Closed-loop Automated Detection System on Distributed Denial of Service (DDoS) Attacks via Machine Learning and Deep Learning Models

Oscar Hernandez, Rooj Rinchokechai, Supak Kunopasvorakul, Yogadhveep Arora, and Yuhan Liang

School of Electrical, Electronic and Mechanical Engineering, University of Bristol

Summary

The project focuses on developing a closed-loop automated detection system of Distributed Denial-of-Service (DDoS) attacks. DDoS attacks are one of the most significant threats to current online services. These attacks flood websites and servers with malicious traffic, making them unavailable for users to access. Detection and identification of DDoS attacks can aid in minimising such disruption to online services.

In this project, there are four key components. A pipeline has been developed that preprocesses the data. A pipeline has been developed that trains and evaluates Machine Learning models on known network traffic. Another pipeline for the simulation of DDoS attacks has also been developed, which the models are tested on. A final pipeline integrating all these components together in a closed-loop autonomous system. These simulations have been made to replicate a variety of real-world DDoS scenarios. This ensures that the models can identify a diverse range of DDoS attack patterns.

The findings from this project demonstrate that a closed-loop automated Machine and Deep Learning system can enhance the detection of DDoS attacks. The project therefore provides a foundation for online services to defend against such network-based threats, while also highlighting the potential of Artificial Intelligence (AI) within cybersecurity.

Supervised by
Dr. Yulei Wu

School of Electrical, Electronic and Mechanical Engineering
University of Bristol

2025

Declaration

[illegible]

Oscar Hernandez

[illegible]

Rooj Rinchokechai

[illegible]

Supak Kunopasvorakul

Yogadhveep

Yogadhveep Arora

[illegible]

Yuhan Liang

Certification of ownership of the copyright

Work Allocation

Weighting scheme: Sections that involve hands-on implementation (overall pipeline, data preparation, ML development, simulation, component integration, and results & evaluation; i.e. Sections 3–8) are weighted according to the actual development effort (coding, experiments, pipeline setup).

Sections focused primarily on narrative, analysis or overview (Introduction, Literature Review, Discussion, Conclusion; i.e. Sections 1–2, 9–10) are weighted according to the amount and depth of written content, and so carry proportionally lower percentages in the table below.

Task / Section	**YL**	**RR**	**SK**	**OH**	**YA**
1] Summary	-	–	–	–	100%
2] Introduction	100%	–	–	–	–
3] Literature Review	20%	20%	20%	20%	20%
4] Overall Pipeline	20%	20%	20%	20%	20%
5] Dataset Acquisition	100%	–	–	–	–
6] ML Development	–	50%	50%	–	–
7] Simulation Environment	–	–	–	50%	50%
8] Component Integration	20%	20%	20%	20%	20%
9] Results and Evaluation	–	50%	50%	–	–
10] Discussion	33.3%	–	–	33.3%	33.3%
11] Conclusion	–	–	–	100%	–

We confirm that the information on this page accurately describes our individual contributions to the project.

Oscar Hernandez (OH)
Rooj Rinchokechai (RR)
Supak Kunopasvorakul (SK)
Yogadhveep Arora (YA)
Yuhan Liang (YL)

Contents

1 Introduction

1.1 Background & Motivation

Distributed Denial of Service (DDoS) attacks have escalated significantly in frequency, scale, and sophistication in recent years, posing critical threats to global digital infrastructure. In 2024, Cloudflare's autonomous defense systems mitigated approximately 21.3 million DDoS attacks, representing a 53 % increase compared to 2023.(1) In January 2025, DeepSeek experienced a series of large-scale DDoS incidents involving botnets such as HailBot and RapperBot, resulting in severe service disruptions and the suspension of new user registrations.(2)

These incidents illustrate the growing limitations of traditional detection systems, which often fail to adapt to the rapid evolution of attack methodologies. Modern DDoS attacks increasingly exploit a combination of volumetric flooding, protocol manipulation, and application-layer vulnerabilities, creating complex threat landscapes that static, rule-based systems struggle to address. Consequently, there is an urgent need for automated detection frameworks that leverage machine learning to classify diverse attack types accurately and adapt dynamically to emerging threats.

1.2 Problem Statement

Current DDoS detection frameworks face two fundamental limitations: insufficient generalisability across diverse attack types and inadequate adaptability to rapidly evolving threat patterns. Models frequently trained on outdated or narrowly scoped datasets exhibit elevated false positive rates and demonstrate poor resilience to hybridised or unseen attacks. Furthermore, the absence of continuous feedback from simulated environments restricts their ability to self-adapt in real time to new attack behaviours.

This project seeks to address these challenges by developing a generalisable and self-adaptive DDoS detection system. The system aims to maintain high classification accuracy across volumetric, protocol-based, and application-layer attacks, while dynamically integrating simulation feedback to iteratively refine its performance. By leveraging recent datasets and realistic traffic simulations, the project aspires to enhance both the robustness and adaptability of machine learning-based DDoS detection mechanisms.

1.3 Objectives

The primary objective of this research is to develop an automated, machine learning-based detection system capable of accurately identifying and classifying various forms of DDoS attacks. Specifically, the project pursues the following goals:

- Implement robust machine learning models trained on comprehensive and representative network traffic datasets.
- Establish an iterative ML–Simulation integration pipeline to enable dynamic refinement and optimisation of model performance.
- Ensure models demonstrate high adaptability to newly emerging and evolving DDoS attack patterns.
- Develop a closed-loop, autonomous detection framework capable of continuous self-improvement through simulated feedback mechanisms.

1.4 Scope and Structure

This project focuses on building an automated machine learning–based DDoS detection system using both real-world and simulated data. It covers data acquisition, preprocessing, model development, simulation design, system integration, and evaluation.

The report is structured as follows: a review of related work (Section 2), system architecture and pipeline design (Section 3), dataset selection and feature engineering (Section 4), model development and simulation setup (Sections 5–6), system integration (Section 7), testing and evaluation (Section 8), followed by discussion, limitations, and future work (Sections 9–10). References and acknowledgements are included in Section 11.

1.5 Potential Challenges

Each phase of this project presents distinct challenges. Obtaining sufficiently detailed and representative datasets remains difficult, as publicly available datasets often suffer from outdated attack profiles, severe class imbalance, and a lack of real-time traffic complexity. Additionally, the large size of raw network traffic files imposes substantial computational demands, constraining both data preprocessing and model training efficiency. These data acquisition issues directly impact model development, as training robust machine learning models under such conditions may result in suboptimal accuracy, particularly when distinguishing between closely related attack types or minority classes. Addressing these concerns necessitates careful dataset handling, advanced feature engineering, and the selection of appropriate evaluation metrics.

Furthermore, the simulation component of the project introduces its own set of challenges: designing and executing realistic DDoS attacks must comply with strict legal and ethical regulations. Ensuring operational safety while generating meaningful and diverse attack traffic requires careful balancing between simulation realism and compliance constraints.

2 Literature Review / Background

2.1 DDoS Attacks and Data Research

2.1.1 Multi-Vector DDoS Attack Detection

Recent developments in DDoS detection research highlight a critical shift from binary attack detection towards multi-class classification systems. Modern DDoS threats, including volumetric floods (e.g., UDP, ICMP), protocol-based attacks (e.g., TCP SYN, ACK floods), and application-layer attacks (e.g., Slowloris, HTTP floods), often combine multiple techniques simultaneously to overwhelm targets. This multi-vector approach exploits different network layers concurrently, enhancing attack potency and complicating detection. For instance, a single campaign may integrate a UDP flood with an NTP amplification attack to bypass single-layer defences. (3)

Such complexity necessitates detection models capable of distinguishing subtle anomalies across diverse protocols and traffic features. Furthermore, it has been consistently reported that the majority of modern DDoS incidents involve multiple coordinated attack types within a single event.(4) As a result, multi-class classification provides greater operational value by enabling systems to differentiate between various attack categories. Consequently, the datasets used for machine learning training must contain a wide range of attack types with accurate labelling, allowing models to learn the distinct characteristics of different DDoS behaviours and enhancing their adaptability to complex threat environments.

2.1.2 Requirements for Datasets and Existing Resources

Various criteria for selecting DDoS detection datasets have been highlighted in the literature. Datasets should include real-world network traffic and encompass a diverse range of attack types and benign behaviours.(5) They must be reliably labelled for training and evaluation. Temporal features are another important consideration, as many public intrusion datasets lack time-series attributes.(6) Additionally, dataset size should be large enough to capture complex patterns but remain manageable—extremely large datasets can require substantial preprocessing and computational resources (e.g., GPUs).

Following these criteria, several widely used datasets were reviewed. Table 1 (7) summarises their key characteristics and trade-offs, and will guide our final dataset selection in Section 4.

Table 1: Summary and comparison of DDoS detection datasets

Dataset	Strengths	Limitations
UNSW-NB15	Modern attacks and broad (49) feature set with labelled flows captured via realistic testbed.	No distributed attacks included; generated in a controlled (synthetic) environment.
CIC-IDS2017	Multiple days of traffic with diverse attacks and realistic user behaviours.	Severe class imbalance; very large volume (millions of flows).
CSE-CIC-IDS2018	Wide attack coverage, includes internal infiltration scenarios and updated tools (e.g. LOIC/HOIC).	Multi-attack dataset requiring extensive preprocessing; suffers similar class imbalance issues.
BoT-IoT	IoT-focused; contains extensive realistic botnet-generated DDoS and other IoT attack traffic.	Extremely large (72 M records); covers limited types of DDoS.
CIC-DDoS2019	Targets modern volumetric DDoS techniques, including multiple reflection/amplification attacks.	Specialised to DDoS; lacks benign diversity; huge volume hinders preprocessing and ML training.

2.2 Machine Learning Research

2.2.1 Overview of Machine Learning for DDoS Detection

With the rapid development and creation of DDoS attacks, machine learning (ML) is widely used in detection systems to continuously monitor network traffic and distinguish between normal and malicious activities. By learning from the data, these systems adapt to evolving threats and improve overall cybersecurity resilience.

The characteristics of modern-day DDoS attacks are dynamic, mostly zero-day, and include multi-vector components.(11) Traditional detection systems rely on static thresholds, signature-based matching, and fixed heuristics, which often fall short at detecting complex modern DDoS attacks. To catch new attacks, the system needs manual updates—creating an urgent need for intelligent, flexible measures that can recognise and adapt in real time. (12)

ML emerges as a compelling solution because of its adaptive learning: it can incrementally retrain or fine-tune itself on fresh data, making it well suited to identify evolving attack patterns. Unlike static models, ML algorithms can tease out subtle, high-dimensional patterns via feature extraction, continuously updating their notion of “benign” vs. “attack” and thus minimising false positives.

Beyond accuracy, ML frameworks scale and automate easily: distributed training on GPUs or clusters handles large volumes of network flows, and once trained, a compact model can be deployed in line to make instant predictions on live traffic.

2.2.2 Types of Machine Learning

The main forms of ML fall into four categories, as shown in Figure 1. Each has distinct strengths and trade-offs:

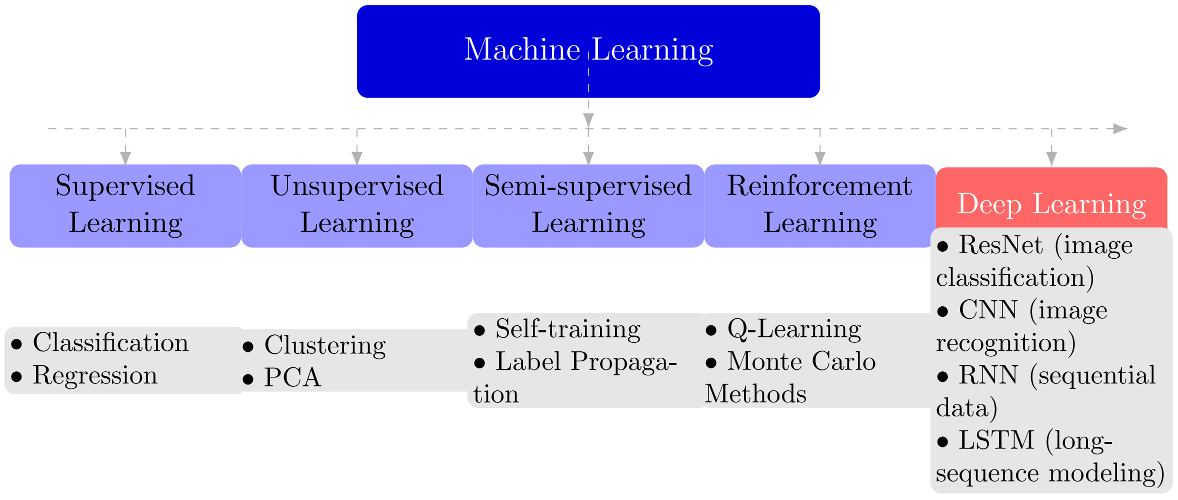

Figure 1: Types of Machine Learning and their common algorithms with an extension to Deep Learning, a subset of ML.

The main forms of ML can be categorised as presented in Figure 1. Each of these types comprises distinct strengths and weaknesses which are primarily determined by how they interact with data.

For supervised learning, output labels are part of the training data where models map the input features to these labels. The main advantages are high accuracy and the ability to fine-tune the model using well-defined classes. However, an update of the class labels is required to catch up with new attack patterns. Some of the commonly used models of this type comprise Support Vector Machines (SVMs), Decision Trees, Naïve Bayes, and Boosting algorithms.

Conversely, since unsupervised models do not depend on labelled data, unknown or new attack patterns can be discovered through anomaly detection. This feature reflects their ability to learn normal behaviour from raw data without prior labelling. Still, this comes with a trade-off of higher false positive rates and lower accuracy, as predictions are made without explicit guidance.

In hybrid and ensemble approaches, models in these fields have become more prevalent due to their combined accuracy and robustness. Hybrid systems fuse supervised and unsupervised methods to benefit from both labelled-driven precision and anomaly detection. Ensemble methods, such as Random Forest, combine multiple instances of the same or different model types (e.g. bagging, boosting, stacking) to produce diversified, majority-voted predictions.(13)

Moreover, Deep Learning (DL), a subfield of machine learning, is often chosen for its strength in handling complex or unstructured data. It utilises multi-layered neural networks to automatically extract hierarchical features from high-dimensional data. This approach extends traditional machine learning by reducing the need for manual feature engineering.

2.2.3 ML Pipeline or Workflow

During the implementation, an ML pipeline is often established to automate and streamline the end-to-end process of developing, deploying, and maintaining machine learning models. This setup offers simplicity, scalability, and efficiency in producing robust detection systems.

A. Data Acquisition and Preprocessing The literature establishes that effective machine learning for DDoS detection requires systematic data preprocessing. First, data cleaning is necessary to remove redundant flows, handle missing values, and ensure data quality, as noisy inputs can distort model training. Scaling and encoding follow, where numerical features are normalised to a standard range and categorical attributes are transformed for algorithm compatibility (8). Feature engineering is critical to highlight patterns such as burstiness or flow intensity that are characteristic of DDoS attacks (10). Imbalance handling addresses the common issue where benign traffic vastly outnumbers attack instances, typically through resampling techniques like SMOTE. Without these steps, models are prone to biased learning and poor generalisation. A typical preprocessing pipeline summarising these stages is illustrated in Figure 2, providing a structured approach to ensure data integrity and model reliability before training.

