Developing Attack Detection Models for Microservice Applications: A Comprehensive Framework and Its Illustration and Validation on DoS Attacks

Jessica Castro, Nuno Laranjeiro, *Member, IEEE*, Katerina Goseva-Popstojanova, *Senior Member, IEEE*, and Marco Vieira, *Member, IEEE*

***Abstract*—Microservice architectures offer scalability and flexibility, but due to their distributed nature and complex service structures, raise new security challenges, particularly in detecting DoS attacks. Although addressing these challenges calls for innovative attack detection approaches, developing effective solutions requires large-scale experiments and data collection to create representative datasets. This paper proposes a comprehensive framework to support research on the cybersecurity of microservice applications and the development of different methods to detect cyberattacks. The framework comprises two modules: (i) a data generation module that contains the components necessary to create datasets that reflect the behavior of microservices under attack and (ii) a model development and evaluation module suitable for different methods for detecting attacks on microservices. The framework is illustrated and validated by generating realistic high- and low-volume DoS attack data and developing models using supervised and unsupervised Machine Learning (ML) algorithms and a method based on Logic Scoring of Preference (LSP). The results indicate that supervised ML models have the best classification performance, especially with the XGBoost algorithm. Even though unsupervised ML and LSP models have worse performance, they can be used when the attack data are not available or are costly to generate.**

***Index Terms*—Microservice, security, denial of service attacks, attack detection, machine learning, logic scoring of preference.**

Received 26 December 2024; revised 6 July 2025; accepted 10 July 2025. Date of publication 17 July 2025; date of current version 4 November 2025. This work was supported in part by FCT - Fundação para a Ciência e Tecnologia grant https://doi.org/10.54499/2023.00207.BD. Also partially financed by national funds by FCT - Fundação para a Ciência e a Tecnologia, I.P., in part the framework of the Project under Grant UIDB/00326/2025 and Grant UIDP/00326/2025. Also partially supported by Project "NEXUS - Pacto de Inovação - Transição Verde e Digital para Transportes, Logística e Mobilidade", and in part by the Portuguese Recovery and Resilience Plan (PRR), under Grant C645112083-00000059 (investment project no. 53). *(Corresponding author: Jessica Castro.)*

Jessica Castro and Nuno Laranjeiro are with the University of Coimbra, CISUC/LASI - Centre for Informatics and Systems of the University of Coimbra, Department of Informatics Engineering, 3004-531 Coimbra, Portugal (e-mail: jesscmaciel@dei.uc.pt).

Katerina Goseva-Popstojanova is with the Lane Department of Computer Science and Electrical Engineering, West Virginia University, Morgantown WV 26506 USA.

Marco Vieira is with the Department of Computer Science, University of North Carolina at Charlotte, Charlotte NC 28223 USA.

Digital Object Identifier 10.1109/TDSC.2025.3590197

I. Introduction

THE adoption of microservice applications has grown significantly in recent years [1]. A microservices architecture allows developers to create highly scalable, flexible, and manageable systems. In this architecture, a single application consists of an orchestration of multiple small, independent services [2], where each service implements a distinct function and uses diverse computational resources. Some services are more CPU-intensive, while others tend to use more memory or storage [3]. The individual services can be implemented using different technologies and can be deployed, updated, and deleted independently [1]. Despite the popularity, this architecture introduces serious security challenges due to its distributed nature and service heterogeneity.

The trend of deploying microservices has been accompanied by a notable increase in the number of cyberattacks, in particular, Denial of Service (DoS) attacks [2], [4]. For these applications, collecting and analyzing data is of crucial importance, as early and effective attack detection is vital to allow protection or mitigation measures to prevent such attacks from causing damage to the system. However, analyzing the collected data to detect attacks in microservices is challenging due to the complexity of the architecture and the huge volume of data generated by multiple services [3], [5].

An attack detection mechanism (e.g., a model) for microservice applications must be capable of addressing key aspects, such as considering the specific architecture of the application and configuration (e.g., reaction to increasing loads), the occurrence of genuine peaks in client loads, and the potential presence of attacks. A key challenge is the lack of appropriate datasets for developing new solutions (e.g., for training Machine Learning (ML) models) [6], [7]. As a result, developing novel models commonly requires performing large-scale experiments and collecting data to create datasets.

In this paper, we propose a comprehensive framework designed to support new research on cybersecurity of microservice applications, especially on attack detection. Our proposal is set in the context of the OECD (Organization for Economic Cooperation and Development) framework [8] and is composed of two modules: i) a *data generation* module that defines the

required components to generate realistic attack data and aims at addressing the lack of significant datasets for microservice application research; and ii) a *model development and evaluation module* that focuses on supporting the definition of novel attack detection models using different types of methods (e.g., ML, multi-criteria decision-making) and their evaluation in the scope of microservice applications.

The framework is illustrated and validated on DoS attacks through an experimental campaign. To generate data, we deployed TeaStore [9], a representative microservice benchmark used in prior research with similar contexts [5], [10], the Hulk tool https://github.com/grafov/hulk to perform high-volume DoS attacks, and Torshammer https://sourceforge.net/projects/torshammer/ to perform low-volume DoS attacks. Regarding the model development and evaluation, we explore two methods, one based on ML algorithms (both supervised and unsupervised) and the other based on Multi-Criteria Decision Making (MCDM), namely the Logic Scoring of Preference (LSP) method. Results show the usefulness of the framework and the effectiveness of the models across various scenarios.

The main contributions of this paper are as follows:

- We propose the first comprehensive framework for generating realistic attack datasets and supporting the development and evaluation of attack detection models tailored to microservice applications.
- We conduct an extensive experimental campaign to illustrate and validate the proposed framework using low-volume and high-volume DoS attacks. Additionally, we provide comprehensive instructions to facilitate the reproduction of results and the use of the generated data.
- For the first time, in a single work, we develop and evaluate supervised and unsupervised ML models along with LSP-based models for the detection of DoS attacks on microservice applications.
- We provide the first thorough comparative analysis of supervised ML, unsupervised ML, and LSP methods, highlighting their strengths and limitations for DoS detection in the context of microservices applications.
- Based on our findings, we provide a set of actionable recommendations for both practitioners and researchers.

In our prior work, we explored basic elements needed for developing DoS attack detection in microservices, with each study contributing with different insights. In [11], we analyzed monitored data in a microservices context and started the preliminary identification of the features affected by DoS attacks using some supervised ML models, serving as an initial exploration into data features that could be relevant to attack detection. In [12], we examined the LSP method as a possible way to detect attacks in microservices applications, assessing how modifications in LSP components could affect detection models, with a focus only on high-volume DoS attacks. Lastly, in [13], we set up an experimental testbed for DoS attack data generation, detailing basic aspects like environment configurations and attack tool selection.

This paper generalizes and extends our previous works by proposing the first comprehensive framework for developing and assessing attack detection models of different types. Unlike [11], which was limited to supervised ML and used a minimal dataset, this work uses both supervised and unsupervised algorithms to compare detection effectiveness across techniques, including deep learning. Furthermore, building on [12], in this paper, we develop LSP models that detect both high- and low-volume DoS attacks and explore model transferability for a unified detection approach. Also, unlike the setup-oriented perspective in [13], the proposed framework offers detailed guidelines for each component, including workload profiles, enabling the creation of realistic datasets that address the specific challenges in DoS detection for microservices.

The paper is organized as follows. Section II reviews related work. Section III presents the framework, with Section IV detailing data generation. Sections V and VI cover model development and evaluation via ML and LSP. Section VII offers implications and recommendations, Section VIII addresses threats to validity, and Section IX concludes.

II. Related Work

Many studies have addressed gaps in monitoring microservices and detecting attacks or anomalies [1], but these often face the challenge of creating their own data as there is a lack of datasets with representative attack data specific to the microservices context. Baarzi et al. [14] proposed an application-agnostic, non-intrusive, and unsupervised approach to protect microservice systems against DoS attacks at the application level. Experiments emulated both the attacks and applications. They analyzed historical data on resource use (CPU and memory) of different applications under various load levels with legitimate users. Cao et al. [7] generated a dataset to define learning state machines for identifying and detecting attacks and anomalies in Kubernetes clusters. The paper, however, offers limited information on the data generation.

Several works have explored monitoring metrics. Du et al. [15] injected faults to determine relevant metrics for anomaly detection in containers. Samir and Pahl [16] also monitored the container's CPU and memory usage to detect and locate abnormalities using Spearman's rank correlation coefficient and Hierarchical Hidden Markov Models (HHMM). Grohmann et al. [5] attempted to predict the performance degradation of the system by collecting several performance metrics (such as CPU and memory usage and the throughput of I/O devices) and using different supervised machine learning techniques to assess their approach. While these studies contribute to understanding resource-based anomalies, they tend to focus on faults instead of deliberate attack scenarios.

Traditional Application Performance Management tools such as Prometheus (prometheus.io), Grafana (grafana.com), and Dynatrace (dynatrace.com) are adopted widely for monitoring and alerting in production microservice environments. However, they are not designed for experimental research, as they lack native support for attack injection, realistic workload generation, and systematic model evaluation. While such tools could be integrated into our framework for monitoring and data collection, they would require substantial augmentation to support rigorous, repeatable security research.

Several works have focused on trace analysis for anomaly or failure detection in microservice applications. For example, Panahandeh et al. [17] proposed a practical anomaly detection approach using distributed tracing and profiling data to

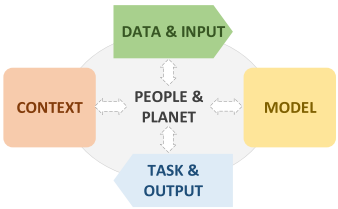

Fig. 1. OECD framework. Adapted from [8].

build a Context Propagation Graph (CPG). Chen et al. [18] proposed DAM (Deep Attentive anomaly detection with Multimodal data), an unsupervised anomaly detection approach for microservice systems. Wang et al. [19] proposed an approach based on multimodal ML, so called Multi-modal data and Multi-feature extraction (MADMM), that detects injected failures. These works [17], [18], [19] represent important advances in anomaly detection, yet they do not provide infrastructure for synthetic data generation, attack specification, and systematic comparison of detection models.

Other studies, such as [20], have begun to explore data generation and benchmarking for intrusion detection. However, these are generally focused on low-level or network-based intrusions. In contrast, our work targets attack detection at the application layer and supports configuring user behavior, workload profiles, attack profiles, and model development.

Although there are approaches to detect anomalies based on CPU usage, memory consumption, and performance degradation, a general framework that supports research on attack detection in microservices does not exist. This paper aims to fill this gap by proposing a framework that includes a module to generate data in a microservice scenario and another module to develop novel attack detection approaches.

Our aim is not to propose a specific detection approach; instead, we develop a modular, experiment-driven framework that allows building, evaluating, and refining different detection approaches in controlled and reproducible conditions.

III. Proposed Framework

The *OECD Framework for the Classification of Artificial Intelligence (AI) Systems* [8] served as a foundation for defining our framework. OECD helped structure and define the essential parts of the framework, making it relevant and comprehensive for addressing the challenges in detecting attacks in microservice applications. The OECD Framework has five key high-level dimensions (see Fig. 1) that reflect in our framework (Fig. 2) that consists of two modules: (A) **Data generation** and (B) **Model development and evaluation**.

Module (A) addresses three dimensions:

- *People & Planet* covers the actors impacted by the system. In our framework, it refers to the users and attackers of a microservice application. This is captured by the Workload Execution (1) and Attack Injection (2), which emulate user behavior and attacks, respectively.
- *Context* encompasses the economic and sectoral environment of the system. For microservice applications, this relates to the application's criticality, the computation capacity to deal with attack detection and the type of attack it must handle. The Microservice Application development is outside of the scope of our framework (3), thus the framework can adapt to different contexts.
- *Data & Input* focuses on the data the model uses to construct a representation of the environment. In our framework, data collection is essential for developing effective attack detection models. Monitoring (4) and Data Collection (5) collect runtime data from the application under regular workload and attack conditions. These data are then processed through the Query & Preprocessing (6) component, preparing them for model development.

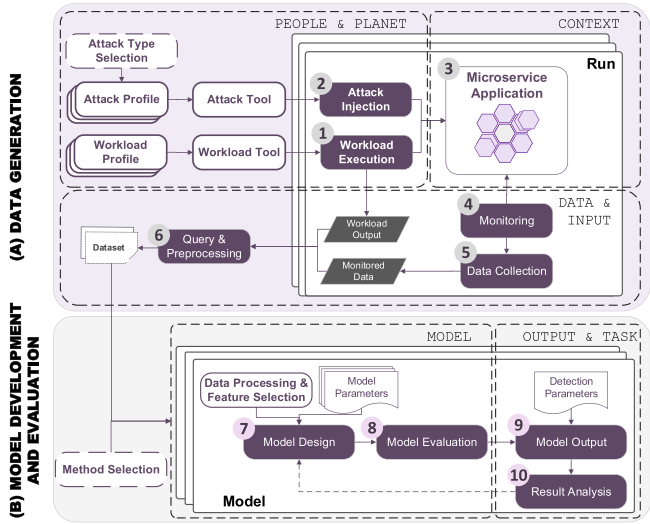

Fig. 2. The proposed framework design.

