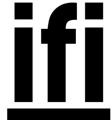
Feature Integration for an Open-source Decentralized Federated Learning Platform

Timothy-Till Näscher, Witold Rozek
Zurich, Switzerland
Student IDs: 18-935-338, 19-751-312

Supervisor: Chao Feng, Dr. Alberto Huertas Celdran
Date of Submission: January 20, 2025

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland



MASTER'S PROJECT – Communication Systems Group, Prof. Dr. Burkhard Stiller

Master's Project
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Declaration of Independence

I hereby declare that I have composed this work independently and without the use of any aids other than those declared (including generative AI such as ChatGPT). I am aware that I take full responsibility for the scientific character of the submitted text myself, even if AI aids were used and declared (after written confirmation by the supervising professor). All passages taken verbatim or in sense from published or unpublished writings are identified as such. The work has not yet been submitted in the same or similar form or in excerpts as part of another examination.

Zürich, 20.01.2025

Signature of student

Abstract

Federated Learning is a novel approach to Machine Learning, leveraging the multitude of available edge devices while at the same time offering a way to deal with the distributed datasets available on such devices. Federated Learning also offers some privacy, as updates are only shared in the form of parameters and gradients. While this approach seems promising, it does not come without its own set of challenges. Nebula is a container-based platform for simulating such Federated Learning scenarios, with a particular focus on decentralized federated learning scenarios. In this project, various extensions have been added to the Nebula platform, including various node selection strategies, data/update manipulation poisoning attacks, update aggregation mechanisms, as well as shadow model and metric based membership inference attacks. The works of this thesis highlight the importance of deploying robust systems, capable of withstanding the impact of malicious clients through the use of various defense mechanisms.

Acknowledgments

We would like to thank our supervisors Dr. Alberto Huertas Celdran and especially Chao Feng for his continual support and expert guidance throughout the duration of this thesis. We are equally appreciative of Prof. Dr. Burkhard Stiller for allowing us to complete our Masters Project in the Communication Systems Group.

Contents

Declaration of Independence **i**

Abstract **iii**

Acknowledgments **v**

1 Introduction **1**

1.1 Motivation 1

1.2 Description of Work 2

1.3 Thesis Outline 2

2 Background **5**

2.1 Federated Learning 5

2.1.1 Decentralized Federated Learning 7

2.2 Nebula 8

3 Related Work **13**

3.1 Node Selection Strategy 13

3.2 Poisoning Attacks 14

3.3 Moving Target Defense 14

3.4 Privacy Auditing Component 15

4 Architecture **17**

4.1 Docker . . . 17

4.1.1 Frontend Configuration . . . 17

4.1.2 Backend Process Flow . . . 18

4.2 Dataset Management . . . 18

4.3 Attack Implementation . . . 19

4.4 Training and Aggregation . . . 19

4.5 Node Selection . . . 19

4.6 Monitoring and Evaluation . . . 20

5 Implementation **21**

5.1 Node Selection Strategy . . . 21

5.1.1 Feature Extraction . . . 21

5.1.2 Messaging . . . 23

5.1.3 Algorithms . . . 26

5.1.4 Virtual Constraints . . . 27

5.1.5 Frontend . . . 28

5.2 Poisoning Attacks . . . 28

5.2.1 Attacks . . . 28

5.2.2 Aggregation Rules . . . 36

5.2.3 Frontend . . . 38

5.3 Moving Target Defense . . . 39

5.3.1 Frontend . . . 42

5.4 Privacy Auditing Component . . . 43

5.4.1 Frontend . . . 50

6 Evaluation **53**

6.1 Node Selection Strategy 53
6.2 Poisoning Attacks 58
6.3 Moving Target Defense 69
6.4 Privacy Auditing Component 70
6.5 Usability 72

7 Summary and Conclusions **75**

7.1 Summary 75
7.1.1 Key Insights 76
7.2 Conclusion 77
7.2.1 Key Takeaways 77
7.2.2 Future Directions 77

Bibliography **79**

Abbreviations **81**

List of Figures **81**

List of Tables **84**

List of Listings **85**

A Model Summaries **89**

B Additional Resources **91**

C Usability Evaluation Questions and Answers **115**

Chapter 1

Introduction

In recent years, the rise of Machine Learning (ML) has significantly transformed numerous industries, driving advancements in technologies that were once deemed far out of reach. However, as ML continues to evolve, so do the challenges it faces, particularly those related to data privacy and security. In the current era of Big Data, leveraging the computational potential of millions, or billions even, of edge-devices, together with their distributed datasets has become a critical focus for research as well as industry. This project expands on concerns of Federated Learning (FL) by implementing new capabilities into the Nebula framework designed to address these challenges, with a particular emphasis on Decentralized Federated Learning. By building upon existing platforms and introducing new features, this work seeks to advance the field of FL through practical implementation and evaluation.

1.1 Motivation

Machine Learning is growing year by year as everyday technologies and new innovations are introduced. But as ML grows, so do the large amounts of data, with its privacy and security challenges. So, how can security be maintained while utilizing all of the data? In 2016, Google introduced Federated Learning (FL) to allow ML models to train their data in a distributed way and keep their privacy [1]. The most widely adopted method in FL is Centralized Federated Learning (CFL). In CFL, a central server serves as the aggregator to merge the participants' models into one global model. Having only one global model leads to various challenges with this approach, such as communication bottlenecks or a single point of failure. To address these issues, Decentralized Federated Learning (DFL) was introduced. In DFL, there is no central server and the participants can communicate directly to aggregate their model updates. [1]

Nebula, a platform for DFL, plays the center role in this thesis, serving as the base for implementing and evaluating new features. The platform Nebula, developed by [2] in May 2024 as the successor of their platform Fedstellar, is an open-source DFL platform

that allows users to generate and simulate DFL scenarios. It offers a modular architecture, a user-friendly interface and robust communication protocols, essential for privacy-preserving ML models. [3] In the past, many projects/theses from the Communication Systems Group (CSG) used the Fedstellar platform to perform research in the field of DFL by introducing new features. Four specific projects were selected to be implemented in this project: implementation of a node selection strategy [4], poisoning attack behavior detection [5], mitigating poisoning attacks through moving target defense [6], and implementation of a privacy auditing component [7]. All these projects implemented new features and were also introduced to the Fedstellar platform, yet they could not be integrated into the latest version nor its successor Nebula, due to the rapid evolution of the platform and the different versions used for the projects. The architecture of Nebula, in comparison to Fedstellar, changed a lot, and thus, integrating the various features poses a challenge. The goal of this Master Project is to redesign these features, reimplement them and finally merge them into the newest version of Nebula.

1.2 Description of Work

This Master Project is divided into multiple stages. In the first stage, the foundations of the technologies and concepts involved in the project must be reviewed to gain knowledge and information for later design decisions. Namely, the foundations, architecture, and basic concepts of FL, DFL, and the Nebula platform must be understood, as well as the related work relevant to the project.

In the second stage of this project, it is essential to choose which features will be considered and to which extent they will be implemented in the current Nebula platform. Potential problems and the exact scope and implementation effort need to be considered.

This leads to the third stage, redefining Nebula's architecture and deciding on where the required parts of the new features will need to be implemented.

The fourth stage is to implement the chosen features from the defined scope into the Nebula platform. In this stage, the integration must happen bug-free to allow the platform to work correctly after the integration. Additionally, detailed documentation has to be provided to help understand all the steps. After the implementation, several evaluations must be provided in the last stage to ensure that all modules and their interconnections work correctly.

1.3 Thesis Outline

In chapter 2, the background for this project is presented. The background covers the introduction to FL, DFL and the Nebula platform. Chapter 3 summarizes the related work, where all four relevant theses for this project are outlined. Chapter 4 gives an overview of the Nebula architecture as well as the specific parts where the new implementations are done. Chapter 5 shows the implementation of all new features in Nebula. In the sixth

chapter, these implementations are evaluated. In the last chapter, the work is summarized and concluded based on the evaluations.

Chapter 2

Background

In this chapter, some of the main concepts required for this project are introduced. FL is a new training paradigm that stands in contrast to traditional Machine Learning due to its distributed nature.

As such, Federated Learning is introduced in both its centralized (CFL) and decentralized (DFL) formats, with a broad overview of their functionality, as well as the processes that happens during their application. In the case of DFL, network topology also plays an important role, which will subsequently be elaborated on.

The chapter concludes with an overview of Nebula, the successor of Fedstellar, a platform for FL simulations.

2.1 Federated Learning

Federated Learning (FL) is a Machine Learning (ML) technique that trains a shared model without the need to propagate training data over the network. This is accomplished by using nodes to train the model on local data and then distributing only the updated model parameters. In 2016, [1] introduced FL to address difficulties in traditional centralized ML, including data protection rules and privacy. By allowing data to remain locally on the node's side and sharing only the model updates, FL supports distributed training across several devices or users while protecting sensitive information. [8]

FL aims to enable shared model training while preserving data privacy and reducing node communication. Traditional ML approaches require data aggregation at a central location, which can lead to data privacy breaches and high communication costs. By permitting each node to train a local model on its own data and share only model parameters with an aggregator, a node which is chosen to consolidate these updates into a global model, FL mitigates these issues. Moreover, this decentralized technique allows the usage of much bigger datasets, distributed across the various nodes. [9], [2]

A typical CFL environment consists of four important entities that work together to enable a secure and distributed model training: the central server, nodes (or ”parties”), the communication framework, and the aggregation algorithm. [10]

- Central Server: The central server coordinates the learning process and aggregates the model updates from the nodes to form the global model.
- Nodes: Nodes are devices that store local data and participate in training the model.
- Communication Framework: This architecture connects the central server and the nodes.
- Aggregation Algorithm: This algorithm integrates the local model updates from nodes to build a global model refined with each aggregation cycle.

By working together, these four entities interact and communicate together and form a typical, iterative CFL process, which can be described in five steps [10]:

1. Model Initialization: The global model is initialized by the central server and is sent to the selected nodes.
2. Local Training: The received model is trained using local data from the nodes and the model updates are sent back to the server.
3. Aggregation: In this step, the central server aggregates all received model updates using an aggregation algorithm to improve the global model.
4. Model Redistribution: After all node updates have been updated in the global model, they are sent back to the nodes to repeat this process. This step is repeated until a desired performance is achieved.

Using the distributed local data from the nodes, this iterative training method helps to constantly develop the global model without requiring storage on a central server. An illustration of such a CFL training process is shown in Figure 2.1.

CFL is an effective and relatively easy method but still has some weaknesses. One of the weaknesses is the bottleneck the central server can cause when many nodes in a large-scale environment transmit their model updates simultaneously. Another problem is that this method can lead to security problems, as the central server might be compromised by attackers or system failure, leading to data leaks. These problems can be critical in highly sensitive scenarios like healthcare and national security, where system availability and data protection are essential. [2]

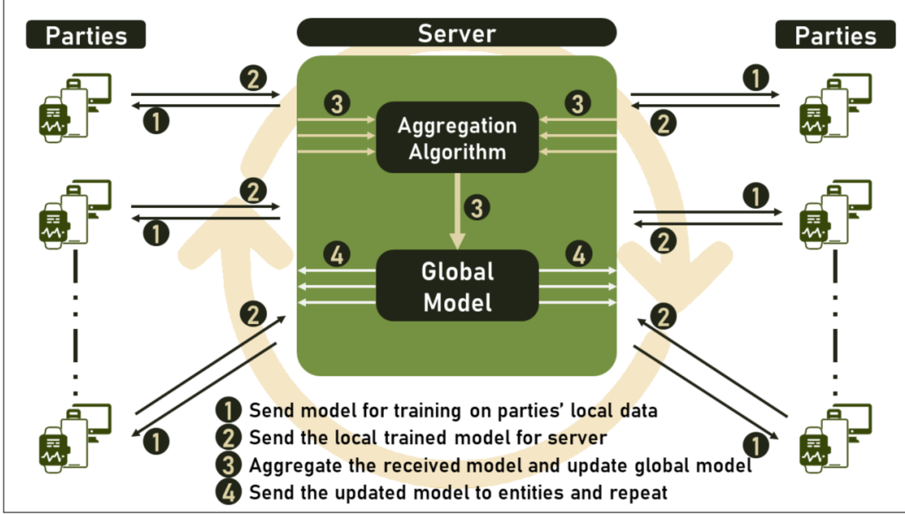


Figure 2.1: Centralized Federated Learning Process [10]

2.1.1 Decentralized Federated Learning

DFL is the second primary FL technique. In contrast to CFL, DFL aggregates the model updates without requiring a central server. Instead, it is based on a peer-to-peer network in which all the nodes directly communicate and exchange model updates with each other. The DFL technique has been developed to address CFL's limitations, which are mentioned in Section 2.1.

As DFL does not use a central server, the learning process is different than the one from CFL and looks like the following:

1. Local Model Training: Every node trains the model on local data and updates its parameters.
2. Parameter Exchange: Each node exchanges the updated model parameters with neighboring nodes.
3. Local Aggregation: After receiving the updated model parameters, every node aggregates all updates and creates a local version.
4. Parameter Exchange: All the previous steps are repeated until a desired model performance is achieved.

In DFL, as no central server manages the coordination of the process, a node can have one or more roles that define the task of the DFL process. In total, there are four roles a node can have: trainer, aggregator, proxy, and idle. The task of a trainer is to use their own dataset to train the local model and send the parameters to aggregators. After the aggregation, the trainer receives the parameters and updates his local model. The task of an aggregator is to get parameters from neighboring nodes, aggregate them, and send them back to the nodes. Sometimes, a trainer node cannot reach an aggregator directly due to a complex network topology (discussed in the next paragraph). For this situation,

a proxy node is needed. A proxy node forwards the parameters to the aggregator node if the aggregator is not directly connected with the trainer. The last role for a node is to be idle. An idle node does not send any parameters or participate in the training process. [2]

Network topology is the structure in which the nodes communicate and are organized. In DFL, the network topology is essential as it impacts the model's performance, robustness, and efficiency. Three different network topologies can be distinguished in DFL: fully connected networks, partially connected networks, and node clustering networks.[2]
The fully connected network has all nodes linked together. As each node can reach all others, the communication cost and management of connections are very high and increase quadratically. However, it is the most robust and reliable topology, and even if a few nodes fail, this network stays functional.
The partially connected network has two typical structures: the ring and the star structure. A ring-structured network shows the nodes connected in a ring, so each node has two neighbors. With that, there is only a linear increase in communication costs. In a ring structure, it can be distinguished between a bidirectional network, where a node sends its parameters for update to both neighbors, and a unidirectional network, where it sends the parameters to only one neighbor. A bidirectional network outperforms a unidirectional network in reliability and fault tolerance. A star structure can be compared to a CFL setup, where one central node is like the central server and is connected with all other nodes. However, as it is similar to a CFL setup, it has the same weaknesses as in CFL.
In node clustering networks, nodes are grouped into hierarchical clusters based on, for example, similarities. So, nodes with similar local model parameters are grouped into clusters, which leads to a more stable performance. These connected clusters can be in different topologies connected with a single node. As there is one linking point between these clusters, it comes with the weakness of a single-point failure or bottleneck problem. In Figure 2.2, all the presented network topologies are visualized. [2] [5]

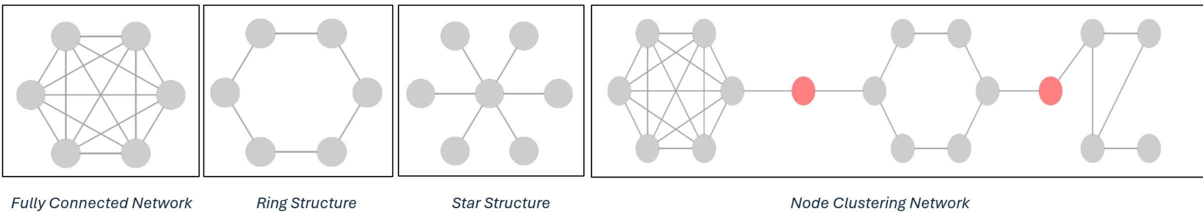


Figure 2.2: Network Topologies Overview [5]

2.2 Nebula

Nebula is a platform for FL that allows users to create and run different CFL, DFL, and semi-DFL scenarios. Enrique Tomás Martínez Beltrán launched Nebula in May 2024 as the successor of Fedstellar, the earlier version of this platform. In collaboration with Armasuisse and the universities of Zurich and Murcia, Nebula is now presented as an open-source project. It aims to help users build and analyze FL applications for virtual and physical devices. Nebula's architecture has three main parts: Frontend, Controller,

and Core. The front end provides a user-friendly interface to set up and run different FL scenarios. As an operator, the controller manages and ensures efficient operations on the platform. The core is the heart of Nebula and handles the whole FL process on each device. [3]

Nebula has many features to make the FL scenarios more secure and efficient. Besides having the features a DFL setup has, like operating without a central server and maintaining data only locally, the platform allows one to choose one of the many aforementioned topologies for the scenario. Moreover, Nebula is also compatible with many of the traditional ML libraries. With its efficient and secure communication between devices and its trustworthiness, where the completeness of the learning process is ensured, the platform is attractive for projects that seek security. Features like integrated blockchain and real-time monitoring of the running scenarios complete Nebula's robust ecosystem. These features enable Nebula to be used for different use case applications like the healthcare sector, where medical devices could be used to train models, or the military to enhance the armed equipment. [3]

The Nebula interface allows the configuration of various different settings, such as:

- Metadata: Name and description
- Federation Type: DFL, CFL or semi-DFL.
- Topology: Custom topology or a predefined topology as shown in Section 2.1.1
- Dataset Type: MNIST, CIFAR10, Custom Dataset
- Dataset Partition Method: How the dataset will be distributed among the nodes
- NN Type: MLP or CNN
- Aggregation Type: FedAvg, Krum, TrimmedMean, Median or BlockchainReputation

Additionally, the scenario is visualized on the right side of the screen, where the topology and the single nodes are shown. It is possible to move the nodes, change their role, and choose which nodes should be malicious. Figure 2.3 shows a ring topology with six nodes, one being a trainer, one being malicious and the rest being aggregators.