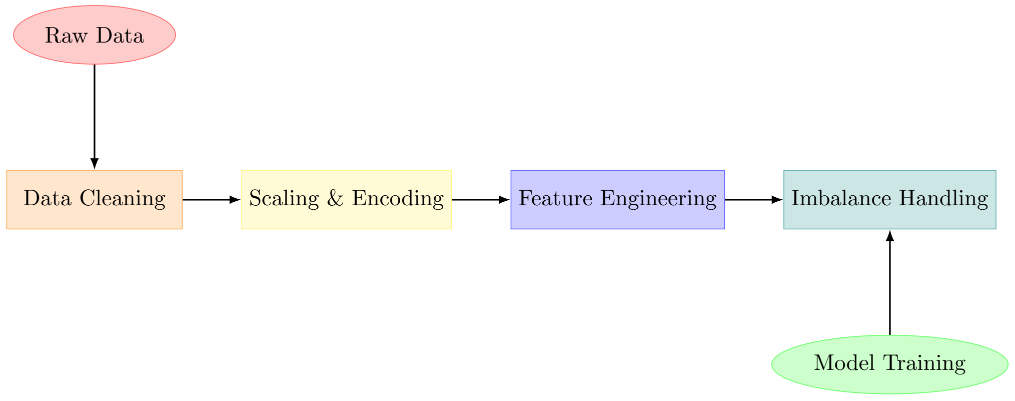

Figure 2: Flowchart illustrating the data preprocessing pipeline.

B. Model Training and Evaluation To train and evaluate the model, researchers normally split the whole dataset into training and testing subsets. A common type is the hold-out split; a normal configuration would be 70% for training and 30% for testing or an 80:20 ratio. They can be more robust by adding the validation set in the middle, i.e. train/validate/test; a normal configuration would be 70/10/20. The validation set is used as an intermediate evaluation step to help tune the hyperparameters, monitor the performance during training, and decide when to stop training (e.g. overfitting).

K-fold cross-validation (CV) techniques is also employed to ensure more robust assessment. K-fold CV is a technique used to assess the performance of all the train/test combinations used. It divides the dataset into k folds; it trains on the k-1 folds and tests on the remaining fold. Repeat the process k times for every combination and average the accuracy/performance metrics for all combination.

Each ML model has its own set of hyperparameters, which affect the performance directly. For example, the tree depth in Random Forest, learning rate in neural networks, etc.

Hyperparameters can be tuned using the grid search or random search on the training data. However, in ML-based DDoS detection research, many studies tend to select hyperparameters arbitrarily, often relying on initial default settings rather than thoroughly optimising them via systematic search methods (16).

C. Deployment Considerations During this stage, turning the trained models into products, the first aspect to achieve is model packaging and environment management for simplicity and integrity of the model's environment. This can be done by encapsulating each component such as code and dependencies of the model into a single container or unit. Then Continuous Integration and Continuous Deployment (CI/CD) is followed to automatically streamline the processes of the deployment ranging from building, testing, to launching the system – code and model (9). After the system is deployed, a monitoring and maintenance phase is introduced to assess the performance of the models as factors like concept drift could occur in a real-world scenario where data is evolving. Correspondingly, on the maintenance side, a retrained process is constructed to handle the model related issues (15). Lastly, to increase the robustness of the system, moving it on cloud or edge devices could enhance the scalability or low latency processing, respectively.

2.2.4 Evaluation Metrics

According to Sokolova and Lapalme's systematic analysis of performance measures for classification tasks, key evaluation metrics include the confusion matrix, accuracy, error rate, precision, recall, specificity, F-measure (F1-score) and ROC curve.

These metrics can be categorised into three classes: overall performance metrics; class-specific or detailed performance metrics; and threshold-independent metrics, based on their roles and purposes (17).

With respect to Table 2. the detailed explanation of this conversion and labelling process, Accuracy and error rate provide a high-level summary of the model's correct and incorrect predictions. In terms of class-specific metrics, precision evaluates the accuracy of positive predictions, recall measures the ability to capture all actual positives, specificity assesses the accuracy of negative predictions, and the F1-score balances precision and recall into a single measure.

Threshold-independent metrics, such as the ROC curve, ROC–AUC and confusion matrix, depict classification performance without depending on a specific threshold. While the former summarise performance by summarising the trade-off between true positive and false positive rates across all decision thresholds into a single scalar value, the latter returns a table counting true positives, false positives, true negatives and false negatives between predicted and actual labels.

Table 2: Key evaluation metrics and their formulae

Accuracy	$\frac{TP+TN}{TP+TN+FP+FN}$	**Recall**	$\frac{TP}{TP+FN}$
Error rate	$1-\text{Accuracy}$	**Specificity**	$\frac{TN}{TN+FP}$
Precision	$\frac{TP}{TP+FP}$	**$\mathbf{F_1}$-score**	$2\times\frac{\text{Precision}\times\text{Recall}}{\text{Precision}+\text{Recall}}$

Moreover, Fawcett compared these metrics to highlight each measure's strengths. The ROC-based metric is often preferred with balanced class distributions because it provides a general, accurate view of classification results. Conversely, on imbalanced datasets, metrics such as precision, recall, and F1 score better capture performance on minority classes. As a result, these performance-based metrics will be applied to evaluate the classification models for DDoS detection (18).

2.2.5 Challenges and Limitations

Model Complexity and Under/Overfitting When implementing the models, some considerations are posed to avoid problems during training and evaluating phases. One of the common issues is model under or overfitting, a balance between bias and variance is needed to avoid falling into one of the model complexity problems. To illustrate, while underfitting refers to an inability of the model to capture patterns both in training and testing data due to being too simple, overfitting happens when the model learns highly complex or specific details of training data including noise and is not generalize on unseen data (19).

Computational Constraints Another potential challenge is computational constraints, including model size, inference latency, and energy consumption. This set of constraints, happened more often in deep learning than in traditional machine learning models, is mainly attributed to the size of dataset, complex algorithms, and number of trainings or epochs. For instance, 1D and 2D CNNs required a generation of images to be used as a training input which required large storage and memory consumption (20).

Deployment Issues Additionally, during the deployment phase, constructing the ML pipeline can involve scalability and CI/CD related problems. These obstacles include handling large datasets with real time or rapid response and automated executions on each stage of the pipeline, respectively (9).

2.3 Simulation Team Research

2.3.1 Simulation on Virtual Machines (VMs)

One of the methods that was evaluated for the simulation of Distributed Denial-of-Service Attacks was through Virtual Machines (VMs). VMs have been used in industry and research due to their flexibility, isolation, and reproducibility.

VMs allow for safe simulation of malicious traffic without impacting production systems when used in cybersecurity experiments (21). VMs create controlled, repeatable scenarios which are very useful for research (22). Especially, for the use of DDoS simulations, VMs offer strong isolation between the host and guest systems (23).

VMs have been utilised to simulate botnets and flood attacks in the past (24). This makes VMs a reliable platform to simulate intrusion attacks such as DDoS. VMs also make it easy to reset the state of the environment through its rollback feature. VM clusters have also been used by researchers to study the propagation patterns of DDoS attacks and assess defence mechanisms.

However, there are also notable drawbacks to VMs. The resource overhead with VMs can become significant, especially for DDoS attacks as these can require hundreds to thousands of nodes (25). Each VM would also require its own operating instance, overall consuming a lot of memory and processing power. Furthermore, the network traffic generated by VMs may differ slightly from real-world DDoS scenarios (26).

Given these limitations, the use of VMs made it difficult to achieve scalability and efficiency, which were key requirements for the simulation of DDoS in this project. Therefore, lighter alternatives such as containerisation through Docker were evaluated to achieve these requirements.

2.3.2 Simulation on Docker

Docker was found to be an excellent tool to be able to simulate Distributed Denial-of-Service (DDoS) attacks. Docker was found to be easy to set up and enabled cross-platform deployment. The Docker setup can be easily distributed among team members with different Operating Systems without any configuration setup. This is essential for efficient integration with the Machine Learning (ML) Pipeline.

Compared to Virtual Machines (VMs), the containers are lightweight and start up quickly (27). Thus, multiple attacking instances can be executed on a single machine. Docker also provides process-level isolation for each container, including its encapsulated applications and dependencies. Importantly, containers make these simulations reproducible because the container images define and fix the execution environment (28). The Docker setup also allowed for realistic traffic generation. Each container can produce realistic packet flows that closely mimic real-world traffic. Docker is open-source and free to use. There are no additional costs associated with creating multiple containers. Under budget constraints for this project, Docker appeared to be a cost-effective option.

Overall, Docker offers an optimal balance across all elements required in this project. Using Docker for the simulation means being able to simulate a diverse range of attacks, gather data, automate the entire process, and easily integrate the Simulation Pipeline with the ML Pipeline.

The simulation system is split into different containers, where one is the victim server, and another container is the attack type. In this system, the server logs the incoming traffic to be used as data. Managing containers using Docker helps track attacker sources in the traffic data.

A range of different attack types were selected for the simulation environment guided by the attacks found in the data that the ML detection system used. In addition, a benign traffic pattern was included to mimic regular client behaviour. The benign traffic was designed to generate periodic requests and wait for responses, like how regular clients act.

Two selected tools for these attacks are: SlowHTTPTest for the simulation of application layer Denial of Service (29) and Low Orbit Ion Cannon (LOIC) of its HTTP flood variant using packet-based generation. Both attacks use curl, which is an open-sourced tool for transferring data (30).

In addition, two Python libraries were researched to simulate two other attacks. Goldeneye (31) is an HTTP DoS Test Tool that uses the 'HTTP Keep Alive + NoCache' as an attack vector (32). Slowloris (33) is a DoS attack that attempts to attack a targeted server by simultaneously opening and maintaining multiple HTTP connections, overwhelming the victim server with the active connections (34).

2.3.3 Automation using Ansible

Ansible is an automation tool that is well-suited for orchestrating complex workflows and excels at orchestrating multi-step Docker workflows (35). Ansible is easy to use as it uses YAML syntax, which is readable and easy to understand for users that do not have DevOps expertise. Since Ansible is declarative, it allows the actions to be described without having the handle the low-level details. There is no additional infrastructure setup required as Ansible is agentless.

Ansible helps to ensure reproducibility of the simulation environment. Using Ansible for the automation for DDoS simulations guarantees that the simulation is deterministically the same way every time. Ansible utilises idempotency resulting in each execution being consistent and the environment being setup uniformly every time.

In contrast to other automation tools, Ansible has built-in error checking and halts the playbook and reports the error when an error is detected, whereas bash scripting will continue to execute even after a command fails (35). Shell scripts are also imperative making in difficult to scale and lack idempotency, consequently re-creating resources that already exist. Ansible module, however, can handle this logic and will only create resources that are required. Ansible also provides better orchestration capability than Docker compose. For a complex simulation, which requires to sequentially run containers for a specific amount of time with multiple iterations, cannot be handled by Docker compose itself. However, Ansible can coordinate Docker containers alongside other commands and file operations.

Ansible is a powerful and convenient automation tool for the simulation of DDoS. Ansible provides deterministic outcomes, improving the reproducibility of the simulation. Overall, Ansible offers the greatest blend of flexibility and control along with automatic error-handling, making it the most suitable automation tool for the simulation environment.

Ansible playbooks can seamlessly integrate with Docker and the local file system. Ansible playbooks can build images from Dockerfiles and run containers. Ansible aids in orchestration the overall sequence of the running containers using tasks (36). Each task in a playbook has a descriptive name, which details the action being performed by that task. The use of tasks smoothens the process for adding attacks and adjusting the workflow, since this only requires only editing or adding another task into the playbook. Ansible also supports task tagging, which allows to run a subset of tasks within a playbook.

Therefore, playbooks in Ansible make it seamless to automate and control Docker containers. It helps to reduce duplication of specific actions and is easy to understand. Overall, Ansible is a suitable tool to automate the environment for the Distributed Denial-of-Service (DDoS) simulation.

3 Overall System Architecture and Integration

3.1 ML Pipeline

Dataset Components: The datasets (i.e. `2018.csv`, `2019.csv`) are initially loaded as the primary data sources containing the processed DoS and DDoS attacks from the CSE-CIC-IDS2018 and CIC-DDoS2019 datasets. The two datasets are used as the foundation for both model training and evaluation.

Input to ML Component: Each dataset is fed to the ML pipeline, which contains the code implementation of various ML models for training, validation, and testing. Every combination of the datasets and the models is run by the Papermill library that orchestrated the whole system.

Outputs from the ML Component: The detailed evaluation reports (e.g. classification reports, confusion matrices, precision, recall, F1-score) and the trained model files along with associated scaler files (such as '.pkl', '.pth', '.scaler' files) are outputted by the ML components.

Testing and Prediction Component (two available paths):

Path 1 This is another test apart from train, validate, and test in "Input to ML Component". The test dataset, which follows the same attack patterns as those in 2018.csv, is simulated by the simulation team. Each trained model, along with its corresponding scaler, is loaded to the testing component with the test dataset. A set of evaluation matrices is then generated to indicate the model's performance on the unseen test set, and these results are used to determine if further data augmentation or model retraining is required.

Path 2 Alternatively, a separate prediction script can be used for making real-time predictions. A single datapoint (or a stream of data), formatted with the exact same set of features, is taken by this component along with the corresponding trained model and scaler, and a prediction (e.g. Benign vs. DDoS types) is generated.

Model Evaluation, Decision Making, and Data Augmentation: The output matrices (e.g. average accuracy, F1-score) of each model are compared against a predetermined threshold. For example, an average accuracy score above 0.8 is considered acceptable; if not, it is concluded that the dataset requires further enrichment. The enrichment process is initiated by the simulation team as they generate additional attack data to cover a more diverse instance of DDoS attacks. The newly generated attack data is then concatenated with the existing 2018.csv dataset to form a new enriched version.
Retraining the Models: This is the start of a new training cycle. The updated dataset is then fed back to the ML pipeline. The entire process of training, validating, and testing is repeated with this new dataset. The iterative steps are repeated until the models' accuracy exceeds the threshold.

3.2 Simulation and Data Team Coordination

The simulated system includes an automated pipeline for collecting, labelling and converting network traffic data into a format that the ML pipeline can use. The data collection system is stored in two files called `attack_timeline.csv` and `capture.pcap`. Then, both are converted into the usable `capture.csv` in the required format.

At the start of simulating, the `attack_timeline.csv` and `capture.pcap` files are recreated. The entries in the `attack_timeline.csv` file store the current time and label associated with the logged event of each attack. The server begins with a shared directory accessible outside the container in which all generated files are stored. When the victim container is initiated, `tcpdump` is used to capture all connections to the server in the new `capture.pcap` file.

Once the attacks have finished and the containers have been stopped, a new container is run to convert the `capture.pcap` and `attack_timeline.csv` into `capture.csv`. This conversion process involves parsing raw packet-level data, structuring it into flows, and associating each flow with a label based on the attack timeline. The detailed explanation of this conversion process is provided in Section 4.4.

This pipeline produces, from a simulated environment, a fully labelled and consistent dataset. The system is designed to be robust and adaptable, regardless of the attack type, duration, or number of concurrent attackers. Its modular architecture ensures that changes to the victim server, attack scripts, or network configurations do not compromise the structure or reliability of the resulting dataset. This makes the pipeline suitable for continuous experimentation and scalable data generation in adversarial traffic scenarios.

4 Dataset Acquisition and Preprocessing

4.1 Dataset Selection

4.1.1 Selection Criteria

The selection of the dataset was guided by five main criteria identified in the literature review:

- Realism and Recency: Inclusion of recent and realistic DDoS traffic to reflect modern attack methods and network behaviours.

- Attack Diversity: Coverage of volumetric, protocol-based, and application-layer attacks to support multi-class detection.
- Reliable Labelling: Well-labeled flows enabling supervised learning and accurate evaluation.
- Temporal Integrity: Presence of timestamps or flow sequencing to support time-based feature engineering.
- Computational Feasibility: Dataset size must be sufficient to train deep models, while remaining manageable given available processing resources.

4.1.2 Choice and Justification of Datasets

To comprehensively address the challenges of multi-vector DDoS detection, two datasets were selected: CSE-CIC-IDS2018 and CIC-DDoS2019. The CSE-CIC-IDS2018 dataset provides DDoS-specific traffic captured on February 15, 16, 20, and 21, 2018, featuring attacks such as Slowloris, SlowHTTPTest, Hulk, LOIC, and HOIC embedded within realistic benign flows. This dataset offers rich application-layer attack diversity while maintaining manageable data size for data exploration.