Module (B) addresses the remaining dimensions:

- *Model:* This dimension involves the processes and methods used in the system. In our framework, the Model Design (7) and Model Evaluation (8) components cover the development of attack detection models.
- *Task & Output:* Refers to the specific tasks the system performs and the results it produces. The Model Output (9) and Result Analysis (10) components are responsible for analyzing the attack detection outcomes, ensuring that the model meets its objective of identifying attacks.

A. Data Generation

Workload Execution (1) is responsible for emulating the behavior of real users interacting with the microservice application. The workload is generated using a *Workload Tool* that executes a *Workload Profile*, which has two parts:

a) *Workload Intensity Specification:* Responsible for defining the volume and intensity of the workload over time.
b) *User Behavior Modeling:* Defines the sequence of actions and the specific pages the users interact with, simulating realistic user behavior.

The strategy to specify the workload intensity depends on the availability of load data and the research objectives. We consider the following three strategies:

1) *Real workloads* that are derived from actual workload data collected from the target application, providing an accurate representation of the intensity. These workloads reflect the number of users and requests observed in the monitored system, but are often unavailable.
2) *Realistic emulated workloads* that are designed to closely mimic typical user intensity over time, incorporating adjustments to fit the experimental environment. These emulate real workloads collected from other systems but can be adapted or scaled to suit the research needs.
3) *Synthetic workloads* that are entirely constructed based on predefined parameters, allowing for controlled experimentation. These workloads offer flexibility, enabling researchers to model various usage scenarios.

When specifying the intensity for realistic emulated and synthetic workloads, specific patterns can be identified or constructed, respectively. These patterns include [21]: the *gentle type*, where the load remains constant or exhibits minimal variation over time; the *burst type*, characterized by one or more sudden spikes in load; the *rise type*, where the load gradually increases; and the *decline type*, where the load gradually decreases. Workload intensity can follow a single pattern or a combination of patterns, e.g., it might start with a rise, feature bursts of activity, and finish with a decline [22].

Regarding modeling user behavior, there are two main types of user profiles: (i) *Observed Profiles*, where data are collected from the target application and applied to model user behavior based on actual interaction patterns; and (ii) *Simulated Profiles*, where one or more types of user behavior profiles are defined in the absence of real data. For example, benchmarks for microservice applications like TeaStore [9] and TrainTicket [23], have three typical (simulated) user profiles: Browser/Visitor, Buyer, and Order.

Profiles depend on the specific microservice application, relying on experimental strategies based on the application's actual operations as each application has different services and workflows [24]. Some applications already provide predefined profiles, such as the browsing profile of the TeaStore [9]. To define user behavior profiles, the following steps are crucial:

1) Develop a comprehensive understanding of the microservice application, its services, and its functions;
2) Map the actions that trigger each service, identifying what users may interact with;
3) Create user profiles that include potential behaviors for each type of user;
4) Ensure that different user types are appropriately represented. It is important to recognize that some profiles may be more prevalent than others.

A proper definition of the workload profile is essential for generating realistic data to address the challenges of monitoring microservice applications. For instance, to differentiate between legitimate load spikes and attacks, the workload profile must include load spikes that reflect increased resource usage. Similarly, to investigate the dynamic scaling of services, the workload intensity must prompt the application to scale up or down during experiments. Additionally, since we are dealing with microservice applications, it is essential to define workloads that exercise all the services that compose the application. If the workloads do not exercise all the services in a representative manner, it can result in models that are unable to handle attacks that target a specific service.

The duration of the workload execution is another crucial parameter to consider. To ensure the system reaches a stable state before meaningful data collection, a warm-up period is required. Following this, workload data are collected during different phases of the experiment, including before and after the attacks. This allows for accurate measurement of the system's behavior during regular operations (pre-attack) and after the attack occurs (post-attack). Finally, the Workload Execution outputs the *Workload Output* file, which contains the run results from the client side that are used to analyze measurable attributes such as the total number of request samples, response time, and throughput over time.

The **Attack Injection (2)** component simulates attacks against the microservice application. To perform the attacks, an *Attack Tool* is necessary, which should be decided based on the *Attack Type Selection*. For each type of attack (e.g., high-volume DoS attack, Adversary-in-the-Middle (AitM), or API Abuse), we have to define the *Attack Profile* to be able to perform the attack injection. The Attack Profile dictates the behavior of the attackers and determines the configuration of the tool used to perform the attacks. It specifies when the attacks start and end and which services should be targeted.

Diverse attack profiles are essential for generating a comprehensive dataset. In the context of DoS attacks, these profiles should cover different durations and frequencies, as attackers are constantly changing their strategies to avoid detection and mitigation systems [25]. To create a robust and varied dataset, we suggest the following types of attack frequencies:

- *Short Attack:* A brief, defined attack duration, e.g. as short as a couple of 2 minutes;
- *Persistent Attack:* A prolonged attack that can last several minutes or the entire duration defined for the attacks;
- *Intermittent Attack - Variable Duration:* Repeated attacks every X minutes, with each having a different duration;
- *Intermittent Attack - Variable Interval:* Attack duration remains constant, but the interval between attacks varies.

Including diverse attack profiles ensures that the dataset reflects a wide range of potential attack scenarios, which enhance the ability of the detection model or tool to detect and respond to increasingly sophisticated attack techniques. The attack profiles must also target different services (e.g., one targeting the UI service and another the persistence service) to capture a broader range of potential impacts on microservice performance. It is also crucial to record precise timestamps for the start and end of each attack (and for each occurrence in the case of intermittent attacks), as this is needed for accurately labeling the observations monitored from the application side as either attack or regular (non-attack) behavior.

To perform a run, both the Workload Execution and Attack Injection are executed, targeting the **Microservice Application (3)**. There are two types of runs: Golden Run, where only the workload is executed, and Attack Run, where both the attack

injection and workload execution are performed. To construct the dataset, we must perform multiple golden runs and multiple attack runs, depicted in the multi-layers of Run in Fig. 2. Based on previous work [12], [13], we propose dividing a single run into five stages:

1) *Initialization (Warm-up)* period where the system performance may exhibit significant variability, making it unsuitable for analysis.
2) *Pre-attack* stage during which only regular requests are generated, establishing a baseline of regular behavior.
3) *Attack window* during which one or multiple attacks can occur while regular requests are generated, allowing the assessment of the impact of the attack in the application.
4) *Post-attack* period where only regular requests are made, establishing system behavior after an attack, which may generate false positives at attack detection.
5) *Completion (End)* stage designed to allow any pending requests to be processed and completed; these are not analyzed due to their significant variability.

The users of our framework are responsible for determining finer-grain aspects, such as the duration of each stage in a run. These details may vary depending on the infrastructures and context, such as the number of machines, the type of microservice application, or the type of attacks being considered.

Monitoring (4) comprises one or more tools or monitors to observe the system at the application level by collecting specific system attributes, such as resource consumption (e.g., memory consumption and the number of threads), the state of the services (e.g., active running services and timestamp of replica creation), and traffic-related data (e.g., request duration and request size) [3]. The monitoring tools may collect information regarding each running service instance and each node containing one or more services. They may also gather data on the application's response time and throughput (when handling user requests) and the response time and throughput of requests among the services composing the application [26].

One challenge in detecting attacks in microservice applications is determining what specific features should be tracked [3]. Selecting the right features is crucial in production environments, where performance is critical and monitoring every feature is impractical due to the associated overhead [27]. To address this challenge, our framework allows utilization of various monitoring tools during the experiments to capture data. Then, by analyzing the generated data, the researchers can identify the most effective features for attack detection, enabling the optimization of the feature set for production.

In practice, during the execution of the runs, the monitoring tools collect data for each service of the microservice application. Such data are then stored by the **Data Collection (5)** component. The data should preferably be stored in a time-series database as the chronological aspect is essential for efficient organization and analysis.

The **Query and Preprocessing (6)** is responsible for querying the stored *monitored data* and gathering the *workload output* files provided by the workload execution tool. The workload output is needed for providing timestamps for each run, as the workload profile determines the entire data generation process, including start and end times. Using these timestamps, we can query the data from the monitors for the respective run. Although these processes are independent, timestamps enable synchronization between the workload output and the monitored data, allowing for data merging if desired.

After retrieving the data, necessary preprocessing tasks are performed to build the dataset, e.g., it is crucial to label the samples correctly as regular (pre- and post-attack) or as an attack. This labeling is achieved by saving the timestamps of when each attack begins and ends, allowing precise identification of the periods when the application was under attack.

B. Model Development and Evaluation

Model development and evaluation begins with the *Method Selection*, which refers to the method that will be used to build the models for detecting attacks. Each type of method implies different models being generated, represented by the multi-layers of the Model of Fig. 2. These models must be designed considering the context of the application. In other words, the context of the microservice application (e.g., low detection precision is not acceptable due to limited budget), the type of attacks being addressed (e.g., very destructive attacks may demand high recall, regardless of the false positive rate), and the specific singularities regarding the analysis of microservice applications monitoring data (i.e., the need to analyze a large amount of data generated by the several services within the application), may call for a certain family of models. Such variety may require different families of methods (e.g., machine learning, multi-criteria decision-making, stochastic models) or different algorithms within the same method (e.g., different ML algorithms).

The **Model Design (7)** component for detecting attacks in microservice applications at runtime brings several challenges compared to other models used to detect attacks in monolithic applications. An attack detection model must consider the multiple services the application can have. These services have unique functions and can be impacted differently by user requests and attacks. Also, a model must easily adapt to the dynamic nature of microservice applications, where services are continually created and destroyed [12].

Our framework allows setting different *Model Parameters* for a given model. These determine the operational characteristics of the model during the attack detection, influencing aspects such as learning behavior, feature importance, and decision thresholds. This includes hyperparameters commonly found in ML models, like learning rates and regularization terms, as well as specific configurations relevant to other methodologies, such as weights, attribute criteria, and operators in LSP. Some techniques can help in an parameter selection, such as grid search in ML models. Other strategies, such as Bayesian optimization and genetic algorithms, can also be effective for hyperparameter tuning [28].

Data Processing and Feature Selection is crucial to transform raw data into a format suitable to build the model. In practice, the data are preprocessed by performing the necessary calculations (e.g., the CPU usage can be provided as an incremental value

starting from the beginning of the service instance's lifecycle) and handling missing values. Feature selection should consider the system properties and information that the model will use to select a subset of relevant features from the available dataset to improve the model performance. This selection can be based on related work, including techniques such as filter methods (e.g., correlation analysis), wrapper methods (e.g., recursive feature elimination), and embedded methods (e.g., feature importance from decision trees) [29]. In the case of ML models, the dataset is split into training and testing subsets to accurately evaluate the model's performance.

Model Evaluation (8) consists of assessing the model's performance using the testing set generated by the Data Generation module. This set can either encompass the entire dataset when no training is required or exclude the data used for training. The model is then executed with input data to evaluate its attack detection performance. The details of the evaluation process vary depending on the technique used (we provide examples regarding the evaluation of ML and LSP models in Sections V and VI, respectively). Techniques like k-fold cross-validation or leave-one-out cross-validation can ensure a robust performance evaluation by iterative training and testing the model on various subsets of the data. After that, the model's detection results constitute the **Model Output (9)**.

This final result is then assessed using various metrics and visual representations during the **Result Analysis (10)**. Metrics such as Recall, Precision, and F1-score provide quantitative insights into the model's performance, supporting a comprehensive understanding of its effectiveness. On the other hand, visual representations like Boxplots and ROC curves offer graphical interpretations of the model's behavior across different data samples from different runs with various attack and workload profiles. This helps identify outliers and understand the distribution of results. A functional model is found if the results prove satisfactory; otherwise, the process returns to the model design component to improve the detection capabilities.

IV. Data Generation: DoS Attacks

In this section, we describe how we instantiated the first module of our framework to generate realistic attacks against microservice applications. The proposed framework intendeds to be generic, thus the Data Generation module can be used for different types of attacks by selecting different Attack Profiles and Attack Tools. We illustrate and validate the framework on DoS attacks, which directly impact on the availability of the applications, compromising their capacity to respond to requests from users. Specifically, we generate two types of DoS attacks: high-volume and low-volume. However, the Attack Injection component is agnostic to attack type and adaptable to other types of attacks. Refer to **Appendix B**, available online for other compatible attacks and tools. High-volume attacks send many requests to mimic legitimate users and overwhelm the target system by consuming available resources. In contrast, low-volume attacks exploit system resources by slowly sending requests and keeping them open to exhaust available resources needed to initiate new connections [30].

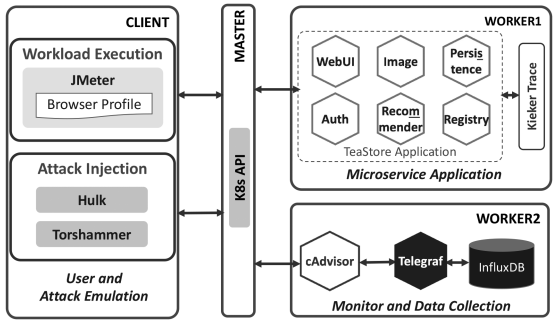

Fig. 3. Experimental setup for data generation.