Figure 2.4 shows the advanced settings that appear further down after clicking on the advanced mode. These settings include participants settings where it is possible to view every participant and its detailed information. Next, in advanced deployment, the CPU or GPU can be chosen as the accelerator of the scenario. Then, it is possible to define a distance between the participants, which simulates some delays between them, as well as the number of epochs during training. The robustness setting allows defining an attack type, like label flipping or poisoning model, that will be configured during the simulation. As a defense, it is possible to decide whether the reputation system will be enabled or disabled. As a last setting, in mobility, the default location of the participant and

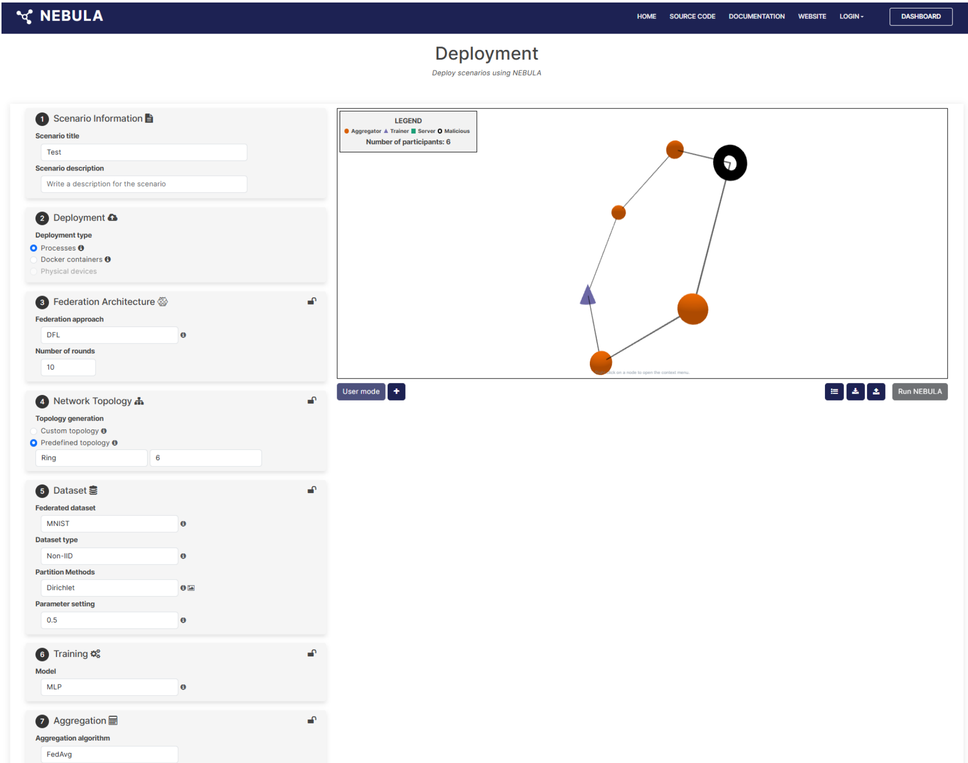


Figure 2.3: User Mode setting in Nebula Scenario Deployment

the mobility configuration can be chosen: either the participants don't move during the simulation, or they can move around geographically and/or in the topology.[3]

In conclusion, Nebula allows scenario simulations to be set up that cover a lot of use cases and are helpful for many applications.

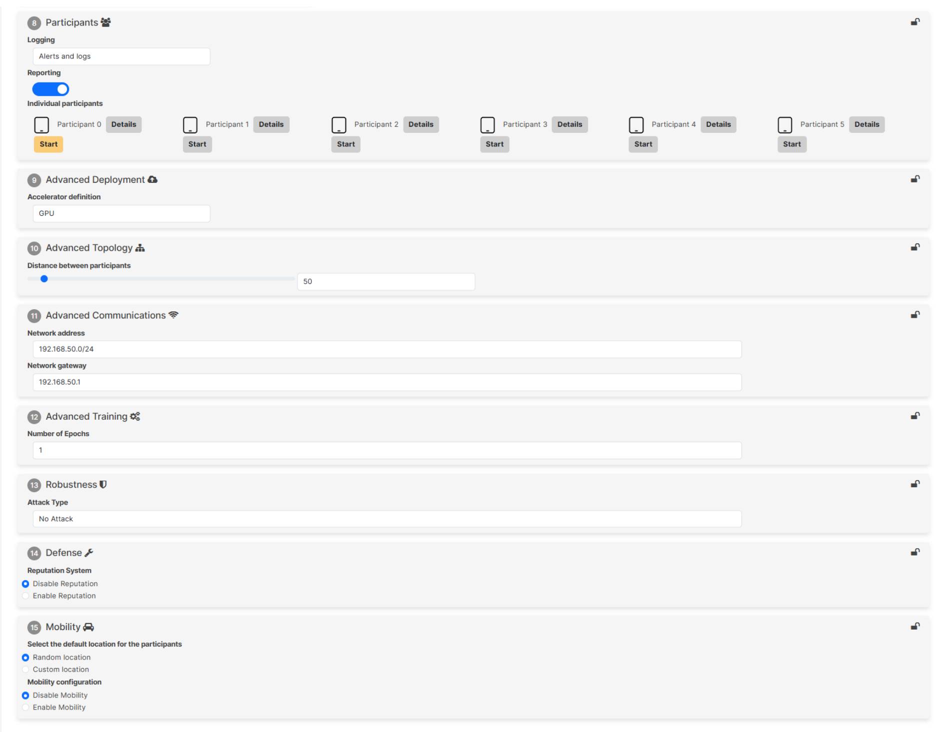


Figure 2.4: User Mode setting in Nebula Scenario Deployment

Chapter 3

Related Work

This part presents all four related works, which are the basis for the implementations in this work.

3.1 Node Selection Strategy

The node selection strategy used in this work was implemented according to thesis [4], ”Design and Prototypical Implementation of the Node Selection Strategy in Federated Learning”. This thesis focuses on optimizing the selection of participating nodes during the training in DFL and CFL environments. The chosen nodes in each training round significantly influence the overall performance of the global model. While traditional node selection methods, such as random or default selection, do not consider key factors like computational power, latency and node reliability, this work introduces a new priority selection method to address these issues.

The priority selection algorithm is designed to evaluate each node’s characteristics in real time. It evaluates parameters such as computational power, latency, data traffic size, loss metrics, the volume of data, node age, and availability, generating a comprehensive score for each node. Using a probabilistic selection process to improve model convergence, this score ensures all nodes can participate while still favoring those with higher capabilities.[4]

This strategy is especially beneficial in DFL environments, where the absence of a central server complicates the coordination of nodes. Unlike in CFL, where a central server handles the model updates, DFL requires nodes to directly communicate and aggregate model updates. The priority selection algorithm supports balancing the computational load and improving fault tolerance by integrating node performance metrics, specifically in large, heterogeneous networks with different node capacities.[4]

After integrating and implementing the new algorithm into the Fedstellar platform and running several scenarios, the evaluations show that it outperforms traditional random or default selection in several key areas. It enhances system stability in both CFL and DFL as it optimizes the use of available resources, speeds up model convergence, and ensures a

more balanced workload distribution. Moreover, it enhances the robustness and scalability of FL systems by addressing the challenges of heterogeneity and decentralization.[4]

In conclusion, thesis [4] contributes to FL by offering a dynamic and efficient node selection strategy. Integrating the new algorithm into the FedStellar platform provided a valuable tool for future research to improve the performance and scalability of CFL and DFL systems.[4]

3.2 Poisoning Attacks

In this thesis, the FedStellar framework was expanded by adding functionality to simulate various poisoning attacks and defense methods (aggregation rules). The implementation supports both centralized (CFL) and decentralized (DFL) setups. [5]

The thesis implements two types of attacks: Data Manipulation and Update Manipulation. Data Manipulation attacks change the dataset before the malicious node trains its local model on it. Update manipulation attacks change the updates that the node sends to the aggregator (CFL) or to his neighbors (DFL), the training data is not necessarily manipulated. [5] Two targeted label-flipping attacks were implemented as representatives of the data manipulation attacks. The attacks change the labels of the local dataset to another. They do this either unspecifically (a new label is chosen randomly) or specifically (given by the setup, for example, change 4, 5 to 7). The update manipulation part implements the attack from [11].

The defense methods (aggregation rules) implemented in this thesis are Krum [12] and Bulyan [13].

3.3 Moving Target Defense

While the decentralized nature of FL offers advantages regarding privacy and scalability, it also introduces vulnerabilities, especially to poisoning attacks. In thesis [6], "Mitigating Poisoning Attacks in Decentralized Federated Learning through Moving Target Defense," a Moving Target Defense (MTD) strategy is proposed to mitigate these weaknesses. Poisoning attacks can occur when malicious participants submit fake model updates, compromising the integrity and accuracy of the global model. By introducing a Dynamic Aggregation Function (DAF) within an MTD framework, this research focuses on improving the security of DFL systems.

The main contribution of this thesis is the design and implementation of Dynamic Aggregation Functions that allow proactively or reactively switching between aggregation methods in response to a possibly detected anomaly. This approach aims to continuously change the attack surface, making it difficult for adversaries to predict the system's behavior. This method allows to dynamically switch between the aggregation functions

such as FedAvg, Krum, Median, and TrimmedMean in a randomized way, creating unpredictability, which is supposed to assist in mitigating poisoning attacks.[6]

The implementation was done within the FL platform FedStellar. For the evaluation, standard benchmark datasets and various poisoning scenarios, including model and data poisoning, were used. This dynamic aggregation method was evaluated against reactive and proactive MTD strategies. The results show the high effectiveness of a proactive MTD strategy for low-level poisoning scenarios, reducing the impact of poisoned updates on the global model. However, the efficacy of dynamic aggregation decreased in scenarios with a high-level poisoning rate, suggesting that more defense methods might be required in such scenarios.[6]

In conclusion, thesis [6] introduces a novel approach to enhancing the robustness of DFL systems against adversarial attacks. Using an MTD strategy with dynamic aggregation functions is a significant step in making DFL environments more secure. Future works might focus on refining the algorithm to handle more serious poisoning scenarios and exploring strategies for more complex DFL systems.[6]

3.4 Privacy Auditing Component

Thesis [7] ”Design and Implementation of a Privacy Auditing Component for the Decentralized Federated Learning Framework” analyses the effectiveness of Membership Inference Attacks (MIA) in DFL systems. MIAs pose a significant threat to privacy, allowing attackers to determine whether a specific data point was used in the training process. Using a privacy auditing component to measure the risks, this thesis focuses on the vulnerability of DFL to these MIAs.[7]

The main part of the research involved implementing binary classifier-based and metric-based MIAs to evaluate their ability to breach the privacy of DFL systems. The study reveals that DFL already offers some inherent resistance to MIAs due to the absence of a central aggregation point, which distributes the attack surface across multiple nodes. However, another finding is that different network topologies, such as fully connected star and ring structures, affect the participants' vulnerability. [7]

The privacy auditing component implemented in this thesis includes a user-friendly front end in FedStellar that allows users to select and configure different MIAs. The backend was expanded with an attack-performing module that runs the MIAs without interfering with the original training process. This separation allows the system to simulate realistic attacks in a non-disturbing way. Additionally, a logging module records and visualizes the attack outcomes, helping the users better understand how the different MIAs impact the system. [7]

Through evaluations using standard datasets like MNIST or CIFAR-10, the thesis shows that while metric-based MIAs are simpler but reasonably accurate, binary classifier-based MIAs are more effective in FL. The data distribution and network topology greatly impact the results, as ring topologies offer more robustness, and star topologies are more vulnerable due to their central structure.[7]

However, the research also revealed a significant reduction in the effectiveness of MIAs in FL environments compared to traditional machine learning due to two main factors. FL mitigates overfitting by continuously aggregating models across nodes, making it more difficult for MIAs to distinguish between in-sample and out-sample data. In addition, the decentralized structure of FL disrupts the assumptions on which MIAs are usually based, such as the ability to train shadow models that closely mimic the target model. The decentralized setup limits the attack surface, especially in topologies where no single node has enough data for accurate inference, further weakening the attack's success. [7]

In conclusion, this thesis provides valuable insights into the privacy risks of DFL and highlights the importance of network topology in determining the effectiveness of MIAs. Furthermore, the work states that while DFL reduces the success rate of many traditional MIAs, it is not immune to privacy breaches. Future works might focus on advanced MIAs that may overcome the current limitations and offer potential directions for different topology defenses. [7]

Chapter 4

Architecture

This chapter details the architecture of the Nebula platform. This includes the various modules that interact with each other for the Docker Container Frontend, as well as the application itself. Instead of code snippets, a holistic overview of the Nebula architecture is given. The chapter begins with the Docker container setup, both in frontend and backend, after which the dataset and attack modules are introduced. After an introduction to the training/aggregation and node selection modules, an overview of the available monitoring and evaluation features is given.

Figure 4.1 depicts an overview of the various Python modules and how they interact with each other.

4.1 Docker

4.1.1 Frontend Configuration

The Nebula platform's frontend is implemented as a Docker container. The user interface allows configuration of various parameters for the simulation of FL scenarios such as:

- Network topologies
- Machine learning models, training epochs, and rounds
- Simulation of attacks and defenses (optional)
- Other parameters required for a specific scenario (dataset distribution, etc.)

The configuration details are passed to an API hosted by the nebula-frontend Docker container, which dynamically generates Dockerfiles for each node and launches them as separate Docker containers. The `nebula/scenarios.py` module receives this information and populates the node Dockerfiles accordingly.

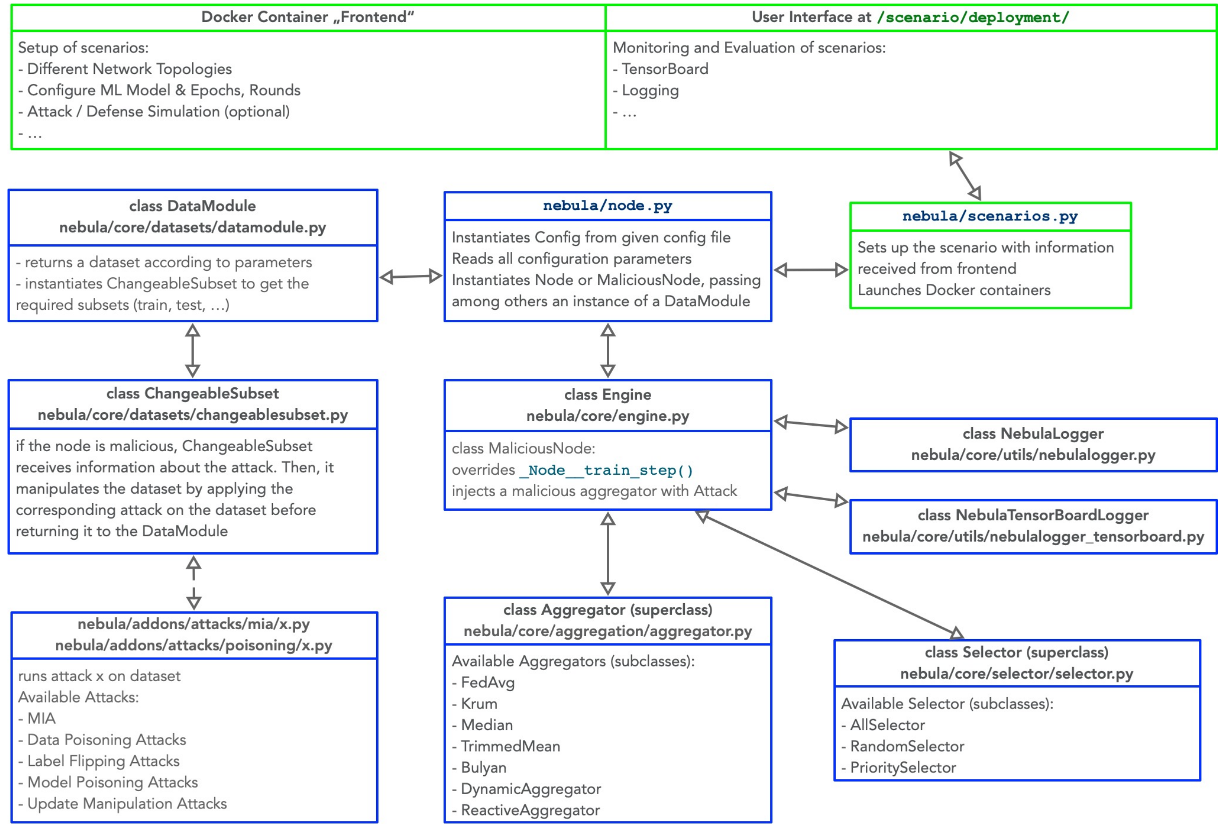


Figure 4.1: An overview of the Nebula Architecture.

4.1.2 Backend Process Flow

During node initialization, each node fetches its configuration from a JSON file specified in the Dockerfile.

The configuration is read and instantiated via `nebula/node.py`, which sets up the node or malicious node and instantiates the `DataModule`.

4.2 Dataset Management

`DataModule` (`nebula/core/datasets/datamodule.py`) handles dataset generation and management for each node.

For malicious nodes, the `ChangeableSubset` class applies specific attacks to manipulate the dataset before returning it to the `DataModule`.

4.3 Attack Implementation

The `nebula/addons/attacks` module contains implementations of various attacks, such as:

- Membership Inference Attacks (MIA)
- Data Poisoning Attacks
- Label Flipping Attacks
- Model Poisoning Attacks
- Update Manipulation Attacks

4.4 Training and Aggregation

Once the configuration is finalized, the `Engine` (`nebula/core/engine.py`) handles the training process.

Aggregation strategies are selected via the `Aggregator` superclass (`nebula/core/aggregation/aggregator.py`), which supports:

- FedAvg
- Krum
- Median
- TrimmedMean
- Bulyan
- DynamicAggregator
- ReactiveAggregator

4.5 Node Selection

The `Selector` superclass (`nebula/core/selector/selector.py`) provides node selection strategies, including:

- `AllSelector`
- `RandomSelector`
- `PrioritySelector`

4.6 Monitoring and Evaluation

Users can monitor and evaluate scenarios through the user interface, accessible at `/scenario/deployment/`. The monitoring tools include:

- TensorBoard for visualizing training progress and performance metrics.
- Logging mechanisms for scenario details and outcomes.
- Additional evaluation metrics, depending on the scenario.

The `NebulaLogger` and `NebulaTensorBoardLogger` modules handle logging and visualization during training and evaluation.

Chapter 5

Implementation

In this chapter, the development of the various components that are introduced to Nebula is documented. The first task comprises the various node selection strategies outlined in task [4]. An outline of the various features used for the `PrioritySelector` is also shown. Moving on, the second task [5] that has been implemented includes the various Data and Update Manipulation Attacks such as targeted/untargeted labelflipping, the FANG [14] labeflipping attack, as well as the LIE [11] attack, together with some more aggregation rules such as Bulyan [13]. In the third task, a moving target defense from [6] is implemented. More specifically, a dynamic aggregator is implemented. This aggregator reactively changes the aggregation function when possible anomalies are detected. In the fourth and final task, a privacy audititing component from [7] is introduced. This mainly consists of two different membership inference attacks, whose implementations are outlined in this chapter.

