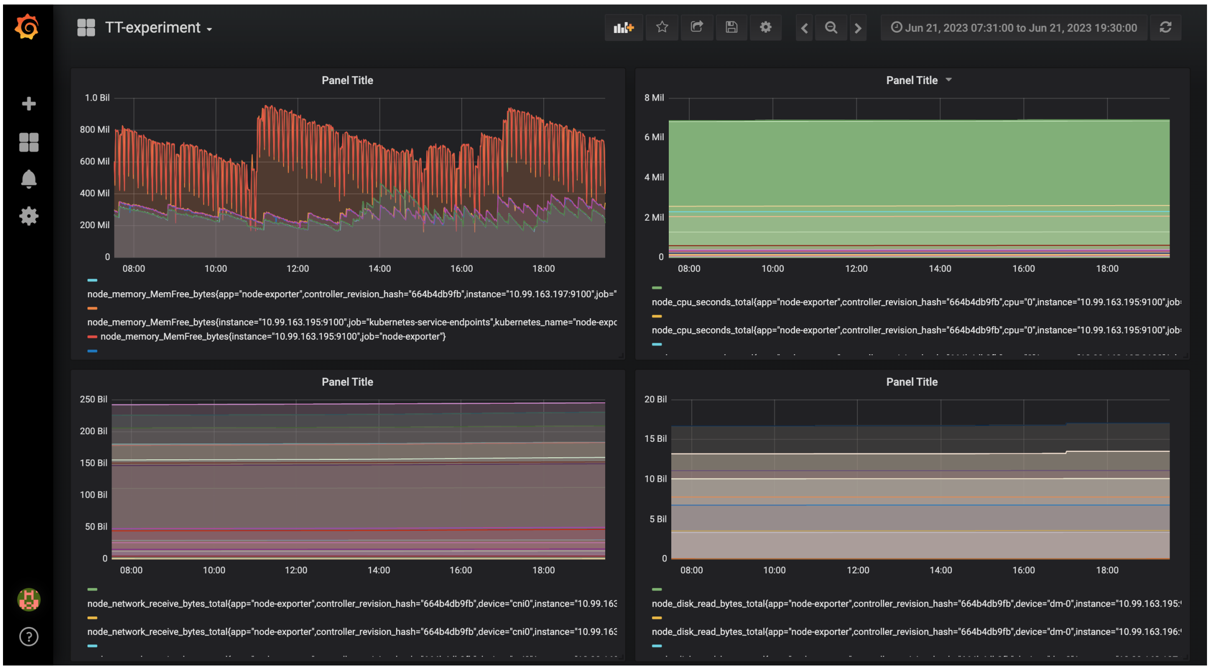


Figure 4.4: Prometheus Architecture

Overall, Prometheus is a powerful and flexible monitoring and alerting system that provides users with deep insights into the performance and health of their applications and systems. Its open-source nature and active community make it a popular choice for organizations of all sizes.

Once the dashboard have been created, the data is available to be collected and monitored. Grafana provides a useful functionality to export data directly in CSV format for further analysis.



```
node_memory_MemFree_bytes{app="node-exporter",controller_revision_hash="664b4db9fb",instance="10.99.163.197:9100",job="
node_memory_MemFree_bytes{instance="10.99.163.195:9100",job="kubernetes-service-endpoints",kubernetes_name="node-expc
node_memory_MemFree_bytes{instance="10.99.163.195:9100",job="node-exporter"}
```

```
node_cpu_seconds_total{app="node-exporter",controller_revision_hash="664b4db9fb",cpu="0",instance="10.99.163.195:9100",job:
node_cpu_seconds_total{app="node-exporter",controller_revision_hash="664b4db9fb",cpu="0",instance="10.99.163.195:9100",job:
```

```
node_network_receive_bytes_total{app="node-exporter",controller_revision_hash="664b4db9fb",device="cni0",instance="10.99.163
node_network_receive_bytes_total{app="node-exporter",controller_revision_hash="664b4db9fb",device="cni0",instance="10.99.163
```

```
node_disk_read_bytes_total{app="node-exporter",controller_revision_hash="664b4db9fb",device="dm-0",instance="10.99.163.195:
node_disk_read_bytes_total{app="node-exporter",controller_revision_hash="664b4db9fb",device="dm-0",instance="10.99.163.196:
```

Figure 4.5: Grafana dashboard

2.2 Elasticsearch

The EFK stack, also known as the Elastic-Fluentd-Kibana stack, is an alternative to the ELK stack that replaces Logstash with Fluentd. Here's a brief introduction to the EFK stack:

- **Elasticsearch**: Elasticsearch remains the core component of the EFK stack. It **is a distributed search and analytics engine** that stores and indexes data for fast retrieval. Elasticsearch provides powerful full-text search capabilities and real-time data analysis.

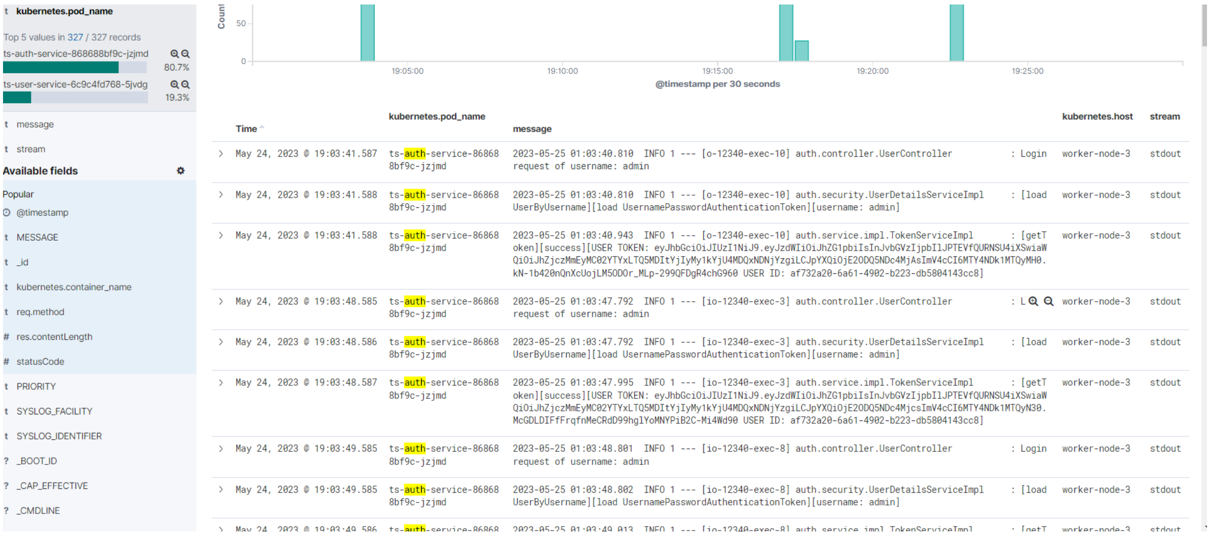
- **Fluentd**: Fluentd **is a data collection and processing framework that acts as the data pipeline** in the EFK stack. It collects log data from various sources, such as application logs, system logs, or any other log files, and then routes and transforms the data to the desired destinations. Fluentd offers flexible configuration options and supports numerous input and output plugins, making it highly versatile.

- **Kibana**: Kibana serves as the **data visualization and exploration tool** in the EFK stack, similar to its role in the ELK stack. It provides a web-based interface that allows users to interact with the indexed data stored in Elasticsearch. With Kibana, you can create visualizations, dashboards, and perform ad-hoc queries to gain insights from your log data.

The EFK stack, leveraging Fluentd as the log collector, offers an alternative to Logstash in the ELK stack. Fluentd is known for its lightweight and efficient nature, making it

suitable for high-performance log collection and processing. It provides extensive configuration options, allowing you to customize log parsing and routing based on your specific requirements.
Overall, the EFK stack **provides a scalable and efficient solution for log management, analysis, and visualization**. It is widely used for centralized logging, real-time monitoring, and troubleshooting in various applications and systems.