Fig. 4. Stages of an experimental run.

We present the experimental setup and provide a detailed description of how we generated the workload, injected attacks, and collected the data needed to build the dataset. Details on conducting the experiments, along with the generated data, can be found at http://doi.org/10.5281/zenodo.11204757.

A. Experimental Setup

The experimental setup consisted of four virtual machines (VMs): Client, Master, Worker1, and Worker2. As in Fig. 3, these VMs are equipped with all the necessary components to mimic real-world attacks and monitor system behavior during both regular and attack scenarios. The Client VM is responsible for generating a workload using JMeter. In addition, it includes two tools for performing high- and low-volume DoS attacks, namely Hulk and Torshammer, respectively. The Master, Worker1, and Worker2 VMs were deployed using the orchestrator Kubernetes. The Master VM manages and controls the cluster while Worker2 is responsible for running the monitoring and data collection tools. Finally, Worker1 runs the microservice application, TeaStore [9].

TeaStore is a microservice-based application with six services, designed for benchmarking, performance, and resource management research. This is a representative microservices application and using a single one to illustrate and validate the approach is a common practice in attack and anomaly detection studies. For example, prior studies [15], [16], [18] used Clearwater, TPC-W, and TrainTicket applications, respectively. Details on the setup are given in **Appendix A**, available online.

B. Workload Execution and Attack Injection

We performed multiple runs to generate data by combining different attack profiles and attack types. As illustrated in Fig. 4, each run lasted 35 minutes, which were divided as follows: 10-minutes for the warm-up, 5-minutes for pre-attack with only regular requests, 10-minutes for the attack window, 5-minutes for the post-attack stage, and the final 5-minute completion (end) stage. This configuration was defined to ensure system stability

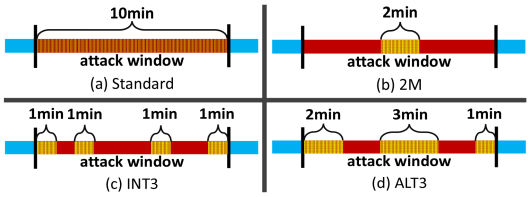

Fig. 5. Attack profiles.

TABLE I
ATTACK PROFILES

Profile	Number of Runs per Tool		Pages	Frequency (attack window)	Attack Duration
	Hulk	Tors-hammer			
STD-Home	10	10	Homepage	one attack at min. 0	10 min.
STD-PROD	10	0	Product	one attack at min. 0	10 min.
STD-CAT	10	0	Category	one attack at min. 0	10 min.
2M	10	10	Homepage	one attack at min. 0, 3 or 8	2 min.
INT3	10	10	Homepage	attack at min. 0, 2, 6 and 9	1 min.
ALT3	10	0	Homepage, PROD, CAT	attack at min. 0, 4 and 9	2 min., 3 min., 1 min.
Golden Run	10 runs without attack				

during the warm-up period. It includes a phase for gathering information without interference from attacks (pre-attack) and another phase for monitoring system behavior after the attack (post-attack). The 10-minute attack window was selected to address the varying frequencies of DoS attacks. Consequently, each attack run generated (1) samples while running only the regular workload (i.e., non-attack) and (2) samples generated while running both the workload and attacks (i.e., attack).

To generate the workload, i.e., to emulate real users interacting with the application, we used the TeaStore Browsing profile provided by the TeaStore project [9] with the following values would be suitable for our available hardware: 300 users, a ramp-up time of 1 s, and a constant rate of 10,000 requests per minute. These values are based on the results obtained in our performance analysis, as explained in [13].

We employed four attack profiles carefully designed to create diversity in the attack scenarios. For this purpose, we considered factors such as attack frequency, attack duration, the targeted services by attacking different TeaStore webpages, and attack type. Fig. 5 illustrates the four attack profiles, while Table I describes the sub-profiles, the number of attack runs generated by Hulk and Toshammer tools, targeted pages, attack frequency, and attack duration (see **Appendix A**, available online).

As shown in Table I, a total of 90 experimental runs with different attack types were executed, including 60 with the Hulk attack tool and 30 with the Torshammer tool. In addition, 10 golden runs were generated, executing only the regular workload, without running any attack profiles.

C. Monitoring, Data Collection, Query & Preprocessing

To monitor the resource usage and performance of TeaStore, we used cAdvisor which provides real-time information about the resource consumption of the Kubernetes pods (also known as containers). Telegraf agent was used to collect the monitored data, and store them into the InfluxDB database.

TABLE II
DATA PREPROCESSING

name	time	counter	container
container_cpu_cfs_periods_total	1652046164	8608	teastore-webui
container_cpu_cfs_periods_total	1652046182	8790	teastore-webui
container_cpu_cfs_periods_total	1652046201	8981	teastore-webui

[a] Data as collected

time	container	container_cpu_cfs_periods_total	timestamp	container_cpu_cfs_throttled_periods_total	container_cpu_cfs_throttled_seconds_total
2022-05-08 21:42:44	teastore-webui	8608	1652046164	5024	834.855
2022-05-08 21:43:02	teastore-webui	8790	1652046182	5168	859.901

[b] Samples merged per service

time	timestamp	container_sockets_webui	container_threads_webui	type	momentum	throttled%_webui	cpu_cfs_throttled_seconds_webui	cpu_system_webui
2022-05-08 21:43:21	1652046201	404	421	hulk	pre	0.524	16.158	0.188
2022-05-08 21:43:34	1652046214	449	421	hulk	pre	0.772	18.202	0.221

[c] Samples merged per application

Once all the data were collected, we labeled them into regular samples (pre and post) and attack data. The labeling was based on timestamps provided by the attack scripts, which recorded the exact start and end times of each attack episode.

After collecting the data, the next step was data preprocessing. Initially, the dataset containing the collected data was structured as follows: feature name, timestamp, value, and container name (representing the service within Kubernetes), as shown in Table II(a). Note that we use the terms ‘feature’ and ‘attribute’ as synonyms: ML methods use ‘features’, while the LSP method uses ‘attributes’. These data were initially linearly arranged in a CSV file, with features and their values listed sequentially over time. However, to analyze the data, we needed to reorganize them by transposing the feature names into columns and merging the samples by timestamp. This way, each sample contains all feature values for a particular service at a specific time. Table II(b) displays a cutout of the dataset without all the columns and rows. Additional merging was necessary to prepare these data to be used by the ML algorithms. Samples from all services at a given timestamp were aggregated into a single sample, so we expanded the feature names to indicate the service. For example, the feature ‘container_sockets’ was expanded to include the service names such as ‘container_sockets_webui’ and ‘container_sockets_persistence’. This organizes the data as in the example in Table II(c), which shows only a selection of columns. This process ensured that every sample in the dataset included the values for all features across all services.

It is important to note that since these features come from a microservice application, the actual set of monitored features is multiplied by the number of running service instances. For example, in our setup, which has one instance per service, we have one feature for the CPU usage of each of the five TeaStore services (e.g. WebUI_CPU_usage, Persistence_CPU_usage). The dataset is thus composed of multiple files, each representing a golden or attack run. Each run comprises various samples organized chronologically, with each sample representing a single observation of the run. Since each run lasts 20 minutes, with cAdvisor providing monitoring data at an average interval of 15 seconds, it results in approximately 77 ($\pm$ 3.55) samples per run. Note that the 15-second interval was not deliberate; rather,

it represents the average frequency at which cAdvisor collects and reports data. While the reporting times can vary, this does not affect the results since the latest value can always be used for each sample, ensuring a consistent dataset. Each sample includes the value of each feature, a timestamp, and a label indicating whether it is an attack or a regular sample. In total, the 100 runs comprise 7,709 samples.

Some samples may have missing data for one or more features. This may be because, at the time of collection, the cAdvisor monitor did not update the value of that feature for a particular service within the 15-second time window. In such cases, we did data imputation. We used the 'bfill' method [31] for the features that vary over time, which backward filled the missing values in the dataset. For instance, if a missing value occurred for the number of active threads and the previously value was 257, this method filled the missing value with 257.

The features collected by cAdvisor can be of two types: Gauges and Counters. Gauges provide the value of the feature at the time of measurement (e.g., the number of sockets is 280 at time t and 260 at time $t + s$), with no need for further calculation. On the other hand, a Counter accumulates the feature values over time (e.g., CPU usage is 400 millicores at time t and 450 millicores at time t+s, meaning that CPU usage between t and t+s is 50 millicores). Therefore, for the counter features, we had to calculate the proper value at all measurement intervals. We computed the normalized value for the metrics that are based on the available resources as follows:

$$\text{CPU usage} = \frac{\text{cpu\_usage\_seconds\_total}}{\text{spec\_CPU\_quota}} \tag{1}$$

$$\text{CPU Throttled} = \frac{\text{container\_cpu\_cfs\_throttled\_periods\_total}}{\text{container\_cpu\_cfs\_periods\_total}} \tag{2}$$

$$\text{Memory usage} = \frac{\text{container\_memory\_usage\_bytes}}{\text{container\_spec\_memory\_limit\_bytes}} \tag{3}$$

V. ML Models Development and Results

This section discusses the use of our framework to develop ML models for DoS attacks in microservice applications.

A. Feature Selection

To choose the most relevant features, we first conducted an exploratory analysis, and then investigated the feature importance using the Random Forest algorithm, as presented in our previous work [11]. In our exploratory analysis, we investigated how the values of each feature fluctuated over time. In practice, we created graphs to visually assess the behavior of each feature during each run, specifically comparing the attack periods with regular workloads. This way, we identified the features that exhibited value changes during attacks. In addition, we calculated the average values and standard deviations of these features, allowing for a better understanding of their behavior, and compared values of different features which allowed us to identify the ones that, despite having different names, exhibited the same patterns and values. For example, if the memory_usage_bytes value in a sample was 1000, memory_working_set_bytes would also be 1000; when the value of the former increased to 1200, the latter would also have a value of 1200. To avoid redundancy, in such cases we selected only one of the features.

TABLE III
Selected Features

Name	Description	Model
cpu_cfs_periods_total	Elapsed enforcement period intervals	Sup., Unsup., LSP
cpu_cfs_throttled_seconds	Duration container has been throttled	Sup., Unsup., LSP
cpu_user_seconds_total	User CPU time consumed	Sup., LSP
cpu_system_seconds_total	CPU time consumed	Sup., LSP
cpu_usage_seconds_total	Combined CPU time consumed of user and system	Unsup.
memory_usage_bytes	Memory usage	Sup., Unsup., LSP
sockets	Open sockets for the container	Sup., Unsup., LSP
threads	Threads running inside the container	Sup., Unsup., LSP

We also explored the feature importance using the Random Forest algorithm to determine the significance of each feature in the model's decision-making process. This helped us to understand which features contribute the most to accurately detecting high- and low-volume DoS attacks by measuring the impact of individual features on the model's performance.

Based on the explanatory analysis and the investigation of features' importance, we selected the features that had the most significant impact on detection performance (Table III). Note that for this work, we focused on the features provided by cAdvisor as these are natively available in Kubernetes. Each feature was collected for each of the five services of the TeaStore application. Therefore, as can be seen in the column 'Model', the supervised and LSP models utilized a total of 35 features simultaneously. In contrast, the unsupervised models used either 1 or 2 features per service, resulting in 5 or 10 features at a time for the five services. This grouping strategy was adopted because such models, particularly traditional ones like Isolation Forest, tend to perform better with fewer input features. However, deep learning-based models, such as Autoencoders, can handle a larger number of features.

More details are in Section V-E.

B. Evaluation of Classification Performance

ML model classifies each sample in an execution run as either regular (annotated with 0) or attack (annotated with 1). We compare the model's output with the ground truth to obtain the confusion matrix with true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN). To evaluate our models we use well-known metrics:

- Recall: rate of actual attack samples that are correctly classified: $recall = TP/(TP + FN)$;
- Precision: proportion of predicted attacks that are correctly classified: $precision = TP/(TP + FP)$;
- F1-score: harmonic mean between precision and recall: $F1 = 2 \cdot precision \cdot recall/(precision + recall)$.
- Markedness: difference in predictive value for positive and negative classes, reflecting the model's ability to correctly identify both attack and regular samples: $markedness = (TP + TN)/(TP + FP + TN + FN)$
- Informedness: probability that the model will make an informed decision rather than a random guess, considering

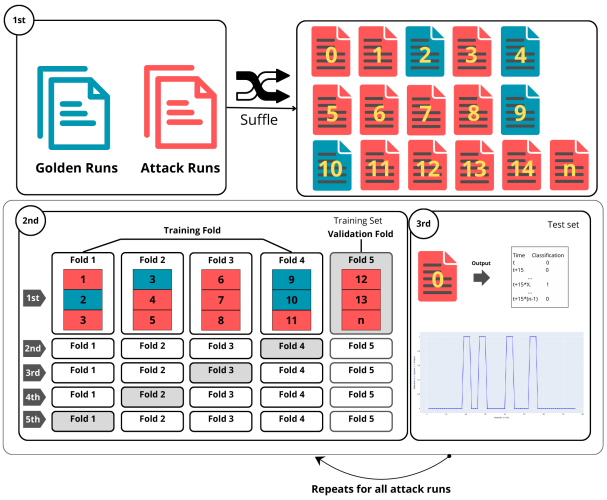

Fig. 6. Supervised ML model training and testing.

both recall (i.e., sensitivity) and Specificity: $Informed. = TP/(TP + FN) + TN/(TN + FP) - 1$.