To complement this, CIC-DDoS2019 was incorporated, targeting modern reflection and amplification-based DDoS attacks (e.g., DNS, NTP, LDAP, MSSQL, SSDP, SNMP, PortMap, UDP-Lag, SYN floods). It reuses the CICFlowMeter-V3 feature set, providing over 50 million labeled flows and large PCAP collections (37). This addition was motivated by three factors: increased scale for training robust models, expanded attack breadth aligned with real-world trends, and a homogeneous feature schema facilitating consistent preprocessing.

Compared to datasets like CIC-IDS2017, UNSW-NB15, and BoT-IoT, this combination offers greater recency, attack diversity, and feature compatibility, forming a strong foundation for developing and evaluating adaptable DDoS detection systems.

4.2 Data Processing

4.2.1 Cleaning and Normalization

Each daily CSV was processed in isolation to avoid temporal leakage, and only after all four days had been explored were their feature tables concatenated. During each day's exploration, IP-address columns were immediately dropped to force the model to learn behaviour-based patterns rather than overfitting to static addresses, which lack generalisability across new network environments. Next, all infinite values were converted to NaN and any rows containing NaN were removed; this strict imputation policy was adopted because even a few extreme or missing values can skew the moments (mean, variance) used in subsequent feature construction.

Once the four per-day tables had been merged, each numeric feature was standardised (zero mean, unit variance). This choice was informed by empirical studies showing that gradient-based classifiers (e.g. logistic regression, neural networks) converge more stably under standardization than under min–max scaling when feature distributions exhibit heavy tails, which is the case for packet counts (10^5–10^6) versus rate features (10^{-3}–10^0) in CIC-style flow data.(38) By doing standardization on the merged set rather than per day, consistency of scale across days was guaranteed, preventing day-to-day shifts from degrading model performance.

4.2.2 Handling Imbalanced Data

Severe skew toward benign flows was observed in each day's labels, which may cause any error-minimizing classifier to collapse to "always-benign" predictions. To mitigate this, a two-stage sampling protocol was executed on each day's dataset:

1. Benign samples were under-sampled to achieve a 1:1 ratio relative to the combined attack classes, thereby preventing the majority class from dominating gradient updates.
2. Any attack subtype whose representation fell below 30% of the total was synthetically augmented via SMOTE, with the oversampling factor capped at 1.2 times the original minority size; this interpolation among genuine minority instances preserved the intrinsic manifold of attack patterns while introducing sufficient diversity to avoid underfitting on subtle behaviours.

Only after these per-day balancing operations were the daily sets merged, ensuring equal contribution from each day and preventing day-specific sampling biases from compounding. Importantly, SMOTE was applied exclusively to the training data; in the closed-loop detection system no synthetic samples are ever generated nor are any real observed flows removed, thereby ensuring the integrity of the real-time data.(39)

4.3 Feature Engineering

This feature engineering step was designed to capture temporal patterns and distribution characteristics of network traffic. It should be noted that the original feature columns between the CSE-CIC-IDS2018 and CSE-CIC-IDS2019 datasets are almost identical, which made it convenient to apply consistent feature engineering strategies across both datasets.

4.3.1 Forming New Features

To better characterize the behaviour of network flows, several new features were derived, aiming to capture flow intensity, throughput behaviour, and packet size variability do not present in the original dataset. First, Flow Packets/s and Flow Bytes/s were derived to measure the throughput intensity of each flow. As shown in Figure 3, these were calculated by dividing the total number of packets and bytes (forward and backward combined) by the flow duration in seconds:

```python
# (TotLen Fwd + TotLen Bwd) / (Flow Duration in sec)
df["Flow Bytes/s"] = np.where(sec > 0, (df["TotLen Fwd Pkts"] + df["TotLen Bwd Pkts"
    ]) / sec, 0)

# Flow Packets/s: (Tot Fwd Pkts + Tot Bwd Pkts) / (Flow Duration in sec)
df["Flow Packets/s"] = np.where(sec > 0, (df["Tot Fwd Pkts"] + df["Tot Bwd Pkts"]) /
     sec, 0)

```

Figure 3: Calculation of Flow Packets/s and Flow Bytes/s to measure throughput intensity per flow.

Second, rolling statistics on flow durations across time were computed to detect short-term variability in connection patterns. Specifically, a 10-flow moving average and standard deviation of flow durations were calculated after sorting flows by timestamp, as shown in Figure 4.

```python
# rolling mean and std across flows (by timestamp)
df.sort_values("timestamp", inplace=True, ignore_index=True)
df["Flow_Duration_rolling_mean"] = df["Flow Duration"].rolling(window=10,
    min_periods=1).mean().fillna(0)
df["Flow_Duration_rolling_std"]  = df["Flow Duration"].rolling(window=10,
    min_periods=1).std().fillna(0)

```

Figure 4: Code for calculation of Rolling Mean and Standard Deviation of Flow Durations to Capture Short-Term Variability.

Third, packet-length entropy was computed by binning the mean forward packet lengths into 20 equal-width buckets and applying Shannon entropy. This feature captures the diversity of packet sizes within a flow, helping to distinguish templated attack flows from more irregular benign traffic, as shown in Figure 5.

```python
# entropy from distribution of Fwd Pkt Len Mean (20 bins)
bins_arr = np.floor(df["Fwd Pkt Len Mean"] / 10)
if np.all(bins_arr == 0):
    ent_val = 0
else:
    hist_counts, _ = np.histogram(bins_arr, bins=20)
    ent_val = entropy(hist_counts) if hist_counts.sum() > 0 else 0
df["entropy_pkt_len"] = ent_val

```

Figure 5: Code for Calculation of Packet Length Entropy to Capture Diversity in Flow Packet Sizes.

All these newly engineered features are lightweight, interpretable, and designed to be computed efficiently.

4.3.2 Selecting Key Features

After additional features were generated, the most relevant subset was selected based on domain knowledge and feature importance analysis.

Although Principal Component Analysis (PCA) was initially considered, it was not adopted for feature reduction. PCA transforms original features into abstract combinations, complicating interpretation in real-time intrusion detection systems. Therefore, preserving the original semantics of features was prioritized. Instead, SHAP (SHapley Additive exPlanations) values derived from a Random Forest model were utilized. SHAP provides consistent, model-agnostic attributions and correctly handles correlated inputs, offering a more reliable feature ranking than simple Gini impurity scores.

Building on prior analyses, features representing volumetric, temporal, and behavioural aspects of each flow were emphasized:

- **Flow Duration:** Capturing the total time span of each connection.
- **Packet Counts/Bytes (forward and backward):** Reflecting the overall traffic volume of each flow.
- **Packet Length Distributions (mean, max, standard deviation):** Characterizing the variability and size of transmitted packets.

- **Inter-Arrival Times (IAT statistics):** Measuring timing patterns between successive packets.
- **Flags:** Indicators for TCP-level events such as SYN, ACK, FIN, which aid in protocol behavior analysis.

The final selected features include:

- **Temporal alignment:** Timestamp
- **Protocol and service identifiers:** Destination Port, Protocol
- **Duration metrics:** Flow Duration, Flow Duration Rolling Mean, Flow Duration Rolling Standard Deviation
- **Volume metrics:** Total Forward Packets, Total Backward Packets, Total Length of Forward Packets, Total Length of Backward Packets
- **Packet size statistics:** Forward Packet Length Maximum, Forward Packet Length Minimum, Forward Packet Length Mean, Forward Packet Length Standard Deviation
- **Throughput measures:** Flow Bytes per Second, Flow Packets per Second, Bytes per Duration
- **Directional imbalance:** Packets Ratio
- **Inter-arrival time metrics:** Forward Inter-Arrival Time Total/Mean/Max/Min, Backward Inter-Arrival Time Total/Mean/Max/Min
- **Flag counts:** SYN Flag Count
- **Entropy metrics:** Entropy of Packet Length
- **Subflow volume:** Subflow Forward Bytes

By retaining only about 30% of the original features, redundancy was eliminated, and computational efficiency was significantly improved. Furthermore, by combining this carefully engineered feature set with advanced machine learning models, substantial time and effort can be saved while maintaining high detection performance and model interpretability.

4.4 Interface between Simulation and ML Models

To form a closed-loop autonomous system, an automated data processing interface must be implemented between the simulation and model training pipeline. The goal of this interface is to convert the raw traffic data generated by the simulation stored in PCAP format into structured CSV files interpretable by machine learning models.

4.4.1 Tool Selection

To extract relevant fields from PCAP files and convert them into tabular format, `tshark`, the command-line utility of Wireshark, was selected. Alternative libraries such as PyShark, Scapy, and `dpkt` were initially considered. However, they exhibited critical limitations in terms of speed, memory efficiency, and support for complex network protocols. These factors are especially significant when handling large-scale datasets generated from high-volume DDoS simulations. `Tshark` is implemented in C/C++, which allows it to process millions of packets efficiently without excessive memory overhead. This performance advantage has also been recognized in prior work focused on high-throughput packet analysis pipelines. (40)

Furthermore, using `tshark` via Python's `subprocess` module allows for precise field selection, reproducible CSV output, and easier debugging compared to black-box wrappers like PyShark. The separation of decoding and processing also makes the workflow more modular and practical.

4.4.2 Converting and Processing

The automated data processing pipeline follows a structured sequence from raw packet capture to feature-enriched, labeled CSV output. This workflow is illustrated in Figure 6.

The process begins with converting the PCAP file into a tabular format using `tshark` in `-T fields` mode, extracting only the relevant fields—such as timestamps, IP addresses, ports, protocols, and frame lengths—into a structured CSV file. This selective extraction improves memory efficiency and processing speed while ensuring consistent column alignment, even when certain fields are absent in specific packet types.

Next, the parsed packet data is grouped into bidirectional flows based on the combination of source and destination IP addresses, source and destination ports, and the associated protocol stack information extracted from the packet headers. To standardize flow direction, a canonicalization step is applied to ensure consistency between forward and backward flows.

Once flows are established, basic statistical and temporal features are computed to characterize each flow's behavior. These include packet counts, byte volumes, flow durations, and timing metrics. In cases where certain features cannot be calculated due to insufficient data (e.g., no backward packets), zero values are inserted to maintain a consistent schema.

The resulting structured flows are exported to a CSV file, followed by a data processing and feature engineering pipeline that ensures the final dataset provided to machine learning models remains structured and consistent (41).

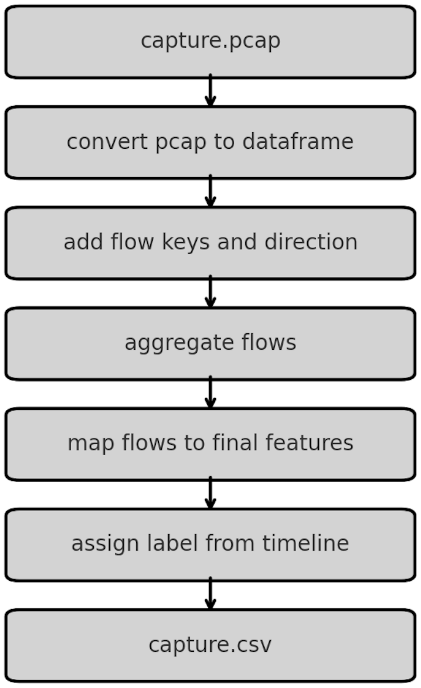
capture.pcap
convert pcap to dataframe
add flow keys and direction
aggregate flows
map flows to final features
assign label from timeline
capture.csv

Figure 6: Functional Pipeline of the Pcap-To-Csv Conversion Script

5 Machine Learning Development

5.1 Model Research and Selection

Complementing the literature review of ML research in Section 2.2, a more in-depth exploration has been conducted to explain how each model is selected and implemented. Supervised based model, including an ensemble method, is the most common choice for DDoS detection due to its interpretability and robustness. Some prevalent names are Random Forest, XGBoost, Logistic Regression, and SVM. However, other types of ML, namely unsupervised and deep learning, are observable regarding their strengths in discovering new attacks and ability to learn complex temporal and spatial features from network traffic data, respectively (42).

To select the stated models, a set of criteria is applied to obtain the models which effectively meet the deliverables of the project. Based on numerous papers, detection accuracy, training-inferencing speed, scalability, and interpretability, applications are the main aspects used for selection process (43). For accuracy, it is often incorporated into classification report metrics, available via Scikit Learn in Python, which shows a more sophisticated evaluation on the model performance. Also, a level of interpretability of model's algorithm is essential for developers to identify problems when model performance is affected by the decision making of the model (44). In terms of the use case, models which are strongly applicable to classification tasks will be prioritized due to the objectives of the project which aims to distinguish between benign and numerous types of DDoS attacks.

Based on the selection criteria, the following models in the paragraphs below were researched for the development. A literature review on each of the selected model had been conducted to provide the motivation and rationale of the selection result.

Decision Trees (DT) are simple supervised tree-based classifier that classifies data based on their features. The outcomes of the test (e.g. T if packet size is greater than 100 MB) are represented by each branch and the final class labels are represented by the leaf node (e.g. Benign vs Attack). The DT's performance on the DDoS datasets demonstrates high accuracy performance as shown on the 2019 dataset which achieved approx. 99%. However, individual DTs may exhibit tendencies toward overfitting and are less robust when compared to ensemble method. On the other hand, this simplicity resulted in a lower computational cost; one study observed the DT performed better (in computational complexity) than the Random Forest model when both models achieving the same accuracy (45).

Random Forest (RF), shown in Figure 7, is an ensemble method built from multiple decision trees which are trained on random subsets of features and training data. Patterns of its sub-samples are modelled by the individual tree and the final results are aggregated from all the individual trees. RFs are known for "precision, robustness, and ability to handle complex data interactions," thus reducing overfitting by pooling multiple trees (46). From a survey that used the 2018 dataset benchmark, RF is among the top performers (97–99%) in achieving the highest precision and accuracy score (47).

Support Vector Machines (SVM) are defined as supervised machine learning algorithms that classify data by finding an optimal hyperplane to separate different classes in high dimensional space. During training, a decision boundary is determined such that the distance between the hyperplane and the nearest data points (support vectors) from each class is as large as possible. Strong accuracy has been shown in small intrusion benchmarks, but a lag in performance is observed in large performance like the 2018 dataset (49).

The probability of a binary outcome can be predicted via **logistic regression (LR)** by modelling the relationship between independent variables and the log-odds of the dependent variable. LR is characterised by the simplicity, interpretability, and efficiency on linearly separable data. A linear decision is assumed in LR, which is considered a weakness as accuracy

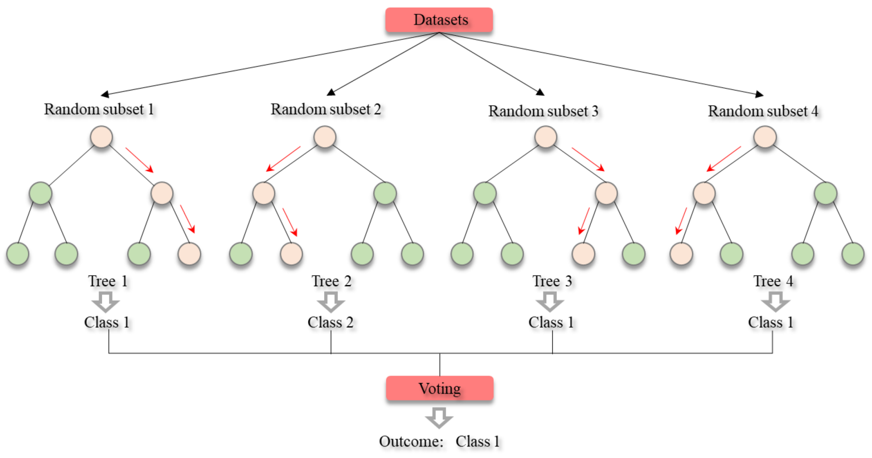

Figure 7: Random Forest model structure algorithm where multiple decision trees are trained on random subsets of data and combined through majority voting. (48).

may be limited if the attacks are found to not be linearly separable (50).

