The Journal of Systems and Software 216 (2024) 112142

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Evaluating intrusion detection for microservice applications: Benchmark, dataset, and case studies☆

José Flora *, Nuno Antunes

CISUC, Department of Informatics Engineering, University of Coimbra, Portugal

ARTICLE INFO

Dataset link: https://doi.org/10.5281/zenodo.10655126

Keywords:
Microservices
Security benchmarking
Datasets
Intrusion detection
Attack injection

ABSTRACT

Microservices are predominant for cloud-based applications, which serve millions of customers daily, that commonly run business-critical systems on software containers and multi-tenant environments; so, it is of utmost importance to secure these systems. Intrusion detection is a widely applied technique that is now being used in microservices to build behavior detection models and report possible attacks during runtime. However, it is cumbersome to evaluate and compare the effectiveness of different approaches. Standardized frameworks are non-existent and without fairly comparing new techniques to the state-of-the-art, it is difficult to understand their pros and cons. This paper presents a comprehensive approach to evaluate and compare different intrusion detection approaches for microservice applications. A benchmarking methodology is proposed to allow users to standardize the process for a representative and reproducible evaluation. We also present a dataset that applies representative workloads and technologies based on microservice applications state-of-the-art. The benchmark and dataset are used in three case studies, characterized by dynamicity, scalability, and continuous delivery, to evaluate and compare state-of-the-art algorithms with the objective of tackling intrusion detection in microservices. Experiments show the usefulness and wide application range of the benchmark while showing the capacity of intrusion detection algorithms in different applications and deployments.

1. Introduction

Microservice-based applications emerged from the recent shift in the software development industry, where large monolithic applications are broken down into a collection of small and independent services that communicate with each other using lightweight mechanisms (Fowler and Lewis, 2014). Microservices can be independently scaled, replaced, or upgraded without jeopardizing system availability (Fowler and Lewis, 2014; Newman, 2015). Combined with continuous integration and continuous delivery (CI/CD) pipelines, organizations achieve higher availability, flexibility, and scalability (Newman, 2015), providing users with an on-demand and elastic service (Mell and Grance, 2011).

Using software containers for deployment enables fast and cost-effective provisioning, providing users with an elastic service that can be accessed from anywhere (Mell and Grance, 2011). As an alternative to traditional hardware virtualization, operating system (OS)-level virtualization leverages kernel features such as *control groups* and *namespaces* (Kerrisk, 2023). These mechanisms provide resource isolation and limitation, allowing a physical host to be shared by multiple lightweight and self-contained containers.

The utilization of containers and microservice-based applications is rising and expected to keep the trend (Kong, 2020). Nevertheless, security concerns are still present and in some cases may deter the adoption of these technologies (Kong, 2020). The main challenges are related to ensuring system security and reliability, particularly data and resource isolation. A relevant concern in multi-tenant environments, where legitimately obtained resources can allow a bad actor to perform malicious actions against neighbors or the host, leading to harmful consequences. The use of vulnerable container images, privilege escalation, and poor container isolation are examples of security risks that can harm an organization (Souppaya et al., 2017). Microservices increase the attack surface and network complexity due to their distributed nature (Dragoni et al., 2017).

It is important to ensure that applications running inside containers are secure from threats. Research in microservices security has mainly focused on authentication and authorization (Pereira-Vale et al., 2019; Hannousse and Yahiouche, 2021). Still, intrusion detection is a possible solution that monitors events occurring on systems or networks and analyzes them for signs of compromise (Bace and Mell, 2001). This

☆ Editor: Raffaela Mirandola.

* Corresponding author.
E-mail address: jeflora@dei.uc.pt (J. Flora).

https://doi.org/10.1016/j.jss.2024.112142
Received 14 February 2024; Received in revised form 7 June 2024; Accepted 17 June 2024
Available online 24 June 2024

technique has recently been applied to container and microservice-based systems with promising results (Flora et al., 2020, 2023a). The proposed techniques show interesting effectiveness with high detection rates and manageable false positives. However, there is a lack of an evaluation standard that allows practitioners and researchers to compare different works and understand how they perform under the same circumstances. This gap is largely due to the lack of up-to-date datasets that realistically represent microservices environments.

Generally, these evaluations ignore characteristics of microservice environments, namely their dynamic, scalable, and elastic nature. By using CI/CD pipelines, microservice applications are continuously being updated and modified, either to introduce new features or to fix previous defects. Intrusion detection approaches that are based on anomaly detection (where service profiling is involved) can become obsolete, compromising the security of business-critical applications.

In this paper, **we propose a benchmarking approach for intrusion detection in microservice applications** that considers the inherent characteristics of applications following this design. Our procedure considers representative workloads based on state-of-the-art technologies for microservice applications and environments. For this, we designed and developed a microservice intrusion detection dataset, with a wide range of representative applications, dynamic scenarios with scalable behaviors, and common attacks in such environments.

The benchmarking approach is instantiated in a benchmark for host-based intrusion detection in microservices. We evaluate several state-of-the-art algorithms and research works in several representative and meaningful scenarios, resulting in three case studies commonly found in microservice environments. Experiments demonstrate the utility of the benchmark in evaluating and comparing different intrusion detection algorithms in the context of microservice applications.

The main contributions of this paper are as follows:

- A specification-based approach to implement a benchmark for evaluating and comparing intrusion detection systems that considers i) representative workloads for different complexity scenarios, and (ii) specific metrics for different risk tolerance levels (see Section 3)
- A general methodology for generating datasets in the context of microservices that applies the characteristics of these environments and allows developers to overcome guidelines limitations and generate representative and useful data (see Section 4)
- A concrete instantiation of the benchmarking approach to demonstrate its applicability and feasibility, with six different IDSs evaluated in three case studies in the context of microservice applications (see Section 5)
- A dataset for intrusion detection in microservice applications generated using state-of-the-art technologies used in three different microservice applications with three different types of data: for training, validation, and testing, with ≈900 h of recorded data (see Section 6)

The rest of the paper is organized as follows. Section 2 analyzes the state of the art and related work, Section 7 presents three case studies of benchmark application and results. Section 8 concludes and highlights future work.

2. Background and related work

This section presents the background of intrusion detection, including a review of commonly used algorithms, and methodologies for evaluating detection mechanisms, as well as microservice characteristics and security challenges. An overview of benchmarking approaches and how to evaluate intrusion detection mechanisms is also included.

2.1. Intrusion detection approaches and algorithms

Intrusion detection is the process of continuously monitoring system events and analyzing them for possible security incidents, automated through intrusion detection systems (IDSs) (Bace and Mell, 2001). In recent years, this approach has been applied to various contexts (Hofmeyr et al., 1998; Bharadwaja et al., 2011; Abed et al., 2015; Srinivasan et al., 2019; Flora and Antunes, 2019). Recently, attention has shifted to microservice-based applications despite still existing limitations (Pereira-Vale et al., 2019; Hannousse and Yahiouche, 2021).

The process of detecting security intrusions follows mainly two **approaches**. A signature-based approach that compares new events with known malicious ones (Bace and Mell, 2001), and an anomaly-based approach that learns the benign behavior and identifies deviations (Chandola et al., 2009). Some works have taken a hybrid approach that mixes both techniques (Milenkoski et al., 2015b). Regardless of the approach applied, deploying IDSs as a security countermeasure is recommended because they provide valuable information about events occurring on the infrastructure, enabling system administrators to respond quickly to attacks (Bace and Mell, 2001).

For **algorithms**, the Bags of System Calls (BoSC) uses a sliding window over a system call trace to define a baseline behavior database containing *bags* of normal system calls and detect intrusions. These *bags* contain the frequency of each system call within the window (Dae-Ki Kang et al., 2005). The Sequence Time-Delaying Embedding (STIDE) algorithm operates in the same way BoSC. However, unlike BoSC, STIDE preserves the original order of system calls (Forrest et al., 1996). Isolation Forest has also been used to identify intrusion intrusions by isolating anomalous data (Liu et al., 2008). Rather than profiling the normal data, the model assumes that anomalies are seen in small numbers and isolated, i.e., distinguishable from benign data. An algorithm with a similar goal is the Local Outlier Factor (LOF), which identifies outliers in multidimensional data (Breunig et al., 2000). The algorithm is based on the concept that outliers are those data points that are most distant from their neighbors, which allows for the identification of possible intrusions.

Hidden Markov Model (HMM) has been widely used in intrusion detection, defining an anomalous sequence of events if its probability of belonging to the defined profile falls below a user-defined threshold (Warrender et al., 1999). We have also covered other commonly used machine learning (ML) algorithms for intrusion detection, such as Support Vector Machines (SVM). SVM uses trained knowledge to divide events into different classes separated by a hyperplane, with a special variant called One-Class SVM (OCSVM) used to define normal behavior without the need for anomalous data during training (Chen et al., 2005).

2.2. Microservices and security

Usually deployed using software containers, microservices leverage their lightweight and easy service instantiation (Dragoni et al., 2017; Sultan et al., 2019). To scale microservice applications to numerous services, administrators manage the system through orchestrators such as Kubernetes (Cloud Native Computing Foundation, 2022).

Because each service is containerized, a well-defined monitoring surface and boundaries are available for security mechanisms to use. For instance, system call traces can be collected from each container without instrumentation or intrusiveness, treating them as black boxes.

During the production phase, microservice environments can experience multiple events. To adapt to user demand, services are replicated to handle the increasing workloads, resulting in multiple instances running in parallel to distribute requests and achieve better response times (Dragoni et al., 2018). By scaling events and adjusting the number of service replicas, applications become elastic and provide a cost-effective use of resources.

These operational deployments change continuously over time, creating a very dynamic environment. The operational profile of an application is defined by the actions performed by its users (Musa, 1996). This profile directly influences the system calls generated by the application, which can be used by host-based intrusion detection algorithms to learn the behavior of the application. Operational profiles have a significant impact on the effectiveness of detection models used to secure the system, which must be reflected in the workload of a benchmark.

Anomaly-based IDSs for containers have been explored in previous works (Flora et al., 2020; Abed et al., 2015; Srinivasan et al., 2019; Lin et al., 2018; Röhling et al., 2019). However, some studies have limitations in the representativeness of their experiments and evaluations, and comparisons between them are difficult due to the limited availability of frameworks for doing so.

DevSecOps involves integrating security practices into a DevOps environment (Mohan and Othmane, 2016). To accommodate rapid development and deployment using CI/CD methodologies, security mechanisms must be seamlessly integrated with applications, leading to the concept of Runtime Application Self-Protection (RASP) (Strom, 2016; Madden, 2020). RASP aims to automate protection mechanisms using different detection and prevention strategies tailored to different technologies.

Container-based application security has seen progress, with works proposing anomaly-based IDS solutions (Abed et al., 2015; Srinivasan et al., 2019; Cavalcanti et al., 2021). An anomaly-based IDS for container applications with interesting results was proposed (Abed et al., 2015), and a similar methodology was applied in other work with an NGram probability, achieving high accuracy with low false positive rates (Srinivasan et al., 2019). A bag representation was applied with different machine learning algorithms applied and showed good results (Cavalcanti et al., 2021). Although these works achieved high accuracy, they suffer from limitations in workload and attack representativeness.

The security of microservice applications has primarily focused on authentication and authorization (Pereira-Vale et al., 2019; Hannousse and Yahiouche, 2021). Intrusion detection for microservices has received limited attention , with one work focusing on microservice fingerprinting based on system call data for data center management (Chang et al., 2019) and another on addressing scalable and elastic behavior (Flora et al., 2023a). There is an intrusion detection tool for microservice environments that integrates effectively with Kubernetes and KubeEdge (Flora et al., 2023b).

2.3. Benchmarks

The standard approach to evaluate and compare different tools or techniques for a given purpose is benchmarking (Vieira et al., 2012). A benchmark is a standard tool that allows its users to evaluate and compare different systems in a fair and useful manner, according to specific aspects such as performance, dependability or security (Vieira et al., 2012) A benchmark should adhere to various criteria to ensure it is meaningful and useful (Vieira et al., 2012):

- *Relevance*: the benchmark should be relevant to the objectives of the evaluation, providing insight on the effectiveness of intrusion detection in microservice environments and their challenges.
- *Repeatable and Reproducible*: the results obtained can be reproduced under similar conditions, allowing for independent verification. This means that the IDS results should be statistically similar across multiple runs.
- *Portability*: the benchmark should be possible and easy to implement on different systems. That is, it should be able to accommodate multiple types of intrusion detection algorithms and techniques.
- *Simplicity*: it should be easy to understand the benchmark and its results to increase credibility. It should be simple to understand what is being reported and how the results translate to a practical application for security intrusions.
- *Economical*: to run the benchmark should be cost-effective. The benchmark should be able to run on average hardware and should not significantly increase these requirements as the complexity of the workload increases.