C. Supervised ML Models Training and Testing

When developing ML models, two important steps are Model Training and Testing (components (7) and (8) in Fig. 2). During training, a model is trained using a portion of the dataset to identify patterns in the data. This involves iteratively adjusting the model parameters to improve the results. Model testing is responsible for testing the trained model on a separate portion of the dataset.

Fig. 6 illustrates the process used for training and testing the supervised models, which consisted of three stages. In the first stage, all runs (golden runs and attack runs) were loaded. As each run was stored in a separate file, we compiled a list of filenames for all runs in the dataset and shuffled the filenames using the Python's *random.shuffle* method, with a fixed seed value (we used 42 in these experiments), ensuring that all ML experiments consistently use the same shuffled dataset.

The second stage is the training process, while the third stage is dedicated to testing the model. Given the limited number of runs, we structured the training and testing sets based on the Experiment-wise Leave-One-Out Cross-Validation (ELOOCV) procedure [32]. The use of ELOOCV allowed us to train the classification models on a large portion of the dataset in each interaction. Specifically, we trained the model using all golden runs and all attack runs except one attack run, which was reserved as the test set. This process was repeated, ensuring that each attack run was used in the test set once. For example, when training with low-volume attack runs and golden runs, the dataset consisted of 40 runs (approximately 3,080 samples). So, we trained the model using 39 runs (approximately 3,003 samples) and tested the model using a single attack run (around 77 samples). In this case, the process was repeated 30 times, one for each attack run. Thus, the results presented in this paper represent the average performance across all 60 runs for high-volume attacks and across all 30 runs for low-volume attacks. It is important to remember that the model classifies each sample in the test run as either regular or attack and not a run as a whole.

TABLE IV
SUPERVISED ML ALGORITHMS AND HYPERPARAMETERS

Algorithm	Parameters
RF	n_estimators: [100, 200, 300], max_features: [0.1, 'sqrt', 'log2'], min_samples_leaf: [1, 2, 5], min_samples_split': [2, 4, 5], random_state: [42]
SVM	C: [1, 10, 100], kernel: ['linear', 'rbf', 'poly'], gamma: ['scale', 'auto'], degree: [2, 3],
KNN	n_neighbors: [3, 5, 10, 20], weights: ['uniform', 'distance'], metric: ['euclidean', 'manhattan']
XGB	max_depth: [3, 5, 7], learning_rate: [0.1, 0.01, 0.001], subsample: [0.5, 0.7, 1], random_state: [42]
CNN	filters: [16, 32], kernel_size: [3], dropout_rate: [0.2, 0.5], learning_rate: [0.0001, 0.001], epochs: [25], batch_size: [77], kernel_regularizer: [l2(0.001)]

We selected five supervised ML algorithms to build models for detecting DoS attacks: K-Nearest Neighbors (KNN), Random Forest (RF), Support Vector Machines (SVM), XGBoost (XGB), and Convolutional Neural Network (CNN).

For the CNN, we had to define additional configurations. First, we reshaped the data to a one-dimensional format to ensure compatibility with the CNN architecture. Furthermore, the defined model consisted of a sequential structure that began with a convolutional layer applying the *ReLU* activation function. The input shape was set to match the reshaped data dimensions. Following the convolutional layer, a max pooling layer was added to reduce the spatial dimensions of the output, enhancing feature extraction with poll_size equal to 2. A dropout layer was included to mitigate overfitting. The output from the pooling layer was then flattened and passed through a dense layer with 100 units and a *ReLU* activation function. To further address overfitting, another dropout layer was applied before the final output layer, which used a sigmoid activation function to produce binary classifications. The model was compiled with the *Adam optimizer* and trained using *binary cross-entropy loss*, with accuracy as the performance metric.

We employed the Grid Search Cross-Validation approach using the *GridSearchCV* method from sklearn in the training stage. To execute this, we applied the *GroupKFold* method with five splits, performing five-fold cross-validation. Since runs are independent of each other, each run was treated as a group. This ensured that when *GridSearchCV* divided the data into folds, all samples from the same run remained in the same fold. Note that this process was repeated for each ELOOCV fold, ensuring that parameter tuning was specific to the training data of each fold. Employing grid search for parameter tuning established the best set of parameters for each model, improving the classification performance.

The parameters used for each supervised ML algorithm in the grid search are detailed in Table IV and correspond to the *Model Parameter* module shown in Fig. 2(b). After the parameter tuning, the best model was tested on the previously excluded attack run. Specifically, the model classified each one of the 77 samples in the test run as either regular or attack. Therefore, if the model is deployed to identify attacks at runtime, it could

TABLE V
SUPERVISED ML PERFORMANCE FOR HIGH-VOLUME DOS ATTACKS

	Precision	Recall	F1-Score	Markedness	Informedness
RF	0.97±0.07	0.98±0.03	0.98±0.05	0.96±0.07	0.97±0.03
SVM	0.96±0.13	0.97±0.13	0.96±0.13	0.94±0.18	0.95±0.13
KNN	0.91±0.15	0.93±0.13	0.92±0.14	0.85±0.19	0.91±0.13
XGB	**0.98±0.04**	**0.98±0.03**	**0.98±0.02**	**0.97±0.04**	**0.97±0.03**
CNN	0.95±0.14	0.96±0.13	0.95±0.13	0.93±0.19	0.95±0.13

TABLE VI
SUPERVISED ML PERFORMANCE ON LOW-VOLUME DOS ATTACKS

	Precision	Recall	F1-Score	Markedness	Informedness
RF	**0.99±0.02**	0.95±0.07	0.97±0.04	0.94±0.03	0.94±0.07
SVM	0.93±0.18	0.94±0.18	0.94±0.18	0.92±0.20	0.93±0.18
KNN	0.97±0.04	0.94±0.07	0.95±0.05	0.93±0.05	0.93±0.07
XGB	0.98±0.03	**0.97±0.06**	**0.98±0.04**	**0.98±0.04**	**0.96±0.06**
CNN	0.98±0.04	0.97±0.05	0.97±0.04	0.97±0.04	0.96±0.06

indicate whether the application is under attack every 10 or 15 seconds.

D. Results of the Supervised ML Models

The results for high-volume DoS attacks are presented in Table V, where we can see that XGBoost produced the best result for all the metrics, with recall, precision, and F1-score of 0.98, and markedness and informedness of 0.97. Following XGBoost, RF showed strong performance with results between 0.96 and 0.98 for the various metrics. SVM also performed well, with scores ranging from 0.94 to 0.97, Followed by the CNN which had a recall of 0.96 but markedness of 0.93. Finally, the KNN achieved F1-score of 0.92, but had the lowest markedness of 0.85. Despite being last, KNN still delivered good results, particularly regarding recall and F1-score.

The results indicate that XGBoost is highly effective at correctly identifying and classifying high-volume DoS attacks, similarly to RF, as both are based on decision trees. On the other hand, SVM, KNN and CNN models had lower performance, particularly struggling to detect attacks from the ALT3 and INT3 profiles. This difficulty contributed to an increased standard deviation of their performance metrics, reflecting less consistent performance compared to XGBoost and RF. Due to limited space, detailed results divided by run and attack profile are provided as supplementary material at http://doi.org/10.5281/zenodo.11204757.

Table VI shows the results for low-volume DoS attacks. These results are slightly different, especially regarding the recall. When compared to high-volume DoS attacks, the three best algorithms RF, SVM, and XGB had worse performance in terms of recall. However, the other two, KNN and CNN, had slightly better recall. Still, XGBoost had the best result for all the performance metrics except recall, tying with RF regarding precision (0.98) and SVM regarding informedness (0.97). In low-volume DoS attacks, the precision was slightly higher than the recall for all the supervised ML algorithms, except for SVM which had slightly higher recall than precision. Similarly to the high-volume DoS models, these models struggled to detect attacks with the ALT3 profile. Remember that the low-volume DoS attacks did not have an INT3 profile.

TABLE VII
SUPERVISED ML PERFORMANCE FOR COMBINED DOS ATTACKS

	Precision	Recall	F1-Score	Markedness	Informedness
RF	0.97±0.06	0.97±0.08	0.97±0.07	0.96±0.08	0.96±0.09
SVM	0.97±0.04	0.98±0.04	0.97±0.03	0.96±0.05	0.97±0.05
KNN	0.92±0.15	0.92±0.16	0.92±0.15	0.90±0.19	0.90±0.16
XGB	**0.98±0.04**	**0.98±0.04**	**0.98±0.03**	**0.97±0.04**	**0.97±0.05**
CNN	0.97±0.06	0.98±0.05	0.97±0.05	0.96±0.06	0.96±0.05

When comparing the low- and high-volume attack results, we observed that the average recall for all ML algorithms was slightly lower for low-volume attacks, whereas precision was marginally higher. This suggests that low-volume DoS attacks may be harder to detect than the high-volume DoS attacks.

We also built ML models using both attack types, with the results presented in Table VII. The training was conducted according to the process shown in Fig. 6, where the attack run files of both types were shuffled together. Consistent with the previous findings, XGBoost (XGB) achieved the highest performance across all metrics, followed closely by the CNN, SVM, and RF algorithms. KNN, however, showed worst performance, with metric values around 0.90.

In order to provide additional insights, we conducted an in-depth analysis of the ML models performance for specific attack profiles which allowed us to identify the profiles that posed the greatest challenges for the models. These results divided by attack profile are provided in our supplementary material at http://doi.org/10.5281/zenodo.11204757. For high-volume attacks, XGB had difficulties with the INT3 profile, and RF exhibited the lowest precision with 2 M, followed closely by INT3. When it came to recall, RF struggled the most with INT3. On the other side, KNN, SVM, and CNN encountered the most challenges with the ALT3 profile. For low-volume attacks, most algorithms struggled with the INT3 profile. The only exception was SVM, for which the 2 M profile was the most challenging. When analyzing the models' performance for both types of attacks, SVM, XGB, and CNN exhibited consistent difficulty with the INT3 profile for both precision and recall. RF maintained similar precision across the ALT3, INT3, and 2 M profiles, with ALT3 being the most challenging for recall. Lastly, KNN faced the most difficulty with ALT3 for precision and with INT3 and 2 M for recall.

E. Unsupervised ML Models Training and Testing

The unsupervised models were trained exclusively on the golden runs, which allowed us to develop anomaly detection models without the need for attack data during the training phase. Specifically, the training set consisted of the ten golden runs, around 770 samples, and the testing set incorporated all attack runs, i.e., around 4,620 samples from high-volume attack runs and 2,310 samples from low-volume attack runs.

We selected five unsupervised algorithms: Isolation Forest (ISOF), One class SVM (OneSVM), Local Outlier Factor (LOF), Autoencoder (AE) and Variational Autoencoder (VAE). Note that for ISOF and LOF we used the same model for both low- and high-volume DoS attacks. However, further investigation was necessary for OneSVM to improve the results. Specifically,

TABLE VIII
UNSUPERVISED ML ALGORITHMS AND PARAMETERS

Algorithm	Parameters
ISOF	n_estimators: 100, max_samples: auto, random_state: [7, 21, 42, 123, 999], contamination: auto
OneSVM	kernel: [linear, rbf, sigmoid, poly], gama: auto
LOF	novelty: True
AE	learning_rate: 0.0001, epochs: 100, batch_size: 64, validation_split: 0.2, loss_function: mse
VAE	learning_rate: 0.0001, epochs: 100, batch_size: 64, validation_split: 0.2, loss_function: [mse, KL divergence]

since the OneSVM's results for high-volume attacks were unsatisfactory, we analyzed different kernel parameters and normalized the data using Z-score. We found that the models that achieved better results for high-volume attacks differed from those that performed well for low-volume attacks. Therefore, for high-volume we used normalized data and an RBF kernel, while for low-volume attacks we used a nonnormalized data and linear kernel. The parameters for unsupervised ML algorithms are in Table VIII.

The AE with all features includes an encoder with three Dense layers (32, 16, and 8 neurons), followed by a 4-neuron bottleneck. The decoder mirrors this structure in reverse (8, 16, 32 neurons). For the grouped features model, due to reduced input size, the encoder uses layers with 8, 4, and 2 neurons, and omits the bottleneck. All layers use ReLU activation. Models were compiled with the Adam optimizer and Mean Squared Error (MSE) loss. Anomalies were detected based on reconstruction error (Mean Absolute Error), with a threshold set at the 99th percentile of the training loss, selected after analyzing the long-tailed error distribution and comparing performance across multiple percentiles. The VAE mirrors the AE structure, but replaces the bottleneck with Dense layers to compute the latent mean (z_mean) and log variance (z_log_var), and uses a single-sampling approach.