K-Means is selected to present the unsupervised clustering algorithm for anomaly detection. K distinct, non-overlapping dataset subsets are represented by "clusters" each characterised by its centroid. The within-cluster variance is minimised by the algorithm: each datapoint should be as close as possible to the cluster's centroid. The optimal configuration is achieved iteratively by alternating between assigning datapoints to the nearest cluster and recalculating the centroids distance until convergence is satisfied. The optimal number of clusters (k) is determined by evaluation metrics such as the Within-Cluster Sum of Squares (WCSS), Silhouette Score, and Calinski–Harabasz Index (51).

An extension to normal K-Means to represent the semi-supervised nature is represented by the **Seeded K-Means**. The dataset is similarly partitioned into k distinct clusters defined by the number of unique class labels. However, in Seeded K-Means, the initial centroids are intentionally determined using labelled samples (seeds) rather than random initialisation. The clustering process is guided by these seeds toward more label-aligned grouping (52).

Convolutional Neural Networks (CNNs) are deep learning models designed to automatically extract spatial features from input data. Alongside this, a focus on 1D and 2D CNNs has been made to explore their functionality. To illustrate, the former approach is often applied to sequential data such as network traffic flows to capture temporal dependencies. The latter one involves dealing with spatial patterns where input features are in the form of a matrix or image. Based on Al-Dulaimi and Aksu papers, both types of CNN exhibited a strong performance with CICIDS2017 dataset at a rate of 98.96% and 99.99% in accuracy for 1D and 2D. Moreover, their advantages and limitations have been highlighted in the areas of performance and resource efficiency. While both models effectively detect complex patterns and high-dimensional relationships in network behaviour, low interpretability and high computational cost for training and testing are the key concerns [(53), (54)].

Long Short-Term Memory (LSTM) networks, a specific type of recurrent neural network (RNN), manage long-term dependencies in sequential data via specialized gating mechanisms, mitigating the vanishing gradient problem. LSTMs have achieved some of the best results on intrusion datasets; LSTMs reached 97% accuracy on the 2019 dataset, the highest within the deep learning group (45). LSTMs are "particularly adept at capturing temporal patterns and sequential relationships" (55).

Another group of ML method is **hybrid and ensemble models**, where multiple models are combined to make a more powerful and unbiased decision. In a hybrid model, an integration between unsupervised and supervised ML models is often established to increase generalization and robustness. One common example is combining K-Means for preprocessing or anomaly detection with SVM for final classification (56). For ensemble models, three major types include boosting, bagging, and stacking, where each form aims to maximise performance by reducing variance, bias, or both. According to Sultana et al. (2019), ensemble methods achieved over 99% accuracy with low false positive rates on CICIDS2017 (57).

5.2 Model Implementation – Classification Based

All models are built and developed locally through Visual Studio Code IDE and Python programming language. The main libraries utilised in this project reflect the selected ML and DL models, namely scikit-learn (58), PyTorch (59), and TensorFlow (60). Moreover, other tasks in implementation, including data preprocessing, visualisation, and model handling, are handled by pandas (61), NumPy (62), Matplotlib (63), Seaborn (64), Joblib (65), Optuna (66), along with their submodules. GitHub is integrated for version control, file management, CI/CD integration, and backup.

A general data preprocessing framework was applied to the datasets to ensure they are clean and suitably formatted. The steps are as follows:

- **Data Cleaning:** Irrelevant features such as low-level identifiers were removed. Missing and infinite values were also removed. Any records with undefined feature values were discarded to avoid errors in model training.

- **Label Encoding:** After data cleaning, label encoding was performed on the target variable. Categorical labels such as "Benign" and "DDoS–Goldeneye" were converted to numeric codes. Input features were already numeric, so no encoding was needed for them.

- **Feature Scaling:** Features were normalised by subtracting the mean and dividing by the standard deviation, producing distributions with zero mean and unit variance. This prevents features on different scales from dominating training.

- **Train–Test Split + Window Sampling:** The dataset was partitioned into training (80%) and testing (20%) sets, with an optional validation split for hyperparameter tuning (e.g. 70/10/20). K-fold cross-validation was employed for compatible models (e.g. logistic regression, K-Means).

Alongside this, standard evaluation metrics—classification report and confusion matrix—were used to assess model performance.

The sub sections below will be presented with 4 key areas of each model implementation namely model architecture, training-validation-testing process, optimization techniques, and evaluation metrics distinct to each model. Note that the general pre-processing and some evaluation metrics steps have been mentioned to avoid repetition among the models.

5.2.1 K-Means and K-Means Seeded

Similar structure is used in the K-Means and K-Means Seeded models. Initially, Principal Component Analysis (PCA) was applied to standardised dataset to reduce dimensionality, minimize noise, and expedite clustering computations. A cumulative explained variance analysis was computed from components one to fifteen to determine the number of PCA component. Nine principal components are chosen as they accounted for 88.8% of the total dataset variance.

Following dimensionality reduction, the transformed dataset is partitioned into training and testing subset with an 80% and 20% ratio. Although K-Means clustering normally employs a training with the entire dataset due to its unsupervised nature, a train/test split was nonetheless performed to facilitate evaluation method since the dataset come with labels.

The optimal number of clusters (k) is determined by using three evaluation scores, namely Within-Cluster Sum of Square (WCSS), Silhouette Score, Calinski-Harabasz Index. For each k in the range of one to fifteen, K-Means clustering is computed and then plotted to determine where the convergence occurred on these three metrics. Based on these plots, the decision is made to select ten clusters as the optimal number. All three metrics are converged after increasing more than ten clusters.

For the corresponding evaluation curves and a visualisation of how each metric converges as k increases, see Section 8.2.4 and Figure 17 below.

K-Means With the optimal number of clusters (k=10) determined, the K-Means algorithm is retrained with the TRAIN dataset. The training process utilised the k-means++ initialisation strategy and set a random state of 42 to ensure reproducibility.

Post-training, a mapping of each identified cluster to its most frequent true labels from the train dataset is established. In each cluster, the dominant class is identified thus: enable the classification on future new unlabelled instances.

From the trained model, predictions are generated on the testing subset. Each test instance is assigned to a cluster based on the trained model, then cross-checked with the cluster-to-label mapping based on the previous step to come up with a prediction. Additionally, the composition of each cluster is evaluated for both the training and testing sets to assess the distribution and purity of each cluster.

Finally, the evaluation of the model is performed on the testing dataset using supervised metrics, including accuracy, precision, F1-Score, confusion matrix, and AUC-ROC. All generated evaluations and trained model-related artefacts, such as the model, PCA transformations parameters, and scaler objects, are saved.

K-Means Seeded The clusters in the K-Means Seeded are guided by the labelled samples; thus, the number of clusters is set to the number of unique class labels.

All structure steps and parameters applied in standard K-Means implementation are also followed in K-Means Seeded with modification to incorporate a semi-supervised nature through the seeded initialisation of cluster centroids. The cluster centroids are randomly initialised by using 250 samples per class from the training dataset. This sample size is decided based on the total volume of data and is considered sufficient to provide representative initial centroids.

The model is trained with these initialised seeded centroids as the initial centre clusters. Cluster-to-label mapping, predictions on the testing set, and evaluation of cluster composition are performed identically to the procedures described for unseeded K-Means. Lastly, any outputted artifacts are saved.

5.2.2 Logistic Regression (LR)

Hidden layers are not included in LR. An output neurone with softmax (multi-class classification) activation function is connected directly to the input layer and receives the pre-processed, selected input features.

The regularisation strength (L2 regularisation) is used as the main hyperparameter to prevent overfitting. Model performance was assessed using 5-fold cross-validation (via `cross_val_predict`) to generate the confusion matrix and classification report, while the L2 regularisation strength remained fixed. A more robust approach is created compared to a single train-test split.

5.2.3 Support Vector Machine (SVM)

The architecture of SVM can be viewed as a 3-layer system namely input, hidden, and output layers. The input layer of size n, containing input signal vector, sends a set of n vectors to the hidden layer where a kernel function is executed on them and n support vectors. The result of the kernel will then be delivered to a neuron in the output layer from summing all hidden-layer neurons. The calculated value will determine a specific class label that datapoint belongs to.

In this project, an algorithm of SVM, `SVC()` of `sklearn.svm`, is applied from Scikit-learn, a Python library for implementing ML models. Regarding the model architecture, a set of parameters, a kernel function, regularization parameter (C), and kernel coefficient (γ), is initialized to be used for training. For an ideal case, linearly separable data without transformations, a linear kernel and $C = 1$ are chosen for initial training before being finalized by a cross-validation of grid search. In a scenario where the kernel function is not linear, either `poly` or `rbf`, two additional variables will be added namely `degree` and γ to define polynomial shape and shape of Gaussian curves, respectively.

During model training, validating and testing, functions from `sklearn.svm` were applied to the split dataset. While `model.fit()` is used in the training phase, the two other phases are performed by `model.predict()` through a distinct group of datasets.

Prior to model testing, the hyperparameter tuning is conditionally executed based on the accuracy threshold from a validation result. For each variable, a range of value is defined to set up different combinations of the hyperparameter. For instance, three types of kernel function are presented, including `linear`, `poly`, and `rbf`. Similarly, C value is set from 0.1 to 100 in an interval of a multiple of 10. For small magnitude of C, it allows wider margin or misclassification but could lead to underfitting. Conversely, hard margin or when C is large, a misclassification is more sensitive and prone to overfitting. To select the best hyperparameter, `GridSearchCV()` from `sklearn.model_selection` is used to do a 3-fold cross validation on the candidates and pick the one with the highest accuracy. After the finding, if the best hypermeters are not the ones from the training, it will be updated to the model and retrained before testing.

In the evaluation stage, predicted classes with a set of hyperplanes after a PCA of the features space is constructable in SVM. PCA is performed to reduce the dimension of input features to 2D for simplicity of visualization. Also, since this is not a binary classification, multiple hyperplanes are established for each pair of classes via One vs One approach.

5.2.4 Random Forest Classifier (RFC)

Random Forest Classifier (RFC) is a bagging ensemble model comprising of multiple decision trees which represent the classifiers of the system. In bagging of RFC, each tree is responsible for a bootstrapped subset of the dataset where a random number of features are applied. The results are then evaluated based on the majority votes attributed from the votes of each tree.

The algorithm of the model, `RandomForestClassifier()`, is sourced from Scikit-learn or `sklearn.ensemble` to be specific. The parameters presented on this development consist of `n_estimators` or number of trees, `max_depth` or limit the depth of each tree, `min_samples_split` or minimum samples for a split, `min_samples_leaf` or minimum samples at a leaf, `max_features` or $\sqrt{n\_features}$ features for splits, `bootstrap` for bootstrap sampling, and `class_weight` to handle class imbalance. Moreover, most values of these parameters are set based on their default numbers provided in Scikit-learn document, such as `n_estimators=100` and `max_depth=None`.

The train-validate-test and hyperparameter tuning phases of random forest are similar to other Scikit-learn based models, including `model.fit()`, `model.predict()`, and `GridSearchCV()`. Precedingly, due to its tree-based algorithm, feature importance is possible to be calculated. It is used to depict the predictive power of each input feature on the target variable based on a level of decrease in Gini impurity. This metric does not only enhance feature selection, but also interpretability via model transparency.

5.2.5 1D Convolutional neural networks (1D CNNs)

The implemented 1D CNNs is constructed using PyTorch, `torch.nn.Conv1d`, designed for sequential feature extraction and multi-class classification. The model architecture comprises of three convolutional blocks, each containing a 1D convolutional layer, batch normalization, Leaky ReLU activation, and max pooling. These layers are then connected to two fully connected layers with Dropout regularization, Xavier initialization, and Adam optimizer.

Initially, an input data, typically a tensor of shape (batch size, channels=1, number of features), is loaded into the model. Then, it is applied to the convolutional layers with the specification of these layers shown in Figure 8. A feature map—patterns detected in the input data—is outputted from each convolutional block after being applied on a kernel where a dot product is performed. This output then becomes the input to the next layer. In each convolutional layer, a batch normalization and Leaky ReLU are executed to lower the internal covariate shift from unstable input distribution during training and avoid neurons outputting zero (dying ReLU), respectively. This is followed by a max pooling operation on the feature map to reduce the spatial dimensions.

```
self.conv1 = nn.Conv1d(in_channels=1, out_channels=32, kernel_size=5,
                       stride=1, padding=2)
self.bn1   = nn.BatchNorm1d(32)
self.conv2 = nn.Conv1d(32, 64, kernel_size=3,
                       stride=1, padding=1)
self.bn2   = nn.BatchNorm1d(64)
self.conv3 = nn.Conv1d(64, 128, kernel_size=3,
                       stride=1, padding=1)
self.bn3   = nn.BatchNorm1d(128)

```

Figure 8: Implementation details of the 1D-CNN model architecture in PyTorch, highlighting the definition of convolutional layers including I/O channels, kernel size, stride, and padding.

After the convolutional blocks, the feature map is flattened and passed through two fully connected layers, `fc1` and `fc2`, with dropout regularization (`dropout=0.3`) to prevent overfitting. Ultimately, an output layer, `fc3`, produces predictions corresponding to the specified number of classes. The model uses Leaky ReLU with a slope of 0.01 for all activations, aiding in mitigating issues like dying neurons.

Xavier initialization ensures efficient weight initialization, and the Adam optimizer with a learning rate of 0.0001 facilitates effective training. Early stopping based on validation loss improvement further enhances the model's generalization performance.

Proceeding into the training and evaluating phase, while Scikit-learn is utilised to perform data splitting tasks as with other models, PyTorch's `Dataset` and `DataLoader` are additionally introduced for data handling. Since CNNs typically train with mini-batches rather than

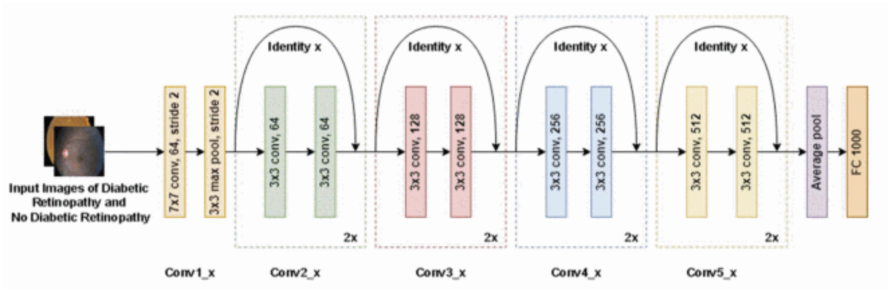

Figure 9: The structure of Resnet-18 model, a pretrained 2D CNNs from PyTorch (67).

single samples, both libraries are beneficial for managing the complexity of preprocessing, batching, and parallel data loading. During training and validation, an *epoch*—a complete forward and backward pass on the entire dataset—is set with an early stopping condition when validation loss has not decreased for n consecutive iterations. The functions `model.train()` and `model.eval()` from `torch.nn` are used for these phases.

As mentioned above, optimisation techniques such as the Adam optimizer, batch normalization, dropout, Xavier weight initialization, and early stopping are incorporated. Moreover, a visualisation of training loss and accuracy across epochs is provided to demonstrate the model's learning dynamics and the risk of underfitting or overfitting.

5.2.6 2D Convolutional neural networks (2D CNNs)

Similar to 1D CNNs, PyTorch is selected for the implementation but with the `torchvision` library to handle image-based tasks. A pre-trained ResNet-18 model, a deep convolutional neural network with 18 layers, is chosen among other 2D CNNs architectures due to its capabilities in transfer learning and avoidance of overfitting on small datasets. Comparing to VGG16 and ResNet-50/101, these options have no residual connections to support gradient flow and are much deeper in number of layers which require more data for generalisation, respectively.

