UNIVERSIDADE D
COIMBRA

Paulo Alexandre da Silva Gonçalves

DETECTING INTRUSIONS IN MICROSERVICES ARCHITECTURES

Dissertation in the context of the Master in Informatics Engineering, specialization in Software Engineering, advised by Professor Dr. Nuno Antunes and Master José Flora and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

September of 2022

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D
COIMBRA

DEPARTMENT OF INFORMATICS ENGINEERING

Paulo Alexandre da Silva Gonçalves

Detecting Intrusions in Microservices Architectures

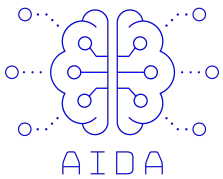
Dissertation in the context of the Master in Informatics Engineering, specialization in Software Engineering, advised by Prof. Dr. Nuno Antunes and Master José Flora and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

September 2022

The work presented in this thesis was carried out within the Software and Systems Engineering (SSE) group of the Centre for Informatics and Systems of the University of Coimbra (CISUC)

This work is partially supported by the project AIDA: Adaptive, Intelligent and Distributed Assurance Platform FCT (CMU-PT) (`POCI-01-0247-FEDER-045907`), co-funded by the Portuguese Foundation for Science and Technology (FCT) and by the *Fundo Europeu de Desenvolvimento Regional* (FEDER) through *Portugal 2020 - Programa Operacional Competitividade e Internacionalização (POCI)*.

This work has been supervised by Prof. Nuno Antunes and José Flora, and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.



Cofinanciado por:

COMPETE 2020
PROGRAMA OPERACIONAL COMPETITIVIDADE E INTERNACIONALIZAÇÃO

PORTUGAL 2020

UNIÃO EUROPEIA
Fundo Europeu
de Desenvolvimento Regional

Acknowledgements

I would like to thank my family for their great patience and dedication during the 22 years of my life. I also want to express my deep appreciation for all their efforts to provide me with a good education and guide me on the right path.

I also want to express my gratitude to my advisors, Prof. Dr. Nuno Antunes and Master José Flora, for all their guidance but especially for putting up with me for the past 3 years. Both are people I admire and respect, having had a major positive impact on my life that I hope to continue to carry for many years to come.

Finally, to all my friends, I also want to leave a note of recognition, even through all the yelling, “feeding" and arguing, we somehow managed to still stick together until the end.

Abstract

Microservice architectures have been on the rise in recent years, as they favor loosely coupled services with specialized goals. The use of microservices eases development and maintainability, reducing overall development costs. To take full advantage, each service is deployed in a container, as they are lightweight and creation and destruction is cheap and efficient. This potentiates an elastic behaviour that allows for on-demand adaptation. However, the transition from monoliths to microservices presents security challenges. When systems are split into multiple small components, the number of communication links grows extensively, increasing the possibility of attacks. During development, automated vulnerability assessment tools are used to find and correct software weaknesses. Despite this process, software continues to have unknown vulnerabilities that attackers can exploit, therefore it is indispensable to apply other measures. Anomaly-based intrusion detection could be an effective approach, but it needs to deal with large amounts of data and services, and adapt to the dynamic number of service instances. Therefore, the available solutions must become lightweight and scalable. Past work has demonstrated the usefulness of data processing techniques to deal with scalability. A similar approach can be useful for detecting attacks targeting multiple services. Techniques can also be used to reduce the amount of data processed without compromising the effectiveness of the solution.

In this work, we propose an approach for host based intrusion detection aimed at microservices, and four different techniques based on a classifier fusion procedure. The approach as the goal of facilitating the detection of attacks targeting different microservices. We also analyse the removal of loops as a method to reduce the data processed without affecting the classifiers. The techniques proposed are evaluated using two representative microservice testbeds, Sockshop and TeaStore, as they provide a wide range of technologies and services, allowing 10 different attack scenarios based on 12 different exploits. Not all attacks were successful, and some were only done to gather information; therefore, evaluating the effectiveness of techniques in intrusion attempts.

The results show that only one technique was capable of detecting attacks without a high number of false positives, achieving an F-Measure of up to `0.673`, never lower than `0.521`, and Precision as high as `0.782`. Furthermore, the loop removal algorithm managed to reduce the total size of the training datasets to `55%` of the original size. Both still have room for improvement; however, the results are still satisfying and open new research paths.

Keywords

Microservices; intrusion detection, vulnerability assessment, attack injection, containers.

Resumo

Arquiteturas de microsserviços têm crescido em popularidade nos últimos anos, favorecendo serviços simples fracamente acoplados e com objetivos bem definidos. Utilizar microsserviços facilita o desenvolvimento e manutenção, reduzindo custos gerais de desenvolvimento. Para tirar proveito destas vantagens, cada serviço é colocado em "containers" dado que são leves e a criação e destruição é barata e eficiente. Tal potencia um comportamento elástico, maximizando o uso dos recursos, e permitindo uma adaptação com base na demanda. No entanto, a transição para estes ambientes cria alguns desafios. Quando os sistemas se dividem em múltiplos pequenos componentes, o número de comunicações aumenta, agravando a probabilidade de ataques. Durante o desenvolvimento, técnicas de avaliação de vulnerabilidades são usadas para detetar e corrigir fraquezas no software. Mesmo assim, o software continua a ter vulnerabilidades desconhecidas que atacantes podem aproveitar, pelo que é indispensável utilizar medidas de deteção. Deteção de intrusão com base em anomalias pode ser uma técnica eficaz, mas tem de conseguir lidar com elevadas quantidades de informação e adaptar-se a um número dinâmico de instâncias de serviços. Portanto, as soluções têm de ser leves e escaláveis. Trabalho já realizado demonstra que técnicas de processamento de informação são úteis para lidar com escalabilidade. Uma abordagem semelhante ser útil para detetar ataques a vários serviços. Técnicas podem também ser usadas para reduzir o tamanho dos "datasets" sem comprometer a eficácia da solução.

Neste trabalho propomos uma abordagem para deteção de intrusão "host based" com foco nos microserviços, e quatro técnicas baseadas numa abordagem de fusão de classificadores. A abordagem tem o objetivo de facilitar a deteção de ataques a diferentes microserviços. Também analizamos a remoção de repetições, como método para reduzir o tamanho dos datasets sem afetar significativamente os classificadores. As técnicas propostas são avaliadas usando duas testbeds representativas, Sockshop e TeaStore, que utilizam um leque variado de tecnologias e serviços, permitindo 10 cenários de ataque, com base em 12 vulnerabilidades. Nem todos os ataques têm sucesso, e alguns só conseguem informação; permitindo assim avaliar a eficácia das técnicas em tentativas de intrusão.

Os resultados mostram que só uma técnica é capaz de detetar ataques sem um número elevado de falsos positivos, conseguindo uma F-Measure de até `0.673`, nunca inferior a `0.521` e uma Precision até `0.782`. Além disso, o algoritmo de remoção de repetições de system calls conseguiu reduzir o tamanho dos datasets até `55%` do tamanho original. Ambos têm espaço para melhoria, no entanto, os resultados são satisfatórios, havendo espaço para pesquisa.

Palavras-Chave

Microsserviços; deteção de intrusão, avaliação de vulnerabilidades, injeção de ataque, containers.

List of Publications

Work presented in this document has groundwork on two publications:

- José Flora, **Paulo Gonçalves**, Miguel Teixeira, Nuno Antunes, "My Services Got Old! Can Kubernetes Handle the Aging of Microservices?", *The 13th International Workshop on Software Aging and Rejuvenation (WOSAR 2021)*, Wuhan, China, October 25-28, 2021.
- José Flora, **Paulo Gonçalves**, Nuno Antunes, "Using attack injection to evaluate intrusion detection effectiveness in container-based systems", *2020 IEEE 25th Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2020, pp. 60-69.

Work presented in this document that has been submitted but not yet accepted:

- José Flora, **Paulo Gonçalves**, Miguel Teixeira, Nuno Antunes, "A Study on the Aging and Fault Tolerance of Microservices in Kubernetes", *IEEE Access*.
- José Flora, Miguel Teixeira, **Paulo Gonçalves**, Nuno Antunes, "μDetector: Intrusion Detection for Scalable and Elastic Container-based Microservice Systems", *44th IEEE Symposium on Security and Privacy (IEEE S&P 2023)*, at the Hyatt Regency, San Francisco, California, USA, May 22-26, 2023.

Contents

Acronyms

BOSC Bags Of System Calls.

DDoS Distributed Denial of Service.

DES Data Encryption Standard.

DoS Denial of Service.

DRDoS Distributed Reflection Denial of Service.

FPR False Positive Ratio.

HIDS Host-based Intrusion Detection System.

HMM Hidden Markov Model.

HPA Horizontal Pod Autoscaler.

IDS Intrusion Detection System.

NIDS Network-based Intrusion Detection System.

POC Proof Of Concept.

RCE Remote Code Execution.

STIDE Sequence Time Delaying Embedding.

VM Virtual Machine.

XSS Cross-Site Scripting.

XXE XML external entity injection.

List of Figures

List of Tables

Chapter 1

Introduction

A microservice is a simple singular service that has a specific goal, the ability to scale and can be mostly tested independently of the system [Larrucea et al., 2018; Newman, 2021; Thönes, 2015]. It is very specific to the task to which it is assigned and, therefore, has a clear definition. An architecture based around them can be logically separated, resulting in a collection of different components or services, allowing developers to split their responsibilities and develop without interference from each other [Newman, 2021]. This saves time on many different tasks, such as unit and non-functional testing, since the service can be tested independently, without worrying about integration, thus only doing the necessary tests with the whole application, making integration with the whole application faster [Newman, 2021].

Agile development methodologies take advantage of this architecture, given their fundamentals around continuous development with many short increments [Abrahamsson et al., 2017; Cohen et al., 2003; Dyba and Dingsoyr, 2009]. DevOps, for example, follows this development methodology, enabling continuous integration and delivery, based on short iterations [Ebert et al., 2016]. Since there are many components which require continuous work, coordination inside the team and high maintainability are crucial to reduce development costs to the maximum and take advantage of these methodologies. A solution is to work independently, by splitting the software into smaller components, such as different microservices.

In addition, to fully maximise the benefits of microservices, they are often deployed in containers, which are small packed virtual environments that contain only the libraries and files necessary to run the specified software. Deployments become faster, as opposed to a deployment using a virtual machine; and their elasticity allows for better resource management since it is easy to instantiate and to destroy a container. Container engines manage these environments and ensure all virtualisation requirements, such as the isolation of the code running on each virtual guest. Cloud users and providers capitalise on this combination of microservices and containers that reduces costs by using the available resources more adequately [Pahl and Jamshidi, 2016; Zimmermann, 2017] When a microservice in a container is busy, with almost all resources being used, a new container can be instantiated, creating a replica. The opposite can also happen; if

the microservice is idle, one of its replicas can be destroyed.

Although microservices have their potential, we should analyse the impact of transitioning from a monolith to a system based around microservices. Many works have been done on quality attributes such as scalability, monitorability, and security [Li et al., 2021]. Some quality attributes, like scalability, are naturally supported by the architecture itself. However, others, namely monitorability, although supported, have some challenges. An example is the large number of services and replicas that need to be monitored. Container orchestrators, such as Kubernetes, ease this task by monitoring all container instances and being configurable to react faster and automatically to some events, such as a system failure. They also support on-demand system management, allowing containers to scale up or down according to current needs. [Al Jawarneh et al., 2019; Lee et al., 2020; Rossi et al., 2020]. Security, on the other hand, needs to be addressed as many more problems are raised.

Security is a timeless topic, that has been continuously researched, and lays on three different fundamental properties: confidentiality, integrity, and availability [Kumar et al., 2015; Vieira and Antunes, 2013]. Additionally, systems are not perfect, and vulnerabilities are prone to being found and abused by attackers. Software developers and researchers often face the task of finding vulnerabilities. Penetration Testers are responsible for finding and sometimes actually performing an exploit on said vulnerabilities. Doing this assessment by hand takes valuable time, and finding or even creating a Proof Of Concept (POC) is not guaranteed to be easy [Wang et al., 2019]. Therefore, vulnerability assessment tools have been developed that allow penetration testers to increase their productivity [Wallis, 2022]. However, every software has vulnerabilities, making it practically impossible to evade this fact. In a microservice-based system, given the various different services, possibly even built using different technologies, vulnerabilities are prone to be found. Also, there is more room for error, since developers need to be extra careful and implement secure communications between the different services. It is only a matter of time until an intrusion happens. Therefore, it is absolutely necessary to have support systems in place that can detect, react, and block these attacks.

Intrusion Detection has been used several times to provide part of this support. It consists on the process of monitoring a system and automatically detecting signs of intrusions when they happen [Bace and Mell, 2001]. An intrusion is an attack with the goal of compromising the fundamentals of security, by trying to gain control or disturb the system; and / or bypassing the security measures that were implemented [Bace and Mell, 2001]. Intrusion Detection System (IDS) are systems that detect intrusions and report them to the administrator and / or another system that will react. The whole system should detect intrusions in a timely manner, without drastically affecting the performance and with a low number of false positives. IDS are commonly divided into three different categories when looking at what they are monitoring and the information they are gathering: network, which consists of analysing network traffic; host, which analyses data that are generated by the host of the application; and application, taking only into account the data generated by the application [Bace and Mell, 2001].

Regarding their analysis, they can be split into: signature-based, where the IDS has a profile describing the known attacks, therefore considering an intrusion if an event matches a known attack pattern; and anomaly-based, in which IDS has a profile consisting of normal behaviour, and each event outside of that profile is considered an intrusion [Bace and Mell, 2001]. In addition, some work has been done in which signature-based and anomaly-based detections are used at the same time, creating a hybrid [Depren et al., 2005; Hajisalem and Babaie, 2018]. An analysis of its history and the research being carried out was performed, stating that interest in the evaluation of IDS started to increase only in the late 1990s [McHugh, 2001]. IDS still require to advance as they tend to detect only 3 out of 4 attacks and have high false positive rates [McHugh, 2001].

However, with respect to microservice architectures, because they usually have hundreds or thousands of services, the data produced is quite large. IDS are forced to be lightweight, so that they can be paired with microservices and the performance of the system is not affected, while also presenting the challenge of needing to be effective and having to provide a fast response. This obviously creates a constraint on these systems that cannot be taken lightly, restricting the mechanisms and algorithms that can be used. Another problem appears in order to cope with the many different services, as a way to join multiple IDSs is needed. Some studies have been conducted on classifier fusion techniques, which unify different data sources, classifiers, or decisions, resulting in a single final output [Guo et al., 2019a]. However, it was mostly done in other areas [Guo et al., 2019a,b; Xu et al., 2020]. Regarding host-based intrusion detection systems in microservice applications, most of the work done is related to Network-based Intrusion Detection System (NIDS) [Cinque et al., 2019; Mateus-Coelho et al., 2021]. Regarding Host-based Intrusion Detection System (HIDS), the work in this field is more focused on general single containers. For example, Abed et al. applied BOSC to detect intrusions on a Linux container [Abed et al., 2015a]. They were able to detect all attacks, with a low False Positive Ratio (FPR). We also did some work with respect to attack injection to evaluate IDSs [Flora et al., 2020] for container-based systems. This work was based on previous solutions to IDSs to then find the best configurations and present their effectiveness. However, work on intrusions targeting multiple services is lacking.

This work presents an approach for host-based intrusion detection in microservice architectures where multiple microservices can be target of an attack. The approach can operate any combination of techniques proposed in this work, that are based on a classifier fusion procedure. However, this work only evaluates a single technique at a time. A technique to reduce data size for more efficient processing is also presented. The effectiveness of the approach and techniques is evaluated using representative testbeds, workloads, and attacks in an experimental campaign. The proposed conceptual solution is able to collect data from the different services and then make a final decision. It takes into account the system limitations following a microservice architecture. Although a network intrusion detection system could be used, the focus of this work is on host-based intrusion detection systems. For that reason, and due to previous work already performed on the matter, we focused on intrusion detection mechanisms that use system calls from containers and simple lightweight algorithms to detect intru-

sions. Many attacks can be carried out in a distributed manner, that is, attacking various services at the same time, and any one of them could be chosen to test the solution. An analysis was done to find the best representative attacks that can be performed. Unfortunately, not all attack scenarios were performed since some problems appeared time was a constraint, we opted for a set of 10 scenarios, using 13 different attacks. Some of these were successful, and others were partly unsuccessful. They are related to denial of service, information gathering and disclosure, remote code execution, brute force, and buffer overflow. These attacks were executed in parallel to different services.

To evaluate the proposed approach, two representative microservice-based testbeds were used; TeaStore [von Kistowski et al., 2018] and Sockshop [Sockshop, 2021]. These were chosen on the basis of previous work done, their size, and their representativeness. 10 different representative attack loads were generated, based on the 10 different scenarios depicted, four for TeaStore and six for Sockshop. These were put together from simple adaptations of the representative workloads, with the attack occurring during the middle of the experiments.

The results show that the only technique currently being able to be used in a real scenario is the interleaving of system calls from different services; however, there is still room for improvement. This technique managed to reach `0.707` F-measure values, while not falling below `0.521` when considering the top 10 configurations with the best results. Also, it is possible that other techniques get better results if trained for longer periods, as it is possible that the classifiers were not fully trained due to the lower number of system calls. The loop removal algorithm also showed good results, being able to reduce the datasets used for training, becoming only 55% of their original size.

1.1 Research Contributions

This work has five main contributions, three of which are the core of the thesis:

- **Evaluation of state of the art techniques**. Evaluation and comparison of state-of-the-art intrusion detection techniques, typically used for single containers. An insight into the pros and cons of current techniques is provided, as well as what we found to be the best configurations.
- **Understand how these algorithms perform when considering microservices scalability** Microservice-based systems leverage horizontal scalability having multiple instances of the same service. We explored data processing techniques as a useful approach to provide the capacity of dealing with higher amounts of data whilst maintaining high effectiveness.
- **Evaluation of the impact on performance and effectiveness when removing loops of system calls**. Datasets are considerably large, averaging more than 100GB per training run, therefore, analysing various algorithms and techniques becomes a time consuming task. The removal of system call

loops shows promising results, greatly reducing the required disk memory, while maintaining effective classifiers.

- **An approach to detect multi-target attacks in a microservice architecture**. To detect attacks targeting multiple services in a microservice-based system, novel techniques were considered and adapted to dynamic environments. The given approach can be useful for testing other techniques or for implementing these in a real system.
- **Proposal and evaluation of four fusion techniques for intrusion detection in representative microservice scenarios**. This evaluation demonstrates the effectiveness of the proposed techniques and allows an analysis of the advantages and drawbacks of each of them. It also allows us to validate the proposed approach. Two different testbeds were used and 10 different attack scenarios were performed.

1.2 Document Structure

The present document is structured as follows:

Chapter 2 presents the background and also the work that has been done on the different topics. It gives the foundations necessary to understand the work.

Chapter 3 presents and discusses the work conducted in the study of intrusion detection techniques for single services. We have evaluated state-of-the-art techniques and proposed the use of data processing approaches to deal with scalability while maintaining effectiveness.

Chapter 4 details our approach and the solution that we propose for detecting attacks targeting multiple microservices. It describes the loop remover algorithm and techniques to cope with multi-target attacks and presents the evaluation and results.

Chapter 5 looks at the risks and all the planning that was done during the thesis.

Chapter 6 concludes the work with some final remarks.

Chapter 2

Background and Related Work

In this chapter, we will give an overview of all the topics necessary to understand this document, while also reviewing previous work related to the research conducted. We start by presenting microservice concepts and its security challenges, with special attention to Intrusion Detection and Intrusion Detection System (IDS). Furthermore, we discuss attacks that can target a microservice system.

2.1 Microservices

Microservices are simple services that have the goal of performing a simple task, being able to cooperate, forming a microservice-based system [Larrucea et al., 2018; Newman, 2021; Thönes, 2015]. This type of system is built on very small independent components, each with a clear definition, allowing split development and testing of their functions [Larrucea et al., 2018; Newman, 2021; Thönes, 2015]. Agile development methodologies, since they are based on continuous development and increments, take advantage of these features to the fullest [Abrahamsson et al., 2017; Cohen et al., 2003; Dyba and Dingsoyr, 2009]. DevOps is an example of one of these methodologies, which focusses on continuous integration and testing, which is facilitated by employing a microservice architecture [Larrucea et al., 2018]. This synergy enables fast delivery cycles, with smoother development [Larrucea et al., 2018].

Monolithic applications, on the other hand, are applications shipped in one big software component. This means that its modules are coupled together, making it impossible to separate them and deploy them on their own. Although for small projects with a short lifespan, it is an acceptable solution, many issues appear for long-term projects. For example, if a single component or module is the bottleneck of the system, it is necessary to scale the entire application to meet higher demand [Dragoni et al., 2017]. Additionally, changes in the requirements that affect the frameworks or technology being used may be unfeasible, as they may require a change in most components, which would be expensive [Dragoni et al., 2017]. Finally, a simple bug fix to a single component would require the entire application to be restarted to update itself, instead of being only necessary to restart

the service that was changed [Dragoni et al., 2017].

Given these benefits, the latest trends have shown that companies are migrating from monolithic applications to these microservice architectures [Dragoni et al., 2017; Eski and Buzluca, 2018]. This transition is possible by first identifying the different modules and then turning each one into a different microservice, as we can see in Figure 2.1. Following this trend, a tool was developed with the aim of helping with this transition [Eski and Buzluca, 2018].It analyses Java source code and repositories, different files, and applies a graph clustering technique based on the changes made, the frequency, and the coupling between components [Eski and Buzluca, 2018]. As a result, it is capable of suggesting candidate microservices within the architecture so that developers can have an easier transition between architectures [Eski and Buzluca, 2018].

Figure 2.1: Transition from monoliths to microservices (from Yarygina [Yarygina, 2018]).

A website was created with all the standards and patterns of microservices, which is constantly updated and consists of many resources such as presentations, step-by-step guides, books, and articles [Richardson, 2022]. Due to the large amount of content and in-depth analysis, this website should be viewed by anyone who wants to know more about microservices.

2.2 Containers

Most of the time, microservices are deployed inside containers, which are single compact virtualisation environments that have only the necessary files and libraries to run the specified program [Bernstein, 2014; Dragoni et al., 2017; Merkel et al., 2014]. Therefore, they have the bare minimum size and take advantage of all the resources allocated, allowing for fast deployment and on-demand changes. Container engines, such as LXC and Docker, are responsible for managing containers [Bernstein, 2014; Chen and Zhou, 2021; Merkel et al., 2014]. To instantiate a container, an image must first be created, containing all the information needed to run. These images can be stored locally or in a repository, but, in the latter case, they still need to be downloaded before running the specified container. Docker has a public repository called DockerHub where anyone can get their images [Merkel et al., 2014; Rad et al., 2017]. It has also been adopted as a standard by the community and has received significant attention, and there is a wide variety of work in various fields such as performance, scalability, and security [Bui, 2015; DataDog, 2018; Rad et al., 2017; Shanmugam, 2017].

However, managing every container is not easy, especially when the number of services in a microservice architecture can vary from a few dozens to hundreds

of different services [Xiang et al., 2021]. Thus, container orchestrators are used to automate the management of groups of containers, which generally form a microservice system, performing tasks such as taking care of communication between them and automatically restarting them when needed [Al Jawarneh et al., 2019]. Kubernetes, the most popular container orchestrator, provides many features by default, such as, for example, probing services and horizontal autoscaling, which is automatic scale up and down according to demand [Burns et al., 2018]. Still, the community also provides many frameworks, plugins, and addons that can be used with Kubernetes, allowing, for example, easier networking management [Kapočius, 2020]. One example is KubeEdge, a framework that extends the use of Kubernetes to edge nodes in an edge architecture, which is gaining popularity in recent years, as it allows faster response times in large distributed networks, by pushing computational work closer to user devices [Shi and Dustdar, 2016; Xiong et al., 2018].

Many quality attributes such as performance, availability, and scalability have been addressed with technologies such as container orchestrators [Al Jawarneh et al., 2019; Burns et al., 2018]. Their configuration is utterly important to deal with these requirements. However, some challenges are yet to be faced or must be taken into consideration when developing new methodologies or when transitioning old methodologies to this new architecture.

2.3 Challenges of Microservice based Architectures

Although other quality attributes could be studied and developed further, security is a really important topic, and considering the previous challenges and the constant increase in popularity, security might become the top priority. Security can be considered as the normal execution of software, even under a malicious attack, including three fundamental concepts: confidentiality, integrity, availability, and authenticity [Kumar et al., 2015; Vieira and Antunes, 2013]. Despite not being enough, we already referred to some solutions to availability, such as the use of container orchestrators, like Kubernetes. More mechanisms need to be put in place to secure these fundamentals in microservices. Previously, for other architectures, mechanisms have already been studied to deal with various problems; however, they need constant evolution to follow the development of attackers [Ruefle et al., 2014]. When it comes to microservices, due to the architecture fundamentals, new challenges appear, forcing these mechanisms to be adapted.

The decomposition into different services increases the number of communication links through APIs thus, creating new attack points [Yarygina and Bagge, 2018; Zimmermann, 2017].Therefore, **the attack surface increases**, forcing developers to check every service, to ensure that they are secure. This requires even more time if we consider that the technologies being used on the different services are not always the same [Yarygina and Bagge, 2018; Zimmermann, 2017]. Another factor that increases this attack surface comes from multitenancy. Within cloud deployments, various **tenants** use these environments to deploy their microservice systems, creating an aggregation of services **sharing the same infras-**

tructure, making them easier to access [Yarygina and Bagge, 2018; Zimmermann, 2017].

Given that the technology being used may be different between different services **the number of vulnerabilities scales with the number of services**, therefore, additional work is required to prevent attackers from abusing them [Yarygina and Bagge, 2018; Zimmermann, 2017].

With the use of system discovery mechanisms and registries to connect the different services, it creates an opening for an attacker to infiltrate inside the system by disguising himself as a service [Dragoni et al., 2017; Sun et al., 2015]. Furthermore, by compromising one single service, they can gain access to the entire system [Dragoni et al., 2017; Sun et al., 2015]. Therefore, it is absolutely important to guarantee **trust between the different services**, to prevent attackers from abusing it.

Additionally, due to decoupled services, **network will be much more complex, with a higher amount of traffic**, since all services are communicating with each other [Dragoni et al., 2017; Sun et al., 2015; Yarygina and Bagge, 2018], which leads to attackers having multiple ways of accessing the system. Furthermore, by decoupling, much more data will need to be analysed, making it harder to enforce security mechanisms throughout the system [Dragoni et al., 2017]. These security mechanisms can be, for example, secure communications or even monitorization. Furthermore, data must be stored somewhere, either from a third party system, a database or another service, which makes it important to always sanitise the data [Laigner et al., 2021; Pautasso et al., 2017].

Since most microservice applications use containers and Virtual Machine (VM)s for deployment, all **issues raised from virtualisation** must also be taken into account [Dragoni et al., 2017; Zimmermann, 2017]. This includes, but is not limited to, container-to-container, container-to-host, and host-to-container attacks. Work has been done with respect to container escape, such as gaining root privileges outside the container, to prevent these attacks by inspecting the namespace status of the process [Jian and Chen, 2017].

Faults in software engineering correspond to a software defect that may lead to an error, which is an unstable or unknown state of the system. This error can then manifest itself as a failure when the system deviates from the expected behaviour [Grottke and Trivedi, 2005; Munson et al., 2006]. **Aging** corresponds to the accumulation of resources due to these software faults over long execution times, caused by many factors such as memory leaks or unreleased locks [Grottke et al., 2008a; Huang et al., 1995; Parnas, 1994]. Although aging has been extensively studied in software [Castelli et al., 2001; Garg et al., 1998; Grottke et al., 2008a; Huang et al., 1995; Parnas, 1994], little work has been done on microservice architectures [Flora et al., 2021; Yue et al., 2020a]. Kubernetes has been evaluated to see how probes deal with aging, and has been shown to fail in identifying this issue [Flora et al., 2021]. However, the same study shows that fault injection to accelerate aging is a fair technique and gives developers and researchers a way to evaluate their approaches to deal with aging [Flora et al., 2021]. Since many cloud deployments with high availability are currently destined for long-term

deployments, **microservices are prone to displaying this issue**. Therefore, it is extremely important to analyse them in order to prevent complete service failures leading to availability problems [Papapanagiotou and Chella, 2018; Štefanko et al., 2019; Wang et al., 2020].

As mentioned previously, one of the topics regarding security is availability; more specifically, a system needs to be available and ready to be used by users. Availability can be affected by various issues, such as attacks and software faults. Faults are software bugs that lead the system into an erroneous state, which may lead to the occurrence of a failure. Some faults can cause increased resource use, which, when accumulated, is referred to as aging [Parnas, 1994]. Since these faults usually only result in a failure after long periods of time, they often go undetected during testing and appear in production environments. Faults are inevitable; therefore, building resilient software is a must, in order to prevent or reduce damage. One way to tackle the aging process is by applying rejuvenation techniques, which are techniques that mitigate or reduce the effects of aging. An example of a software rejuvenation technique is simply restarting an application and returning it to the initial state.

Software aging is a well researched topic with several papers on the matter [Grottke et al., 2008b]. Some of the work performed was done on real case scenarios. A model was proposed for continuous monitoring, detection of anomalies, and automatic application of measures that adapt the system to sudden changes [Magableh and Almiani, 2020]. The proposed model follows six steps that perform data collection and model training, which is further applied to detect anomalies that motivate the application of adaptations. These adaptations are selected in the fourth phase of the process, are then executed, and finally verified. The results presented in the paper show a high success rate in enforcing vertical and horizontal scaling of the infrastructure in varying contexts.

Monitoring models for detecting aging, such as analytical modelling and measurement-based approaches, were also analysed and compared [Trivedi et al., 2000]. Although the former performs better in the evaluation of the trade-off between performance and availability, the latter detects aging with more effectiveness. Other models were studied with Web Servers, some of them being performed on Apache [Grottke et al., 2006; Li et al., 2002]. One of the main conclusions is that overwhelming workloads cause Apache to have some traces of aging, but also minor rejuvenation [Grottke et al., 2006].

Lately, there has been research focused on virtualisation platforms. Docker aging effects were evaluated and it was possible to observe aging effects [Torquato and Vieira, 2019]. Other work demonstrated problems not directly related to Docker, but it would also affect the ability to create new containers [Oliveira et al., 2020].

In regards to microservices, research on the subject is lacking, with only a single paper on the matter. It applies previous research on aging, but in a microservice scenario, using a deep learning model, showing good results [Yue et al., 2020b]. However, rejuvenation was not treated as thoroughly and only some concepts were analysed. It is a good starting point, but it shows how the literature is behind in this specific field.

Container orchestractors, such as Kubernetes, have been the target of various studies, focussing mainly on scalability. Horizontal Pod Autoscaler (HPA) is a feature that allows services to be automatically scaled according to demand, performing a service scale-up after a certain amount of time receiving a continuous stress workload, therefore, preventing unnecessary scale-ups during relatively short peak intensity loads. The same applies to scale-downs. HPA has receiving much attention and has been intensively and extensively researched, allowing developers to achieve greater flexibility and effectiveness [Khaleq and Ra, 2019]. Probes are another feature of kubernetes that allows service monitoring, by performing periodic health checks, being then able to execute predefined actions as an answer to any unhealthy service detection [Kubernetes, 2020]. Lastly, it is also worth noting that Kubernetes has some self-healing capabilities that have also been a focus of research, where services were automatically redirected to healthy pods by changing their labels [Abdollahi Vayghan et al., 2019].

2.4 Intrusion Detection Systems

Intrusion Detection is one of the mechanisms used to increase the security of a system, with the goal of detecting intrusions in the system. Intrusions are attackers who want to defy the fundamentals of security within the system, for example, by abusing a vulnerability in the system and / or bypassing the security measures that were implemented [Bace and Mell, 2001]. An IDS is a system that is specialized in Intrusion Detection.

IDS can be divided into three different categories, considering the information that is being monitored: network, where the system is focused only on network traffic; host, in which the IDS looks at data generated by the application host; and finally application, taking into consideration only the data that are being outputted by the application [Bace and Mell, 2001]. Then IDS also needs to analyse the data and come up with a conclusion; to do that, it can look at a profile describing a list of possible attacks, and all events that match this profile are considered intrusions. This is called signature or misuse based detection.

Snort is an open source network intrusion detection and prevention system that can be deployed with three different uses: as a packet sniffer, a packet logger, and finally as a network intrusion detection and, optionally, a prevention system [Snort, 2022]. Kumar et al. applied it in a small application to test its effectiveness [Kumar and Sangwan, 2012]. Overall, although they state that they were able to detect intrusions in real time traffic, the results are not sufficient, as there are no data related to the number of attacks that were performed, nor a statement on the amount of false positives [Kumar and Sangwan, 2012]. In addition, a problem arises with the use of signature-based detection, that is, zero-day exploits or abuse of permissions, as this type of detection cannot detect them [Benaicha et al., 2014; Garg and Maheshwari, 2016; Jyothsna et al., 2011; Kumar and Sangwan, 2012]. On a more positive note, this detection, when the profile is clearly defined, can be negligible False Positive Ratio (FPR) [Garg and Maheshwari, 2016; Kumar and Sangwan, 2012].