In this implementation elasticsearch has been used to collect the application logs, specifically the logs coming from each pod. Logs are provided by the pods using the stdout/stderr streams by default, so fluentd is responsible of capturing these streams and forward them to elasticsearch.



```
                              kubernetes.pod_name                                                                                      kubernetes.host  stream
Time                                                message
May 24, 2023 @ 19:03:41.587  ts-auth-service-86868  2023-05-25 01:03:40.810  INFO 1 --- [o-12340-exec-10] auth.controller.UserController        : Login   worker-node-3  stdout
                             8bf9c-jzjmd            request of username: admin
May 24, 2023 @ 19:03:41.588  ts-auth-service-86868  2023-05-25 01:03:40.810  INFO 1 --- [o-12340-exec-10] auth.security.UserDetailsServiceImpl  : [load   worker-node-3  stdout
                             8bf9c-jzjmd            UserByUsername][load UsernamePasswordAuthenticationToken][username: admin]
May 24, 2023 @ 19:03:41.588  ts-auth-service-86868  2023-05-25 01:03:40.943  INFO 1 --- [o-12340-exec-10] auth.service.impl.TokenServiceImpl    : [getT   worker-node-3  stdout
                             8bf9c-jzjmd            oken][success][USER TOKEN: eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJhZG1pbiIsInJvbGVzIjpbIlJPTEVfQURNSU4iXSwiaW
                                                    QiOiJhZjczMmEyMC02YTYxLTQ5MDItYjIyMy1kYjU4MDQxNDNjYzgiLCJpYXQiOjE2ODQ5NDc4MjAsImV4cCI6MTY4NDk1MTQyMH0.
                                                    kN-1b420nQnXcUojLM5ODOr_MLp-299QFDgR4chG960 USER ID: af732a20-6a61-4902-b223-db5804143cc8]
May 24, 2023 @ 19:03:48.585  ts-auth-service-86868  2023-05-25 01:03:47.792  INFO 1 --- [io-12340-exec-3] auth.controller.UserController        : L       worker-node-3  stdout
                             8bf9c-jzjmd            request of username: admin
May 24, 2023 @ 19:03:48.586  ts-auth-service-86868  2023-05-25 01:03:47.792  INFO 1 --- [io-12340-exec-3] auth.security.UserDetailsServiceImpl  : [load   worker-node-3  stdout
                             8bf9c-jzjmd            UserByUsername][load UsernamePasswordAuthenticationToken][username: admin]
May 24, 2023 @ 19:03:48.587  ts-auth-service-86868  2023-05-25 01:03:47.995  INFO 1 --- [io-12340-exec-3] auth.service.impl.TokenServiceImpl    : [getT   worker-node-3  stdout
                             8bf9c-jzjmd            oken][success][USER TOKEN: eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJhZG1pbiIsInJvbGVzIjpbIlJPTEVfQURNSU4iXSwiaW
                                                    QiOiJhZjczMmEyMC02YTYxLTQ5MDItYjIyMy1kYjU4MDQxNDNjYzgiLCJpYXQiOjE2ODQ5NDc4MjcsImV4cCI6MTY4NDk1MTQyN30.
                                                    McGDLDIFfFrqfnMeCRdD99hglYoMNYPiB2C-Mi4Wd90 USER ID: af732a20-6a61-4902-b223-db5804143cc8]
May 24, 2023 @ 19:03:49.585  ts-auth-service-86868  2023-05-25 01:03:48.801  INFO 1 --- [io-12340-exec-8] auth.controller.UserController        : Login   worker-node-3  stdout
                             8bf9c-jzjmd            request of username: admin
May 24, 2023 @ 19:03:49.585  ts-auth-service-86868  2023-05-25 01:03:48.802  INFO 1 --- [io-12340-exec-8] auth.security.UserDetailsServiceImpl  : [load   worker-node-3  stdout
                             8bf9c-jzjmd            UserByUsername][load UsernamePasswordAuthenticationToken][username: admin]
May 24, 2023 @ 19:03:49.586  ts-auth-service-86868  2023-05-25 01:03:49.013  INFO 1 --- [io-12340-exec-8] auth.service.impl.TokenServiceImpl    : [getT   worker-node-3  stdout
```

Figure 4.6: Kibana dashboard

The EFK stack version deployed in this experiment does not support CSV data export, so the only way to retrieve data is using the elastic query language (with python in this implementation).

Elastic query for retrieving logs separately:

```
query = {
    "size": 10000,
    "_source": ["message", "@timestamp", "kubernetes.pod_name", "kubernetes.host"],
    "query": {
        "bool": {
            "must": [
                {"exists": {"field": "message"}},
                {"exists": {"field": "@timestamp"}},
                {"exists": {"field": "kubernetes.pod_name"}},
                {"exists": {"field": "kubernetes.host"}},
                {"range": {"@timestamp": {"gte": "2023-06-06T18:00:00",
                "lt": "2023-06-07T17:50:00"}}}
            ]
        }
    },
    "sort": [{"@timestamp": "asc"}]
}
```

Aggregate (logs per minute) query:

```
query = {
    "size": 0,
    "query": {
        "bool": {
            "must": [
                {"exists": {"field": "message"}},
                {"exists": {"field": "@timestamp"}},
                {"exists": {"field": "kubernetes.pod_name"}},
                {"exists": {"field": "kubernetes.host"}},
                {"range": {"@timestamp": {"gte": "2023-06-06T18:00:00",
                "lt": "2023-06-07T17:50:00"}}}
            ]
        }
    },
    "aggs": {
        "logs_per_minute": {
            "date_histogram": {
                "field": "@timestamp",
                "interval": "minute"
            }
        }
    }
}
```

2.3 Traffic Generator

The *Locust* package is an open-source, Python-based load testing tool used for **measuring the performance and scalability of web applications**. It allows developers and testers to simulate user behavior and generate high loads on web applications to analyze their performance under stress.

Here are some key features and concepts related to the Locust package:

- **Python-based**: Locust is written in Python, making it easy to use and extend with custom functionality. Test scenarios and user behavior can be defined using Python code.

- **Distributed Load Generation**: Locust supports distributed load generation, allowing you to run tests using multiple worker nodes to simulate a large number of concurrent users. It uses a master-worker architecture for load distribution.

- **Test Scenarios**: Locust allows you to define test scenarios by creating classes that inherit from the Locust base class. Test scenarios consist of user actions, such as making HTTP requests, and can be customized to simulate different user behaviors, such as browsing, logging in, or performing specific actions on a website.

- **User Behavior Modeling**: With Locust, you can define the behavior of virtual users using Python code. This includes specifying the number of users to simulate, the frequency of user actions, and any necessary parameterization of requests.

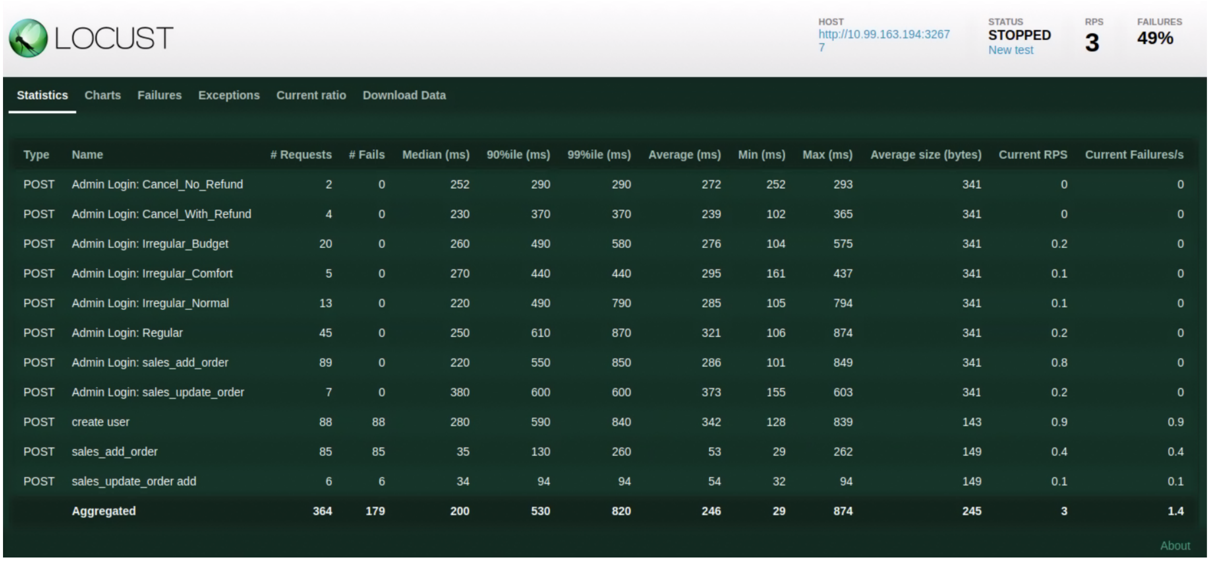
- **Real-time Reporting**: Locust provides real-time reporting during load testing, including metrics like response times, request rates, and error rates. It generates HTML reports and offers integration with various monitoring systems and tools.

- **Web User Interface**: Locust provides a web-based user interface (UI) that allows you to monitor and control the load test in real-time. You can start and stop the test, adjust the number of simulated users, and view statistics and charts.

- **Integration and Extensibility**: Locust can be easily integrated with other tools and frameworks. It provides various hooks and extensibility points for customizing the load testing process and integrating with external systems.

Overall, Locust is a powerful and flexible load testing tool that is widely used in the industry. Its Python-based approach and distributed load generation capabilities make it suitable for testing the performance, scalability, and robustness of web applications.

In this implementation a workload file has been added to define the number of users performing actions every minute, aiming to represent the different traffic volume throughout the day. During this experiment Locust has been used for simulating traffic on the website, instead of being used as a proper load tester generator.
It is important to perform the training and testing under a realistic scenario, with active traffic on the website, to really evaluate the capabilities of the proposed solution in the most correct way.



HOST
http://10.99.163.194:32677

STATUS
STOPPED
New test

RPS
3

FAILURES
49%

Type	Name	# Requests	# Fails	Median (ms)	90%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
POST	Admin Login: Cancel_No_Refund	2	0	252	290	290	272	252	293	341	0	0
POST	Admin Login: Cancel_With_Refund	4	0	230	370	370	239	102	365	341	0	0
POST	Admin Login: Irregular_Budget	20	0	260	490	580	276	104	575	341	0.2	0
POST	Admin Login: Irregular_Comfort	5	0	270	440	440	295	161	437	341	0.1	0
POST	Admin Login: Irregular_Normal	13	0	220	490	790	285	105	794	341	0.1	0
POST	Admin Login: Regular	45	0	250	610	870	321	106	874	341	0.2	0
POST	Admin Login: sales_add_order	89	0	220	550	850	286	101	849	341	0.8	0
POST	Admin Login: sales_update_order	7	0	380	600	600	373	155	603	341	0.2	0
POST	create user	88	88	280	590	840	342	128	839	143	0.9	0.9
POST	sales_add_order	85	85	35	130	260	53	29	262	149	0.4	0.4
POST	sales_update_order add	6	6	34	94	94	54	32	94	149	0.1	0.1
	Aggregated	**364**	**179**	**200**	**530**	**820**	**246**	**29**	**874**	**245**	**3**	**1.4**

Figure 4.7: Locust Dashboard

3 Proposed IDS dataset

The obtained dataset contains more than 500 columns (features) collected from prometheus on hardware metrics. The data was generated over a period of 4 days, using a sampling strategy of 1 minute, resulting in about 5700 rows.
Some data cleaning and feature engineering steps were executed since most of the data was raw and not properly organized. In particular:

- **Delta calculation**: most of the data collected by prometheus are *counters*: positive monotone sums of values. In this case we performed feature transformation in delta values, with the n_{th} row becoming the difference between the n_{th} row and the n_{th-1}: $\Delta O_n = O_n - O_{n-1}$ (this process leads to the loss of the first row).