A typical benchmark consists of three main components (Antunes and Vieira, 2010): i) workload, ii) measures, and (iii) procedure and rules. The workload is the representative set of cases that contribute to the evaluation. The measures used allow users to calculate and compare results for each system to understand how they perform against the target. The procedure outlines how the benchmark should be executed according to the defined rules. When following a measurement-based approach, **the workload is the most influential component**, and therefore it is focused next.

The workload used by a benchmark should ensure a group of five properties that directly affect its representativeness, usefulness, and potential impact (Vieira et al., 2012; Nunes et al., 2018). The workload applied should be *representative* of the real-world use case of the systems or components being benchmarked. It should be composed of a representative set of interactions that the system faces in its operational environment. The workload should be *comprehensive* in the components or features of the system that are exercised, according to the target domain. It should realistically cover different scenarios of the system's operation.

The workload should be a true representation of the systems being benchmarked. Three criteria are essential: *coverage*, to ensure a wide range of tests and applications; *relevance*, the load significance in the area of the benchmark; and *ground truth*, so that it is possible to compare the results of the evaluation with what was expected. The workload should also be *configurable* to allow its users to adapt it to their application scenarios and objectives. The workload should also be *scalable*, allowing to have different levels of complexity in the evaluation, to better represent possible intensity use cases for the systems.

To provide an accurate benchmark for intrusion detection, the workload should closely represent the real-world scenarios that IDSs are expected to encounter during operation. To be comprehensive, it should include different types of attacks and intrusions that test the different classifications provided by the detection models. To ensure broad coverage, the workload should consist of event data from at least two different testbed applications of different types, covering their full feature sets when exercised. The attacks and applications used should reflect current technologies and attackers to ensure relevance. They should provide labeled data for ground truth comparison, be partially used (e.g. a portion of the attacks) to facilitate configurability, and be scalable with varying complexity based on the testbed applications.

2.4. Datasets for benchmarking intrusion detection

Benchmarking an IDS or approach requires a comprehensive dataset that considers different scenarios and consists of benign and malicious data. Various datasets have been generated and published for evaluation purposes, allowing operators perform an impartial evaluation and comparison of different techniques. Their characteristics differ, making each proposed dataset advantageous for a given purpose. Table 1 provides an overview of the most relevant publicly available datasets. The table is not an exhaustive or complete analysis of available datasets, but rather focuses on the most recent and relevant datasets that have been more widely adopted by the research community. The datasets were collected from several surveys on the topic and characterized according to different criteria. Some of these datasets have been available and used for more than a decade and may no longer be representative of the current challenges (Tavallaee et al., 2009; CAIDA, 2007; UNM, 1999; MIT Lincoln Laboratory, 1998).

Table 1
Relevant datasets for intrusion detection published over recent years.

Dataset	Year	Experimental environment	Workload types	Type of attacks	Label data	Up to date?	Data types available	Ref.
Sever & Dogan	2023	several systems with single vulnerable container	synthetic load (scenario-dependent)	different CWE types (e.g., Auth., SSRF)	✓	✓	network traces	Sever and Dogan (2023)
CB-DS	2022	e-Shop application (dotnet, 2023)	synthetic load	container breakout misconfigurations	✓	✓	system calls	El Khairi et al. (2022)
LID-DS	2019	single vulnerable container	synthetic load (scenario-dependent)	common weaknesses	✓	✓	system calls network traffic	Grimmer et al. (2019)
CICIDS2017	2017	diverse multi host network	multi-day tailored user profiles	most common attacks from 2016 McAfee report	✓	✓	network traces	Sharafaldin et al. (2018)
ADFA	2014	web application	synthetic data	web-based attacks	✓	✓	system calls	Creech and Hu (2014)
UNSW-NB15	2014	multi-hosts network	IXIA Tool generated traffic	IXIA Tool generated attacks (nine families)	✓	×	network traces	Moustafa and Slay (2015)
ISCX 2012	2012	multi-system network	multi-profile data generation	different scenarios of attack	✓	×	network traces	Shiravi et al. (2012)
NSL-KDD	2009	network hosts	synthetic data (filtered KDD99)	DoS, Probe, U2R, R2L	✓	×	network traces	Tavallaee et al. (2009)
CAIDA	2007	single system	synthetic data (mostly removed)	DDoS Attacks	×	×	network traces	CAIDA (2007)
UNM 99	1999	Linux processes	live data synthetic data	different intrusions (buffer overflows, Trojans)	✓	×	system calls	UNM (1999)
DARPA 98	1998	multi-system simulation network	background data	network attacks (probing, sniffing)	✓	×	network traces, audit logs	MIT Lincoln Laboratory (1998)

The analysis presented focuses on the representativeness of the environment and workloads used on the monitored system. Representativeness is a very important property that has a huge impact on the applicability of the evaluated techniques in real-world scenarios and on the acceptance of a benchmark. A technique may perform well according to a given dataset used during the evaluation, but if that dataset is not representative of a real use case, the results are meaningless, or at least not of much use. Table 1 also indicates whether the datasets have labeled data, which is useful to for validating the results obtained during evaluations. It also indicates whether the dataset is outdated, describes the types of data available, and includes references to the papers in which the datasets were published or described.

The majority of publicly available datasets have their focus on network environments, typically providing *pcap* files containing traffic data across multiple hosts (Shiravi et al., 2012; Tavallaee et al., 2009). Only a small portion of the datasets consist of host information (e.g. audit logs, file system data, system calls, etc.) (Grimmer et al., 2019; Creech and Hu, 2014). This is a significant drawback for researchers attempting to propose techniques for host-based intrusion detection.

The challenge is compounded for those attempting to address security through intrusion detection in emerging environments, such as microservice-based systems. Of the datasets analyzed, only two focus on container-based deployments (the common case for microservices) (El Khairi et al., 2022; Grimmer et al., 2019). The LID-DS dataset was proposed in 2019, with some updates in 2021, and collects information at the level of the container used by an application (Grimmer et al., 2019). Still, it has some limitations; for instance, it lacks a representative testbed with users to collect benign and attack data, and the dataset primarily consists of vulnerable deployments from vulhub (The Vulnerabilities Hub, 2021) without a realistic use case. While useful for initial testing, a scenario that ensures effectiveness in a real microservice deployment is still needed as the traces are also generally short, about 45 s, with a total recording duration of about four days (El Khairi et al., 2022). The CB-DS dataset focuses on container breakout scenarios in a containerized application with ten attack scenarios with known vulnerabilities and misconfigurations, resulting in ≈23 h of recording (El Khairi et al., 2022).

Some works run experiments with a single containerized application, a simple scenario for microservice applications, but do not publish the datasets used (Flora et al., 2020; Flora and Antunes, 2019; Cavalcanti et al., 2021). A similar LID-DS work follows an approach that does not focus on microservice applications, but rather on collecting information from multiple Docker images containing known vulnerabilities (Oğur, 2022).

3. Benchmarking approach for intrusion detection in microservice applications

This section proposes a benchmarking approach that follows a specification-based style, describing the functions to be achieved by the detection models, the required inputs (workload), and the outcomes (Nunes et al., 2018; Kistowski et al., 2015). The implementation of the benchmark uses measurement-based techniques, in which the user exercises the tools under benchmarking using to identify security intrusions in a dataset of host data with mixed data, benign and malicious, to obtain a small set of measures that represent the intrusion detection capabilities of the tools.

The objective is to evaluate and compare a set of IDSs on a representative dataset (workload), containing benign and malicious data, and verify their effectiveness in detecting the attacks; then, rank them using a set of metrics to understand which are more applicable to each scenario.

The general architecture of the proposed benchmark for microservice intrusion detection models is depicted in Fig. 1. It consists of four components, which are introduced here and described in the following subsections:

(1) *Workload (Dataset)*: data required to train and test the detection models. It should be representative of the normal environment and operation of the target system and contain malicious data that is representative of the malicious attacks that the system may face.

(2) Scenarios: represent the different levels of risk tolerance for the system to which the detection models are to be applied. The risk tolerance is lower for a system with a higher probability of attack and higher criticality, thus requiring the detection models to provide higher detection rates.

(3) Metrics: an easy to understand and meaningful way to evaluate and compare the IDSs. The metrics define the effectiveness of the detection models in identifying malicious data in the workload.

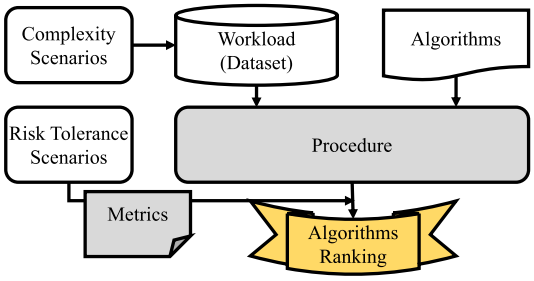


Fig. 1. General architecture of the benchmark.

(4) Procedure: rules to be followed when running the benchmark. It consists of the steps to be taken to prepare for the execution of the benchmark and the process to be followed during its execution. If followed, the benchmark will produce the measurements and the ranking for the desired scenarios in a reproducible and representative manner.

3.1. Workload

The workload used to benchmark detection models is critical to the tests performed and the results obtained. The quality of the information and its representativeness will strongly influence the validity of the results. In the case presented, the dataset defines the events processed by the algorithms/techniques used to build the detection models. To increase the usefulness of the results, the workload should be as close as possible to the real deployment.

To ensure a valid picture of microservice environments and their typical behavior, the dataset should include the following information:

– Data from a **stable deployment**, which represents scenarios where the services are not changed and remain with the same configuration.

– Data from a **dynamic deployment** where dynamicity is present. The scaling events take place to accommodate modifications in user demand.

– Data from a **continuous deployment** that is characterized by the continuous delivery of new versions of the software.

There are different types of workloads that can be used in a benchmark: real, realistic, and synthetic (Antunes and Vieira, 2010; Gray, 1993). *Real workloads* are collected while real systems are running in production environments. These workloads tend to be more representative, but are difficult to obtain because it would mean having a live system to monitor and information about malicious events taking place within it. *Realistic workloads* are artificial workloads based on real systems, vulnerabilities, and attacks in the domain of the benchmark. When the methodology for generating realistic workloads is well defined, the results obtained ensure representativeness of the real system, and even have the advantage of depicting some edge cases that are rare in real systems but may be important in highly critical scenarios. A common approach to generating realistic datasets is attack injection (Flora et al., 2020; Milenkoski et al., 2015b). *Synthetic workloads* are generated according to a set of guidelines and premises that facilitate data collection but raise concerns regarding representativeness.

3.2. Scenarios

For our security benchmark, a scenario is based on the level of risk tolerance that an organization is willing to accept in pursuit of the system's objective (Dempsey et al., 2011). In this way, this approach considers not only the criticality of the system, but also the likelihood of attacks and their business impact. In the case of an air traffic control system (critical scenario), the detection models must detect as many possible intrusions as possible, even if, at a tolerable level, some are misclassified. Conversely, at the other end of the criticality scale, a public bulletin board (non-critical scenario) is concerned with minimizing the number of false positives, since they are costly to investigate, and resources are scarce. This benchmark takes into account three levels of risk tolerance, established as realistic scenarios and defined based on previous research (Antunes and Vieira, 2015):

• ***Low-tolerance***: the system/organization has a low risk tolerance, so it requires security intrusions to be detected at the highest rate.

• ***Moderate-tolerance***: security intrusions may go undetected at the expense of reducing false positives. The system's tolerance level allows for some risk.

• ***High-tolerance***: it is essential that the detection model minimize false positives as much as possible. The objective is to identify security problems without wasting resources.

In short, the acceptable level of false positives and false negatives is defined by the risk tolerance of the system/organization. At lower tolerance levels, the acceptable level of false positives is higher than at higher tolerance levels, while at higher risk tolerance levels the main goal is to reduce false positives even at the expense of reporting intrusions.