For deterministic models like OneSVM and LOF, the training and testing process was straightforward – we trained the models on the golden runs and tested them on the attack runs. However, a slightly different strategy was required for the ISOF algorithm due to its inherent randomness during training. Specifically, we divided the attack runs into five folds, using each fold as the test set while training the model on the golden runs with one of five different *random_state* (seed) parameters. The seeds used for each experiment are in Table VIII. This ensured a robust evaluation of the model's performance across different random initialization scenarios. For the AE and VAE were implemented using TensorFlow, so TensorFlow was configured with the option enable_op_determinism.

Note that we selected groups of features to train the models using unsupervised algorithms compared to the supervised ones. Initially, we experimented using the same set of features, but due to the nature of traditional unsupervised models, this yielded poor results, with recall values as low as 1%. To improve the performance, we reduced the number of features by defining four groups, where each feature is collected per service, resulting in 5 or 10 input features depending on the group size. The groups are: (1) CPU usage (CPU: 1 $\times$ 5); (2) CPU and memory usage (C&M: 2 $\times$ 5); (3) sockets and threads (S&T: 2 $\times$ 5); and (4) CPU CFS periods and throttled seconds (C&T: 2 $\times$ 5). In contrast, deep learning-based models (AE and VAE) can handle a larger number of features, and we also trained them using the full feature set.

TABLE IX
PERFORMANCE OF THE COMBINATIONS OF UNSUPERVISED ML ALGORITHMS AND SELECTED GROUPS OF FEATURES

High-volume attacks					
Ranking	Precision	Recall	F1-Score	Markedness	Informedness
1st	**VAE-S&T (0.88±0.14)**	**ISOF-S&T (0.99±0.02)**	**VAE-S&T (0.92±0.1)**	**VAE-S&T (0.85±0.23)**	AE-S&T (0.91±0.14)
2nd	AE-S&T (0.86±0.15)	AE-S&T (0.99±0.02)	AE-S&T (0.92±0.130	AE-S&T (0.84±0.24)	VAE-S&T (0.90±0.14)
3rd	ISOF-CPU (0.80±0.17)	LOF-S&T (0.99±0.03)	ISOF-CPU (0.84±0.16)	ISOF-CPU (0.74±0.25)	ISOF-CPU (0.78±0.18)
Low-volume attacks					
Ranking	Precision	Recall	F1-Score	Markedness	Informedness
1st	**AE-CPU (0.84±0.17)**	**ISOF-C&M (0.99±0.04)**	**AE-CPU (0.86±0.10)**	**AE-CPU (0.78±0.34)**	**VAE-CPU (0.76±0.10)**
2nd	VAE-CPU (0.84±0.17)	1SVM-CPU (0.98±0.04)	VAE-CPU (0.86±0.15)	VAE-CPU (0.78±0.33)	AE-CPU (0.83±0.20)
3rd	LOF-CPU (0.71±0.19)	LOF-C&M (0.96±0.05)	LOF-CPU (0.80±0.16)	LOF-CPU (0.69±0.21)	LOF-CPU (0.81±0.20)

F. Results of the Unsupervised ML Models

Table IX presents the best results of combining the unsupervised ML algorithm with the different sets of features. For high-volume DoS attack detection, the VAE-S&T model consistently outperformed other configurations across most metrics. It achieved the highest scores for precision (0.88), F1-score (0.92), and markedness (0.85), and the second highest for informedness (0.90). Although it was not ranked in recall (0.99), the difference from the top result was marginal. The AE-S&T model followed closely, the best result for informedness and the second best for all other metrics.

The ISOF model with S&T feature excelled in recall (0.99), despite not ranking highly in other metrics and having a lower precision of 0.67. In contrast, ISOF-CPU offered a more balanced outcome, ranking third in precision (0.80) F1-score (0.84), markedness (0.74), and informedness (0.78), showing a more balanced performance than its S&T model. The LOF algorithm performed best when combined with the S&T feature across most metrics, except for precision, where LOF-CPU slightly outperformed it (0.74 vs. 0.72). This shows the S&T feature group is particularly effective for identifying high-volume attacks, especially when paired with deep learning models such as AE and VAE.

In low-volume attack detection, AE-CPU achieved the best overall performance, ranking first in precision (0.84), F1-score (0.86), markedness (0.78), and second in informedness (0.83) and obtained a high recall of 0.92. VAE-CPU also showed strong results, comparable with the AE-CPU results. The difference was only in the third or fourth decimal place.

Despite not topping recall, the LOF-CPU model led in precision (0.71), F1-score (0.80), markedness (0.69), and informedness (0.81), making it the most balanced non-deep learning alternative. Even though the LOF algorithm did not reach the top of the ranking for recall, it still delivered a solid recall of 0.96. This result contrasted with the best result obtained by LOF

in the high-volume attacks, where the S&T feature was more effective. The ISOF algorithm was out of the rank, but also yielded the best results using the CPU feature. For recall, the ISOF model using the C&M feature was ranked first with 0.98, but it performed poorly on the other metrics, such as F1-score of 0.46. This indicates the ISOF with C&M only achieved high recall because it incorrectly labeled most of the samples as attacks. The ISOF with CPU did not achieve the highest recall, but still performed well with a rate of 0.95. The OneSVM algorithm showed high recall, ranking second, but performed poorly on the other metrics. For example, OneSVM-C&M had a precision of 0.47, an F1-score of 0.59, markedness of 0.37, and informedness of 0.58.

VAE and AE models detected high-volume attacks effectively, but their performance dropped significantly for low-volume attacks. VAE-CPU had the best precision (0.78) but only reached an *F1 Score* of 0.43, revealing its difficulty maintaining balance under low-volume conditions. Overall, for the low-volume attacks, the CPU usage was the most crucial feature for all unsupervised ML algorithms. When using unsupervised ML models, consistent with supervised ML models, low-volume DoS attacks were harder to detect than high-volume attacks, and the most challenging attack profiles to detect were INT3, 2 M, and ALT3.

We also evaluated the performance of models using the complete set of features. While ISOF, LOF, and One-Class SVM struggled in this configuration, AE and VAE achieved strong results, particularly in high-volume attack detection. VAE had precision of 0.86, recall of 0.93, and F1-score of 0.89, while AE followed closely with scores of 0.83, 0.98 and 0.89, respectively. However, both models underperformed regarding low-volume attacks. For these attacks, VAE resulted in precision of 0.57, recall of 0.94, and F1-score of 0.68, while AE obtained scores of 0.42, 0.95, and 0.53, respectively. Notably, when considering a single model for detecting both high- and low-volume attacks, the VAE trained with the complete feature set achieved the best overall performance, with a precision of 0.76, recall of 0.93, and F1-score of 0.82.

These results suggest that VAE and AE could benefit from an expanded and more informative feature space.

VI. LSP Model Development and Results

Next, we use our framework to develop LSP models that can detect DoS attacks in microservice applications, and present the results. LSP is a multi-criteria decision-making method designed to evaluate, compare, and optimize complex systems by decomposing the systems's characteristics into measurable attributes that can be aggregated into a single score [33]. We use the LSP model to calculate a global score that characterizes the security status of a microservice application at runtime.

A. LSP Models Development

An **LSP model** consists of a *suitability attribute tree* that includes the following *Model Parameters: attribute criteria*, *weights*, and *operators*. The *tree* structure shows how the elementary attributes should be aggregated to obtain the global score that is compared to a threshold to determine if the application is under attack. When applying the LSP method to microservice application environments, the application is decomposed into different services, such as a user interface service, a storage-centric service, and others. Each service is further decomposed into one or more service instances that can be replicated into multiple simultaneous instances, allowing for the application's scalability. Finally, each instance is decomposed into the elementary attributes, which are the attributes that can be monitored and measured (e.g., the number of active sockets). This structure corresponds to Model Design (7) component in Fig. 2(b).

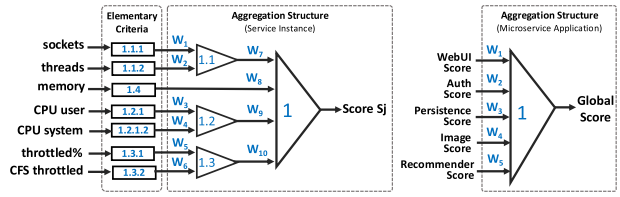

Fig. 7. LSP tree structure for DoS attacks.

We propose using the structure presented in Fig. 7 for the LSP technique. This structure contains the selected attributes and how they are aggregated in the model. While it builds on the work [12], we introduce new values for high-volume attacks and define the model values for low-volume attacks, and address transferability between models for a unified detection approach. In this paper, we used this structure for all the experiments but defined different values for elementary criteria, weights, and operators, as discussed in Section VI-B.

1) Attribute Criteria Specification: In the context of LSP models, *Feature Selection* involves determining the set of elementary attributes. In this study, we used previously analyzed and selected attributes for ML models discussed in Section V. After selecting the attributes, it is essential to specify their **Attribute Criteria**. The criteria consist of defining reference values used to map the input values to their corresponding suitability degrees and can be one of three types [33]: *Type S*, which represents preferred small values (the smaller the value, the better); *Type L*, which represents preferred large values (the higher the value, the better); *Type R*, which means a preferred range of values (the desired values are between a range).

To ensure that all the criteria are precisely specified, it is necessary to analyze how each attribute is affected by the regular usage of the application and during attacks. It is important to highlight that the attributes can be impacted differently depending on the attack and service types. Since each service consumes different amounts of resources, it is necessary to define the criteria per attribute and service. Each service must be analyzed to specify its proper attribute criteria, resulting in a total number of criteria equal to the *number of attributes* multiplied by the *number of service types*. For instance, a high-volume DoS attack can increase CPU usage, while a low-volume DoS attack may decrease CPU usage [11]. Thus, for high-volume attacks, the CPU usage is classified as *Type S*, while for low-volume attacks it is classified as *Type L*. The values are mapped to their preferred value using a linear interpolation function, which is commonly applied with the LSP method [33] and provides good results [12].

2) Weights Definition: The weight of each attribute shows its importance for the aggregation in the attack detection model. Determining these weights can be complex as their relevance varies depending on the attack type. Our prior study compared the weight assignment methods (i.e., ranking, direct weight assessment, and importance decomposition) and showed comparable results across all methods [12].

We chose the *direct weight assessment* for its flexibility in assigning values and the *importance decomposition method* for its ease in specifying aggregation operators. In the latter, we needed to define the overall importance of the attribute for the aggregation; according to Dujmovic [33], the overall must be a value between 0 and 16, where 0 is the lowest importance and 16 is the highest. After defining the overall importance for all the attributes from the same aggregation, the weight for each attribute W_i is calculated using (4).

$$W_i = \frac{S_i}{\sum_{j=1}^{n} S_j}, \tag{4}$$

where S_i is overall importance value for the attribute i.

3) Operator Selection: The operators represent the relationship between the attributes participating in the aggregation. An operator belongs to one of the three types of logical models: substitutability (OR), simultaneity (AND), and neutrality. There are two options to define the operators: directly select them or use the importance decomposition method we previously used for weight definition. In the latter case, we use the (5) to calculate

$$\alpha = \frac{\sum_{j=1}^{n} S_j}{n * S_{\max}}, \tag{5}$$

where $S_{\max}$ is the maximum degree of importance of an attribute (in this work $S_{\max}$=16).

We then map this α to an operator using a table that maps operators in a group of operators. We use the GGCD17 group [12], [33]. For instance, if $\alpha = 0.0625$, the respective operator symbol is D++. The method already provides the operators, so we only need to define whether this aggregation is the type of simultaneity and substitutability. If it is simultaneity, α maps using the andness column; otherwise, it is considered a degree of orness (ω) and maps using the orness column.

4) Scores Calculation: After designing an LSP model, Model Evaluation (i.e., component 8 in Fig. 2(b)) is performed to calculate the score using the generated dataset. We use the Weighted Power Mean (WPM) to calculate the scores, which allowed us to apply the operator in the mean calculation. For each aggregation, there is a summation of all weighted attribute values following the function given by (6).

$$y = \left(\sum_{j=1}^{n} w_j x_j^r \right)^{1/r} \tag{6}$$

where: x_j is the attribute value of the aggregation; w_j is the weight for each attribute value x_j; r is the exponent for WPM based on the logic operator.

Note that r is obtained from a group of operators, such as the GGCD17 group and it depends on two parameters: 1) the degree of simultaneity or substitutability, and 2) the number of attributes in the aggregation (2, 3, 4, 5, or 6+) [33].

In order to calculate scores for all samples in a dataset, we select a particular type of attack, such as a high-volume DoS attack. We then calculate the score for each sample within that attack type. The model generates a score for each sample, which we compare against a defined *threshold* to determine if the sample indicates an attack. If a sample's score exceeds the threshold, the application is considered not under attack; otherwise, the application is deemed under attack.