Alternatively, it can use an anomaly detection approach, where it uses a profile created with benign information collected from the system during a training phase and if the events do not match the profile, it is considered an intrusion [Bace and Mell, 2001]. This category features a wide range of techniques constantly evolving; therefore, it is difficult to create a taxonomy, varying with the opinion of the authors [Axelsson, 2000; Garcia-Teodoro et al., 2009; Gyanchandani et al., 2012; Jyothsna et al., 2011]. In general, it is agreed that there are three main types: Statistical, Knowledge and Machine Learning. However, the Intrusion Detection community does not accept anomaly-based solutions, preferring signature-based approaches, due to their generally low number of false positives, making it easier for administrators and users [Jyothsna et al., 2011].

However, work has been carried out that unifies both types of detection, having a profile for misuse behaviour and another for benign behaviour, called hybrid detection [Depren et al., 2005; Hajisalem and Babaie, 2018]. For example, Hajisalem et al. take advantage of two algorithms, ABC and AFS, which are two swarm algorithms, creating the hybrid ABC-AFS algorithm [Hajisalem and Babaie, 2018]. Based on the training dataset, they generate a set of rules for both anomaly and benign activity and then apply the algorithm to obtain the final rules [Hajisalem and Babaie, 2018]. These results are then applied to the training data set to obtain the various metrics and then compared with the testing dataset. The results show detection and accuracy rates higher than `0.986` and FPRs lower than `0.002` for the same configuration [Hajisalem and Babaie, 2018].

The history of IDS was analysed, with a description of the work already done [McHugh, 2001]. It is noted that after the late 1990s the evaluation of IDS started becoming a matter of research, creating doubts on its effectiveness [McHugh, 2001]. It showed that improvements are still to come, with some systems displaying results of only detecting 3 attacks out of 4, with a high false alarm rate [McHugh, 2001]. Nevertheless, IDS has been researched for several decades, with a large amount of work done on the topic.

When it comes to microservice architectures, network-based intrusion detection systems have received more attention [Li et al., 2018; Liang et al., 2021; Zhang et al., 2017]. For example, one of the works tries to deal with two issues: the effectiveness of Network-based Intrusion Detection System (NIDS); and the non-monolithic provisioning, which causes many issues, such as the need for scalability to deal with the higher amount of data [Li et al., 2018]. In the end, they were able to get really good detection rates, with good improvements in packet processing time and bandwidth usage, but they failed to provide in the results what their FPR, or their precision [Li et al., 2018]. When it comes to Host-based Intrusion Detection System (HIDS), work on microservices is lacking. The work done focusses only on containerised systems, such as MySQL and MariaDB [Abed et al., 2015a,b; Flora et al., 2020; Röhling et al., 2019]. Bags Of System Calls (BOSC) is used on an anomaly based IDS, to create a benign profile displaying more than 90% detection rates, and with a FPR of around 2% [Abed et al., 2015a]. BOSC is an algorithm that passes through the system calls with a sliding window, recording the frequency of each system call in the sliding window, and, when testing, if a sliding window appears that is not represented in the database, it is consid-

ered an intrusion [Abed et al., 2015a; Flora et al., 2020]. Sequence Time Delaying Embedding (STIDE) is a very similar algorithm, but takes into account the sequence of system calls in each window, so it has a bigger profile [Abed et al., 2015a; Flora et al., 2020]. Another algorithm also tested for containers is Hidden Markov Model (HMM), a statistical model that also takes sequences of system calls into account to build a model [Abed et al., 2015a; Flora et al., 2020; Wang et al., 2004]. When testing, a probability threshold is defined; if the sequence falls below that probability, it is deemed an intrusion [Wang et al., 2004]. These algorithms were used to evaluate their effectiveness on a MariaDB containers, using three different representative workloads based on the TPC-C benchmark [Flora et al., 2020]. The results show that HMM shows poor performance compared to both BOSC and STIDE, which were able to achieve, on Docker, `0.971` recall and `0.903` precision values [Flora et al., 2020].

To evaluate IDS, many metrics can be used, being the most common: recall 2.1 (also called detection rate), precision 2.2, f-measure 2.3 and FPR 2.4.

$$recall = \frac{TP}{P} = \frac{TP}{TP + FN} \tag{2.1}$$

$$precision = \frac{TP}{TP + FP} \tag{2.2}$$

$$F_1 = \frac{2 \times TP}{2 \times TP + FN + FP} \tag{2.3}$$

$$FPR = \frac{FP}{FP + TN} \tag{2.4}$$

In general, research is lacking when it comes to microservices as a whole, with all the work being applied in containers. Therefore, we will try to move away from single attacks that target a specific service, and focus on multi-service attacks and HIDS.

2.5 IDS Effectiveness Evaluation

To assert the effectiveness of these intrusion detection techniques, developers and researchers must test the system under a representative scenario of real-world attacks on vulnerable systems.

2.5.1 Vulnerability Discovery and Exploitation

To do that, vulnerabilities and their exploits must be found. Although vulnerabilities are common, Proof Of Concept (POC) exploits are hard to find, and even harder to make. However, they allow developers and researchers to emulate a real world attack, which can help developers to test the security mechanisms they have in place, and researchers to evaluate the techniques they are proposing. It is also important for developers to keep track of vulnerabilities, as they are inevitable and expected to be part of any project of considerable size.

Manually assessing the vulnerabilities inside a system can be time consuming and is prone to errors. Vulnerability assessment tools allow for an automated way to find vulnerabilities within a software. Due to the nature of microservices, a considerable amount of services have to be looked at, with each service possibly having using different technologies or development languages. Therefore, the amount of work scales astronomically, being unfeasible to be done manually, raising the importance of such tools.

Also, simply exploiting random vulnerabilities, although representative of different attacks, may not be representative of a real attack. Typically, an attack to a system has weeks, months, or even years of prior planning, with information gathering and infiltration. Knowing the attack paths of the attackers gives a huge advantage in regards to the defence mechanisms that are employed. There are techniques that allow software engineers to understand the attackers and what part of the system is more vulnerable.

Various tools and techniques were then evaluated to determine the best choice for our study.

An **Attack graph** is a graph designed to trace different paths that an attacker can take in order to breach the system. It is a technique that enables penetration testers to gather more information and analyse the security of the network in order to then take action accordingly. These used to be drawn by hand, but as it became more critical and important, tools started appearing enabling the automatic generation and also management of these graphs [Sheyner and Wing, 2004; Swiler et al., 2001]. Previous work shows the use of model checking as a means of generating attack graphs based on both intrusion and normal user behaviour. The authors also show how their technique for minimising attack graph analysis can be used to more precisely determine: which measures should be taken into account in order to better secure the system, but also what needs to be done to prevent a specific set of attacks [Jha et al., 2002a].

MulVAL is an example of an automated attack graph generator, available on their website and on github [Romano, 2015; Xinming , Simon]. It was released more than 10 years ago, with the last update being 7 years ago and, for the most part, deprecated, since it no longer works with current applications and technologies. MulVAL has been used to reach higher levels of scalability, while also clarifying the user, displaying the system configuration responsible for a certain vulnerability [Ou et al., 2005, 2006; Xinming , Simon]. It can be used to output both a text, with all the vulnerabilities, and a visual representation of the graph. This tool also supports the user with a script that provides a quantitative risk assessment algorithm based on the severity of the vulnerabilities to better understand the main issues of the network. Other tools have used MulVAL to try and perfect it, building an abstration layer on top of it, however, these are also deprecated [cyberImperial, 2019].

However, all this work is focused on traditional systems and networks. Although they can be applied, in theory, to microservice scenarios, some assumptions may differ, such as the way the attacker enters the system. Therefore, **attack graphs generators for Microservices** is a subject that lacks attention, with only one ex-

ample found, which was untouched for a year and a half [Ibrahim et al., 2019b; Khalifah, 2018].

The theory work done in regards to the architecture and methodology used is remarkable, with each phase being well detailed and explained. Unfortunately, the attack graph generator has an enormous amount of bugs. Although some may be related to some natural deprecation of unmaintained software, it is also easy to spot that it was probably built under a time restriction and the developer also lacked the ability to perfect and take care of the software [Ibrahim et al., 2019b; Khalifah, 2018]. For example, Clairctl was one of the choices used for vulnerability assessment. It is a command-line tool that enables the use of registries such as Docker Hub with Clair, which is a vulnerability scanner specialised for containers. However, Clairctl has been untouched for almost 5 years, and currently, it is unnecessary, since Clair basically implemented its features. Although some time was invested to fix and upgrade the attack graph generator, we quickly felt like it was too much work, requiring a whole masters thesis to completely remake the software from scratch [Gonçalves, 2022].

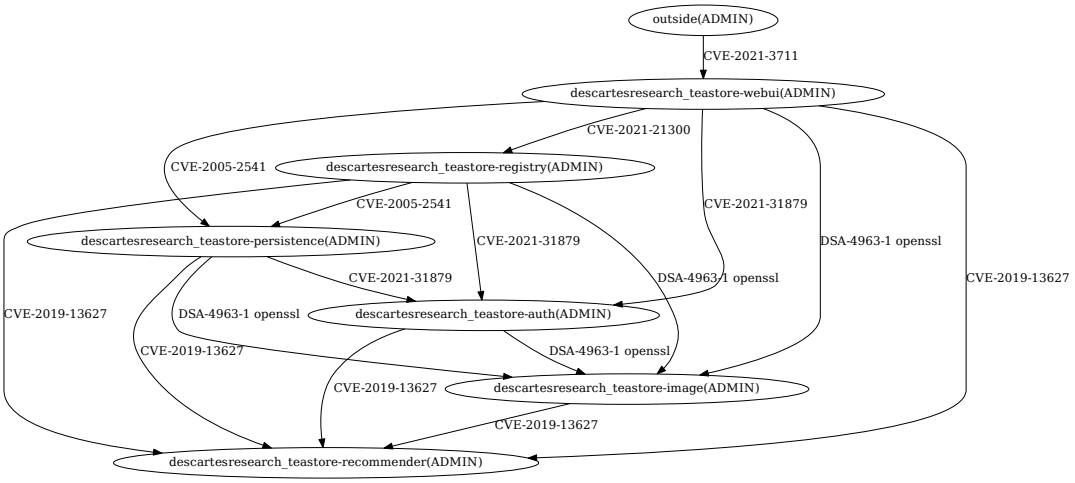


Figure 2.2: Example of an attack graph by an attack graph generator for Microservices [Gonçalves, 2022; Khalifah, 2018]

Bloodhound is a really good and well accepted tool, used by a considerable number of penetration testers [BloodHound, 2022b]. It is more focused on the authorisation part of an organisation; however, it still provides good information on how an attacker can enter a system. An example of its power is the large amount of queries it allows, even within large networks with many connections [BloodHound, 2022a]. Bloodhound also comes with a very detailed and well built GUI, that eases the various tasks a user can do, allowing for a faster identification of privilege escalation issues that a network may have. As of now, the tool continues to receive regular updates [BloodHoundAD, 2022], with recently introduced support for Microsoft Azure. Unfortunately, it does not quite help us exploit the system, by not giving a complete vulnerability assessment but instead focussing only on the different users and groups inside a network.

Snyk

Snyk is a company that was created in late 2015 and has been gaining traction, continuing to do so [Podjarny, 2021]. Snyk offers many different products, but all are related to the security of container applications, either from the source code, or even by looking at cloud configurations. It supports a wide range of languages and deployments, being almost all the major, well known and used systems covered. One of the products offered by Snyk comes within the docker package, allowing the vulnerability assessment of any docker image. By specifying an image to Snyk, it will look at the dependencies and packages installed, compare it with a database full of vulnerability information, and then outputs a list of vulnerabilities, sorted by severity, with detailed information regarding each of them.

Due to the very consistent and large database, it is able to provide a complete and detailed list that the developer can then take a look at and look for fixes. Although it helps with the vulnerability assessment of various containers, being also scalable for systems with a large number of microservices, it lacks the ease in regards to exploitation, leading to a neverending report of vulnerabilities that need to be checked one by one in order to find a working POC.

Metasploit

Metasploit allows researchers and pentesters to test various vulnerabilities in the system, by giving them all the tools needed for that job. It is a very complete tool that gives a reasonable set of exploits that can be used by anyone who downloads the tools. Additionally, it offers a wide range of payloads while also allowing custom made payloads that can be paired with the exploits. There are still many other features, such as brute forcing well known passwords of common systems. The approach here is a little bit different, focussing on finding a list of exploits first and only then checking for vulnerabilities. This eases the problem, as the number of exploits is quite small compared to the number of vulnerabilities.

2.5.2 Classical attacks and their applicability

Reviewing the challenges and literature, many classic attacks are relevant. In this section, we review some of these attacks directed at software applications or systems and their applicability in a microservice architecture.

A **Denial of Service (DoS) attack**, as the name suggests, is an attack that aims to disrupt the availability of a service, preventing legitimate users from accessing the service [Mallikarjunan et al., 2016]. Often these attacks are exacerbated by the use of many computers or devices, usually compromised by the attacks and without the owners' awareness, therefore being called Distributed Denial of Service (DDoS) [Mallikarjunan et al., 2016]. The DoS of a critical system can be extremely problematic, leaving many users hanging and leading to money losses [Specht and Lee, 2003]. DoS can be done at a network level, potentially being amplify as a Distributed Reflection Denial of Service (DRDoS) attack, which exploits other systems' services in order to reflect the attack and amplify it [Ryba et al., 2015; Specht and Lee, 2003]. However, these attacks are noisy, due to the high amount of packets associated with them, and their distinctiveness from nor-

mal usage [Mallikarjunan et al., 2016; Mantas et al., 2015]. The application layer DoS on the other hand is much more stealthy for the most part, due to the fact that it is disguised by normal behaviour [Mantas et al., 2015]. A taxonomy was created to better understand and specify DoS attacks [Mantas et al., 2015]. For a microservice based system, both network and application based DoS are applicable, and a DoS to a single service can have many effects on the other services.

Asymmetric Workloads are a special type of application layer DoS that aims to produce a large amount of work for the victim, with only a single request [Mantas et al., 2015]. This causes a significant reduction of the available resources, with the use of a low attack traffic rate disguised as normal behaviour, making it harder to detect [Mantas et al., 2015]. If an attacker knows the architecture and its services, he will have an easier time generating an asymmetric workload capable of slowing down the system.

Malware can be used to enforce a DoS, either by creating various botnets that then execute a DDoS, or even by infecting machines with a worm, which will then activate itself on the target system, causing a DoS [Antonakakis et al., 2017; Baezner and Robin, 2017]. Malware is usually made up of two parts, a decryptor and the encrypted body [Sharma and Sahay, 2014]. When malware infects a computer, it uses the decryptor to obtain the decrypted body, and then executes itself [Sharma and Sahay, 2014]. Antiviruses have a signature for the virus, and when a file is searched, the antivirus checks if a certain file has that signature, if it has, it is considered a virus [Sharma and Sahay, 2014]. Another form of malware, to disguise itself from antivirus, comes as oligomorphic, polymorphic, and metamorphic [Sharma and Sahay, 2014; You and Yim, 2010]. Oligomorphic has a various range of descriptors that it can choose when replicating itself, however, they are only changed slightly and are limited to just a few hundred descriptors [Sharma and Sahay, 2014; You and Yim, 2010]. Polymorphic tries to fix these issues by having countless possible decriptors, by adding obfuscation code [Li et al., 2011; Sharma and Sahay, 2014; You and Yim, 2010]. Antiviruses then evolved to be able to decrypt the virus in a sandbox environment, where no harm can be done, and apply their signature based detection on the now decrypted virus [Li et al., 2011; Sharma and Sahay, 2014; You and Yim, 2010]. So then polymorphic evolved into metamorphic virus, which does not have a decryptor and is constantly changing, applying obfuscation techniques to itself, changing its body when replicating [Li et al., 2011; Sharma and Sahay, 2014; You and Yim, 2010]. In theory, it is possible to generate a metamorphic virus that enters a microservice and attacks all the other services, with different behaviours per service.

Side-Channel attacks have the goal of obtaining information via a method not intended for that purpose, such as cache in a cache-based side-channel attack [Godfrey and Zulkernine, 2014]. It was even possible to break the Data Encryption Standard (DES) algorithm, used for, as the name suggests, encryption [Tsunoo et al., 2003]. This type of attack is even more important in cloud environments, where tentants are sharing the same resources and, therefore, sharing, for example, cache between different cores, such as LLC, thus giving attackers information regarding their neighbors [AlJahdali et al., 2014; Godfrey and Zulkernine, 2014; Liu et al., 2016]. A tenant can deploy a microservice based system, therefore being

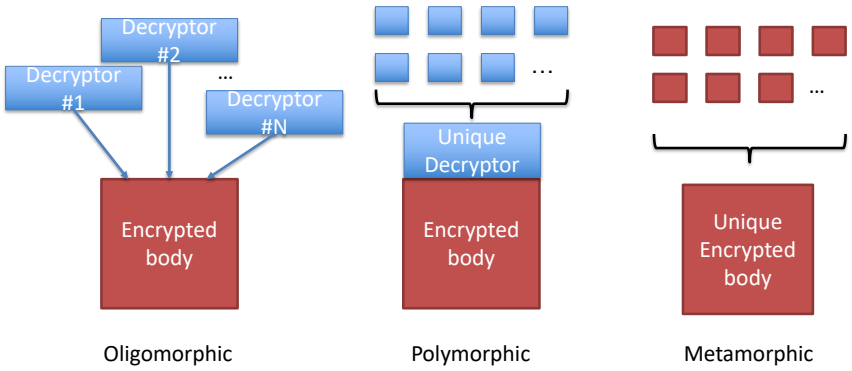


Figure 2.3: Oligomorphic, Polymorphic and Metamorphic malware

susceptible to these types of attack.

Current enterprises currently feature complex networks, with many systems that integrate them, making it difficult for security experts to protect these systems [Chen et al., 2007; Jha et al., 2002b]. Therefore, they often resort to techniques such as vulnerability scanners, which allow the automatic identification of vulnerabilities of the different systems, and attack graph generators, which generate a graph where each edge is a possible security exploitation [Chen et al., 2007; Jha et al., 2002b]. Security experts can then analyse the data more easily and draw many conclusions, such as: what is the minimum number of steps required in order to secure the system [Jha et al., 2002b]. **Sequential attacks**, attacks being performed in quick secession, can then be extracted and performed through the analysis of these attack graphs. The generation of attack graphs for a microservice based system will be conceptually similar to what is being done on enterprises [Ibrahim et al., 2019a].

Remote Code Execution (RCE) is an example of a classic vulnerability in which the attacker can execute a piece of code on the target. This type of attack allows for new targets, since the compromised target can be used as a vector for a new attack on another one [Biswas et al., 2018]. Considering a microservice architecture, given that most containers have good defined boundaries, with a very low number of open connections possible, therefore making it hard to get into one. Remote Code Execution then becomes a problem, as it is much more appealing to attackers, simply due to the opening of many more possibilities.

Second order attacks are attacks used to pass through the initial security provided against first order attacks, requiring two steps, inserting the data, and activating the attack [Liu and Wang, 2018; Ping, 2017; Ray and Ligatti, 2012]. These attacks mask the initial intent and are only visible after the attack has been performed, making it difficult to analyse and detect [Liu and Wang, 2018; Ping, 2017]. Most of the work done is related to SQL injections; however, RCE and Cross-Site Scripting (XSS) are also possible [Liu and Wang, 2018; Ray and Ligatti, 2012]. In a microservice architecture, since components are separated, data processing is also separated from sources, and by using message queues and third party APIs the risk of these attacks is exacerbated [Laigner et al., 2021; Pautasso et al., 2017].

Another example of a classic vulnerability is **Privilege escalation**. It is a vulner-

ability that enables an attacker to overcome a missing privilege necessary for an action. One of the largest areas of research on privilege escalation is within the cell phone environment, more specifically, Android phones [Bugiel et al., 2012; Davi et al., 2010; Rangwala et al., 2014]. On the other hand, considering the miceroservices scenario, containers may not have access to each other, being only able to communicate through a single channel. Privilege escalation vulnerabilities enable an attacker to gain unauthorised access to another container, therefore being able to crawl through the network and the system.

Information Disclosure is normally considered a more tolerant vulnerability, usually showing low CVE scores in terms of severity. This type of vulnerability allows attackers, as the name suggests, to gather information regarding their target. It may not be a problem at first; however, it allows attackers to gather many little details during an initial scouting phase, to then find vulnerabilities and exploit them [Dibbets et al., 2021; Gionta et al., 2015]. It should be noted that microservices are virtualised and, usually, like previously stated, tightly closed, making it harder for attackers to find vulnerabilities. Therefore, these vulnerabilities are expected to be exploited to obtain information about the system and then plan the attack.

Finally, **XML external entity injection (XXE)** is a type of attack where the attacker takes advantage of a poorly protected system that does not prevent external entities at the time of parsing, therefore forcing the server to system them, opening up doors for other issues such as information disclosure and RCE. These vulnerabilities continue to appear, even the most popular XML parsers and open-source software that use them [Jan et al., 2015].

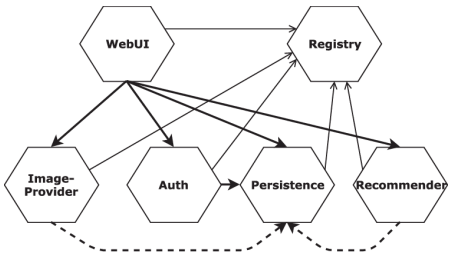
To evaluate the proposed solutions, a system needs to be attacked and the traces need to be collected. For the scenario to be representative of a real case, the chosen system has to also be representative.

2.5.3 Representative Testbeds

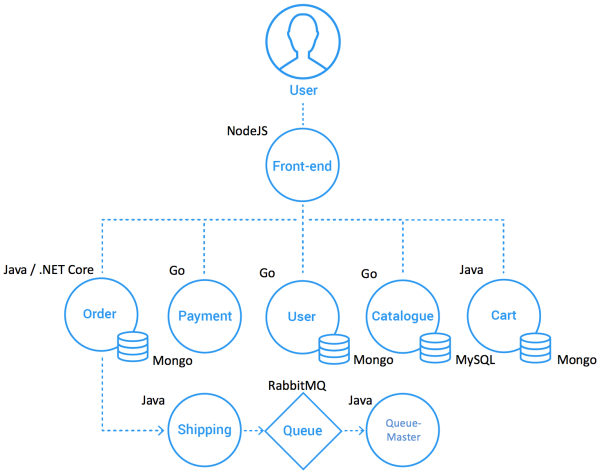
Various testbeds were analysed to understand how fruitful they would be to our work, analysing their representability, their complexity, and the feasibility of the attacks.

TeaStore is a microservice based testbed that simulates an online tea store, and it was one of the chosen. Although it has been more focused on benchmarking and testing, some work has been done in regards to security. It was built in Java, using Jakarta, and is composed of five services, a Registry, and a database. These services are as follows: WebUI which is responsible for the front-end and communicates with all the other services; Image-Provider, taking care of rescaling the images as requested by the WebUI; Auth, responsible for authenticating the users and managing their sessions; Persistence, a middleware to the database, being the only service that requests data directly to the database; and the Recommender, which sends information to the WebUI regarding which teas to recommend, based on past purchases and reviews. Every request is done using REST, and all services are deployed on Apache Tomcat.

Sockshop was the other testbed chosen. In this scenario, a sock shop is simulated. It is slightly more complex than TeaStore, having eight different services, four databases (three MongoDB and one MySQL), and a RabbitMQ message queue. The first service is the FrontEnd, a NodeJS server, which then communicates with five different services. Cart, responsible for dealing with the carts of the users; Catalogue, which stores information regarding all the socks that can be bought or viewed; User, taking care of the authentication of the users and their sessions; Payment, a simple service that executes the payment; and, finally, Order, responsible for ordering the products. While the Catalogue, User, and Payment services use GO as the language of choice, Cart and Order both use Java, with Payment also using .NET Core. The order service also communicates with another service, Shipping, built in Java, that takes care of the shipping and sends the details to the message queue, which will then be read by the Queue-Master, also built in Java. We can see both architectures in Figures 2.4b and 2.4a.



(a) TeaStore Arquitecture [von Kistowski et al., 2018]



(b) Sockshop Arquitecture [Sockshop, 2017]

PetClinic was considered, being a distributed version of the Spring PetClinic Sample Application, using Spring Cloud. It has a total of four business services and four infrastructure services, which includes a discovery server, a configuration server, the spring boot admin panel server, and a tracing server. Since this testbed was similar to the testbeds already chosen, we opted out due to lack of experience. Figure 2.5 shows the architecture of the testbed.

Trainticket was also considered due to the large amount of services, 41, therefore being more representative of a real scenario. However, the deployment, although possible, required considerable amounts of resources, even without workloads running. Although tests were not made to determine whether it was possible to experiment using it, we considered that it would not be worth doing so, at least for now. One of the reasons comes from the fact that a profile would have to be built and then argued that it is representative. Trainticket architecture can be seen in Figure 2.6.

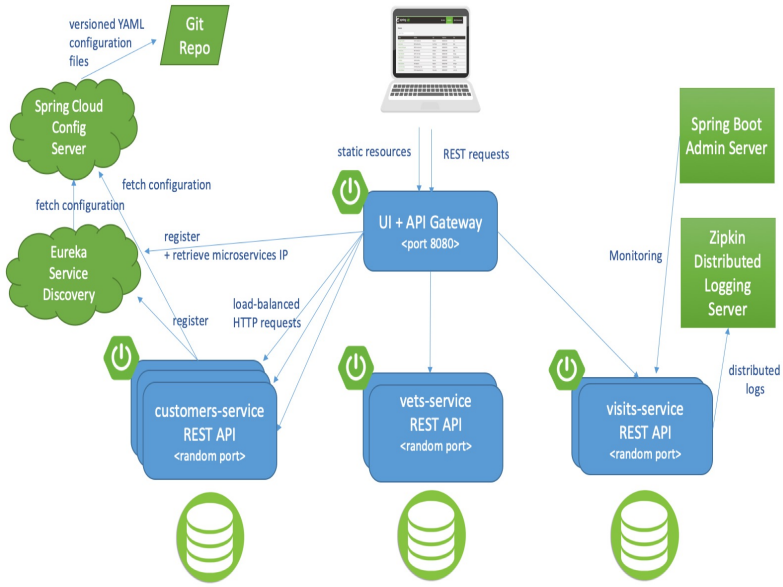


Figure 2.5: PetClinic arquitecture [PetClinic, 2022]

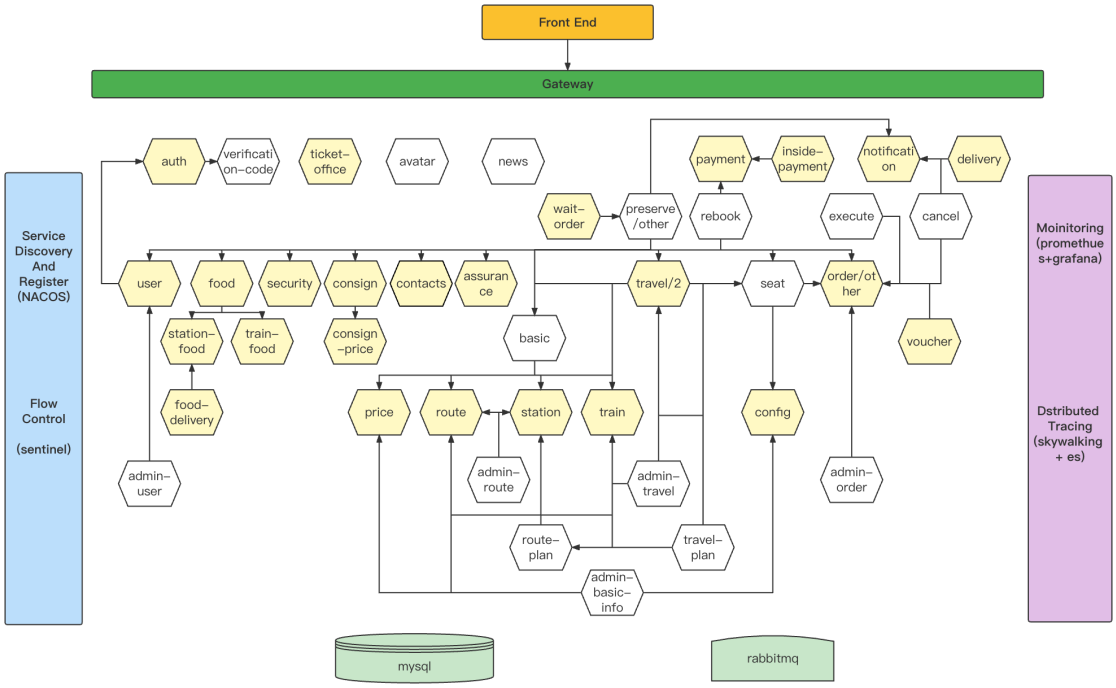


Figure 2.6: Train Ticket arquitecture [Train-Ticket, 2022]

2.6 Fusion Techniques

There is a set of techniques that are responsible for joining classifiers to give one single output. These techniques appear in the literature with many different names, from which we choose fusion techniques [Kuncheva et al., 2001; Polikar, 2006]. They have been used in various fields, such as medicine and even industrial fault detection [Chan et al., 2018; Guo et al., 2019a,b; Xu et al., 2020]. For example, neural networks, more specifically CNNs, have been used as a way

to fuse classifiers. Work has been carried out on a soft tissue sarcoma dataset, showing better results than single model classifiers [Guo et al., 2019a].

Fusion techniques can be divided into 3 different categories, according to the literature, based on the stage at which they are applied [Guo et al., 2019a]:

- **Feature Level**: The earliest possible stage, when all the features from different sources or with different information are fused together to form a single unique stream for the machine learning model;
- **Classifier Level**: This is a middle stage where some classifiers will be fused together or some features are fused, with the goal of generating new features, called single-modality features, that will then be used to train a new classifier to provide a final result;
- **Decision-Making Level**: At this point, all classifiers have already made their decision, so the technique will only be used to provide a final result based on one or various decisions.

They can also be divided into two different types according to the order of the fusion [Ruta and Gabrys, 2000; Uhl and Wild, 2009]:

- **Iterative (also called Serial)**: when the classifiers are trained/used iteratively, one after the other, meaning that a change in the output of the first classifier affects the input of the next classifier.
- **Paralel**: when the classifiers can be trained/used independently in parallel.

DIDS, Distributed Intrusion Detection System is a network based intrusion detection system that monitors distributed machines, while also employing data reduction and centralised data analysis. It uses a series of lan monitors and individual hosts to send information directly to the Expert System. This expert system is then responsible for defining the security of each individual host and aggregated the information accordingly. Then it infers, based on all the information collected, the state of the whole system [Jones and Sielken, 2000; Snapp et al., 1992].

NSTAT, Network State Transition Analysis Tool, is another tool that chronologically orders the information, giving it to USTAT. Hosts must have consistent sync messages between themselves to ensure that the clocks are synchronised [Jones and Sielken, 2000; Kemmerer, 1997].

EMERALD has various layers, going from services that do the same thing, moving up to domains that have similar purposes, and finally reaching the highest level, the enterprise, where the whole system is analysed. Layers can communicate with each other through a subscribe system between layers that is in place that allows or disables the flow of information [Jones and Sielken, 2000].

A series of meta learners were proposed and tested, showing a fusion of many classifiers and forming a single final meta-data answer.

Bagging is a technique inspired by bootstrapping, where all learning sets, although subsets of the initial dataset, are independent, as well as the techniques used, resulting in different models. These models then vote to reach a final decision. **Boosting** is based on the principle that weak learning algorithms will tend to perform better, until they can learn everything, allowing the algorithm to train a series of weak learners that can then vote and make the final decision. Very similar to Boosting, we have **Cascading**, a technique that takes advantage of the fact that models will have different certainties than others, so the models are tested in sequence, and when one displays a high level of certainty, above the pre-defined threshold, the answer is accepted. Cascading Generalisation tries to improve the previous version by adding the detail that each iteration of the model not only receives the initial dataset, but also any information the previous classifiers added. It also differs from the previous one in the sense that there are no longer thresholds and that all models have to be passed through. **Delegating** is a mixture of both, but has a whole other idea in mind. It uses the principle of divide-and-conquer, where each model will be trained for a specific part of the problem. Then, just as earlier, based on their confidence levels, the first model in the sequence to display a good certainty will be used for the answer.

Stacking instead uses the complete dataset, but different algorithms, forming a new single meta-dataset, that will then be used to train a final classifier. An adaptation can be made to also add the initial dataset, forming a larger and more complete meta-dataset. This is called Stacking Generalisation.

Voting and Arbitrating are the last two schemes. The first one is simple; each model is trained independently and votes. The difference from bagging comes from the fact that bagging uses only a subset. Arbitrating, on the other hand, has a new model, like, for example, a decision tree, that is called a referee. The referee is responsible for deciding which classifier fits the best and chooses it as the final classifier answer. The idea is that it splits the data into a subset that allows the chosen classifier to have a high certainty in the answer, while the remaining subset is where the classifier shows less confidence.