3.3. Metrics

To compare and rank intrusion detection models, we use appropriate metrics commonly used in intrusion detection evaluation (Flora et al., 2020; Milenkoski et al., 2015b). For classification problems like this, the events that are correctly classified by the detection models are known as true positives (TP) (for intrusions) and true negatives (TN) (for non-intrusions). The intrusion events that are misclassified as non-intrusions are called false negatives (FN), while the symmetric misclassification is called false positives (FP).

• **Recall**: a ratio of attacks in the ground truth that are correctly reported as attacks.

$$recall = \frac{TP}{P} = \frac{TP}{TP + FN} \tag{1}$$

• **Precision**: a ratio of reported attacks that are correctly classified.

$$precision = \frac{TP}{TP + FP} \tag{2}$$

• **F-Measure (FM)**: the harmonic mean of precision and recall.

$$FM = \frac{2 \cdot precision \cdot recall}{precision + recall} \tag{3}$$

• **Markedness**: how consistently the outcome has the detection model as a marker, i.e. how marked a condition is for the specified detection model, versus chance.

$$Markedness = \frac{TP}{TP + FP} + \frac{TN}{FN + TN} - 1 \tag{4}$$

For each risk tolerance scenario, we select a primary metric to rank the IDSs and a secondary metric to act as a tiebreaker (see Table 2). These metrics are adapted from other work (Antunes and Vieira, 2010). The choice of metrics depends on the tolerance level that the system can withstand, which is closely related to the criticality of the system and the resources available to ensure its security.

In scenarios where the risk tolerance is low, we choose the recall metric. This metric aims to identify the maximum number of intrusions at any cost, even if it means accepting some false positives. In the event

Table 2
Metrics by risk tolerance level.

Risk tolerance	Metric	Tiebreaker
Low	Recall	Precision
Moderate	F-Measure	Recall
High	Markedness	Precision

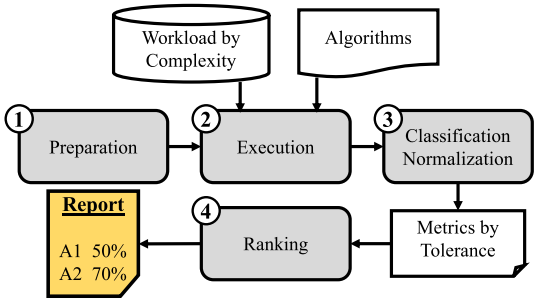


Fig. 2. Benchmark procedure diagram.

of a tie, we turn to precision to prioritize the detection model that produces fewer false positives. These evaluation metrics, which have different focuses, effectively represent the performance of detection models.

• **Model Reusability Index (MRI)**: evaluate whether a detection models is more likely to be reused for different releases of a microservice (Flora and Antunes, 2024).

$$MRI = \sum_{i=1}^{n} \omega_i C_i = \omega_1 CD + \omega_2 Al + \omega_3 RI \quad (5)$$

The Model Reusability Index (MRI) metric is a special case. The definition of the metric has been published in previous work (Flora and Antunes, 2024). This metric is intended to be used optionally for cases where the selected workload (dataset) contains information on different releases of a microservice. The metric requires information about the source code of the different releases and core data regarding the events used for training and testing of the detection models. Originally applied using system call data, it is not included in the scenarios defined in Table 2. Rather it can be used as an optional configuration by the benchmark user.

3.4. Procedure

The execution of the benchmark follows a simple procedure consisting of four high-level steps, as shown in Fig. 2.

(1) *Preparation*: comprises the selection of the set of algorithms/techniques to be benchmarked. The algorithms or techniques may allow for different configurations, and thus it is possible to benchmark the same one with different parameters to tune it and obtain the most effective.

(2) *Execution*: consists of training and testing the detection models on the workload according to the selected scenarios and measures. This step is the one that allows to compare the performance of the algorithms/techniques. Details in Section 3.4.1.

(3) *Classification Normalization*: normalization of the results produced by the algorithms/techniques (Section 3.4.2)

(4) *Ranking*: ranks the detection models used in the evaluation according to the benchmark scenarios and metrics used.

3.4.1. Execution

The *execution* step of the procedure must prepare the workload to be processed by the algorithms and techniques to be benchmarked. The training dataset should be cleaned and prepared to fit the operation of the algorithm, for example, some ML algorithms require the data be organized into feature vectors, so a data transformation step is required.

Once the workload has been prepared to meet the requirements of the algorithms, training takes place. Typical ML pipelines include parameter tuning to achieve a stable and usable model, for which a validation dataset should be used. For other algorithms that do not follow an ML approach, but require a training step and process the information to obtain patterns of behavior from the data, we propose a tuning mechanism that allows to identify whether the detection model has reached a stable and usable state, or steady-state.

For this, we use the formula $0 \leqslant \frac{S_{t_{s2}} - S_{t_{s1}}}{t_{s2} - t_{s1}} \leqslant \sigma$ (Milenkoski et al., 2015a) to decide whether an interval is at learning steady-state. A detection model reaches steady state when this inequality is satisfied five consecutive times for $\sigma = 0.15$, based on previous work (Flora and Antunes, 2019). This is a simple but effective approach that allows benchmark users to identify when detection models reach steady state.

3.4.2. Classification normalization

Different detection models can produce different types of outputs. According to the output they produce, they can be classified as *soft* or *crisp* detectors (Islam et al., 2018). Models that produce a score (e.g. intrusion probability) instead of a decision (i.e., anomalous or normal) are called soft detectors. Crisp detectors produce a decision. Soft detectors can be converted to crisp detectors by applying a threshold on the output score (Islam et al., 2018).

To compare the detection models, we need to ensure that they produce the same type of output, which is a final decision. This can be either a binary decision (i.e., anomalous or normal) or a class decision (e.g. normal, user2root, etc.). For a class-based decision to be comparable, all detection models should be able to produce the same set of classes. If some detection models produce a binary decision and others do not, the ones that produce classes can be converted to an "anomalous" decision by generalizing the attack classes. To convert soft detectors into crisp binary detectors, the benchmark user must define a threshold that separates anomalous from normal decisions (Islam et al., 2018).

Second, detection models may process information differently, and so the output may refer to different events (e.g. a system call or a window). To allow a fair comparison between detection models, the classifications should be translated into meaningful and comparable data points. For this purpose, we propose a mechanism based on a time interval (Flora et al., 2023a) that can aggregate different outputs of each model. This implies that detected anomalies are reported for a user-defined time interval, which allows for fair comparisons.

4. Dataset generation methodology

This section outlines the methodology designed to generate datasets for evaluating intrusion detection techniques in microservice applications. The approach consists of two main phases: planning and execution, as illustrated in Fig. 3, and explained in the following sections.

4.1. Planning phase (P)

The planning phase encompasses 3 steps to establish the components of the dataset generation environment.

4.1.1. Environment definition (P1)

To ensure the representativeness of the dataset, the selected systems and deployment environment must be representative of real-world applications. In the **systems selection** step, applications should resemble actual scenarios, such as e-commerce or video streaming, exhibiting complexity and rich feature sets for users to interact with. That is, when a user makes a request the flows to solve it should have different interactions among the microservices of the testbed. To facilitate representativeness, the selection should be based on previous research use, using applications with broad community acceptance.

Next, the **deployment and workloads definition** phases establish user behavior profiles and workload intensities for the load generators. User behavior profiles detail the set of operations that users perform on the application, that is, the set of requests to which the application responds. This set of requests should apply a high coverage (e.g. >80%) of the features provided by the application, while having a similar distribution to that of real users. The workload intensity describes the

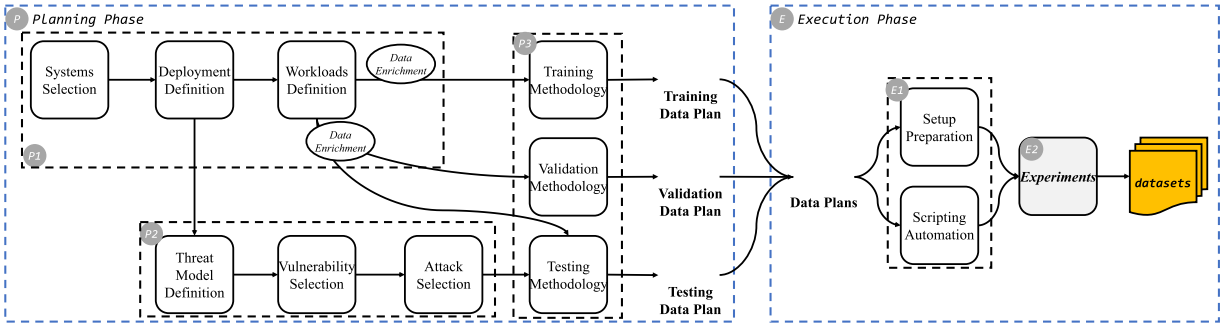


Fig. 3. Methodology proposed for the production of microservice intrusion detection datasets.

number of active users over the duration of the experiments. The workload intensity should be derived using practical application profiling, and then defined according to the different patterns of performance testing (Kistowski et al., 2014). A load generator is an automated tool that takes the user behavior profiles and the workload intensity definition, and conducts the requests to the application endpoints at the rates defined by the workload intensity. Examples are JMeter (jmeter.apache.org) and Locust (locust.io).

To improve the quality of the dataset, we propose to introduce benign operations that are neither malicious nor the result of user requests. During the **data enrichment** step, we conduct maintenance tasks that provide varied data beyond standard client interactions. These operations result in unseen behavior from the system without being malicious. The data produced allows techniques and algorithms to learn more broadly, covering more than the typical response to client requests. In this way, detectors can prevent false positives resulting from common system administrator operations. It can also be used to identify whether the algorithms can generalize the knowledge acquired during training and expand it to other system administrator operations. The operations should cover typical administrative actions, such as memory, process, or storage related.

4.1.2. Malicious actor behavior (P2)

Simulating realistic attackers involves defining a **threat model** that outlines attack points and mechanisms using frameworks like MITRE ATT&CK (Strom et al., 2018). The definition of the threat model includes the attack vectors against the system and how those attacks will be executed. For **vulnerability selection**, databases like NVD and CVE Mitre are extremely useful, while **attack selection** identifies exploits that target the vulnerabilities. The former is especially important when it comes to the vulnerability injection phase of the process, where vulnerabilities are added to the application. Using known vulnerabilities also increases the representativeness of the attacks performed. For the attack selection, proof of concept code (exploits) is available for download online, in databases such as exploit-db or in online GitHub repositories, and are recommended to ease the implementation of the attack injection procedure.

MITRE ATT&CK is a knowledge-based framework that incorporates adversary tactics and techniques based on real-world observations. It is commonly used for threat modeling in enterprise, mobile, and industrial control system environments. The framework is extensive and provides details about 14 tactics, 201 techniques, and 424 sub-techniques (attack.mitre.org) that are commonly used by malicious actors in an enterprise setting. It is a useful tool for modeling the attacker and defining attack vectors that can be exploited. The selection of threats to represent malicious behavior can be derived from a risk-based assessment to represent the most relevant and impactful attacks.

4.1.3. Experimental methodologies (P3)

Datasets encompass various data types used by intrusion detection algorithms/techniques. These mechanisms are different in nature and operation, and require different information during the training and testing phases.

First, it is necessary to define what data should be collected from the environment in which the application is deployed. Host information (system calls, filesystem data), network information (connection flows, traffic), and application data (performance data from load generators) can be gathered.

Second, the type of data files composing the dataset should be defined. The dataset files can contain benign, purely malicious, or mixed information (Milenkoski et al., 2015b). Benign data involves non-malicious client interactions, pure malicious data records only attacker operations on the application, and mixed data combines both. Each file type requires a specific **experimental methodology** detailing data collection steps, timing, workload initiation, and attack injection (if applicable).

There should be three types of experimental methodologies to produce data files. This increase usefulness and representativeness of the dataset, as it allows test data files to be completely independent of the training and validation used to construct the model, and avoids the typical data splitting applied when training machine learning models (Russell, 2010). A **training methodology** that clearly defines how the duration of the data collection procedure and the steps taken to obtain data are used to train algorithms/techniques. A **validation methodology** that can be used as an independent data source from training to provide evaluation of the model during the training phase, so it is possible to tune the parameters (Russell, 2010). A **testing methodology** that clarifies how the data used to test the final model should be collected, with injection details and workloads used.