Figure 9 depicts the structure of ResNet-18, consisting of 7 distinct stages or 18 layers. The first stage is responsible for extracting low-level features, such as edges and textures, and reducing input size. Then, four consecutive 2-residual blocks are followed to increase feature depth and lower spatial dimensions. In the last two stages, global average pooling and the fully connected layer are presented transform the feature maps into a single value and map these features to final class scores for classification.

Before training, the dataset, in a tabular format, is converted to a square grayscale image and transformed to match input shape and scale of the pretrained model. To illustrate, the grayscale conversion involved extracting all numerical columns, scaling feature values to [0, 255] range, zero padding to form a perfect square, reshaping the padded vector into a 2D matrix and storing as `PIL.Image` in grayscale format. This followed by a transformation on the images where the input is resized to a 224×224-pixel resolution and normalised with mean and standard deviation of the RGB channels of ImageNet.

The training-validating-testing phase in 2D CNNs is resembled the one in 1D CNNs incorporating Scikit-learn Train–Test Split and PyTorch `Datasets` & `DataLoader` modules. Optimisation mechanisms and evaluation metrics are also identical to the previous model, except an addition of transfer learning. The weights pre-trained on ImageNet of the model is loaded by setting the parameter `pretrained=True` in `models.resnet18()`. Then the input size of the output layer is adjusted in accordance with the number of classes label of the dataset.

5.2.7 Long Short-Term Memory (LSTM)

An LSTM model is classified as a type of recurrent neural network in which long-term dependencies are handled by the LSTM cells. The sequences of network traffic data are ingested by the system using the time-window sampling strategy. The flow of the data datapoints are sorted by timestamp and then grouped by a fixed length of 10 consecutive datapoints (1 window). This is noted as a requirement for LSTM modelling.

The input layer is designed to accept the input sequence length of 10 datapoints (as set from the sampling) along with the feature size of 24. The first LSTM layer consisted of 64 units; each unit is made of input, forget, and output gates. Additionally, the (`return_sequences=True`) is set, thus the entire sequences are seen by the next recurrent layer. A dropout rate of 0.2 is selected to prevent overfitting by randomly silencing 20% of the LSTM's outputs on each update. The second layer consisted of 32 units, creating the sequences into a single 32-dimensional vector. The third "bottleneck" layer is now made of 16 units with a ReLU activation function to introduce non-linearity before the final projection. Multi-class classification is accomplished by the last output layer by using the softmax activation function with units equal to the number of aimed class labels.

The Adam optimiser is chosen to ensure that the model's weights are updated in the correct direction using estimates of past gradients' mean and variance. The learning rate for each parameter is adapted dynamically so that weights updated frequently receive smaller step sizes, while those updated less often are granted larger ones.

6 Simulation Environment

6.1 Design of Simulated Attacks

The simulation uses multiple Docker environments. Each component in the simulation has its own Dockerfile. The Dockerfile consists of the instructions that are required for each component in the simulation. The victim-server container is simulated as the target of the DDoS attack. There are several attacker containers, each consisting of their own Dockerfile with the instruction on simulating the required attack. The pcap-to-csv Dockerfile consists of the instructions that are required to covert the traffic captured in the PCAP file to a CSV format. Using Docker for the simulation ensures that the same tools and dependencies are used regardless of the machine the simulation is being hosted on. The containerisation of the victim-server and attacks also helps in isolating the system. This safeguards the host machine from being affected from the attacks and traffic being generated.

The benign attacker container employs a script with an infinite loop in which batches of connections are sent every few seconds. The script is written in a shell script (`.sh` file extension) and uses `curl` to perform basic data transfer and wait for the response by the server. After receiving the response, the attacker waits for some seconds and then acts again. The batch size and wait times are randomised each time between some set limits.

In comparison, the Low Orbit Cannon (LOIC) HTTP attacker uses `curl` to send the attack. LOIC is a commonly used tool that works by flooding a target server with packets, which in this case are HTTP packets (68). This attack is launched to the server with `--silent` to keep the output clean and `--output /dev/null` to not display the response body. This is kept inside an infinite loop with a short delay between iterations.

A very different attack employed was GoldenEye, for which a Python library was available (31). GoldenEye is an open-source HTTP DoS Test Tool. However, the default version of the code is too strong to simulate in this environment; thus, the strength was reduced by adding a line of code. The code waits for a small amount of time between requests.

Another attack that uses a Python library is Slowloris (33); however, how this attack works is very different from most of the previously seen ones. Slowloris is an application layer attack that starts connections to a server and keeps them active with partial HTTP requests. This uses a low amount of bandwidth with requests that seem slower than normal but mimic regular traffic (34).

The last container is the SlowHTTPTest open-source tool. This tool implements low-bandwidth DoS. In this implementation, SlowHTTPTest simulates 1000 concurrent connections and 200 requests per second. It is configured to send slow headers to overwhelm the server (69).

6.2 Data Generation

The data generation process was split into 2 different files. The first file is called `capture.pcap` and this file stores the packet information from attacks. The second one is `attack_timeline.csv` which stores the timeline of attacks to track the labels of each packet in a space-efficient way. After generation, both files are used to generate the `capture.csv` file which is the final dataset.

For safety, before re-generating `attack_timeline.csv`, this file is deleted from the output directory. Once the victim server starts, it also removes the `capture.pcap` and `capture.csv` that could be left in the output folder.

The `attack_timeline.csv` file stores the `time` and `label` variables of each event. The `time` variable is structured as a single number in this style `YYYYMMDDHHMMSSmmm` which is required to follow the same format as the time in `capture.pcap`. The `label` tracks the name of the event, which can either be the name of the attack start or `start` to indicate the start of the server.

Logging events in `attack_timeline.csv` is done by the automation of the simulation when orchestrating when everything runs (The details about the orchestration of the system will be described in the next section, §6.3). The automated system triggers the server when starting a simulation; however, before that, the `start` event is stored. Then, different attacks are run, and their labels are logged by the automated system.

The other file, `capture.pcap`, gets generated by the victim server using a tool called `tcpdump`. Tcpdump is a tool that can print or store a description of packet contents in a network, including a timestamp in the previously mentioned format (70).

Once the server container in Docker is stopped by the automated system, the `tcpdump` stops, and the final versions of the `capture.pcap` and `attack_timeline.csv` files are created. However, these two files are big, and not a good way to store the information needed for the ML models. Instead, another container called `pcap-to-csv` runs with a Python script that calculates the features and labels in the dataset and outputs it as the `capture.csv`.

6.3 Automation of Simulation

The simulation of the Distributed Denial-of-Service (DDoS) has been automated by using Ansible. The `ddos_attack_sequence.yml` playbook defines the series of tasks required to automate the running of the simulation.

Listing 1: Starting of the playbook

```
# ddos_attack_sequence.yml
- name: Build and run DDoS attack simulation
  hosts: localhost
```

At the start of the playbook the play is described in the name: section. The next line hosts: refers to the target host for the play. In the case of the simulation, it is localhost as all the Docker containers are being run locally on the machine.

Listing 2: Defining variables in Ansible

```
vars:
  attack_duration: 2
  attack_list:
    - name: attacker-benign
      path: attacks/attacker-benign
    - name: attacker-goldeneye
      path: attacks/attacker-goldeneye
    - name: attacker-loic-http
      path: attacks/attacker-loic-http
    - name: attacker-slowhttp
      path: attacks/attacker-slowhttp
    - name: attacker-slowloris
      path: attacks/attacker-slowloris
  victim_image: victim-server
  victim_path: victim-server
```

The variables being used in the playbook are defined next in the vars: section. These variables store values that are later being used in the playbook. Variables in Ansible are defined using key-value pairs. In this section of the playbook, the duration of each attack has been set and provided alongside is an attack list which defines all the paths to the directories in which the Dockerfile is located to execute that attack. The image name and path of the victim-server have also been passed as variables to be used later in this playbook.

Listing 3: Building the victim-server image

```
tasks:
  - name: Build victim Docker image
    shell: docker build -t {{ victim_image }} {{ victim_path }}
```

Ansible has its own built-in `shell` module, which is used throughout the playbook to execute shell commands. The very first task is to build the victim Docker image. Here, the `shell` command leverages the variables defined earlier—`victim_image` and `victim_path`—to construct the `victim-server` Docker image:

Listing 4: Removing any existing `victim_server` container

```
- name: Remove existing victim container (if any)
  shell: docker rm -f victim_server || true
```

The next task defined is to remove any already existing victim container. This is done as part of handling the error of a victim–server container already existing when the image is run later in the playbook. The use of '—— true' ensures that the command never fails even when the container does not exist, so that the playbook does not halt on an error:

Listing 5: Deleting any previous attack timeline

```
- name: Delete previous attack timeline log
  file:
    path: out/attack_timeline.csv
    state: absent
```

This task uses Ansible's built-in 'file' module to delete the previous attack timeline log. The 'path' key directs where the file is located. The 'state: absent' setting ensures that the file does not exist: if the file does exist it is deleted, and if it does not exist the playbook simply proceeds without error:

Listing 6: Capturing UTC timestamp

```yaml
- name: Save tcpdump start timestamp
  shell: |
    now_date=$(date -u +"%Y%m%d%H%M%S")
    now_ms=$(python3 -c "from datetime import datetime; print(f'{datetime.utcnow().
    microsecond // 1000:03d}')")
    now="${now_date}${now_ms}"
    echo "$now,start" >> out/attack_timeline.csv
```

This task captures the current UTC timestamp to the millisecond. That timestamp is then appended to the `attack_timeline.csv` file in the `out` directory with a “start” tag, indicating the time at which the simulation began:

Listing 7: Starting the victim container

```yaml
- name: Start victim container
  shell: |
    docker run -d \
      --name victim_server \
      --network host \
      -p 80:80 \
      -v "{{ playbook_dir }}/out:/output" \
      {{ victim_image }}
```

This task starts the running of the `victim_server` Docker container. The `-d` flag runs the container in detached mode, allowing it to be run in the background. The port is mapped by `-p 80:80`, which maps port 80 of the container to port 80 of the host. The `-v` flag mounts the `/out` directory from the playbook into the container's `/output` directory. The `victim_server` has been started and is ready to be attacked. All traffic to the `victim_server` is also logged in the `/out` directory.

Listing 8: Running of attacker simulation

```yaml
- name: Run attacker simulations
  include_tasks: tasks/run_attack.yml
  loop: "{{ attack_list }}"
  loop_control:
    loop_var: attack
```

The next task utilises Ansible's code-reusability feature (71). The `include_tasks` module runs the tasks in `tasks/run_attack.yml`. The `loop:` iterates over every attacker defined earlier in `attack_list`, reusing `run_attack.yml` for each attack. The loop variable is set to `attack` under `loop_control`, enabling `run_attack.yml` to access {{ `attack.name` }} and {{ `attack.path` }} to build and run each DDoS attack defined in `attack_list`.

Listing 9: Building, logging and running attackers

```yaml
- name: Build attacker image for "{{ attack.name }}"
  shell: docker build -t {{ attack.name }} {{ attack.path }}

- name: Log attack start timestamp
  shell: |
    now_date=$(date -u +"%Y%m%d%H%M%S")
    now_ms=$(python3 -c "from datetime import datetime; print(f'{datetime.utcnow().
    microsecond//1000:03d}')")
    now="${now_date}${now_ms}"
    echo "$now,{{ attack.name }}" >> out/attack_timeline.csv

```

```
- name: Run attacker container for "{{ attack.name }}"
  shell: |
    docker run -d \
      --name {{ attack.name }} \
      --network host \
      {{ attack.name }}
```

In `run_attack.yml`, each attacker Docker image is built, its timestamp recorded, and the container run similarly to the victim server. The variables {{ `attack.name` }} and {{ `attack.path` }} iterate through each attack in `attack_list`.

Listing 10: Setting duration of attacks

```
- name: Let the attack "{{ attack.name }}" run for a bit
  pause:
    seconds: "{{ attack_duration }}"
```

This task utilises Ansible's `pause:` module to ensure each attacker container runs for a set duration. The `seconds:` value is taken from the {{ `attack_duration` }} variable defined in `ddos_attack_sequence.yml`.

Listing 11: Attacker cleanup

```
- name: Stop and remove attacker container "{{ attack.name }}"
  shell: docker rm -f {{ attack.name }} || true

- name: Wait 10 seconds before next attack
  pause:
    seconds: 10
```

In the next two tasks, each attacker container is removed if present, and a 10-second delay is inserted before the next attack. This buffer prevents overlap, since containers may not stop and remove immediately.

Listing 12: Capturing late packages on the victim server

```
- name: Wait 10 seconds before shutting down the victim
  pause:
    seconds: 10
```

A 10-second buffer is also added before the victim server is removed, allowing late packets to be logged.

Listing 13: Victim server cleanup

```
- name: Stop and remove victim container
  shell: |
    docker rm -f victim_server || true
```

This task stops and removes the `victim_server` container. If the container is already stopped, the playbook proceeds without error.

Listing 14: PCAP to CSV conversion

```
- name: Remove existing pcap-to-csv container (if any)
  shell: docker rm -f pcap-to-csv || true

- name: Build pcap-to-csv image
  shell: docker build -t pcap-to-csv pcap-to-csv/

```

```
- name: Start pcap-to-csv container
  shell: |
    docker run -d \
      --name pcap-to-csv \
      -v "{{ playbook_dir }}/out:/output" \
      pcap-to-csv
```

These final tasks automate running the `pcap-to-csv` container: first removing any existing container, then building the Docker image, and finally running it. The resulting CSV is output to the `/out` directory.

Listing 15: Variable prompt for specific attacks

```
vars_prompt:
  - name: selected_attacks
    prompt: "Enter the attacker names you want to run (comma-separated)"
    private: no
```

The `ddos_attack_specific.yml` playbook mirrors `ddos_attack_sequence.yml` but uses `vars_prompt:` to request user input for specific attacker names. Setting `private: no` allows the input to be displayed. Only the selected attacks from `attack_list` are run, and the CSV output contains data for those attacks alone.

Listing 16: Running specific attacks

```
- name: Run attacker simulations
  include_tasks: tasks/run_attack.yml
  loop: "{{ attack_list | selectattr('name','in', selected_attacks.split(',') | map
    ('trim') | list) }}"
  loop_control:
    loop_var: attack
```

The only other task with a slight difference is the running of attacks. This playbook also utilises the `run_attack.yml` playbook. However, the only attacks selected from the `attack_list` are those that have been selected by the user at input. The rest of the playbook runs the same, with a CSV file outputted into the `out` directory consisting of the data only of the specified attacks.

7 Components of the Integration System

7.1 Machine Learning Automated Pipeline (ML_System)

One of the components of the final integrated system is the ML_System where an automated pipeline is constructed to train, evaluate, export the models to test with the simulated data. The diagram below depicts a specific architecture and workflow of the system, extending from the one in Section 3.

Based on Figure 10, six main components or folders in grey are instantiated to implement the pipeline. The initial stage of this system starts by running the `integrator.ipynb` which loads data, executes each model, and saves the results in the stated sequence. Then the `testscript.ipynb` handles another evaluation with a dataset generated by the simulation team. The overall steps of the pipeline are as follows:

1. The `integrator` file in the `scripts` folder is executed to initiate the ML pipeline including data ingestion and processing, model training–validation–testing, extraction of the trained models, and another evaluation with a simulated dataset.