Code snippets are included where deemed relevant, with much of the code being replaced for brevity's sake. Instead, there are comments outlining their functionality.

5.1 Node Selection Strategy

This section describes the implementation of the "Node Selection Strategy" as proposed by [4] into Nebula.

5.1.1 Feature Extraction

As described in [4], the selection mechanisms require certain features of each node to decide which nodes to aggregate. This section describes the extraction, the messaging and the processing of those features in detail.

nebula/core/engine.py

```
def __nss_extract_features(self):
    """
    Extract the features necessary for the node selection strategy.
    """
    nss_features = {}
    nss_features["cpu_percent"] = psutil.cpu_percent()
    net_io_counters = psutil.net_io_counters()
    nss_features["bytes_sent"] = net_io_counters.bytes_sent
    nss_features["bytes_received"] = net_io_counters.bytes_recv
    nss_features["loss"] = self.trainer.model.loss
    nss_features["data_size"] = self.trainer.get_model_weight()
    self.nss_features = nss_features
```

Listing 5.1: NSS Features extraction

CPU Usage

The CPU usage feature is obtained using `psutil.cpu_percent`, a function that returns ”a float representing the current system-wide CPU utilization as a percentage” [15]

nebula/core/engine.py

```
psutil.cpu_percent()
```

Listing 5.2: NSS Features extraction (CPU)

Networking Bytes Sent / Received

The networking features (bytes_sent and bytes_received) are extracted using `psutil.net_io_counters()`, a function that returns ”system-wide network I/O statistics as a named tuple including the following attributes: bytes_sent: number of bytes sent; bytes_recv: number of bytes received” [15]

nebula/core/engine.py

```
psutil.net_io_counters().bytes_sent
psutil.net_io_counters().bytes_recv
```

Listing 5.3: NSS Features extraction (Networking)

Loss

The loss is an attribute (see listing 5.4) from the trainer (an instance of `Lightning`, defined in *nebula/core/training/lightning.py*)

nebula/core/engine.py

```
self.trainer.model.loss
```

Listing 5.4: NSS Features extraction (Loss)

Data Size

The data size is retrieved using `get_model_weight` (see listing 5.5), a function defined in the instance of the trainer (see listing 5.6).

nebula/core/engine.py

```
self.trainer.get_model_weight()
```

Listing 5.5: NSS Features extraction (Data Size)

nebula/core/training/lightning.py

```
class Lightning:
    ...
    def get_model_weight(self):
        return len(self.data.train_dataloader().dataset)
    ...
```

Listing 5.6: NSS Features extraction (Data Size)

Latency

The latency is the only metric not submitted by the neighbor, but measured by the aggregating node. When the aggregating node receives the message of the neighbor with his features, he measures the latency from himself to the source of the message using `__nss_get_latency` (see exact implementation in listing 5.7 for details). The aggregating node then adds the latency to the features list of the source node and stores it.

nebula/core/engine.py

```
def __nss_get_latency(self, source):
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    host, port = source.split(":")
    start = time.time()
    s.connect((host, int(port)))
    s.close()
    return (time.time() - start) * 1000
```

Listing 5.7: NSS Features extraction (Latency)

5.1.2 Messaging

In Nebula, nodes exchange information using *Protocol Buffers* („Protobuf“) messages. *Protocol Buffers* are an extensible mechanism for serializing structured data [16]. The features required for the Node Selection Strategy are also exchanged through Protobuf messages, specifically using the `NSSFeaturesMessage` Type. The definition of the message type can be seen in listing 5.8. When a node has features to share, it creates an `NSSFeaturesMessage` containing the required metrics such as *cpu_percent*, *bytes_sent*, *bytes_received*, *loss*, and *data_size* (see listing 5.10). It does so by using `generate_nss_features_message` as shown in listing 5.11. Note that the

latency is measured by the receiving node itself and is therefore not included in the message (see listing 5.7). The `NNSSFeaturesMessage` is wrapped inside the `Wrapper` message, which includes the source node's identifier (see listing 5.9).

nebula/core/pb/nebula.proto

```
message NSSFeaturesMessage {
  float cpu_percent = 1;
  int32 bytes_sent = 2;
  int32 bytes_received = 3;
  float loss = 4;
  int32 data_size = 5;
}
```

Listing 5.8: Protobuf Features Message

nebula/core/pb/nebula.proto

```
message Wrapper {
  string source = 1;
  oneof message {
    ...
    NSSFeaturesMessage nss_features_message = 8;
  }
}
```

Listing 5.9: Protobuf Message Wrapper

nebula/core/pb/nebula.proto

```
async def _learning_cycle(self):
    while self.round is not None and self.round < self.total_rounds:
        ... ... ...
        if self.node_selection_strategy_enabled:
            # Extract Features needed for Node Selection Strategy
            self.__nss_extract_features()
            # Broadcast Features
            logging.info(f"Broadcasting NSS features to the rest of the
   topology ...")
            message = self.cm.mm.generate_nss_features_message(self.
   nss_features)
            await self.cm.send_message_to_neighbors(message)
            ... ... ...
            selected_nodes = self.node_selection_strategy_selector.
   node_selection(self)
            self.nebulalogger.log_text("[NSS] Selected nodes", str(
   selected_nodes), step=self.round)
```

Listing 5.10: Sending and Receiving NSS Features Messages

nebula/core/pb/nebula.proto

```
def generate_nss_features_message(self, nss_features):
    message = nebula_pb2.NSSFeaturesMessage(
        cpu_percent = nss_features["cpu_percent"],
        bytes_sent = nss_features["bytes_sent"],
        bytes_received = nss_features["bytes_received"],
        loss = nss_features["loss"],
        data_size = nss_features["data_size"],
    )
    message_wrapper = nebula_pb2.Wrapper()
    message_wrapper.source = self.addr
    message_wrapper.nss_features_message.CopyFrom(message)
    data = message_wrapper.SerializeToString()
    return data
```

Listing 5.11: Generating the Protobuf Message

The wrapped message is serialized and sent over the network to his neighboring nodes asynchronously. Upon receiving features messages from neighbors, the receiving node triggers the `__nss_features_message_callback` function through the event handler (see listing 5.12 and 5.13). This callback processes the message, extracts the feature metrics, and updates its local dict with the features of his neighbors (see listing 5.14).

nebula/core/engine.py

```
@event_handler(nebula_pb2.NSSFeaturesMessage, None)
async def __nss_features_message_callback(self, source, message):
    logging.info(f"handle_nss_features_message | Trigger | Received NSS
   features message from {source}")
    if message is not None:
        latency = self.__nss_get_latency(source)
        features = {}
        features["cpu_percent"] = message.cpu_percent
        features["bytes_sent"] = message.bytes_sent
        features["bytes_received"] = message.bytes_received
        features["loss"] = message.loss
        features["data_size"] = message.data_size
        features["latency"] = latency
        self.node_selection_strategy_selector.add_neighbor(source)
        self.node_selection_strategy_selector.add_node_features(source,
   features)
```

Listing 5.12: NSS Features Message Handler

nebula/core/network/communications.py

```
    async def handle_nss_features_message(self, source, message):
        try:
            logging.error(f"handle_nss_features_message | Received
   Message from: {source}")
            await self.engine.event_manager.trigger_event(source,
   message)
        except Exception as e:
            logging.error(f"handle_nss_features_message | Error while
   processing: {message} | {e}")
```

Listing 5.13: NSS Features Message Event Handler

nebula/core/engine.py

```
1 if self.node_selection_strategy_enabled:
2     # Extract Features needed for Node Selection Strategy
3     self.__nss_extract_features()
4     # Broadcast Features
5     logging.info(f"Broadcasting NSS features to the rest of the topology
    ...")
6     message = self.cm.mm.generate_nss_features_message(self.nss_features
    )
7     await self.cm.send_message_to_neighbors(message)
8     _nss_features_msg = f"""NSS features for round {self.round}:
9     CPU Usage (%): {self.nss_features['cpu_percent']}%
10    Bytes Sent: {self.nss_features['bytes_sent']}
11    Bytes Received: {self.nss_features['bytes_received']}
12    Loss: {self.nss_features['loss']}
13    Data Size: {self.nss_features['data_size']}"""
14    print_msg_box(msg=_nss_features_msg, indent=2, title="NSS features (
    this node)")
15    selected_nodes = self.node_selection_strategy_selector.
    node_selection(self)
16    self.nebulalogger.log_text("[NSS] Selected nodes", str(
    selected_nodes), step=self.round)
```

Listing 5.14: NSS Features extraction

5.1.3 Algorithms

Selector (Base Class)

The `Selector` class serves as the superclass for the different selection strategies designed in [4]. It handles core functionalities such as maintaining a list of neighbors, tracking their features (e.g., CPU usage, bytes_sent, bytes_recv, latency, data size and loss), and providing basic methods to add neighbors, reset lists, and manage feature data. It is designed to be extended by subclasses that implement the selection strategies (`RandomSelector`, `AllSelector`, `PrioritySelector`). The `node_selection` method is intended to be overridden by these subclasses, allowing them to define the custom logic for selecting the nodes for aggregation. The implementation of the `Selector` class is shown in listing B.1.

AllSelector

The `AllSelector` subclass represents the selection strategy where all available neighbors are selected for aggregation. It copies the list of neighbors, adds the current node itself, and logs the selected nodes. If no neighbors are available, it defaults to selecting only the current node. The Implementation is shown in listing B.2.

RandomSelector

The `RandomSelector` selects a random subset of neighbors. It ensures that at least one is chosen but not exceeding a predefined percentage of the total available nodes. The following implementation differs from the one described in [5]. The original implementation, given in listing B.3, would always choose the maximum possible amount of nodes. For example, in a scenario with 10 available neighbors and 100% maximum selectable neighbors, it would always choose all 10 nodes for aggregation. The implementation for Nebula randomly samples the standard distribution to decide the number of nodes being used for aggregation.

PrioritySelector

The `PrioritySelector` selects nodes for aggregation based on a weighted scoring system. Each participant's score is computed from various features such as CPU usage, data size, bytes sent/received, packet loss, latency, and node age. These features are assigned specific weights to prioritize certain aspects over others, with the default weights shown in listing B.5. Nodes are selected randomly (with the weighting applied) according to the calculated scores, with a minimum and maximum number of neighbors ensured. The aggregating node adds itself to the selection made in any case. The full implementation is shown in listing B.5.

5.1.4 Virtual Constraints

The following implementation details outline how resource constraints are applied to simulate varying node performance in the Dockerized environment where Nebula runs its scenarios. The constraints concern two aspects: CPU availability (through allocation) and network latency. It should allow realistic simulation of nodes with different computational capabilities and network conditions. In Nebula, every scenario is represented by a `docker-compose.yml` file that contains the definition of each participant. An example of such a file is given in listing B.7. This example scenario has two participants, with participant0 being limited to 0.3 CPUs and an additional 50ms delay to all networking operations.

CPU

Each node's CPU allocation is controlled via the deploy.resources.limits.cpus attribute. The attribute "configures a limit or reservation for how much of the available CPU resources, as number of cores, a container can use." [17] As seen in the implementation shown in listing 5.15, if no CPU constraint is configured for the scenario the maximum number of available CPU's is used as the limit. Using `os.cpu_count()` (a function that returns "the number of logical CPUs in the system" [18]), the maximum value is retrieved and later embedded into the docker-compose file. The example in listing B.7 shows the CPU constraints applied on line 15 and 39.

nebula/scenarios.py

```python
if node["resource_args"]["resource_constraint_cpu"] == 0:
    # If 0, the node shall have no CPU constraints
    resource_constraint_cpu = os.cpu_count()
    logging.info("Node has no Resource Constraint on CPU")
```

```
5 else:
6     resource_constraint_cpu = node["resource_args"]["
    resource_constraint_cpu"]
7     logging.info(f"Node has the following Resource Constraint on CPU :{
    resource_constraint_cpu}")
```

Listing 5.15: NSS Resource Constraints Setup (CPU)

Networking (Latency)

Network conditions are manipulated using `tcset`, „a command to add a traffic control rule to a network interface“ [19]. As seen in the implementation shown in listing 5.16, the command `tcset eth0 --delay <latency>` is used to add a specified delay to the network interface of the node (*eth0*), simulating a scenario where the affected node has higher latency then the others. The example in listing B.7 shows the networking constraints applied on line 20, line 44 shows the default without constraints.

nebula/scenarios.py

```
1 tcset_cmd = ""
2 if node["resource_args"]["resource_constraint_latency"] != 0:
3     tcset_cmd = f"tcset eth0 --delay {node['resource_args']['
    resource_constraint_latency']} && "
```

Listing 5.16: NSS Resource Constraints Setup (Network)

5.1.5 Frontend

The scenario setup dashboard was extended by a new section ”Node Selection Strategy” shown in Figure 5.1. The virtual constraints were implemented in the section, where the details of the scenario participants are set up (shown in Figure 5.2). After clicking on the ”Details”-Button

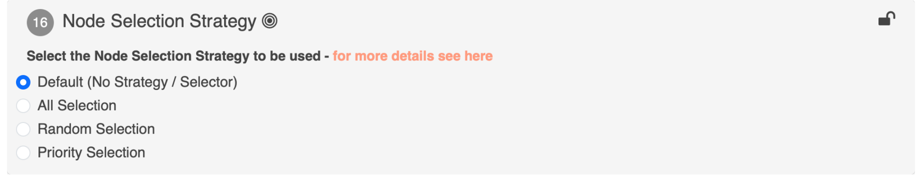


Figure 5.1: Choose the Node Selection Strategy

next to each participant, the constraints can be set for each participant individually (shown in figure 5.3).

5.2 Poisoning Attacks

5.2.1 Attacks

This section describes the implementation of attacks and aggregations rules (described in [5]) into Nebula.

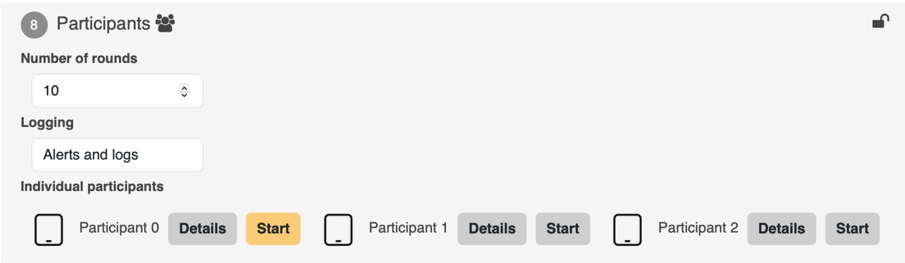


Figure 5.2: Scenario Participants

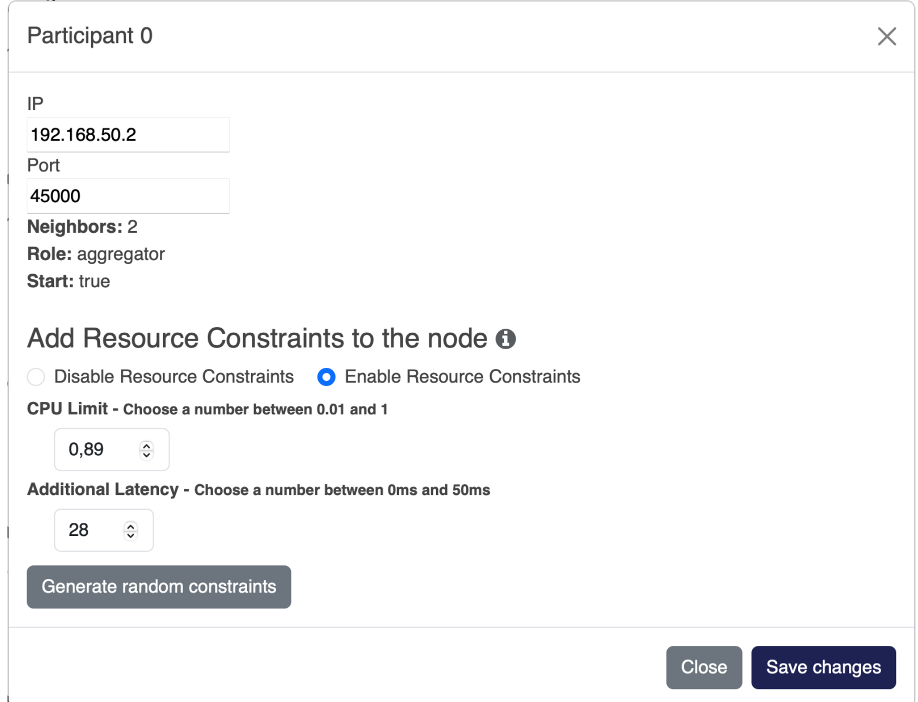


Figure 5.3: Add Resource Constraints to Participant

Data Manipulation Attacks

Data manipulation attacks manipulate the training data used by the malicious node. The behaviour of the node itself is not different to a benign node per se, only the training data is manipulated. To implement this behaviour in Nebula, the attacks are performed before the training process launches. When the nodes are being initialized (see `nebula/node.py` for details), each node instantiates a dataset (instance of `DataModule`) that is initalized with different parameters. They include the information about the dataset itself (splitting into train, test and validation set, ...) and also information about the attack (for example the specific classes targeted by a label flipping attack, ...). The `DataModule` instance then creates the requested sets (training, test, validation) as an instance of `ChangeableSubset` which applies the attacks if needed (it checks whether the configuration contains an attack, and if yes, calls the functions defining the attack on the dataset). Listing B.10 shows the process of initializing the node, listing B.8 and B.9 show the creation of the (malicious) dataset through `DataModule` and `ChangeableSubset`.