4.2. Execution phase (E)

Once the planning phase is complete, the framework and environment for data generation should be developed. A framework reduces human interaction in the generation of datasets and increases the repeatability of the experiments conducted.

4.2.1. Experiments preparation (E1)

To prepare the experimental campaigns, it is important to conduct a **setup preparation**: define the architecture and set up the environment; a **scripting automation** (the recipes to deploy the environment and the application), and execute the framework responsible for controlling the experiments. Infrastructure automation tools that support several systems, like Ansible (ansible.com), can streamline the setup. A general programming language can be used to implement the different methodologies for generating the various data files.

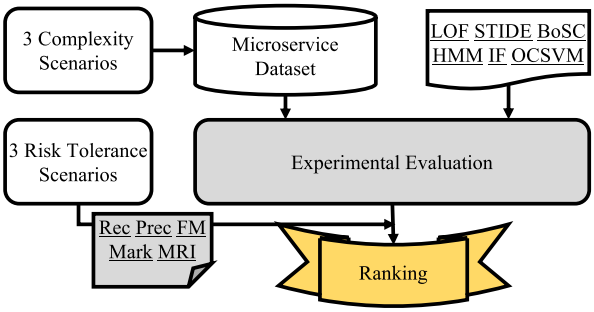


Fig. 4. Instantiation of the intrusion detection benchmark for microservice applications.

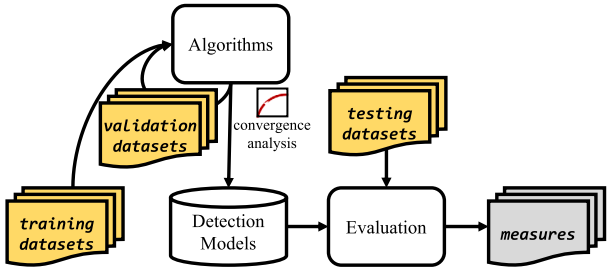


Fig. 5. Benchmark instantiation procedure.

4.2.2. Experiments execution (E2)

All selected configurations (e.g. scenarios, attacks, testbeds) are combined into produce the execution queue and executed. The **experiments** are conducted using the developed framework. The data files are collected according to the methodologies and should be validated for accuracy and completeness.

5. A benchmark for host-based intrusion detection in microservices

The benchmarking approach presented (Section 3) aims to be a general approach that can be applied to different intrusion detection mechanisms, with different targets, and operation modes, that rely on different data types to detect attacks and intrusions. In this section, we present an instantiation of the proposed benchmark approach for host-based intrusion detection in microservice applications. The instantiation of the benchmark uses measurement-based techniques where the selected IDSs are executed to detect security intrusions on the workload (dataset) containing different types of attacks to obtain the measures and calculate the metrics to rank the tools (see Fig. 4).

5.1. Workload

As mentioned earlier, the workload used in the benchmark should be representative of the environments in which the IDSs may operate. To ensure this characteristic, one needs to provide a wide range of applications that are dynamically exercised to expose the IDSs to different operational profiles and test them for different possible uses. Since no microservice intrusion detection dataset exists, we created our own **workload** for this instantiation of the benchmark. We applied the dataset generation methodology presented in Section 4 to generate 900 h of recording data, using three different testbeds with multiple configurations to obtain data for the three complexity scenarios described. The details of the datasets produced using host data from microservice applications are provided in Section 6.

5.2. Scenarios

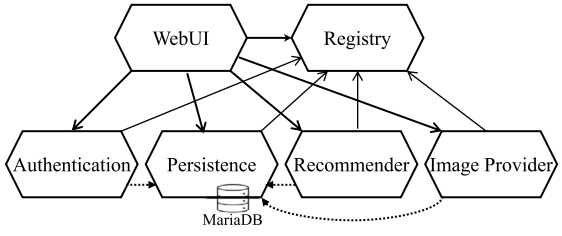
When instantiating the benchmark, one should aim for a comprehensive coverage of the real-world scenarios in which an IDS would operate and try to cover them. The definition of the complexity scenarios should follow an increasing approach, starting with the baseline case and ending with the most relevant and challenging case the IDS may face. For our instantiation, we defined **three complexity scenarios** for this instantiation of the benchmark based on its specification. In the first scenario, the algorithms are trained and applied in the same operational environment, where the testbeds have the same deployment scenario. The second considers a scalability environment where the monitored microservice scales and has a varying number of active service replicas. The third considers a more complex scenario where the microservice is continuously deployed across its different releases. This means that as the microservice evolves, new features and bug fixes are added and its behavior may change. The benchmark for the first two case studies uses data from all three testbeds, while the third case study uses data from the TeaStore testbed only.

The three **risk tolerance scenarios** defined in the benchmark guidelines are applied in this instantiation as a way to rank the different detection models according to the criticality of the system in which they might be applied. Hence, we adopt the low, moderate, and high risk tolerance scenarios, which are defined and detailed in Section 3.2. The adoption of these risk tolerance scenarios also defines the **ranking metrics** for each scenario (presented in Table 2).

5.3. Procedure

For the procedure of the instantiation of the benchmarking approach, we start by selecting the algorithms and then define the concrete steps based on the specification.

The selection of algorithms depends on the goals of the benchmark user. The algorithms used should be those that are most relevant to the intended goal and the final candidates after an initial study and selection phase. For our instantiation, we intend to demonstrate the breadth of the benchmark and therefore selected several **algorithms**, used in previous published work, to be benchmarked based on the research conducted in Section 2.1. The selected algorithms have different approaches and characteristics, demonstrating the range of applicability of the benchmark. In the evaluation, the use of the STIDE and BoSC algorithms is based on previously published research (Flora

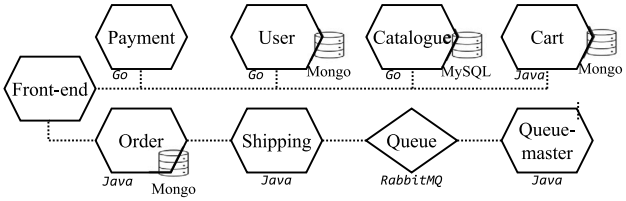


(a)

	Configuration								
Service	#1	#2	#3	#4	#5	#6	#7	#8	#9
WebUI	1	1	2	3	3	4	5	5	7
Image	1	1	2	3	3	4	5	5	6
Authentication	1	2	3	4	5	6	7	8	9
Recommender	1	1	1	2	2	2	3	3	3
Persistence	1	2	2	3	4	4	5	6	6

(b)

Fig. 6. TeaStore application: (a) architecture adapted from von Kistowski et al. (2018); (b) scaling scenarios, adapted from von Kistowski et al. (2018).



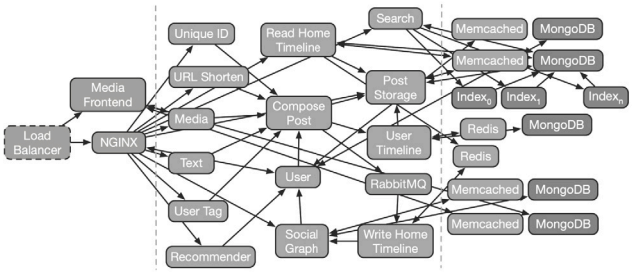
(a)

Configuration

Service	#1	#2	#3	#4	#5	#6	#7	#8	#9
carts	1	2	2	3	4	4	5	6	8
catalogue	1	1	2	2	2	3	3	3	5
front-end	1	2	3	3	5	5	7	7	9
orders	2	3	3	4	6	6	7	9	11
payment	1	1	1	1	1	2	2	2	4
queue-master	1	2	2	2	2	3	3	4	4
rabbitmq	1	2	2	2	2	3	3	4	4
shipping	1	2	2	3	3	3	4	5	6
user	1	1	1	1	1	1	2	2	3

(b)

Fig. 7. SockShop application: (a) architecture adapted from Weaveworks (2017); (b) scaling scenarios.



(a)

Configuration

Service	#1	#2	#3	#4	#5	#6	#7	#8	#9
compose-post-service	1	2	2	3	3	4	4	5	5
home-timeline-service	1	1	1	2	2	2	3	4	4
media-frontend	1	1	2	3	3	3	3	4	4
media-service	1	1	2	2	2	3	4	4	5
nginx-thrift	1	2	2	3	4	5	6	7	8
post-storage-service	1	1	2	2	2	3	3	3	4
social-graph-service	1	2	2	2	3	3	3	3	4
text-service	1	1	1	2	2	2	3	3	4
unique-id-service	1	1	1	1	2	2	2	2	3
url-shorten-service	1	1	1	1	2	2	2	2	2
user-mention-service	1	1	1	2	3	3	3	3	3
user-service	1	1	2	2	3	3	4	5	5
user-timeline-service	1	2	2	3	4	5	5	5	5

(b)

Fig. 8. Social Network application: (a) architecture from DeathStarBench (2023); (b) scaling scenarios.

et al., 2020; Abed et al., 2015) along with algorithms LOF, OCSVM, Isolation Forest, and HMM.

The algorithms are benchmarked using a segment of the datasets generated in Section 6. To train the detection models, we used the initial 24-hour segment of each data collection for each testbed, with the validation data used to fine tune the parameters, and for testing we excluded the data samples that had CVE IDs from 2014. For the STIDE and BoSC algorithms we followed the approach of previous research (Flora et al., 2020), where the sequence of system calls is processed into windows of configurable size and reports are based on those windows. For the remaining algorithms, we followed the approach of Araujo et al. (2023), where the sequences of system calls are converted into feature vectors using ngrams of 50 system calls for training and testing.

The **procedure** implemented is detailed in Fig. 5 and based on the specification proposed in Section 3.4. It initiates with the selection, and preparation, of the training datasets according to the applied complexity scenario. The prepared training data is fed to the algorithms for the training phase, where the validation datasets are included to fine tune the algorithms. The convergence analysis dictates when to stop the training phase and the detection models are ready for testing. The models are then evaluated using the test datasets, with different configurations tested depending on the complexity scenario. The measures are collected and processed to normalize the output of the different detection models. Finally, the metrics for the three risk tolerance levels are computed, and the detection **models are ranked** according to the benchmark guidelines (see Table 2). The experimental results are analyzed in Section 7.

6. Dataset for host-based intrusion detection in microservices

This section demonstrates the application of the outlined methodology (Section 4), detailing the concrete techniques, artifacts, and experiments carried out to generate datasets for host-based intrusion detection in microservice-based applications. A sample of the dataset is available at: https://doi.org/10.5281/zenodo.10655126.

6.1. Planning phase

The design phase of the experiments involves several steps to determine the methodology for each type of data that makes up the datasets. These steps are described below.

6.1.1. Systems and deployment definition

To ensure the representativeness of the generated datasets, we analyzed state-of-the-art open-source and publicly available microservice applications (von Kistowski et al., 2018; Weaveworks, 2017; The Spring PetClinic Community, 2016; Zhou et al., 2018; Gan et al., 2019). From this pool, we selected TeaStore, SockShop, and the SocialNetwork from DeathStarBench based on the number of services, development technologies, and prior research utilization. The architectures are presented in Fig. 6(a), Fig. 7(a), and Fig. 8(a).

TeaStore simulates an e-commerce tea store with five services sharing a base image and Java for development (von Kistowski et al., 2018). During the experiments, we use two versions of TeaStore, v1.3.0 and v1.3.8. It was developed for research purposes with an emphasis on performance testing and benchmarking. `WebUI` serves as the storefront, with three inner services handling tasks like image provision, authentication, and product recommendation. The services communicate using REST and are all deployed into an Apache Tomcat server configured on the base image. **SockShop** portrays an e-commerce store selling socks with eight microservices (Weaveworks, 2017). The `front-end` coordinates interactions while other services handle tasks like orders, payments, and catalogue management. It includes diverse programming languages for the services, such as NodeJS, Java, and Golang; and different databases like MySQL and MongDB. The services interact using REST.