- **Feature removal**: lots of feature were really sparse(containing mostly 0 or constant numbers) and were deleted because of the lack of discriminant power.

- **Feature engineering**: other features have been transformed, mostly condensing information in less columns. For instance in the cpu metrics information was stored for every core and mode on every node ($3\ nodes * 8\ cores * 8\ modes = 192\ features$). These have been condensed in 24 features(aggregating modes and cores).

The refined dataset obtained from Prometheus has 46 features. For ease of expainability here are the feature available for each node (the features are the same across the three nodes):

Feature	**Description**
time	timestamp
CPU metrics	
idle_node_delta	seconds spent in idle mode
system_node_delta	seconds spent in system mode
user_node_delta	seconds spent in user mode
nice_node_delta	seconds spent in nice mode
softirq_node_delta	seconds spent in softirq mode
irq_node_delta	seconds spent in irq mode
steal_node_delta	seconds spent in steal mode
iowait_node1_delta	seconds spent in iowait mode
Memory metrics	
node_memory	free bytes in memory
pods_memory	free bytes in pods
endpoints_memory	free bytes in k8s service endpoints
Disk metrics	
read_bytes_total	number of bytes read from disk
iotime_sdb	time spent in i/o on sdb (default) partition
reads_completed	number of completed disk reads
Network metrics	
ens192_traffic	bytes received on ens192 interface

Table 4.1: Overview of the prometheus generated dataset

To this prometheus-scraped dataset other features from elasticsearch have been added, including the number of logs produced every minute (see aggregate query in elasticsearch section):

Feature	Description	Targeted by
log_count	total number of logs generated in that timestamp	
ts-auth-service	logs generated by auth service in that timestamp	Auth BF
ts-travel-service	logs generated by travel service in that timestamp	PT
ts-ui-dashboard	logs generated by dashboard service in that timestamp	GET Flood
ts-contacts-service	logs generated by contacts service in that timestamp	XSS

Table 4.2: Overview of the elastic data

The choice of including only 4 of the 40+ services has been made to simplify the training and testing of machine learning models: having to deal with numerous features leads to the need of performing feature selection techniques. To simpify this step we only collected logs from the services that are specifically targeted by cyber attacks, also including the total number of logs to also monitor the general state of the application (log production-wise).

4 Test Set

While the train set only contains normal activity, hence all the same label, the test set is composed of both normal activity and attacks. In particular the test set is made up of 120 observations (2 hours of data) with the attacks distributed in the following way:

- 16 Dos (GET flood),
- 16 Authentication bruteforce,
- 2 Parameter tampering,
- 15 XSS.

The remaining data is normal activity to evaluate how the model behaves with false positives.

4.1 Cross-site Scripting (XSS)

Cross-Site Scripting (XSS) is a web security vulnerability that allows attackers to inject malicious scripts into web pages viewed by other users. It occurs when a web application does not properly validate or sanitize user-generated input and includes it in the output without appropriate encoding.

The vulnerability arises when an attacker finds a way to inject malicious JavaScript or other scripting code into a web page that is subsequently rendered in a user's browser. This can happen through input fields, URL parameters, cookies, or other sources of user input.

There are three main types of XSS attacks:

1. **Stored XSS**: In this type of attack, the malicious script is permanently stored on the target server, such as in a database. When a user visits a specific page that displays the stored data, the script is executed in their browser, potentially leading to unauthorized actions or theft of sensitive information.

2. **Reflected XSS**: In a reflected XSS attack, the malicious script is embedded in a URL and sent to the victim via email, social engineering, or other means. When the victim clicks the manipulated link and the web application reflects the user input back in the response, the script is executed in their browser.

3. **DOM-based XSS**: This type of XSS occurs when the vulnerability lies in the Document Object Model (DOM) of a web page. The malicious script modifies the DOM and can impact the behavior of the web page, potentially leading to unauthorized actions or data theft.

The consequences of a successful XSS attack can vary. Attackers may aim to steal sensitive information, such as login credentials, session tokens, or personal data. They can also use XSS to deface websites, redirect users to malicious websites, or exploit vulnerabilities in other users' browsers.

To prevent XSS attacks, web developers should implement proper input validation and sanitization techniques. This includes validating and filtering user input, using appropriate output encoding when displaying data, and employing security mechanisms such as Content Security Policy (CSP) headers to restrict the execution of scripts. Regular security audits and patching of known vulnerabilities are also crucial to maintaining web application security.

In this implementation the attack has been exectued using the intruder function of Burp Suite, a popular web application security testing software. Using a typical set of payloads commonly used in XSS attacks.

4.2 Get Flood

A GET flood attack, also known as an HTTP GET flood or a GET request flood, is a type of Distributed Denial of Service (DDoS) attack that targets web servers. It involves sending a large volume of GET requests to overwhelm the server's resources and disrupt its normal functioning.

In a GET flood attack, the attacker exploits the HTTP GET method, which is used to request resources from a web server. By sending a massive number of GET requests simultaneously or in rapid succession, the attacker aims to exhaust the server's processing capacity, network bandwidth, or other system resources.

The attack typically involves the use of botnets, which are networks of compromised computers or devices under the control of the attacker. These botnets distribute the attack traffic, making it difficult to identify and block individual malicious sources.

The impact of a GET flood attack can be severe. The excessive volume of GET requests can cause the targeted server to become overwhelmed and unresponsive, resulting in denial of service for legitimate users. This can lead to website downtime, slow response times, and loss of business.

To mitigate GET flood attacks, web administrators can implement various security measures, including:

1. **Rate Limiting**: Setting limits on the number of requests that can be made from a single IP address or within a specific time frame helps mitigate the impact of excessive traffic.

2. **Traffic Monitoring**: Implementing traffic monitoring and anomaly detection mechanisms allows administrators to identify and respond to abnormal request patterns indicative of an ongoing attack.

3. **Content Delivery Network (CDN)**: Utilizing a CDN can help distribute and manage incoming traffic, providing better resilience against DDoS attacks.

4. **DDoS Protection Services**: Employing specialized DDoS protection services or hardware appliances can help detect and filter out malicious traffic before it reaches the target server.

5. **Web Application Firewalls (WAF)**: Implementing a WAF can help identify and block malicious requests, including those associated with GET flood attacks.

By employing these preventive measures, organizations can enhance the resilience of their web servers against GET flood attacks and maintain the availability and performance of their online services.

In this implementation the attack has been executed using a python script that allow to set the number of get request to forward.

4.3 Authentication Brute Force

An authentication brute force attack is a type of cyber attack where an attacker attempts to gain unauthorized access to a system, application, or user account by systematically trying various combinations of usernames and passwords until a valid combination is found. The attack relies on the assumption that weak or commonly used passwords can be easily guessed through trial and error.

In a brute force attack, the attacker uses automated tools or scripts to rapidly submit numerous login attempts to the target system. The attacker may target various entry points, such as login pages(as in this case), remote desktop services, SSH, or any other authentication mechanism that requires a username and password.

The attack works by systematically iterating through a large number of possible usernames and passwords, such as common words, dictionary words, or known password patterns. The attacker tries each combination until the correct one is discovered or until the available options are exhausted.

Brute force attacks can be resource-intensive and time-consuming. However, they can be effective against weak passwords, poorly implemented security measures, or systems that lack mechanisms to detect and mitigate such attacks.

To protect against authentication brute force attacks, organizations and individuals can implement several security measures, including:

1. **Strong Password Policies**: Encouraging users to create strong, complex passwords that are resistant to dictionary-based attacks significantly reduces the success rate of brute force attacks.

2. **Account Lockouts**: Implementing mechanisms that temporarily lock user accounts after a certain number of failed login attempts helps prevent repeated login attempts.

3. **Multi-Factor Authentication (MFA)**: Enabling MFA adds an extra layer of security by requiring users to provide additional authentication factors, such as a one-time password generated on a mobile device or a biometric verification.

4. **Intrusion Detection/Prevention Systems (IDS/IPS)**: Deploying IDS/IPS systems can help detect and block suspicious login attempts or patterns associated with brute force attacks.

5. **Rate Limiting**: Implementing rate-limiting mechanisms that restrict the number of login attempts from a single IP address or within a specific time frame helps prevent rapid and repeated login attempts.

By implementing these security measures and regularly educating users about the importance of strong passwords, organizations and individuals can significantly reduce the risk of successful brute force attacks and protect their systems and accounts from unauthorized access.

Burp Suite was used to implement the attack.

4.4 Parameter Tampering

Parameter tampering is a type of web application attack where an attacker modifies or manipulates the parameters or values in a request sent to a web server to gain unauthorized access or bypass security controls. It involves altering the input fields or parameters in a request to manipulate the intended behavior of the application.

Web applications often rely on parameters to receive user input and process requests. These parameters can be part of URLs, form submissions, cookies, headers, or hidden fields in HTML forms. Parameter tampering attacks occur when an attacker modifies these parameters with malicious intent.

The goal of parameter tampering attacks may vary depending on the specific scenario, but common objectives include:

1. **Unauthorized Access**: Attackers may attempt to modify parameters to gain access to restricted areas of a website, escalate privileges, or view sensitive information.

2. **Data Manipulation**: By altering parameters related to data submission or processing, attackers can manipulate data in the application's database or impact its behavior.

3. **Bypassing Security Measures**: Attackers may try to modify parameters to bypass security controls, such as authentication mechanisms, access controls, or input validation routines.