Labelflipping (from [14])

The first data manipulation attack implemented in this thesis is the Label flipping attack proposed in [14]. The attack flips "a label `l` as `L-l-1`, where `L` is the number of classes in the classification problem and `l = 0, 1, ⋯, L-1`." [14]

nebula/addons/attacks/poisoning/labelflipping_fang.py

```python
import copy
import logging
import torch

def labelflipping_fang(dataset):
    logging.info("[Attack Labelflipping_fang] running attack on dataset")

    new_dataset = copy.copy(dataset)
    targets = new_dataset.targets.detach().clone()
    class_list = new_dataset.class_to_idx.values()

    for i in range(len(targets.tolist())):
        t = targets[i].numpy()
        targets[i] = torch.tensor(len(class_list) - t - 1)

    new_dataset.targets = targets
    return new_dataset
```

Listing 5.17: Labelflipping Attack (Fang)

Labelflipping targeted

The attacks in this section are targeted labelflipping attacks, meaning the attack targets a specific class. In other words, only the training samples of the dataset that define a specific class are manipulated. The manipulation itself is done in either a specific or a unspecific manner. For the specific attack, the scenario explicitly defines which classes the attacked classes are replaced with. The unspecific attack decides the new class randomly. The implementation of these attacks are shown in 5.18 resp. 5.19.

nebula/addons/attacks/poisoning/labelflipping_targeted.py

```python
def labelflipping_targeted_specific(dataset, indices, label_og: Union[list, int], label_goal: int):
    logging.info("[Attack Labelflipping_targeted_specific] running attack on dataset")
    logging.info(f"received: label_og{label_og}, label_goal{label_goal}")
    new_dataset = copy.copy(dataset)
    try:
        targets = new_dataset.targets.detach().clone()
    except AttributeError:
        targets = new_dataset.targets
    logging.info("[LabelFlipping Attack] Changing labels from {} to {}".format(label_og, label_goal))

    for i in indices:
        try:
            t = targets[i].numpy()
        except AttributeError:
            t = targets[i]
        if (t in label_og) or (str(t) in label_og):
            targets[i] = label_goal
```

```
18      new_dataset.targets = targets
19      return new_dataset
```

Listing 5.18: Labelflipping Attack (Targeted, Specific)

nebula/addons/attacks/poisoning/labelflipping_targeted.py

```
1  def labelflipping_targeted_unspecific(dataset, indices, label_og: Union[
       list, int]):
2      new_dataset = copy.copy(dataset)
3      targets = new_dataset.targets.detach().clone()
4      class_list = new_dataset.class_to_idx.values()
5      logging.info("[LabelFlipping Attack] Changing labels from {}
       randomly.".format(label_og))
6
7      for i in indices:
8          t = targets[i]
9          if str(t) in label_og:
10             targets[i] = torch.tensor(
11                 random.sample(sorted([x for x in class_list if x != t]),
     1)
12             )
13
14     new_dataset.targets = targets
15     return new_dataset
```

Listing 5.19: Labelflipping Attack (Targeted, Unspecific)

Labelflipping untargeted

The attack described in this section is an untargeted labelflipping attack, meaning the attack targets all classes. All (or a percentage of them) training samples of the dataset (regardless of which class they belong to) are manipulated. The implementation of this attack is shown in 5.20. *nebula/addons/attacks/poisoning/labelflipping_untargeted.py*

```
1  def labelflipping_untargeted(dataset, indices, flipping_persent):
2      logging.info("[Attack labelflipping_untargeted] running attack on
       dataset")
3      logging.info("[Attack labelflipping_untargeted] Received Config:
       flipping_percent: {}".format(flipping_persent))
4      sys.set_int_max_str_digits(0)
5      new_dataset = copy.copy(dataset)
6
7      if type(new_dataset.targets) == list:
8          new_dataset.targets = torch.tensor(new_dataset.targets)
9      targets = new_dataset.targets.detach().clone()
10     num_indices = len(indices)
11     classes = new_dataset.classes
12     class_to_idx = new_dataset.class_to_idx
13     class_list = [class_to_idx[i] for i in classes]
14     num_flipped = int(float(flipping_persent)*0.01 * num_indices)
15     if num_indices == 0:
16         return new_dataset
17     if num_flipped > num_indices:
```

```
        return new_dataset
    flipped_indice = random.sample(indices, num_flipped)

    for i in flipped_indice:
        t = targets[i]
        flipped = torch.tensor(random.sample(class_list, 1)[0])
        while t == flipped:
            flipped = torch.tensor(random.sample(class_list, 1)[0])
        targets[i] = flipped
    new_dataset.targets = targets

    return new_dataset
```

Listing 5.20: Labelflipping Attack (Fang)

Update Manpiulation Attacks

LIE

The update manipulation attack described in this section was introduced by [11]. It is different from the other attacks mentioned above in that it doesn't try to inject updates with high disturbance, but instead applying minimal changes. These changes lead to a lower effect when targeting averaging aggregation rules such as FedAvg, but allow the attack to stay unrecognized when targeting other aggregation rules such as Trimmed Mean. [5] The attack is described in more detail in algorithm 1. The actual implementation of function `Z(n, f)` is shown in listing 5.23. Note that, as the nodes in Nebula don't have access to the information required to calculate `Z`, it is therefore calculated during the setup of the scenario. This approach also allows users to manipulate the value as desired. The implementation of the attack itself is shown in listing 5.21. In Nebula, the data manipulation attacks are executed before the node itself starts his learning process. The update manipulation attacks however, are applied after the training process is done and the aggregation starts. Listing 5.22 shows how the attack is applied by intercepting the updates broadcasted for aggregation.

Algorithm 1 Update Manipulation as seen in [11], taken from [5]

U_{benign}: Benign Update
n: total nodes m: smallest m fulfilling $m + f < n$ (amount of byzantine nodes "missing" to control median)
f: total byzantine nodes

1: **function** Z(n, f)
2: $\quad m \leftarrow \lfloor \frac{n}{2} + 1 \rfloor - f$
3: $\quad z \leftarrow \max_x(\phi(x) > \frac{n-m}{n})$
4: $\quad$ **return** z
5: **function** POISONEDUPDATE(U_{benign}, z)
6: $\quad$ **for** dim **in** U_{benign} **do**
7: $\quad\quad \mu_{dim} \leftarrow \text{mean}(dim)$
8: $\quad\quad \sigma_{dim} \leftarrow \text{std}(dim)$
9: $\quad\quad P_{poisoned} \leftarrow \mu_{dim} + \sigma_{dim} \cdot z$
10: $\quad$ **return** $D_{poisoned}$
11: ▷ *All malicious nodes now send the same $D_{poisoned}$ for aggregation.* ◁

nebula/addons/attacks/poisoning/update_manipulation.py

```
import logging
import torch

def update_manipulation_LIE(parameters, z):
    logging.info("[Attack update_manipulation_LIE] running attack on model parameters")
    malicious_parameters = {}
    for key, value in parameters.items():
        if key.endswith("bias"):
            malicious_parameters[key] = value
        else:
            new_weights_list = []
            for weights in value:
                new_weights = []
                avg = torch.mean(weights, dim=0)
                std = torch.std(weights, dim=0)
                for _ in weights:
                    new_weights.append(avg + z * std)
                new_weights_list.append(new_weights)
            malicious_parameters[key] = torch.tensor(new_weights_list)
    logging.info("[Attack update_manipulation_LIE] finished")
    return malicious_parameters
```

Listing 5.21: Update Manipulation Attack (from [11])

nebula/core/engine.py

```
class AggregatorNode(Engine):
    ...
    async def _extended_learning_cycle(self):
        # Define the functionality of the aggregator node
        logging.info(f"[Testing] Starting...")
        self.trainer.test()
        logging.info(f"[Testing] Finishing...")

        logging.info(f"[Training] Starting...")
        self.trainer.train()
        logging.info(f"[Training] Finishing...")

        if self.lie_atk:
            from nebula.addons.attacks.poisoning.update_manipulation import update_manipulation_LIE
            await self.aggregator.include_model_in_buffer(update_manipulation_LIE(self.trainer.get_model_parameters(),self.lie_atk_z), self.trainer.get_model_weight(), source=self.addr, round=self.round)
        else:
            await self.aggregator.include_model_in_buffer(self.trainer.get_model_parameters(), self.trainer.get_model_weight(), source=self.addr, round=self.round)

        await self.cm.propagator.propagate("stable")
        await self._waiting_model_updates()
```

Listing 5.22: Executing Update Manipulation Attacks

nebula/frontend/app.py

```python
@app.post("/nebula/calc_lie_z")
async def calc_lie_z(request: Request):
    data = await request.json()
    total_nodes = int(data.get("total_nodes"))
    percent_malicious = int(data.get("percent_malicious"))
    malicious_nodes = math.ceil(total_nodes * (percent_malicious / 100))
    print(percent_malicious, total_nodes, malicious_nodes)
    if malicious_nodes > total_nodes:
        # If malicious_nodes > total_nodes, the median is already under
    control of the attacker, and convergence of the global model is no
    longer possible
        return "0"

    # Calculate the number of nodes needed to control the median (
    majority)
    nodes_required_for_majority = math.ceil(((total_nodes / 2) + 1) -
    malicious_nodes)

    # Calculate the z_max using the percent point function (ppf)
    # ppf = Percent point function (inverse of cdf - percentiles)
    # est_ppf() retrieved from https://stackoverflow.com/questions
    /74817976/alternative-for-scipy-stats-norm-ppf

    def est_ppf(x):
        a = -9
        b = 9
        v2 = math.sqrt(2)
        while b - a > 1e-9:
            c = (a + b) / 2
            r = 0.5 + 0.5 * math.erf(c / v2)
            if r > x:
                b = c
            else:
                a = c
        return c

    z_max = est_ppf((total_nodes - nodes_required_for_majority) /
    total_nodes)
    return str(math.floor(z_max * 100) / 100.0)
```

Listing 5.23: Calculation of Z for Update Manipulation Attack from [11]

5.2.2 Aggregation Rules

Bulyan

The aggregation rule *Bulyan* was introduced in [13]. The concept of *Bulyan* is the combination of a byzantine–resilient aggregation rule ([13] propose to use Krum) and TrimmedMean. It first uses Krum to generate a subset of clients which are (probably) benign. This subset is then aggregated using TrimmedMean. For more details on the Bulyan algorithm see algorithm 2, for the implementation in Nebula see listing 5.24.

Algorithm 2 Bulyan Algorithm (taken from [5])

n: received update vectors
f: amount of malicious clients
A: any (α, f)-Byzantine–resilient aggregation rule, e.g. Krum

1: **function** BULYAN(β, n, f)
2: $S_n \leftarrow []$
3: **while** LENGTH(S_n) $< (n - 2f)$ **do**
4: $n_{rest} \leftarrow n \setminus S_n$
5: $S_n \leftarrow S_n + \mathrm{A}(n_{rest})$
6: **return** TRIMMEDMEAN(β, S_n)

nebula/core/aggregation/bulyan.py

```
import logging

import torch
import numpy as np
from nebula.core.aggregation.aggregator import Aggregator
from nebula.core.aggregation.trimmedmean import TrimmedMean


class Bulyan(Aggregator):
    def __init__(self, config=None, **kwargs):
        super().__init__(config, **kwargs)
        self.config = config
        self.role = self.config.participant["device_args"]["role"]
        self.KRUM_SELECTION_SET_LEN = 4
        self.TRM_BETA = 1


    def run_aggregation(self, models):
        if len(models) == 0:
            logging.error("[Bulyan] Trying to aggregate models when
    there is no models")
            return None

        # Krum Step of Bulyan:
        # The implementation of the Krum Function is copied from krum.py
     [Author: Chao Feng].
        # This implementation was then modified to return a list of
    models ordered by their distance
        # instead of the single update with the best score to make it
    suitable for use in the Bulyan AGR

        models = list(models.values())

        # initialize zeroed model
        accum = (models[-1][0]).copy()
        for layer in accum:
            accum[layer] = torch.zeros_like(accum[layer])

        logging.info(
            "[Bulyan(Krum Step).aggregate] Aggregating models: num={}".
    format(
```

```
                len(models)
            )
        )

        # Create model distance list
        total_models = len(models)
        distance_list = [0 for i in range(0, total_models)]
        models_and_distances = []

        # Calculate the L2 Norm between xi and xj
        for i in range(0, total_models):
            m1, _ = models[i]
            for j in range(0, total_models):
                m2, _ = models[j]
                distance = 0
                if i == j:
                    distance = 0
                else:
                    for layer in m1:
                        l1 = m1[layer]
                        l2 = m2[layer]
                        distance += np.linalg.norm(l1 - l2)
                distance_list[i] += distance

            # Add the model and its distance to the dictionary containing all models and their distances
            models_and_distances.append((distance_list[i], models[i]))

        # Order the models by distance ascending -> potentially malicious models are at the end of the list
        models_and_distances.sort(key = lambda tup: tup[0])

        # remove the potentially malicious models
        if len(models_and_distances) <= self.KRUM_SELECTION_SET_LEN:
            logging.error(
                "[Bulyan(TRMstep)] Trying to aggregate models when there are less or equal models than the set length of the krum selection set ..."
            )
            return None
        else:
            for i in range(self.KRUM_SELECTION_SET_LEN):
                models_and_distances.pop()
        # calculate new global model using trimmedmean
        models = [x[1] for x in models_and_distances]
        TRM = TrimmedMean(config = self.config, beta = self.TRM_BETA)
        return TRM.run_aggregation(models)
```

Listing 5.24: Bulyan Aggregation Rule

5.2.3 Frontend

The configuration of the attacks was integrated into the already existing configuration section (*13: Robustness*). Figure 5.4 shows the attacks that are available in Nebula now. After selecting

Figure 5.4: Frontend Attack Setup

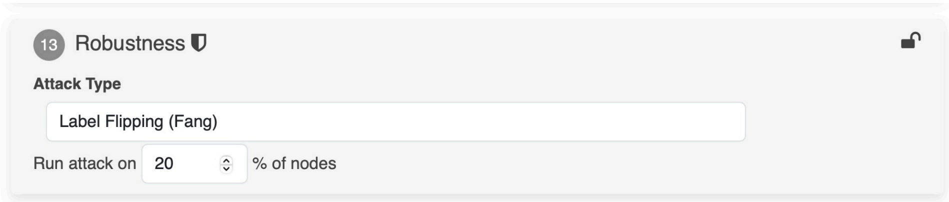


Figure 5.5: Label Flipping Attack (from [14])

the attack the necessary configuration fields are displayed. Figure 5.5 shows the configuration fields for the label flipping attack from [14], figure 5.7, 5.8 and 5.9 the other label flipping attacks and figure 5.6 shows the update manipulation attack from [11]. The implemented aggregation rule, *Bulyan*, doesn't need any further configuration and is simply selected (see figure 5.10). The only difference to the other aggregation rules is that, as Bulyan requires 5 nodes to work properly [5], a message is displayed when the user tries to create a scenario with less nodes using Bulyan as the aggregation rule.

5.3 Moving Target Defense

This section describes the implementation of the moving target defense strategies (MTD) as proposed by [6] into Nebula.

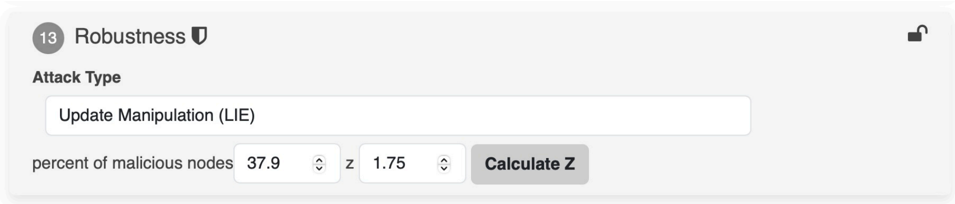


Figure 5.6: Update Manipulation Attack (from [11])

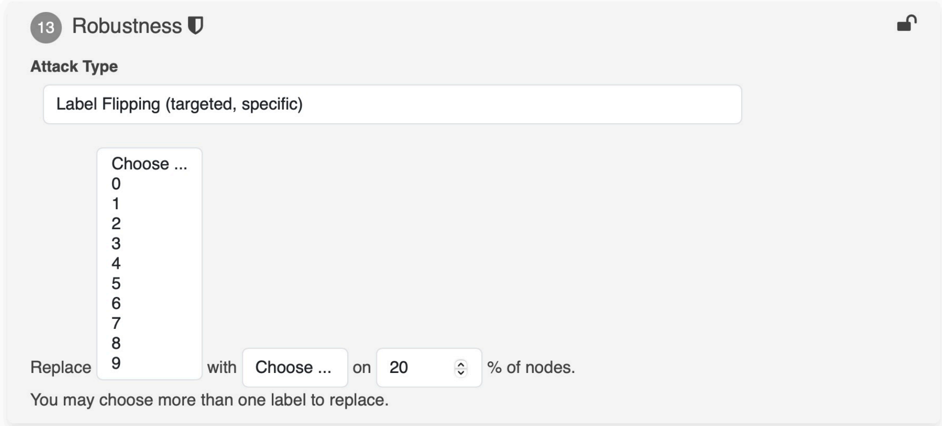


Figure 5.7: Label Flipping Attack (targeted, specific)

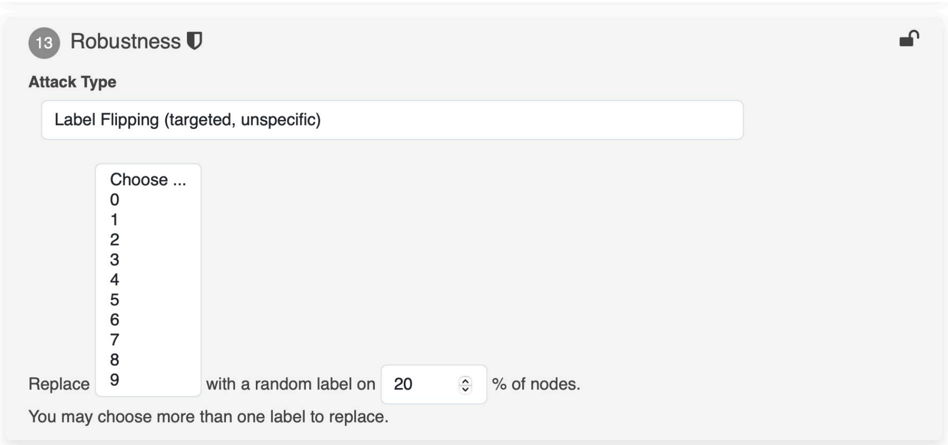


Figure 5.8: Label Flipping Attack (targeted, unspecific)

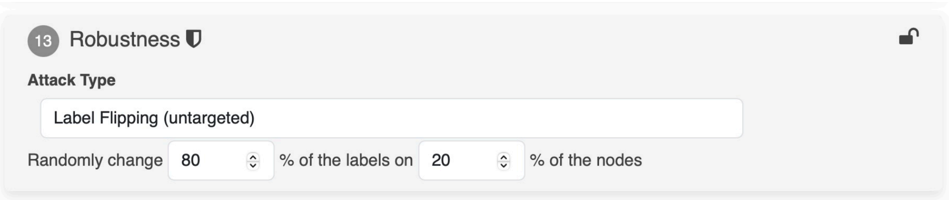


Figure 5.9: Label Flipping Attack (untargeted)

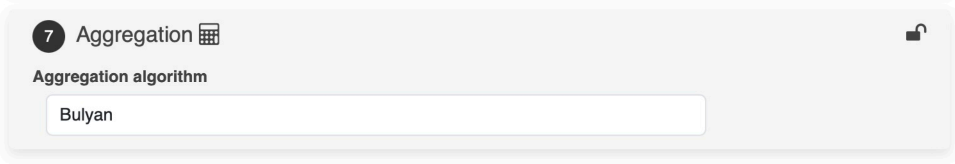


Figure 5.10: Selection of Bulyan in the Fronend

Dynamic Aggregator

The `DynamicAggregator` dynamically changes the aggregation rule used after every round. It does so in any case, not taking any information of the scenario and the other participants into consideration (proactively). The `DynamicAggregator` is implemented into Nebula as a subclass of `Aggregator`, meaning it behaves as a regular aggregation rule and the random selection of the aggregator is implemented in `run_aggregation`. Listing 5.25 shows the implementation. Note that in [6], the dynamic aggregation is implemented differently. After the end of each round, the proposed implementation checks via a configuration value if the dynamic aggregation is requested. If it is, the configured aggregation function is overwritten with a randomly chosen one just before aggregation. The integration into Nebula uses the subclass-approach mentioned earlier, mainly for the sake of simplicity and consistency.

nebula/core/aggregation/dynamicAggregator.py

```
class DynamicAggregator(Aggregator):
    def __init__(self, config = None, **kwargs):
        super().__init__(config, **kwargs)

    def run_aggregation(self, models, tensorboard_log=True):
        logging.info(f"[DynamicAggregator] Initializing Aggregation")
        ...
        super().run_aggregation(models)
        available_aggregators = [FedAvg, Krum, Median, TrimmedMean, Bulyan]
        chosen_aggregator_cls = random.choice(available_aggregators)
        logging.info(f"[DynamicAggregator] Chosen Aggregator: {chosen_aggregator_cls}")
        if tensorboard_log:
            self.engine.nebulalogger.log_text(tag="[DynamicAggregator] Chosen Aggregator", text=chosen_aggregator_cls.__name__, step=self.engine.round)
        chosen_aggregator = chosen_aggregator_cls(config=self.config)
        return chosen_aggregator.run_aggregation(models)
```

Listing 5.25: MTD DynamicAggregator

Reactive Aggregator

The `ReactiveAggregator` dynamically changes the aggregation rule if malicious model updates have been detected by the participant. To do so, it uses `reputation_calculation` to calculate the cossim-metric score for each received model. A cutoff (0.5) is then used to decide whether a model is malicious or not. If a malicious model was received, the aggregation rule is changed by invoking the `DynamicAggregator`. If no malicious model was detected, the default Aggregator configured in the frontend is used. The full implementation is shown in listing 5.26.