The **Social Network** application portrays a social network with unidirectional follow relationships that is implemented using microservices that communicate through Thrift RPCs (Weaveworks, 2017). The application is part of a benchmark suite of microservice applications developed for research purposes (Gan et al., 2019). Users can create posts, with embedded text, media, links, and also tag other users, which are sent to all their followers. Users can also read, mark a post as

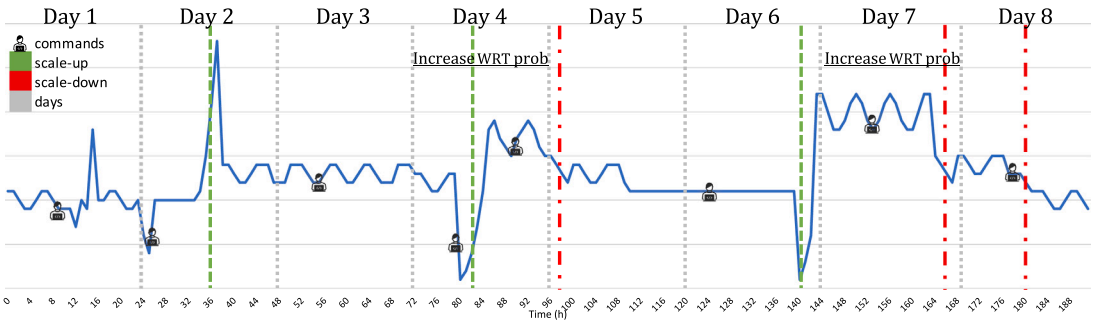


Fig. 9. Long series characterization for the training phase. Eight days of scaling events and data enrichment.

favorite, and repost previously created posts. Over the experiments conducted, we loaded the `'socfb-Reed98'` social graph available in the repository. The application covers multiple technologies, such as programming languages (e.g. C/C++, Java, Python, NodeJS); storage systems like Redis, Memcached, and MongoDB; and search engine such as Xapian.

Nine scaling scenarios were defined for each system, influencing deployment and active service replicas. Deployments vary from 7 to 31 containers (active service replicas) for TeaStore, from 15 to 59 for SockShop, and from 27 and 70 to SocialNetwork. Fig. 6(b) was taken from the original paper that presented TeaStore (von Kistowski et al., 2018), with a slight adaptation for scenario #9, where in our case the `WebUI` service has seven active replicas, to have a larger sample of active service replicas. To derive the scenarios for SockShop and SocialNetwork, we subjected them to different intensity loads and used HPA (Kubernetes, 2020) to observe the auto-scaling behavior of the application. Fig. 7(b) (SockShop) and Fig. 8(b) (SocialNetwork) describe the number of active service replicas for the nine scenarios.

6.1.2. Workloads and operator commands

For representativeness, we adopted recommended workload profiles for each application with different intensities. An eight-day training phase (see Fig. 9) collected diverse data by applying diverse user behaviors, complemented by benign operator commands for data enrichment. We selected eight days for the duration of the training data collection as it is extensive enough to have sufficient time to record a significant amount of data from a system that is continuously operating in a dynamic environment with several modifications. The user behavior profiles are composed of different requests that reflect the e-commerce and social network activities referred. The writing operations (e.g. creating a post or making a purchase) have varying execution rates (10% in general, with days 4 and 7 having 20%). This distribution is consistent with the common behavior of e-commerce users who browse pages for short periods of time (German Zenetti, 2022) and other of read/write workload experiments. The load intensities included various curves to increase variability: constant (e.g. $[26, 34[$ hours); quadratic (e.g. $[34, 38[$ hours); sinusoidal with different periods (e.g. $[16, 18[$ and $[38, 72[$ hours); cubic (e.g. $[80, 85[$ hours); and reciprocal (e.g. $[96, 100[$ hours). Six scaling events occurred during the 8-day period, incorporating scale-up and scale-down shifts among four different deployment scenarios.

Benign operator commands, that emulate system administrators' actions, were introduced as data enrichment steps. These operations were derived from real-world scenarios and covered resource usage checks, network verification, and file handling. Fig. 10 presents the four sets of data enrichment commands used in the experiments performed. They were selected after researching several reference blogs and forums for system administrators (Ran, 2023; Brown, 2022; DevOpsArticle, 2021).

6.1.3. Threat model

To have a representative experimental campaign, using attack injection, we defined different layers that could be targeted by a malicious agent in such environments, adapting a previous threat model for container-based deployments (Flora and Antunes, 2019). The shared infrastructure of microservices allows attackers to compromise multiple layers. Targets include services, the virtualization engine, and the host operating system. An attacker can try to compromise other services (or tenants) to obtain access to it or endanger its operation. An attacker can target the virtualization engine where the microservices are running. This could be through attempts at escaping the isolation of its container or gaining access to or control of the engine to manipulate the other services sharing the infrastructure. Because container-based solutions use kernel cgroups and namespaces to achieve isolation and data separation, malicious services may also attempt to target the host operating system. These attacks allow agents to escalate privileges and gain control of the host machine, allowing them to perform a range of activities.

The Mitre Att&ck framework informed attacker tactics such as reconnaissance, code execution, privilege escalation, and lateral movement (Strom et al., 2018). An attacker applies reconnaissance to gather information, execution of malicious code to establish a persistent connection, and gain continuous access. It is also common to perform privilege escalation to increase the level of access in the system. A common operation in microservices is lateral movement, where after gaining control of one service, the attacker attempts to move to another related service. In short, we have a malicious service that shares infrastructure with the application and attempts three types of attacks against: i) an application service; ii) the virtualization engine; and iii) the host OS.

```
# DE Set #1          |# DE Set #2          |# DE Set #3          |
#---------------------|---------------------|---------------------|
df -h                |ps faux              |uptime               |
top                  |ipcs -a              |uname -a             |
cat /proc/meminfo    |ss -s                |ls -l $HOME          |
vmstat               |ip route             |mkdir -p $HOME/dir   |
ps faux              |uptime               |echo "OK" >> ok      |
ipcs -a              |uname -a             |cd /var/log/         |
ss -s                |ls -l $HOME          |rm -r $HOME/dir      |
ip route             |mkdir -p $HOME/dir   |ip a                 |
#-------------------------------------------------------------|
# DE Set #4                                                   |
#-------------------------------------------------------------|
du -sch .                                                     |
jobs -p                                                       |
cd $HOME/                                                     |
mkdir -p new_folder                                           |
echo "Lorem Ipsum text example #1." >> new_folder/file1.txt   |
echo "Lorem Ipsum text example #2." >> new_folder/file2.txt   |
diff new_folder/file1.txt new_folder/file2.txt                |
tar -cvf folder.tar new_folder                                |
rm -r new_folder folder.tar                                   |
#-------------------------------------------------------------|
```

Fig. 10. Data Enrichment command sets used. These sets are representative of common system administrator tasks for system operation assessment.

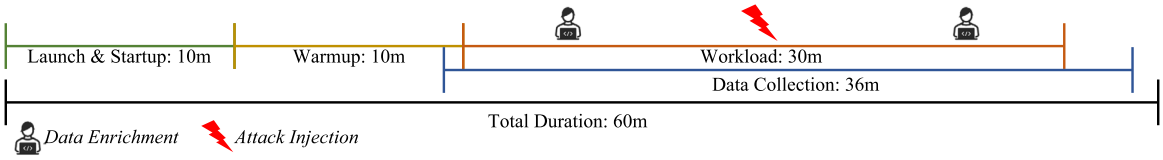


Fig. 11. Attack data generation methodology followed. Validation data also follows this approach, but the attack injection step is not performed.

6.1.4. Vulnerabilities and attacks

Several real-world vulnerabilities at multiple levels of the stack were collected to increase the representativeness of the attacks executed. These weaknesses have been reported by security researchers over the past decade, with common vulnerability enumeration (CVE) IDs assigned ranging from 2014 to 2022, an eight-year period. Although the 2014 vulnerabilities are a bit old, they are representative of a class of vulnerabilities that are still being found in current software and exploited in the wild. For example, the Cybersecurity and Infrastructure Security Agency's (CISA) top 2022 routinely exploited vulnerabilities include vulnerabilities with CVE IDs from 2018 and 2017 (CISA, 2023).

The vulnerabilities are detailed in Table 3, with severity data, with common vulnerability scoring system (CVSS) base score, and the category assigned based on the common weakness enumeration (CWE) categorization system. The presented vulnerabilities were selected to ensure the representativeness of the experiments and directly contain different 11 CWEs that are present in the top 25 CWEs of 2023 (CWE, 2023). The weaknesses in our dataset also represent the same CWE class as another 6 CWEs present in the top 25, which means that our vulnerabilities represent directly and indirectly represent 17 of the weakness categories in the most dangerous software weaknesses of 2023 (CWE, 2023). This selection is also based on the tactics and techniques of the Mitre Att&ck Framework, as described in Section 4.1.2. Table 3 presents vulnerabilities covering reconnaissance (CVE-2016-6816), code execution (CVE-2017-12617), privilege escalation (CVE-2016-5195), obtaining credentials (CVE-2016-1240), and container breakout (CVE-2019-5736; CVE-2022-0492).

To exploit the presented vulnerabilities, we have collected proof-of-concept code (i.e., exploits) that demonstrate their presence and activate them. This proof-of-concept code is publicly available and was collected from different sources, such as exploit-db (*exploit-db.com*), 0day.today (*0day.today*), and GitHub repositories. After a preliminary analysis and testing of the found candidates, we selected an exploit for each vulnerability and prepared the setup to be used. The services attacked in each testbed are: TeaStore, `WebUI`; SockShop, `front-end`; and SocialNetwork, `nginx-thrift`.

6.2. Execution phase

To generate the datasets, we devised a methodology with microservice applications deployed on a Kubernetes cluster and workloads to exercise the target application. We collected system call data using μDetector (Flora et al., 2023b), which uses sysdig (Sysdig, Inc, 2022). The methodology considers a set of **17 different exploits that target the same number of real-world known vulnerabilities in three layers of the service attacked through an attack injection procedure**. The fact that we use a version of the service with known vulnerabilities increases the representativeness and confidence of the experiments and the results obtained.

Fig. 11 provides a graphical overview of the methodology depicting the multiple phases of the experiments performed to collect information from the microservices, for attack and validation data purposes. It starts with 10 min to launch and start the applications, populating and initiating databases. This is followed by a 10-minute warm-up period with a light-intensity workload to exercise the testbed. Data collection lasts for 36 min, starting one minute before the end of the warmup phase and ending five minutes after the main workload, which runs for 30 min. The validation data does not include the attack injection step depicted in Fig. 11.

During the experimental campaign, the monitored system is exercised with only benign workloads to collect training traces. The detection (or testing) traces are collected when an attack is injected while the benign workload is running, providing a mixed data set that can be used to test the detection models. The process is repeated for multiple deployment configurations of each system. This diverges from common datasets where the only varying factor is the attack used in each scenario. The information on the timespan of the attack is collected and stored, so it is possible to define the ground-truth for each testing dataset, allowing metrics of algorithm effectiveness to be computed.

6.2.1. Experimental setup

We used four physical machines: one to serve as the load generator, we used Locust (locust.io), which interacts with the application and making the requests; and three that composed the cluster, a master and

Table 3
List of vulnerabilities used and respective CVE information.

Component	CVE-ID	CWE ID	Vulnerability Type(s)	CVSS Base Score	Severity
Tomcat	CVE-2016-1240	CWE-20	Improper Input Validation	7.8	High
	CVE-2016-6816	CWE-20	Improper Input Validation	7.1	High
	CVE-2017-12617	CWE-434	Unrestricted Upload of File with Dangerous Type	8.1	High
	CVE-2020-1938	CWE-285	Improper Authorization	9.8	Critical
NodeJS	CVE-2017-5941	CWE-502	Deserialization of Untrusted Data	9.8	Critical
Docker	CVE-2019-5736	CWE-78	OS Command Injection	8.6	High
Kernel	CVE-2014-3153	CWE-269	Improper Privilege Management	7.2	High
	CVE-2014-4014	CWE-264	Permissions, Privileges, and Access Controls	6.2	Medium
	CVE-2014-4699	CWE-362	Race Condition	6.9	Medium
	CVE-2016-0728	CWE-416	Use after free	7.8	High
	CVE-2016-5195	CWE-362	Race Condition	7.8	High
	CVE-2016-9793	CWE-119	Improper Restriction of Ops within the Bounds of Buffer	7.8	High
	CVE-2017-1000112	CWE-362	Race Condition	7.0	High
	CVE-2017-16939	CWE-416	Use after free	7.8	High
	CVE-2017-16995	CWE-119	Improper Restriction of Ops within the Bounds of Buffer	7.8	High
	CVE-2017-7308	CWE-681, CWE-787	Incorrect Conversion, Out-of-bounds write	7.8	High
	CVE-2022-0492	CWE-287, CWE-862	Improper Authentication, Missing Authorization	7.8	High

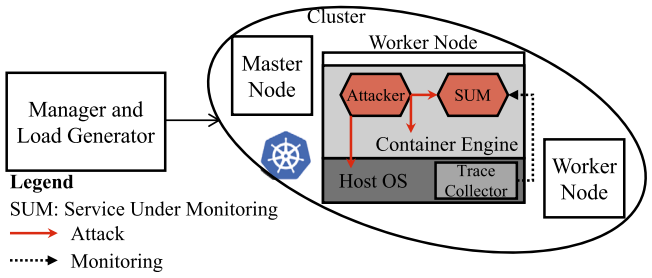


Fig. 12. Overview of the setup used on the experiments.

two worker nodes (see Fig. 12). The load generator and master node machines had an Intel Core i5 with 32 GB of RAM running Ubuntu 18.04, while the worker node machines had an Intel Core i5 with 64 GB of RAM running Ubuntu 16.04.