Some basic techniques have been studied. There was a theoretical analysis of six different simple classifier fusion techniques where the following were reviewed: minimum, maximum, average, median, and majority vote. Each one has its advantages and disadvantages, for example, using the minimum guarantees that when a positive answer is given, everyone agrees, while the maximum is the opposite. Although the median and the average work in pretty similar ways, there are cases, such as when one classifier might be shifting the results of the rest, where differences might be viewed. The sum can also be used and is the same as the average. This sum can then be linked to a decision model that uses a threshold to define the result. However, a weighted sum is an alternative, where each classifier can have a different weight. Finally, voting is a discrete and rough method for selecting a result, since a classifier that is near the threshold has the same weight for the output as a classifier with a more certain answer.

The **Bayesian combination rule** is based on the Bayesian combination rule. It starts with the assumption that the classifiers are independent, meaning that the choice of a classifier is only related to its training and algorithm and does not

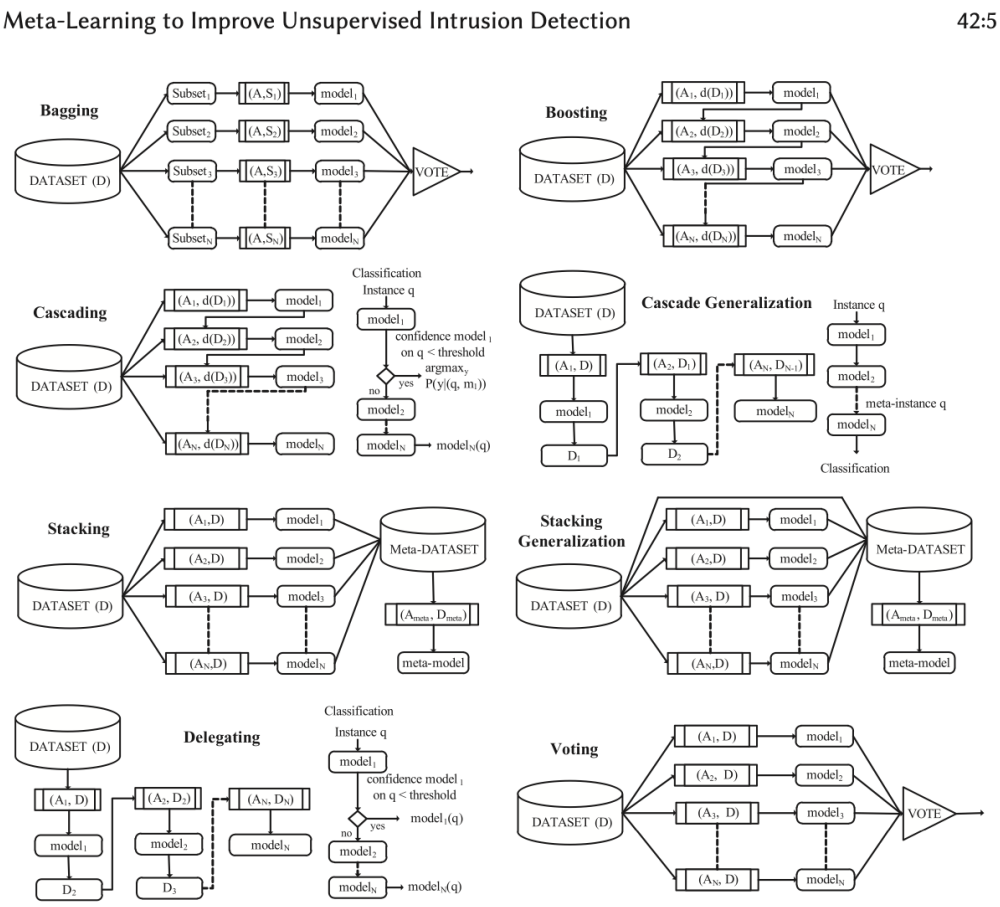


Figure 2.7: Meta-Learning schemes (from [Zoppi et al., 2021])

take into account what other classifiers decided. The algorithm uses the Bayes formula to determine a value that is related to the certainty of the result. This value is then compared to an adjustable threshold that defines when the result can be accepted. A genetic algorithm assigns a weight to the vote of each classifier (also called an expert), and this weight would be applied to all patterns regardless of the decision made by the expert.

The **ER rule model** uses a set of models that are fused together, taking into account their reliability and using adjustable weights. These weights can be adjusted using any algorithm, although a genetic one was used. Then it gives two outputs, one for the probability of having a value and the other for having another value. An example would be to get the probability of having an intrusion and another of normal behaviour. The higher probability defines the output of the device.

behaviour-Knowledge Space, or **BKS** for short, is a method that combines n experts in an n-dimensional matrix. Each point in the matrix, called the focal unit, is a combination of the answers of n experts. For example, a point (i,j), for a

2-dimensional matrix, represents the focal unit for when e(1) = i (expert 1 says it is i) and e(2) = j (expert 2 says it is j). A focal unit (i,j) holds 3 different values: the total number of incoming samples where expert 1 gives the output i and expert 2 gives the output j; the best representative class; and finally, the total number of samples from each class. To make a final decision, we first need at least one sample (T > 0). Then, the algorithm looks at the best representative class (R), and calculates the number of samples that represent the best representative class, divided by the total number of samples. If the result is higher than or equal to the threshold, then the class is accepted. If not, it is rejected.

The work carried out could not be directly applied to our scenarios. It focused on testing classifiers that are doing exactly the same thing, which is not the same as what we are doing here, since classifiers are totally different between services. Also, the fusion of information is mainly targeted to heterogeneous systems, which is also not our scenario. Therefore, these techniques were adapted to fit our scenario. The next section goes in depth into the previous work conducted, where the algorithms used are initially tested, and container scalability is dealt with. Then in Section 4 we return to these techniques, showing our four adaptations and how we evaluated them.

Chapter 3

Detecting Attacks in a Single Microservice

This chapter will elaborate on the work performed regarding the detection of attacks using an anomaly-based IDS, for single services in a microservice architecture. First, we present the work done regarding a simple system consisting of a MariaDB container, in order to provide a view of attack injection with the goal of evaluating container-based systems [Flora et al., 2020]. Then the work will be related to the detection of attacks performed on a service of a representative testbed, TeaStore [von Kistowski et al., 2018]. This work was presented in **PRDC 2020** and submitted in **IEEE S&P 2023**, respectively.

My contribution for the first topic, with regard to attack injection for container-based systems, was focused on the analysis of the results. It is worth noting that the amount of data collected was considerably large, with more than 230 thousand data points. For the second topic, regarding how to deal with the elasticity of microservices, I helped during the preparation of the experiments and the analysis of the results. The analysis was performed on over 1 million data points, therefore requiring careful attention and discussion.

3.1 Attack injection for container based systems

Taking into account the **approach to evaluate intrusion detection algorithms in container-based systems**, the development was made by Flora.

The goal of the experiments was to validate the approach in terms of its ability to compare and facilitate the evaluation of different intrusion detection algorithms. In addition, it allowed us to understand which algorithms and configuration perform better, showing that the approach allows us to compare and evaluate several intrusion detection algorithms.

My contribution was in regards to the analysis of the results looking at the different combinations and analysing how to present them. In total, more than 230 thousand rows of results were saved in a database. Each row relates to how

a classifier, trained with a specific training scenario, performs in a certain test scenario This enabled me to gain a feeling about the research being conducted, with great emphasis on results analysis.

3.1.1 Experimental Methodology

The experimental methodology was conducted as a way to help administrators who lack publicly available datasets or well-defined approaches that allow them to measure effectiveness and compare different solutions [Abed et al., 2015a]. The approach relies on a target system that is a representative service running on containers, which is subject to realistic workloads, and innovates in a novel view on false positive analysis, taking into account the characteristics of the containers. Testing workloads are generated by performing an attack injection technique in the middle of the experiment. Figure 3.1 depicts an overview of the proposed approach consisting of four main phases: 1) generation of a training dataset, using representative workloads; 2) generation of a testing dataset, by adding attack injection to the workloads; 3) train classifiers, using training convergence to decide the stopping point; 4) execute the testing phase to obtain results.

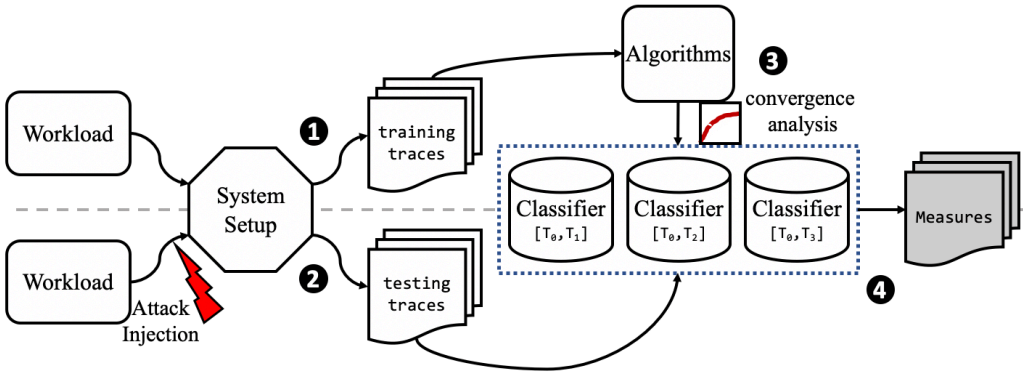


Figure 3.1: Overview of the proposed IDS evaluation approach.

In order for the classifiers to be complete, the workloads being used for training must also be complete. It is utterly important to carefully select the workloads, guaranteeing that realistic operations are being performed against the target system, thus conserving the representativeness of the evaluation and the validity and acceptance of the results. To select an appropriate stopping point for the training procedure, a convergence approach was adopted based on the slope of the growth curve of the profile database, proposed in [Milenkoski et al., 2015]. In the end, for both STIDE and BOSC, the best training time was determined to be around 24 hours when using docker.

In addition, the procedure for injecting an attack must be prepared taking into account the selection of the target vulnerabilities and the corresponding attacks. These must be realistic vulnerabilities that are representative of real-case scenarios, prioritising known ones in older versions of the systems. Finally, the moment the attack will be injected must also be selected.

The experimental campaign was depicted as follows: the collection of benign in-

formation (up to `24H`), and the testing which contained both benign and malicious traces (over `30min`), resulting from the attack injection procedure. The focus was on algorithms that were lightweight and effective in detecting anomalies in sequences of events. Therefore, two sliding window algorithms, STIDE and BOSC, were used, which group the system calls into a defined window size. In the first case, it stores the sequence of calls and in the second case, it stores the frequency of each call within a window. HMM, a stochastic-based algorithm that is based on a sequence of system calls, was also used. Classifiers were generated, using each algorithm with various configurations, on the basis of training datasets.

The state of the classifier was stored at each `6H` of training for BOSC and STIDE, and at `30min`, `1H`, and `2H` for HMM. The classifiers were then used to evaluate the test datasets to classify the events collected, giving measurements that will then be used to calculate metric values. Each collection consisted on starting the collection of data and the execution of a workload, and an injection attack procedure noticeably at the middle (`15min`) of the collection period (`30min`). The collection was then run through the algorithms that were trained using one of the workloads.

Three diverse workloads, based on the TPC-C benchmark, were tested to represent different interactions with the system. They represent a steady, a variable, and another variable workload with operator interaction. The experiments included all combinations of workloads. This allowed us to analyse the impact that the workload being trained has on the IDS in different scenarios.

The attackload consisted of a normal workload, but in the middle of the collection period (`15min`) the attack to MariaDB is made. The reasoning behind the time interval comes from the fact that it shows how the IDS performs before, during, and after the attack. This guarantees that it is possible to get representative data on how the IDS would behave if the attack were carried out on a real system. In total, five different attacks were performed, as we can see in Table 3.1 These were performed on MariaDB version 5.5.28.

Table 3.1: List of vulnerabilities used and respective CVE information.

CVE ID	Access	Vulnerability Type(s)	CVSS
CVE-2012-5611	Remote	Execute Code, Overflow	6.5
CVE-2012-5627	Remote	Execute Code	4.0
CVE-2013-1861	Remote	Denial Of Service, Overflow	5.0
CVE-2016-6662	Remote	Execute Code, Bypass	10.0
CVE-2016-6663	Local	Gain privileges	4.4

The other goal of this work was to compare intrusion detection techniques between three different deployments: Docker, LXC, and a traditional OS. These deployments cover the most popular container platforms and also portray a representative OS-based infrastructure; thus, all these environments represent real-world scenarios. Docker and LXC have different characteristics, even if both are container virtualisation software. Docker is designed to support only one application per container, known as Application Containers. LXC is more similar to VMs, although lighter and easier to create, known as OS of System Containers. To

collect the system calls from each deployment, `sysdig` was used due to its native support for containers. It also supports OS systems, allowing a correct comparison between environments. In containerised setups, we collected the system calls issued from the container as a whole. For the OS deployment, we collected the system calls issued by the MariaDB application: its root and child processes.

During the evaluation phase, every combination of the training workloads was explored with the testing workloads; thus, resulting in fifteen different combinations. In addition, many configurations of the algorithms were used. This generated a large amount of data, with more than 230 thousand entries.

3.1.2 Overview of the Results

The experiments show that the approach can be generalised, as being applied to three different scenarios (Docker, LXC, and OS), the results are still meaningful. More specifically, it demonstrates the general higher recall values in container-based deployments, regardless of the platform, compared to traditional deployments (OS).

The observation of the general results portrayed in Figure 3.2 allows us to understand some trends in our data. Namely, the improvement in precision values with more training time, as well as with more complex and complete workloads. This fact is mainly due to the greater amount and variety of data provided to the classifiers. In turn, this contributes to more stable and complete profiles with more capacity to distinguish malicious events from normal ones.

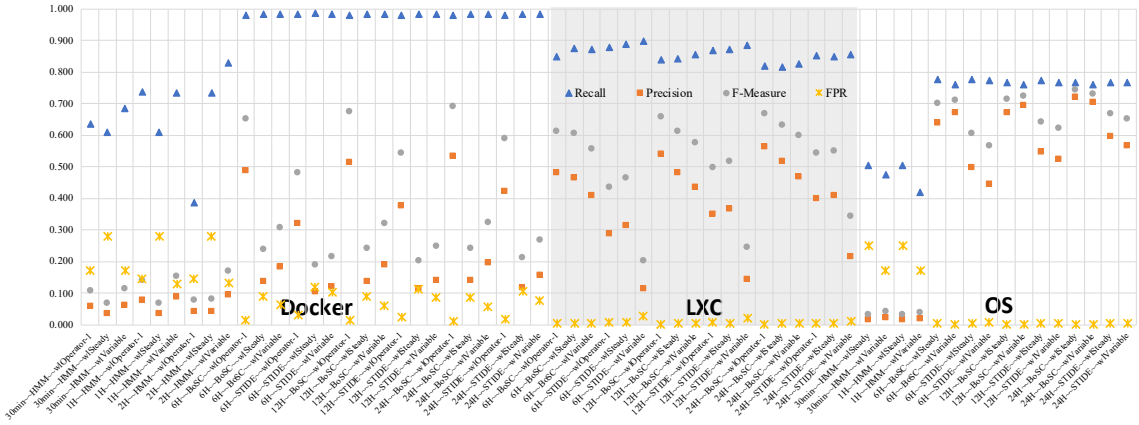


Figure 3.2: Overview of the experimental campaign results, aggregated by training time, algorithm, and training workload for each platform.

In general, and as shown in Figure 3.2, the recall values are mainly high (hitting low points only for HMM), remaining between `0.387` and `0.984` while the precision varies from `0.017` to `0.720` having at most a FPR of `0.281`. The analysis of the graph also allows us to understand the prevailing detection capacity of STIDE and BOSC over HMM in both Docker and OS deployments. In fact, HMM is responsible for producing the lowest results in both precision and recall.

Another interesting observation is that recall values are higher in container de-

ployments, in general, and in Docker specifically, demonstrating the impact that a well-defined monitoring surface has on the outcome obtained. Furthermore, the differences between Docker and LXC demonstrate that even between container solutions the numbers diverge. In this case, we believe that the main reason is the amount of data produced by each target under monitoring. LXC datasets were several times larger than Docker, on average [Flora and Antunes, 2019].

Despite the low precision values in some cases, we must note that these results are aggregated and contain all the testing workloads and configurations; careful tuning and configuration allows to obtain better outcomes. Then we analyse the data in more detail by filtering the best results at each step and going deeper into the configurations used to analyse their impact on the results and select the best ones.

3.1.3 Training Analysis: Configuration tuning

In the overall analysis, we were already able to observe some tendencies with regard to the impact of the amounts of time used during training. Now we will consider the test slots where the training workload is the same as the one used during the testing phase, in order to better understand the effects of the training time. First, we have to highlight that due to computation resources, HMM classifiers were only trained using `30min`, `1H` and `2H` of the data, which is significantly lower than BOSC or STIDE; hence, lower quality profiles and results.

However, classifiers whose training time was longer, and even in the case of HMM, demonstrate, across all platforms and configurations, a higher capability to detect malicious events, while also being more precise. Although the longer time does not significantly impact detection rates, the impacts on precision and FPR are visible. These observations are clear in Figure 3.3 where the increase in precision is noticeable, especially in the OS case.

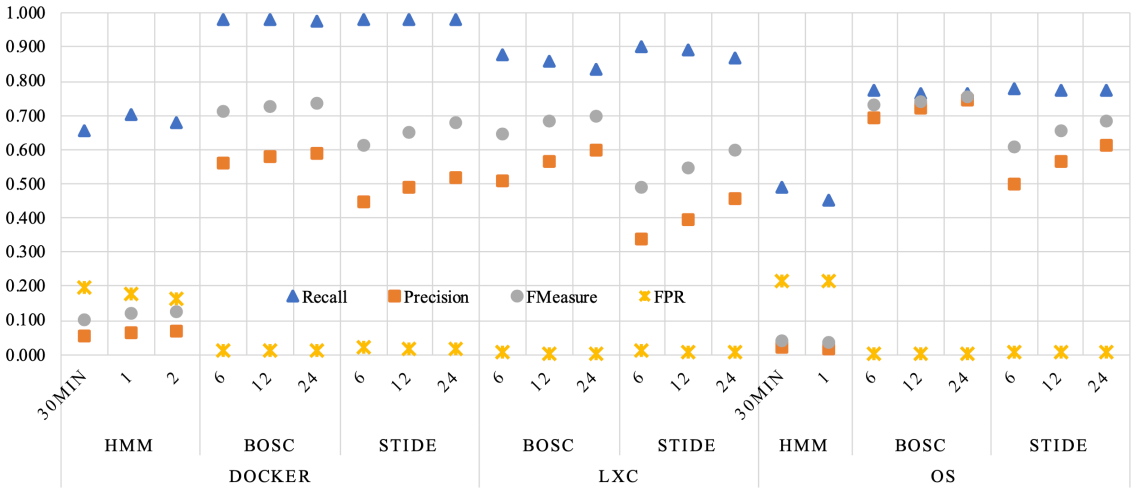


Figure 3.3: Impact of the training time in the classifiers effectiveness.

As we can see, Docker has the highest recall values for BOSC and STIDE, all close to `1.000`. For LXC, although worse, they still remain high (`0.800`). On the other

hand, the OS is only possible to get below `0.800`. For HMM the values drop around `0.300` for both OS and Docker deployments.

These results demonstrate the ability of the approach to show the importance of selecting an adequate training time. The information collected can be used to make a cost-effective decision given the multiple training times applied, allowing users to segment the training procedure and stop when the requirements are met.

Another important factor during training is the workload used. In this experimental campaign, three training workloads were used for LXC and Docker, while only two were used for the OS. The workloads are diverse, having different operation profiles and tasks. Figure 3.4 depicts the impact of each workload on the outcome of the classifiers evaluated for each platform.

The results show an increase in classification correctness when the most intense and complex workload is used (`wlOperator-1`). This workload is based on `wlVariable` and complemented with maintenance tasks conducted by an administrator. In both Docker and LXC there is an increase in precision values for both BOSC and STIDE. Again, HMM performs worse than the other techniques, showing slight increases in precision at high cost for detection rates.

For both LXC and OS, false positives continue low for every workload used, but for Docker, there is a clear decrease when more complex workloads are used. Additionally, BOSC achieves better results than STIDE, which may indicate that frequency-based techniques may be more applicable to this environment. It is worth noting that sequence-based techniques (STIDE), due to the higher number of possible combinations, are expected to take longer to learn.

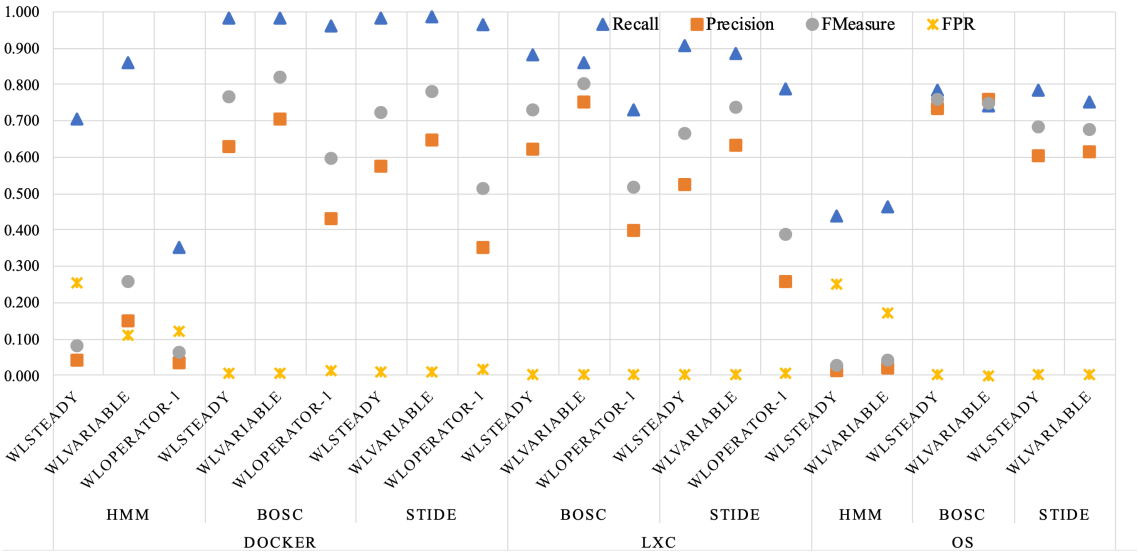


Figure 3.4: Impact of the training workload in the classifiers effectiveness.

In short, the data show better results for Docker in terms of detection and demonstrate the importance of considering several training times and workloads, both characteristics that are covered by our approach. Classifiers trained for longer periods of time and using more complex and diverse workloads show better results in terms of precision and FPR. The calculated values favour `wlVariable` and

`wlOperator-1` over the more simplistic `wlSteady`; nevertheless, the latter seems to provide LXC classifiers with more clear information owing to the results obtained, which in this case are better when this workload is used.

As mentioned above, our approach acknowledges that multiple workloads should be used during training and testing procedures. The results show that considering only one workload would contribute to misleading information; having workloads that contain representative operations and map with diverse operation profiles generates a more complete and fair comparison and evaluation of the algorithms utilised.

The algorithms selected and the methodology have multiple parameters that can influence the outcome. So, here, we analyse the impact of the window size, the epoch size, and the detection threshold on the final results obtained. Figure 3.5 depicts the results of the classifiers for the three deployments.

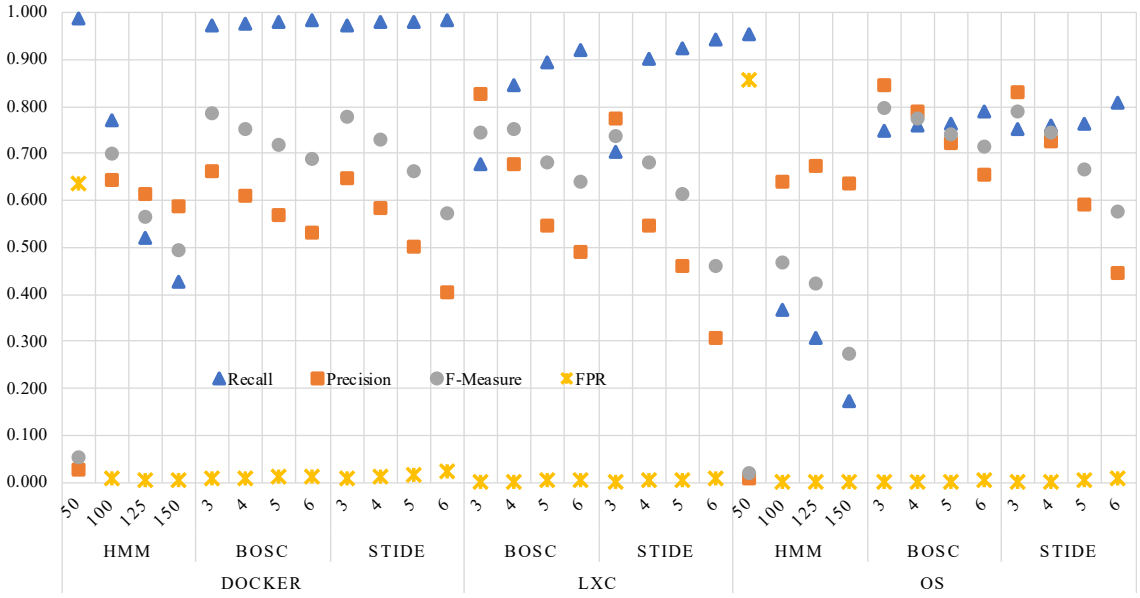


Figure 3.5: Impact of the window size parameter in the results.

We can observe a very low false positive rate across all classifiers, with the particular exception of HMM when the threshold used was `50`. This noticeably indicates that this decision threshold still remains within the normal behaviour range of operations.

The general impact of the window size remains constant; that is, with the increase of the window size, the level of recall increases (the algorithms find it easier to detect malicious events) while reducing the precision with which those are detected. The root cause for this observation lies in the stability and completeness of the profiles created; higher values for the window size increase the combinations of sequences a model can hold; thus, resulting in a more incomplete profile that frequently raises alarms of malicious behaviour that often are incorrect, hence precision goes down. In summary, to maximise both precision and recall, a smaller window size must be selected; `3` or `4` seem to be the most correct values in this case.

Other factors that impact the results obtained are the epoch size and the detection threshold. These parameters are applied in the final phase of the result analysis and their outcome is used for metric calculation. To analyse their influence, Recall-Precision curves were selected, so that it is possible to observe for multiple values the evolution of the detection capabilities.

Figure 3.6 shows the evolution of recall and FPR values over different combinations of epoch size and detection threshold for Docker and OS. The selected values for the epoch size are 500 and 1000 whereas all the values used for the detection threshold during the experiments are present in the graph.

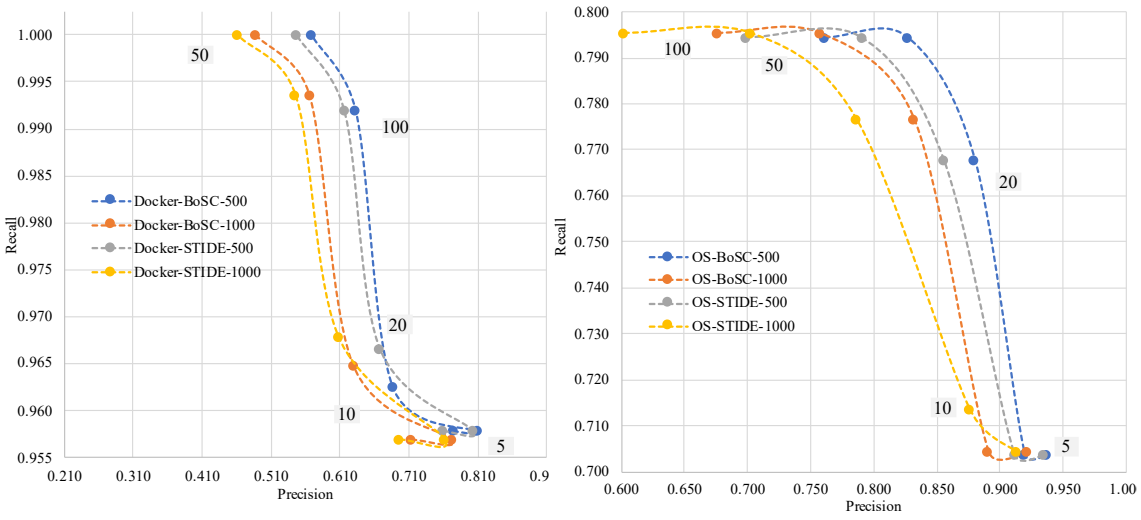


Figure 3.6: Docker and OS Recall-Precision curves for epoch size and detection threshold impact analysis.

The analysis of Figure 3.6 makes clear that, regardless of the epoch size, the evolution of the Recall-Precision curves is similar, that is, higher detection rates result in lower precision values. Here, a commitment has to be made when deciding whether to enforce higher or lower values for the epoch size. Still, the analysis of the epoch size shows a clear distinction since smaller values are able to maximise both metrics easier. For both algorithms (BOSC and STIDE) and deployments (Docker and OS) the smallest epoch size (500) produced a more fitting recall-precision curve, which means that this value produces results closer to the desirable, and hence can be considered better than the alternative. Additionally, despite similar behaviour, there are small differences between BOSC and STIDE in both deployments; BOSC performs slightly better than STIDE achieving values of higher detection rates with higher precision for the same configurations.

The underlying reason why smaller epochs perform better is due to the granularity level. As the number of events in an epoch is larger, there is a higher probability of overcoming the detection threshold, resulting in higher false positive rates for the same detection rate.

In summary, from the analysis of the outcome and the impacts of these three parameters, we can conclude that, in general, window sizes of 3 or 4 and an epoch size of 500 achieve higher detection rates. Hence, these are the prevailing values for obtaining improved results in this context.

3.1.4 Workload analysis

Subsequently, we conducted an analysis to verify the capacity of the classifiers to generalise the application of the learnt knowledge. For this, we detail the results of the classifiers trained with all three workloads and tested with all workloads and commands combinations, aiming at understanding which workload generates a broader and more complete profile of the system. Figure 3.7 presents the results, which are filtered by training time (`24H`) and by window size using window `3` and `4`. Additionally, we only focus our attention on Docker and LXC and to BOSC and STIDE, owing to their better performance.

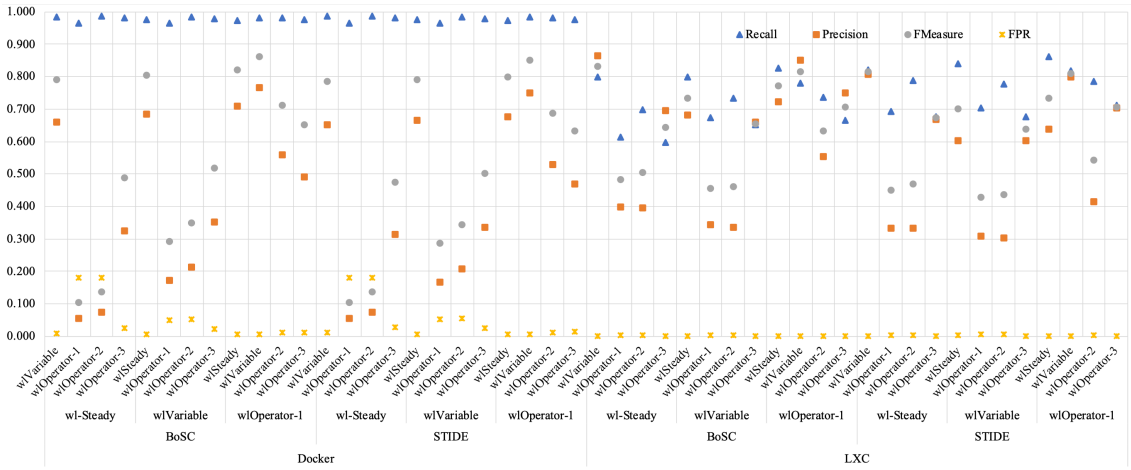


Figure 3.7: Generalisation capacity of the classifiers. – The horizontal axis has on the lower line the platforms, on the middle lines the algorithms and the training workloads and on the highest line the testing workloads.

In general, the results are similar among classifiers for the same deployment, demonstrating good generalisation capacity for the classifiers. There is, however, a noticeable increase in precision in LXC when `wlSteady` is used during training. This fact contradicts what one might intuitively expect, since `wlVariable` is composed of unstable customer behaviour and holds more connections; still, LXC produces higher amounts of data [Flora and Antunes, 2019] and the utilisation of `wlSteady` could generate cleaner profiles. In general, Docker classifiers prevail in terms of detection rate at the expense of producing a slightly higher FPR. In this case, the differences between the two workloads used during training are in the opposite direction, with classifiers trained with `wlVariable` generating alerts with slightly higher precision but significantly lower FPR. Hence, the use of `wlVariable` during training for Docker tends to ensure better overall results with low cost for the detection rates, which remain significantly higher than for LXC.

Regarding the utilisation of `wlOperator-1` workload, it causes clear gains in terms of precision for Docker and, in general, LXC as well. This workload can be described as very complete and effective for use during training, it can not only provide classifiers with the ability to generalise information with regard to commands used (`wlOperator-2` and `wlOperator-3`) but also contributes with the capacity to maintain high detection rate and precision values for `wlSteady` and

wlVariable. This workload proves its generalisation capacity into other command sets as well.