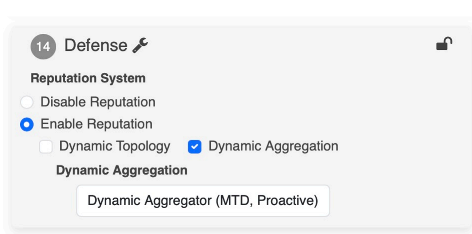
nebula/core/aggregation/reactiveAggregator.py

```
from nebula.core.aggregation.dynamicAggregator import DynamicAggregator

class ReactiveAggregator(Aggregator):
    def __init__(self, config = None, **kwargs):
        super().__init__(config, **kwargs)

    def run_aggregation(self, models):
        logging.info(f"[ReactiveAggregator] Initializing Aggregation")
        super().run_aggregation(models)
        malicious_nodes, reputation_score = self.engine.
    reputation_calculation(models)
        if len(malicious_nodes) > 0:
            # ...
            # log notifications
            dynamic_aggregator = DynamicAggregator(config=self.config,
    engine = self.engine)
            return dynamic_aggregator.run_aggregation(models,
    reactive_aggregator = True)
        else:
            default_aggregator = self.config.participant["
    aggregator_args"]["reactive_aggregator_default"]
            # ...
            # various logging
            ALGORITHM_MAP = {
                # ...
                # Map various algorithms such as FedAvg, Krum, Median,
    TrimmedMean, Bulyan, BlockReputation and DynamicAggregator
            }
            if default_aggregator not in ALGORITHM_MAP:
                logging.error(f"[ReactiveAggregator] Invalid default
    aggregator {default_aggregator}, falling back to FedAvg")
                default_aggregator = "FedAvg"
            default_aggregator_cls = ALGORITHM_MAP[default_aggregator]
            default_aggregator = default_aggregator_cls(config=self.
    config)
            return default_aggregator.run_aggregation(models)
```

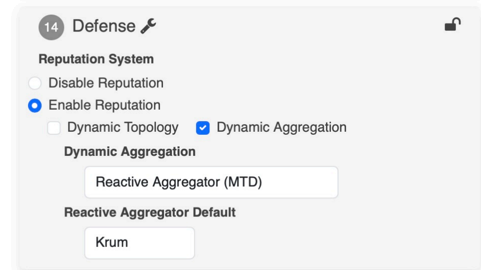
Listing 5.26: MTD ReactiveAggregator

5.3.1 Frontend

The configuration of the aggregators is shown in figure 5.11. As proposed by [5] can only be configured when the reputation system is enabled, as the `ReactiveAggregator` depends on the



DynamicAggregator Configuration



ReactiveAggregator Configuration

Figure 5.11: Frontend Configuration of the MTD Aggregators

reputation system for information about the other participants.

5.4 Privacy Auditing Component

This section describes the implementation of the Membership Inference Attacks as described in [7] into Nebula.

MIA Base Class

The `MembershipInferenceAttack` class serves as the foundation for the implemented attacks in nebula. All MIAs inherit from it, and override some of the methods with their specific implementation. The `MembershipInferenceAttack` class is initialized with a model to be attacked, a global dataset, two `DataLoader` objects for in-sample and out-sample evaluations, and an index mapping that enables decomposition of the in-sample dataset into subsets corresponding to specific nodes. The function `execute_attack` is the placeholder designed to be overridden by specific attack implementations. `evaluate_tp_for_each_node` provides the evaluation of true positives at the node level. Using the index mapping provided during initialization, the method iterates through each node's subset of in-sample data and calculates the number of true positives for that node. `evaluate_metrics` calculates key metrics (precision, recall, the false positive rate (FPR), and the F1 score) for assessing the effectiveness of an attack. The implementation of the `MembershipInferenceAttack` class is shown in listing 5.27.

nebula/addons/attacks/mia/base_MIA.py

```python

import torch

class MembershipInferenceAttack:
    def __init__(self, model, global_dataset, in_eval, out_eval,
   indexing_map):
        self.model = model
        # ...
    # various initializations, including predictions and index mapping
   etc.
    def _compute_predictions(self, model, dataloader):
        model.eval()
        predictions = []
        labels = []

        with torch.no_grad():
            for inputs, label in dataloader:
                # ...
                # perform inference and append predictions and labels
            predictions = torch.cat(predictions, dim=0)
            labels = torch.cat(labels, dim=0)
        return predictions, labels

    def execute_attack(self):
        raise NotImplementedError("Must override execute_attack")

    def evaluate_metrics(self, true_p, false_p):
        size = len(self.in_eval_pre[0])

        total_positives = true_p + false_p

        precision = true_p / total_positives if total_positives > 0 else
    0
        recall = true_p / size
        fpr = false_p / size
        f1 = 2 * precision * recall / (precision + recall) if (precision
    + recall) > 0 else 0

        return precision, recall, f1

    def evaluate_tp_for_each_node(self, in_predictions):
        nodes_tp_dict = {}

        for key, index in self.index_mapping.items():
            node_tp = in_predictions[index].sum().item()
            nodes_tp_dict[key] = node_tp

        return nodes_tp_dict
```

Listing 5.27: ”Base Class MembershipInferenceAttack”

Shadow Model Based Attack

The `ShadowModelBasedAttack` class extends the base `MembershipInferenceAttack` to implement a shadow model-based membership inference attack. This approach uses multiple shadow models, which mimic the behavior of the target model, to generate a labeled attack dataset. Using the predictions from these shadow models, an attack model is trained to infer whether specific data samples belong to the target model's training set. `_generate_attack_dataset` generates the attack dataset by training multiple shadow models and collecting their predictions and labels. The shadow models mimic the target model's behavior and are trained on subsets of the data. For each shadow model, the method instantiates a new model of the same class as the target model and trains it using the corresponding data loader from `shadow_train`. Once trained, the shadow model's predictions and labels are computed for both its training and test datasets using the `_compute_predictions` method inherited from the base class. Then, predictions and labels for all shadow models are concatenated to form the testing and training set that are used as the input for the attack model training. The `MIA_shadow_model_attack` method executes the membership inference attack. It builds an attack dataset, trains an attack model, and evaluates the attack's effectiveness (see `in_out_samples_check`, which evaluates whether each sample in a dataset is classified as a member of the training set by the attack model). The implementation of this attack is shown in listing 5.28

nebula/addons/attacks/mia/base_MIA.py

```
class ShadowModelBasedAttack(MembershipInferenceAttack):
    def __init__(self, model, global_dataset, in_eval, out_eval, indexing_map, max_epochs, shadow_train,
                 shadow_test, num_s, attack_model_type):
        super().__init__(model, global_dataset, in_eval, out_eval, indexing_map)
        self._generate_attack_dataset()
        # ...
        # various initializations, including training hyperparameters, the number of  shadows and the dataloaders

    def _generate_attack_dataset(self):
        model_class = type(self.model)

        # ...
        # create empty datasets
        for i in range(self.num_shadow):
            # ...
            # create a shadow model and trainer, fit shadow model i, compute and store predictions in the empty datasets
        self.shadow_train_res = (torch.cat(s_tr_pre, dim=0), torch.cat(s_tr_label, dim=0))
        self.shadow_test_res = (torch.cat(s_te_pre, dim=0), torch.cat(s_te_label, dim=0))

    def MIA_shadow_model_attack(self):
        # ...
        # init models, datasets and dataloaders for attack dataset
        if self.attack_model_type == "Neural Network":
            attack_model = SoftmaxMLPClassifier(10, 64)
        else:
            pass
        # ...
        # create trainer and fit model
        def in_out_samples_check(model, dataset):
            # ...
            # Load predictions from dataset and create dataloader
            # Create empty dataset for labels

            with torch.no_grad():
                for batch in dataloader:
                    # ...
                    # perform predictions and take the max value
                    # append prediction labels to empty dataset
                predicted_label = torch.cat(predicted_label, dim=0)
            return predicted_label

        # ...
        # use in_out_samples_check and calculate f1, precision, recall from true and false positives
        return precision, recall, f1
```

Listing 5.28: "Shadow Model Based Attack"

Class Metric Based Attack

The `ClassMetricBasedAttack` is a subclass of the `ShadowModelBasedAttack` class and implements a specific type of membership inference attack that utilizes class-based metrics. This approach uses a single shadow model to derive thresholds based on metrics like confidence, entropy, and modified entropy. These thresholds are then applied to the target model to infer membership. The implementation of this attack is shown in listing 5.29

nebula/addons/attacks/mia/ClassMetricMIA.py

```python
import numpy as np
import torch
from nebula.addons.attacks.mia.ShadowModelMIA import
    ShadowModelBasedAttack
class ClassMetricBasedAttack(ShadowModelBasedAttack):
    def __init__(...):
        super().__init__(...)
        self.num_classes = 10
        # Compute confidences for shadow and target datasets
        # Includes self.s_in_conf, self.s_out_conf, self.t_in_conf, self
    .t_out_conf
        # Compute entropies and modified entropies for all datasets
        # Includes self.s_in_entr, self.s_out_entr, self.t_in_entr, self
    .t_out_entr,
        # and their modified versions
        self._compute_entropies()
        # ...
    def _log_value(self, probs, small_value=1e-30):
        return -np.log(np.maximum(probs, small_value))
    def _entr_comp(self, probs):
        # Compute prediction entropy for given probabilities
        return np.sum(np.multiply(probs, self._log_value(probs)), axis
    =1)
    def _m_entr_comp(self, probs, true_labels):
        # Compute modified entropy for given probabilities and true
    labels
        # Modifies log probabilities for true labels to reverse
    probabilities
        # ...
    def _thre_setting(self, tr_values, te_values):
        # Determine optimal threshold for membership inference using
    accuracy
        # ...
    def _mem_inf_thre(self, s_tr_values, s_te_values, t_tr_values,
    t_te_values):
        # Perform membership inference attack by thresholding feature
    values
        # ...
    def mem_inf_benchmarks(self):
        # Select method for attack based on method_name and perform
    membership inference
        # Perform membership inference attack using confidence, entropy,
     or modified entropy based on the selected method.
        # ...
    def _compute_confidences(self):
        # Compute class confidence for shadow and target datasets
        ...
    def _compute_entropies(self):
        # Compute entropy and modified entropy for shadow and target
    datasets
        ...
```

Listing 5.29: "MIA Class Metric Based"

Metric Based Attack

The `MetricBasedAttack` class extends the `MembershipInferenceAttack` base class and implements several metric-based membership inference attack strategies. These attacks use properties like correctness, loss, maximal confidence, entropy, and sensitivity of model predictions. The `MIA_correctness_attack`-method determines membership based on prediction correctness. If the predicted label matches the true label, the sample is classified as part of the training set. The `MIA_loss_attack`-method infers membership based on the prediction loss. Samples with loss below a precomputed training threshold are classified as training samples. The `MIA_maximal_confidence_attack`-method evaluates membership based on maximal prediction confidence. It determines thresholds that maximize the F1 score to distinguish between training and non-training samples. The `MIA_entropy_attack` evaluates membership based on prediction entropy. Thresholds are applied to minimize uncertainty and maximize the F1 score for membership inference. The `MIA_sensitivity_attack`-method evaluates membership based on prediction sensitivity. It clusters samples using the L2 norm of the Jacobian matrix (the partial derivatives from the model's prediction function) and infers membership based on the clustering results. The implementation of these attacks is shown in listing B.14.

5.4.1 Frontend

The configuration interface for the MIAs in Nebula was added as suggested in [7]. Figure 5.12 showcases the extended frontend configuration options. By default, when no attack is selected, the configuration screen remains minimal, as shown in the first subfigure. Upon enabling the "Shadow Model Based Attack," additional fields become visible, allowing users to configure parameters such as the number of shadow models, node data samples, attack model type, and defense methods as illustrated in the second subfigure. Similarly, selecting the "Metric Based Attack" option reveals configuration fields for specifying node data sample size, and metric details relevant to these methods, as shown in the third subfigure.

14 Privacy

Attack Type

No Attack

Defense Method

No Defense

Configuration without Attack

14 Privacy

Attack Type

Shadow Model Based MIA

Size of Node Data Samples

Enter size of node data samples

Shadow Model Number

Enter number of shadow models

Attack Model

Neural Network

Defense Method

No Defense

Configuration with Shadow Model Based Attack

14 Privacy

Attack Type

Metric Based MIAs

Size of Node Data Samples

Enter size of node data samples

Metric Detail

Prediction Correctness

Defense Method

No Defense

Configuration with Metric Based Attack

Figure 5.12: Frontend Configuration of MIAs

Chapter 6

Evaluation

For all evaluations, the participant names correspond to the IP addresses as shown in table 6.1.

Participant Name	**IP:Port**
participant_0	192.168.50.2:45000
participant_1	192.168.50.3:45000
participant_2	192.168.50.4:45000
participant_3	192.168.50.5:45000
participant_4	192.168.50.6:45000
participant_5	192.168.50.7:45000
participant_6	192.168.50.8:45000
participant_7	192.168.50.9:45000
participant_8	192.168.50.10:45000
participant_9	192.168.50.11:45000
⋮	⋮

Table 6.1: Mapping of participant name to IP-Address

6.1 Node Selection Strategy

AllSelector

To evaluate whether the `AllSelector` works as expected, a scenario with 5 nodes training for 5 rounds was used. For this evaluation, the other settings (model, dataset, ...) are irrelevant. According to [4], `AllSelector` should select all available neighbors as well as himself for aggregation. Figure 6.1 shows the TensorBoard logs indicating that `AllSelector` behaves as intended.

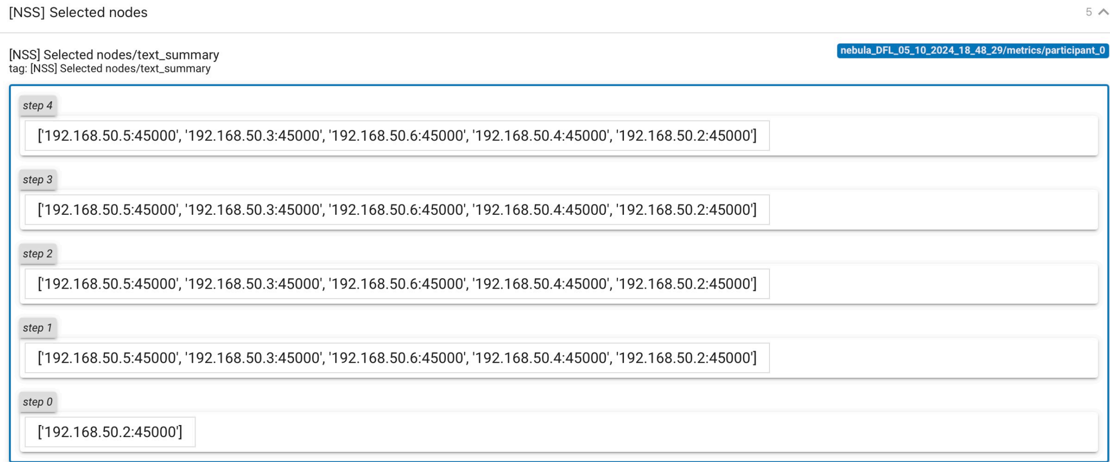


Figure 6.1: TensorBoard Logs AllSelector

RandomSelector

To evaluate whether the `RandomSelector` works as expected, a scenario with 10 nodes training for 5 rounds was used. For this evaluation, the other settings (model, dataset, ...) are irrelevant. According to the mechanism of `RandomSelector` shown in section 5, it should select a random amount of nodes indiscriminately. The only node always included should be itself. Figure 6.2 shows the TensorBoard logs indicating that `RandomSelector` behaves as intended. We can see that the nodes selected in each round change randomly, and only the initial node is included in every selection. The amount of nodes selected also varies, as intended.