Table 4
Dataset details about the number of files, their storage space, and recording time.

Data type	Files count	Compressed size (GB)	Uncompressed size (GB)	Recording time (hours)
Training data (benign)	52	359	5,201	696
Validation data (benign)	40	12	157	12
Testing data (mixed)	594	230	3,881	186
Total	686	601	9,239	894

6.2.2. Preparation and execution of the campaign

Data collection, attacks, and data enrichment were automated for repeatability. Deployments were scripted using YAML scripts from public repositories. To accommodate the vulnerabilities elicited (see Table 3), we installed Docker version 18.06.3, used Tomcat version 7.0.57, NodeJS version 10.1.0, and Kernel 4.4.0 with Ubuntu 16.04. Although these versions are a bit old, the systems are still in use today and are representative of technologies commonly found in production software. The information we collect (system calls) has a stable interface across different versions of this software stack, and its stability ensures that our dataset does not suffer from reduced representativeness.

6.3. Characterization of the datasets

Each collection experiment produces a set of files. The main file contains system calls collected from all microservices, including their replicas, forming the monitored application. This file has multiple features per line: event timestamp, container name, thread ID, system call direction ('>' for entry, '<' for exit), system call category (according to sysdig), system call name, and list of arguments. This data was split into files of up to *100GB* each before compression, organized by worker node. Additional files detail execution intervals of data enrichment commands (identifying set and container), and attacks, if applicable. For training data, timestamps are provided for scaling events and *write* probability increase timestamps.

From the client side, different files are collected: a log file of the load generator operation, exception logs for errors, details on failed requests, chronological request details with URLs and statistics, and transaction data. These files offer throughput, response time, percentiles, and performance insight. Data was collected separately for the monitoring and warm-up phases.

Table 4 outlines file specifics containing system call data across applications and data file types. Training datasets consist of 52 files, gathered over eight days per microservice application, incorporating data enrichment commands, scaling scenarios (#1,#3,#5, and #7), and *write* probability increases. This resulted in `5.2TB` (`359GB` compressed) of versatile data for algorithm training across different scenarios and testbeds. The test data follows the testing methodology (see Fig. 11), with `30min` workload execution periods, injected attacks, and data enrichment commands. This results in mixed-data datasets, consisting of benign and malicious events, suitable for algorithm evaluation. The test data comprises `3.9TB` (`230GB` compressed) over 372 runs. Benign validation data, similar to the attack methodology but without attacks, amounts to `157GB` (`12GB` compressed) of system call events from 24 runs. This division of data is a suggestion for the data scientist to use, but they can still perform different divisions using the training datasets available. The key reasoning is that in the current separation, the users have different data enrichment activities performed in training and validation data, which allows the user to tune their approaches in terms of generalization ability, before using the test data, which, as

Table 5
Benchmarking results for stable and scalability case studies.

Testbed	Algorithm	Stable Case Study							Scalability Case Study						
		Metrics				Rank			Metrics				Rank		
		Recall	Prec.	F-M	Marked.	Low	Mod.	High	Recall	Prec.	F-M	Marked.	Low	Mod.	High
TeaStore	BoSC-3	0.683	0.185	0.292	0.184	4	**1**	**1**	0.155	0.944	0.266	0.941	8	3	**1**
	STIDE-3	0.683	0.174	0.277	0.173	6	3	3	0.155	0.944	0.266	0.941	8	3	**1**
	BoSC-4	0.683	0.185	0.292	0.184	4	**1**	**1**	0.164	0.857	0.275	0.854	7	**1**	3
	STIDE-4	0.683	0.164	0.264	0.162	7	4	4	0.170	0.679	0.271	0.675	6	2	4
	BoSC-10	0.537	0.156	0.242	0.154	8	5	5	0.220	0.152	0.180	0.149	5	5	5
	HMM	1.000	0.060	0.113	0.060	**1**	6	7	0.591	0.039	0.074	0.038	4	6	6
	iForest	1.000	0.059	0.111	0.059	2	7	8	1.000	0.035	0.068	0.035	**1**	8	8
	OCSVM	1.000	0.059	0.111	0.059	2	7	8	0.773	0.036	0.069	0.036	2	7	7
	LOF	0.045	0.091	0.061	0.089	9	9	6	0.652	0.032	0.061	0.031	3	9	9
SockShop	BoSC-3	0.406	0.481	0.441	0.479	7	2	2	0.319	0.857	0.465	0.854	5	3	**1**
	STIDE-3	0.438	0.500	0.467	0.498	6	**1**	**1**	0.362	0.829	0.504	0.827	4	**1**	2
	BoSC-4	0.500	0.382	0.433	0.381	4	3	3	0.396	0.667	0.497	0.664	3	2	3
	STIDE-4	0.500	0.361	0.419	0.359	5	4	4	0.255	0.632	0.364	0.629	6	4	4
	BoSC-10	0.250	0.333	0.286	0.331	8	8	5	0.218	0.354	0.270	0.352	7	6	7
	HMM	0.250	0.333	0.286	0.331	8	8	5	1.000	0.101	0.184	0.101	**1**	8	8
	iForest	1.000	0.194	0.324	0.194	3	7	9	0.125	0.375	0.188	0.373	9	7	6
	OCSVM	1.000	0.196	0.327	0.196	2	6	8	0.196	0.500	0.282	0.498	8	5	5
	LOF	1.000	0.202	0.336	0.202	**1**	5	7	0.979	0.101	0.183	0.101	2	9	9
SocialNetwork	BoSC-3	0.679	0.475	0.559	0.474	4	7	6	0.495	0.517	0.506	0.515	7	6	5
	STIDE-3	0.679	0.452	0.543	0.451	5	8	7	0.505	0.511	0.508	0.509	6	5	6
	BoSC-4	0.607	0.630	0.618	0.628	7	4	4	0.604	0.364	0.455	0.362	5	7	9
	STIDE-4	0.607	0.630	0.618	0.628	7	4	4	0.363	0.440	0.398	0.437	8	8	7
	BoSC-10	0.400	0.414	0.407	0.411	9	9	9	0.286	0.371	0.323	0.368	9	9	8
	HMM	1.000	0.444	0.615	0.444	2	6	8	0.708	0.667	0.687	0.666	4	2	2
	iForest	1.000	0.667	0.800	0.667	**1**	**1**	3	0.750	0.563	0.643	0.562	3	4	3
	OCSVM	0.813	0.722	0.765	0.722	3	2	**1**	1.000	0.522	0.686	0.522	**1**	3	4
	LOF	0.625	0.714	0.667	0.713	6	3	2	0.813	0.765	0.788	0.764	2	**1**	**1**

recommended in the machine learning best practices, should never be used for tuning and only for the final results. The values reported here refer to the current state of the datasets. They are still being expanded to more scenarios and testbeds as well as other workloads and attacks.

7. Experimental evaluation

The objective of these experiments is to demonstrate the proposed benchmark and validate its process. Simultaneously, we can evaluate different intrusion detection algorithms in the context of different microservice environment scenarios and compare their effectiveness. The benchmarking results and algorithms ranking are performed in three case studies that focus on different microservice application operating environments, as described in Section 5.

7.1. Stable deployment case study

This case study describes the base case of intrusion detection evaluations, where an algorithm is trained in the same environment in which it is deployed to detect intrusions. That is, the configurations of the environment are the same and it does not face modifications when trying to identify malicious actions. In this case study, the profiled microservice version (on whose data the detection models are trained) is the same as the one used to evaluate the algorithms. We use deployment scenario #1, which considers only one service replica in operation.

The results are shown on the left side of Table 5. We can observe that, across the three testbeds, the ML algorithms (HMM, IForest, OCSVM, and LOF) generally perform better for the low risk tolerance scenario. These detection models detect the attacks performed with higher frequency, but sometimes at the expense of precision. They generally perform worse than STIDE/BoSC for moderate and high tolerance scenarios, except in the case of the SocialNetwork testbed, where the ML algorithms still outperform STIDE/BoSC. This may be due to the fact that the SocialNetwork service produces less data that needs to be processed by the algorithms, and therefore the ML algorithms may be able to produce clearer detection models for this case. In general, all detection models perform better with less data to process, which may indicate that data processing techniques would help to obtain better behavior profiles.

The results also allow us to evaluate the performance of different configurations of the same algorithms. The performance of BoSC decreases as the window size used increases. BoSC with window 10 always ranks below the configuration with window 3, demonstrating the usefulness of the benchmark in testing not only different algorithms but also different configurations of the same algorithms.

7.2. Scalability case study

This case study describes a dynamic case of intrusion detection evaluation, where a model is trained in a deployment configuration that is different from the configuration in which it is used to detect intrusions. That is, the configurations of the environment are dynamic due to scalability and elasticity. The detection model is subject to changes as it attempts to identify malicious behavior. In this case study, the profiled microservice remains the same, in terms of behavior, but there may be a varying number of replicas to increase the capacity to respond to user requests. During testing, the deployment scenarios for each testbed are #3, #5, and #7, which consider different numbers of replicas for the profiled microservice.

A general overview of the results obtained in this case study are presented on the right side of Table 5. Here, we can observe similar trends to the stable deployment case study. The ML algorithms tend to rank better in the low risk tolerance scenarios but are outperformed by STIDE/BoSC in the moderate and high tolerance scenarios. Still, for SocialNetwork, the top ranking detection models still come from ML algorithms. OCSVM and LOF rank better for the three risk tolerance scenarios in this testbed. For SockShop and TeaStore, STIDE/BoSC perform better in the moderate and high risk tolerance scenarios, particularly with smaller window sizes.

Table 6
Detailed benchmark results for the SocialNetwork testbed for the scalability case study.

Testing Scenario	Algorithm	Scalability Case Study Details for SocialNetwork						
		Metrics				Rank		
		Recall	Prec.	F-M	Marked.	Low	Mod.	High
#3	BoSC-3	0.900	0.429	0.581	0.428	4	7	8
	STIDE-3	0.550	0.579	0.564	0.578	8	8	5
	BoSC-4	0.750	0.600	0.667	0.599	7	4	4
	STIDE-4	0.850	0.472	0.607	0.472	6	6	6
	BoSC-10	0.368	0.412	0.389	0.410	9	9	9
	HMM	1.000	0.667	0.800	0.667	2	2	2
	iForest	1.000	0.444	0.615	0.444	3	5	7
	OCSVM	1.000	0.727	0.842	0.727	**1**	**1**	**1**
	LOF	0.875	0.609	0.718	0.608	5	3	3
#5	BoSC-3	0.517	0.517	0.517	0.515	7	6	4
	STIDE-3	0.552	0.516	0.533	0.514	6	5	5
	BoSC-4	0.690	0.339	0.455	0.338	5	7	9
	STIDE-4	0.387	0.414	0.400	0.411	8	8	7
	BoSC-10	0.310	0.375	0.340	0.372	9	9	8
	HMM	1.000	0.727	0.842	0.727	**1**	2	2
	iForest	1.000	0.727	0.842	0.727	**1**	2	2
	OCSVM	1.000	0.471	0.640	0.471	3	4	6
	LOF	0.813	0.929	0.867	0.928	4	**1**	**1**
#7	BoSC-3	0.875	0.304	0.452	0.304	4	5	7
	STIDE-3	0.875	0.304	0.452	0.304	4	5	7
	BoSC-4	0.563	0.305	0.396	0.303	7	7	9
	STIDE-4	0.344	0.440	0.386	0.437	9	8	5
	BoSC-10	0.353	0.333	0.343	0.332	8	9	6
	HMM	1.000	0.727	0.842	0.727	**1**	**1**	**1**
	iForest	0.750	0.600	0.667	0.599	6	3	3
	OCSVM	1.000	0.444	0.615	0.444	3	4	4
	LOF	1.000	0.667	0.800	0.667	2	2	2

A detailed analysis of the SocialNetwork testbed results is presented in Table 6. The results obtained are detailed across each testing deployment configuration. This makes it possible to understand how the detection models perform with a higher or lower number of service active replicas. The general trend remains, and ML algorithms generally perform better across the three risk tolerance scenarios. Still, it is possible to observe that as the number of active service replicas increases, the HMM detection models outperform the others. OCSVM is the most impacted, and its performance decreases as the number of replicas increases. From the top of the ranking in three risk tolerance scenarios for deployment #3, it drops to third and fourth for deployment #7. The STIDE/BoSC algorithms are also impacted by the increase in active replicas, with lower precision and a slight decrease in recall.