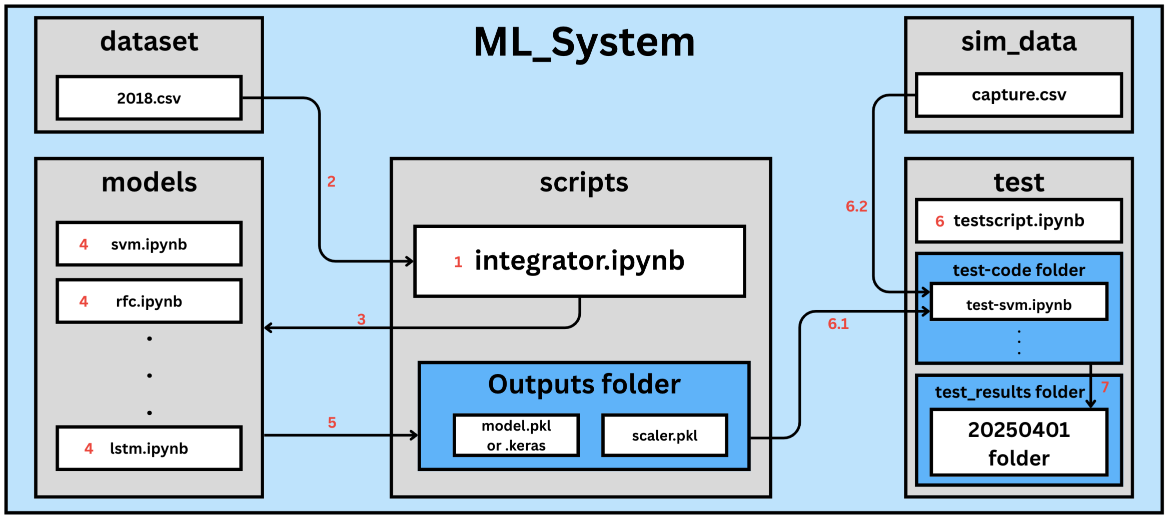

Figure 10: An overall architecture and workflow of the ML_System which illustrates the components and step-by-step execution of the system when being automated.

2. A dataset for training, `2018.csv`, is loaded to the `datasets` folder.
3. The integrator injects the dataset path to each model in the model folders.
4. In each model folder, basic data processing is performed to ensure {data characteristics}, followed by a train–validate–test cycle with hyperparameter tuning when necessary. Each model then exports evaluation results (classification report, confusion matrix) and saves the trained model and its scaler.
5. All outputs, evaluation results, and model files are stored in the `Outputs` folder within `scripts`.
6. A `testscript` in the `test` folder loads the pretrained models from `test-code` and evaluates them on the simulated `capture.csv` from `sim_data`.
7. The final test outputs are saved in `test_results` for analysis.

7.2 Simulation Pipeline (Sim_System)

1. First, the Ansible Playbook `ddos_attack_sequence.yml` is run to initiate the automated simulation.
2. Next, the Playbook builds the Docker image for the `victim-server` and then runs the container.
3. Then, the Playbook runs the `run_attack.yml` Ansible Playbook located in the `tasks` directory.
4. The `run_attack.yml` playbook iterates through all the attacks in the `attack_list` specified in the `ddos_attack_sequence.yml`, which are located in the `attacks` directory. Each attack runs for 30 seconds.
5. The `victim-server` is continuously capturing traffic during each attack and saving it to the `capture.pcap` file.

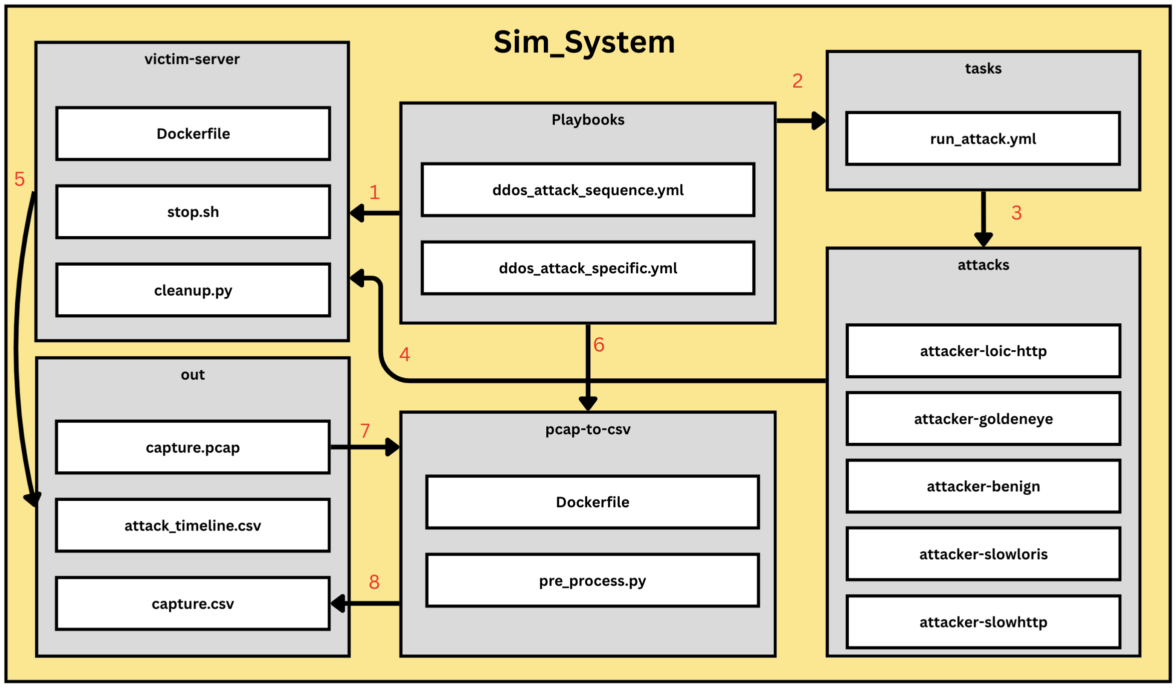

Figure 11: Architecture and workflow of the Sim_System illustrating automated DDoS simulation, traffic capture, and dataset preparation for ML models.

6. Once all the attacker containers have been run and removed, the `victim-server` container is stopped and removed as part of the cleanup process. The Playbook then builds the `pcap-to-csv` Docker image and runs the `pcap-to-csv` container.

7. This container executes the `pre_process.py` script, which serves as the interface between the simulation output and the ML input pipeline. During this stage, the `pre_process.py` script processes the captured `capture.pcap` file and generates `attack_timeline.csv`.

8. The `attack_timeline.csv` file is generated by extracting the start times of each attack based on the simulation schedule and associating each start time with the corresponding attack name (attack type). Each record includes an attack name and its corresponding start timestamp, formed as `(AttackType, YYYYMMDDHHMMSSmmm)`.

9. Next, the labeling process is performed using the information in `attack_timeline.csv`. The attack start times are converted into numerical formats compatible with the timestamps in the parsed network flows. Attack intervals are constructed, where each interval starts at the recorded attack time and ends at the start time of the next attack. Each flow is assigned a label based on which interval its timestamp falls into. If a flow falls within an attack interval, it is labeled with the attack name specified at the start of that interval; otherwise, it is labeled as `"BENIGN"`.

10. Ultimately, the converted `capture.csv` is stored in the `out` directory. The ML models can then use this data generated by the simulation of the DDoS attacks.

11. In the case that the `ddos_attack_specific.yml` Playbook is run, an input of specific attacks is taken. The rest of the steps above remain the same, with the only difference being that specific attacks are selected from the `attack_list` and only those attacker containers are run.

7.3 Final System Integration (Orchestration)

The last component of the system played a crucial role in merging ML and Simulation systems together. It mainly comprised of a single file named `controller.ipynb`, with seven functions and a loop, where all activities of the other two systems are monitored and regulated.

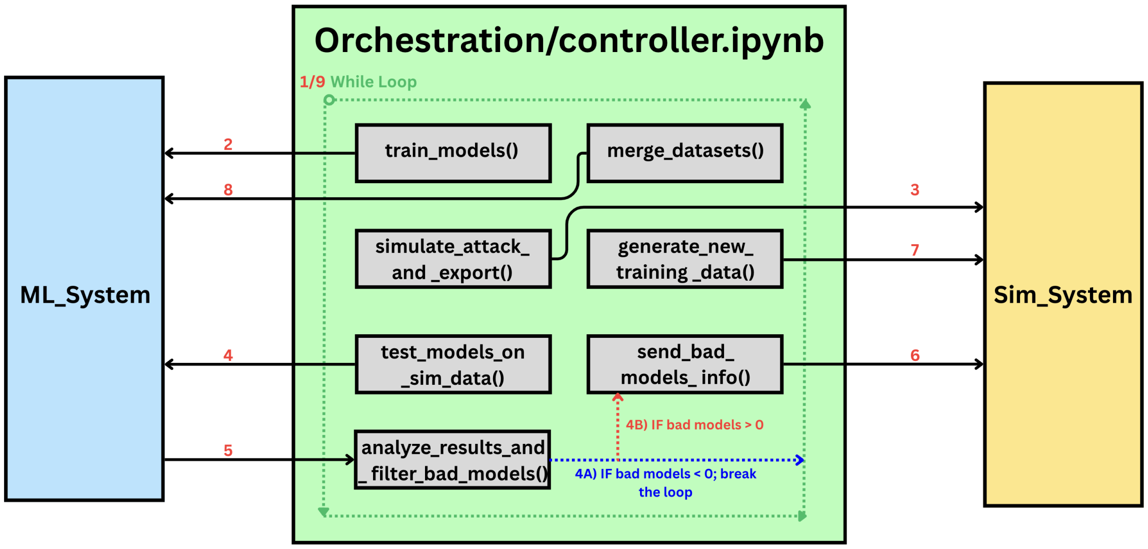
Orchestration/controller.ipynb
1/9 While Loop
ML_System
Sim_System
train_models()
merge_datasets()
simulate_attack_and _export()
generate_new_training _data()
test_models_on_sim_data()
send_bad_models_ info()
analyze_results_and_filter_bad_models()
2
8
3
7
4
6
5
4B) IF bad models > 0
4A) IF bad models < 0; break the loop

Figure 12: An overall architecture and workflow of all activities performed within the controller file of the Orchestration directory. It highlights how each function regulates and automates the executions among the other two systems.

Corresponding to the diagram above (Fig. 12), while each function resembles an interface between the two systems, a loop is introduced to allow an iterative development of the models and data simulation until all models have been satisfied, or the maximum number of iterations has been reached. The overall steps of the pipeline are as follows:

1. The system is started by running a while loop in the `controller.ipynb` of the Orchestration folder where the seven functions are called for executions in the ML_System or Sim_System.
2. In the loop, function 1, `train_models()`, is the first aspect to be executed with a triggering effect on `integrator.ipynb` for producing a set of trained models.
3. Simulated DDoS attacks and data related activities in Sim_System are generated by function 2, `simulate_attack_and_export()`, to be used for testing on the trained models.
4. Once the tested dataset from Sim_System has arrived at `ML_System_sim_data` directory, function 3, `test_models_on_sim_data()`, is signaled for an evaluation of the trained models on this new simulated data.
5. The test accuracy of each model is then extracted from the `test-code` folder to filter a good and bad model based on an accuracy threshold at 90%. For good models, the trained models are stored for further uses; however, the bad ones will be trained again until one of the criteria is met. If all models are good before the last iteration, the execution will break at this point. These actions are operated by function 4, `analyze_results_and_filter_bad_models()`.

6. Based on the results of the previous step, function 5, `send_bad_models_info()`, is set up to provide an update of bad model(s) to Sim_System for new data generation.

7. This is followed by a call on function 6, `generate_new_training_data()`, to perform similar work as function 2, except that this generated data from this step will be used for training alongside `2018.csv` and not for testing.

8. The last step is handled by function 7, `merge_datasets()`, where the newly generated training dataset from Sim_System will be combined with the latest version of `2018.csv`, if not in the first iteration, and used for model training in the next iteration.

9. As stated in step 5, the loop will be stopped if either all trained models are good, or the last iteration has been reached.

8 Results and Evaluation

8.1 Testing Procedures and Setup

8.1.1 Test System 1: Results from train-validate-test stages in "models folder"

In test system 1, tested results are produced in the testing step after a training and validation stages of the developed model in `model_name.ipynb`. The dataset used for this test is from the 15–20% partition of `2018.csv` during the data splitting step. A set of parameters of the model, validated in the previous step, will be used for testing where optional hyperparameter tuning or other optimisations are executed if required. Alongside this, an automated development of the models is enabled via `integrator.ipynb` as described in section 7.1. All the models will be executed sequentially with the dataset being loaded into each model's file.

8.1.2 Test System 2: Results from testing procedures in "test folder"

Proceeding into the second test, tested outputs in this part are attributed from applying trained models from the development in each `model_name.ipynb` with an unseen dataset, `capture.csv`, from the simulation team. The models are saved in `.pkl` or `.pth` files before being loaded into the tested file, `test_model_name.ipynb`. The structure of each test code follows similar patterns to system 1, apart from the replacement of training and validating steps by directly loading the pretrained model. Test system 2 is also automated by the `testscript.ipynb` where the `capture.csv` and pretrained model are injected into each test code for evaluation outcomes.

8.1.3 Test System 3: An integration of test system 1 & 2 in "orchestration folder"

Ultimately, as illustrated in section 7.3, the objective of the result from this system is to depict the improvement of the models based on `2018.csv` and newly generated `capture.csv` files. In each iteration, each model will be tested in systems 1 and 2 where their accuracy in system 2 will be assessed to determine additional rounds of training and testing. With extra datapoints added to the training dataset from an incremented iteration, the performance of the models will be analysed to see whether the new training dataset has generalised and improved their classification performance.

8.2 Evaluation Metrics

8.2.1 Results – Test System 1

Table 3 gives a summary of the classification report, including accuracy, weighted average precision, weighted average recall, and weighted average F1-score, of each model's predictive performance on the `2018.csv` dataset (in %).

Table 3: A summary of the classification report, including accuracy, weighted average precision, weighted average recall, and weighted average F1-score, of each model's predictive performance on the `2018.csv` dataset (in %).

Model	Accuracy	WA Precision	WA Recall	WA F1-Score
K-Means (K=10)	74.05	86.77	88.03	87.21
K-Means Seeded (K=8)	83.96	85.04	83.96	83.95
Logistic Regression (LR)	99.00	100.00	99.00	100.00
Support Vector Machine	99.78	100.00	99.88	99.00
Random Forest (RFC)	100.00	100.00	100.00	100.00
1D CNN	99.96	99.92	99.70	99.50
2D CNN	51.73	54.62	49.68	52.07
LSTM	100.00	100.00	100.00	100.00

The prediction performance of the eight models is shown in Table 3, with an average accuracy of 88.56%. The other three metrics of the classification report also indicate promising results, as they are weighted averages that remove potential biases from class imbalance in the testing data. While ensemble methods such as Random Forest and deep learning models such as LSTM outperform the others, K-Means and 2D CNNs are the two worst performers, with accuracies of 74.05% and 51.73%, respectively. It is noted that the 2D CNN's lower performance is primarily due to the limited dataset size (1:100 input ratio) used for training and evaluation, a necessary compromise to manage the high computational demands of 2D convolutional architectures.

8.2.2 Results – Test System 2

The recall accuracy of each type of simulated attack with the greatest model predictors is shown in Table 4. A clear predictive strength in accurately identifying at least one attack is demonstrated by each machine learning model. Outstanding recall performance for benign traffic is established by K-Means Seeded and 1D CNNs, though their classification accuracy across other attack types remains low. This suggests a specialised effectiveness in recognising normal traffic behaviour. Similarly, notable recall for detecting the Slowloris attack is represented by Logistic Regression and K-Means models. Lastly, exceptional recall to detect specific attacks—i.e., Loic-HTTP—is achieved by the Random Forest Classifier, and for SlowHTTP by the SVM.

8.2.3 Results – Test System 3

Random Forest Classifier was chosen to exemplify the integration system which involved executions both in ML and Simulation systems. Based on Figure 13, the model achieved an improvement of 2.34% over the initial five iterations. In each new iteration, the training

Table 4: A summary of each specific attack being detected most effectively by the trained models via recall score (%).