4. **Denial of Service**: Parameter tampering attacks can also be used to manipulate parameters in a way that causes application errors, crashes, or resource exhaustion, leading to a denial of service.

To protect against parameter tampering attacks, developers and organizations should implement the following security measures:

1. **Input Validation**: Implement strong input validation routines on the server-side to ensure that parameters are properly validated and sanitized. This helps to detect and block any malicious or unexpected input.

2. **Authentication and Authorization**: Implement robust authentication and authorization mechanisms to prevent unauthorized access to sensitive functionality or data.

3. **Integrity Checks**: Apply integrity checks, such as digital signatures or message digests, to validate the integrity of critical parameters and detect any tampering attempts.

4. **Secure Session Management**: Implement secure session management practices, including proper session token handling, session expiration, and protection against session fixation attacks.

5. **Parameter Encryption**: Encrypt sensitive parameters to protect them from unauthorized modification or tampering.

By implementing these security measures and regularly testing and auditing the web application for vulnerabilities, organizations can mitigate the risk of parameter tampering attacks and ensure the integrity and security of their web applications and user data.

Chapter 5

Malicious Activity Detection

Once the application (and all its side components like prometheus and elasticsearch) have been deployed and the data has been generated and collected, the next step is the training and testing of machine learning models to evaluate their effectiveness in this scenario.
One of the contribution of this thesis is the focus on unsupervised machine learning models. The chosen models are three: Autoencoders, Isolation Forest and One-class SVM. In the context of unsupervised learning the problem has been addressed in different ways depending on the model:

- **Autoencoders**: defining a threshold based on the ability of the model to reconstruct the input data (reconstruction error).
- **One-class SVM**: experimenting with different kernel types and nu values to obtain the best results.
- **Isolation Forest**: setting the contamination parameter close to zero to reflect the characteristic of missing anomaly instances in training data.

In the first model anomalies are then classified based on a threshold.
Another contribution is the usage of log production as an added feature. **The models will be trained in two different settings: with and without log production**. This is **to evaluate the discriminant power this feature has** on the models ability to detect anomalies.

1 Autoencoders

Autoencoders are a type of artificial neural network that are commonly used for unsupervised learning tasks such as data compression, feature learning, and anomaly detection. They are particularly effective at learning compact representations of input data.

The basic architecture of an autoencoder consists of an encoder and a decoder. The encoder takes an input and maps it to a lower-dimensional latent space representation. The decoder then takes this code and attempts to reconstruct the original input. The objective of an autoencoder is to minimize the reconstruction error, encouraging the model to learn meaningful representations of the input data.
Autoencoders can be used for anomaly detection by leveraging their ability to learn compressed representations of normal data and their capability to reconstruct the input data. The idea is that if an autoencoder is trained on a dataset that contains only normal

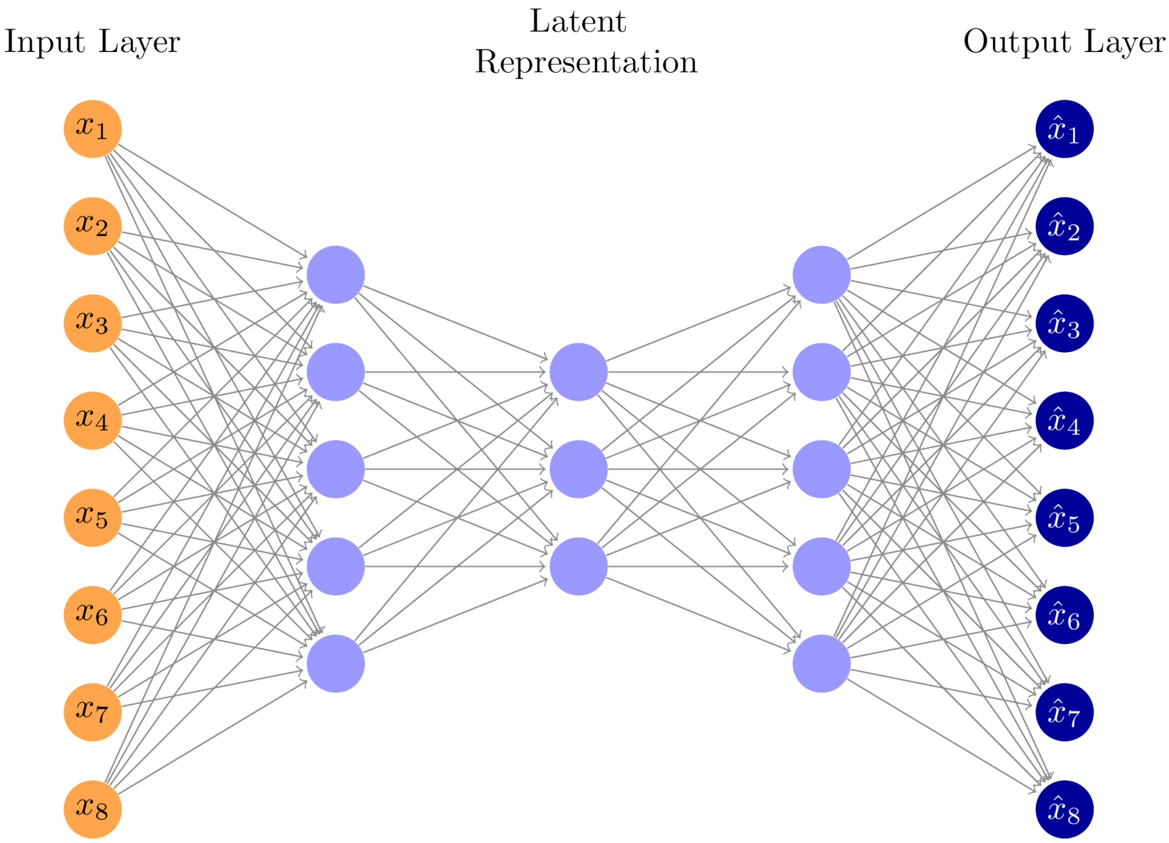


Figure 5.1: Autoencoder architecture

instances, it will learn to reconstruct these normal instances accurately. However, when presented with anomalous or unfamiliar data, the reconstruction error will be higher.

Here's a general approach to using autoencoders for anomaly detection.

Training phase:

1. Collect a dataset containing only normal instances.
2. Train an autoencoder on this dataset, optimizing the reconstruction error between the input and output.
3. The encoder part of the trained autoencoder learns to encode normal instances into a compressed representation.

Testing phase:

1. For a new instance, pass it through the trained autoencoder.
2. Calculate the reconstruction error, which represents the dissimilarity between the input and the reconstructed output.
3. If the reconstruction error exceeds a predefined threshold, the instance is considered an anomaly or outlier.

The assumption behind this approach is that the autoencoder will be more effective at reconstructing normal instances, while anomalous instances will have higher reconstruction errors due to their dissimilarity from the learned representations of normality.

There are variations and enhancements to this basic approach, such as using different types of autoencoders (e.g., variational autoencoders), incorporating regularization techniques, or employing ensemble methods for improved anomaly detection performance.

Autoencoders offer a flexible and powerful framework for anomaly detection, as they can learn meaningful representations from unlabeled data and capture complex patterns in the input data. However, it's important to note that the effectiveness of autoencoders for anomaly detection depends on the quality and representativeness of the training data and the appropriate choice of hyperparameters and thresholds.

1.1 Overcomplete fully connected Autoencoders

Overcomplete fully connected autoencoders are a type of autoencoder architecture where the dimensionality of the hidden layer is larger than the dimensionality of the input layer. In other words, the number of neurons in the hidden layer exceeds the number of neurons in the input layer.

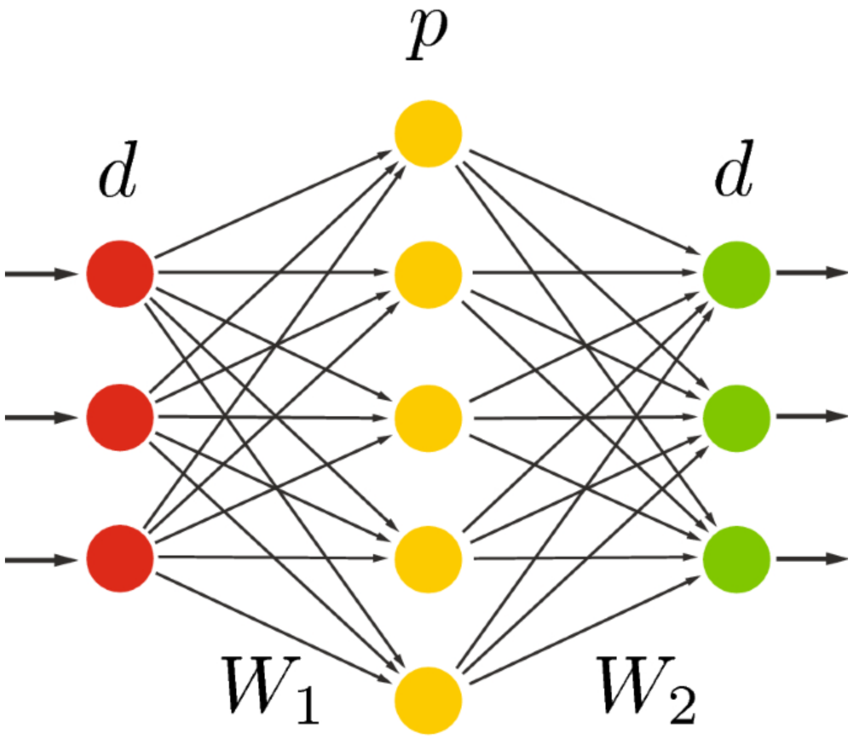


Figure 5.2: Overcomplete autoencoder architecture

Traditional autoencoders aim to learn a compressed representation of the input data in the hidden layer, typically with a lower dimensionality. This forces the model to capture the most important features and discard unnecessary details. However, overcomplete autoencoders challenge this notion by allowing the hidden layer to have more neurons than the input layer.