[NSS] Selected nodes/text_summary
tag: [NSS] Selected nodes/text_summary
nebula_DFL_30_10_2024_16_27_21/metrics/participant_1

step 4
['192.168.50.2:45000', '192.168.50.9:45000', '192.168.50.6:45000', '192.168.50.4:45000', '192.168.50.10:45000', '192.168.50.7:45000', '192.168.50.3:45000']

step 3
['192.168.50.10:45000', '192.168.50.6:45000', '192.168.50.8:45000', '192.168.50.7:45000', '192.168.50.2:45000', '192.168.50.11:45000', '192.168.50.3:45000']

step 2
['192.168.50.6:45000', '192.168.50.2:45000', '192.168.50.10:45000', '192.168.50.3:45000']

step 1
['192.168.50.2:45000', '192.168.50.8:45000', '192.168.50.4:45000', '192.168.50.10:45000', '192.168.50.5:45000', '192.168.50.7:45000', '192.168.50.3:45000']

step 0
['192.168.50.2:45000', '192.168.50.3:45000']

Participant 1

[NSS] Selected nodes/text_summary
tag: [NSS] Selected nodes/text_summary
nebula_DFL_30_10_2024_16_27_21/metrics/participant_2

step 4
['192.168.50.10:45000', '192.168.50.7:45000', '192.168.50.5:45000', '192.168.50.4:45000']

step 3
['192.168.50.8:45000', '192.168.50.9:45000', '192.168.50.5:45000', '192.168.50.4:45000']

step 2
['192.168.50.8:45000', '192.168.50.4:45000']

step 1
['192.168.50.2:45000', '192.168.50.10:45000', '192.168.50.7:45000', '192.168.50.8:45000', '192.168.50.4:45000']

step 0
['192.168.50.2:45000', '192.168.50.4:45000']

Participant 2

Figure 6.2: TensorBoard Logs RandomSelector

PrioritySelector

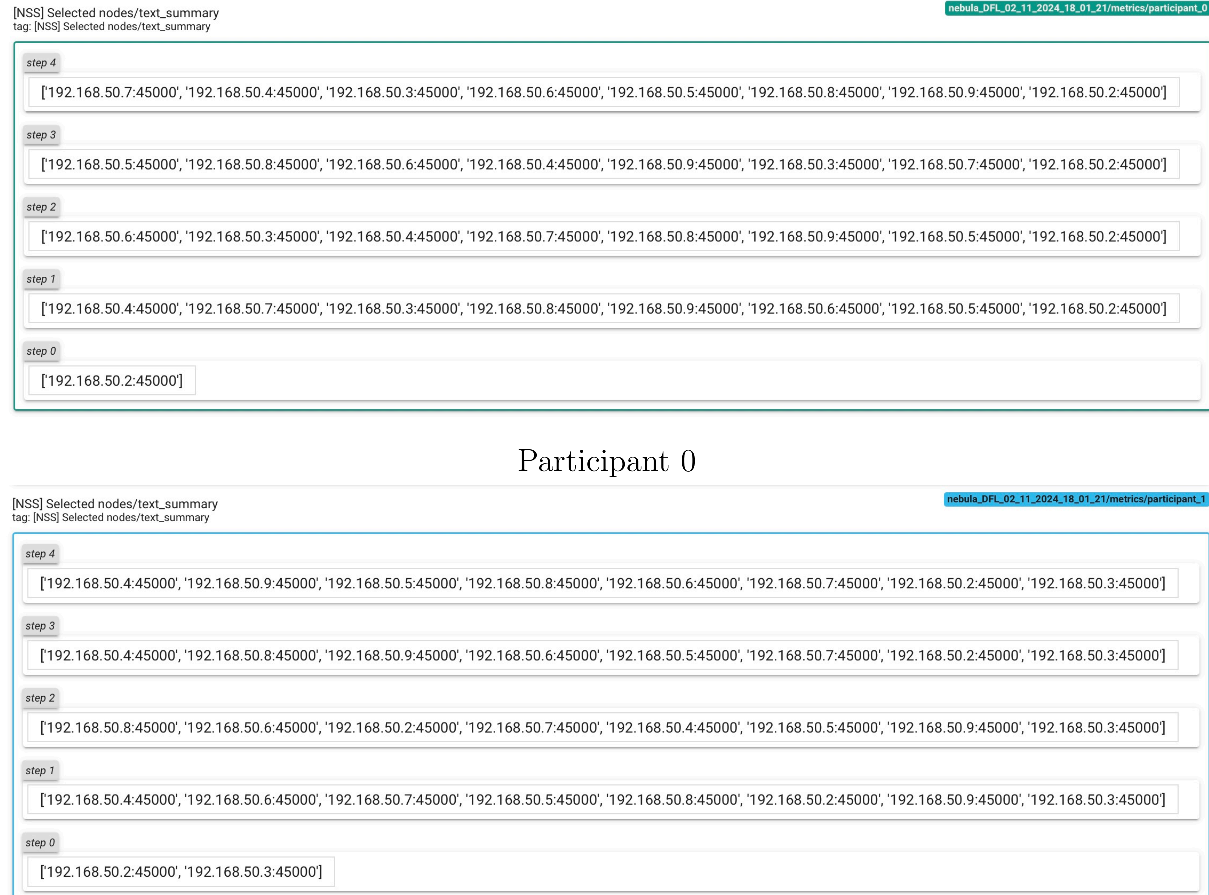
As mentioned in section 5, the `PrioritySelector` uses the features submitted by each node to calculate a score. This score is then used to calculate weights. These weights then define the probability of each node being chosen for aggregation. While this approach certainly has its benefits, it makes checking the correctness of the implementation difficult due to the inherent randomness of the results. To make the evaluation more deterministic, the weighting of the features and the algorithm was changed. The `latency` feature is weighted much more than the others, as this feature can be manipulated reliably through Nebula (see subsection 5.1.4). Also, the random weighting was replaced by selecting the nodes with the best features. Listing 6.1 shows the specific changes applied.

The scenario used in this evaluation consists of 10 nodes training for 5 rounds. The nodes participant_8 and participant_9 have a network delay of 150ms. The TensorBoard logs of the nodes without latency constraints are shown in figure 6.3, the ones with latency constraints are shown in figure 6.4. We can see that the participants 0 and 1 always choose nodes 0-7, excluding the nodes with latency constraints. The TensorBoard logs of the nodes 8 and 9 (the nodes with added constraints) show that they never contain each other, but do contain themselves. This behaviour is expected, as each node always adds itself to the aggregation set.
The features of the participants (as extracted by participant 2 in round 2) can be seen in listing B.11, it also shows the the weights and scores calculated.

nebula/core/selectors/priority_selector.py

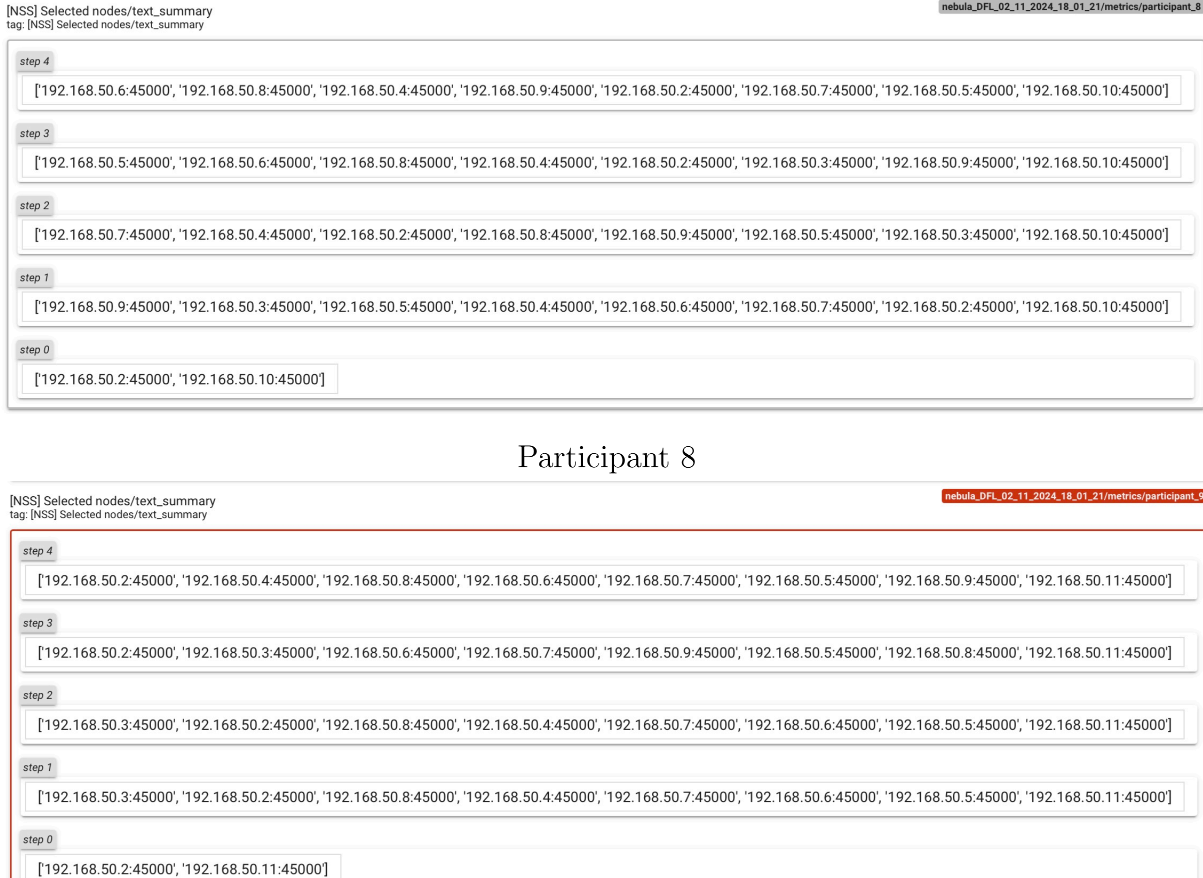
```
1  ...
2  # Original Feature Weights provided in Report / Thesis
3  # FEATURE_WEIGHTS = [1.0, 1.0, 1.0, 0.5, 0.5, 10.0, 3.0]
4  # Feature Weights for Testing (Latency can be changed reliably by
       virtual constraints)
5  FEATURE_WEIGHTS = [0, 0, 0, 0, 0, 100, 0]
6  ...
7  # Select nodes according to thesis (weighted probability)
8  # selected_nodes = np.random.choice(
9  #     neighbors, num_selected, replace = False, p = p[0]
10 # ).tolist()
11 # Select num_selected nodes with the highest score (or the derived
       probability) for easier evaluation
12 selected_nodes = [neighbors[i] for i in np.argsort(scores)[-num_selected
       :]]
13 ...
```

Listing 6.1: Changes to PrioritySelector for Evaluations



Participant 1

Figure 6.3: TensorBoard Logs PrioritySelector (Nodes without latency constraint)



Participant 8

Participant 9

Figure 6.4: TensorBoard Logs PrioritySelector (Nodes with latency constraint)

6.2 Poisoning Attacks

In this section, the various poisoning attacks that have been implemented are evaluated and scrutinized. This includes targeted labelflipping, both with a specific target to flip to and without, as well as untargeted labelflipping and the fang labeflipping attack [14]. Finally, an update manipulation attack is also evaluated.

The summaries of the models (defined by Nebula) used in this evaluation are available in the Appendix. If not stated otherwise, n refers to the total amount of nodes (benign + malicious) and f to the amount of malicious nodes.

Labelflipping targeted (specific)

In the targeted specific label-flipping attack, the malicious nodes swap specific label pairs (e.g. flipping label 1 to 7). The evaluation scenarios are defined as follows:

- $n = 5$ nodes.

#	Global Accuracy
LTS_0	0.9542
LTS_1	0.9567
LTS_2	0.9350
LTS_3	0.8586
LTS_4	0.8588
LTS_5	0.8586

Table 6.2: Global model accuracy in a targeted label flipping attack with specific target, depending on the number of malicious nodes.

#	Setup	n	f	AGR	Rounds	Changed Labels
LTS_0	DFL	5	0	FedAvg	5	0% (No Attack, Baseline)
LTS_1	DFL	5	1	FedAvg	5	100%
LTS_2	DFL	5	2	FedAvg	5	100%
LTS_3	DFL	5	3	FedAvg	5	100%
LTS_4	DFL	5	4	FedAvg	5	100%
LTS_5	DFL	5	5	FedAvg	5	100%

Table 6.3: Evaluation Scenarios Label Flipping Attack (targeted, specific)

- **LTS_0**: baseline without attack for comparison purposes
- **LTS_1 to LTS_5**: The number of malicious nodes ('f') gradually increases from 1 to 5, with all malicious nodes executing the attack.

Expected results:

- No degradation in **LTS_0** (all nodes are benign).
- Increasing degradation in accuracy of the targeted class as the number of malicious nodes increases.
- At **LTS_5**, no node classifies the targeted class correctly at all, accuracy should be 0 for this class (Confusion matrix shows 0 in the intersection of predicted/correct).
- The global accuracy should be not impacted in **LTS_0**, with the accuracy drop rising sharply after the percentage of malicious nodes exceeds 50%. With 10 classes, accuracy should eventually drop to 10% (random guessing) at **LTS_5**

Table **??** shows the achieved global accuracies depending on the number of malicious clients. As expected, accuracy drops sharply after 3 malicious nodes, as a majority of the participants have turned malicious.

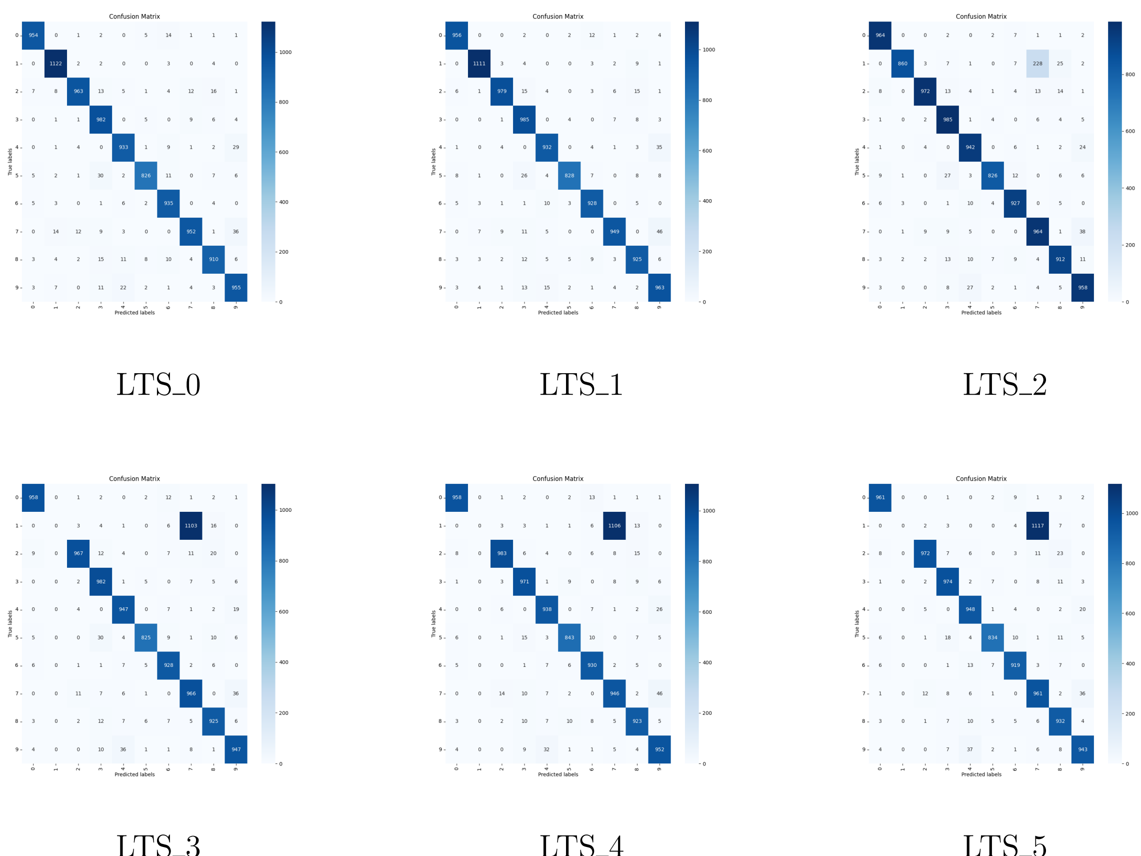


Figure 6.5: Confusion Matrices Label Flipping Attack (targeted, specific)

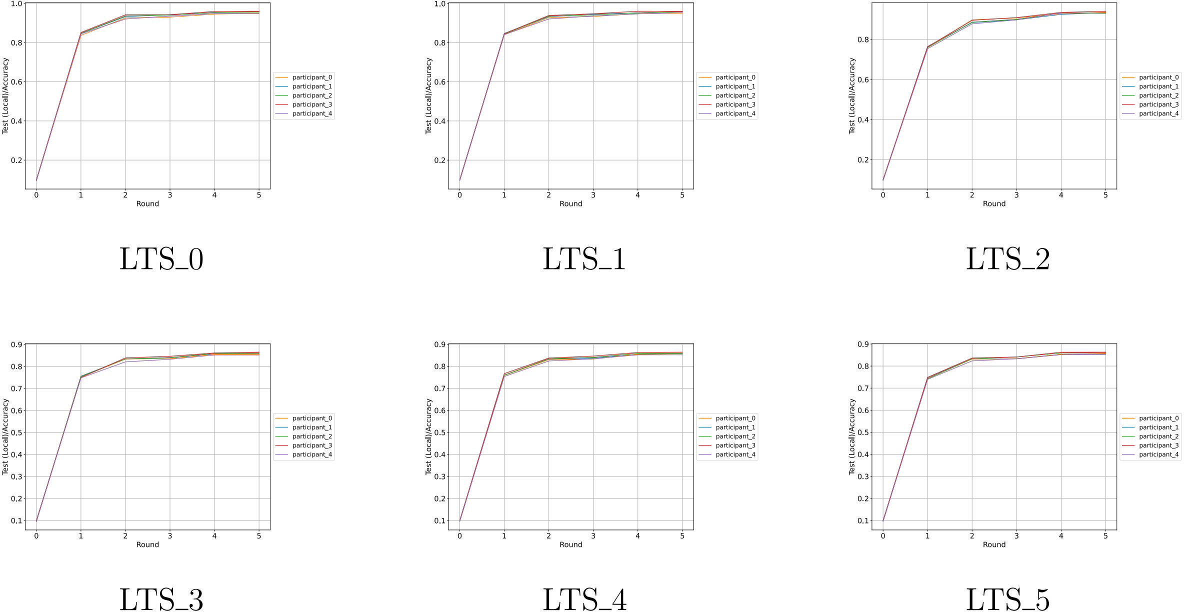


Figure 6.6: Local Accuracies Label Flipping Attack (targeted, specific)

#	Global Accuracy
LTU_0	0.9528
LTU_1	0.9532
LTU_2	0.9397
LTU_3	0.8578
LTU_4	0.8549
LTU_5	0.8554

Table 6.4: Global model accuracy in a targeted label flipping attack without a specific target (unspecific), depending on the number of malicious nodes.