To determine the threshold, we analyzed the global score output of the golden runs, which provided a reference range for regular scenarios. By averaging the global scores from golden runs, we identified a threshold value that typically signifies a regular sample. This threshold was then empirically verified.

As in case of ML models in the previous section, we compare the model output with the ground truth to generate the confusion matrix and compute the performance metrics.

TABLE X
CRITERIA FOR HIGH-VOLUME DOS ATTACKS

EC1						
Measure	**Criterion Function**	**WebUI**	**Auth**	**Image**	**Persistence**	**Recomme.**
Sockets	lower-the-better	[450,1000]	[40, 140]	[50, 150]	[100, 180]	[10, 70]
Threads	lower-the-better	[400, 500]	[250, 270]	[250, 270]	[200, 240]	[70, 100]
CPU user	lower-the-better	[0.6, 0.8]	[0.3, 0.6]	[0.5, 0.8]	[0.4, 0.8]	[0.2, 0.5]
CPU system	lower-the-better	[0.15, 0.25]	[0.05, 0.1]	[0.11, 0.16]	[0.12, 0.2]	[0.05, 0.08]
Memory	lower-the-better	[0.3, 0.4]	[0.18, 0.2]	[0.2, 0.25]	[0.14, 0.16]	[0.1, 0.11]
Throttled%	lower-the-better	[0.4, 0.8]	[0.1, 0]	[0.05, 0.5]	[0.05, 0.2]	[0.1, 0.11]
CFS throttled	lower-the-better	[10, 30]	[0, 2]	[0, 20]	[0, 5]	[0, 1]
EC2						
Measure	**Criterion Function**	**WebUI**	**Auth**	**Image**	**Persistence**	**Recomme.**
Sockets	lower-the-better	[590,900]	[50, 80]	[50, 150]	[100, 200]	[10, 70]
Threads	lower-the-better	[330, 450]	[250, 270]	[250, 270]	[200, 240]	[70, 100]
CPU user	lower-the-better	[0.5, 0.8]	[0.34, 0.4]	[0.5, 0.8]	[0.4, 0.6]	[0.2, 0.25]
CPU system	lower-the-better	[0.2, 0.25]	[0.05, 0.09]	[0.11, 0.16]	[0.14, 0.2]	[0.02, 0.04]
Memory	lower-the-better	[0.38, 0.41]	[0.0, 0.1]	[0.2, 0.25]	[0.14, 0.16]	[0.0, 0.11]
Throttled%	lower-the-better	[0.4, 0.8]	[0.1, 0.18]	[0.05, 0.2]	[0.02, 0.17]	[0.0, 0.1]
CFS throttled	lower-the-better	[12, 35]	[0, 2]	[0, 5]	[1, 5]	[0, 1]

TABLE XI
WEIGHTS FOR HIGH-VOLUME ATTACKS

Instance Level										
	W1	**W2**	**W3**	**W4**	**W5**	**W6**	**W7**	**W8**	**W9**	**W10**
WeD1	0.7	0.3	0.7	0.3	0.7	0.3	0.3	0.4	0.2	0.1
WeD2	0.9	0.1	0.9	0.1	0.6	0.4	0.35	0.45	0.15	0.05
Application Level										
	W1	**W2**	**W3**	**W4**	**W5**					
WeD1	0.3	0.2	0.2	0.2	0.1					
WeD2	0.3	0.2	0.3	0.1	0.1					

B. Results of the LSP Models

1) High-Volume Attacks: We defined two elementary criteria attributes (EC1 and EC2) which are given in Table X, defined two combinations of weights (WeD1 and WeD2) and two combinations of operators (OpD1 and OpD2). Weights and operators are given in Tables XI and XII, respectively. While other possible combinations exist, their results were not better or substantially different from the ones presented in the paper. We considered the following four configurations:

CFG1: Criteria: **EC1**, Weights: **WeD1**, Operators: **OpD1**.

TABLE XII
OPERATORS FOR HIGH-VOLUME ATTACKS

	OP 1.1	OP 1.2	OP 1.3	1	(AP Level) 1
OpD1	C- (0.2621)	C- (0.2621)	C- (0.2621)	D+- (4.8250)	DA (1.5962)
OpD2	C+- (-0.40845)	CA (-0.7201)	C-+ (0.06705)	D- (3.3180)	D+- (2.21057)

TABLE XIII
LSP RESULTS FOR MULTIPLE CONFIGURATIONS CFG1 - CFG4

Config.	Precision	Recall	F1	Markedness	Informedness
CFG1	0.85±0.19	0.80±0.33	0.77±0.30	0.75±0.25	0.74±0.33
CFG2	0.74±0.27	0.80±0.32	0.73±0.29	0.69±0.28	0.68±0.29
CFG3	0.61±0.25	0.88±0.26	0.67±0.26	0.56±0.26	0.56±0.27
CFG4	0.72±0.24	0.92±0.21	0.77±0.23	0.69±0.24	0.74±0.22

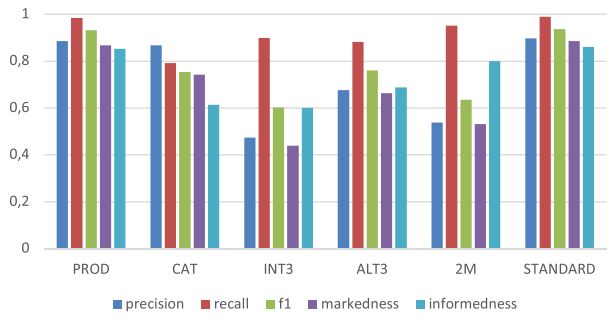

Fig. 8. Performance of CFG4 configuration for high-volume DoS attacks.

TABLE XIV
WEIGHTS AND OPERATORS FOR LOW-VOLUME DOS ATTACKS

Attributes	Weights	Operators	Weights	Operator	Scores	Weights	Operators
Sockets	0.5	C-	0,15	D+-	WebUI	0.5	DA
Threads	0.5				Auth	0.15	
CPU user	0.7	C-	0.4		Persisten.	0.15	
CPU system	0.3				Image	0.1	
Throttled%	0.7	C-	0.1		Recomm.	0.1	
CFS throttled	0.3						
Memory			0.35				

TABLE XV
CRITERIA FOR LOW-VOLUME DOS ATTACKS

Measure	Criterion Function	WebUI	Auth	Image	Persistence	Recomme.
Sockets	lower-the-better	[450,580]	[40, 140]	[50, 150]	[100, 180]	[20, 70]
Threads	range	[300, 400, 400, 500]	[100, 250, 250, 270]	[100, 250, 250, 270]	[150, 240, 240, 300]	[20, 70, 70, 100]
CPU user	higher-the-better	[0.3, 0.5]	[0.2, 0.3]	[0.2, 0.4]	[0.2, 0.4]	[0.1, 0.2]
CPU system	higher-the-better	[0.1, 0.15]	[0.03, 0.05]	[0.05, 0.11]	[0.1, 0.15]	[0.02, 0.05]
Memory	higher-the-better	[0, 0.2]	[0, 0.18]	[0, 0.2]	[0, 0.13]	[0.0, 0.1]
Throttled%	higher-the-better	[0.1, 0.4]	[0.1, 0.2]	[0.1, 0.2]	[0.05, 0.1]	[0, 0.1]
CFS throttled	higher-the-better	[0, 10]	[0, 2]	[0, 0]	[0, 5]	[0, 1]

TABLE XVI
LSP MODEL FOR LOW-VOLUME DOS ATTACKS RESULTS

Threshold	Precision	Recall	F1	Markedness	Informedness
0.90	0.70±0.21	**0.97±0.06**	0.77±0.18	0.64±0.25	0.78±0.24
0.85	**0.87±0.16**	0.94±0.08	**0.89±0.15**	**0.85±0.18**	**0.87±0.19**
0.80	0.08±0.26	0.04±0.18	0.04±0.17	0.07±0.23	0.04±0.18

CFG2: Criteria: **EC2**, Weights: **WeD2**, Operators: **OpD2**.
CFG3: Criteria: **EC2**, Weights: **WeD1**, Operators: **OpD2**.
CFG4: Criteria: **EC1**, Weights: **WeD1**, Operators: **OpD2**.

Regarding the global score threshold, for all the configurations we used 0.70, which was selected based on calculated scores using only the golden runs.

Table XIII displays the results for the four different configurations (CFG1, CFG2, CFG3, and CFG4). The CFG1 configuration had the highest precision score with 0.85. The CFG4 configuration yielded the best recall result with 0.92. As a result, CFG1 and CFG4 achieved the highest F1-scores. CFG1 also led to the best markedness value, while CFG4 had the best informedness result. Note that CFG1 and CFG4 configurations are pretty similar and only differ in the set of operators (i.e, CFG1 uses OpD1, whereas CFG4 uses OpD2).

Fig. 8 presents the results for all six attack profiles (i.e., Standard, Standard-PROD, Standard-CAT, INT3, ALT3, 2 M), using the CFG4 configuration. This figure shows that the performance metrics had considerably different values for the different attack profiles. In general, the profiles for which the attacks lasted 10 minutes (i.e., PROD, CAT, and STANDARD) resulted in excellent results, with all metrics above 0.80 and most of them above 0.90. However, other profiles, such as the 2 M and INT3, had attacks that were challenging to detect because they were very short.

2) Low-Volume Attacks: For the low-volume attacks, we presented a single configuration for the attribute elementary criteria, weights, and operators in the LSP model. While the impact of different configurations was analyzed for high-volume attacks, repeating that analysis here would significantly lengthen this section. Instead, we later present the analysis of the impact of varying thresholds for the global score. The weights and operators are visualized in Table XIV and they match the tree structure in Fig. 7, while Table XV presents the attribute elementary criteria.

In this case we experimented with global score threshold values of 0.9, 0.85, and 0.80 to understand how these thresholds impact the detection performance of the LSP model. The results for each threshold value are presented in Table XVI. As we can observe, the threshold of 0.90 achieved high recall of 0.97, but it resulted in precision of 0.70. When the threshold was 0.85, the recall was slightly lower (0.94), but the precision was considerably higher (0.87). The 0.80 threshold value resulted in almost zero for all metrics.

3) Transferability Between LSP Models: LSP enables the construction of a model that aggregates the outputs from two or more models. Therefore, LSP allowed us to develop a unified model composed of models for high-volume and low-volume attacks. We used our previously defined models for high-volume and low-volume DoS attacks. Then, we merged the global scores from both models to create a new final global score. For low-volume attacks, we utilized the values shown in Table XIV, while for high-volume attacks we used the CFG1 configuration that yielded the best results in terms of F1-score. To perform the aggregation, we applied the neutrality operator, defined as the arithmetic mean, which reflects a balance between substitutability and simultaneity. Given that the models operated under different thresholds for attack classification (0.85 for low-volume and 0.70 for high-volume attacks), we balanced these thresholds using weights of [0.4, 0.6] for high-volume

TABLE XVII
LSP RESULTS FOR HIGH-, LOW-VOLUME, AND COMBINED ATTACKS

	Precision	Recall	F1	Markedness	Informedness
High-volume	0.74±0.13	0.95±0.06	0.81±0.09	0.70±0.11	0.77±0.06
Low-volume	0.73±0.16	0.83±0.22	0.76±0.18	0.68±0.19	0.72±0.21
Overall Avg.	**0.74±0.19**	**0.91±0.17**	**0.79±0.19**	**0.70±0.22**	**0.75±0.21**

and low-volume attacks, respectively. The final aggregation is represented by (7).

$$R = \begin{cases} 0, & 0.6 \times LVScore + 0.4 \times LVScore \geq 0.85 \\ 1, & 0.6 \times LVScore + 0.4 \times HVScore < 0.85 \end{cases} \quad (7)$$

where *R* is the final result for an observation, and *LV_Score* and *HV_Score* are the global score for low- and high-volume models, respectively.

Results in Table XVII show the average performance for high- and low-volume attacks, and the overall average for both types of attacks. The results showed that the higher threshold applied to the high-volume model led to higher recall, but at the cost of more false positives, resulting in a lower precision. On the other hand, all performance metrics for low-volume attacks showed a decline, likely due to the fact that the low-volume model was focused on recall. Nevertheless, the overall performance, for combined attacks was comparable to that of unsupervised ML algorithms, yielding an average precision of 0.74, recall of 0.91, and F1-score of 0.79.

We identified two key points that may help building a better unified model, capable of handling multiple attack types. First, the current models were developed independently, each focusing on a specific attack type, resulting in different threshold values for final scores (0.70 for high-volume and 0.85 for low-volume). Ideally, both models should produce similar global scores with more closely aligned thresholds. This can be achieved by re-defining the LSP components to ensure the values are more closely ranged. Second, the models were designed based on 'higher-is-better' for classifying the application as regular. This limited the use of an appropriate disjunction operator, which could have accounted for a higher degree of substitutability between the models. Future work that would address these issues could enhance the model's robustness and performance across different attack types.