The motivation behind overcomplete autoencoders is to provide the model with greater capacity to capture intricate and complex relationships within the data. By increasing the number of neurons in the hidden layer, the autoencoder can potentially learn a more expressive and detailed representation of the input.

In practice, overcomplete autoencoders face a trade-off. The increased capacity to capture complex relationships comes at the risk of overfitting. With more degrees of freedom in the hidden layer, the model may learn to simply memorize the training data rather than extracting meaningful features. Therefore, regularization techniques such as dropout, batch normalization, or regularization terms like L1 or L2 may be employed to prevent overfitting and encourage more robust representations.

Overcomplete fully connected autoencoders have found applications in various domains, including image denoising, feature learning, anomaly detection, and dimensionality reduction. By leveraging the additional capacity of the hidden layer, these models can potentially learn more detailed and informative representations of the input data.

1.2 Results

The training and testing process steps executed were the following:

1. Splitting of dataset in train and validation sets (80/20).
2. Feature scaling with sklearn scaler tool
3. Training on the train set
4. Testing on the **validation set** to then extract the mean squared error
5. Calculation of the mean and standard error of the mean squared errors(used as reconstruction errors)
6. Definition of a threshold: $T = mean(\overline{mse}) + stderr(\overline{mse})$ with $stderr = \sigma(\overline{mse})/\sqrt{n}$
7. Testing on the test set
8. Comparing mse obtained in the test set with threshold:
   $if\ mse > threshold \rightarrow attack$

To define the threshold we computed the standard error to add it to mean so to have some tolorance over the mean of the mse in the validation set.

Different autoencoder versions have been evaluated, with differences in hidden dimensions and number of layers, the models that follow are te best performing ones:

Name	Hidden Layers	Layer size
Autoencoder	5	32-16-8-16-32
Overcomplete AE 1	4	128-64-32-64-128
Overcomplete AE 2	2	64-64

Table 5.1: Autoencoders architectures

The input and output layers are defined by the shape of the dataset. Having 50 columns our input and output size needs to be set as 50.

All models have been trained for 100 epochs, with batch sizes of 64, learning rate of 0,001 and mean squared error as loss function.

Model	Classes	Precision	Recall	F1	Accuracy
Autoencoder 1	Normal Attack	78% 70%	80% 67%	79% 69%	75%
Overcomplete AE 1	Normal Attack	75% 76%	87% 57%	81% 65%	75%
Overcomplete AE 2	Normal Attack	66% 82%	96% 29%	78% 42%	68%

Table 5.2: Autoencoder Results

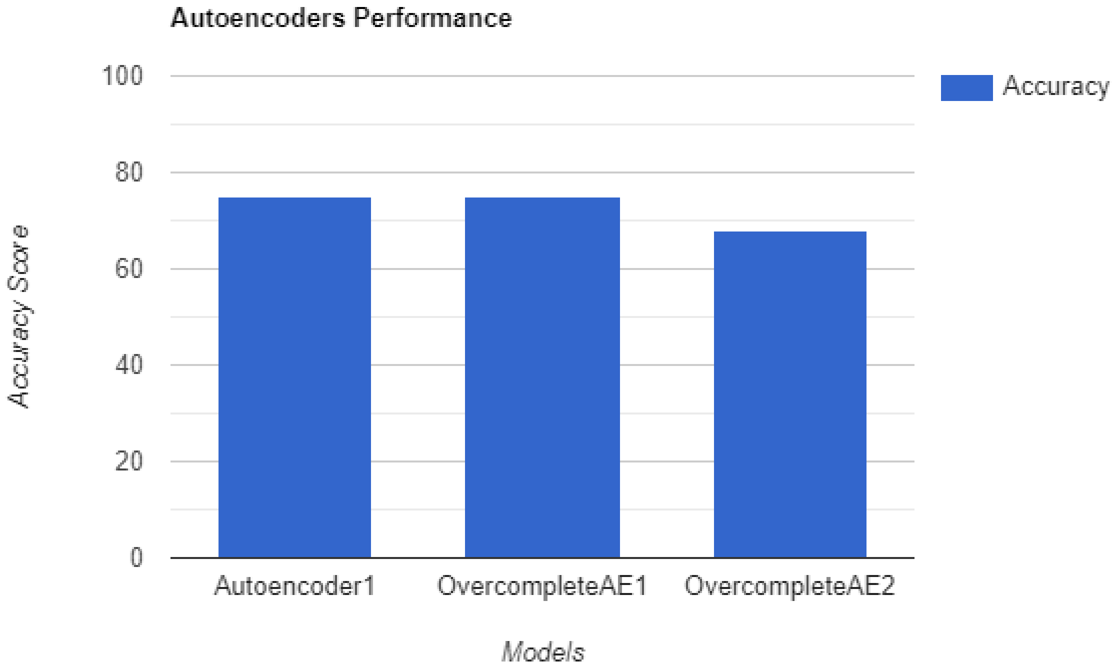


Figure 5.3: Autoencoders Model Accuracy

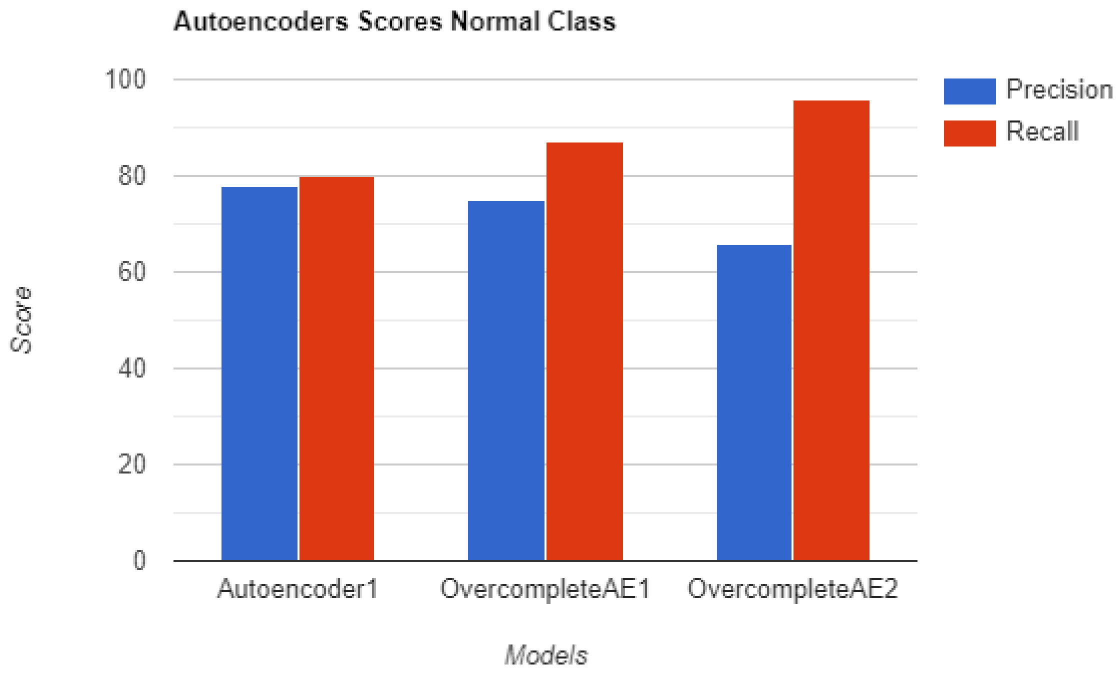


Figure 5.4: Autoencoders Model Precision and Recall on **Normal** class

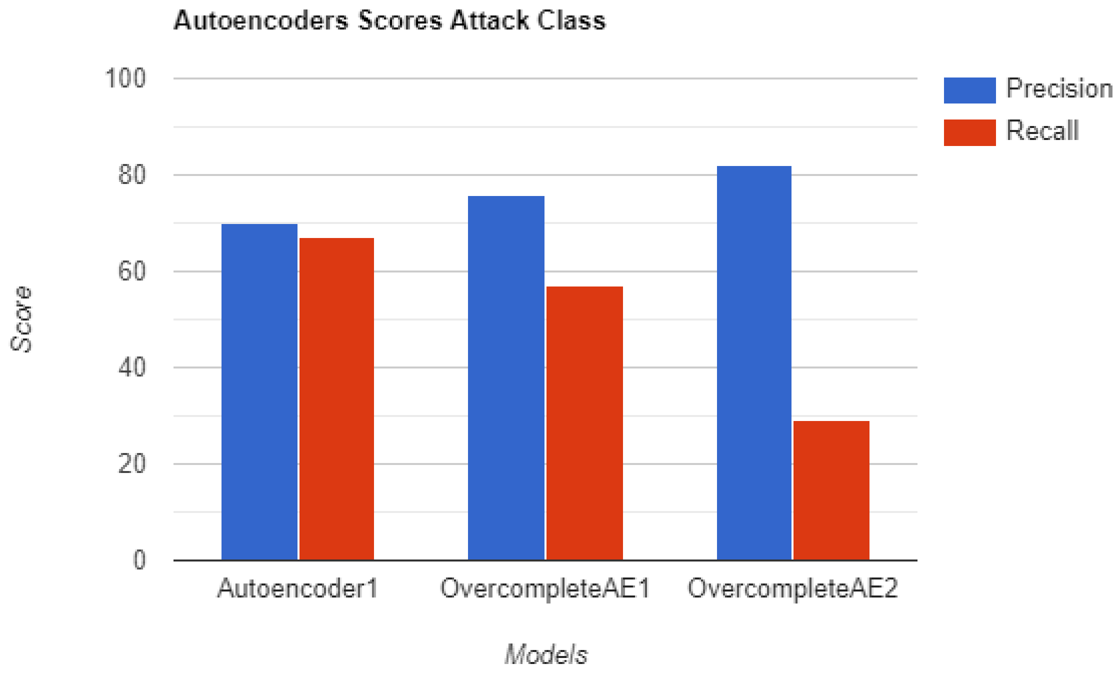


Figure 5.5: Autoencoders Model Precision and Recall on **Attack** class

2 Isolation Forest

An isolation forest is an algorithm used for anomaly detection, specifically in the field of machine learning and data mining. It is designed to identify and isolate unusual or anomalous data points within a dataset. The concept behind isolation forests is based on the idea that anomalies are typically few and different, making them easier to separate from the majority of normal data points.