Attack Name	**Model**	**Recall (%)**
Benign	K-Means Seeded	99.32
	1D CNNs	98.53
	2D CNNs	50.22
Slowloris	Logistic Regression (LR)	99.94
	K-Means	78.43
Slowhttp	Support Vector Machine	99.76
Loic-HTTP	Random Forest Classifier	99.93

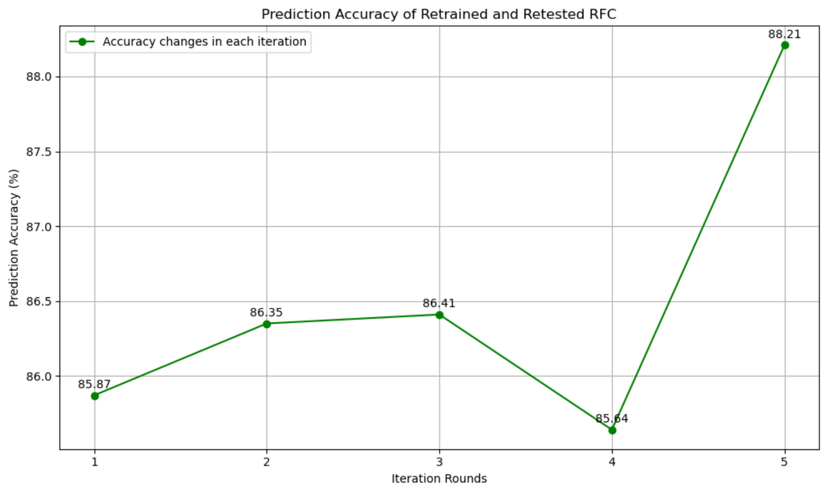

Figure 13: Accuracy (%) of iterative training and testing of the Random Forest Classifier with simulation-generated test datasets.

dataset generated by the simulation system is concatenated with the previous round's training set. This increase in data quantity leverages the model's predictive power.

8.2.4 Additional Results from Evaluation Metrics

Confusion Matrix – 1D CNNs According to Figure 14, a high rate of successful predictions is visible along the diagonal, where the model's predictions match the true labels. Out of 325,474 total predictions, only 134 were misclassified (95 false positives between Slowloris and Benign). The model correctly detected both abundant classes (e.g. 180,620 benign flows) and minority classes (e.g. 128 LOIC–UDP flows), illustrating both overall accuracy and sensitivity to rare attacks.

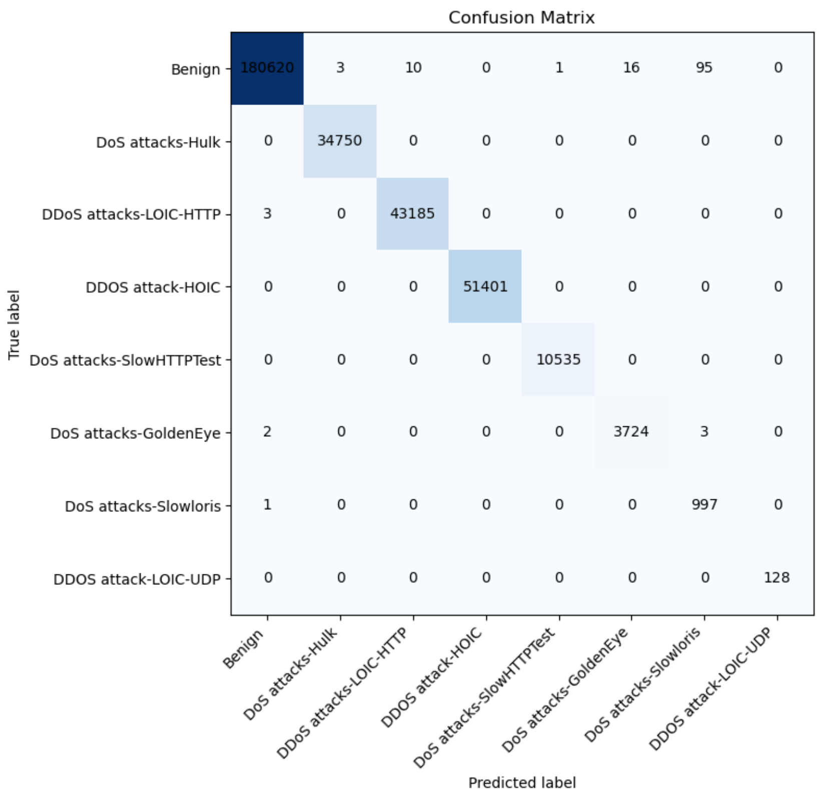

Figure 14: Confusion matrix for the 1D CNN model on Test System 1, showing true positives, false positives, true negatives, and false negatives.

Feature Importance – Random Forest Figure 15 ranks input features by their Gini-based importance. `timestamp` and `entropy_pkt_len` top the list with scores of 0.175 and 0.150, respectively—together contributing over one third of the total importance. Remaining features each contribute below 7.5%, with `Bwd Pkt Len Min` and `SYN Flag Cnt` under 1%.

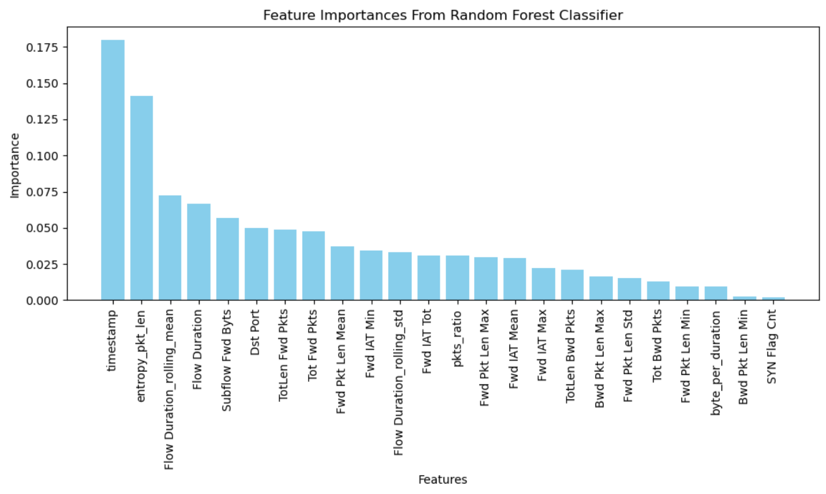

Figure 15: Feature importance scores calculated by the Random Forest Classifier on the 2018.csv dataset.

Hyperplane Visualization – SVM Figure 16 shows the SVM decision regions in 2D after PCA reduction. While classes such as SlowHTTPTest and Benign are clearly separable, several decision boundaries overlap, reflecting the intrinsic similarity of some attack patterns when projected to two dimensions.

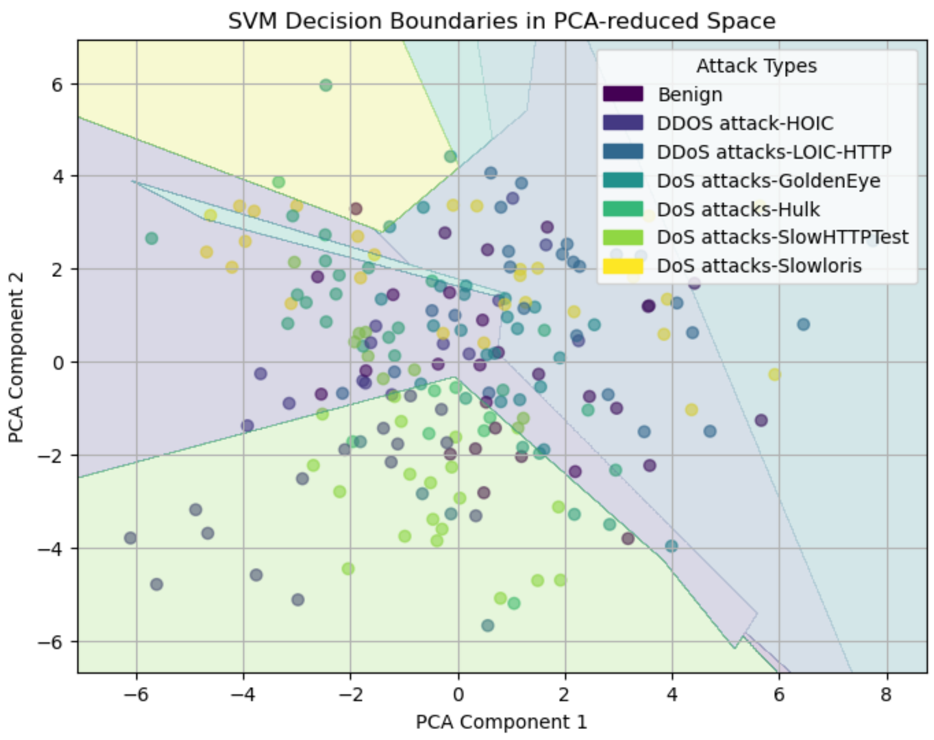

Figure 16: PCA-reduced SVM decision boundaries for the seven-class classification problem.

Cluster Evaluation Metrics Figure 17 plots Silhouette and Calinski–Harabasz scores for k from 2 to 14. Both metrics increase until $k = 10$, where they plateau, indicating that ten clusters optimally balance cohesion and separation for this dataset.

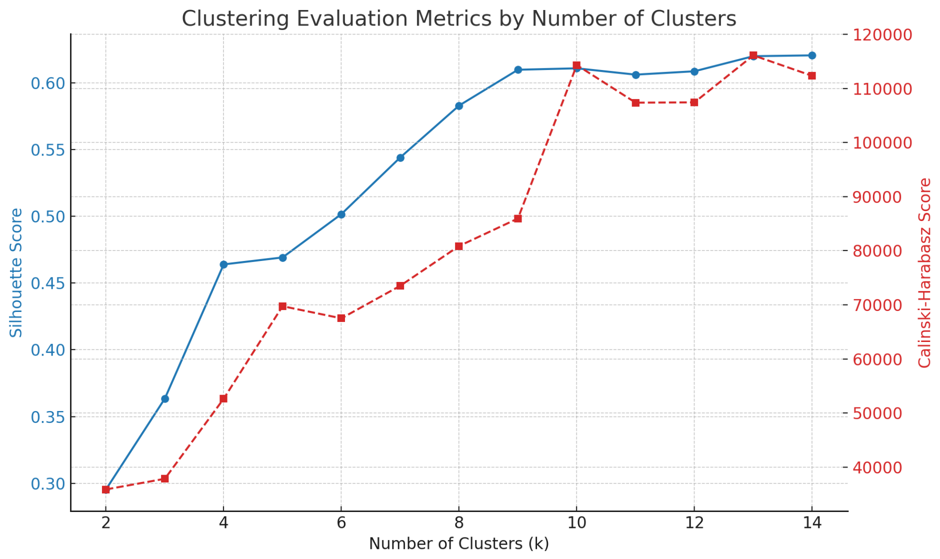

Figure 17: Silhouette and Calinski–Harabasz scores as functions of cluster count k.

How well each data point fits within its assigned cluster compared to the nearest neighbouring cluster is measured by the Silhouette Score, while the ratio of between-cluster dispersion to within-cluster dispersion is represented by the Calinski–Harabasz Score. As the number of clusters, k, is increased from 2 to 14, both the Silhouette and Calinski–Harabasz scores are seen to rise overall signalling that the clusters are more well separated A steep climb is observed from 0.30 at k=2 to approximately 0.46 at k=4, after which it continues to rise until k=10, where it plateaus. A similar trajectory is observed in the Calinski–Harabasz Score, which is seen to rise from around 0 to about 70 000 before being stabilised at approximately 110 000 by k = 10. At k=10 is suggested to be the optimal number of clusters for this dataset.

9 Discussion

9.1 Analysis of Findings

The project aimed to create a closed-loop automated system that was able to accurately detect Distributed Denial-of-Service attacks using a machine learning and deep learning models.

Random Forest Classifier (RFC) and Long Short-term Memory (LSTM) models achieved an almost perfect performance. It was accurately able to detect benign and attack traffic. 1D CNN and K-Means Seeded specialised in detecting benign traffic, achieving over 98% recall. Even though it was less effective at detecting complex attacks, they were excellent in distinguishing non-malicious traffic. In the integrated system, 2.34% of improvement was achieved by iteratively training on the new generated simulation data.

Critical features that were identified were 'Timestamp' and 'entropy_pkt_len', which contributed to over 32% of the total predictive power. This highlights the importance of temporal and structural characteristics of network traffic to detect anomalies. Despite high accuracy, PCA visualisations showed overlaps between the attack types. In the lower dimensions the attack patterns appear to be similar, which emphasises the complexity of distinguishing subtle DDoS variations.

Ensemble and deep learning models are highly capable of detecting DDoS attacks in a simulated environment. The closed-loop automation system design effectively improves itself over time, which can change the DDoS threat landscape in the real-world.

9.2 Challenges and Limitations

These are the several challenges that arose. The DDoS dataset is incomplete with missing values and zeros. The incomplete data problem was affected by having a large amount of data because the preprocessing needed was harder to implement and use.

The size of the datasets also impacted the speed and resources required to train the models, especially because they were run locally, for the extra flexibility in ML training. The slower training also obstructed the process of fixing errors and loops.

Another challenge was to simulate DDoS attack data while remaining within the law; to solve this issue, a closed environment with Docker was used. However, Docker caused an issue with the Ansible library for Docker that we could not fix; this issue was resolved using shell commands instead.

In addition to the challenges encountered during development, the final system has inherent limitations. A limitation in the final simulation implementation is how the system tracks attackers using the current time. Using time, the possibility of multiple attacks being simulated at the same time isn't possible.

Since the simulated attacks were developed manually, any other attacks to be simulated also had to be created manually. This is not a very big limitation since the automation was coded with modularity in mind; however, this modularity of the automation does not allow triggering attacks in a different way from the rest.

Another limitation of the fully automated system is how attack patterns evolve to bypass detection. Any automated system must also keep evolving to fit the ever-changing network and traffic environments in real-world scenarios.

9.3 Future Work

Potential future research directions span both software and hardware improvements. On the software side, one major avenue is to retrain the model periodically, ensuring it remains effective against evolving and previously unseen DDoS patterns. Another important enhancement is to incorporate explainability tools, such as SHAP or LIME, to better interpret the model's decisions and improve transparency and trust in deployment environments.

Moreover, a critical step forward is enabling real-time DDoS detection, which necessitates making the model more lightweight and computationally efficient. This would allow deployment on live networks, such as SDN controllers or edge devices, to perform instant traffic filtering and mitigation as attacks occur.

Another significant direction is to redo the entire pipeline using a binary classification dataset, labelling traffic as either "BENIGN" or "ATTACK" regardless of the specific attack type. This simplifies the classification problem, improves generalizability, and aligns the model with broader security applications.

On the hardware side, future work can explore edge-based detection, developing lightweight modules that operate at the periphery of networks—such as gateways or routers—to intercept and mitigate threats before they reach core infrastructure. Existing research has demonstrated the feasibility of compact intrusion detection systems for resource-constrained environments like IoT, which can be further refined and adapted to the current DDoS simulation pipeline.

10 Conclusion

10.1 Summary of Contributions

To be productive, the project was organised into 3 teams: Data, Machine Learning (ML) models, and Simulation. Each team was responsible for their part of the closed-loop DDoS detection pipeline, while continuously feeding insight back to the entire team. Team selection was done by voting on what each member wanted to do the most.

The Data team was composed of one person in charge of gathering the original datasets and doing pre-processing on the datasets. She also worked on pre-processing the generated data by the simulation and automating that process.

The ML team was composed of two members who had to work on researching and implementing different models. They split all the models used between them, training and testing them. After that, they worked together on making the Machine Learning Automated Pipeline and Final System Integration.

Two people formed the Simulation team. This team oversaw simulating DDoS attacks and gathering the data for the ML models to train on. One of the members of the team developed the basic design of the automation using Docker, after that he made the Simulation Pipeline with Ansible, and finally he contributed to the Final System Integration. Meanwhile, the other member focused on simulating the different DDoS attacks and automating gathering the data from the simulation.

10.2 Final Remark

This project's objective was to discover if a closed-loop automated detection system for DDoS attacks was viable with the use of Machine Learning and Deep Learning models. By combining a Docker-based simulation and a wide range of ML architectures, it has been demonstrated that it is viable, achieving over 99% recall on some models. The strength of the system comes from the self-learning capabilities of the closed-loop automation, providing extra learning capacities to the system.