Labelflipping targeted (unspecific)

In the targeted unspecific labelflipping attack, the malicious nodes swap a specific label to a randomly selected other label (e.g., flipping label 1 to x $\in$ all classes). The evaluation scenarios are defined as follows:

- $n = 5$ nodes
- **LTU_0**: baseline without attack for comparison purposes.
- **LTU_1** to **LTU_5**: The number of malicious nodes ('f') gradually increases from 1 to 5, with all malicious nodes executing the attack.

Expected results:

- No degradation in **LTU_0** (all nodes are benign).
- Increasing degradation in accuracy of the specific class as the number of malicious nodes increases.
- At **LTU_5** no node classifies the targeted class correctly at all, so accuracy of this class should be 0 (Confusion matrix shows 0 in the intersection of predicted/correct).
- The global accuracy should be not impacted in **LTU_0**, with the accuracy drop rising sharply after k exceeds 50%. With 10 classes, accuracy should eventually drop by 10% (random guessing of 1 of 10 classes) at **LTU_5**

Table **??** shows global model accuracy as the number of malicious nodes increases. As expected, the accuracy decreases steadily, with a sharp drop as the malicious nodes enter into majority.

Labelflipping, untargeted

In the untargeted labelflipping attack, the malicious nodes swap the labels of a certain percentage (k) of the training data to a random label. The evaluation scenarios are defined as follows:

#	Setup	n	f	AGR	Rounds	Changed Labels
LTU_0	DFL	5	0	FedAvg	5	0% (No Attack, Baseline)
LTU_1	DFL	5	1	FedAvg	5	100%
LTU_2	DFL	5	2	FedAvg	5	100%
LTU_3	DFL	5	3	FedAvg	5	100%
LTU_4	DFL	5	4	FedAvg	5	100%
LTU_5	DFL	5	5	FedAvg	5	100%

Table 6.5: Evaluation Scenarios Label Flipping Attack (targeted, unspecific)

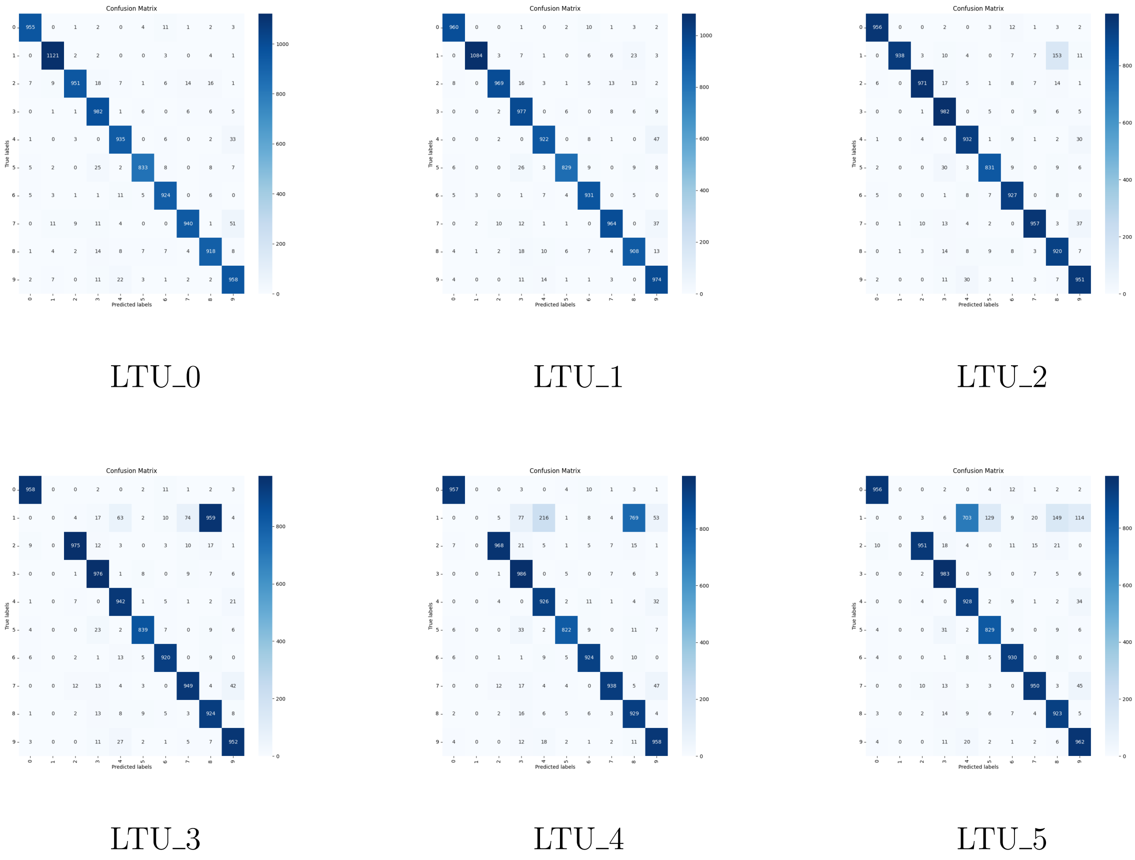


Figure 6.7: Confusion Matrices Label Flipping Attack (targeted, unspecific)

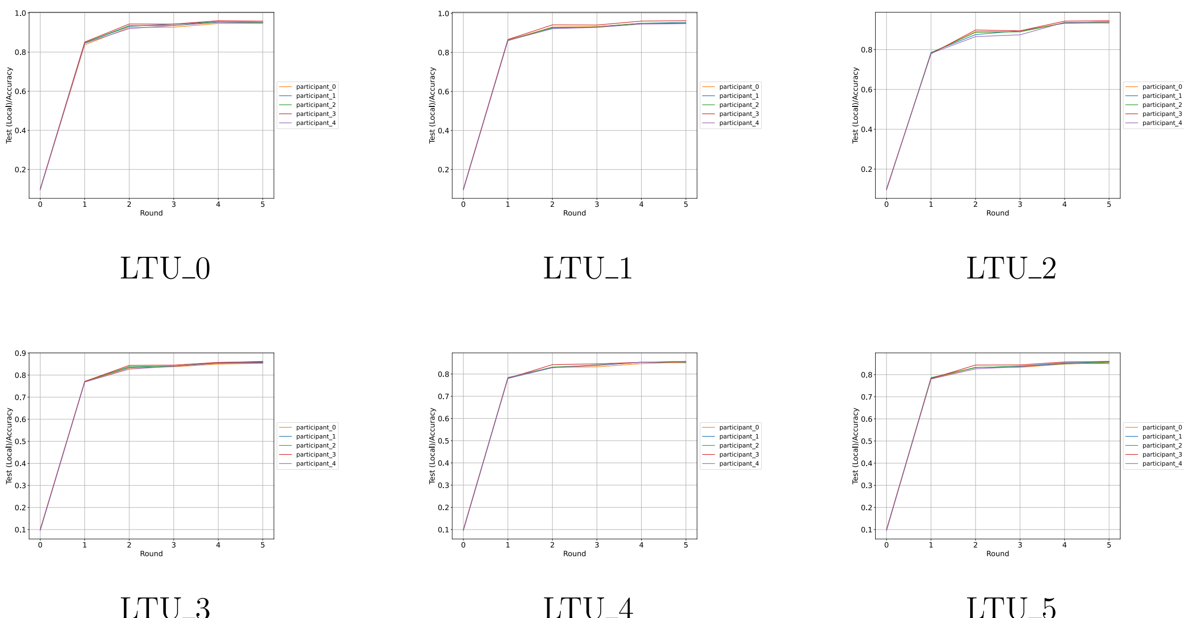


Figure 6.8: Local Accuracies Label Flipping Attack (targeted, unspecific)

- **LU_0**: baseline without attack for comparison purposes
- **LU_1** to **LU_5**: The number of malicious nodes ('f') gradually increases from 1 to 5, with all malicious nodes executing the attack. ($k = 20$)
- **LU_6** to **LU_9**: The number of malicious nodes ('f') gradually increases from 1 (**LU_6**) to 5 (**LU_9**), with all malicious nodes executing the attack. ($k = 80$)

Expected results:

- No degradation in **LU_0** (all nodes are benign)
- Increasing degradation in accuracy of the specific class as the number of malicious nodes increases.
- The global accuracy should not be impacted in LU_0 and gradually decrease towards 10% (random guess, given 10 classes) as the number of malicious nodes and k increase. The decrease should be sharper with a higher k.

6.6 shows the global model accuracy both for $k = 20$ (**LU_0** to **LU_5**) and $k = 80$ (**LU_6** to **LU_11**). As expected, the global model accuracy declines in both cases, while $k = 20$ drops significantly less than $k = 80$. In both cases, the drop is most significant when the malicious nodes enter majority.

FANG Label Flipping Attack

In the FANG [14] label flipping attack, the malicious nodes swap the labels to a different one with the following logic for classes 1-9:

Table 6.6: Global model accuracy in a targeted labelflipping attack without a specific target (unspecific), depending on the number of malicious nodes.

#	Global Accuracy
LU_0	0.9539
LU_1	0.9537
LU_2	0.9508
LU_3	0.9424
LU_4	0.9369
LU_5	0.9336
LU_6	0.9686
LU_7	0.9512
LU_8	0.9203
LU_9	0.7999
LU_10	0.6918
LU_11	0.5365

#	Setup	n	f	AGR	Rounds	Changed Labels
LU_0	DFL	5	0	FedAvg	5	0% (No Attack, Baseline)
LU_1	DFL	5	1	FedAvg	5	20%
LU_2	DFL	5	2	FedAvg	5	20%
LU_3	DFL	5	3	FedAvg	5	20%
LU_4	DFL	5	4	FedAvg	5	20%
LU_5	DFL	5	5	FedAvg	5	20%
LU_6	DFL	5	0	FedAvg	5	0% (No Attack, Baseline)
LU_7	DFL	5	1	FedAvg	5	80%
LU_8	DFL	5	2	FedAvg	5	80%
LU_9	DFL	5	3	FedAvg	5	80%
LU_10	DFL	5	4	FedAvg	5	80%
LU_11	DFL	5	5	FedAvg	5	80%

Table 6.7: Evaluation Scenarios Label Flipping Attack (untargeted)

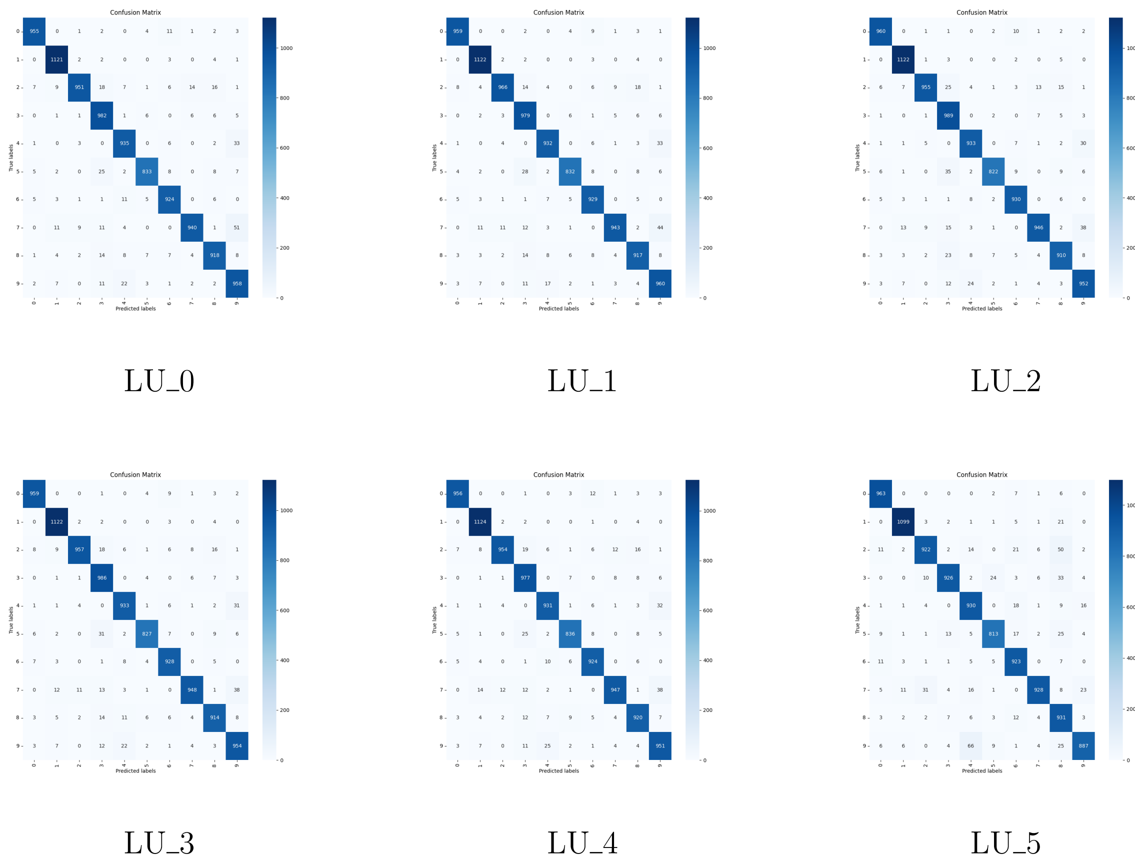


Figure 6.9: Confusion Matrices Label Flipping Attack (untargeted, scenario 0-5)

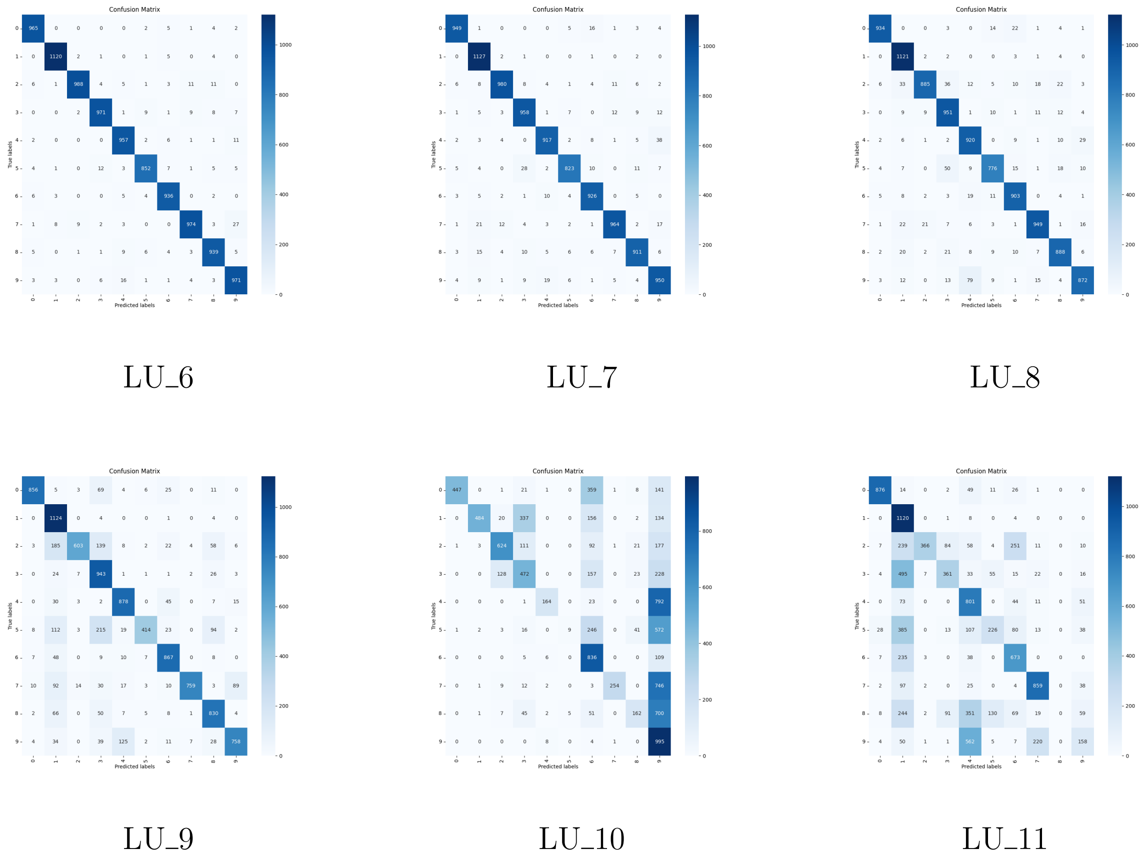


Figure 6.10: Confusion Matrices Label Flipping Attack (untargeted, scenario 5-11)

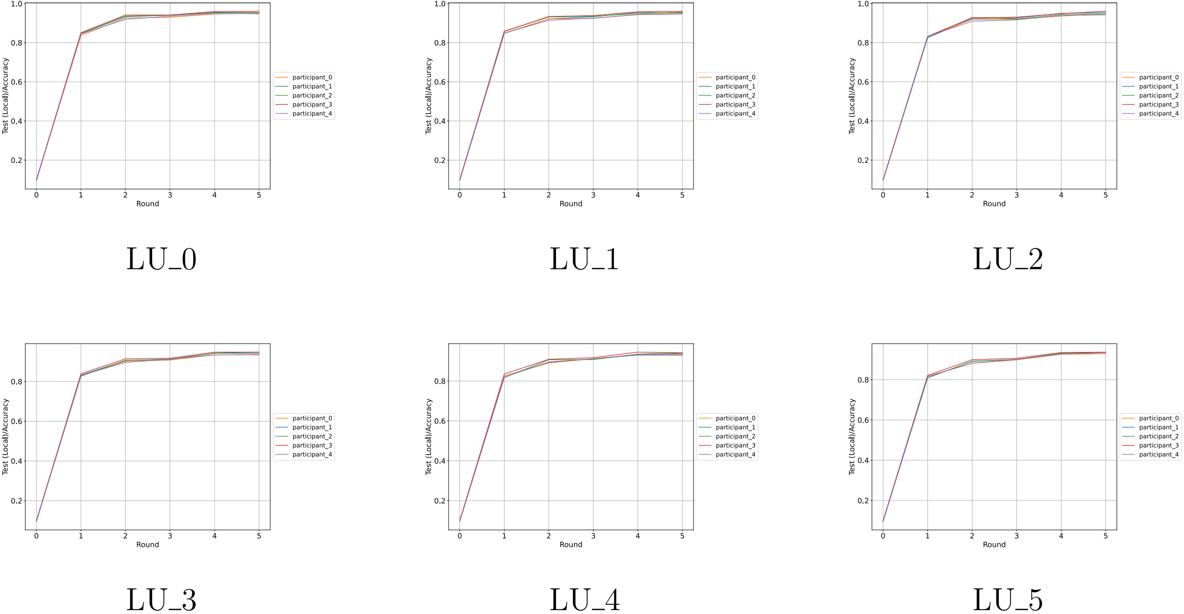


Figure 6.11: Local Accuracies Label Flipping Attack (untargeted, scenario 0-5)