Here's how isolation forests work:

1. **Random Selection**: The algorithm randomly selects a feature from the dataset and a random split value within the range of that feature.

2. **Recursive Partitioning**: The selected feature and split value are used to partition the data into two subsets. Data points with values below the split value go to the left subset, while those with values above it go to the right subset.

3. **Recursive Subdivision**: The above process is repeated recursively for each subset until all data points are isolated or a predefined stopping criterion is met.

4. **Building the Isolation Tree**: The isolation forest is built by repeating steps 1-3 to create multiple isolation trees. Each tree is constructed independently and doesn't depend on the results of previous trees.

5. **Anomaly Scoring**: To identify anomalies, a scoring mechanism is used. For a given data point, the algorithm measures the average path length required to isolate that point across all the isolation trees. Anomalies tend to have shorter average path lengths since they require fewer partitions to isolate.

6. **Anomaly Detection**: Based on the anomaly scores, a threshold can be set to determine which data points are considered anomalous. Points with scores above the threshold are classified as anomalies, while points below the threshold are considered normal.

Isolation forests have several advantages for anomaly detection. They are computationally efficient, as they require minimal memory usage and have a linear time complexity. They can handle high-dimensional datasets and are not affected by the presence of irrelevant features. Additionally, isolation forests are robust to outliers and can be effective even when only a small portion of the data contains anomalies.

Overall, isolation forests provide a powerful and efficient method for identifying anomalies or outliers in various domains, such as fraud detection, network intrusion detection, and system monitoring.

2.1 Results

When using an Isolation Forest for anomaly detection, it is generally expected that the training dataset contains both normal instances and anomalous instances. The Isolation Forest algorithm works by isolating anomalies as a minority class with fewer instances that are easily separable from the majority normal instances.

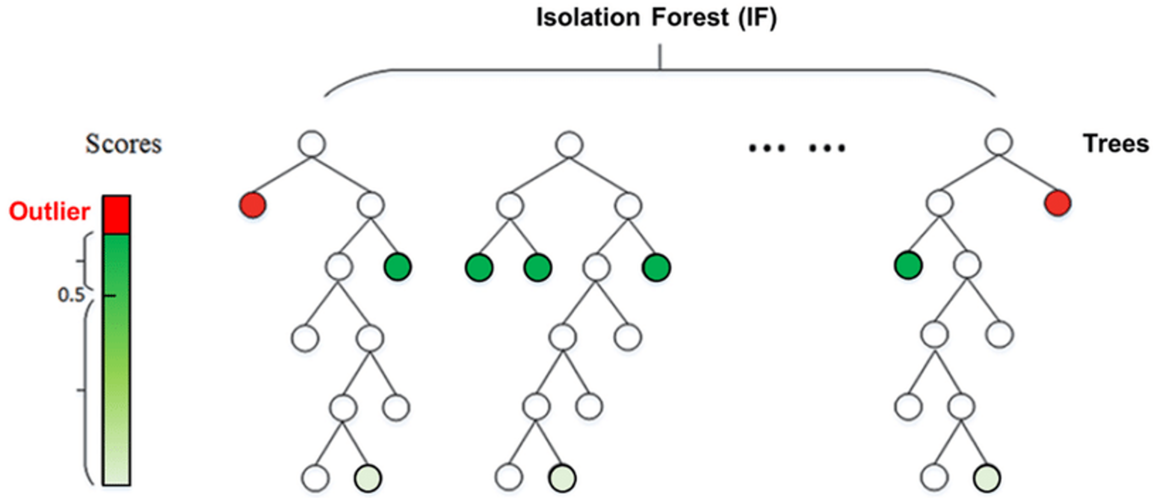


Figure 5.6: Isolation Forest Model

However, if the training dataset only contains normal instances and you want to detect anomalies in the test dataset, you can still use Isolation Forest, but you need to be cautious in setting the contamination parameter. The contamination parameter determines the proportion of anomalies in the dataset that the Isolation Forest algorithm expects to find. In this case, since the training dataset does not contain anomalies, we should **set the contamination parameter very low**, close to zero (0.001 in this case), to reflect the fact that anomalies are rare or almost non-existent.

Model	Classes	Precision	Recall	F1	Accuracy
Isolation Forest	Normal	94%	45%	61%	66%
	Attack	55%	96%	70%	

Table 5.3: Isolation Forest performance

Very high recall at the cost of a not so good precision(lot of false positives).

3 One-Class SVM

A one-class SVM (Support Vector Machine) is a machine learning algorithm used for anomaly detection and novelty detection. It is designed to identify data points that deviate significantly from the normal or expected patterns of a given dataset.

The goal of a one-class SVM is to create a boundary or decision boundary that encompasses the normal data points and separates them from the potential anomalies. It learns the representation of normal data points during the training phase and then uses this representation to classify new data points as either normal or anomalous during the testing phase.

Training Phase:

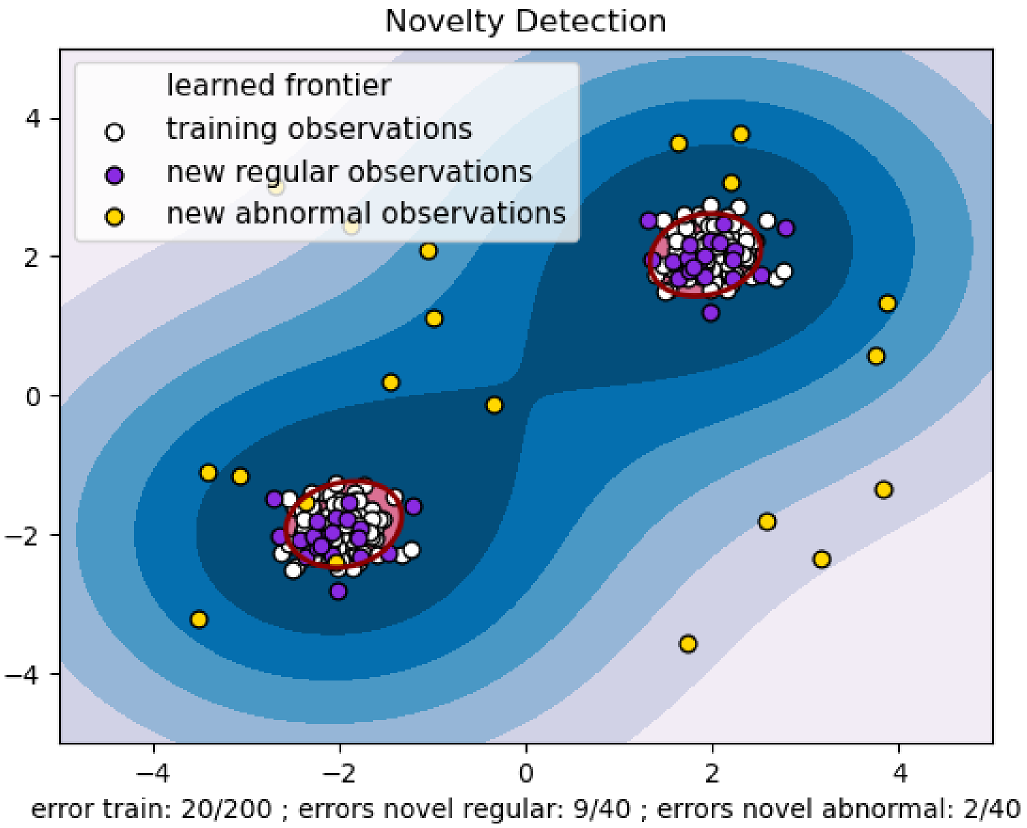


Figure 5.7: One-Class SVM Model

1. Only normal data points are used for training the model, assuming that anomalies are rare and difficult to obtain for training purposes.

2. The algorithm constructs a hyperplane in a high-dimensional feature space that maximizes the margin around the normal data points. The margin is the region between the hyperplane and the nearest normal data points.

3. The model aims to find the hyperplane that includes as many normal data points as possible while excluding any anomalies.

Testing Phase:

1. New data points are evaluated to determine whether they are normal or anomalous.

2. The algorithm assigns a score to each data point based on its distance from the learned hyperplane. Data points that are far from the hyperplane are more likely to be considered anomalies, while points closer to the hyperplane are considered normal.

3. A threshold is set to determine the classification. Data points with scores above the threshold are classified as anomalies, while those below the threshold are considered normal.

One-class SVMs have several **advantage**s for anomaly detection:

- They can **handle high-dimensional data effectively**.
- They are capable of **detecting anomalies even when they differ significantly from the training data**.
- They **do not rely on assumptions about the underlying data distribution**.
- They can be **used in both unsupervised and semi-supervised settings**, as they do not require labeled anomalous data during training.

One-class SVMs are commonly used in various applications, such as fraud detection, intrusion detection, system monitoring, and outlier detection in general. They provide a powerful tool for identifying anomalies by learning the boundaries of normality based on the available training data.