11 Acknowledgements and References

We would like to express our sincere gratitude to Dr. Jun Wang and Prof. Paul Wilcox, the EEME Project Directors and Unit Organisers, for their support and guidance throughout the course of this project. We are especially thankful to our project supervisor, Dr. Yulei Wu, for his expert advice, valuable insights, and continuous encouragement, all of which were instrumental in shaping the direction and outcomes of our work.

References

[1] Cloudflare (2024) 'DDoS Threat Report for 2024 Q4'. Available at: https://blog.cloudflare.com/ddos-threat-report-for-2024-q4/

[2] Orcasia (2025) 'CICM – 31 January 2025'. Available at: https://orcasia.org/article/cicm-31st-january-2025

[3] Cloudflare (n.d.) 'What is a DDoS attack?'. Available at: https://www.cloudflare.com/learning/ddos/what-is-a-ddos-attack/

[4] Boin, C., Guillaume, X., Grimaud, G., Groléat, T. and Hauspie, M. (2022) 'One Year of DDoS Attacks Against a Cloud Provider: an Overview'. Available at: https://arxiv.org/pdf/2208.14205

[5] Hesford, J., Cheng, D., Wan, A., Huynh, L., Kim, S., Kim, H. and Hong, J. B. (2024) 'Expectations Versus Reality: Evaluating Intrusion Detection Systems in Practice'. Available at: https://arxiv.org/pdf/2403.17458

[6] Luay, M., Layeghy, S., Hosseininoorbin, S., Sarhan, M., Moustafa, N. and Portmann, M. (2025) 'Temporal Analysis of NetFlow Datasets for Network Intrusion Detection Systems'. Available at: https://arxiv.org/abs/2503.04404

[7] Alahmadi, A.A., Aljabri, M., Alhaidar, F., Alharthi, D.J., Rayani, G.E. and Marghalani, L.A. et al. (2023) 'DDoS Attack Detection in IoT-Based Networks Using Machine Learning Models: A Survey and Research Directions'. Available at: https://www.mdpi.com/2079-9292/12/14/3103

[8] Towards Data Science (n.d.) 'Categorical Variables for Machine Learning Algorithms'. Available at: https://towardsdatascience.com/categorical-variables-for-machine-learning-algorithms-d2768d587ab6/

[9] Amershi, S., Chickering, M., Drucker, S. M., Lee, B., Simard, P. and Suh, J. (2019) 'Software engineering for machine learning: A case study', in *Proceedings of ICSE 2019*, pp. 291–300. https://doi.org/10.1109/ICSE.2019.00053

[10] Teslim, B. (2024) 'Feature Engineering Techniques for Enhancing DDoS Attack Prediction'. Available at: https://www.researchgate.net/publication/384467058

[11] Adedeji, K. B., AbuMahfouz, A. M. and Kurien, A. M. (2023) 'DDoS attack and detection methods in Internet-enabled networks: Concept, research perspectives, and challenges', *Journal of Sensor and Actuator Networks*, 12(4), p. 51. https://doi.org/10.3390/jsan12040051

[12] Shakya, S. and Abbas, R. (2024) 'A comparative analysis of machine learning models for DDoS detection in IoT networks', arXiv preprint, arXiv:2411.05890. Available at: https://arxiv.org/abs/2411.05890 (Accessed: 28 Apr. 2025)

[13] Ardabili, S., Mosavi, A. and Várkonyi-Kóczy, A. R. (2019) 'Advances in machine learning modeling reviewing hybrid and ensemble methods', in *International Conference on Global Research and Education*, Cham: Springer, pp. 215–227

[14] Ali, T. E., Chong, Y.-W. and Manickam, S. (2023) 'Comparison of ML/DL approaches for detecting DDoS attacks in SDN', *Applied Sciences*, 13(5), p. 3033. https://doi.org/10.3390/app13053033

[15] Domingos, P. (2012) 'A few useful things to know about machine learning', *Communications of the ACM*, 55(10), pp. 78–87. https://doi.org/10.1145/2347736.2347755

[16] Gaur, V. and Kumar, R. (2022) 'HPDDoS: A Hyper Parameter Model for Detection of Multiclass DDoS Attacks', Mathematical Statistician and Engineering Applications, 71(3 s2), pp. 1444–1470. Available at: https://www.philstat.org/special_issue/index.php/MSEA/article/view/367

[17] Sokolova, M. and Lapalme, G. (2009) 'A systematic analysis of performance measures for classification tasks', *Information Processing & Management*, 45(4), pp. 427–437

[18] Fawcett, T. (2006) 'An introduction to ROC analysis', *Pattern Recognition Letters*, 27(8), pp. 861–874

[19] Pothuganti, S. (2018) 'Review on over-fitting and under-fitting problems in Machine Learning and solutions', *International Journal of Advanced Research in Electrical, Electronics and Instrumentation Engineering*, 7(9), pp. 3692–3695. Available at: http://www.ijareeie.com/upload/2018/september/8_Review%20on.pdf (Accessed: 28 Apr. 2025)

[20] Sze, V., Chen, Y.-H., Yang, T.-J. and Emer, J. S. (2017) 'Efficient processing of deep neural networks: A tutorial and survey', *Proceedings of the IEEE*, 105(12), pp. 2295–2329. https://doi.org/10.1109/JPROC.2017.2761740

[21] Sommers, J. and Barford, P. (2004) 'Self-Configuring Network Traffic Generation', *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement.*

[22] Garfinkel, T. and Rosenblum, M. (2003) 'A Virtual Machine Introspection Based Architecture for Intrusion Detection', *Proceedings of the Network and Distributed Systems Security Symposium (NDSS).*

[23] Popek, G. J. and Goldberg, R. P. (1974) 'Formal Requirements for Virtualizable Third Generation Architectures', *Communications of the ACM*, 17(7), pp. 412–421.

[24] Chonka, A., Singh, J. and Zhou, W. (2010) 'Chaos Theory Based Deterrence Control Mechanism for DDoS Attacks', *Computer Communications*, 33(18), pp. 2040–2049.

[25] Bou-Harb, E., Fachkha, C., Pourzandi, M., Debbabi, M. and Assi, C. (2013) 'Communication Security for Smart Grid Distribution Networks', *IEEE Communications Magazine*, 51(1), pp. 42–49.

[26] Owezarski, P. et al. (2006) 'A Virtual Machine Approach for Testbeds in Security Research', *Computer Networks*, 50(12), pp. 2152–2166.

[27] Backblaze (2024) 'VMs vs. Containers: Key Differences'. Available at: https://www.backblaze.com/blog/vm-vs-containers/ [Accessed: 04-Apr-2025].

[28] Imran, D., Ahmed, A., Alghamdi, A., Ahmad, F. and Khan, A. (2024) 'Virtual Machines and Containers for Cybersecurity Applications', *Frontiers in Computer Science*, 6. Available at: https://www.frontiersin.org/articles/10.3389/fcomp.2024.1491823/full [Accessed: 04-Apr-2025].

[29] Shekyan, Sergey (2023) 'SlowHTTPTest'. GitHub. Available at: https://github.com/shekyan/slowhttptest (Accessed: 17 April 2025)

[30] Stenberg, Daniel (2025) 'Curl'. curl.se. Available at: https://curl.se/ (Accessed: 17 April 2025)

[31] jseidl (2014) 'GoldenEye'. GitHub. Available at: https://github.com/jseidl/GoldenEye.git (Accessed: 17 April 2025)

[32] Kali Linux (2014) 'Goldeneye — Kali Linux Tools'. Kali Linux. Available at: https://www.kali.org/tools/goldeneye/ (Accessed: 17 April 2025)

[33] gkbrk (2015) 'Slowloris'. GitHub. Available at: https://github.com/gkbrk/slowloris.git (Accessed: 17 April 2025)

[34] Cloudflare (n.d.) 'Slowloris DDoS Attack'. Cloudflare. Available at: https://www.cloudflare.com/en-gb/learning/ddos/ddos-attack-tools/slowloris/ (Accessed: 17 April 2025)

[35] Ansible Documentation (2024) 'Docker and Ansible Scenario Guide'. Available at: https://docs.ansible.com/ansible/2.8/scenario_guides/guide_docker.html [Accessed: 29-Apr-2025].

[36] Kuwv (2024) 'Why I Use Ansible Over Docker Compose', *Dev.to*. Available at: https://dev.to/kuwv/why-i-use-ansible-over-docker-compose-edg [Accessed: 29-Apr-2025].

[37] Canadian Institute for Cybersecurity (2019) 'CIC DDoS 2019 Dataset'. Available at: https://www.unb.ca/cic/datasets/ddos-2019.html

[38] Karatas, G. (2021) 'The Effects of Normalization and Standardization an Internet of Things Attack Detection'. *European Journal of Science and Technology*, Special Issue 29, pp. 187–192. Available at: https://dergipark.org.tr/en/download/article-file/2057090

[39] Disha, R.A. and Waheed, S. (2022) 'Performance analysis of machine learning models for intrusion detection system using Gini Impurity-based Weighted Random Forest (GIWRF) feature selection technique'. *Cybersecurity*, 5(1). Available at: https://cybersecurity.springeropen.com/articles/10.1186/s42400-021-00103-8

[40] Hameed, S. and Ali, U. (2015) 'On the Efficacy of Live DDoS Detection with Hadoop'. Available at: https://arxiv.org/pdf/1506.08953

[41] Wireshark Foundation (n.d.) 'tshark – The Wireshark Network Analyzer CLI Tool'. Available at: https://www.wireshark.org/docs/man-pages/tshark.html

[42] Bala, B. and Behal, S. (2024) 'AI techniques for IoT-based DDoS attack detection: Taxonomies, comprehensive review and research challenges', *Computer Science Review*, 52, p. 100631. https://doi.org/10.1016/j.cosrev.2024.100631

[43] Haseeb-Ur-Rehman, R. M. A., Aman, A. H. M., Hasan, M. K., Ariffin, K. A. Z., Namoun, A., Tufail, A. and Kim, K. H. (2023) 'High-speed network DDoS attack detection: A survey', *Sensors (Basel)*, 23(15), p. 6850. https://doi.org/10.3390/s23156850

[44] Doshi-Velez, F. and Kim, B. (2017) 'Towards a rigorous science of interpretable machine learning', arXiv preprint, arXiv:1702.08608. Available at: https://arxiv.org/abs/1702.08608 (Accessed: 28 Apr. 2025)

[45] Ramzan, M., Shoaib, M., Altaf, A., Arshad, S., Iqbal, F., Castilla, Á. K. and Ashraf, I. (2023) 'Distributed denial of service attack detection in network traffic using deep learning algorithm', *Sensors*, 23(20), p. 8642. https://doi.org/10.3390/s23208642

[46] Songma, S., Sathuphan, T. and Pamutha, T. (2023) 'Optimizing intrusion detection systems in three phases on the CSE-CIC-IDS-2018 dataset', *Computers*, 12(12), p. 245. https://doi.org/10.3390/computers12120245

[47] Fathima, A., Devi, G. S. and Faizaanuddin, M. (2023) 'Improving distributed denial of service attack detection using supervised machine learning', *Measurement: Sensors*, 30, p. 100911. https://doi.org/10.1016/j.measen.2023.100911

[48] Shih, Y.-H., Gong, J., Tang, P., Shen, Y., Liu, T.-Y. and Gao, W. (2019) 'Delineation of urban growth boundaries using a patch-based cellular automata model under multiple spatial and socio-economic scenarios', *Sustainability*, 11(21), p. 6159. https://doi.org/10.3390/su11216159

[49] Göcs, L. and Johanyák, Z. C. (2024) 'Identifying relevant features of CSE-CIC-IDS2018 dataset for the development of an intrusion detection system', *Intelligent Data Analysis*, 28(6), pp. 1527–1553. https://doi.org/10.3233/IDA-230393

[50] Hosmer, D. W., Lemeshow, S. and Sturdivant, R. X. (2013) *Applied Logistic Regression*, 3rd edn., Hoboken, NJ: Wiley & Sons

[51] MacQueen, J. (1967) 'Some methods for classification and analysis of multivariate observations', in *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, vol. 1, pp. 281–297

[52] Basu, S., Banerjee, A. and Mooney, R. J. (2002) 'Semi-supervised clustering by seeding', in *Proceedings of ICML 2002*, pp. 27–34

[53] Al-Dulaimi, K., Saleh, A. and Hammoudeh, M. (2022) ‘Deep learning-based DDoS detection system in cloud computing using feature reduction and 1D-CNN classifier’, *Applied Sciences*, 12(16), p. 7986. https://doi.org/10.3390/app12167986

[54] Aksu, H., Uluagac, A. S. and Babun, L. (2020) ‘A 2D-CNN based deep learning model for DDoS detection in smart grid environments’, arXiv preprint, arXiv:2012.01971. Available at: https://arxiv.org/abs/2012.01971 (Accessed: 28 Apr. 2025)

[55] Patil, V. T. and Deore, S. S. (n.d.) ‘Deep learning-driven IoT defence: Comparative analysis of CNN and LSTM for DDoS detection and mitigation’, Unpublished manuscript

[56] Ullah, I. and Mahmoud, Q. H. (2020) ‘A hybrid model for detecting DDoS attacks in software defined networks’, *Journal of Network and Computer Applications*, 157, p. 102537. https://doi.org/10.1016/j.jnca.2020.102537

[57] Sultana, S., Chilamkurti, N. and Peng, W. (2019) ‘Detection of DDoS attacks using machine learning algorithms in cloud computing environments’, *Future Generation Computer Systems*, 100, pp. 278–285. https://doi.org/10.1016/j.future.2019.05.013

[58] scikit-learn developers (2024) *scikit-learn: Machine learning in Python*. Available at: https://scikit-learn.org/

[59] PyTorch developers (2024) *PyTorch: An open-source machine learning framework*. Available at: https://pytorch.org/

[60] TensorFlow developers (2024) *TensorFlow: An end-to-end open-source machine learning platform*. Available at: https://www.tensorflow.org/

[61] pandas developers (2024) *pandas: Python data analysis library*. Available at: https://pandas.pydata.org/

[62] NumPy developers (2024) *NumPy*. Available at: https://numpy.org/

[63] Matplotlib developers (2024) *Matplotlib: Visualization with Python*. Available at: https://matplotlib.org/

[64] Seaborn developers (2024) *Seaborn: Statistical data visualization*. Available at: https://seaborn.pydata.org/

[65] Joblib developers (2024) *Joblib documentation*. Tcpdump2018Available at: https://joblib.readthedocs.io/

[66] Optuna developers (2024) *Optuna: Hyperparameter optimization framework*. Available at: https://optuna.org/

[67] Ahmed, F., Fatima, A., Mamoon, M. and Khan, S. (2024) ‘Identification of the diabetic retinopathy using ResNet-18’, in *Proceedings of ICCR 2024*, Dubai, UAE, February 2024. IEEE. https://doi.org/10.1109/ICCR61006.2024.10532925

[68] Cloudflare (n.d.) ‘What Is the Low Orbit Ion Cannon (LOIC)?’. Cloudflare. Available at: https://www.cloudflare.com/en-gb/learning/ddos/ddos-attack-tools/low-orbit-ion-cannon-loic/ (Accessed: 17 April 2025)

[69] Kali Linux (2024) ‘Slowhttptest — Kali Linux Tools’. Slowhttptest — Kali Linux Tools. Available at: https://www.kali.org/tools/slowhttptest/#tool-documentation (Accessed: 17 April 2025)

[70] The Tcpdump Group (2018) ‘Manpage of TCPDUMP’. Tcpdump.org. Available at: https://www.tcpdump.org/manpages/tcpdump.1.html (Accessed: 17 April 2025)

[71] Cloudsmith, A. (2022) *Guide to Mastering Ansible*. Independently published, p. 57.