VII. DISCUSSION

In this section, we summarize our results and discuss the generalizability of the framework.

A. Implications and Recommendations

Table XVIII presents the metrics for the best performing supervised and unsupervised ML models, and LSP models.

Our framework supports attack data generation, and development and evaluation of models that detect attacks on microservice applications: Its versatility allows incorporation of various attack types, supports developing models based on diverse methods, and facilitates comprehensive evaluations for distinct scenarios, helping optimizing attack detection performance. The framework is illustrated on DoS attacks.

TABLE XVIII
RESULTS OF THE BEST PERFORMING MODELS

Model	Attack	Precision	Recall	F1	Marked.	Informed.
XGBoost	HV	**0.98**	0.98	**0.98**	**0.97**	**0.97**
XGBoost	LV	**0.98**	0.94	**0.96**	**0.97**	**0.94**
XGBoost	Both	**0.98**	**0.98**	**0.98**	**0.97**	**0.97**
VAE-S&T	HV	0.85	**0.99**	0.91	0.83	0.90
AE-CPU	LV	0.84	0.92	0.86	0.78	0.76
VAE	Both	0.76	0.93	0.82	0.72	0.79
ISOF-CPU	Both	0.75	0.92	0.81	0.72	0.78
LSP (CFG1)	HV	0.85	0.80	0.77	0.75	0.74
LSP (CFG4)	HV	0.72	0.92	0.77	0.69	0.74
LSP (t=0.85)	LV	0.87	**0.94**	0.89	0.85	0.87
LSP	Both	0.74	0.91	0.79	0.70	0.75

(HV and LV stand for high-voume and low-volume DoS attacks, respectively).

Determining the proper configurations to generate the data requires significant effort: The main challenge was preparing the setup, which involved defining appropriate request rates and resource limits for each service within the TeaStore application, as well as carefully designing the workload and attack profiles. This preparation proved to be significantly more time-intensive than the actual data generation.

Considering different attack profiles allows designing solutions that deal with a broad range of attack possibilities: Results showed that classification performance varies across different attack profiles and attack types. This phenomenon resulted in using different models and/or model parameters to achieve appropriate attack detection performance.

Standard attack profiles were successfully detected: These are persistent 10-minute attacks during which the system eventually fails to deliver its services, allowing the models to identify the attacks effectively. However, detection performance varied among profiles, with STD-Home and STD-PROD achieving the best results.

INT3 attack profile was the most challenging for most models, which suggests that, in general, all types of models struggle to detect short-duration attacks effectively. One possible explanation is that these attacks can momentarily disrupt the system, making the detection harder. Even after the attack ceases, the residual effects on the application may remain, leading to the incorrect classification of samples. Additionally, the rapid recurrence of the attacks, soon after the system begins to recover, complicates the detection efforts even more. These observations emphasize the need for more effective strategies for identification of short, intermittent attacks and mitigation of their effect on the system.

Supervised ML outperforms unsupervised ML and LSP, but requires the generation of attack data: The supervised ML algorithms outperformed both the unsupervised ML algorithms and the LSP models, with XGBoost performing the best, followed by RF for both high- and low-volume DoS attacks. However, unlike unsupervised ML and LSP, supervised ML algorithms require attack data to build the models, which may not be available for all applications.

XGBoost is a recommended algorithm for developing attack detection models for microservice applications: XGBoost had F1-scores of 0.98 for high-volume DoS attacks, 0.96 for low-volume DoS attacks and 0.98 when training with both types

of attacks together. ISOF-CPU attained F1-scores of 0.83, 0.80, and 0.81 for high-volume, low-volume, and both types of attacks, respectively, which is still good but significantly worse than XGBoost. LSP models had worse performance than unsupervised ML models in case of high-volume and both types of attacks, but outperformed them in case of low-volume attacks.

Unsupervised ML algorithms can be used to build anomaly detection models for both high- and low-volume DoS attacks with satisfactory performance: When using grouped features, ISOF algorithm with CPU feature produced the best results. However, when using the full feature set, the VAE model outperformed all others, achieving a precision of 0.76, recall of 0.93, and F1-score of 0.82. More in depth investigation showed that the CPU feature was the most effective for attack detection because low-volume attacks resulted in decreased CPU usage, while high-volume attacks led to increased CPU usage. In comparison, the number of sockets increased during high-volume attacks as well as during low-volume attacks, making it less sensitive and consequently less effective for attack detection.

ML models have an advantage over LSP because they can detect both high- and low-volume attacks using the same model, unlike the LSP method which requires defining different attribute criteria, weights, operators, and thresholds for each type of attack. *In cases when the attack data are not available and the resources are limited, our recommendation is to use unsupervised ML algorithms (such as ISOF) which produce one model with a small number of features for both high- and low-volume attacks.*

VAE and AE are promising unsupervised deep ML algorithms for detecting DoS attacks: For high-volume attacks, VAE achieved a recall of 0.99 and F1-score of 0.91, effectively identifying most attack instances, which makes it suitable when minimizing the false negatives is critical. Future work focused on expanding the feature space has potential to improve the precision in case of high-volume attacks, and enhance the detection of low-volume attacks.

LSP allows the aggregation of existing LSP modes into new models, providing flexibility for updating and adding models as needed to address emerging threats without requiring training data. This adaptability enables the same LSP configuration to be used to detect various types of attacks, including previously unseen attacks. However, as shown, for the LSP models to perform effectively, they must be constructed with attributes and thresholds within similar ranges.

When selecting the type of model, in addition to performance, several trade-offs need to be considered: Using ML is easier, more straightforward than LSP, but LSP is easier to customize by changing the weights and operators. Thus, the selection of the operators can alter model's purpose. For example, the CFG1 configuration is suitable for applications that require high precision, while the CFG4 configuration is more appropriate for critical applications that require high recall. On the cons side, LSP models are highly sensitive to changes. For instance, even a slight change in the threshold value for the global score can significantly change the outcome.

Defining an appropriate LSP model is even more challenging for microservice applications because it is necessary to define suitable elementary attribute criteria for each type of service. Our recommendation is to use LSP models when more human resources are available to analyze the system behavior, define the model components utilizing a user interface with precision, and create different models.

B. Framework Generalizability

1) Application Transferability: Although our framework was validated using TeaStore, its modular and generalizable design ensures broad applicability to other microservice applications, due to the fact that it separates the microservice application from monitoring, data generation, and model development. Workload profiles are customizable and can be tailored to different applications through user behavior modeling, which involves identifying and mapping actions that trigger specific services. Similarly, the attack injection and monitoring components are technology-agnostic and support different applications. Feature selection and model development rely on system-level features, commonly available in containerized and Kubernetes-based setups, and are independent of application logic. As a result, the framework can be applied to a range of microservice applications, including TrainTicket [23], SockShop, or real-world production microservices, by adapting workload profiles, defining service-specific actions, and selecting relevant attack types.

2) Fitting Other Types of Attacks: This work focuses on DoS attacks to illustrate the framework, but the *Attack Injection (2) component is modular and adaptable*. Appendix B, available online describes types of attacks and their integration with our framework. For example, API abuse can be injected using fuzzing tools or scripted requests that target specific service endpoints. Service spoofing and Adversary-in-the-Middle (AitM) can be simulated by deploying malicious service replicas. Supply chain attacks can be modeled by deploying compromised containers. Each scenario can be defined by a specific attack profile and executed during a designated attack window in a run. Thus, the framework enables the creation of diverse and realistic attack types beyond DoS, allowing future research to explore a broader range of microservice threat landscape. To illustrate this, we implemented an API fuzzing attack targeting the login endpoint by injecting malformed inputs. The attack follows the same profile-based mechanism and is available at http://doi.org/10.5281/zenodo.11204757.

3) Framework Across Different Platforms: Our experimental setup was based on local virtual machines, providing a controlled and reproducible environment. Nonetheless, the framework described in Section III was deliberately designed to be platform-agnostic, enabling its deployment across various execution environments, including public cloud platforms and container-based infrastructures. All components used to illustrate the framework (Kubernetes, JMeter, cAdvisor, and the attack tools) are compatible with cloud environments. These tools can be seamlessly deployed on managed Kubernetes services, e.g., Amazon EKS, Google GKE, and Azure AKS.

Moreover, the framework supports deployment in microservice-based applications that allow horizontal and

vertical scaling. Concerning the data generation experiments (Section IV), the framework can be readily adapted to incorporate scalability scenarios. This adaptation only requires the definition of new workload profiles capable of exercising the application in a way that triggers service scaling operations. In contrast, the detection models (Sections V and VI), as currently designed, may not perform effectively under such conditions. Future work will investigate how both existing and newly developed models can manage the scalability of microservice applications.

4) Scalability and Real-World Representativeness: TeaStore is a relatively lightweight application and that our experiments were conducted in a fixed, non-elastic environment without dynamic scaling, load balancing, or failure recovery mechanisms. These aspects are critical in production-grade microservice environments. However, our primary objective was to validate the framework's functionality, including its support for data generation, monitoring, attack injection, and detection model evaluation in microservice-based systems. While the results obtained from our experiments are not intended to generalize across all microservice deployments, as discussed in Section VIII, the current setup suffices to demonstrate the framework's modular and extensible design.

The framework supports experiments involving scalable applications. For instance, using Kubernetes-native features such as the Horizontal Pod Autoscaler (HPA), researchers can configure microservices to scale dynamically in response to varying workload intensities. Our framework accommodates this through customizable workload profiles and deployment configurations, which are crucial when evaluating detection models in more complex, elastic environments. Although TeaStore is simpler than applications like TrainTicket, which includes over 40 services, it remains a commonly adopted benchmark for microservices research and is suitable for validating our framework across different research contexts.

VIII. Threats to Validity

Construct Validity: We used the TeaStore browser profile for user emulation, as recommended in [9]. This workload profile is representative since it includes requests to various TeaStore web pages. Using other workload profiles, however, may impact the services differently. Therefore, future work needs to assess the impact of using different workload profiles.

We used multiple supervised and unsupervised ML algorithms, but many others could be used. These unexplored algorithms may lead to different results. To prevent overfitting, we used cross-validation strategies, such as GroupKFold, and tuned hyperparameters using grid search. Regarding class imbalance, our dataset consists of 31% attack samples and 69% regular samples. This moderate imbalance did not have a significant impact on the models' performance. Each run produces a limited number of samples, approximately 77. To ensure data sufficiency, we adopted ELOOCV (Leave-One-Out Cross-Validation). This approach maximizes training data by leaving out one attack run for testing while using the rest for training. ELOOCV is particularly effective when the number of samples per run is small, reduces overfitting and improves model generalization across attack instances. To further validate the sufficiency of training data, we conducted an experiment in which the training set size was gradually increased from 10% to 100% of the available data. The observed changes in performance were negligible (under 3%).

Regarding the LSP method, many possible models with different structures, attributes, weights, criteria, and operators could be explored. Despite this, we thoroughly explored these parameters to define a suitable set of LSP components for DoS attack detection. To address improper feature selection for ML and LSP, we conducted a thorough analysis of all available features in previous works [11], [12] and limited the feature set provided by the cAdvisor monitor to those most relevant for attack detection. Finally, training data quality was ensured by careful labeling based on the timestamps from attack scripts to distinguish between regular and attack periods. Each sample was labeled as regular (before and after an attack) or as attack samples. Although there might be a slight delay in the timestamps generated by the scripts, this delay is less than one second and is unlikely to affect the results, as the monitored features are available on average every 15 seconds.

Internal Validity: Experiments where conducted in a controlled environment, utilizing a local physical machine. To ensure similar conditions across all experiments, the VMs were reset to identical states for each run. However, during each run one VM, such as the MasterVM, may be affected by another VM, such as the ClientVM. Additionally, we cannot guarantee the perfect functioning of the VMs and Kubernetes clusters used. However, we thoroughly analyzed the runs to identify potential issues and repeated the experiments when necessary.

The randomness in the ML models can impact the performance results due to variations in random initialization, data splits, and optimization processes. To mitigate this threat, we used the following practices: (1) *Random State Initialization* - we set a fixed seed value of 42 across experiments to ensure reproducibility of results and consistency in model initialization. This was used for supervised models during the shuffling and grouping of files in the ELOOCV process, ensuring each model to be trained and tested on the same groupings. For ISOF, it was used to ensure consistent results across different executions; (2) *Deterministic Execution in TensorFlow* - when using TensorFlow for AE and VAE models, we enforced deterministic behavior using enable_op_determinism command; and (3) *Performance Reporting* - following best practices, we reported the average performance over multiple runs, along with standard deviations to capture variability.

External Validity: We chose the TeaStore application as a representative microservice application for our experiments. The TeaStore application is a well-established reference application in microservice research which has been extensively used by many researchers in numerous papers and studies [5], [10]. Nevertheless, we cannot claim that the models will behave the same for other microservice applications. Note, however, that the models (e.g., LSP models) can be customized to adapt to other microservice applications. Furthermore, this paper considered two types of DoS attack, utilizing two client-side tools with

their own limitations. Despite including two types of attacks and a variety of attack profiles, we cannot claim that our results will generalize to other types of attacks and attack scenarios. However, regardless of the threats, the results illustrate and validate the framework.