3.1 Results

For this experiment the model has been used in the following way: first training on train data, then using the fit_predict on validation data and then predict on the test set. Some parameters of the model include:

- **nu**: the nu parameter controls the trade-off between the training error and the complexity of the decision boundary. It represents an upper bound on the fraction of training errors and a lower bound on the fraction of support vectors.
- **kernel**: the selection depends on the underlying characteristics of your data and the types of anomalies you are trying to detect. The choice of kernel can significantly impact the performance of the OC-SVM model. Here are some commonly used kernels and their characteristics. The available kernels are: linear, polynomial, radial basis function, sigmoid.

In this experiment the kernel chosen was the rbf(radial basis function), it is one of the most popular choices for anomaly detection with OC-SVM. It is the default kernel in scikit-learn's implementation. The RBF kernel is known for its flexibility in capturing complex non-linear relationships in the data. It can model irregular decision boundaries and is effective when the anomalies exhibit non-linear patterns.
As for the nu parameter, different runs have been tested, choosing the optimal value based on performances. In this case it has been set to 0.0005, also considering the absence of anomalies in training and validation data.

Model	Classes	Precision	Recall	F1	Accuracy
One-Class SVM	Normal	88%	32%	47%	57%
	Attack	49%	94%	64%	

Table 5.4: One-Class SVM performance

Results are a bit disappointing: accuracy is pretty low and precision on attack instances is very low, indicating the propensity of the model to generate a lot of false positives.

4 Logs Influence

In this section we are going to evaluate if and how the addition of log production improves the models performance. The best model from each group has been trained and tested in the same conditions as described in the last sections without the logs features: total logs and the 4 services logs.

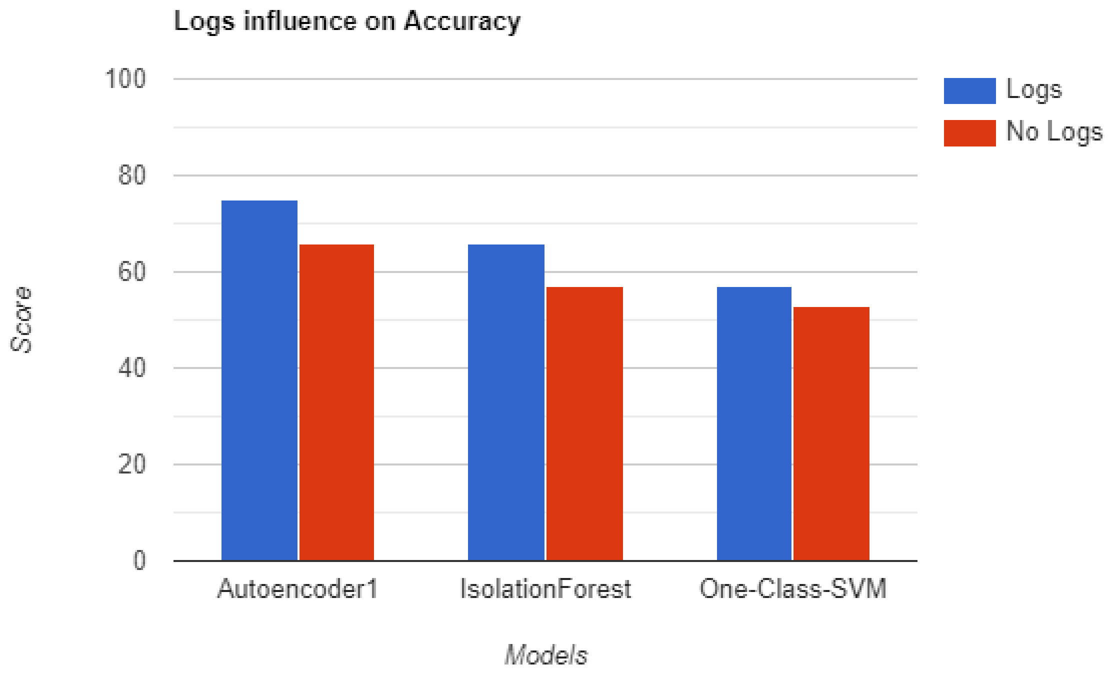


Figure 5.8: Accuracy with and without logs

Model	Classes	Precision	Recall	F1	Accuracy
Autoencoder1	Normal Attack	78% 70%	80% 67%	79% 69%	75%
Autoencoder1	Normal Attack	66% 67%	89% 33%	75% 44%	66%
Isolation Forest	Normal Attack	94% 55%	45% 96%	61% 70%	66%
Isolation Forest	Normal Attack	100% 49%	28% 100%	44% 66%	57%
One-Class SVM	Normal Attack	88% 49%	32% 94%	47% 64%	57%
One-Class SVM	Normal Attack	80% 46%	28% 90%	42% 61%	53%

Table 5.5: Model comparison (blue=with logs | red=without logs)

As we can see **all the models benefited from the added log data**, with an increase in accuracy from 5% to 8%.

5 Focus on the Detected Attacks

In this section we are going to deep dive into the attack class we have seen up to now, analyzing if there is a pattern in model's ability to detect the said attacks. The following table will not consider the false positives problem (for now), that has already been introduced in the previous section (reporting precision metrics in the tables).

Model	GET Flood	Auth Bruteforce	Parameter Tampering	XSS
Autoencoder 1	16/16	10/16	0/2	5/15
Overcomplete AE1	16/16	10/16	1/2	0/15
Isolation Forest	15/16	15/16	2/2	15/15
One-class SVM	16/16	14/16	2/2	10/15

Table 5.6: Models performance on specific attacks (top model for each family)

- **GET flood** is the easiest to detect: a flood attack casuses an abundance in log production and in hardware metrics (increase of network traffic and memory/cpu strain). These are all information we keep track of.
- **Authentication Bruteforce** leads to decent detection performance overall: this attack certainly causes an increase in web requests, but not as significant as a GET flood attack.
- **Parameter tampering** and **cross-site scripting** are the most difficult to detect: these are more of aimed attacks, that don't result in an anomalous behaviour in log production or hardware metrics.

Isolation forest and OneClass-SVM, which are the best models at predicting authentication bruteforce, parameter tampering and cross-site scripting **are also the lowest performing in precision** among the four. This means that, **while they seem great at detecting attacks, this comes at the price of having a lot of false positives**. The **Autoencoders** represent somewhat of a middle ground: they can easily predict GET flood, decently detect authentication bruteforce and they struggle at detecting parameter tampering and XSS attacks. They also **have the most balance between precision and recall**.

What we can evince from these results is that with the data collected and the models that have been evaluated **it is really difficult to detect less volume-centric, aimed attacks** (e.g XSS) since they don't reflect into significative changes in the metrics we observed.

Chapter 6

Conclusions and Future Developments

In conclusion, this thesis was **aimed to design an unsupervised machine learning intrusion detection system** for detecting web attacks. Through extensive research, analysis, and critical evaluation, several key findings and outcomes have emerged, shedding light on this topic.

We've been experimenting **whether hardware performance metrics and log production can be useful to detect web attacks**, both volume centric (GET flood and Authentication bruteforce) and targeted attacks (Parameter Tampering and XSS).

This thesis first tried to **define e new dataset to address this challanges**: based on a microservice application and deployed on a kubernetes cluster, the application has been provided with monitoring and log collection solutions to gather the needed data. Then a load traffic tool has been used to simulate benign traffic on the website and attacks have been performed using BurpSuite and other ready-to-use attack simulation tools.
In this regards, considerations need to be made: one of the **criticalities** of this project could definitely be the **reliability and fidelity of the performed attacks**. Attacks have been executed in a simplified and streamlined manner, since the author does not have the skill required to conduct a proper penetration testing.

Several models have been evaulated, from autoencoders to more classical machine learning models (isolation forest and one-class svm). While some models seem good at predicting attacks, they also fail to contain the number of false positives, with low precision values. While in this context high recall (capability of detecting all attacks) should be the priority, a high number of false allarms could result into a waste of system administrator's time and resources. **The** best (or **most balanced**) **model** obtained **is** an **autoencoder**, which while providing the **best accuracy overall** (75%) has also managed to keep a **decent value in the F1 scores** of the two classes (79% and 69%), being also able to identify some of the most difficult attacks given the dataset.

While this thesis has provided significant insights and contributions to this field, there remain several avenues for future research and development. The following are some potential areas that warrant further exploration (while keeping the same dataset):

- **Increase granularity**: collect data pod wise instead of node wise. This reflects in the data seen in this project: pod metrics instead of node metrics.

- **NLP techniques** in conjunction with the proposed solution: as deductable from the elasticsearch queries in chapter 4, where also the individual logs (with its text) were collected, the authors idea was to then use those logs to generate sentence or word embeddings. The assumption is that under normal load the log semantics could differ from the one under attack. This idea has been briefly explored but couldn't be ready for the dissertation deadline. The approach wanted to use BERT to generate sentence embeddings (grouped by pods) and then use a variety of models (from traditional ones to neural networks) to detect the attacks based on log embeddings. The motivation was to hopefully make up for the lack of accuracy with punctual attack (XSS and Parameter tampering) in the models presented, and use both approach simultaneously to better detect anomalies.

- **Time based approach**: as mentioned in this thesis the concept of time was never overlooked. The author decided to sample prometheus data every minute and kept the timestamp in the obtained dataset. The idea consisted in trying time based approaches like LSTMs to evaulate if introducing time dependency (like seasonality) could improve performance.

- **Production environment**: another interesting follow up could be trying to deploy a model into production to see how it behaves in a real environment, that usually differs from the development environment.In particular another interesting point could be to provide the model with online feedback. In this sense, the model should be updated in an online fashion based on human feedback on the model's detected false positives and(when possible) false negatives.

Chapter 7

Appendix

1 Reproducibility of the experiment

This section is dedicated to the detailed steps performed to build all the technical infrastructure described in the thesis. The section will be divided in different sections:

- Kubernetes Cluster Setup,
- Train ticket, EFK and Prometheus deployment,
- Python code for malicious activity detection.