7.3. Continuous delivery case study

This case study focuses on a continuous integration and continuous delivery (CI/CD) environment where a detection model is trained on an older version of a microservice and used to detect intrusions in newer versions. This means that the codebase of the microservice changes

Table 7
Model Reusability Index (MRI) results for continuous delivery case study. Older version is v1.3.0 and newer version is v1.3.8.

CD	AI	Algorithm	RI	MRI	Rank
0.997	0.802	BoSC-3	0.863	0.861	**1**
		STIDE-3	0.845	0.849	2
		BoSC-4	0.798	0.819	3
		STIDE-4	0.755	0.791	4
		BoSC-10	0.612	0.698	5
		HMM	0.223	0.445	6
		iForest	0.108	0.370	9
		OCSVM	0.223	0.445	6
		LOF	0.223	0.445	6

Table 8
Detection results for continuous delivery case study. The tested detection models were trained on data from v1.3.0 (older version).

Algorithm	Models Tested in v1.3.0				Models Tested in v1.3.8				Variation Rank				MRI
	Recall	Prec.	F-M	Marked.	Recall	Prec.	F-M	Marked.	Low	Mod.	High	Avg	Rank
BoSC-3	0.121	1.000	0.216	0.997	0.683	0.182	0.287	0.180	2	3	8	**1**	**1**
STIDE-3	0.121	1.000	0.216	0.997	0.683	0.177	0.281	0.176	3	4	9	5	2
BoSC-4	0.424	0.146	0.217	0.144	0.683	0.183	0.289	0.182	5	2	**1**	2	3
STIDE-4	0.273	0.161	0.202	0.158	0.683	0.172	0.275	0.170	4	**1**	3	2	4
BoSC-10	0.303	0.147	0.198	0.145	0.488	0.167	0.248	0.164	6	5	2	4	5
HMM	0.273	0.154	0.197	0.152	1.000	0.061	0.114	0.061	**1**	8	7	5	6
iForest	1.000	0.102	0.185	0.102	1.000	0.061	0.114	0.061	7	6	4	7	9
OCSVM	1.000	0.104	0.189	0.104	1.000	0.061	0.115	0.061	8	7	5	8	6
LOF	0.682	0.121	0.205	0.120	0.045	0.042	0.043	0.039	9	9	6	9	6

during the lifetime of the model. As the microservice changes, so does its behavior, and detection models must be able to handle such changes. For training, we use data from the version `v1.3.0` of TeaStore. During testing, we use two versions of the service: `v1.3.0` and `v1.3.8`; both in deployment scenario #1.

For this case study, we compute value of the Model Reusability Index (MRI) metric (described in Section 3.3) proposed in previous research work (Flora and Antunes, 2024). Table 7 presents the results of the MRI calculation. This metric allows users to gain insight into the reusability of intrusion detection models. It provides the likelihood that detection models will be reused in different versions of the microservice. According to the results obtained, the models trained on `v1.3.0` that are more likely to be reused in `v1.3.8` are those produced by STIDE/BoSC, with smaller window sizes having slightly better MRI results.

To compare the MRI results with the detection results, we use the benchmark's procedure to test the detection models trained on `v1.3.0` in both the older (`v1.3.0`) and the newer (`v1.3.8`) versions of the microservice. The results are presented in Table 8. After calculating the metrics, we calculated the variation of each metric for each detection model. Based on the metric variation, we create the ranking for each risk tolerance scenario, and the average ranking position for each detection model to compare it with the MRI ranking position (see Table 8).

This allows us to validate the MRI results since the metric does not take into account the different risk tolerance scenarios. The two rankings demonstrate very similar positions for each detection model. The models produced by STIDE/BoSC rank higher than the ML algorithms, with smaller window sizes taking the upper hand. This confirms that the benchmark can be applied to different configurations to gain insight into the likelihood of reusability of detection models.

7.4. Properties discussion

Here, we discuss how the instantiation of the benchmarking approach demonstrates the benchmark and workload properties defined in Section 2.3.

The instantiation and successful use of the benchmarking approach demonstrates its *relevance*. There is currently no benchmark that addresses the challenges of microservices for intrusion detection. This benchmark allows its users to evaluate and compare how different IDSs perform when faced with dynamic microservice environments. It also provides evidence of the benchmark's *portability*, which was shown during the experimentation, as it can be easily applied to different types of algorithms and configurations, from ML algorithms to statistical approaches and pattern-based techniques, which use different measurement units and techniques to process the data.

The benchmark uses well-known state-of-the-art metrics, whose results are easy to interpret and analyze. This guarantees the *simplicity* of the benchmark and contributes to the credibility of the results obtained. In addition, the execution of the benchmark is simple and *economical*, it has only two time-consuming steps: training the models, only once per configuration, and testing them in each scenario to obtain the measures and compute the results. Multiple runs of the benchmark will yield statistically similar results because the workload has been collected and is not changed between runs, which contributes to making it a *repeatable and reproducible* benchmark.

The different case studies also validate the properties of the workload (dataset). The workload is composed of three testbed applications with state-of-the-art technologies and operating modes that follow the architectural pattern under focus to ensure its *representativeness*, *relevance*, broad *coverage*, and *comprehensiveness*. It also provides labeled attacks to compare the IDSs results with a *ground truth*, and calculate the measures. As demonstrated by the different case studies, the workload is *configurable*, with multiple scenarios, and *scalable*, with different levels of complexity (e.g. scalability and continuous integration) that the user can choose.

8. Conclusions and future work

The utilization of microservices to develop applications enables the use of on-demand resources and scalable and elastic behavior by adapting to user demand. It is necessary to ensure that the security of the applications is not compromised by the lack of proper defense mechanisms.

In this paper, we propose a benchmarking approach to evaluate and compare diverse intrusion detection techniques in the context of microservices. We propose a methodology and use it to generate representative datasets with 900 h of recorded data that overcome the limitations of currently available datasets that do not have the concerns and characteristics of microservices in their genesis. Our datasets also consider a data enrichment process to help reduce overfitting when training the detection models. Furthermore, the benchmark is instantiated and perform an experimental evaluation of six intrusion detection algorithms with different complexity scenarios that are common in microservice deployments and very relevant for the mechanisms that intend to detect intrusions to maintain the security level of the applications.

The experiments illustrate the wide range of application and configuration of the benchmark to evaluate and compare different algorithms. It also shows its usefulness for researchers and practitioners dealing with intrusion detection in these environments, with representative datasets and meaningful results. The results show that ML algorithms generally perform better on testbeds that produce less data for processing while STIDE/BoSC can handle more information, process it effectively, and still achieve a high detection rate. The detection models produced by STIDE/BoSC are easier to reuse across microservice releases; nevertheless, they need to be improved to ensure satisfactory detection rates.

Future work includes maintaining and updating the datasets to include different testbeds and emerging attacks observed in microservices. The datasets and benchmark presented will allow research into techniques that address intrusion detection in scalable microservice applications and multi-target attacks. These could protect the system from more complex and capable attackers and involves detecting these events earlier by studying the relationships between services.

CRediT authorship contribution statement

José Flora: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Data curation, Conceptualization. **Nuno Antunes:** Writing – review & editing, Writing – original draft, Validation, Supervision, Resources, Methodology, Investigation, Funding acquisition, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Dataset is available at Zenodo: https://doi.org/10.5281/zenodo.10655126.

Acknowledgments

This work was supported by the Portuguese Foundation for Science and Technology (FCT) through the Ph.D. grant 2020.05145.BD. Content produced within the scope of the Agenda "NEXUS - Pacto de Inovaçáo - Transiçáo Verde e Digital para Transportes, Log/'istica e Mobilidade", financed by the Portuguese Recovery and Resilience Plan (PRR), with no. C645112083-00000059 (investment project no. º 53) This work is also funded by the FCT - Foundation for Science and Technology, I.P./MCTES through national funds (PIDDAC), within the scope of CISUC R&D Unit - UIDB/00326/2020 or project code UIDP/00326/2020. It was also developed in the context of SPEC RG Security Benchmarking (http://research.spec.org).

References

Abed, A.S., Clancy, C., Levy, D.S., 2015. Intrusion detection system for applications using linux containers. In: International Workshop on Security and Trust Management. Springer, pp. 123–135.

Antunes, N., Vieira, M., 2010. Benchmarking vulnerability detection tools for web services. In: 2010 IEEE International Conference on Web Services. IEEE, pp. 203–210.

Antunes, N., Vieira, M., 2015. On the metrics for benchmarking vulnerability detection tools. In: 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. IEEE, pp. 505–516.

Araujo, I., Antunes, N., Vieira, M., 2023. Evaluation of Machine Learning for Intrusion Detection in Microservice Applications. In: 12th Latin-American Symposium on Dependable and Secure Computing. ACM, pp. 126–135.

Bace, R., Mell, P., 2001. NIST Special Publication on Intrusion Detection Systems. Tech. Rep., BOOZ-ALLEN AND HAMILTON INC MCLEAN VA.

Bharadwaja, S., Sun, W., Niamat, M., Shen, F., 2011. Collabra: A Xen hypervisor based collaborative intrusion detection system. In: 2011 Eighth International Conference on Information Technology: New Generations. pp. 695–700.

Breunig, M.M., Kriegel, H.-P., Ng, R.T., Sander, J., 2000. LOF: identifying density-based local outliers. In: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data. pp. 93–104.

Brown, K., 2022. Linux basic health check commands. linuxconfig.org, linuxconfig.org/linux-basic-health-check-commands.

CAIDA, 2007. The CAIDA DDoS attack 2007 dataset. CAIDA URL https://www.caida.org/catalog/datasets/ddos-20070804_dataset/.

Cavalcanti, M., Inacio, P., Freire, M., 2021. Performance evaluation of container-level anomaly-based intrusion detection systems for multi-tenant applications using machine learning algorithms. In: The 16th International Conference on Availability, Reliability and Security. ACM, pp. 1–9.

Chandola, V., Banerjee, A., Kumar, V., 2009. Anomaly detection: A survey. ACM Comput. Surv. 41 (3), 1–58.

Chang, H., Kodialam, M., Lakshman, T., Mukherjee, S., 2019. Microservice fingerprinting and classification using machine learning. In: IEEE 27th International Conference on Network Protocols. ICNP, pp. 1–11.

Chen, W.-H., Hsu, S.-H., Shen, H.-P., 2005. Application of SVM and ANN for intrusion detection. Comput. Oper. Res. 32 (10), 2617–2634.

CISA, 2023. 2022 Top routinely exploited vulnerabilities. CISA, cisa.gov/news-events/cybersecurity-advisories/aa23-215a.

Cloud Native Computing Foundation, 2022. Kubernetes. Cloud Native Computing Foundation, URL http://kubernetes.io.

Creech, G., Hu, J., 2014. A semantic approach to host-based intrusion detection systems using contiguousand discontiguous system call patterns. IEEE Trans. Comput. 63 (4), 807–819.

CWE, 2023. 2023 CWE top 25 most dangerous software weaknesses. CWE, cwe.mitre.org/top25/archive/2023/2023_top25_list.html.