In summary, we can observe that the results are satisfactory with both workloads; however, in Docker wlVariable contributes with lower false positives and higher precision, while in LXC using wlSteady increases the precision obtained. However, the most significant variations occur when wlOperator-1 is used to train classifiers; it obtains a high detection rate and precision for every test workload. This means that the classifiers are able to generalise the knowledge and the profiles learnt are more complete and effective, demonstrating the importance of having quality in the workloads selected.

In addition to the classifiers from previous analyses, we also focus on additional classifiers trained with wlVariable and data resulting from management commands, which are benign operations that could be led by a system administrator. For training purposes, only Commands Set 1 from Figure 3.8 was used; while during the testing phase each test case was performed with the three patterns present in the referred figure. This experiment aimed at understanding two main points: i) are classifiers without prior knowledge of management commands capable of distinguishing these operations from malicious events; and ii) are classifiers trained with some commands capable of generalising their knowledge and distinguishing new commands from attacks.

# Command Set 1	# Command Set 2	# Command Set 3
df -h	ps faux	uptime
mpstat	lsof | grep FIFO	uname -a
sar -r 1 1	ss -s	ls -l /home
vmstat	tcpdump	mkdir -p /home/dir
ps faux	uptime	cd /var/log/
lsof | grep FIFO	uname -a	echo "OK" >> ok
ss -s	ls -l /home	rm -r /home/dir
tcpdump	mkdir -p /home/dir	ifconfig

Figure 3.8: wlOperator sets of command used in the experiments.

For this, we conducted the testing phase with three patterns executed before and after the attack takes place. Furthermore, we tested existing classifiers (without knowledge of any commands) and classifiers trained with Commands Set 1 and wlVariable, whose results are presented in Figure 3.9.

The overall analysis of Figure 3.9 confirms initial expectations that classifiers without prior knowledge of management commands struggle to distinguish non-malicious maintenance operations from malicious events. Although, in the case of Docker, the recall values are close to 100% every time, classifiers trained with either wlSteady or wlVariable are unable to achieve precision values above 0.400 while, in some cases, generating FPR as high as 0.180. On the other hand, regardless of the presence of commands during training, LXC classifiers do not produce precision values lower than 0.300; still, these results come at the expense of attack detection rates, since the recall values do not exceed the threshold of 0.800, which is significantly lower than in Docker.

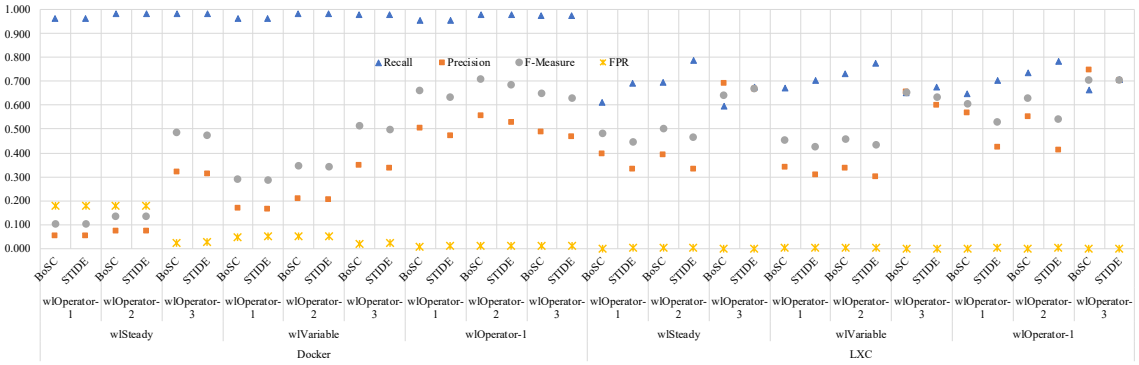


Figure 3.9: Performance of classifiers faced with management tasks and attacks.

Classifiers that were trained with datasets containing maintenance tasks demonstrate higher precision but with similar recall values, demonstrating the importance of having knowledge of how maintenance tasks are regarded so that it is possible to filter them and clearly differentiate them from attacks. Furthermore, there was a noticeable capacity to generalise known management patterns, since for `Commands Set 2` and `Commands Set 3` the results were similar, maintaining high precision and low FPR while detecting the attacks.

In short, the classifiers without prior knowledge of management commands struggle to differentiate non-malicious maintenance tasks from attacks, while classifiers with task patterns understanding are able to generalise it to other previously unknown commands. The overall detection of attacks was high, but the inclusion of management commands in training is of utmost importance to increase the precision of alarms and reduce false positives in environments where this behaviour is customary.

3.1.5 Most representative and promising configurations

In this analysis, we focus on configurations that perform better in the most relevant scenarios with high detection rates and precision. The most diverse attackloads are the most challenging in terms of detection (harder to avoid false positives while reporting correctly the attacks). This means that testing datasets based on the wlSteady and wlVariable were avoided. We performed the same procedure for Docker and LXC and sorted the results in descending order using the F-Measure value and selecting the top 10. These can be observed in Table 3.2 for Docker and in Table 3.3 for LXC.

From the analysis of the values present in Table 3.2 we can conclude that the top 10 classifiers have been trained with the workload `wlOperator-1` and used `e=500` with `d=100`; results in recall values ranging from `0.937` to `0.971` and precision between `0.858` and `0.903` while not surpassing `0.002` of FPR. The classifiers presented achieve their full potential mainly when tested with `wlOperator-2` workload, and with `w=3`. The BOSC algorithm has the top two positions and achieves F-Measure values of `0.936` and `0.931`.

These results clearly demonstrate that having richer and more complex training

workloads indeed makes a significant difference, in both the representativeness of the experiments and the quality of the results obtained.

Table 3.2: Most representative and promising results for Docker.

Algorithm	Train Workload	Train Time	Test Workload	Window Size	Epoch Size	Detection Threshold	Recall	Precision	F-measure	FPR
BoSC	wlOperator-1	24	wlOperator-2	3	500	100	0.971	0.903	0.936	0.002
BoSC	wlOperator-1	12	wlOperator-2	3	500	100	0.971	0.893	0.931	0.002
STIDE	wlOperator-1	24	wlOperator-2	3	500	100	0.971	0.892	0.930	0.002
STIDE	wlOperator-1	12	wlOperator-2	3	500	100	0.971	0.883	0.925	0.002
BoSC	wlOperator-1	6	wlOperator-2	3	500	100	0.971	0.880	0.923	0.002
STIDE	wlOperator-1	6	wlOperator-2	3	500	100	0.971	0.874	0.920	0.002
BoSC	wlOperator-1	24	wlOperator-1	3	500	100	0.937	0.896	0.916	0.001
BoSC	wlOperator-1	12	wlOperator-1	3	500	100	0.937	0.887	0.911	0.001
BoSC	wlOperator-1	24	wlOperator-2	4	500	100	0.971	0.858	0.911	0.002
STIDE	wlOperator-1	24	wlOperator-1	3	500	100	0.937	0.883	0.909	0.001

Of the ten classifiers present in Table 3.2, the majority (6) use BOSC as the algorithm, whereas the remaining 4 use STIDE, showing a prevalence of frequency-based techniques obtaining better results. In fact, the only classifier that uses `w=4` utilises BOSC, which indicates that this method can achieve better results even when using higher granularity. Still, the general preferred configurations reside in the use of `w=3`, `e=500`, and `d=100`.

Regarding LXC, the analysis of Table 3.3 shows that recall values are in the range of `0.806`–`0.895`, precision between `0.877` and `0.980`, with very low FPR. LXC has some similarity to the results from Docker; however, the differences are very interesting. For example, in LXC, all training workloads appear in the top 10 classifiers, although the majority result from the use of `wlOperator-1`. Although the best performing classifier uses `w=3`, the majority of the best results emerge from `w=4` and despite the large number using `d=100`, the best results are with `d=20`.

The reason why a larger window shows up more frequently in these results is related to the larger amounts of data required to be processed in the LXC traces. Despite LXC having on average more unique system calls, there is also a small portion of them that are issued more frequently; thus, the diversity is dispersed among the most frequent system calls, being required to have larger windows to accommodate the repetitions.

Table 3.3: Most representative and promising results for LXC.

Algorithm	Train Workload	Train Time	Test Workload	Window Size	Epoch Size	Detection Threshold	Recall	Precision	F-measure	FPR
BoSC	wlOperator-1	12	wlOperator-2	3	500	20	0.895	0.917	0.906	0.000
BoSC	wlOperator-1	24	wlOperator-2	4	500	50	0.895	0.895	0.895	0.000
STIDE	wlOperator-1	24	wlOperator-2	3	500	20	0.895	0.895	0.895	0.000
BoSC	wlOperator-1	24	wlOperator-2	5	500	100	0.895	0.878	0.887	0.000
STIDE	wlOperator-1	12	wlOperator-2	4	500	100	0.895	0.877	0.886	0.000
BoSC	wlOperator-1	12	wlOperator-3	4	500	100	0.806	0.980	0.885	0.000
BoSC	wlOperator-1	6	wlOperator-3	4	500	100	0.806	0.970	0.880	0.000
STIDE	wlOperator-1	24	wlOperator-3	4	500	100	0.806	0.966	0.879	0.000
STIDE	wlSteady	24	wlOperator-3	4	500	100	0.806	0.966	0.879	0.000
BoSC	wlVariable	12	wlOperator-3	4	500	100	0.806	0.955	0.875	0.000

Other interesting insights from these results are that the best performance is achieved with `12H` of training, which contradicts the general observation in Section 3.1.3

that `24H` resulted in more effective classifiers; and in fact, the same configuration with `24H` of training does not show in the top 10 and resulted in `recall=0.773` and `precision=0.954`. This demonstrates that despite achieving better precision, the loss in detection rate is far greater.

In summary, classifiers achieve better results when trained with `wlOperator-1`, demonstrating the impacts of representative and rich workloads on their performance. Other important observation is that smaller epochs result better, meaning that lower granularity is required for achieving better distinction between malicious and normal events, in fact, the majority requires a high detection rate, assuring that in lower granularity it is required a significant percentage of anomalous events to produce satisfactory outcomes. For instance, when `e=500` and `d=100`, it means that at least 20% of the events within an epoch are considered anomalous. Additionally, the utilisation of smaller windows seems to work correctly with more diverse traces, while larger windows perform better with less diversity or higher amounts of data. In connection with this, the use of smaller epochs with the need of high percentage of anomalous events within it is appropriate in high diversity datasets, whereas lower percentages, such as 4% with `e=500 and d=20`, work better with lower diversity or larger amounts of data.

3.2 Dealing with the elasticity of the microservices

The application of classical behaviour-based intrusion detection requires a profile for each service replica instantiated and results in a localised view, with focus on the instance, completely disregarding the remaining service instances. As attacker traffic can be routed across multiple replicas, a global view is required and would potentially be more effective in deterring attacks. The continuous adaptations of service replica numbers, according to the demand, can lead to a large number of instances, thus increasing resources necessary to maintain the security protection mechanism. Therefore, intrusion detection approaches must be lightweight and process information from the whole replica set, while also coping with scalability required to handle dynamic environments and operate with diverse operational profiles of a continuously changing architecture.

In this work, we **assumed that each service is deployed in its own container**. It is also **assumed that each service may be part of a set of service replicas**, which may scale independently. The proposed approaches process independent data streams, which originate in each of the monitored service replicas. It requires time information, and the streams consisting of sequential and chronologically ordered data from each unit monitored.

The development of data processing approaches together with lightweight algorithms enables intrusion detection application to microservices. Developing profiles of microservices that could be generalised and reused across modifications in active instances, through elastic behaviours, would extend the security levels. These profiles must detect different types of attacks (external and internal) and continuously operate despite the dynamicity of these scalable and elastic environments. The adaptations performed regarding the number of active service

replicas cannot impair the monitoring mechanisms or the algorithms processing the information, which in services with great demand and complexity, such as microservice-based systems, can become a serious threat to the effectiveness of the mechanism.

The evaluation of the proposed approaches intends to demonstrate their operation in multiple scenarios, which are detailed and analysed here. We discuss the main perspectives on effectiveness in detecting intrusions. Initially, we address the different configurations used and their tuning to maximise detection rates and precision. Then we address multiple viewpoints related to the exploits, periods, and scalability scenarios.

The main objective is to validate our approach ability to help intrusion detection approaches deal with multiple operational profiles of microservices that use service replication. We train several behaviour profiles using well-established algorithms with and without the data processing approaches proposed for multiple system configurations.

My contribution was done with regard to the analysis of results, but also helped with the setup, the discussion, and implementation of the data processing techniques.

3.2.1 Data Processing Approaches considering Scalable Microservices

As previously noted, current methodologies do not address security threats looming over a set of multiple replicas of a service. Instead, they addressed the issue by focussing on each unit, thus requiring a higher number of active profiles. However, the direct application of state of the art intrusion detection algorithms to a set or service replicas is not sufficient to ensure the generalisation of the acquired profiles. There are limitations to the behaviour learnt from a single service instance that impairs generalisation. To overcome this challenge, we proposed three approaches to enable detection of intrusions in scalable and elastic microservice-based systems.

Fig. 3.10 graphically represents the inner operation of each approach. The figure uses a notation where e_t^r is an event of replica r with timestamp t and $e_t^r - e_{t+k}^r$ refers to events of replica r over an interval of k units of time.

The proposed data processing approaches provide algorithms with the ability to generalise the behaviour profile obtained and operate effectively in different scenarios of scalable microservices. In this way, algorithms can operate in multiple scaling deployments and maximise the protection time of replicated services. Detectors do not need to adapt to the current configuration of the system, accompanying scaling operations, performed with demand changes, and detecting security intrusions regardless of scaling down or up.

Following, we detail the proposed approaches to process the system call information that is collected from the target system. These approaches can work with

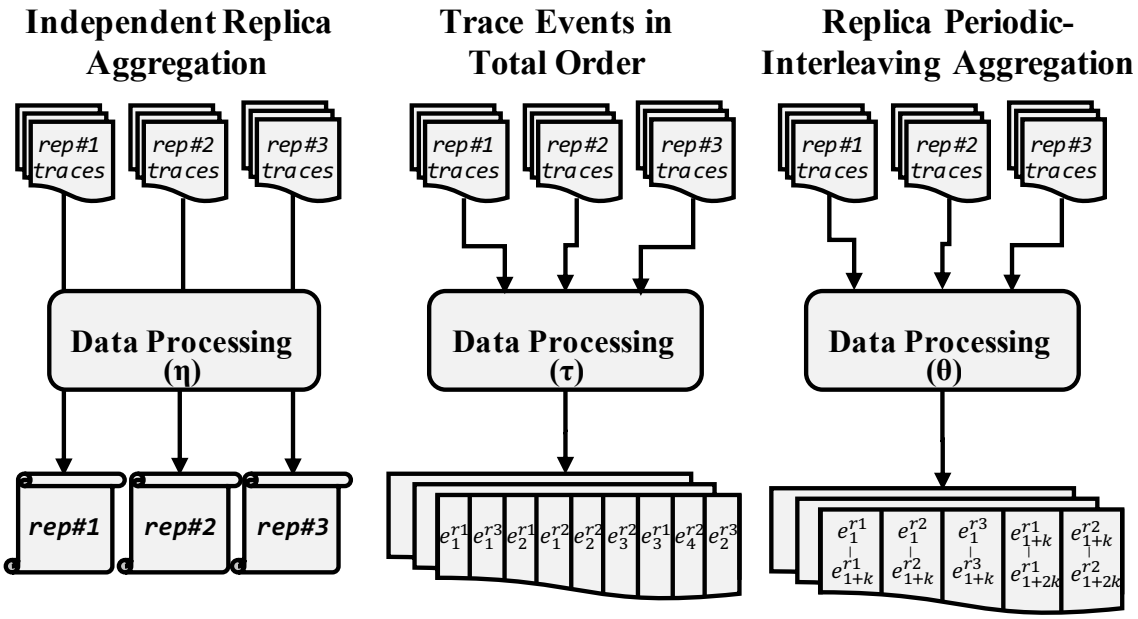


Figure 3.10: Overview of the proposed approaches for data processing for microservices intrusion detection.

any algorithm for the detection of malicious behaviour after the construction of benign behaviour profiles.

Independent Replica Aggregation (η)

The operation of this approach mainly affects the resulting profile during the training phase. The algorithms with data processing store the information from all the service active replicas, expanding the knowledge gathered, which results in more complete behaviour profiles. Data from different replicas are combined without being mixed, increasing the behaviour patterns used to classify events during the detection phase.

In this way, all the patterns of behaviour collected from the replicas operating are stored to enrich the profile used to detect anomalies during the detection phase.

In the remainder of the document, the algorithms with this approach will have as a prefix the letter η (*e.g.*, η-stide).

Trace Events in Total Order (τ)

This approach modifies the way patterns are built from the data traces of the various replicas. That is, the trace fed to the algorithms results from the chronological order of all the entries across each active service replica. This approach requires the data from each microservice to be collected with the corresponding timestamp that is used to establish the chronological order. It also assumes that the replicas have similar timestamps.

Algorithms with data processing combine events from all replicas into a chrono-

logically ordered trace that is used to construct the profile. As depicted in Fig. 3.10, this approach creates a connection between all information produced in a set of service replicas. It corresponds to the incoming order of service requests, as they are distributed across all replicas that are operating.

Algorithms that use this approach create profiles with a clearer idea of the interactions between service replicas. In this way, it makes possible the identification of more complex attacks as they try to compromise the service, and the load balancer directs different requests to different replicas.

Over the remainder of the document, algorithms with this approach will have as prefix the letter τ (*e.g.*, τ-stide).

Replica Periodic-Interleaving (θ)

This approach creates an event trace that results from sequential extraction intervals, with t units of time, of data from alternating replicas. That is, from each replica the events within the interval are appended to the trace.

This approach creates fewer breaks in the sequence of system call data. As the system calls sequences are more relevant than an isolated call, this approach intends to reduce possible sequence breaks caused by the aggregation performed. It allows to retain the proximity of certain system calls that take place in the same replica, while still leveraging the combination of events across replicas.

Over the remainder of the document, algorithms with this approach will have as a prefix the letter θ (*e.g.*, θ-STIDE).

Integration with Intrusion Detection

The approaches integrate with a typical anomaly-based intrusion detection methodology, as presented in Fig. 3.11, which works in two main phases: **training phase**, where the operational profiles are learnt, and **detection phase**, where events are analysed for intrusions. During the training phase, the classifiers use benign information from the system to create its benign behaviour profiles. These profiles are used during the detection phase to identify deviations from the learnt normal behaviour and raise alarms.

Data processing approaches operate on the collected data and feed it to intrusion detection algorithms. Consequently, regardless of the number of active replicas, the profiles can be used continuously to detect security intrusions with effectiveness and ensure uninterrupted security of the service replicas set. This ability removes the need for multiple training conditions to cover all the deployment scenarios faced during the detection phase; thus, promoting the reusability of profiles and generalisation of the acquired knowledge. Variations in load and demand, and consequently autoscaling operations performed, do not affect the capacity of profiles to detect malicious events.

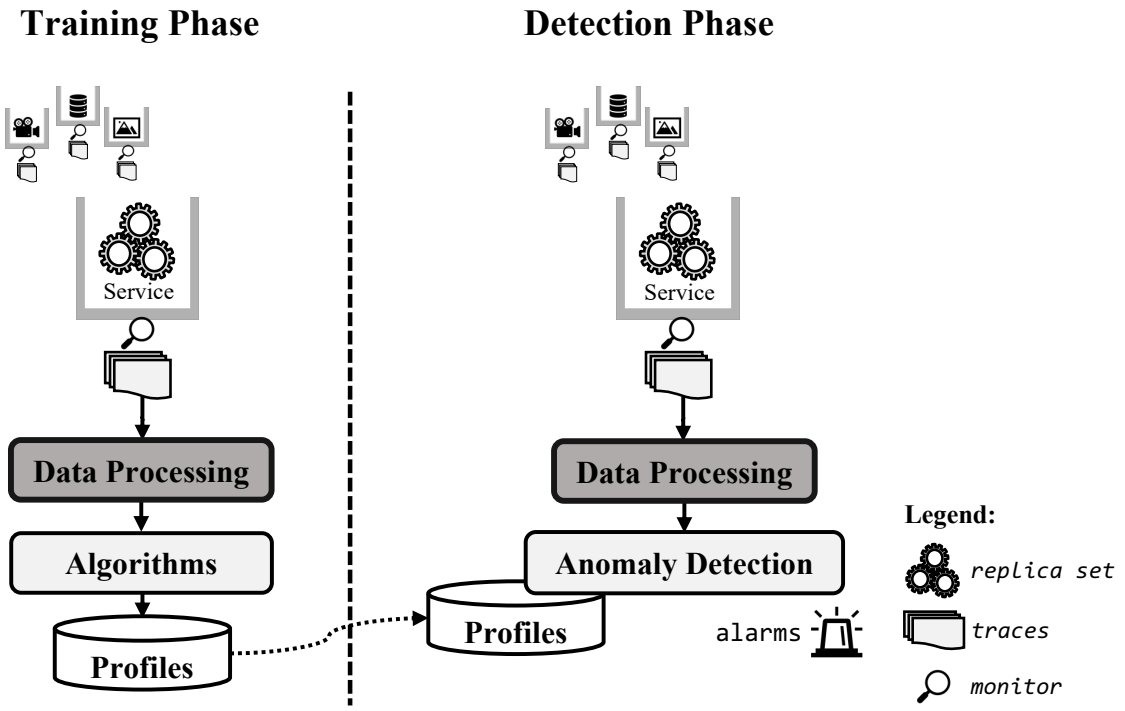


Figure 3.11: Integration of our approaches with an anomaly-based intrusion detection methodology for microservice-based systems.

Training and Detection Phases

The training phase is crucial for capturing effective profiles. The efficacy of this phase impacts the ability of classifiers to identify deviations from the normal behaviour of services. Thus, it intends to acquire the most complete snapshot of the operational profile of the target service. This phase can be split into two main goals: i) sending the information processed by the proposed approaches to the algorithm; and ii) the definition of the normal service behaviour.

For the first objective, the approaches process the information collected from the different replicas and produce the data set received by the algorithms. Regarding the second objective, the algorithm processes the data received from the approaches to organise patterns (i.e., system calls sequences) into storable tokens, which constitute the operational profile of the replicas set. As with typical behaviour-based intrusion detection techniques, the data used in this phase must be exempt from malicious events, as this would poison the profile and potentially leave space for attackers to mislead detectors.

The detection phase operates online in parallel with the services and raises alarms for time intervals that contain multiple events classified by the detectors. When attackers compromise a service and obtain unauthorised access to an application, they can cause significant damage in a short period of time. Therefore, detection must remain lightweight and be able to raise alarms efficiently for security operators, so that they have a rapid response.

The operation of data processing over information from multiple service replicas

data streams allows fewer instances of classifiers. Additionally, since the operational profiles learnt can be used in different scenarios, the security monitoring of the set of replicas is not lost or temporarily impaired when scaling occurs. Our approach reduces adaptations to active detectors and generalises the process of detecting intrusions in scalable and elastic microservices.

3.2.2 Reporting Mechanism

In this work, the anomaly-based intrusion detection methodology analyses the alerts raised by the classifiers during the detection phase through the combination of two perspectives to provide useful and meaningful alarms. An epoch-based analysis that divides results from the classifiers into sequential groups of configurable length. When the number of alerts within an epoch is not less than a detection threshold value, the epoch is classified as anomalous. This technique intends to reduce false positives while ensuring that significant deviations are still reported to system administrators.

After the epoch analysis, an interval-based normalisation aims to make resource allocation more fruitful and propagate clear indications of malicious events. With this technique, administrators are notified that a malicious event has occurred in a given interval after an alarm is issued, as long as there is an anomalous epoch. In this way, the proximity factor of attack-related events is leveraged and notifications do not overload the response mechanisms.

Reducing false positives allows better and more efficient use of resources when trying to apply countermeasures against a security intrusion. It also retains the temporal notion, crucial to having a better understanding of the damage caused by the attack in the infrastructure and the depth of compromise.

3.2.3 Experimental Methodology

Based on previous work, the experimental methodology evolved to support the use of a new representative microservice-based testbed, TeaStore [von Kistowski et al., 2018], and the new techniques that were developed. TeaStore, a web-based e-commerce system, was then subjected to representative and diverse workloads, with one of its services being modified to contain known real-world vulnerabilities, activated through an attack injection procedure; thus, increasing the validity and representativeness of the experiments.

Fig. 3.12 provides an overview of the evolved experimental methodology, depicting the multiple states of the system, and how the training and testing traces interact with the algorithms and resulting classifiers.

Dataset Preparation

The application is exercised with benign-only workloads for the collection of training traces. To evaluate the classifiers obtained, during the experiment, traces

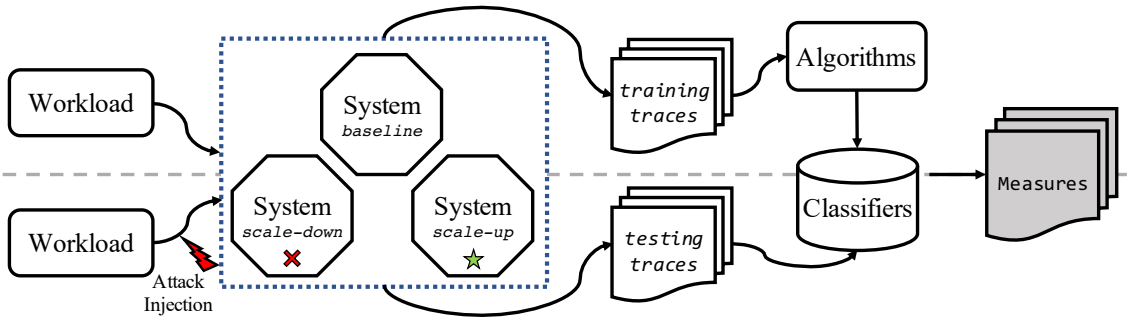


Figure 3.12: Methodology for the experimental validation.

were collected when the application was subjected to a mixed load, with benign workloads and injection of an attack, from the set referred to in Table 3.4. This process was repeated for five service deployments, with different numbers of active replicas of the system attacked. With our approach, the classifiers must provide reliable results regardless of the number of active replicas during the training or detection phases. Our methodology covers both scenarios. The results were produced by feeding the testing traces to the classifiers.

Evaluation Workloads and Attacks. To produce the experimental data (traces of system calls), four different workloads were used to exercise the system. These resulted from the combination of two intensity profiles and two user behaviour profiles. A stable intensity profile, with constant number of 1000 requests per second and other varying between 200-1800. Two behaviour profiles, one consisting of users browsing the store, adding and removing items from the cart, and the other, which also accomplishes purchases. Thus, it modifies the state of the database information and results in a richer workload from an operational point of view. To attack the system at multiple levels, we selected known vulnerabilities with publicly available exploits. The selected vulnerabilities are described in more detail in Table 3.4, where for each vulnerability the corresponding CVE ID is presented along with the component it affects, its class, type of exploitation and consequences, and the CVSS score value (information extracted from *(cvedetails.com)*). They were selected owing to their representativeness, being real security risks that were already patched in newer versions of the affected services.

Data Collection Method. The data collection in our experiments was conducted using sysdig [Sysdig, Inc, 2019], a tool that allows the collection of various types of information from a target system. Sysdig was installed on the main machine used in the experiments and collected system calls invoked by the container of the microservices.

This data collection method ensures standardization of the process, as it eliminates subjective consideration of the operator in the selection of processes to collect system call data. The tool receives the id of the container and performs the data collection automatically, thus increasing the reproducibility of the process.

Datasets. For the experiments considered, a total of 436 data traces were produced: of which 16 were used for training and 420 for the detection phase. Each training trace is composed of 48H of system call data from the system when sub-

Table 3.4: List of vulnerabilities used and respective CVE information.

Comp.	CVE ID	Vulnerability Class	Vulnerability Type(s)	Score
Tomcat	2016-1240	Access-Control	PE	7.2
	2016-6816	High-level	XSS, Gain Info.	6.8
	2017-12617	Access-Control	Exec Code	6.8
Docker	2019-5736	Access-Control	Exec Code	9.3
Kernel	2014-3153	Access-Control	PE	7.2
	2014-4014	Bypass	Bypass a restriction	6.2
	2014-4699	Access-Control	DoS, PE	6.9
	2016-0728	Access-Control	DoS, Overflow, PE	7.2
	2016-5195	Access-Control	PE	7.2
	2016-9793	Corruption	DoS, Overflow, MC	7.2
	2017-1000112	Corruption	MC	6.9
	2017-16939	Access-Control	DoS, PE	7.2
	2017-16995	Corruption	DoS, Overflow, MC	7.2
	2017-7308	Access-Control	DoS, Overflow, PE	7.2

Legend - DoS: Denial of Service; PE: Privilege Escalation; MC: Memory Corruption; XSS: Cross Site Scripting

jected to benign workloads, and each detection trace results from a `10min` period of mixed data from the benign workload and the attack injected in the middle of the experiment.

The 420 detection traces resulted from 3 repetitions of each combination of 5 configurations of the target system with all 14 attacks and 2 request profiles with the variable intensity profile ($3 \times 5 \times 14 \times 2 = 420$). The resulting dataset amounts to a considerable size, with all training and detection traces using around `436GB` when zipped, with `402GB` for training traces and `34GB` for detection traces. Typically, the compression rate is about `90%` so the unzipped size of the dataset is around `4360GB` $\approx$ `4.36TB`. This portrays a more representative dataset than already available data in the state of the art.

Sensitivity to Attacks and Alert Frequency

The proposed approach targets systems supported by software containers, which are lightweight and composed only by the required components for the service to work. In this way, these are sensible systems that can have its operation change significantly with some type of attacks that target them. To assure that the occurrence of intrusions is picked up by detectors the algorithms need to capture a stable and complete behaviour profile.

As some attacks can cause deviations in the operation of services, as the system moves to an unseen or corrupted state, to perform a correct evaluation of the classifiers, we divided the slots into three phases: pre-attack, attack, and post-attack phase, as depicted in Fig. 3.13.

These phases are defined around the injection of the attack and have the objective of establishing three perspectives of analysis. During the pre-attack phase there is no justification to the occurrence of alarms, expect incomplete profiles of

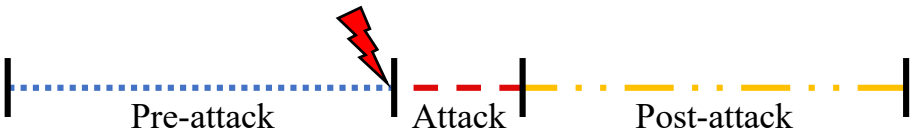


Figure 3.13: Experimental slots phases.

behaviour; during the attack phase, the duration of the attack injected, failing to raise alarms means false negatives; while the alarms raised during the post-attack phase can be justified by effects resulting from the attack conducted, which means that an alarm raised can be caused by lasting effects of the attack and incomplete profiles as well.

In connection with this point, we also analyse the frequency of alerts raised by time interval. It mainly contributes to clarify the quality of the profiles captured, and whether they portray the correct behaviour of the service replicas monitored. Alerts should be more frequent during the attack phase, and some are understandable during a period after the end of the attack.

Scalability and Data Processing Approaches

Replication of services to scale and adapt systems to changes in user demand is common in microservices. To analyse the ability of data processing algorithms to cope with changes in active service replicas, we create multiple scenarios for training and detection. For training, we consider 1, 2, 3, and 5 replicas used during deployment. For detection, we consider 1, 2, 3, 5, and 7 replicas. These deployments are crossed, resulting in 20 evaluation scenarios, presented in Table 3.5.

Table 3.5: Deployment scenarios used for training and detection phases.

#Training Replicas	**#Replicas in Detection**				
	1	2	3	5	7
1	✓	✓	✓	✓	✓
2	✓	✓	✓	✓	✓
3	✓	✓	✓	✓	✓
5	✓	✓	✓	✓	✓

In this way, we can compare the different processing approaches, between each other and with the simple state of the art algorithms. This comparison permits to observe whether the algorithms with processing approaches increase detection rates while having low false positives. This analysis also compares stable scenarios, with the same number of replicas in both cases, and scalable scenarios where different numbers of replicas are active in training and detection time.

Configuration Tuning

The intrusion detection methodology used has multiple configuration parameters, which have an impact on the results obtained. The algorithms are based on a sliding window with configurable size, for the evaluation, we selected length 3 and 4, based on other works [Flora et al., 2020]. Further, the methodology considers more configuration, namely the analysis of classification results. The epochs used along with a detection threshold to reduce false positives are also configurable values. For this work, we considered 500, 1000, and 5000 for the sizes of the epochs, and 5, 10, 20, 50, and 100 for the detection threshold, as presented in Table 3.6. Finally, the time interval used for the reporting mechanism can also be configured, in this work, we used an interval of 5 seconds.

Table 3.6: Combination of epoch and detection threshold values used.

Epoch	**Detection Threshold**				
Size	5	10	20	50	100
500	✓	✓	✓	✓	✓
1000	✓	✓	✓	✓	✓
5000	✓	✓	✓	✓	✓

The execution of the evaluation with the referred configurations allows a broader notion of the impacts of such parameters in the effectiveness of the classifiers produced and, on the results, obtained from their analysis. Also, by allowing several configurations, the methodology used has the potential to be applied to different scenarios and context, operating in less and more critical microservice-based systems.