1.1 K8S Cluster Setup

First, we are going to rename our nodes with their respective roles, each one of the following command must be executed on the respective machine:

```
- sudo hostnamectl set-hostname master-node-1
- sudo hostnamectl set-hostname worker-node-1
- sudo hostnamectl set-hostname worker-node-2
- sudo hostnamectl set-hostname worker-node-3
```

Next we are going to edit the host configuration file on unix (on master node):

```
- sudo nano /etc/hosts
192.168.5.10 master-node-1
192.168.5.25 worker-node-1
192.168.5.26 worker-node-2
192.168.5.27 worker-node-3
```

Be sure to change the ip address of the nodes with your specific addresses. Now we are going to check on other network and node specifications (all nodes):

```
- sudo modprobe overlay
- sudo modprobe br_netfilter
- cat <<EOF | sudo tee /etc/modules-load.d/k8s.conf
overlay
br_netfilter
EOF
```

```
- cat <<EOF | sudo tee /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-iptables  = 1
net.bridge.bridge-nf-call-ip6tables = 1
net.ipv4.ip_forward                 = 1
EOF
- sudo sed -i '/ swap / s/^\(.*\)$/#\1/g' /etc/fstab
- sudo swapoff -a
- sudo sysctl --system
```

Now that our nodes are all set up, we have to download and install a container runtime interface (CRI), all nodes:

```
- curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg
  --dearmor -o /usr/share/keyrings/docker.gpg
- echo \
  "deb [arch=$(dpkg --print-architecture) signed-
  by=/usr/share/keyrings/docker.gpg] https://download.docker.com/linux/ubuntu
  \$(lsb_release -cs) stable" |
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
- sudo apt update && sudo apt install containerd.io
```

It may be necessary to change the Systemd groups if you encounter problem with the CRI, all nodes:

```
- sudo systemctl stop containerd
- sudo mv /etc/containerd/config.toml /etc/containerd/config.toml.orig
- sudo containerd config default > /etc/containerd/config.toml
- sudo nano /etc/containerd/config.toml
    and inside the file set: SystemdCgroup = false
- sudo systemctl start containerd
```

Next we need to install Kubernetes (all nodes):

```
- sudo apt install apt-transport-https ca-certificates curl -y
- sudo curl -fsSLo /usr/share/keyrings/kubernetes-archive-keyring.gpg
  https://packages.cloud.google.com/apt/doc/apt-key.gpg
- lsb_release -cs
- echo "deb [signed-by=/usr/share/keyrings/kubernetes-archive-keyring.gpg]
  https://apt.kubernetes.io/ kubernetes-$(lsb_release -cs) main" | sudo tee
  /etc/apt/sources.list.d/kubernetes.list
- sudo apt update && sudo apt install kubelet kubeadm kubectl
```

Next we are going to install flannel, which is a network fabric for containers (master node):

```
- mkdir -p /opt/bin/
- sudo curl -fsSLo /opt/bin/flanneld https://github.com/flannel-
  io/flannel/releases/download/v0.21.4/flanneld-amd64
- sudo chmod +x /opt/bin/flanneld
```

And finally create the kubernetes cluster using kubeadm:

```
- sudo kubeadm config images pull
- sudo kubeadm init --pod-network-cidr=10.244.0.0/16 \
  --apiserver-advertise-address=10.99.163.194 \
  --control-plane-endpoint=k8s-cluster.lta.disco.unimib.it \
  --ignore-preflight-errors Swap \
  --cri-socket=unix:///run/containerd/containerd.sock
- sudo apt-mark hold kubelet kubeadm kubectl
```

Once the second to last command has been executed, it may took a while to complete the cluster setup (estimate 15-45 mins).

1.2 Train ticket, EFK and Prometheus deployment

Before deploying the application through the makefile, first we need to set some tools: specifically provide local PV(Persistent Volume) support and helm. To install helm you can follow the official guide, while for local pv support we are going to use openebs. All the steps of this section are performed on the MASTER node:

```
- cat <<EOF | kubectl apply -f -
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: local-hostpath-pvc
spec:
  storageClassName: openebs-hostpath
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1G
EOF

- cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Pod
metadata:
  name: hello-local-hostpath-pod
spec:
  volumes:
  - name: local-storage
    persistentVolumeClaim:
      claimName: local-hostpath-pvc
  containers:
  - name: hello-container
    image: busybox
    command:
       - sh
       - -c
       - ’while true; do echo "‘date‘ [‘hostname‘] Hello from OpenEBS Local PV." >> /
    volumeMounts:
    - mountPath: /mnt/store
      name: local-storage
EOF

- kubectl apply -f https://openebs.github.io/charts/openebs-lite-sc.yaml
- kubectl exec hello-local-hostpath-pod -- cat /mnt/store/greet.txt
- kubectl patch storageclass openebs-hostpath -p '{"metadata":
{"annotations":{"storageclass.kubernetes.io/is-default-class":"true"}}}'
```

Once all of this is done just simply use the makefile provided in the repository and choose preferred installation, in this case:

```
- make deploy DeployArgs="--with-monitoring"
```

This option will also deploy prometheus alongside the application. Before deploying I advise to change the tsdb retention (which in the yaml config is set to 24 hours) to 48 or 72 hours at least.

EFK

To deploy the EFK stack I used the following repository. So, once downloaded we just need to modify the Elastic yaml file to deploy it as a loadBalancer, otherwise it wouldn't be able to communicate outside of the cluster, and so it would be impossible to query elastic from other machines. And finally:

```
    kubectl apply -f inside the directory '04-EFK-log'
```

once deployed, just check the services in the cluster to find the exposed ports.

1.3 Malicious activity detection

All the code, that include the feature engineering part and the classification part, can be found in this repo.

Attacks

- GET Flood has been performed using the code in the mentioned repository, just clone it and run it in your terminal.
- The other three attacks have all been performed using BurpSuite: we intercepted web requests and injected custom payload to simulate attacks: in the case of xss for example, we intercepted the message that the client sends to the contacts-service of the application and injected malicious javascript code in text fields. The payloads have been downloaded from the web and can easily be found (example).

Bibliography

[1] C. Abad, J. Taylor, C. Sengul, W. Yurcik, Y. Zhou, and K. Rowe. Log correlation for intrusion detection: a proof of concept. In *19th Annual Computer Security Applications Conference, 2003. Proceedings.*, pages 255–264, 2003.

[2] Mohamed Almorsy, John Grundy, and Ingo Müller. Kubernetes: A review on architecture, challenges, and future directions. *Journal of Systems and Software*, 2017.

[3] Md Zahangir Alom and Tarek M. Taha. Network intrusion detection for cyber security using unsupervised deep learning approaches. In *2017 IEEE National Aerospace and Electronics Conference (NAECON)*, pages 63–69, 2017.

[4] David Boucher and Nick Kratzke. Understanding kubernetes: A comparison of deployment strategies. In *Proceedings of the International Conference on Internet Computing*, 2019.

[5] Lin Chen, Xiaoyun Kuang, Aidong Xu, Siliang Suo, and Yiwei Yang. A novel network intrusion detection system based on cnn. In *2020 Eighth International Conference on Advanced Cloud and Big Data (CBD)*, pages 243–247, 2020.

[6] Zouhair Chiba, Noreddine Abghour, Khalid Moussaid, Amina El omri, and Mohamed Rida. Intelligent approach to build a deep neural network based ids for cloud environment using combination of machine learning algorithms. *Computers Security*, 86:291–317, 2019.

[7] Fabrício S. da Silva, Rafael A. Oliveira, and Paulo F. Pires. Microservices: A systematic mapping study. *Journal of Systems and Software*, 2019.

[8] Elsayed et al. Network anomaly detection using lstm based autoencoder. volume Q2SWinet '20, 2020.

[9] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Jose Rubio. An updated performance comparison of virtual machines and linux containers. *ACM SIGPLAN Notices*, 2014.

[10] Yucheng Han and Ting Chen. Intrusion detection systems: A machine learning approach. *International Journal of Machine Learning and Cybernetics*, 2015.

[11] Ilhan Firat Kilincer, Fatih Ertam, and Abdulkadir Sengur. Machine learning methods for cyber security intrusion detection: Datasets and comparative study. *Computer Networks*, 188:107840, 2021.

[12] Vinod Kumar and Om Prakash Sangwan. Signature based intrusion detection system using snort. *International Journal of Computer Applications & Information Technology*, 1(3):35–41, 2012.

[13] Michael V Mahoney and Philip K Chan. Anomaly-based intrusion detection systems with machine learning. *ACM Computing Surveys*, 2003.

[14] Marty Roesch. Snort-lightweight intrusion detection for networks. In *Proceedings of the 13th USENIX Conference on System Administration (LISA '99)*, 1999.

[15] Kamalakanta Sethi, Rahul Kumar, Nishant Prajapati, and Padmalochan Bera. Deep reinforcement learning based intrusion detection system for cloud infrastructure. In *2020 International Conference on COMmunication Systems NETworkS (COMSNETS)*, pages 1–6, 2020.

[16] Saeid Soheily-Khah, Pierre-François Marteau, and Nicolas Béchet. Intrusion detection in network systems through hybrid supervised and unsupervised machine learning process: A case study on the iscx dataset. In *2018 1st International Conference on Data Intelligence and Security (ICDIS)*, pages 219–226, 2018.

[17] S Tümer and E Türkmen. A study on signature-based intrusion detection systems. *International Journal of Advanced Computer Science and Applications*, 2011.

[18] D'hooge L. Wauters T. et al. Verkerken, M. Towards model generalization for intrusion detection: Unsupervised machine learning techniques. volume J Netw Syst Manage 30, 2022.

[19] Shashank Vohra and Harshal Shah. Kubernetes: A comprehensive overview. *IEEE Potentials*, 2018.