We considered one instance per service of the TeaStore application, so we did not assess microservices under dynamic scaling. Since the models were trained in a static environment, their effectiveness in autoscaling situations still requires validation. Additionally, this dynamic behavior presents new challenges, particularly concerning noise in monitored data. This noise can derive from factors such as short-lived instances that may not generate sufficient data, feature inconsistencies, and auto-scaling that alters the feature space. Also, variability in resource metrics, like CPU usage and the number of sockets, complicates the interpretation of monitoring data. Future work will incorporate features that capture scaling events, such as the number of active pods per service, and propose models for attack detection in elastic workloads.

IX. Conclusion and Future Work

This paper introduced a framework designed to assist in the attack data generation and development of attack detection models for microservice applications. We illustrated and validated the framework by generating two types of DoS attacks (high- and low-volume) and creating diverse models using ML and LSP methods to detect these attacks.

Our results showed that both ML and LSP are effective, with supervised ML algorithms performing the best, specifically when using XGBoost. The unsupervised ML algorithms ISOF and LOF produce good anomaly detection models for both types of attacks when trained with CPU usage features. LSP models are highly adaptable and do not require attack data to train but require more effort to tune than the ML models.

In our future work, we plan to use our framework for elastic microservice applications, where the number of replicas for each service changes depending on the intensity of clients' requests. To address the complexity of dynamic and scalable scenarios, we will expand our feature space to include features related to network behavior, inter-service communication, and active replicas. Additionally, we aim to explore larger set of workload profiles to represent different user behaviors. We will also explore the feasibility of using other ML algorithms for detecting attacks on scalable applications. Finally, we aim to integrate additional attack types into the framework to support more diverse attack scenarios and model development.

References

[1] V. Velepucha and P. Flores, "A survey on microservices architecture: Principles, patterns and migration challenges," *IEEE Access*, vol. 11, 2023, Art. no. 88339.

[2] S. Jacob, Y. Qiao, Y. Ye, and B. Lee, "Anomalous distributed traffic: Detecting cyber security attacks amongst microservices using graph convolutional networks," *Comput. Secur.*, vol. 118, 2022, Art. no. 102728.

[3] M. Waseem, P. Liang, M. Shahin, A. Di Salle, and G. Márquez, "Design, monitoring, and testing of microservices systems: The practitioners' perspective," *J. Syst. Softw.*, vol. 182, 2021, Art. no. 111061.

[4] O. Y. Pacheco, O. Yoachimik, J. Pacheco, and J. Tomé, "DDoS threat report for 2023 Q4," 2024. [Online]. Available: https://blog.cloudflare.com/ddos-threat-report-2023-q4

[5] J. Grohmann, P. K. Nicholson, J. O. Iglesias, S. Kounev, and D. Lugones, "Monitorless: Predicting performance degradation in cloud applications with machine learning," in *Proc. 20th Int. Middleware Conf.*, 2019, pp. 149–162.

[6] M. Odusami, S. Misra, O. Abayomi-Alli, A. Abayomi-Alli, and L. Fernandez-Sanz, "A survey and meta-analysis of application-layer distributed denial-of-service attack," *Int. J. Commun. Syst.*, vol. 33, no. 18, 2020, Art. no. e4603.

[7] C. Cao, A. Blaise, S. Verwer, and F. Rebecchi, "Learning state machines to monitor and detect anomalies on a kubernetes cluster," in *Proc. 17th Int. Conf. Availability, Rel. Secur.*, 2022, pp. 1–9.

[8] OECD, "Oecd framework for the classification of AI EUR," *OECD Digit. Economy Papers*, vol. 2021, no. 323, pp. 1–106, 2022. [Online]. Available: https://www.oecd-ilibrary.org/content/paper/cb6d9eca-en

[9] J. Von Kistowski, S. Eismann, N. Schmitt, A. Bauer, J. Grohmann, and S. Kounev, "TeaStore: A micro-service reference application for benchmarking, modeling and resource management research," in *Proc. IEEE 26th Int. Symp. Model., Anal., Simul. Comput. Telecommun. Eur.*, 2018, pp. 223–236.

[10] S. Caculo, K. Lahiri, and S. Kalambur, "Characterizing the scale-up performance of microservices using teastore," in *Proc. 2020 IEEE Int. Symp. Workload Characterization*, 2020, pp. 48–59.

[11] J. Castro, N. Laranjeiro, and M. Vieira, "Detecting DoS attacks in microservice applications: Approach and case study," in *Proc. 11th Latin-Amer. Symp. Dependable Comput.*, 2022, pp. 73–78.

[12] J. Castro, N. Laranjeiro, and M. Vieira, "Exploring logic scoring of preference for DoS attack detection in microservice applications," in *Proc. IEEE Int. Conf. Web Serv.*, 2023, pp. 573–584.

[13] J. Castro, N. Laranjeiro, and M. Vieira, "Generating realistic attack data for microservices: Framework and case study," in *Proc. 12th Latin- Amer. Symp. Dependable Secure Comput.*, 2023, pp. 60–69.

[14] A. F. Baarzi, G. Kesidis, D. Fleck, and A. Stavrou, "Microservices made attack-resilient using unsupervised service fissioning," in *Proc. 13th Eur. workshop Syst. Secur.*, 2020, pp. 31–36.

[15] Q. Du, T. Xie, and Y. He, "Anomaly detection and diagnosis for container-based microservices with performance monitoring," in *Proc. Algorithms Architectures Parallel Process.: 18th Int. Conf., Proceedings, Part IV 18*, 2018, pp. 560–572.

[16] A. Samir and C. Pahl, "Detecting and localizing anomalies in container clusters using markov models," *Electronics*, vol. 9, no. 1, 2020, Art. no. 64.

[17] M. Panahandeh, A. Hamou-Lhadj, M. Hamdaqa, and J. Miller, "Serviceanomaly: An anomaly detection approach in microservices using distributed traces and profiling metrics," *J. Syst. Softw.*, vol. 209, 2024, Art. no. 111917.

[18] Y. Chen, M. Yan, D. Yang, X. Zhang, and Z. Wang, "Deep attentive anomaly detection for microservice systems with multimodal time-series data," in *Proc. IEEE Int. Conf. Web Serv.*, 2022, pp. 373–378.

[19] P. Wang, X. Zhang, Z. Cao, and Z. Chen, "Madmm: Microservice system anomaly detection via multi-modal data and multi-feature extraction," *Neural Comput. Appl.*, vol. 36, no. 25, pp. 15739–15757, 2024.

[20] J. Flora and N. Antunes, "Evaluating intrusion detection for microservice applications: Benchmark, dataset, and case studies," *J. Syst. Softw.*, vol. 216, 2024, Art. no. 112142.

[21] Z. Ding and Q. Huang, "COPA: A combined autoscaling method for kubernetes," in *Proc. IEEE Int. Conf. Web Serv.*, 2021, pp. 416–425.

[22] Z. Xiao and S. Hu, "DScaler: A horizontal autoscaler of microservice based on deep reinforcement learning," in *Proc. 23rd Asia-Pacific Netw. Oper. Manag. Symp.*, 2022, pp. 1–6.

[23] X. Zhou et al., "Benchmarking microservice systems for software engineering research," in *Proc. 40th Int. Conf. Softw. Eng.: Comput.*, 2018, pp. 323–324.

[24] M. C. Calzarossa, L. Massari, and D. Tessera, "Workload characterization: A survey revisited," *ACM Comput. Surv.*, vol. 48, no. 3, pp. 1–43, 2016.

[25] Imperva, Inc., "2021 DDoS threat landscape report," Imperva, Inc., Tech. Rep., 2021, accessed: Nov. 11, 2024. [Online]. Available: https://www.imperva.com/resources/reports/Imperva_2021-DDoS-Report.pdf

[26] Y. Wang, H. Kadiyala, and J. Rubin, "Promises and challenges of microservices: An exploratory study," *Empirical Softw. Eng.*, vol. 26, no. 4, 2021, Art. no. 63.

[27] L. Giamattei et al., "Monitoring tools for devops and microservices: A systematic grey literature review," *J. Syst. Softw.*, vol. 208, 2023, Art. no. 111906.

[28] H. Alibrahim and S. A. Ludwig, "Hyperparameter optimization: Comparing genetic algorithm against grid search and Bayesian optimization," in *Proc. IEEE Congr. Evol. Comput.*, 2021, pp. 1551–1559.
[29] B. Venkatesh and J. Anuradha, "A review of feature selection and its methods," *Cybern. Inf. Technol.*, vol. 19, no. 1, pp. 3–26, 2019.
[30] G. A. Jaafar et al., "Review of recent detection methods for http DDoS attack," *J. Comput. Netw. Commun.*, vol. 2019, 2019, Art. no. 1283472.
[31] , "Pandas 2.2.2 documentation — pandas.dataframe.bfill," 2024, Accessed: May 06, 2024.https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.bfill.html
[32] J. R. Campos, E. Costa, and M. Vieira, "Online failure prediction through fault injection and machine learning: Methodology and case study," in *Proc. IEEE 34th Int. Symp. Softw. Rel. Eng.*, 2023, pp. 451–461.
[33] J. Dujmovic, *Soft Computing Evaluation Logic: The LSP Decision Method and its Applications*. Hoboken, NJ, USA: Wiley, 2018.

Jessica Maciel de Castro received the BSc degree in computer science and the MSc degree in informatics from the Federal University of Paraíba (UFPB), Brazil, in 2016 and 2019, respectively. She is currently working toward the PhD degree in computer science with the University of Coimbra, Portugal. Her research focuses on dependable and secure software, with particular interest in software engineering, attack detection, and microservices-based applications. She is a member of the Software and Systems Engineering (SSE) Group with CISUC. She has contributed to research projects and published work in the areas of service-based systems, dependability, and security.

Nuno Laranjeiro (Member, IEEE) received the Habilitation and PhD degrees from the University of Coimbra, Portugal, where he currently is an associate professor. His research focuses on dependable and secure software services. His research interests include experimental dependability evaluation, fault injection, robustness of software and web services, web services interoperability, and software security. He is currently mostly involved in developing new techniques towards more reliable and secure cloud systems and microservices, in developing techniques for evaluating the reliability and security of blockchain smart contracts, and using artificial intelligence for software fault and vulnerability management in the software development lifecycle. He has contributed, as an author, reviewer and program committee member to leading conferences and journals in the dependability and services computing areas. Nuno serves on the Program Board and served as PC Chair for the International Symposium on Software Reliability Engineering (ISSRE). He is a member of the Senior PC of the IEEE International Conference on Web Services (ICWS). Nuno has been involved in the organisation of several international events, including several editions of ISSRE. He acted as PC Chair for ISSRE 2023 and LADC 2024. He was guest editor for journals like the Journal of Systems and Software or the Software Quality Journal. He participated in international research projects, including several H2020 projects and FP7 projects.

Dr. Katerina Goseva-Popstojanova (Senior Member, IEEE) is a professor with the Lane Department of Computer Science and Electrical Engineering, West Virginia University, Morgantown, WV. Her research interests are in secure software, software reliability, software quality assurance, intelligent malware detection, intrusion tolerance and resilience, and applied data analytics. She has published more than 120 refereed journal and conference articles on these topics. She received the National Science Foundation (NSF) CAREER award and has served as a Principal Investigator on various NSF, NASA, Department of Defense, Department of Education, NASA West Virginia Space Grant Consortium, and industry funded projects. She is a program director of Undergraduate Studies in Cybersecurity and an Academic Coordinator of the Master of Science in Software Engineering Program at West Virginia University. She is a member of the Steering Committee of the IEEE International Symposium on Software Reliability Engineering (ISSRE) and has served as associate editor of the *IEEE Transactions on Reliability* from 2017 to 2025. She was general co-chair of ISSRE 2022, and a program committee co-chair of QRS 2021, WoSAR 2010, and ISSRE 2007. She also served as guest editor of *Special Sections of the IEEE Transactions on Software Engineering,* and *IEEE Transactions on Reliability*, and on program and organizing committees of many international conferences and workshops.

Marco Vieira (Member, IEEE) is a professor of computer science with the University of North Carolina, Charlotte. Before joining UNC Charlotte in 2023, he was a professor with the University of Coimbra. His research interests include dependable computing, dependability and security assessment and benchmarking, software security, fault and vulnerability injection, failure prediction, static analysis, and software testing, subjects in which he authored or co-authored more than 260 works in refereed conferences and journals. Marco is chair of the IFIPWG10.4 on Dependable Computing and Fault Tolerance, associate editor of the *IEEE Transactions on Dependable and Secure Computing*, Steering Committee vice-chair of the IEEE/IFIP International Conference on Dependable Systems and Networks, and member of the Steering Committee of the IEEE International Symposium on Software Reliability Engineering, the International Symposium on Reliable Distributed Systems, and the Latin-American Symposium on Dependable and Secure Computing. He served as program chair for the major conferences on the dependable computing area.