Dae-Ki Kang, Fuller, D., Honavar, V., 2005. Learning classifiers for misuse and anomaly detection using a bag of system calls representation. In: Proceedings from the Sixth Annual IEEE Systems, Man and Cybernetics (SMC) Information Assurance Workshop, 2005. IEEE, pp. 118–125.

DeathStarBench, 2023. Social network microservices. github.com. github.com/delimitrou/DeathStarBench.

Dempsey, K.L., Chawla, N.S., Johnson, L.A., Johnston, R., Jones, A.C., Orebaugh, A.D., Scholl, M.A., Stine, K.M., 2011. Information Security Continuous Monitoring (ISCM) for federal information systems and organizations. Tech. Rep. NIST SP 800-13, National Institute of Standards and Technology.

DevOpsArticle, 2021. Check linux systems health with 10 powerful commands 2021. devopsarticle.com, devopsarticle.com/check-linux-systems-health-with-10-powerful-commands-2021.

dotnet, 2023. eShop reference application. github.com github.com/dotnet/eShop.

Dragoni, N., Giallorenzo, S., Lafuente, A.L., Mazzara, M., Montesi, F., Mustafin, R., Safina, L., 2017. Microservices: Yesterday, Today, and Tomorrow. In: Present and Ulterior Software Engineering. Springer International Publishing, pp. 195–216.

Dragoni, N., Lanese, I., Larsen, S.T., Mazzara, M., Mustafin, R., Safina, L., 2018. Microservices: How to Make Your Application Scale. In: Perspectives of System Informatics, vol. 10742, Springer International Publishing, pp. 95–104.

El Khairi, A., Caselli, M., Knierim, C., Peter, A., Continella, A., 2022. Contextualizing system calls in containers for anomaly-based intrusion detection. In: Proceedings of the 2022 on Cloud Computing Security Workshop. ACM, pp. 9–21.

Flora, J., Antunes, N., 2019. Studying the applicability of intrusion detection to multi-tenant container environments. In: 2019 15th European Dependable Computing Conference. EDCC, pp. 133–136.

Flora, J., Antunes, N., 2024. Towards a metric for reuse of microservice intrusion detection models. In: 2024 19th European Dependable Computing Conference. EDCC, pp. 161–164.

Flora, J., Gonçalves, P., Antunes, N., 2020. Using attack injection to evaluate intrusion detection effectiveness in container-based systems. In: 25th IEEE Pacific Rim International Symposium on Dependable Computing. pp. 60–69.

Flora, J., Gonçalves, P., Antunes, N., 2023a. Intrusion detection for scalable and elastic microservice applications. In: 28th IEEE Pacific Rim International Symposium on Dependable Computing. pp. 39–45.

Flora, J., Teixeira, M., Antunes, N., 2023b. μDetector: Automated intrusion detection for microservices. In: 2023 IEEE International Conference on Software Analysis, Evolution and Reengineering. SANER, pp. 748–752.

Forrest, S., Hofmeyr, S.A., Somayaji, A., Longstaff, T.A., 1996. A sense of self for unix processes. In: IEEE Symposium on Security and Privacy. IEEE Computer Society Press, pp. 120–128.

Fowler, M., Lewis, J., 2014. Microservices. martinfowler.com. URL martinfowler.com/articles/microservices.html.

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

German Zenetti, 2022. Customer research on Amazon contributes to the offline path to purchase. Amazon Customer Research, advertising.amazon.com/en-gb/library/research/customers-purchase-decisions.

Gray, J., 1993. The Benchmark Handbook for Database and Transaction Systems, second ed. Morgan Kaufmann.

Grimmer, M., Röhling, M.M., Kreusel, D., Ganz, S., 2019. A modern and sophisticated host based intrusion detection data set. IT-Sicherheit als Voraussetzung für eine erfolgreiche Digitalisierung 135–145.

Hannousse, A., Yahiouche, S., 2021. Securing microservices and microservice architectures: A systematic mapping study. Comp. Sci. Rev. 41, 100415.

Hofmeyr, S.A., Forrest, S., Somayaji, A., 1998. Intrusion detection using sequences of system calls. J. Comput. Secur. 6 (3), 151–180.

Islam, M.S., Khreich, W., Hamou-Lhadj, A., 2018. Anomaly detection techniques based on kappa-pruned ensembles. IEEE Trans. Reliab. 67 (1), 212–229.

Kerrisk, M., 2023. Linux man pages. man7.org URL https://man7.org/linux/man-pages/. (Accessed 24 October 2023).

Kistowski, J.v., Arnold, J.A., Huppler, K., Lange, K.-D., Henning, J.L., Cao, P., 2015. How to build a benchmark. In: Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering. ICPE '15, Association for Computing Machinery, pp. 333–336.

Kistowski, J.v., Herbst, N., Kounev, S., 2014. LIMBO: a tool for modeling variable load intensities. In: Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering. pp. 225–226.

Kong, 2020. 2020 Digital Innovation Benchmark. Tech. Rep., Kong Inc., p. 18, URL https://konghq.com/resources/digital-innovation-benchmark-2020/.

Kubernetes, 2020. Horizontal pod autoscaler. kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale.

Lin, X., Lei, L., Wang, Y., Jing, J., Sun, K., Zhou, Q., 2018. A measurement study on Linux container security: Attacks and countermeasures. In: Proceedings of the 34th Annual Computer Security Applications Conference. Association for Computing Machinery, pp. 418–429.

Liu, F.T., Ting, K.M., Zhou, Z.-H., 2008. Isolation forest. In: 2008 Eighth IEEE International Conference on Data Mining. IEEE, pp. 413–422.

Madden, N., 2020. API Security in Action. Simon and Schuster.

Mell, P., Grance, T., 2011. The NIST Definition of Cloud Computing. Computer Security Division, Information Technology Laboratory, National.

Milenkoski, A., Payne, B.D., Antunes, N., Vieira, M., Kounev, S., Avritzer, A., Luft, M., 2015a. Evaluation of intrusion detection systems in virtualized environments using attack injection. In: Research in Attacks, Intrusions, and Defenses: 18th International Symposium. RAID 2015, Kyoto, Japan, Springer, pp. 471–492.

Milenkoski, A., Vieira, M., Kounev, S., Avritzer, A., Payne, B.D., 2015b. Evaluating computer intrusion detection systems. ACM Comput. Surv. 48 (1), 1–41.

MIT Lincoln Laboratory, 1998. 1998 DARPA intrusion detection evaluation dataset. MIT Lincoln Laboratory URL https://www.ll.mit.edu/r-d/datasets/1998-darpa-intrusion-detection-evaluation-dataset.

Mohan, V., Othmane, L.B., 2016. Secdevops: Is it a marketing buzzword?-mapping research on security in devops. In: 2016 11th International Conference on Availability, Reliability and Security. ARES, IEEE, pp. 542–547.

Moustafa, N., Slay, J., 2015. UNSW-NB15: a comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set). In: 2015 Military Communications and Information Systems Conference. MilCIS, pp. 1–6.

Musa, J.D., 1996. The Operational Profile. In: Reliability and Maintenance of Complex Systems. Springer Berlin Heidelberg, pp. 333–344.

Newman, S., 2015. Building Microservices: Designing Fine-Grained Systems. "O'Reilly Media, Inc.".

Nunes, P., Medeiros, I., Fonseca, J.C., Neves, N., Correia, M., Vieira, M., 2018. Benchmarking static analysis tools for web security. IEEE Trans. Reliab. 67 (3), 1159–1175.

Oğur, H.B., 2022. A Novel Container Attacks Data Set for Intrusion Detection. Middle East Technical University.

Pereira-Vale, A., Márquez, G., Astudillo, H., Fernandez, E.B., 2019. Security mechanisms used in microservices-based systems: A systematic mapping. In: 2019 XLV Latin American Computing Conference. CLEI, pp. 01–10.

Ran, 2023. 10 Linux commands to monitor your system's health. unixmen.com, unixmen.com/10-linux-commands-to-monitor-your-systems-health.

Röhling, M., Grimmer, M., Kreubel, D., Hoffmann, J., Franczyk, B., 2019. Standardized container virtualization approach for collecting host intrusion detection data. In: Federated Conference on Computer Science and Information Systems. FedCSIS, pp. 459–463.

Russell, S.J., 2010. Artificial intelligence a modern approach. Pearson Education, Inc.

Sever, Y., Dogan, A.H., 2023. A kubernetes dataset for misuse detection. ITU J. Future Evolv. Technol. 4 (2), 383–388.

Sharafaldin, I., Habibi Lashkari, A., Ghorbani, A.A., 2018. Toward generating a new intrusion detection dataset and intrusion traffic characterization. In: Proceedings of the 4th International Conference on Information Systems Security and Privacy. SCITEPRESS - Science and Technology Publications, pp. 108–116.

Shiravi, A., Shiravi, H., Tavallaee, M., Ghorbani, A.A., 2012. Toward developing a systematic approach to generate benchmark datasets for intrusion detection. Comput. Secur. 31 (3), 357–374.

Souppaya, M., Morello, J., Scarfone, K., 2017. Application Container Security Guide. Tech. Rep. NIST SP 800-190, National Institute of Standards and Technology.

Srinivasan, S., Kumar, A., Mahajan, M., Sitaram, D., Gupta, S., 2019. Probabilistic real-time intrusion detection system for docker containers. In: Security in Computing and Communications. Springer Singapore, pp. 336–347.

Strom, D., 2016. Why runtime application self-protection is critical for next generation app security.

Strom, B.E., Applebaum, A., Miller, D.P., Nickels, K.C., Pennington, A.G., Thomas, C.B., 2018. Mitre Att&ck: Design and Philosophy. Technical Report, The MITRE Corporation.

Sultan, S., Ahmad, I., Dimitriou, T., 2019. Container security: Issues, challenges, and the road ahead. IEEE Access 7, 52976–52996. http://dx.doi.org/10.1109/ACCESS.2019.2911732.

Sysdig, Inc, 2022. Sysdig. sysdig.com URL https://sysdig.com/. (Accessed 14 January 2024).

Tavallaee, M., Bagheri, E., Lu, W., Ghorbani, A.A., 2009. A detailed analysis of the KDD CUP 99 data set. In: 2009 IEEE Symposium on Computational Intelligence for Security and Defense Applications. pp. 1–6.

The Spring PetClinic Community, 2016. Spring PetClinic. The Spring PetClinic Community, spring-petclinic.github.io.

The Vulnerabilities Hub, 2021. Pre-built vulnerable environments based on docker-compose. github.com github.com/vulhub/vulhub.

UNM, 1999. UNM intrusion detection dataset. URL https://web.archive.org/web/20220324084030/https://www.cs.unm.edu/~immsec/systemcalls.htm.

Vieira, M., Madeira, H., Sachs, K., Kounev, S., 2012. Resilience benchmarking. In: Resilience assessment and evaluation of computing systems. Springer, pp. 283–301.

von Kistowski, J., Eismann, S., Schmitt, N., Bauer, A., Grohmann, J., Kounev, S., 2018. 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, IEEE, pp. 223–236.

Warrender, C., Forrest, S., Pearlmutter, B., 1999. Detecting intrusions using system calls: alternative data models. In: Proceedings of the 1999 IEEE Symposium on Security and Privacy. IEEE Comput. Soc, pp. 133–145.

Weaveworks, 2017. Sock Shop : A microservice demo application. github.com. github.com/microservices-demo/microservices-demo.

Zhou, X., Peng, X., Xie, T., Sun, J., Xu, C., Ji, C., Zhao, W., 2018. Benchmarking Microservice Systems for Software Engineering Research. In: 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion. ICSE-Companion, pp. 323–324.

José Flora is a Ph.D. student at the University of Coimbra, Portugal, in the PhD in Informatics Engineering. His research interests include information and software security, particularly software containers and microservices security, intrusion detection and intrusion tolerance, and security services for cloud computing. Flora received an M.Sc. in Information Security in 2019 from the University of Coimbra. Contact him at jeJlora@dei.uc.pt.

Nuno Antunes is an Assistant Professor at the University of Coimbra, where he received his PhD in Information Science and Technology in 2014. He has been with the Centre for Informatics and Systems of the University of Coimbra (CISUC) since 2008. His expertise includes testing, fault injection, vulnerability injection and benchmarking, applied to the assessment of the dependability and security of intelligent systems, virtualized environments, intrusion detection systems, web services, web and mobile applications, and data management systems. He is a member of the IEEE Computer Society. Contact him at nmsa@dei.uc.pt.