3.2.4 Results Overview

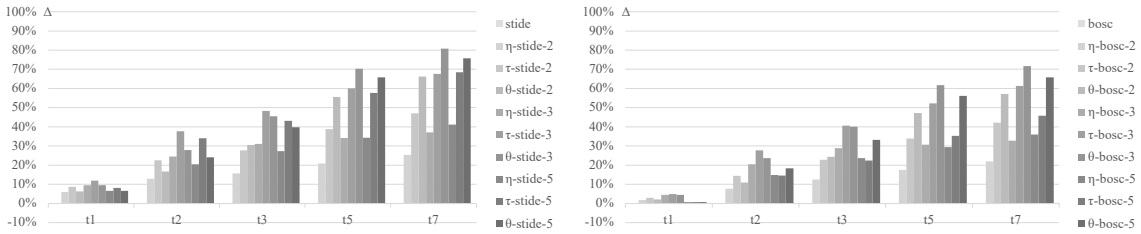


Figure 3.14: Relative results for algorithms with and without proposed data processing approaches, based on pre-attack phase F-Measure values. On the left, STIDE as baseline compared to versions of STIDE with data processing. On the right, BOSC as baseline compared to versions of BOSC with data processing.

The impact of the approaches proposed in the results obtained are presented in Fig. 3.14. It depicts the variation in F-Measure when compared to the simple use of the algorithms, considering the averages of all configurations. The value obtained from the use of STIDE and BOSC represents the baseline (i.e., 0%) while for the remaining cases the percentage of variation compared to STIDE or BOSC. Although, for one replica active during detection, algorithms with and without

processing approaches produce similar results, the lowest performance increase still represents an increase of 6% for η-stide-2. However, when the number of active replicas increases, the algorithms with data processing clearly outperform simple STIDE and BOSC. For three replicas during detection time, increases reach as high as 48% for τ-stide-3 and 40% for τ-bosc-3. For seven active replicas, the gains reach 80% for θ-stide-3 and 62% for θ-bosc-3. The general observation seems to show a tendency for replica periodic interleaving (θ) to produce better results with a higher number of replicas.

The variation trend presented demonstrates clearly depicts a consistent and continuous increase for higher number of active replicas, which indicates that algorithms with data processing are able, without compromising the base case, to detect intrusions more effectively than simple algorithms.

Further, we dive into the details behind the methodology used and how the results are achieved, focussing on configuration tuning, attack alert frequency, and dealing with scalability of services.

3.2.5 Configuration Tuning

The tuning of the configurations used in the algorithms and the alert mechanism is crucial to obtain the best results for the case under monitoring. Algorithms can have their window size configured, which is the length of the patterns that make up the normal behaviour profile. For the reporting mechanism, the user can configure the epoch size, that is, the number of patterns within an epoch, and the detection threshold, which corresponds to the required number of anomalous windows to classify an epoch as anomalous. Another parameter in the methodology is the time interval used to raise alarms to the operator. However, in our analysis, we fixed this value at 5s, based on the average duration of the attacks, and did not modify it during the experimentation. Still, changing the interval span can have an impact on the results produced: the shorter span increases the number of observations that can be passed to the administrator, while the larger span reduces them. It also impacts the delay for notification of attacks; shorter spans mean faster notification, while larger spans increase the delay.

To evaluate the configurations that produce better results, we selected the top 10 results according to F-Measure, and computed the frequency of each configuration. Thus, this shows the configurations that produce better results more frequently. The selection of the top 10 entries resulted in a sample size of 336,000 data points. Results for the combination of the three parameters are presented in Fig. 3.15.

The majority of entries result from algorithms using a window of size 3, with 82% of the most effective configurations resulting from this. Of the three configurations, `window_size` has the greatest impact on the results. Analysing both parameters in an isolated manner: for the epoch size, the value 500 is present in 43% of the cases, while 1000 represents 39% and 5000 represents 19% of the total sample size; for the detection threshold, the percentages range from 14% to 29% of the total sample size, with 50 being the most frequent.

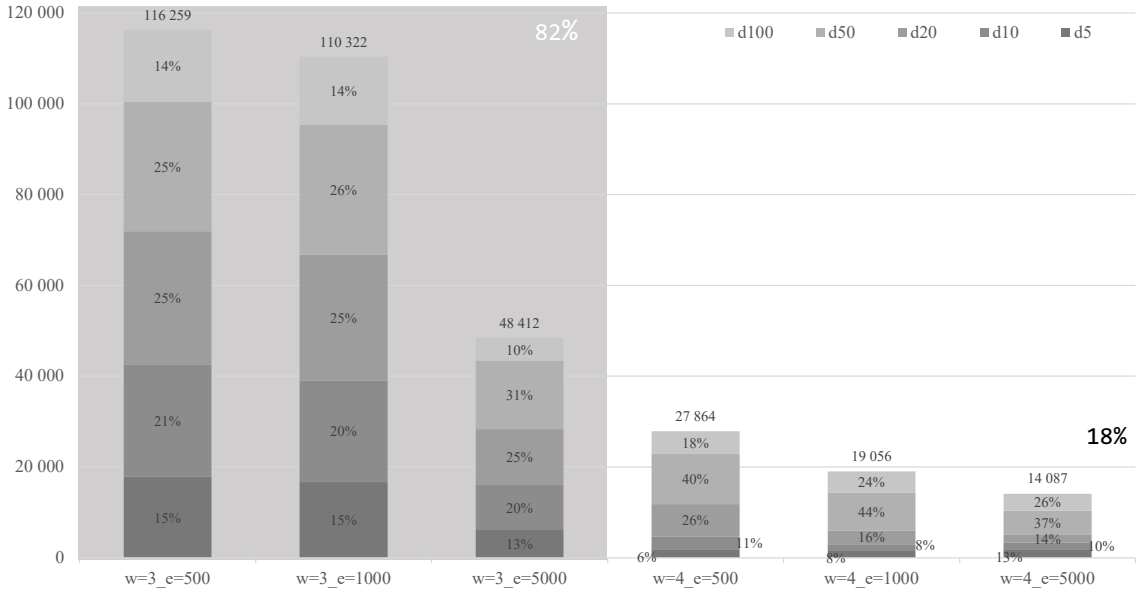


Figure 3.15: Distribution of the top 10 results according to F-Measure for parameter configuration analysis. w: window size; e: epoch size; d: detection threshold

The analysis allows us to understand that the most frequent combination of all parameters resides with a window size of 3, an epoch size of 500, and a detection threshold of 20. This configuration represents 27% of the entries resulting from the combination of this window and epoch size, as shown in Figs. 3.15, and 9% of the total sample size. In fact, the combination of the size of the presented window and the detection threshold values is the most frequent, representing 35% of the total entries analyzed. Thus, the capacity to produce better results is strengthened.

In a particular observation (cf Fig. 3.16), detecting the exploitation of the vulnerability CVE-2017-1000112 raises a number of false positive alerts in the initial steps of the detection phase, when using θ-bosc-2 with a window of size 3 and an epoch size of 500 and detection threshold of 10. However, when configurations are carefully tuned, increasing the detection threshold to 50, false positives are removed, and the attack is still detected successfully.

3.2.6 Alert Frequency

To understand whether the algorithms were capable of effectively distinguishing between normal and malicious behaviour, we analysed the frequency of alerts per time interval. In this work, an alert is when an epoch is considered malicious (cf. Section 3.2.2). If there is at least one alert in an interval of time, we consider that interval anomalous and raise an alarm.

In our experiments, we observed two relevant aspects: the classifiers with highest performance rates clearly have high-quality profiles of the replicas set; and there are attacks in our experiments that cause significant modifications in the system when used.

Fig. 3.17 shows the percentage of alarms raised in each phase (pre-attack, attack,

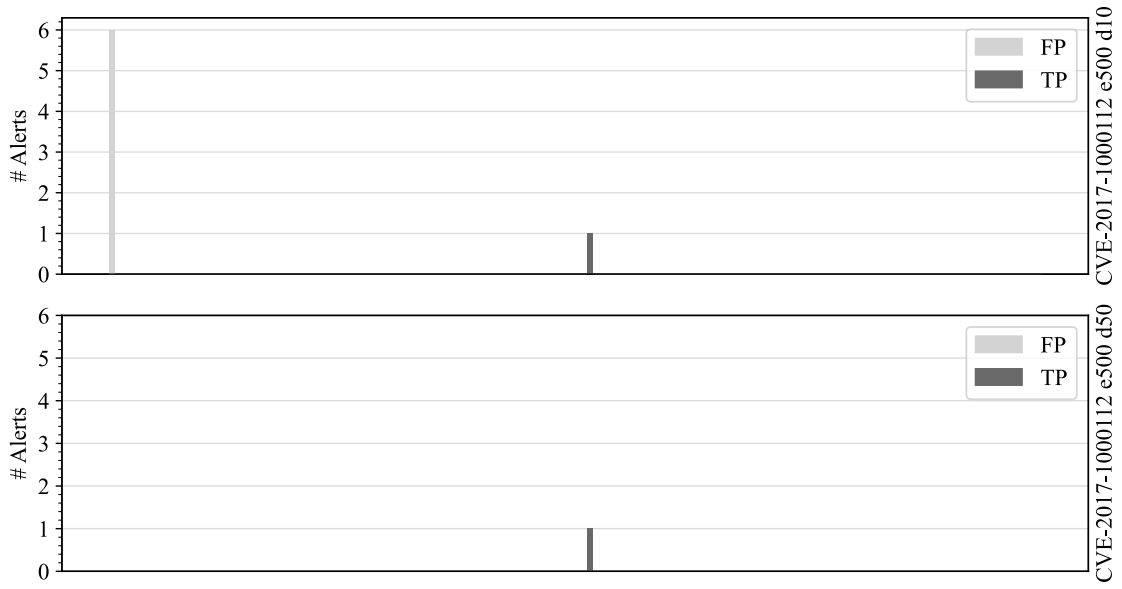


Figure 3.16: Comparison of alert frequency with modification in the detection threshold.

post-attack) by the classifiers used, which were trained with three replicas and are detecting intrusions in a set of three replicas.

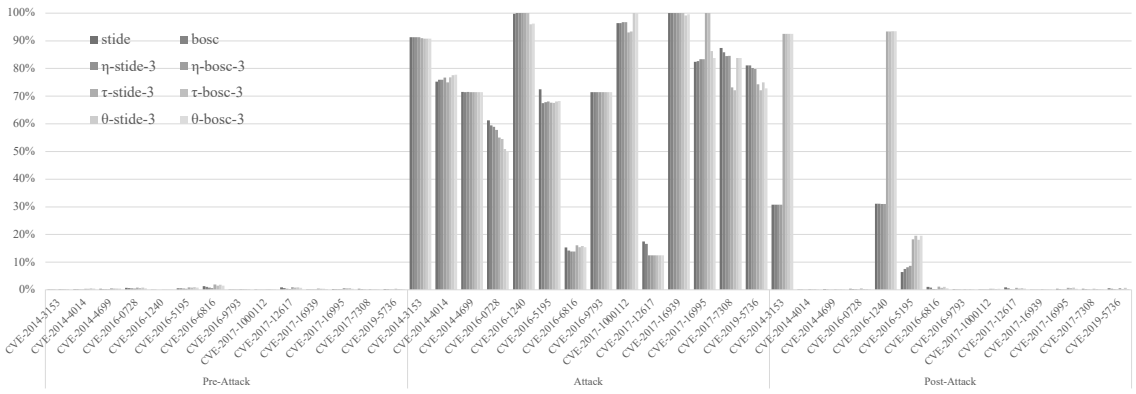


Figure 3.17: Percentage of alarms raised per CVE over all phases of the detection phase in a deployment with three replicas.

The figure confirms a good capture of the behaviour profile of the replica set. During the pre-attack phase, the highest percentage of reports amounts to `1.9%` and an average of `0.4%`. During the attack, the algorithms are generally able to detect all attacks, raising alarms in at least `12.5%` of the cases, with an average of `71.8%`. Finally, after the attack has ended, there are exploits that cause the system to become unstable, such as for `CVE-2014-3153` and `CVE-2016-1240`. These attacks cause lasting effects on the system and increase the general percentage of alarms in the post-attack phase, where the alarms amount to a maximum percentage of `93.5%`, but on average they are at most `10.1%`. There are also two attacks that target vulnerabilities `CVE-2016-6816` and `CVE-2017-12617`, which classifiers find difficult to detect. These vulnerabilities are both in the Tomcat webserver that supports the infrastructure. `CVE-2016-6816` is a high-level vulnerability, targeted

with a XSS attack and is therefore hard to reflect in the traces used. Regarding the vulnerability `CVE-2017-12617`, it corresponds to an attack of code execution that can be detected in some way with some specific configuration.

The observation of a specific case shows this clearly. Fig. 3.18 presents two different vulnerabilities, `CVE-2017-1000112` and `CVE-2016-1240`, with the frequency of alerts per time interval over the detection period. The examples presented are from the results of θ-bosc-2, window size 3, algorithm analysing a replica set of 3 units. The results were selected according to the best values for F-Measure in the pre-attack phase with the intention of demonstrating the two reported aspects.

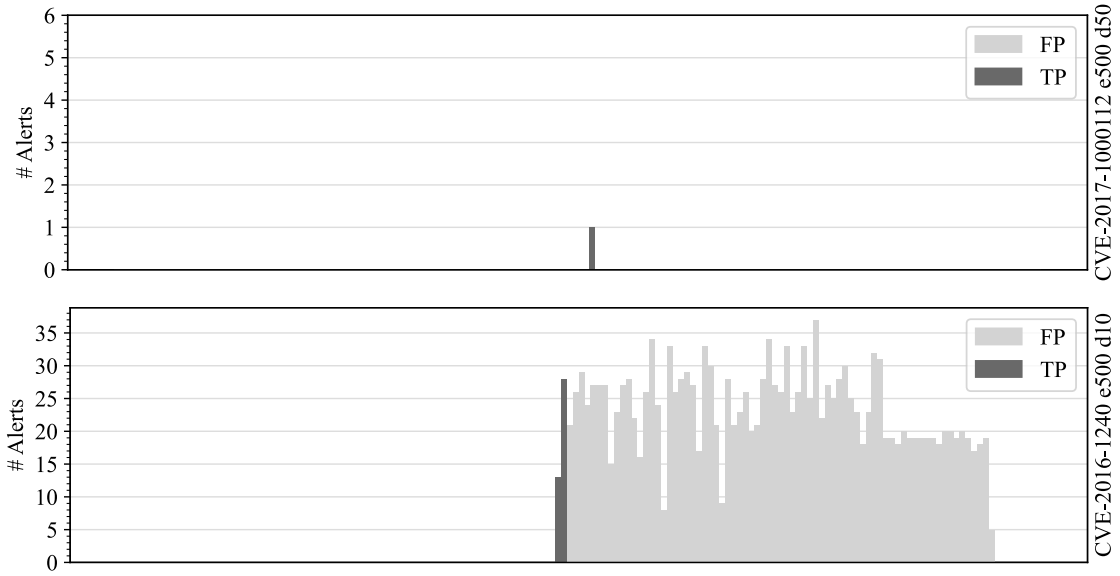


Figure 3.18: Comparison of alert frequency in attacks with and without lasting effects.

As in the initial phase of the experiment, the system is not affected by any kind of anomalous interaction, profiles are, in both cases, able to analyse the events and report no perturbation, while detecting both attacks. However, in the case of `CVE-2016-1240`, the attack creates a significant and lasting modification in the systems operation behaviour. For this reason, the classifiers continuously report anomalous activity throughout the remainder of the experimental slot.

This finding justifies the utilisation of pre-attack values to evaluate the classifiers, as the occurrence of false positives in the post-attack phase can be caused by lasting effects of the attacks. Also, the existence of lasting effects in some, but not all attacks removes the possibility of having issues in the process of attack injection conducted in this experimental evaluation.

3.2.7 Vulnerability Class Reporting

During our evaluation, various vulnerabilities and exploits are considered. This diversity allows us to make different observations and analyse the detection capacity in terms of weaknesses class. Table 3.4 presents details about the vul-

nerabilities used in the practical experiments. Four classes of vulnerabilities are presented, covering access control, bypass, corruption, and high-level real-world problems that were identified and corrected in the past.

Considering a stable deployment with three active replicas during detection and training of the algorithms with data processing, we analyse the effectiveness of the classifiers using F-Measure. For each class of vulnerabilities, Fig. 3.19 presents how the classifiers perform in their detection. In general, all classifiers perform very well across three of the four vulnerability classes. With the exception of high-level vulnerabilities, which perform between `0.075` and `0.130`, the remaining vulnerabilities are detected with high effectiveness (`>=0.632`). The better results are obtained in the detection of Corruption vulnerabilities where F-Measure peaks at `0.822`, averaging at `0.792` among all algorithms. The performance of simple algorithms is worse than algorithms with data processing, with STIDE and BOSC performing in the range of `0.632-0.764`, whereas the best performance with τ-bosc-3 in the range `0.738-0.822`, if high-level vulnerabilities are excluded.

The high-level vulnerability is a XSS attack, so, it becomes difficult to detect this exploit at the system call level; hence, the algorithms struggle to identify the malicious behaviour even with data processing approaches.

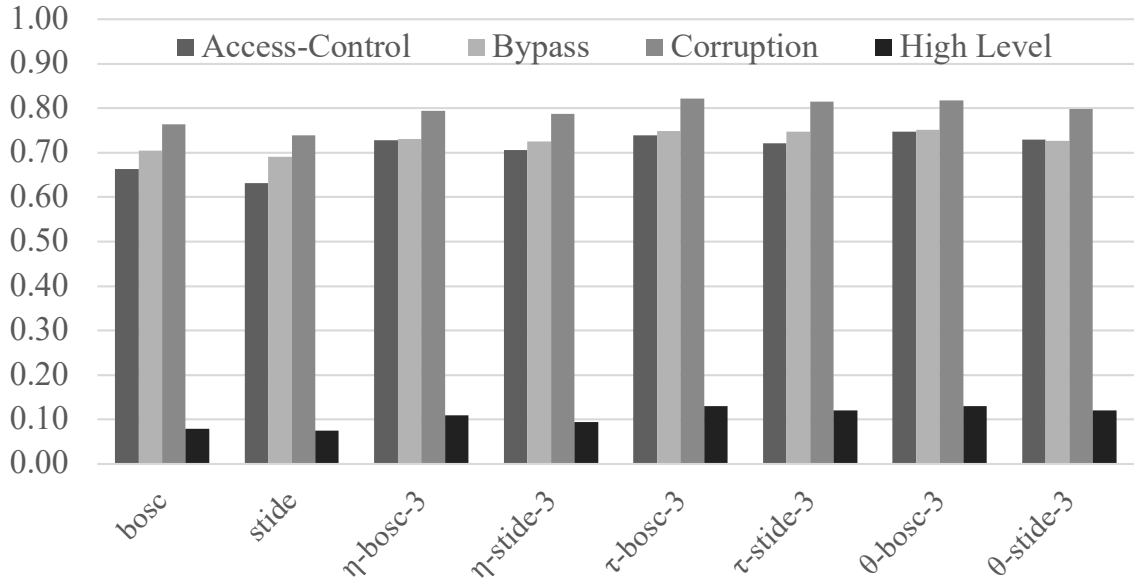


Figure 3.19: Performance of classifiers in the detection of diverse class of vulnerabilities attacked.

3.2.8 Detection in stable scenarios

The stable scenarios are considered direct applications of our approach. These scenarios refer to the application of the classifiers to the same configuration deployment at detection time as the one in which they were trained. That is, these are the scenarios where the number of replicas of a service is the same in the training and detection phases. In theory, these should be easier for the classifiers to deal with as they are a direct application of the learnt profile in the same

configuration.

In our experimental evaluation, three stable scenarios were considered. For one, three, and five active replicas of a service, profiles were trained and applied in the detection phase.

Fig. 3.20 presents the results of the classifiers used in this evaluation context. Classifiers can obtain very good results with two and three replicas at training and detection time (>=0.683). Generally, data processing algorithms achieve better performance than STIDE or BOSC. The case of five replicas also presents better results for algorithms with data processing (0.495-0.575) than for STIDE and BOSC (0.430-0.469).

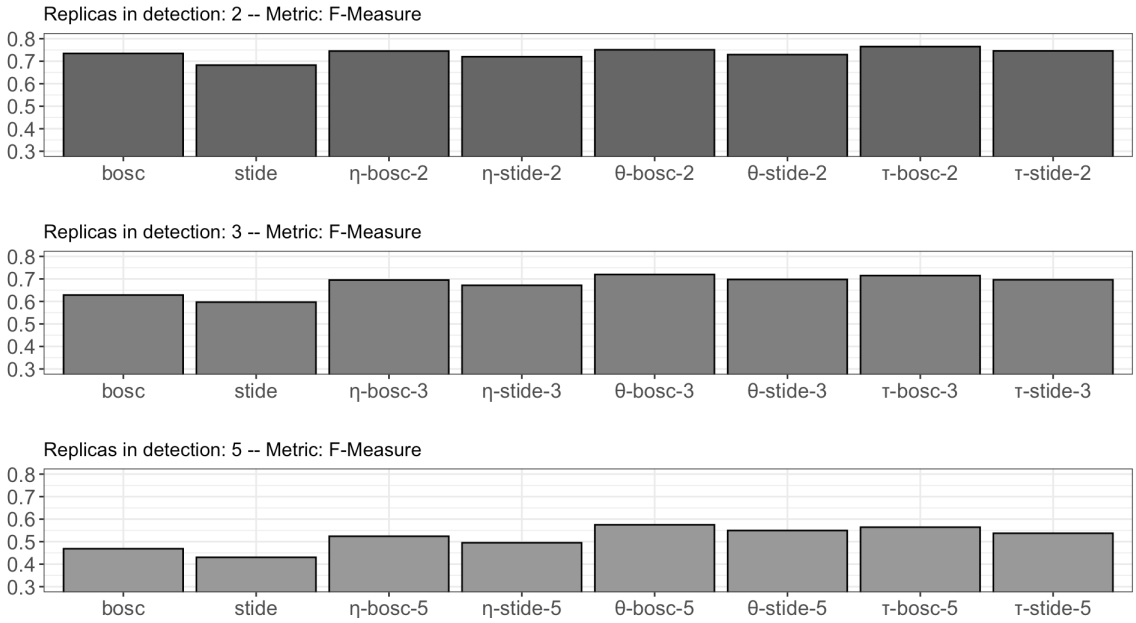


Figure 3.20: Performance of the top 10 configurations for stable scenarios.

The use of algorithms with data processing shows higher effectiveness compared to simple algorithms (STIDE, BOSC). Although there is still room for improvement in the capacity of the approaches proposed to increase detection abilities, they are already higher than previous techniques, showing usefulness in the technique.

3.2.9 Detection in scalable scenarios

Scalable scenarios consist of a very challenging application of our approach. These mean to attain a generalisation of knowledge acquired during training so that classifiers can operate effectively with a different configuration in detection time. These scenarios present interesting challenges to the classifiers, as they learn the behaviour profile of an architecture and apply it to a different one.

In our evaluation, four scalable scenarios are considered. For three and five active replicas of a service, profiles were trained. These classifiers were applied, at detection time, in systems with one, three, five, and seven active replicas; thus,

resulting in four scenarios of analysis.

Fig. 3.21 presents the F-Measure results of the classifiers used in this evaluation perspective. Classifiers perform extremely well with the more basic scenario, one replica at detection time (>=0.899), with all algorithms with data processing outperforming STIDE and BOSC. For the remaining scenarios, there is a decrease in F-Measure values for simple algorithms (>=0.404) and for algorithms with data processing as well (>=0.479). The analysis shows that algorithms with data processing perform better in scale-down scenarios, that is, the detection time has fewer replicas than used during training (3vs1: >=0.906; 5vs3: >=0.628); while in scale-up scenarios, there is a slight decrease in effectiveness (3vs5: >=0.520; 5vs7: >=0.479).

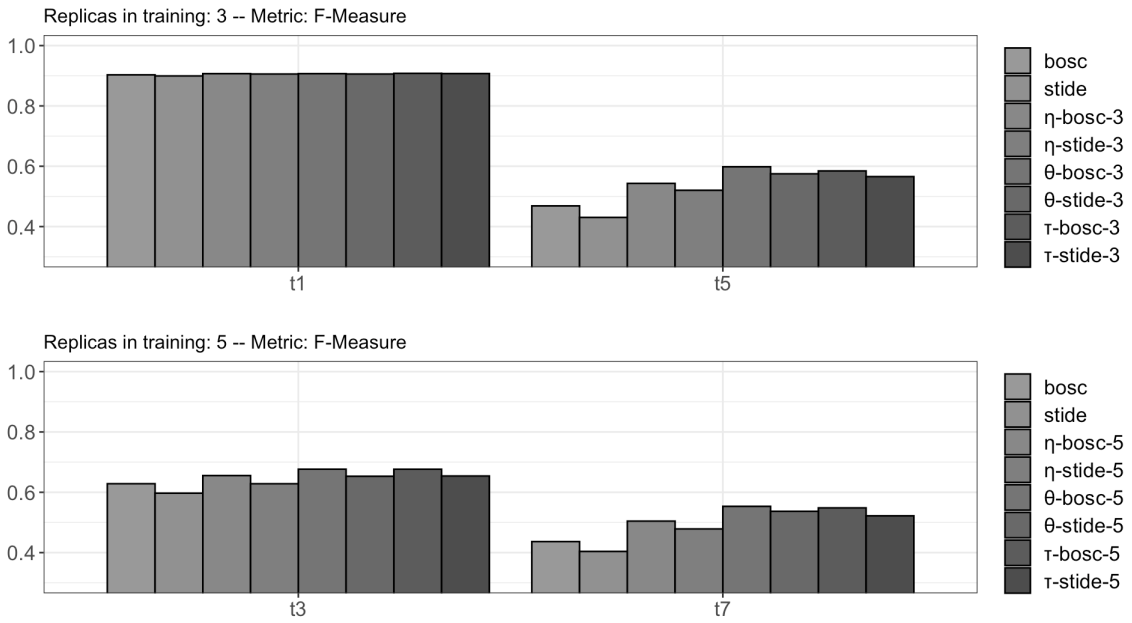


Figure 3.21: Performance of the top 10 configurations for scalable scenarios.

Although the technique proves to be able to achieve generalisation of acquired knowledge, there is still room for improvement towards higher effectiveness. Nevertheless, it is still satisfactory and decreases the time-to-protection, as it removes the need to adapt the profile according to the autoscaling modifications.

3.2.10 Discussion

The evaluation of the approaches demonstrate the applicability and increase in effectiveness when compared to simple application of intrusion detection algorithms. The approach can operate in diverse configurations regardless of the learning environment, generalising the acquired knowledge, and effectively detecting most of the attacks in both scaling scenarios. In fact, algorithms with data processing increase the effectiveness of detection in 80% in the best cases, which correspond to the most complex scenarios of seven replicas during detection.

Configuration Tuning

The methodology also allows for extensive configuration so that users can best tune it to their use case to ensure the best results possible. In our experiments, the careful configuration proved worthy as our results improved significantly with the use of smaller window and epoch sizes. However, this may trend differently depending on the target system and its nature. Therefore, a careful testing and evaluation is recommended for each system. All these factors contribute to behaviour profiles that demonstrate being effective at generalised application with different number of replicas in the set profiled.

In fact, behaviour profiles achieve a clear distinguishability between normal and malicious events. This is demonstrated through the very low frequency of alarms raised before systems are attacked. During the pre-attack phase, the services are operating without any kind of disturbance being conducted; thus, any raised alarm is clearly a false positive. On the other hand, alarms raised in the post-attack phase (after the attack) are not so clearly defined. As the system was subjected to malicious events, it is possible that its internal state was disturbed or contaminated. As a result, there are some cases where the alarm frequency spikes, achieving levels similar to those registered during the attack phase. These modifications are easy to spot and reinforce the quality of the behaviour profiles.

Intrusion Detection in Scalable Microservices

Our evaluation considers four types of workloads resulting from the combination of intensity loads and user profiles, which are increments from the simple ones. This can be viewed as a threat to the diversity of the systems' behaviour produced. Still, the modifications are significant, as, for instance, one of the user profiles creates modifications in the state of the systems data and the intensity loads have different nature, being one more stable and the other having oscillations in the number of requests executed. Thus, the result consists of significant modifications in the pressure over the target system, which influences its behaviour.

Regarding the attacks executed to produce malicious events, these are mainly focused on the kernel supporting the system, despite three of them targeting the Tomcat server and one being directed at Docker, which creates and manages the containers supporting the services. To compensate for this imbalance, all the attacks used target real-world vulnerabilities with CVE ids being made available. In fact, the categorisation of the vulnerabilities allows to understand the effectiveness of the detectors when facing their different kinds. The algorithms with data processing can clearly detect 3 out of the 4 vulnerability classes presented in this work. With High-Level vulnerabilities attacks being harder to distinguish, as they are hardly reflected in the system call data. In this concrete case, the attack is an XSS attack that does not modify the system at the system call level, which in fact reinforces that the algorithms with data processing are indeed detecting the attacks and not raising alarms by chance.

The operation of our approaches intends to tackle the existence of multiple repli-

cas of a service, by considering all replicas as a set and obtaining a profile of it. The main goal is achieved, as the approach effectively detects malicious events in direct applications of the profile and in scale-down and scale-up scenarios. Our evaluation shows that knowledge is generalised effectively and applicable to diverse scenarios, which reduces the need for multiple profiles of the various configurations that a system may have during its adaptive operation.

Chapter 4

Detecting Attacks in Multiple Microservices

This section is split as follows: we will now present the approach and going over our experimental methodology explaining the dataset collection and the algorithms; then we will analyse the fusion techniques employed in Subsection 4.2; in Subsection 4.3.2 the attacks scenarios performed, going over the testbeds used; and finally an analysis of the results in Subsection 4.4.

Our focus is on detecting intrusions targeting multiple services in a microservice-based system, however, it provides more than that, with the proposal and validation of an approach to deal with it. We also propose a set of four fusion techniques which allow the fusion of data from multiple sources, and evaluate them.

4.1 Host-Based Intrusion Detection Approach

The approach was devised to support training dataset size reduction and fusion techniques.It improves by adding two different new phases that allow researchers and architectural designers to explore a new variety of scenarios. Figure 4.1 shows the approach.

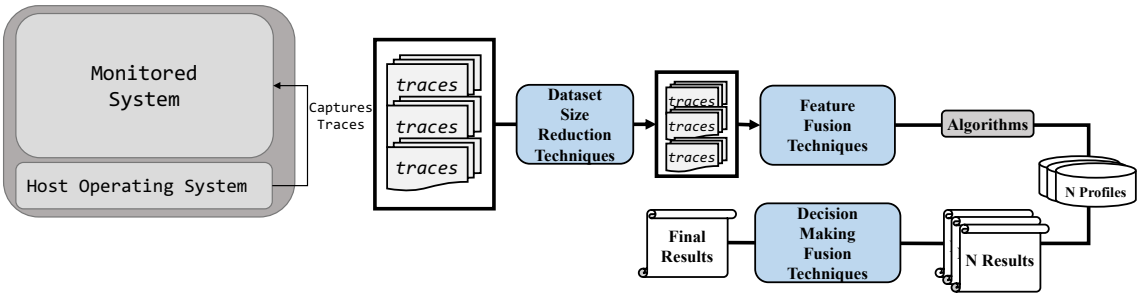


Figure 4.1: Intrusion Detection Approach

The goal of this approach is to support adaptation according to the user needs. The techniques displayed in blue are optional, meaning that if one of those tech-

niques is not being studied or needed, then that component is not necessary. This allows a wide range of combinations to be studied.

The host collects the system calls, originating a trace file per component being monitored. These trace files can then be subjected to a dataset reduction, following a technique, resulting in a smaller set that can then be used to apply the fusion techniques. Then, with the help of the algorithms and the profiles previously constructed, we get the results that can, in the end, suffer another fusion. Although intermediate files can be created, such as the smaller traces represented in the figure, everything also works in a stream, making it possible to implement in a run-time environment.

4.1.1 Experimental Methodology

A new experimental methodology was depicted based on the approach previously presented, supporting a loop removal algorithm to reduce dataset size and fusion techniques. This experimental methodology has both the purpose of validating the approach and of evaluating the effectiveness of the techniques proposed. Like in previous work, we use system calls, obtained through sysdig [Sysdig, Inc, 2019], to feed information to our HIDS and generate traces. We will only study the use of feature-based fusion techniques independently of decision-making fusion techniques, which means that when one is being used, the other component is not. The resulting diagram can be seen in Figure 4.2.

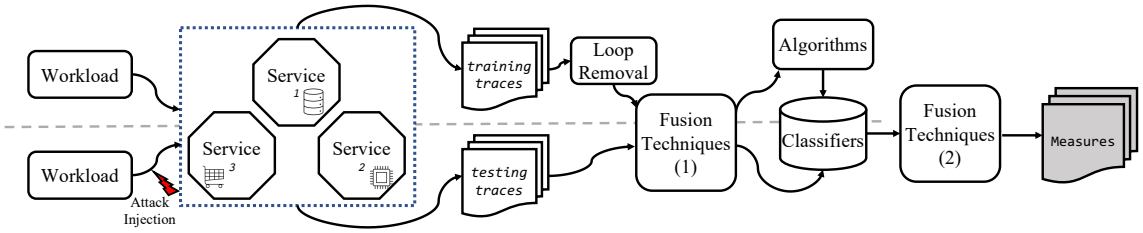


Figure 4.2: Methodology for the experimental campaign to detect attacks in microservice systems

During the training phase, the system received two different intensity workloads. One is stable throughout the experiment, while the other is variable, having peaks that represent periods when the number of users is higher. The intensity was adjusted to the system, so to allow a low value of failed requests (<1%) during the peak times. Two different profiles were also used, one in which the user is only browsing the web service and another in which the user interacts with it, performing some purchases. The testing phase has the same workloads; however, attacks are performed on the system. We then get, for each run, one trace file per service.

Although training traces will first go through a loop removal algorithm, testing traces will not suffer any modification. It will remove loops of 2, 3, 4 and 5 system calls, saving them for later enhancement of the classifiers. After the loops are removed, the dataset is ready for training.

Although it is possible that an experiment can be conducted using both Fusion Techniques (1) and (2), in our experiments, these are mutually exclusive, when (1) is applied, (2) is not, and vice versa. One of the two types of fusion techniques being evaluated, feature level, may modify the dataset before being fed into the algorithms, as represented in the figure, by the Fusion Techniques (1). Then, either we train the classifiers, if we are in the training phase, or we evaluate them and get the measurements when testing. In the scenario where Fusion Techniques (2) are applied, these fuse the classifiers together, through the use of a decision-making technique. The results from the techniques allow us to obtain the measurements. We used two different algorithms to generate the classifiers, BoSC and STIDE.

Taking into account a dataset per service and considering that two workload profiles and two different intensities were used, a total of 92 datasets were collected for training. For testing, for each exploit, three different runs were made, which in total made for 1464 datasets. Each dataset for training comprehends a total of 24 hours of system call collection, while the testing traces only have 30 minutes. It is worth noting that, in total, before removing the system calls, the training traces are over 1.6 Terabytes of data, while the testing traces are of around 450 Gigabytes.

4.1.2 Loop Removal Algorithm

One of the problems we try to deal with is the large size of the datasets. Currently we are storing more than 5 TB of data, mostly compressed and even with some classifiers missing, since they sometimes need to be removed during the experiments in order to prevent running out of space. Also, since the files are so large, it becomes a tedious and time-consuming task to train many different classifiers with different algorithms and configurations. The main goal of the loop removal algorithm is to tackle this issue by removing redundant information from the datasets, therefore, allowing for a larger range of algorithms and techniques to be tested in a shorter period of time; but also without significantly interfering with the effectiveness of the classifiers. This algorithm is based on previous work done, that uses system call loops to reduce the amount of system calls in malware scenarios [Alaeiyan and Parsa, 2015]. The core of the algorithm can be seen in Figure 4.3.

This algorithm runs through the file, and whenever a loop of size equal to the *step* is found, a merge happens, saving the loop. A sliding window, declared as *system_call_sequence* was used to go through the file, and is of size equal or lower than *step* times two. In the end, a new file is generated without the loops found. We look for loops with sizes 2, 3, 4, and 5, therefore, we have to go through a file 4 times in total, iteratively, one per loop. There is room for improvement to make it look for all the loops in one go; however, the complexity of the algorithm gets much bigger.

It is worth noting that for our research we only consider the system call, and not any of the parameters being used; however, considering a loop simply by

```
    finished = False
    merged = False

    system_call_sequence = list()
    with open(initial_dataset, "r") as initial_dataset_fp, open(cleansed_dataset, "w") as cleansed_dataset_fp:
        # Reads file until the end
        while not finished:
            # Gets the next system call sequence, while it is bellow the step and not the end of the file
            while len(system_call_sequence) < step and not finished:
                # Reads the next line
                line_data, initial_dataset_fp = read_line(configuration, initial_dataset_fp)
                if line_data:
                    system_call_sequence.append(line_data)
                else:
                    finished = True
            # If it is not the end of the file, then a system call sequence was found
            if not finished:
                # If the system call sequence is a loop, merge the system calls and declare merged as true
                if system_call_sequence_is_loop(system_call_sequence, step):
                    system_call_sequence = merge_system_calls_version_1(system_call_sequence, step)
                    merged = True
                # If not, then
                else:
                    # If a loop was found (system calls were not merged), then write the whole sequence
                    if merged:
                        while len(system_call_sequence) > step // 2:
                            system_call = system_call_sequence.pop(0)
                            write_one_line(cleansed_dataset_fp, system_call, all_data)

                        merged = False
                    # If it was not, then write the last system call
                    else:
                        system_call = system_call_sequence.pop(0)
                        write_one_line(cleansed_dataset_fp, system_call, all_data)
        # When we finish reading the file, we need to write the remaining system calls
        for system_call in system_call_sequence:
            write_one_line(cleansed_dataset_fp, system_call, all_data)
```

Figure 4.3: Core of the Loop Removal Algorithm for a certain *step*

the system call would be misleading since one can have different parameters. Therefore, we also look at the parameters that are used when calling the system call. This part of the detection can be seen in Figure 4.4.

```
# Checks if a system call sequence is a loop
def system_call_sequence_is_loop(system_call_sequence, loop_size):
    step = loop_size // 2
    # Evaluates if the 2 segments of system calls are the same, "X" = "X+step"
    for j in range(step):

        if system_call_sequence[j]['syscall'] != system_call_sequence[j + step]['syscall'] or \
                system_call_sequence[j]['num_args'] != system_call_sequence[j + step]['num_args']:
            return False

        else:
            for i in range(system_call_sequence[j]['num_args']):
                if system_call_sequence[j]['arg_names'] != system_call_sequence[j + step]['arg_names']:
                    return False
    return True
```

Figure 4.4: Detection of a loop

One of the problems with removing loops comes from the fact that the classifiers may become incomplete, since some windows were lost in the process. Although this may be true for more complex algorithms, for the standard BOSC and STIDE, not much is lost in the process, depending on the size of the windows. If the size of the window is smaller than the size of the loop plus one, then we can get all the combinations; therefore, the classifiers are not incomplete. We will now prove that, by looking at Figure 4.5 which shows an arbitrary sequence of system calls with loops.

First of all, we should know that by saving the loops, we are able to reconstruct the sequences as further as necessary. Now, considering a window with size equal or lower to the loop, it is trivial to know that nothing is lost since the only combinations removed are inside the loop, which can then be recovered by duplicating the saved loop and then obtaining the combinations. When the size of the win-

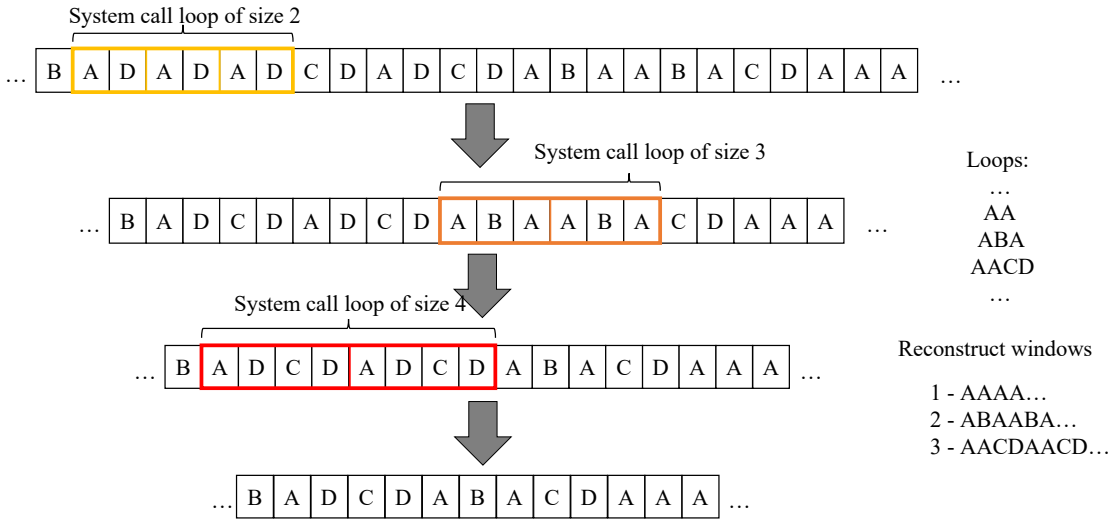


Figure 4.5: Example of removing system call loops. A, B, C and D are different system calls.

dow is one unit larger than the loop, although harder to see, it is still recoverable in the same manner. For example, let us focus on the first step, for a loop size of 2. If the window size is 3, then we can get BAD, AD and ADC, and by reconstructing the windows, we also get DA, getting all the combinations. The problem appears when the size of the window is enough to go all the way from the first system call to the left of the loop, up to the first system call to the right of the loop. When this happens, such as with a window of size 4 with a loop of size 2, memory would be needed of the windows to the left and to the right of the loop in order to then reconstruct as needed. In our example, for a window of size 4, the sequences BADA and DADC are lost. It is also worth noting that a new system call sequence is generated, BADC.

As we can see, it is factual that some sequences are lost; however, we believe this to be negligible as it is a very small part of the whole classifiers. Various different algorithms could be upgraded to fix this issue, such as duplicating the loop, guaranteeing that the problem would appear only for window sizes of 6 or more.

Results

Regarding the size of the files, overall, we were able to reduce the size of the datasets to 55.01%, more specifically, the files related to sockshop were reduced to 53.97% and, for teastore, to 55.43%. Figure 4.6 shows the same metric, however, more detailed, going in depth to the different services and workloads, while Figure 4.7 displays the size of the files in GB.

As we can see, the effectiveness of the loop removal algorithm is not directly related to the size of the files; however, some services do display changes when different profiles are used. That can be explained by the fact that the *Buy* profile exposes the system to new interactions; therefore, if these new interactions are inherently more prone to loops, the reduction will be higher, as we can see in the *webui* service; but if they are less prone to loops, then the reduction will be lower,

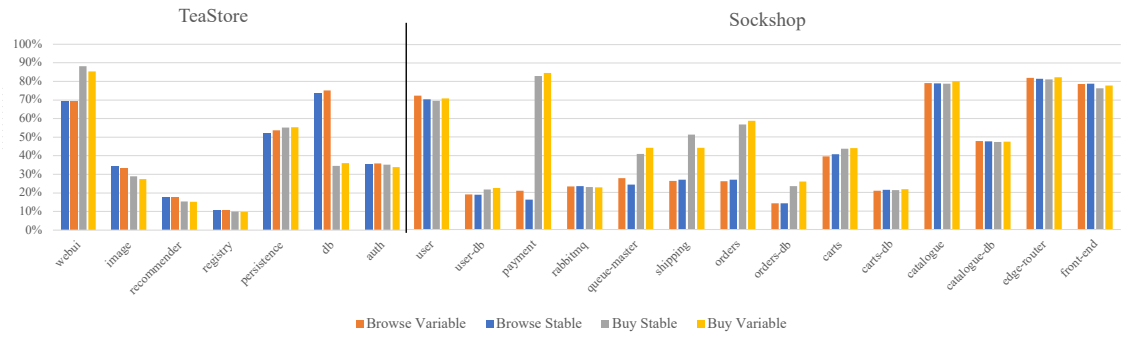


Figure 4.6: % occupied by the files, of the services, according to the training workloads, after removing the loops.

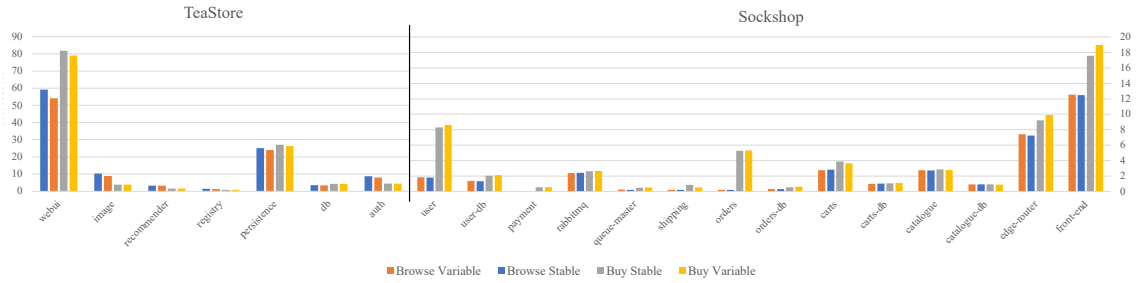


Figure 4.7: Size of the files, of the services, according to the training workloads, before removing the loops.

like in the *db* service. Other services, since their behaviour is independent of the profile, differences are insignificant.

4.1.3 Anomaly Host Based Intrusion Detection Algorithms

Our solution uses an Anomaly host based IDS, focussing on the use of algorithms that gather patterns from system calls. These are BOSC and STIDE, which focus, respectively, on the frequency of system calls and the sequence of system calls. With BOSC, the focus is on the frequency of system calls; when training, it records all unique windows. With STIDE, on the other hand, it takes into account the sequence of these system calls, also recording the unique windows, which results in larger profiles. Figure 4.8 shows an example of a training period with BOSC and STIDE.

The test data will then be passed through one of these algorithms, and all windows that deviate from the benign window patterns will be considered preliminary anomalous behaviour. Due to the large number of system calls and the wide variety of them, it is expected that some windows are not recorded in the classifiers; therefore, simply using a window to determine if an intrusion occurred would result in a very large number of false positives and alerts. Therefore, we join the windows together into epochs. The epochs will then be analysed, one at a time, and if the number of anomalous windows, which are windows not present in the classifiers, exceeds the detection threshold, then the epoch is considered

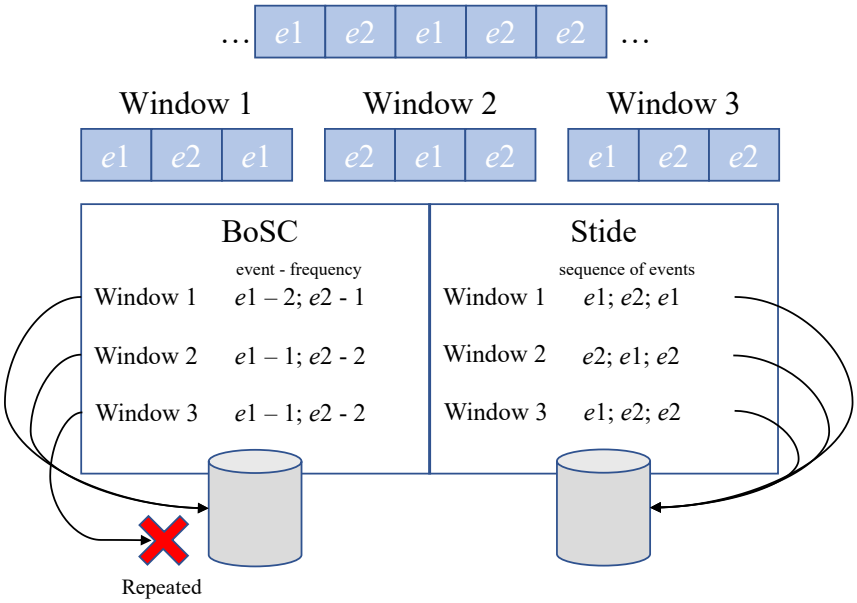


Figure 4.8: Training with BOSC and STIDE algorithms. *e* stands for event (system call), and the number following it represents its uniqueness

anomalous, indicating that an intrusion happened. The size of the epochs and the detection threshold are parameters that can be configured. These parameters were previously analysed in Section 3.2.3. Figure 4.9 shows an example of an epoch analysis for two epochs.

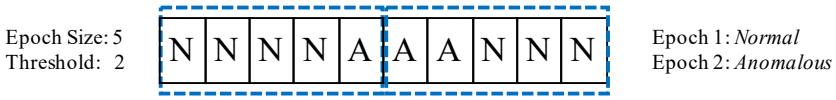


Figure 4.9: Epoch analysis

Epochs can group together, giving a lot of alerts, especially after an attack happened. For this reason, to facilitate the administrator, presenting more clear alerts, we group epochs inside a time interval, for example 5 seconds, and if at least one epoch is classified as anomalous, the whole set is considered an attack period, and an alert is raised. This replicates a real system where the administrator will only receive an alert for that interval, instead of being bombarded by numerous alerts; it also enables a normalisation of the data, making it simpler to analyse the absolute values of the different metrics for the different scenarios, since some runs may generate more system calls than others. An example is shown in Figure 4.10.

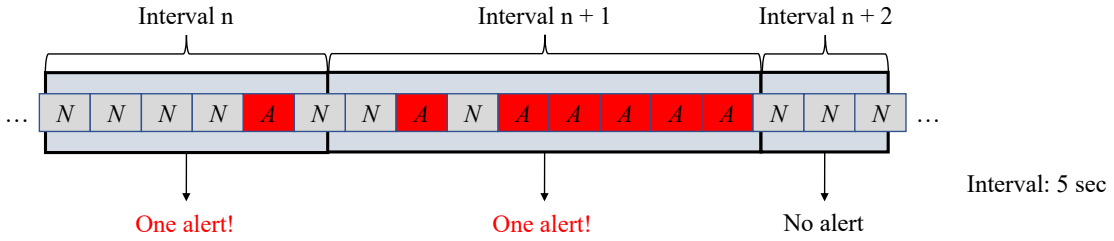


Figure 4.10: Time interval analysis

4.2 Solution based on Fusion Techniques

When looking at feature-based techniques, data has to be transformed or manipulated before being fed to the algorithms, creating a single profile. With decision making techniques, data are fed to the algorithms, generating one profile per service, this will then be used in the detection phase to obtain the results, which will then be merged by the techniques. Figure 4.11 shows an example, with the left side being for feature-based techniques and the right side for decision making.

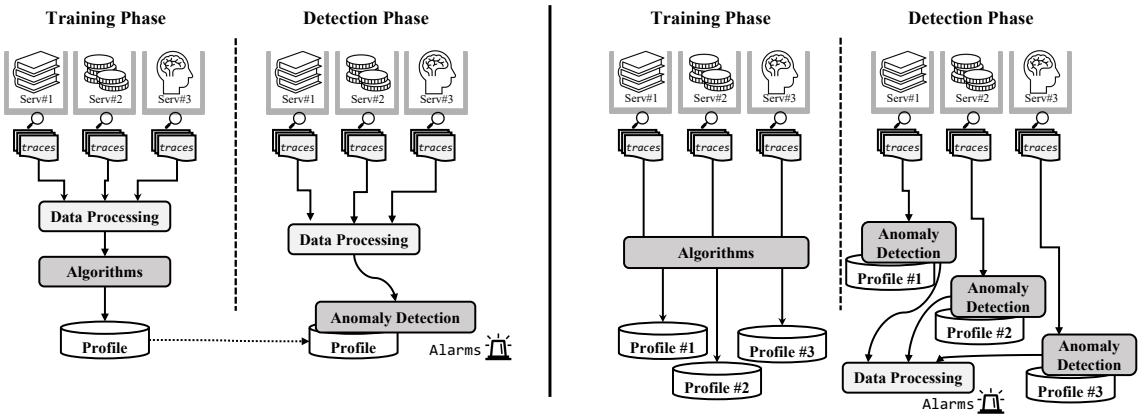


Figure 4.11: Fusion techniques approach

In order to cope with the variable number of services and to have a generalised view of the system, four different fusion techniques were evaluated. Two of them are based on a fusion at the feature level, while the other two focus on the decision making level. We will now go in depth on each technique.

4.2.1 Feature Level Fusion Techniques

Feature Level Fusion is based on the principle of fusing all the traces from the different services. It allows for a more centralised IDS system, while also having more possibilities since the raw traces are used. Previous techniques used to deal with multiple replicas fall into this category.

Service Interleaving

The first technique uses chronological sorting, splitting service data into chunks based on a time interval, and then interleaving them. This allows the system calls to maintain their cohesion within the same services, while also enabling some relationship between services. Figure 4.12 displays the different steps.

S_n and e_m is the representation of the system call (event) m from service n, with Δt as the time interval of the interleaving process, and T an arbitrary timestamp of a system call. Traces are grouped according to their services and timestamps. These groups are then interleaved with each other, one service at a time, following a predefined arbitrary order. So, for example, for service #1, the system calls 1 to 4 are chronologically ordered, and their timestamps are between

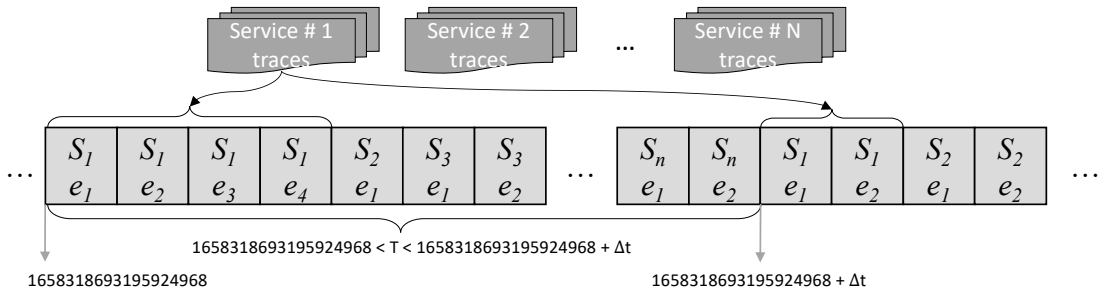


Figure 4.12: Service Interleaving technique

1658318693195924968 and 1658318693195924968 + Δt. We only guarantee that system calls of the same service are chronologically ordered and, inside the same interval, traces are grouped by service. A system call can therefore have a higher timestamp than the following if they are from different services and within the same interval.

The idea of this technique comes from other tools and systems that use chronological order to group data and from our previous studies [Jones and Sielken, 2000; Kemmerer, 1997]. Its foundations lays on the fact that system calls have a relationship between them, so, simply ordering chronologically would break that relationship inside the same service. We considered a Δt equal to 5 seconds, but other intervals can be used and tested following the same methodology.

BoSC and STIDE Matrix

This technique is an extension of the BoSC and STIDE algorithms, where multiple windows are taken into account, building a matrix, enabling the use of a single profile for the whole system. The training and the detection phases start when at least one window per service is read, which means that each service has a minimum of n system calls, with n being the size of a window. Considering the amount of system calls recorded, this is not a problem.

As stated previously, BoSC and STIDE are algorithms that use a sliding window in order to build the classifiers, during the training phase, and to detect intrusions in the detection phase. A window is classified as anomalous, meaning that the algorithm considers an attack to be occurring if the window was not recorded during the training phase. With this technique, instead of simply looking at the windows of each service, the combination of windows is taken into account. An example is shown in Figure 4.13.

For Service #1, each window will have a list of windows possible by service #2, which in turn will do the same for #3 and so on until the last service, which in this case is #4. In the end, we get a matrix that is filled in with the accepted states of the system; anything that is not saved in the matrix is considered anomalous behaviour. It is expected that profiles are much more complex, occupying, and requiring much more memory, scaling with the number of services. This obviously will not be scalable; however, it would allow us to test if it is possible and effective in smaller microservice systems, which in turn would allow for new research in, for example, joining services that are similar and then applying another

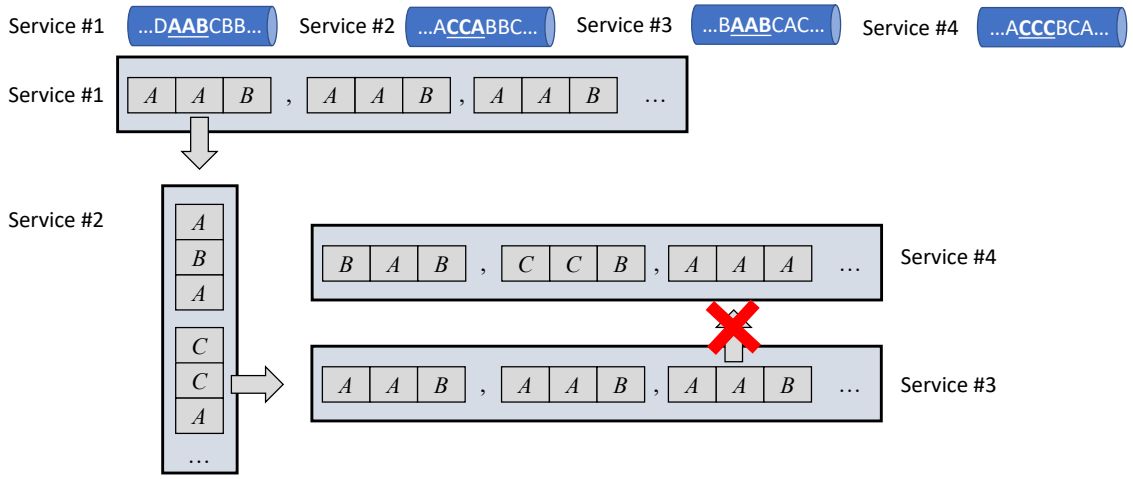


Figure 4.13: BoSC and STIDE Matrix example

technique on top.

After the first few tests, it seemed impossible to create profiles due to the fact that there was no convergence and the system memory was not sufficient. We therefore adjusted and tried running the algorithm on after the services were interleaved, since we were likely to get fewer new profiles. Although we managed to train the profiles in the harder scenarios, we did not get a good growth curve. Figure 4.14 shows three examples of classifier growth curves, one showing growth following the original approach, and then two with the interleaved approach, one for TeaStore and one for Sockshop. Keep in mind that x is the time in minutes of the traces read and y the memory used in GB.

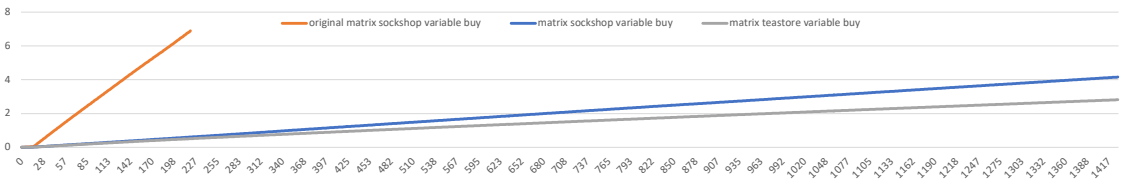


Figure 4.14: STIDE with Window of size 4, classifier growth curve, considering the original and the interleaved matrix approaches, for the variable workload with Buy profile, for Sockshop and TeaStore.

As we can see, neither of them converges, therefore bigger training times were probably needed to get good results. Sockshop has double the amount of services TeaStore has, 14 to 7, and the size of the classifiers is 148% bigger, 4.23 to 2.86 Gigabytes, allowing us to conclude for certain that the algorithm as it is, is not scalable.

4.2.2 Decision Level Fusion Techniques

For decision-level fusion techniques, only the decisions of the different classifiers are merged, based on a model or technique. These techniques tend to be more lightweight and simple since only the decisions need to be merged; however,

they are also less powerful since the raw data cannot be used. For each service, or group of services, a classifier has be be running, feeding the information to a system that then makes the final decision. Grouping services, before the decision level technique, is not taken into consideration as it would be impossible due to time constraints Figure 4.15 shows a generalised view of how the process works for the techniques we used.

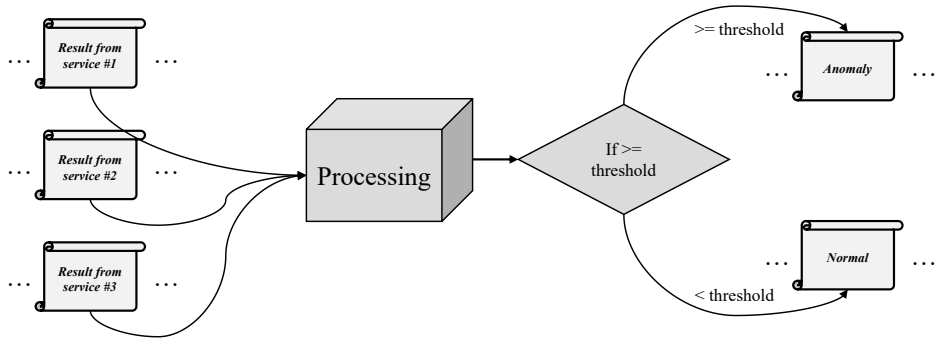


Figure 4.15: Decision making processing

Voting technique

In this technique, all the services vote in order to determine whether an intrusion affects the whole system, is or not happening. The results of each service are grouped in epochs, just as before, outputting a single binary result based on a detection threshold parameter. These results are then aggregated, and if the number of services being classified as anomalous is higher than or equal to the threshold, then it is considered an alarm. The final results obtained are only related to epochs and time, as the model aggregates the results of the epoch analysis. In theory, we could apply the voting threshold to the windows, and only after proceed with the epoch analysis, however, a single anomalous window in a service that is more passive would have much more impact on the result.

Decision Model

Similarly to the voting technique, a decision model can be used in which, instead of having each service give a simple binary decision, a percentage is given, which indicates the certainty of each technique. In our case, this percentage is the fraction of windows anomalous inside an epoch. We would then apply a global detection threshold to evaluate the system as a whole, thus enabling the detection of intrusions that may affect various services, but are much more subtle. This model, as expected, does not allow for an analysis of the windows.

4.3 Validation and Evaluation

To validate our approach, we evaluated the effectiveness of our solution in representative scenarios and confirmed that it allows us to perform a variety of scenarios, working for both the training and detection phases. For evaluation, various testbeds were analysed and tested, such as TrainTicket [Zhou et al., 2018], to determine which are the most representative of real-life scenarios. For each

experiment, a Virtual Machine was created and started, and then the system was deployed. The Virtual Machine had 25GB of memory. The host machine has an Intel Core i5 clocked at 2.80GHz CPU and 32GB of RAM. The client, on the other hand, had a load generator running and the machine had an Intel Core i3 clocked at 3.50GHz CPU and 16GB of RAM.

4.3.1 Testbeds & Workloads

In the end, two mostly out of the box representative testbeds were chosen to carry out the experimental campaign. These are:

- TeaStore [von Kistowski et al., 2018]
- Sockshop [Sockshop, 2021].

As previously stated, TeaStore is an e-Commerce application that contains five different services, a registry, and a database. Every service is written in Java and running on a Tomcat server, with the exception of the Database, which is running a MariaDB server. WebUI is normally the most stressed service, followed by Persistence.

Sockshop has many more services than TeaStore, 12 in total, if we count the four databases and one message queue. It has the front-end built in NodeJS that then contacts 5 other services: Cart, which is built in Java and has access to a MongoDB Database, it takes care of the carts of the clients; Catalogue, which uses GO and a MySQL database, being responsible for showing the catalogue to the user; User, written in GO and with a MongoDB Database, it responds to requests related to the user such as login; Payment, built with GO, it takes care of the payments; and finally Order, written in both Java and .NET Core, it has access to a MongoDB Database, taking care of the orders of the users. The Order service then communicates with the Shipping service, which then inserts a message to a RabbitMQ queue, that will then be read by the Queue-Master, built in Java, that will then take care of the shipping process.

Table 4.1: Technologies used by the services.

Testbed	Technology	Service
TeaStore	Tomcat	WebUI, Image, Recommender, Registry, Persistence and Auth
Sockshop	SpringBoot	Order, Cart, Shipping, Queue-Master
	MongoDB	Order-DB, User-DB, Cart-DB
	MySQL	Catalogue-DB
	NodeJS	Front-End

4.3.2 Attack planning

The attacks were performed considering a multi-tenant scenario where the attacker takes advantage of the fact that the cloud provider shares the same in-

frastructure for their clients. Therefore, all attacks are remote. We also assume that the frontend service has at least one point of entry, as usual in this kind of system; and that every service has root access inside their container. Finally, the same network is shared between all services.

Figure 4.16 shows how the experiments were carried out.

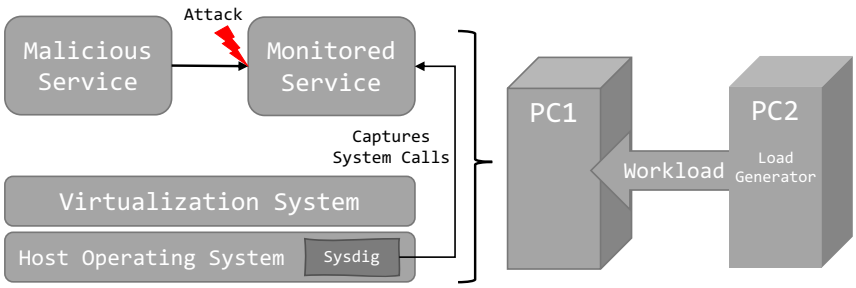


Figure 4.16: Conceptual Solution

These experiments were carried out in 50 minute slots: 15 warm-ups, 30 workloads, and 5 cooldowns. The attack was performed at the 30 minute mark, during the middle of the workload with the help of the Metasploit framework. The collection of system calls occurred during the workload. The timeframe is shown in Figure 4.17.

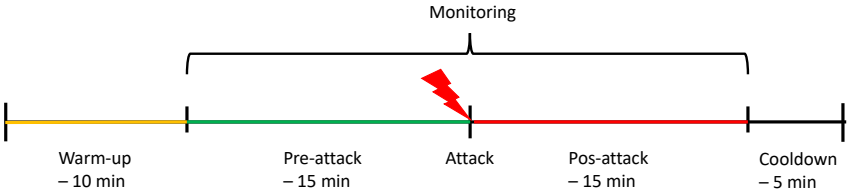


Figure 4.17: Attack injection experiments timeframe

In terms of exploits, 12 were used, targeting a total of 5 different services, these being: Apache Tomcat, Spring Boot, MongoDB, MySQL and Nodejs. First, we present the exploits related to the TeaStore testbed, with unsuccessful attacks marked as such. These exploits can be found in all the TeaStore services, except the database, meaning that in total 7 services are attacked.

We used Metasploit to help us attack the system and these were the types of exploits we used:

- **DoS**;
- **RCE**;
- **XXE**;
- **Bruteforce**;
- **Configuration exploit**;
- **Buffer Overflow**;

- **Information Gathering**.

Exploit 1 - CVE-2020-1938.

Apache JServ Protocol (AJP) has a trust issue that allows an attacker who communicates through AJP to have higher privileges than when communicating through http, enabling privelage escalation scenarios. In our case, the attacker uses this to gather information about the web server; more specifically, he obtains the web.xml file from the target [nvd, 2020].

Exploit 2 - CVE-2010-2227.

An attacker can take advantage of a Tomcat vulnerability that allows him to cause a denial of service, through a crafted Transfer Encoding header, interfering with "recycling of a buffer" [nvd, 2010].

Exploit 3 - Missconfigured Tomcat Manager [Unsucessfull].

Here, the attacker tries to upload a WAR archive that contains a jsp application that, in reality, is the payload that forces a POST request to the endpoint /manager/html/upload component. It takes advantage of many vulnerabilities related to a missconfigured tomcat deployment [rangercha, 2021].

Exploit 4 - Bruteforce Tomcat Manager [Unsucessfull].

The attacker goes through a list of known user-password combinations and tries to login to the Tomcat Application Manager [MC, 2021].

Now, we will go through the exploits for the Sockshop testbed. Unsucessful attacks are also marked.

Exploit 5 - Jolokia XXE.

This exploit targets JAVA application servers that use Spring Boot and have a bad configured Jolokia, allowing an attacker to perform a XXE. It forces the server inside the container to execute a GET request for a file to a malicious server that will then redirect to another malicious server that has the entity to inject. In our case, this XXE allows the attacker to get the list of users and passwords of the container [mpgn, 2018; Stepankin, 2019]. In total, four different services were targeted.

Exploit 6 - Jolokia RCE.

The logic is pretty similar to the past exploit, with all the same configurations in the target server, however, even more ingenious. Instead of the second malicious server being an http server, it is now an RMI server. The first malicious server will then execute a call to the RMI server, but instead of simply giving an entity, it forces the victim to open a reverse shell by communicating with a third malicious server that has a netcat open port. Therefore, through this third server, the attacker has an open shell directly connected to the victim, allowing him to do whatever he wants [mpgn, 2018; Stepankin, 2019]. Just like in the last exploit, four different services have this vulnerability.

Exploit 7 - Bruteforce MongoDB.

Just like exploit 4, the attacker here tries to brute-force a set of passwords for MongoDB [Man, 2019]. This is done in three different services.

Exploit 8 - CVE-2013-1892.

MongoDB has a JavaScript engine with spiderMonkey, which in turn has a feature called nativeHelper that can be exploited. An attacker can take advantage and, by calling it with crafted arguments, use a RCE. [agix, 2020; nvd, 2013b]. Simillar to exploit 7, three different services are exploited.

Exploit 9 - MySQL Server Version Enumeration.

Here is a classic example of information gathering. The attacker executes a simple attack that gives him information about the MySQL server, which in turn can be used for other attacks [kris katterjohn, 2017]. For this exploit and the following 3, only one service is targeted.

Exploit 10 - CVE-2012-2122 [Unsucessfull].

This exploit takes advantage of a bug in the memcpm function that allows, under certain conditions, for a return value to be handled as true when it should not. An attacker can abuse this vulnerability by repeatedly trying to authenticate with the same password until it is accepted [nvd, 2012]. In our case, the attacker then tries to extract the usernames and encrypted password hashes from the MySQL server, which can be cracked later [theLightCosine, 2021].

Exploit 11 - CVE-2008-0226 [Unsucessfull].

There are multiple buffer overflows in yaSSL 1.7.5 and earlier, and MySQL uses this library. Attackers can perform a RCE by abusing a buffer overflow in the ProcessOldClientHello function, for example [nvd, 2008]. In our scenario, the attacker can abuse it by sending a specially designed Hello packet [MC, 2020].

Exploit 12 - CVE-2013-4450.

An attacker can send, to a Node.js server, a large number of pipelined requests, inside a single connection, without reading the responses. This will cause the server to allocate unbounded memory, leading to a remote DoS [Marek Majkowski, 2020; nvd, 2013a].

Based on these exploits, 10 different attack scenarios were constructed. Table 4.2 presents these and provides more details.

4.3.3 Configurations

Unfortunately, not all classifiers were able to be tested; however, we consider that those were not as important, based on the results that we were getting. Even then, most of the configurations were tested.

Considering the single classifiers, for each service, we did not manage to evaluate TeaStore with the stable workload. Therefore, we also did not test the decision-making techniques with that workload. For the matrix technique, we tested the

Table 4.2: List of Attack scenarios.

Testbed	Scenario and Exploit #	Targeted Services	Vulnerability Type(s)	Successful
TeaStore	1 - Exploit 1	Tomcat	Information Gathering Privelege Escalation	✓
	2 - Exploit 2		DoS	✓
	3 - Exploit 3		Configuration Abuse	x
	4 - Exploit 4		Bruteforce	x
Sockshop	5 - Exploit 5	SpringBoot	XXE	✓
	6 - Exploit 6		XXE, RCE	✓
	7 - Exploits 7 and 8	MongoDB	Bruteforce, RCE	✓
	8 - Exploit 9	MySQL	Information Gathering	✓
	9 - Exploit 10 and 11		Privilege escalation, Buffer Overflow, RCE	x
	10 - Exploit 12	NodeJS	DoS, Overflow	✓

variable workload with the buy profile, as it is the most intensive for the technique. Table 4.3 shows the different classifiers tested.

Table 4.3: Classifiers tested

Classifiers	Workload Training							
	TeaStore				Sockshop			
	Stable		Variable		Stable		Variable	
	Browse	Buy	Browse	Buy	Browse	Buy	Browse	Buy
Individual	x	x	✓	✓	✓	✓	✓	✓
Matrix	x	x	x	✓	x	x	x	✓
Round Robin	✓	✓	✓	✓	✓	✓	✓	✓
Voting	x	x	✓	✓	✓	✓	✓	✓
Decision	x	x	✓	✓	✓	✓	✓	✓

Regarding the configurations tested, both BOSC and STIDE were used, with both window sizes of `3` and `4`. When it comes to the epoch sizes, `500`, `1000` and `5000` were tested, in combination with the detection thresholds `5`, `10`, `20`, `50` and `100`.

4.4 Results

In this section, we will present the results of the experiments, starting from the individual classifiers and noting some important observations. Then we analyse the effectiveness of the fusion techniques with each exploit. On the basis of previous work, we focused our analysis on the pre-attack and attack phases, since containers can become unstable after being attacked.

4.4.1 Training Workload and classifier effectiveness

We will now look in depth at the classifiers of the different services and how well they performed individually, without any techniques, against their respective testing traces. First, we will focus our attention on the general results for Sockshop and TeaStore, which are represented in Figures 4.18 and 4.19, with the data grouped considering the training workloads used.

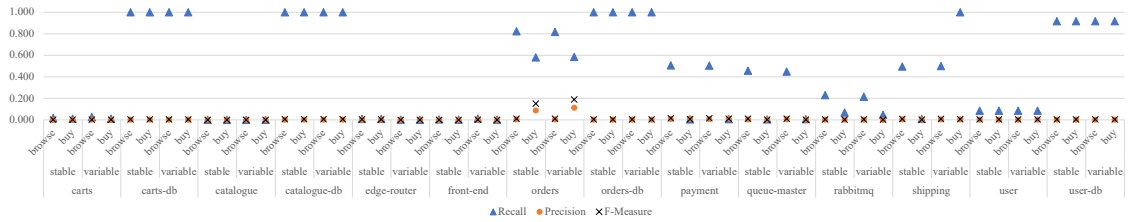


Figure 4.18: General Results Sockshop with different training workloads

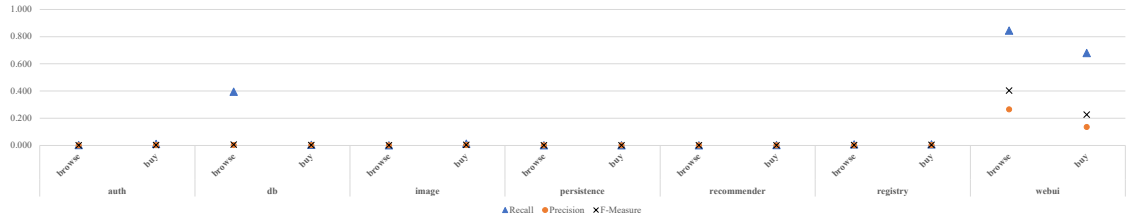


Figure 4.19: General Results TeaStore with different training workloads

Overall, the results are disappointing, with only some of the services detecting the attacks and with many false positives. It should be noted that not all services were targeted; therefore, it is not expected that classifiers for a service that was not attacked will still be able to detect attacks. Even then, some classifiers were detecting all attacks, but with very low precision, indicating that they were classifying almost everything as an attack.

We can see some differences in the results when different workloads were used, as expected. To get a clear view, Figures 4.20 and 4.21 show the improvements in the metrics. On the left, we can see the percentage difference when using the buy profile instead of browse, and on the right, the difference when using the variable intensity instead of stable. Keep in mind that we only analysed the difference between browse and buy in teastore, since we did not get the data for variable vs stable; and also, in sockshop, we focused only on the services that were targeted by the attacks.

For the most part, using the buy workload indeed gives better results than using the browse workload, which was already expected due to our previous work. This can be justified by the fact that the profiles are more complex and have more data, making the classifiers more complete. When looking at the intensity, we had to remove the shipping service from the comparison between the variable and the stable intensities because it would not allow us to see the other services, as the difference was substantially greater, at 10090%. The variable intensity, due

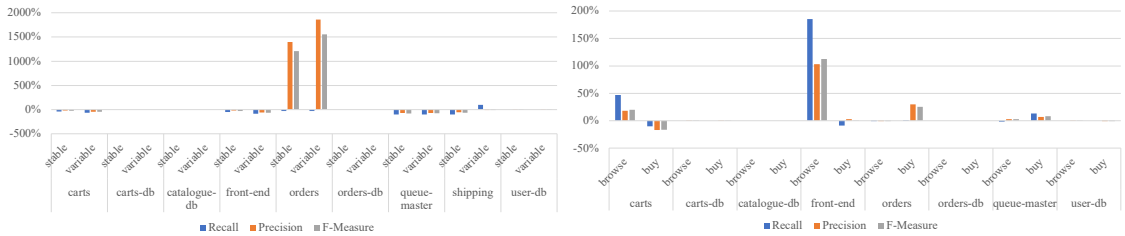


Figure 4.20: Sockshop, percentage difference between workloads, on the left between the buy and the browse profile, and on the right between the variable and the stable intensities.

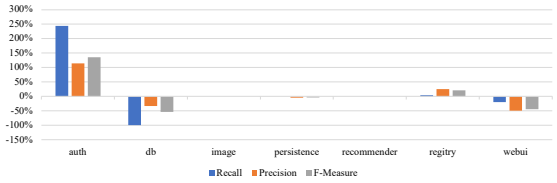


Figure 4.21: Teastore, percentage difference between the buy and the browse workloads

to having a higher number of requests, results in larger datasets; which in turn result in more complete classifiers.

4.4.2 Window Size and algorithms

We will now analyse the impact of the different algorithms and window sizes Figures 4.22and 4.23 shows the metrics for this analysis.

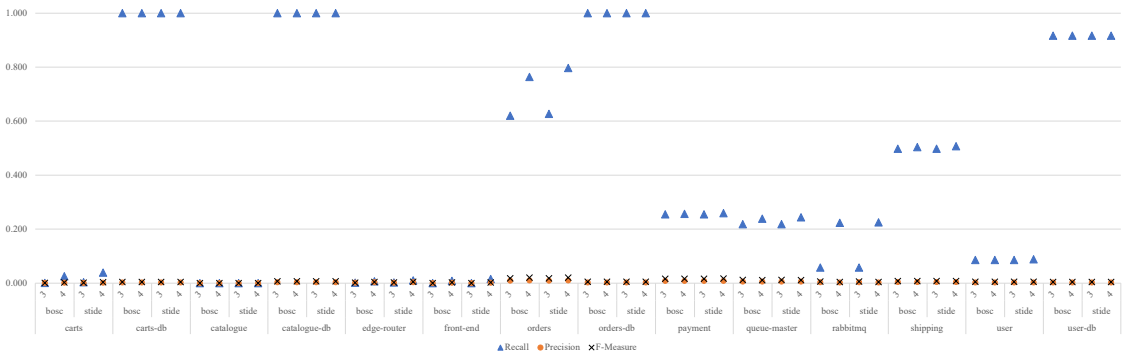


Figure 4.22: Algorithm and window size impact with Sockshop

All classifiers have the same differences, although some are more intense than others; when the window size is four, the recall values are higher, while the precision is lower. Although removal of the loops can be partly responsible, we cannot argue that they make the sole difference, as it is expected that, for the same time period, the classifiers made of windows of size four are less complete than with a size of three. Meanwhile, between algorithms, the changes are not significant.

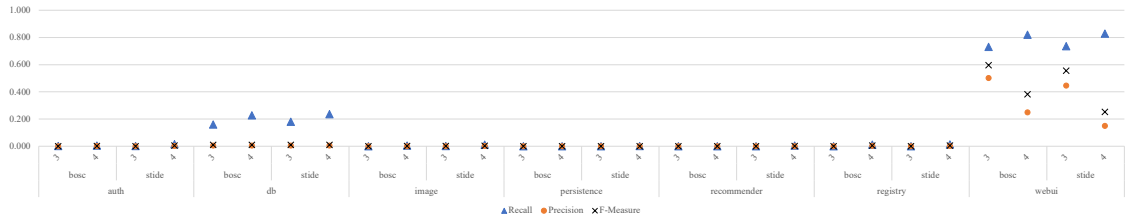


Figure 4.23: Algorithm and window size impact with Teastore

4.4.3 Individual service classifier effectiveness

From now on, we will focus simply on the top 10 best configurations. Figures 4.24 and 4.25 show the results per scenario.

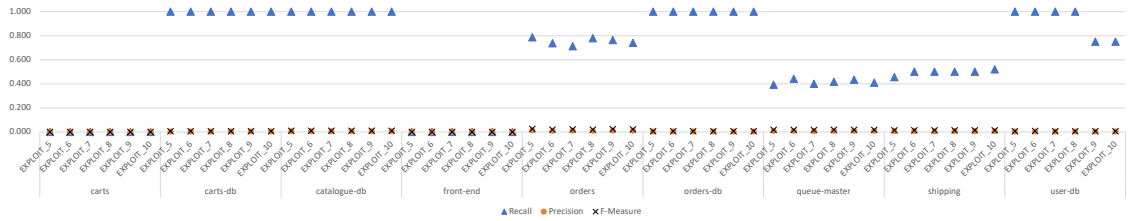


Figure 4.24: Effectiveness of the Sockshop service classifiers per scenario

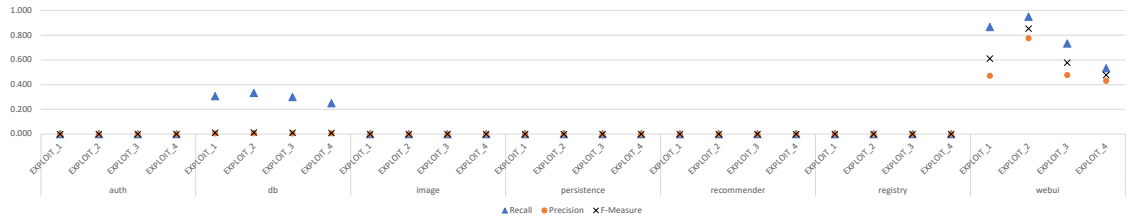


Figure 4.25: Effectiveness of the Teastore service classifiers per scenario

Even after considering the top 10 configurations, the classifiers are still unusable. The only service that can get an F-Measure above `0.450` across all exploits is the webui. However, if we consider only successful attacks, then the F-Measure is greater than `0.600` for the same classifiers of this service. This can be justified by the fact that it is the service with the most data, and, therefore, classifiers are more complete. Sockshop, on the other hand, the higher it can be is `0.019`, which is possible with the classifiers of the orders service.

4.4.4 Decision Making Level Techniques effectiveness

Focusing on decision-making techniques, the thresholds tested were `0.1` for the decision model and 20% for voting. This means that for the voting technique to declare an intrusion, two services have to say that an intrusion is happening, at the same time, for the Teastore testbed; and three services, for the Sockshop testbed. And, considering the decision model, the threshold means that the sum of the ratios between the anomalous windows and the epoch size of the services is higher than `0.1`. The results based on the attack scenarios can be seen in Figure 4.26

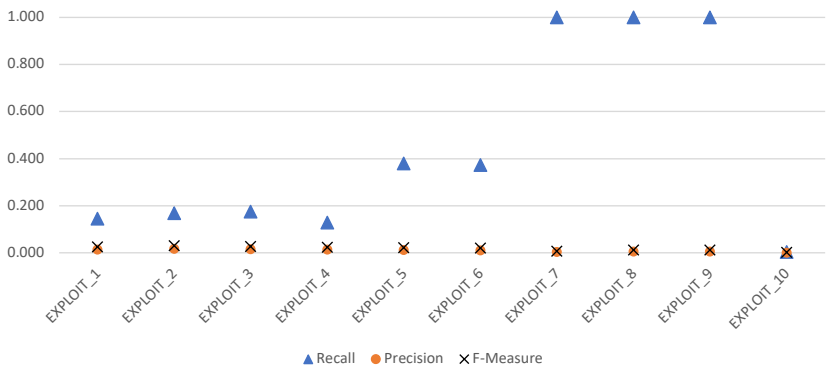


Figure 4.26: Effectiveness of the decision making techniques per exploit

Based on these results, even when only considering the top 10 results, the voting technique is not feasible. The only way we would get a higher recall would be to lower the percentage, but lowering it would mean that the voting technique would become similar to using the classifiers alone. F-Measure is below `0.031` for all exploits, and, even with a high Recall of 1.000 for three exploits, the amount of false positives is too high, bringing the results down.

4.4.5 Feature Level Techniques effectiveness

As noted previously, the matrix technique becomes unfeasible due to scalability issues; however, we still trained some classifiers for the variable workload with the buy profile, using the interleaved dataset. Figure 4.27 shows the results with those classifiers, per attack scenario.

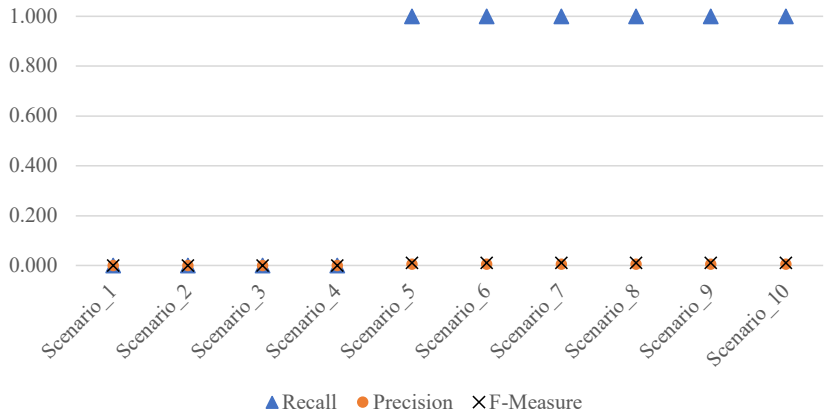


Figure 4.27: Effectiveness of the matrix technique per exploit

Unfortunately, the results are still unsatisfactory, with high recall and low precision, indicating that many alerts are being made.

Finally, we have the interleaving technique in Figure 4.28.

Here, the results are totally different. Considering how the classifiers performed when alone, these results are quite impressive, reaching F-Measure values over

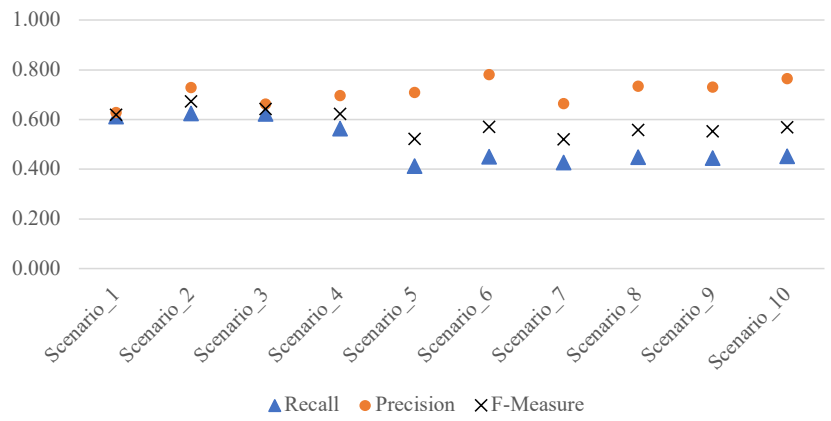


Figure 4.28: Effectiveness of the interleaving technique per exploit

0.500 even for scenarios where the attacks were unsucessfull. Recall values are, on average, 0.606 for TeaStore, while for Sockshop they drop down to 0.440. However, Precision values are way higher, between 0.628 and 0.782, which means a much lower false alarm ratio, with an average of 70% of the alarms being True Positives. Overall, we can say that this technique outperforms all previous analyses, with very good results, being suitable for further studies and enhancements.

4.4.6 Discussion

Based on the results presented, we can safely argue that the loop removal algorithm should be further and improved. Performing the removal of all loops in a single pass through the file and the inclusion of a memory system to allow for a complete construction for classifiers are both improvements that can bring the algorithm to a new level. Even then, it still performs quite well, for the most part not damaging the classifiers and reducing the traces to almost 50%.

Regarding the classifiers, it is possible that more training time would enhance them, especially when working alone or with the decision techniques. For 24 hours of training, the classifiers are, for their majority, useless, with the exception of webui. However, decision techniques could be further researched, possibly with other algorithms, as they allow distributed intrusion detection.

When it comes to feature-level techniques, the matrix is unfeasible. In order for it to be used, improvements must be made, or limit, for example, to a 2-dimensional matrix, and then use a different technique to merge the classifiers.

Finally, the interleaving technique shows promising results. It is the only technique that manages to drastically reduce the amount of false positives, which is really important in an intrusion detection scenario to ensure that the administrator does not waste too much time with false alarms. However, we would like the recall values to be higher because, with the current results, around half of the attacks are not being detected.

Chapter 5

Planning

For the work developed during this thesis we used an incremental model, with each delivery being related to a major milestone of the thesis: technique research and implementation; finding vulnerabilities and respective POCs; development of the loop removal technique; and, finally, orchestration of the experiments. These milestones were mostly decided at the end of the first semester, with the exception of the loop removal algorithm which was an opportunity that appeared in the second semester. However, due to the nature of the thesis, it is hard to follow the project management model exactly, since some steps are unfeaseable. Therefore, some changes to the model were made, in order to better adjust our work.

One of the changes was related to the requirements, where, we decided on a list of research goals and what we needed for a representative scenario. We came together into the following:

- We needed representative attacks, which can happen in the real world, considering a microservice enviornment;
- These attacks had to have some variety, fitting in at least three different categories;
- We should have at least three attack scenarios per testbed;
- We should be using two different testbeds or more;
- The techniques have to be lightweight;
- We need at least three different techniques;

Initially, planning was done considering a delivery in the end of june. Figure 5.1 shows the gantt designed with that goal in mind.

However, some problems appeared, and the opportunity of working on the loop removal algorithm, which made us change our initial plan. We can see the result of the changes on Figure 5.2.

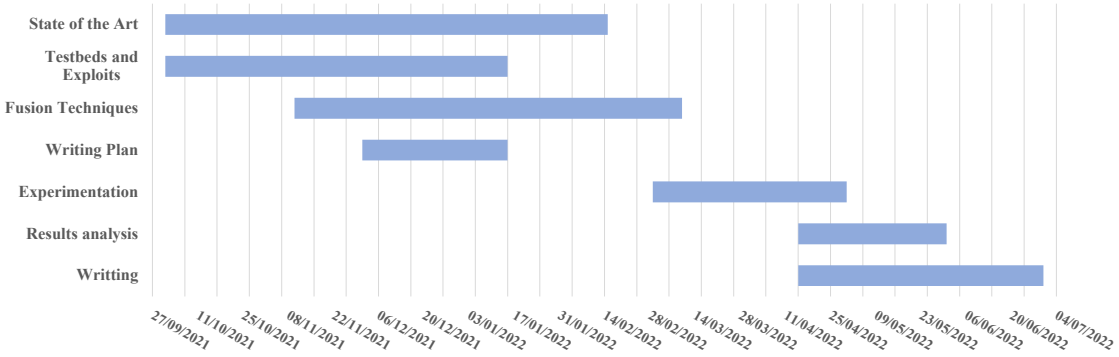


Figure 5.1: Initial Gantt

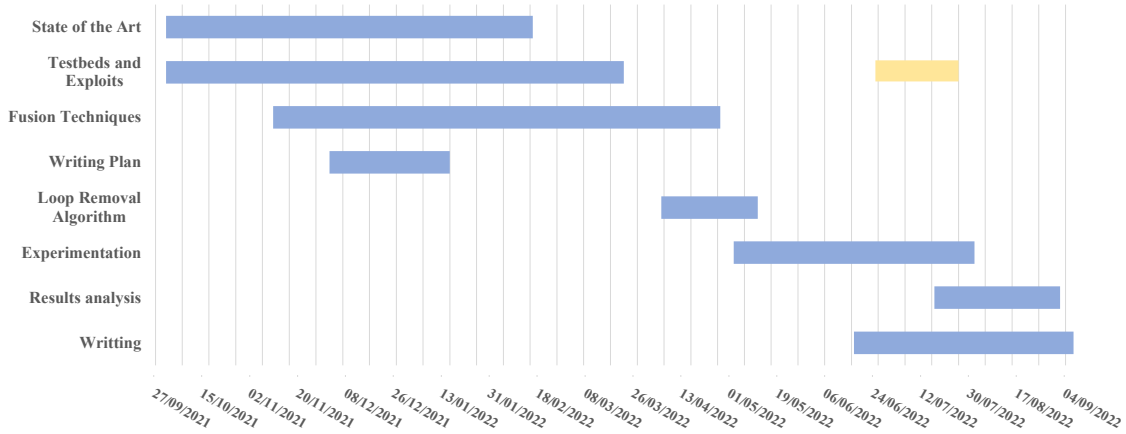


Figure 5.2: Final Gantt

Each milestone is represented as a task in the gantt diagram, being an iteration of the incremental model. Of course, tasks such as the State of the Art and Writing do not have all the steps since they are mostly just a single on-going task, without any validation or testing. In yellow, a continuation of the testbeds and exploits task appears, which was done in order to complement the exploits already gathered, creating more representable experiments.

Now we will go through the risks, the analysis performed along the time, and after we will go through what challenges we faced and how we dealt with them, while also explaining the delays.

5.1 Risk Analysis

Before we start the risk analysis, we need to define the different levels for Impact, Probability and Timeframe. These are as follows:

- **Impact**
  - **High** - Results or the Thesis are compromised even with delays;

  - **Medium** - The Results or the Thesis may present some flaws such as: missing experiments or missing representability. And/or delays are necessary;
  - **Low** - The Thesis may suffer minor changes.

- **Probability**
  - **High** - Higher than 70% chance of happening;
  - **Medium** - Between 70% and 40% chance of happening;
  - **Low** - Lower than 40% chance of happening.

- **Timeframe**
  - **Long** - higher than 4 months;
  - **Medium** - 2 to 4 months;
  - **Short** - 0 to 2 months.

Up to the end of the first semester, we analyzed the top 3 risks that could affect the thesis. The following table is the result of that analysis:

ID	Risk	Impact	Probability	Timeframe	Mitigation Plan
1	Not all attacks have been tested for the testbeds, so it can be tougher than expected and delay the experiments.	Medium	Medium	Short	Study the problem, find alternatives, re-estimate time needed and schedule a meeting/wait for the next meeting in order to determine what to do
2	When the services share similar docker images, the edges of the Attack graphs generated will be similar, thus it is possible that we need to review the source code in order to adapt it.	Low	Medium	Short	Focus on the attacks that we have, and increase the amount of attacks showing per edge
3	Many different combinations will be tested considering the different attacks and processing techniques, so it can take longer than expected to execute the experiments.	Medium	Low	Medium	Reevaluate time needed as time goes by, and in case things start getting too late, we will start doing experiments before everything else is ready

Figure 5.3: Risks identified #1 - 15th December 2021.

Then, risks were being reviewed bi-weekly, and the following three tables summarize the evolution. In the last table, only two relevant risks were identified.

ID	Risk	Impact	Probability	Timeframe	Mitigation Plan
1	The approach used to find the exploits and attacks was changed, which can delay the experiments	Medium	High	Short	Focus on simpler attacks and exploits, and then go for the harder ones
2	Doing a matrix with BoSC and STIDE might be overwhelming in terms of memory	Medium	High	Medium	Try to look for alternatives for the technique
3	Many different combinations will be tested considering the different attacks and processing techniques, so it can take longer than expected to execute the experiments.	Medium	Low	Short	Reevaluate time needed as time goes by, and in case things start getting too late, we will start doing experiments before everything else is ready

Figure 5.4: Risks identified #2 - 1st February 2022.

As we can see during the beginning of the second semester risks got worse, with more aggravated parameters overall. This was the most challenging time of the thesis as delays were starting to appear. These will be analysed in depth in the next section. After this period, the severity of the risks started dropping up until the end of the thesis.

ID	Risk	Impact	Probability	Timeframe	Mitigation Plan
1	There are only 4 different exploits per testbed, with 2 of them for each not working. This can impact the representability of the experiments	Medium	Medium	Medium	Explore other exploits or/and try to get some of them working
2	Doing a matrix with BoSC and STIDE might be overwhelming in terms of memory	Medium	High	Medium	Try to look for alternatives for the technique
3	Many different combinations will be tested considering the different attacks and processing techniques, so it can take longer than expected to execute the experiments.	Low	Low	Short	Focus on the Loop Removal algorithm and then analyse its impact

Figure 5.5: Risks identified #3 - 1st April 2022.

ID	Risk	Impact	Probability	Timeframe	Mitigation Plan
1	Experiments are starting to occupy a lot of space. It is possible that more disk space will be needed	Low	High	Short	Get more space, use space from temporary personal computers as needed, and compress old data that is not currently being used.
2	Analysing the reuslts can take longer than expected due to the large amount of data, which can result in a worse analysis in order to stay within time limitations	Low	Low	Medium	Plan the analysis beforehand

Figure 5.6: Risks identified #4 - 15h June 2022.

5.2 Challenges

We faced various challenges along the development of the thesis, we will now go in depth into each one, and analyse what was the problem and how it was dealt with.

Challenge #1

When we were first analyzing the attacks that we would be performing, we focused on attacks based on the data flows in the system, as well as the communications between the services, and targeted multiple services. So we decided on three different attacks:

- **Sequential attacks on the system**, applied with the help of attack graph generators;
- **Second order attacks**, by adding faulty information to a database that then causes the attack itself;
- **Asymmetrical Workload attacks**, with a set of requests that require large processing power on various services.

However, these **attacks appeared to have problems** with regard to how they would be implemented and detected. The first issue came in regards to the attack graph generators being outdated for the most part or not being helpful in our scenario. Even then, they allowed us to get a list of vulnerabilities to explore, which was extended using snyk. But then exploiting was a problem, since not many POC were able to be found, and those that were did not work.

So we looked at second-order attacks, as these are meaningful and perfectly capable of happening in the real world. We were able to prove the concept on TeaStore, Sockshop and even TrainTicket. However, the focus of the attack was on the user

interface, which is something we are not trying to detect, thus leaving the focus of our work.

Finally, asymmetrical workloads is something that is not much work on, and since it would simply be a generic DoS with a normal workload, it would also be hard to detect with our approach, since these would, in the end, probably resort to just a network congestion.

Therefore, we resorted to focusing on remote attacks against different services, all performed at the same time.

Challenge #2

Finding techniques that would either join the classifiers or the information coming from different services was hard. We had to find techniques that would not only be able to deal with the problems we were facing regarding microservices, such as being lightweight and being able to deal with enormous amounts of data, we also had to check if they made sense and were worth pursuing. Most of the techniques were either aimed at joining information coming from databases, in different formats; or at joining classifiers designed for the same task. These would not work for our study, since classifiers from different services are expected to be different, and the information we are trying to join is already in the same format. For example, two decision making techniques we found were using the highest value of the classifiers, or the lowest. Well, that is too simple for what we are trying to accomplish and we didn't need a "system" to analyze that. Therefore, looking for a technique was harder and longer than expected, delaying the thesis.

Challenge #3

One thing we did not consider during the planning of the experiments was the amount of **memory needed** to run the various classifiers in paralel. Although collecting the data was pretty simillar to what was done before, in terms of analysing the test data with the classifiers, we would need more memory for the same periods of time. That is justified by the fact that, overall, the classifiers are bigger, since they represent different systems, which results in a wider variety of patterns. Therefore, running the classifiers obviously, used more memory than before. This is something that was overlooked which resulted in many runs being aborted and needing to be restarted. To prevent this from becoming a problem, four different machines were used to process the results.

Challenge #4

We knew and were expecting the space required for collecting the data, however, the classifiers and the results took more space than before, which gave us some **space management issues**. This can be explained for two reasons, one was already detailed in the previous challenge, which is the size of the classifiers being bigger; the other is that the testing experiments were three times bigger than the previous ones. Also, the problem was aggravated when we starting using various machines to process the results. In the end, after compressing many unused files, everything worked out great.

Chapter 6

Conclusion

This work proposes an approach to handle host-based intrusion detection with fusion techniques, tackling challenges in microservice systems, and evaluating the techniques proposed.

Work had already been done in regards to the analysis of single service intrusion detection, where I contributed to the analysis of the results and also in regards to the data processing techniques. This analysis allows us to understand how to deal with the scalability of the microservices. Now, we extended and focused on how to deal with various services.

The state of the art was studied to determine which attacks made sense on this architecture, and the testbeds were analysed and chosen. We presented the experimental campaign conducted, taking advantage of the approach and techniques proposed, and evaluate the fusion techniques regarding their detection effectiveness of different attacks. Four different techniques were tested against 10 different attack scenarios.

We also evaluated a loop removal algorithm with the goal of reducing the amount of data without affecting the classifiers. With this algorithm, we were able to reduce the training tests to 55% of their original size, with almost no cost to the effectiveness of the classifiers.

In the end, applying the classifiers to each service does not look like a good tactic as they seem to be unable to detect attacks alone. Research can be extended to apply the same classifiers with bigger training times, so to increase the volume of data per service, allowing for better profiles. Also, decision making techniques do not show good results, which can be related to the point stated above. The matrix technique is unscalable, and some adjustments are needed, while the interleaving technique shows promising results, obtaining the highest F-Measure, going up to `0.673` with a Precision of up to`0.782`, however, there is still room for improvement.

This page is intentionally left blank.

References

L. Abdollahi Vayghan, M. A. Saied, M. Toeroe, and F. Khendek. Microservice based architecture: Towards high-availability for stateful applications with kubernetes. In *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*, pages 176–185, 2019.

Amr S Abed, Charles Clancy, and David S Levy. Intrusion detection system for applications using linux containers. In *International Workshop on Security and Trust Management*, pages 123–135. Springer, 2015a.

Amr S Abed, T Charles Clancy, and David S Levy. Applying bag of system calls for anomalous behavior detection of applications in linux containers. In *2015 IEEE Globecom Workshops (GC Wkshps)*, pages 1–5. IEEE, 2015b.

Pekka Abrahamsson, Outi Salo, Jussi Ronkainen, and Juhani Warsta. Agile software development methods: Review and analysis. *arXiv preprint arXiv:1709.08439*, 2017.

agix. mongod_native_helper. `https://www.infosecmatter.com/metasploit-module-library/?mm=exploit/linux/misc/mongod_native_helper`, 2020. [Online] Accessed: 2022-06-23.

Isam Mashhour Al Jawarneh, Paolo Bellavista, Filippo Bosi, Luca Foschini, Giuseppe Martuscelli, Rebecca Montanari, and Amedeo Palopoli. Container orchestration engines: A thorough functional and performance comparison. In *ICC 2019-2019 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE, 2019.

Mohammad Hadi Alaeiyan and Saeed Parsa. Automatic loop detection in the sequence of system calls. In *2015 2nd International Conference on Knowledge-Based Engineering and Innovation (KBEI)*, pages 720–723. IEEE, 2015.

Hussain AlJahdali, Abdulaziz Albatli, Peter Garraghan, Paul Townend, Lydia Lau, and Jie Xu. Multi-tenancy in cloud computing. In *2014 IEEE 8th international symposium on service oriented system engineering*, pages 344–351. IEEE, 2014.

Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. Understanding the mirai botnet. In *26th {USENIX} security symposium ({USENIX} Security 17)*, pages 1093–1110, 2017.

Stefan Axelsson. Intrusion detection systems: A survey and taxonomy. Technical report, Citeseer, 2000.

Rebecca Bace and Peter Mell. Nist special publication on intrusion detection systems. Technical report, BOOZ-ALLEN AND HAMILTON INC MCLEAN VA, 2001.

Marie Baezner and Patrice Robin. Stuxnet. Technical report, ETH Zurich, 2017.

Salah Eddine Benaicha, Lalia Saoudi, Salah Eddine Bouhouita Guermeche, and Ouarda Lounis. Intrusion detection system using genetic algorithm. In *2014 Science and Information Conference*, pages 564–568. IEEE, 2014.

David Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.

Saikat Biswas, M. Sohel, Md.Mizanur Sajal, Tanjina Afrin, Touhid Bhuiyan, and Md Maruf Hassan. A study on remote code execution vulnerability in web applications. 10 2018.

BloodHound. Bloodhound docs. `https://bloodhound.readthedocs.io/en/latest/`, 2022a. [Online] Accessed: 2022-06-20.

BloodHound. Bloodhound related work. `https://bloodhound.readthedocs.io/en/latest/further-reading/further-reading.html`, 2022b. [Online] Accessed: 2022-06-20.

BloodHoundAD. Attack graph generation with bloodhound. `https://github.com/BloodHoundAD/BloodHound`, 2022. [Online] Accessed: 2022-06-20.

Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. Towards taming privilege-escalation attacks on android. In *NDSS*, volume 17, page 19, 2012.

Thanh Bui. Analysis of docker security. *arXiv preprint arXiv:1501.02967*, 2015.

Brendan Burns, Joe Beda, and Kelsey Hightower. *Kubernetes*. Dpunkt, 2018.

Vittorio Castelli, Richard E Harper, Philip Heidelberger, Steven W Hunter, Kishor S Trivedi, Kalyanaraman Vaidyanathan, and William P Zeggert. Proactive management of software aging. *IBM Journal of Research and Development*, 45(2):311–332, 2001.

Genevieve CY Chan, Ravi Kamble, Henning Müller, Syed AA Shah, Tong Boon Tang, and Fabrice Mériaudeau. Fusing results of several deep learning architectures for automatic classification of normal and diabetic macular edema in optical coherence tomography. In *2018 40th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, pages 670–673. IEEE, 2018.

Hongda Chen, Genshe Chen, Erik Blasch, Martin Kruger, and Irma Sityar. Analysis and visualization of large complex attack graphs for networks security. In *Data Mining, Intrusion Detection, Information Assurance, and Data Networks*

Security 2007, volume 6570, page 657004. International Society for Optics and Photonics, 2007.

Shichao Chen and Mengchu Zhou. Evolving container to unikernel for edge computing and applications in process industry. *Processes*, 9(2):351, 2021.

Marcello Cinque, Raffaele Della Corte, and Antonio Pecchia. Microservices monitoring with event logs and black box execution tracing. *IEEE Transactions on Services Computing*, 2019.

David Cohen, Mikael Lindvall, and Patricia Costa. Agile software development. *DACS SOAR Report*, 11:2003, 2003.

cyberImperial. Mulval based attack graph generator github. `https://github.com/cyberImperial/attack-graphs`, 2019. [Online] Accessed: 2022-06-20.

DataDog. Docker adoption facts. `https://www.datadoghq.com/docker-adoption/`, 2018. [Online] Accessed: 2022-01-21.

Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on android. In *international conference on Information security*, pages 346–360. Springer, 2010.

Ozgur Depren, Murat Topallar, Emin Anarim, and M Kemal Ciliz. An intelligent intrusion detection system (ids) for anomaly and misuse detection in computer networks. *Expert systems with Applications*, 29(4):713–722, 2005.

Maurice Dibbets, Erik Poll, Hugo Jonker, and Ralph Moonen. Discovery of information disclosure vulnerabilities in the software development process. 2021.

Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow. *Present and ulterior software engineering*, pages 195–216, 2017.

Tore Dyba and Torgeir Dingsoyr. What do we know about agile software development? *IEEE software*, 26(5):6–9, 2009.

Christof Ebert, Gorka Gallardo, Josune Hernantes, and Nicolas Serrano. Devops. *Ieee Software*, 33(3):94–100, 2016.

Sinan Eski and Feza Buzluca. An automatic extraction approach: Transition to microservices architecture from monolithic application. In *Proceedings of the 19th International Conference on Agile Software Development: Companion*, pages 1–6, 2018.

José Flora and Nuno Antunes. Studying the applicability of intrusion detection to multi-tenant container environments. In *2019 15th European Dependable Computing Conference (EDCC)*, pages 133–136. IEEE, 2019.

José Flora, Paulo Gonçalves, and Nuno Antunes. Using attack injection to evaluate intrusion detection effectiveness in container-based systems. In *2020 IEEE 25th Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 60–69. IEEE, 2020.

José Flora, Paulo Gonçalves, Miguel Teixeira, and Nuno Antunes. My Services Got Old! Can Kubernetes Handle the Aging of Microservices? In *2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 40–47, 2021. doi: DOI10.1109/ISSREW53611.2021.00042.

Pedro Garcia-Teodoro, Jesus Diaz-Verdejo, Gabriel Maciá-Fernández, and Enrique Vázquez. Anomaly-based network intrusion detection: Techniques, systems and challenges. *computers & security*, 28(1-2):18–28, 2009.

Akash Garg and Prachi Maheshwari. Performance analysis of snort-based intrusion detection system. In *2016 3rd International Conference on Advanced Computing and Communication Systems (ICACCS)*, volume 1, pages 1–5. IEEE, 2016.

Sachin Garg, Aad Van Moorsel, Kalyanaraman Vaidyanathan, and Kishor S Trivedi. A methodology for detection and estimation of software aging. In *Proceedings Ninth International Symposium on Software Reliability Engineering (Cat. No. 98TB100257)*, pages 283–292. IEEE, 1998.

Jason Gionta, William Enck, and Peng Ning. Hidem: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pages 325–336, 2015.

Michael Godfrey and Mohammad Zulkernine. Preventing cache-based side-channel attacks in a cloud environment. *IEEE transactions on cloud computing*, 2 (4):395–408, 2014.

Paulo Gonçalves. Attack graph generation for microservice architecture. `https://github.com/pgoncalvesgit/attack-graph-generator`, 2022. [Online] Accessed: 2022-06-20.

M. Grottke, L. Li, K. Vaidyanathan, and K.S. Trivedi. Analysis of software aging in a web server. *IEEE Transactions on Reliability*, 55(3):411–420, 2006.

Michael Grottke and Kishor S Trivedi. A classification of software faults. *Journal of Reliability Engineering Association of Japan*, 27(7):425–438, 2005.

Michael Grottke, Rivalino Matias, and Kishor S Trivedi. The fundamentals of software aging. In *2008 IEEE International conference on software reliability engineering workshops (ISSRE Wksp)*, pages 1–6. Ieee, 2008a.

Michael Grottke, Rivalino Matias, and Kishor S. Trivedi. The fundamentals of software aging. In *2008 IEEE International Conference on Software Reliability Engineering Workshops (ISSRE Wksp)*, pages 1–6, 2008b.

Zhe Guo, Xiang Li, Heng Huang, Ning Guo, and Quanzheng Li. Deep learning-based image segmentation on multimodal medical imaging. *IEEE Transactions on Radiation and Plasma Medical Sciences*, 3(2):162–169, 2019a.

Zhe Guo, Xiang Li, Heng Huang, Ning Guo, and Quanzheng Li. Deep learning-based image segmentation on multimodal medical imaging. *IEEE Transactions on Radiation and Plasma Medical Sciences*, 3(2):162–169, 2019b. doi: 10.1109/TRPMS.2018.2890359.

Manasi Gyanchandani, JL Rana, and RN Yadav. Taxonomy of anomaly based intrusion detection system: a review. *International Journal of Scientific and Research Publications*, 2(12):1–13, 2012.

Vajiheh Hajisalem and Shahram Babaie. A hybrid intrusion detection system based on abc-afs algorithm for misuse and anomaly detection. *Computer Networks*, 136:37–50, 2018.

Yennun Huang, Chandra Kintala, Nick Kolettis, and N Dudley Fulton. Software rejuvenation: Analysis, module and applications. In *Twenty-fifth international symposium on fault-tolerant computing. Digest of papers*, pages 381–390. IEEE, 1995.

Amjad Ibrahim, Stevica Bozhinoski, and Alexander Pretschner. Attack graph generation for microservice architecture. In *Proceedings of the 34th ACM/SIGAPP symposium on applied computing*, pages 1235–1242, 2019a.

Amjad Ibrahim, Stevica Bozhinoski, and Alexander Pretschner. Attack graph generation for microservice architecture. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, SAC '19, page 1235–1242, New York, NY, USA, 2019b. Association for Computing Machinery. ISBN 9781450359337. doi: 10.1145/3297280.3297401. URL https://doi.org/10.1145/3297280.3297401.

Sadeeq Jan, Cu D Nguyen, and Lionel Briand. Known xml vulnerabilities are still a threat to popular parsers and open source systems. In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 233–241. IEEE, 2015.

Somesh Jha, Oleg Sheyner, and Jeannette Wing. Two formal analyses of attack graphs. In *Proceedings 15th IEEE Computer Security Foundations Workshop. CSFW-15*, pages 49–63. IEEE, 2002a.

Somesh Jha, Oleg Sheyner, and Jeannette M Wing. Minimization and reliability analyses of attack graphs. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE, 2002b.

Zhiqiang Jian and Long Chen. A defense method against docker escape attack. In *Proceedings of the 2017 International Conference on Cryptography, Security and Privacy*, pages 142–146, 2017.

Anita K Jones and Robert S Sielken. Computer system intrusion detection: A survey. *Computer Science Technical Report*, pages 1–25, 2000.

VVRPV Jyothsna, Rama Prasad, and K Munivara Prasad. A review of anomaly based intrusion detection systems. *International Journal of Computer Applications*, 28(7):26–35, 2011.

Narūnas Kapočius. Overview of kubernetes cni plugins performance. *Mokslas–Lietuvos ateitis/Science–Future of Lithuania*, 12, 2020.

Richard A Kemmerer. Nstat: a model-based real-time network intrusion detection system. *Computer Science Department, University of California, Santa Barbara, Report TRCS97-18, http://www. cs. ucsb. edu/TRs/TRCS97-18. html*, 1997.

Abeer Abdel Khaleq and Ilkyeun Ra. Agnostic approach for microservices autoscaling in cloud applications. In *Int. Conf. on Computational Science and Computational Intelligence (CSCI)*, pages 1411–1415, 2019.

Amjad Khalifah. Attack graph generation for microservice architecture. `https://github.com/tum-i4/attack-graph-generator`, 2018. [Online] Accessed: 2022-06-20.

kris katterjohn. mysql_version. `https://www.infosecmatter.com/metasploit-module-library/?mm=auxiliary/scanner/mysql/mysql_version`, 2017. [Online] Accessed: 2022-06-23.

Kubernetes. Configure liveness, readiness and startup probes. kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes, 2020.

Rajeev Kumar, Suhel Ahmad Khan, and Raees Ahmad Khan. Revisiting software security: durability perspective. *International Journal of Hybrid Information Technology*, 8(2):311–322, 2015.

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.

Ludmila I Kuncheva, James C Bezdek, and Robert PW Duin. Decision templates for multiple classifier fusion: an experimental comparison. *Pattern recognition*, 34(2):299–314, 2001.

Rodrigo Laigner, Yongluan Zhou, Marcos Antonio Vaz Salles, Yijian Liu, and Marcos Kalinowski. Data management in microservices: State of the practice, challenges, and research directions. *arXiv preprint arXiv:2103.00170*, 2021.

Xabier Larrucea, Izaskun Santamaria, Ricardo Colomo-Palacios, and Christof Ebert. Microservices. *IEEE Software*, 35(3):96–100, 2018.

Seunghyung Lee, Seokho Son, Jungsu Han, and JongWon Kim. Refining micro services placement over multiple kubernetes-orchestrated clusters employing resource monitoring. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 1328–1332. IEEE, 2020.

Hongda Li, Hongxin Hu, Guofei Gu, Gail-Joon Ahn, and Fuqiang Zhang. vnids: Towards elastic security with safe and efficient virtualization of network intrusion detection systems. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 17–34, 2018.

Lei Li, K. Vaidyanathan, and K.S. Trivedi. An approach for estimation of software aging in a web server. In *Proceedings International Symposium on Empirical Software Engineering*, pages 91–100, 2002.

Shanshan Li, He Zhang, Zijia Jia, Chenxing Zhong, Cheng Zhang, Zhihao Shan, Jinfeng Shen, and Muhammad Ali Babar. Understanding and addressing quality attributes of microservices architecture: A systematic literature review. *Information and Software Technology*, 131:106449, 2021.

Xufang Li, Peter KK Loh, and Freddy Tan. Mechanisms of polymorphic and metamorphic viruses. In *2011 European intelligence and security informatics conference*, pages 149–154. IEEE, 2011.

Wei Liang, Yiyong Hu, Xiaokang Zhou, Yi Pan, I Kevin, and Kai Wang. Variational few-shot learning for microservice-oriented intrusion detection in distributed industrial iot. *IEEE Transactions on Industrial Informatics*, 2021.

Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *2016 IEEE international symposium on high performance computer architecture (HPCA)*, pages 406–418. IEEE, 2016.

Miao Liu and Bin Wang. A web second-order vulnerabilities detection method. *IEEE Access*, 6:70983–70988, 2018.

Basel Magableh and Muder Almiani. A Self Healing Microservices Architecture: A Case Study in Docker Swarm Cluster. In *Advanced Information Networking and Applications*, Advances in Intelligent Systems and Computing, pages 846–858. Springer International Publishing, 2020.

K Narasimha Mallikarjunan, K Muthupriya, and S Mercy Shalinie. A survey of distributed denial of service attack. In *2016 10th International Conference on Intelligent Systems and Control (ISCO)*, pages 1–6. IEEE, 2016.

Gregory Man. mongodb_login. `https://www.infosecmatter.com/metasploit-module-library/?mm=auxiliary/scanner/mongodb/mongodb_login`, 2019. [Online] Accessed: 2022-06-23.

Georgios Mantas, Natalia Stakhanova, Hugo Gonzalez, Hossein Hadian Jazi, and Ali A Ghorbani. Application-layer denial of service attacks: taxonomy and survey. *International Journal of Information and Computer Security*, 7(2-4):216–239, 2015.

joev Marek Majkowski, titanous. nodejs_pipelining. `https://www.infosecmatter.com/metasploit-module-library/?mm=auxiliary/dos/http/nodejs_pipelining`, 2020. [Online] Accessed: 2022-06-23.

Nuno Mateus-Coelho, Manuela Cruz-Cunha, and Luis Gonzaga Ferreira. Security in microservices architectures. *Procedia Computer Science*, 181:1225–1236, 2021.

MC. mysql_yassl_hello. `https://www.infosecmatter.com/metasploit-module-library/?mm=exploit/linux/mysql/mysql_yassl_hello`, 2020. [Online] Accessed: 2022-06-23.

jduck MC, Matteo Cantoni. tomcat_mgr_login. `https://www.infosecmatter.com/metasploit-module-library/?mm=auxiliary/scanner/http/tomcat_mgr_login`, 2021. [Online] Accessed: 2022-06-23.

John McHugh. Intrusion and intrusion detection. *International Journal of Information Security*, 1(1):14–35, 2001.

Dirk Merkel et al. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

Aleksandar Milenkoski, Bryan D. Payne, Nuno Antunes, Marco Vieira, Samuel Kounev, Alberto Avritzer, and Matthias Luft. Evaluation of intrusion detection systems in virtualized environments using attack injection. In *International Symposium on Recent Advances in Intrusion Detection*. Springer, 2015.

mpgn. Spring boot exploit. `https://github.com/mpgn/Spring-Boot-Actuator-Exploit`, 2018. [Online] Accessed: 2022-06-23.

John C. Munson, Allen P. Nikora, and Joseph S. Sherif. Software faults: A quantifiable definition. *Advances in Engineering Software*, 37(5):327–333, 2006.

Sam Newman. *Building microservices*. " O'Reilly Media, Inc.", 2021.

nvd. Cve-2008-0226. `https://nvd.nist.gov/vuln/detail/CVE-2008-0226`, 2008. [Online] Accessed: 2022-06-23.

nvd. Cve-2010-2227. `https://nvd.nist.gov/vuln/detail/CVE-2010-2227`, 2010. [Online] Accessed: 2022-06-23.

nvd. Cve-2012-2122. `https://nvd.nist.gov/vuln/detail/CVE-2012-2122`, 2012. [Online] Accessed: 2022-06-23.

nvd. Cve-2013-4450. `https://nvd.nist.gov/vuln/detail/CVE-2013-4450`, 2013a. [Online] Accessed: 2022-06-23.

nvd. Cve-2013-1892. `https://nvd.nist.gov/vuln/detail/CVE-2013-1892`, 2013b. [Online] Accessed: 2022-06-23.

nvd. Cve-2020-1938. `https://nvd.nist.gov/vuln/detail/CVE-2020-1938`, 2020. [Online] Accessed: 2022-06-23.

Felipe Oliveira, Jean Araujo, Rubens Matos, Luan Lins, André Rodrigues, and Paulo Maciel. Experimental evaluation of software aging effects in a container-based virtualization platform. In *2020 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 414–419, 2020.

Xinming Ou, Sudhakar Govindavajhala, and Andrew W. Appel. MulVAL: A logic-based network security analyzer. In *14th USENIX Security Symposium (USENIX Security 05)*, Baltimore, MD, July 2005. USENIX Association. URL `https://www.usenix.org/conference/14th-usenix-security-symposium/mulval-logic-based-network-security-analyzer`.

Xinming Ou, Wayne F. Boyer, and Miles A. McQueen. A scalable approach to attack graph generation. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, page 336–345, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595935185. doi: 10.1145/1180405.1180446. URL `https://doi.org/10.1145/1180405.1180446`.

Claus Pahl and Pooyan Jamshidi. Microservices: A systematic mapping study. In *CLOSER (1)*, pages 137–146, 2016.

Ioannis Papapanagiotou and Vinay Chella. Ndbench: Benchmarking microservices at scale. *arXiv preprint arXiv:1807.10792*, 2018.

David Lorge Parnas. Software aging. In *Proceedings of 16th International Conference on Software Engineering*, pages 279–287. IEEE, 1994.

Cesare Pautasso, Olaf Zimmermann, Mike Amundsen, James Lewis, and Nicolai Josuttis. Microservices in practice, part 2: Service integration and sustainability. *IEEE Software*, 34(02):97–104, 2017.

PetClinic. Spring petclinic sample application build status. `https://github.com/spring-projects/spring-petclinic`, 2022. [Online] Accessed: 2022-01-18.

Chen Ping. A second-order sql injection detection method. In *2017 IEEE 2nd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*, pages 1792–1796. IEEE, 2017.

Guy Podjarny. Snyk journey. `https://snyk.io/blog/our-journey-to-today/`, 2021. [Online] Accessed: 2022-06-20.

Robi Polikar. Ensemble based systems in decision making. *IEEE Circuits and systems magazine*, 6(3):21–45, 2006.

Babak Bashari Rad, Harrison John Bhatti, and Mohammad Ahmadi. An introduction to docker and analysis of its performance. *International Journal of Computer Science and Network Security (IJCSNS)*, 17(3):228, 2017.

rangercha. tomcat_mgr_upload. `https://www.infosecmatter.com/metasploit-module-library/?mm=exploit/multi/http/tomcat_mgr_upload`, 2021. [Online] Accessed: 2022-06-23.

Mohammed Rangwala, Ping Zhang, Xukai Zou, and Feng Li. A taxonomy of privilege escalation attacks in android applications. *International Journal of Security and Networks*, 9:40–55, 02 2014. doi: 10.1504/IJSN.2014.059327.

Donald Ray and Jay Ligatti. Defining code-injection attacks. *Acm Sigplan Notices*, 47(1):179–190, 2012.

Chris Richardson. Microservice architecture. `https://microservices.io/`, 2022. [Online] Accessed: 2022-01-16.

Martin Max Röhling, Martin Grimmer, Dennis Kreubel, Jorn Hoffmann, and Bogdan Franczyk. Standardized container virtualization approach for collecting host intrusion detection data. In *2019 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 459–463. IEEE, 2019.

Christian Romano. Mulval github. `https://github.com/risksense/mulval`, 2015. [Online] Accessed: 2022-06-20.

Fabiana Rossi, Valeria Cardellini, and Francesco Lo Presti. Hierarchical scaling of microservices in kubernetes. In *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 28–37. IEEE, 2020.

Robin Ruefle, Audrey Dorofee, David Mundie, Allen D Householder, Michael Murray, and Samuel J Perl. Computer security incident response team development and evolution. *IEEE Security & Privacy*, 12(5):16–26, 2014.

Dymitr Ruta and Bogdan Gabrys. An overview of classifier fusion methods. *Computing and Information systems*, 7(1):1–10, 2000.

Fabrice J Ryba, Matthew Orlinski, Matthias Wählisch, Christian Rossow, and Thomas C Schmidt. Amplification and drdos attack defense–a survey and new perspectives. *arXiv preprint arXiv:1505.07892*, 2015.

Aravind Samy Shanmugam. *Docker container reactive scalability and prediction of cpu utilization based on proactive modelling*. PhD thesis, Dublin, National College of Ireland, 2017.

Ashu Sharma and Sanjay Kumar Sahay. Evolution and detection of polymorphic and metamorphic malwares: A survey. *arXiv preprint arXiv:1406.7061*, 2014.

Oleg Sheyner and Jeannette Wing. Tools for generating and analyzing attack graphs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, pages 344–371, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

Weisong Shi and Schahram Dustdar. The promise of edge computing. *Computer*, 49(5):78–81, 2016.

Steven R Snapp, Stephen E Smaha, Daniel M Teal, and Tim Grance. The {DIDS}(distributed intrusion detection system) prototype. In *USENIX Summer 1992 Technical Conference (USENIX Summer 1992 Technical Conference)*, 1992.

Snort. What is snort? `https://www.snort.org`, 2022. [Online] Accessed: 2022-01-16.

Sockshop. Sockshop arquitecture. `https://github.com/microservices-demo/microservices-demo.github.io/raw/HEAD/assets/Architecture.png`, 2017. [Online] Accessed: 2022-01-18.

Sockshop. Sock shop : A microservice demo application. `https://github.com/microservices-demo/microservices-demo`, 2021. [Online] Accessed: 2022-01-18.

Stephen Specht and Ruby Lee. Taxonomies of distributed denial of service networks, attacks, tools and countermeasures. *CEL2003-03, Princeton University, Princeton, NJ, USA*, 2003.

Martin Štefanko, Ondrej Chaloupka, Bruno Rossi, M van Sinderen, and L Maciaszek. The saga pattern in a reactive microservices environment. In *Proc. 14th Int. Conf. Softw. Technologies (ICSOFT 2019)*, pages 483–490. SciTePress Prague, Czech Republic, 2019.

Michael Stepankin. Exploiting spring boot actuators. `https://www.veracode.com/blog/research/exploiting-spring-boot-actuators`, 2019. [Online] Accessed: 2022-06-23.

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

L.P. Swiler, C. Phillips, D. Ellis, and S. Chakerian. Computer-attack graph generation tool. In *Proceedings DARPA Information Survivability Conference and Exposition II. DISCEX'01*, volume 2, pages 307–321 vol.2, 2001. doi: 10.1109/DISCEX.2001.932182.

Sysdig, Inc. sysdig. `https://sysdig.com/`, 2019. URL `https://sysdig.com/`. [Accessed: 2019-01-24].

jcran theLightCosine. mysql_authbypass_hashdump. `https://www.infosecmatter.com/metasploit-module-library/?mm=auxiliary/scanner/mysql/mysql_authbypass_hashdump`, 2021. [Online] Accessed: 2022-06-23.

Johannes Thönes. Microservices. *IEEE software*, 32(1):116–116, 2015.

Matheus Torquato and Marco Vieira. An experimental study of software aging and rejuvenation in dockerd. In *2019 15th European Dependable Computing Conference (EDCC)*, pages 1–6. IEEE, 2019.

Train-Ticket. Train ticket: A benchmark microservice system. `https://github.com/FudanSELab/train-ticket`, 2022. [Online] Accessed: 2022-01-18.

Kishor S Trivedi, Kalyanaraman Vaidyanathan, and Katerina Goseva-Popstojanova. Modeling and analysis of software aging and rejuvenation. In *Proceedings 33rd annual simulation symposium (SS 2000)*, pages 270–279. IEEE, 2000.

Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. Cryptanalysis of des implemented on computers with cache. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 62–76. Springer, 2003.

Andreas Uhl and Peter Wild. Parallel versus serial classifier combination for multibiometric hand-based identification. In *International Conference on Biometrics*, pages 950–959. Springer, 2009.

Marco Vieira and Nuno Antunes. Introduction to software security concepts. In *Innovative Technologies for Dependable OTS-Based Critical Systems*, pages 29–38. Springer, 2013.

Jóakim von Kistowski, Simon Eismann, Norbert Schmitt, André Bauer, Johannes Grohmann, and Samuel Kounev. Teastore: A micro-service reference application for benchmarking, modeling and resource management research. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 223–236. IEEE, 2018.

Chris Wallis. How to perform a vulnerability assessment: A step-by-step guide. `https://www.intruder.io/guides/`

vulnerability-assessment-made-simple-a-step-by-step-guide, 2022. [Online] Accessed: 2022-06-23.

Luping Wang, Qizhen Weng, Wei Wang, Chen Chen, and Bo Li. Metis: learning to schedule long-running applications in shared container clusters at scale. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–17. IEEE, 2020.

Wei Wang, Xiao-Hong Guan, and Xiang-Liang Zhang. Modeling program behaviors by hidden markov models for intrusion detection. In *Proceedings of 2004 International Conference on Machine Learning and Cybernetics (IEEE Cat. No. 04EX826)*, volume 5, pages 2830–2835. IEEE, 2004.

Yan Wang, Wei Wu, Chao Zhang, Xinyu Xing, Xiaorui Gong, and Wei Zou. From proof-of-concept to exploitable. *Cybersecurity*, 2(1):12, March 2019. ISSN 2523-3246. doi: 10.1186/s42400-019-0028-9. URL https://doi.org/10.1186/s42400-019-0028-9.

Qilin Xiang, Xin Peng, Chuan He, Hanzhang Wang, Tao Xie, Dewei Liu, Gang Zhang, and Yuanfang Cai. No free lunch: Microservice practices reconsidered in industry. *arXiv preprint arXiv:2106.07321*, 2021.

Dr. Su Zhang Xinming (Simon) Ou, Wayne F. Boyer. Mulval: A logic-based enterprise network security analyzer. http://www.arguslab.org/software/mulval.html, 2012. [Online] Accessed: 2022-06-20.

Ying Xiong, Yulin Sun, Li Xing, and Ying Huang. Extend cloud to edge with kubeedge. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 373–377. IEEE, 2018.

Xiaojian Xu, Zhuangzhuang Zhao, Xiaobin Xu, Jianbo Yang, Leilei Chang, Xinping Yan, and Guodong Wang. Machine learning-based wear fault diagnosis for marine diesel engine by fusing multiple data-driven models. *Knowledge-Based Systems*, 190:105324, 2020.

Tetiana Yarygina. *Exploring Microservice Security*. PhD. Thesis, University of Bergen, Norway, July 2018.

Tetiana Yarygina and Anya Helene Bagge. Overcoming security challenges in microservice architectures. In *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, pages 11–20. IEEE, 2018.

Ilsun You and Kangbin Yim. Malware obfuscation techniques: A brief survey. In *2010 International conference on broadband, wireless computing, communication and applications*, pages 297–300. IEEE, 2010.

Jing Yue, Xiaojun Wu, and Yunqing Xue. Microservice aging and rejuvenation. In *2020 World Conference on Computing and Communication Technologies (WCCCT)*, pages 1–5. IEEE, 2020a.

Jing Yue, Xiaojun Wu, and Yunqing Xue. Microservice aging and rejuvenation. In *2020 World Conference on Computing and Communication Technologies (WCCCT)*, pages 1–5, 2020b.

Nuyun Zhang, Hongda Li, Hongxin Hu, and Younghee Park. Towards effective virtualization of intrusion detection systems. In *Proceedings of the ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*, pages 47–50, 2017.

Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering*, 2018.

Olaf Zimmermann. Microservices tenets. *Computer Science-Research and Development*, 32(3):301–310, 2017.

Tommaso Zoppi, Mohamad Gharib, Muhammad Atif, and Andrea Bondavalli. Meta-learning to improve unsupervised intrusion detection in cyber-physical systems. *ACM Transactions on Cyber-Physical Systems (TCPS)*, 5(4):1–27, 2021.