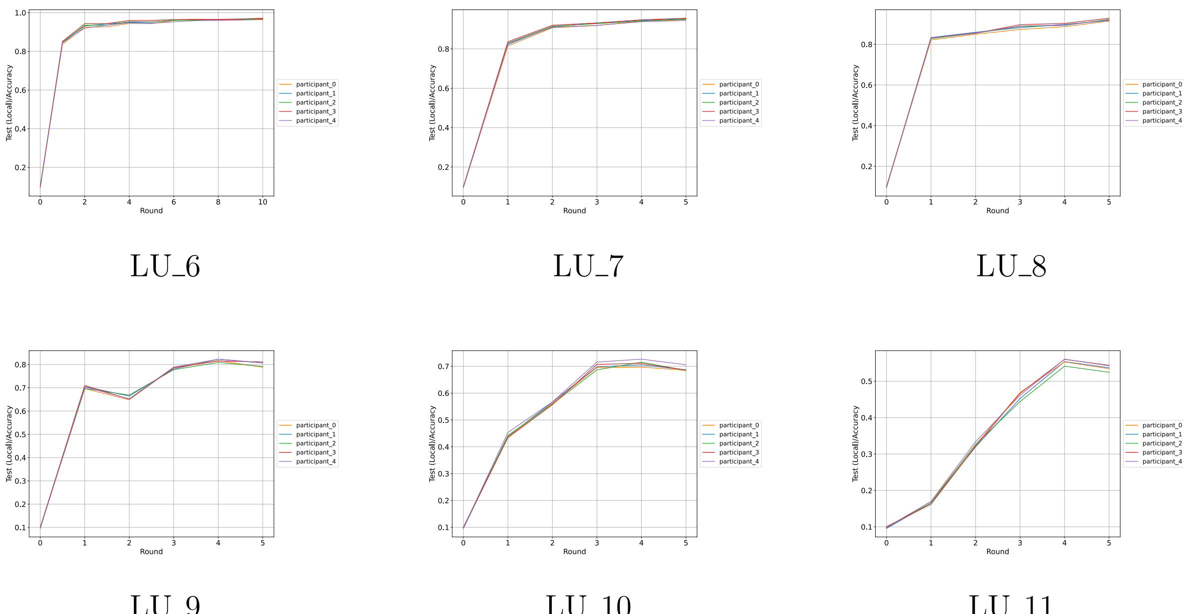


Figure 6.12: Local Accuracies Label Flipping Attack (untargeted, scenario 5-11)

- 1 -> 9
- 2 -> 7
- ...

The evaluation scenarios are defined as follows:

- **LF_0**: baseline without attack for comparison purposes
- **LF_1** to **LF_5**: The number of malicious nodes ('f') gradually increases from 1 to 5, with all malicious nodes executing the attack.

Expected result:

- The accuracy drop should be substantial initially and increase sharply after majority is reached.
- The confusion matrix will get inverted.

As table 6.8 shows, the accuracy sharply drops once majority is reached, while the confusion matrix begins showing signs of disturbance with two malicious nodes already. By the time majority is reached, the inversion of the confusion matrix is clearly visible. These results match expectations.

Table 6.8: Global model accuracy in a FANG [14] labelflipping attack, depending on the number of malicious nodes.

#	Global Accuracy	Confusion Matrix Status
LF_0	0.9516	OK
LF_1	0.9358	OK
LF_2	0.7406	partly disturbed
LF_3	0.1648	disturbed
LF_4	0.0063235	inverted
LF_5	0.004025	inverted

#	Setup	n	f	AGR	Rounds
LF_0	DFL	5	0	FedAvg	5
LF_1	DFL	5	1	FedAvg	5
LF_2	DFL	5	2	FedAvg	5
LF_3	DFL	5	3	FedAvg	5
LF_4	DFL	5	4	FedAvg	5
LF_5	DFL	5	5	FedAvg	5

Table 6.9: Evaluation Scenarios Label Flipping Attack (by [14])

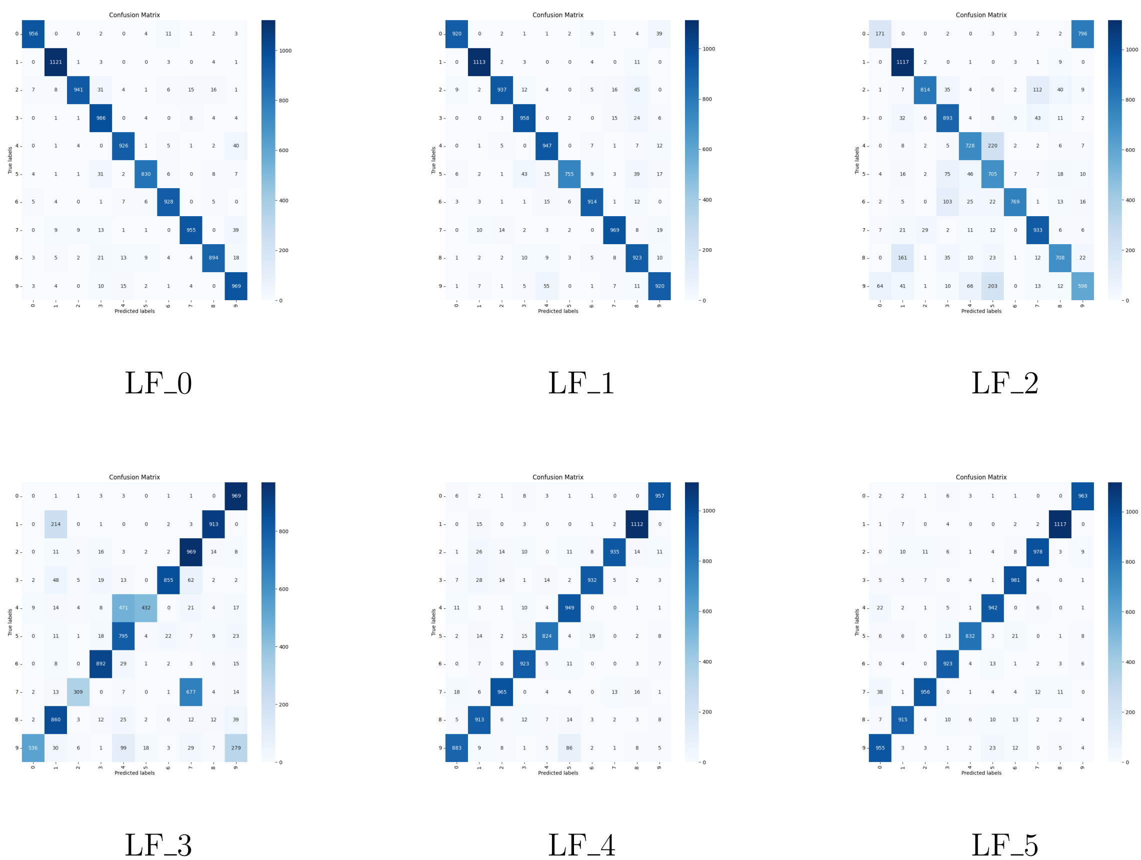


Figure 6.13: Confusion Matrices Label Flipping Attack (by [14])

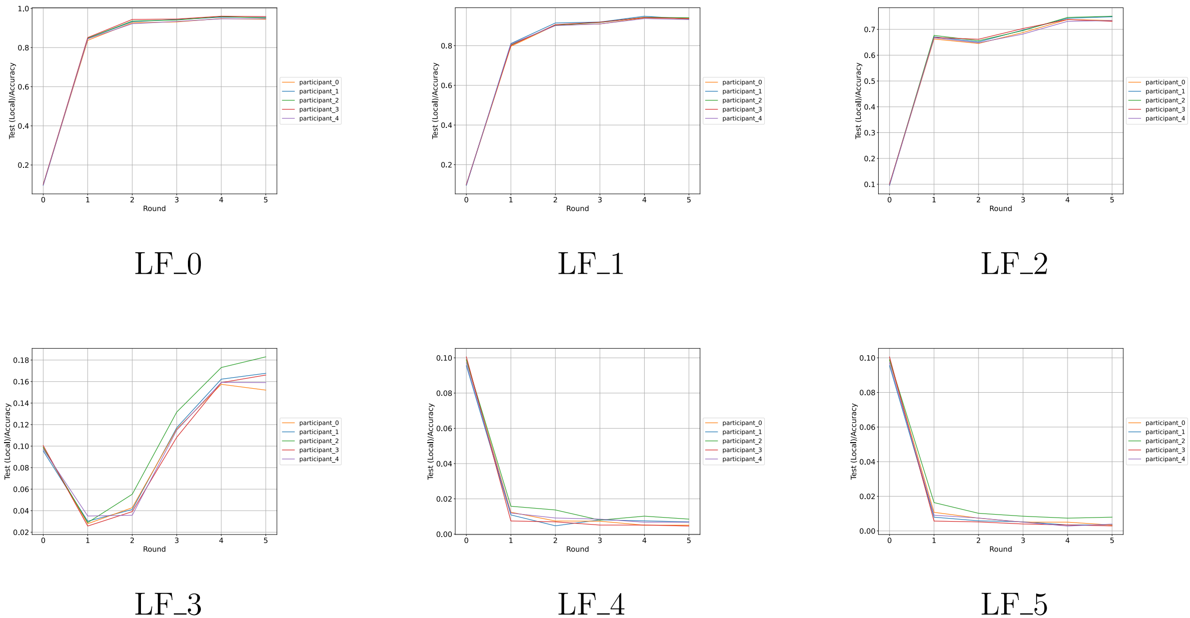


Figure 6.14: Local Accuracies Label Flipping Attack (by [14])

Update Manipulation

To evaluate the calculation of `z`, we use the example from section 3.3 in [11]. For the given scenario with 26 benign nodes and 24 malicious nodes (50 nodes in total) [11] calculates `z` as 1.75. Entering the same scenario in Nebula (50 nodes with 37.9% malicious) gives the correct value.

6.3 Moving Target Defense

Dynamic Aggregator (Proactive)

As mentioned in section 5.3, every round `DynamicAggregator` randomly selects one of the available Aggregators for use. However, all seeds are set to a fixed value in Nebula. For the `DynamicAggregator` to work as intended, the seed needs to be reset to a random value different in each node (otherwise they will all select the same nodes. To do this, the code snippet shown on the lines 4 and 5 of listing 6.2 was inserted into `run_aggregation` before the call to `random.choice`. The scenario used to evaluate the correctness of the implementation has 3 nodes training 4 rounds. Figure 6.15 shows the TensorBoard logs of the scenario, where each node logs the Aggregator selected. Note that without the change to the code (listing 6.2) all nodes would choose the same aggregator each round due to the fixed seed.

nebula/core/aggregation/dynamicAggregator.py

```
1 ...
2 available_aggregators = [FedAvg, Krum, Median, TrimmedMean, Bulyan]
3
4 import time
```

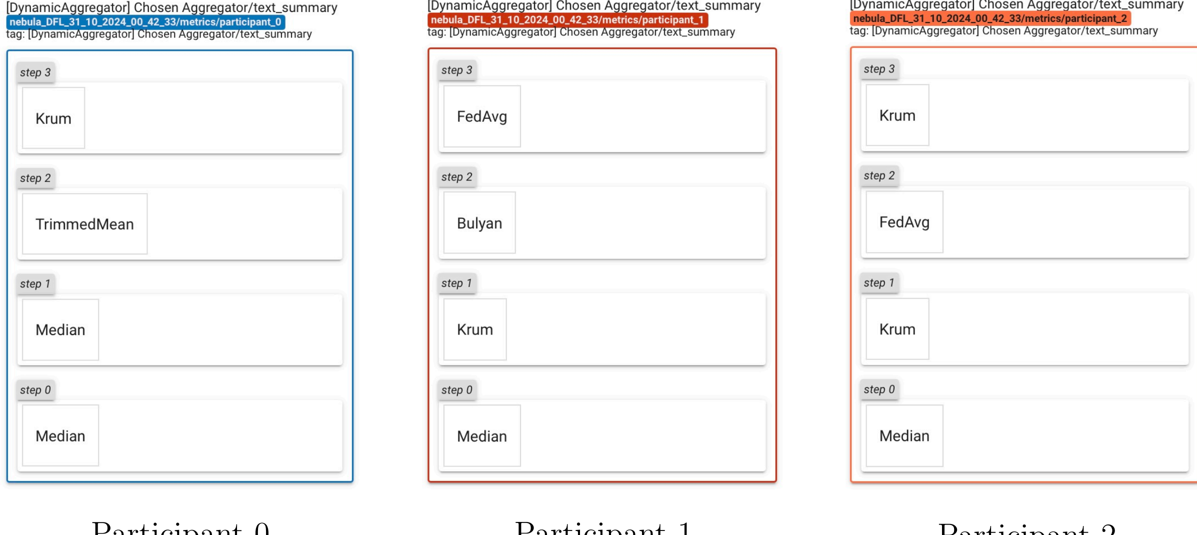


Figure 6.15: TensorBoard Logs of DynamicAggregator Scenario

```
5 random.seed(int(str(time.time_ns())[-8:]))
6
7 chosen_aggregator_cls = random.choice(available_aggregators)
8 ...
```

Listing 6.2: Labelflipping Attack: targeted; unspecific

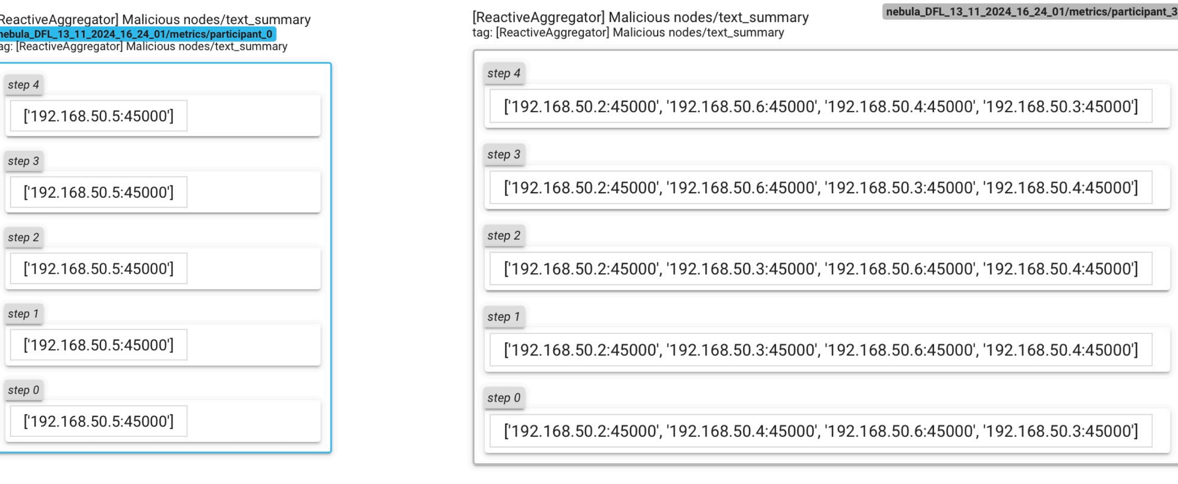
Dynamic Aggregator (Reactive)

As `ReactiveAggregator` creates an instance of `DynamicAggregator` (if a malicious node is detected), the remarks in the section above regarding the random seed also apply.

The scenario used to evaluate the correctness of the implementation has 5 nodes training 5 rounds, with one malicious node (participant 3). Figure 6.16 shows the TensorBoard logs of the scenario of participant 0 and 3. As we can see, the participant 0 correctly identifies participant 3 as a malicious node. We also see that participant 3 identifies all other participants as malicious, which is intended behaviour (see section 5.3 for details). The logs of the scenario (see listing B.12 for participant 0 and B.13 for participant 3), also show that the `DynamicAggregator` is instantiated correctly and changes the Aggregator as intended.

6.4 Privacy Auditing Component

To evaluate the correct implementation of the membership inference attacks, we designed a scenario that allows the membership inference attack to work successfully. As mentioned in [20], membership inference attacks benefit from scenarios where **overfitting** occurs. Overfitting may happen when a machine learning model memorizes the training data instead of learning generalizable patterns. This typically occurs when the model is excessively complex for the amount of training data or, in our case, only little training data is available. In our scenario, we intentionally induced overfitting by running a scenario with 15 nodes. In the default configuration of



Participant 0 Participant 3

Figure 6.16: TensorBoard Logs of ReactiveAggregator Scenario

Table 6.10: Comparison of F1 scores, recall and precision for a well-configured scenario and a scenario in which overfitting is occuring.

Setup	**F1 Score**	**Recall**	**Precision**
3 Nodes Reference, MNIST, 1 epoch, 10 rounds	0.21	0.86	0.12
15 Nodes Overfitting, MNIST, 50 epochs, 5 rounds	0.73	0.93	0.60

Nebula, the training data is split between the nodes. Therefore, it would be possible to decrease the available training data even more through setting up a scenario with more nodes, however, evaluation of such a scenario was not feasible due to lack of computational resources. [20]

This setup is aligned to the findings in [20], where the likelihood of successful attacks increases in overfitted models. To measure the effectiveness of the attack, we used the collected metrics of the attack: F1-Score, Precision and Recall. These metrics are available in the TensorBoard-frontend.

Shadow Model Based Attacks

Reference scenario:

Metric Based Attacks

As table 6.10 shows, the F1 score and precision increase dramatically in case of overfitting. The reference scenario shows poor generalization (low F1 score and precision), making it easier to exploit classification errors. Meanwhile, the overfitting scenario demonstrates improved precision and recall but introduces overfitting risks, which attackers can exploit by introducing adversarial inputs that resemble the training data.

6.5 Usability

In this section, the usability of the Nebula front end will be evaluated. For that, a person with average computer knowledge and no experience with the Nebula platform must perform a specific task: run a particular scenario on Nebula's "scenario deployment" page. Before the task, as a small tutorial to the Nebula platform, the person is asked to read the section 2. After the task, the person has to answer specific questions about the user experience and the difficulty encountered. The user has been given a Nebula running environment with all of this project's new implementations as a prerequisite.

Scenario Task

The exact scenario task is: Run a DFL scenario, five rounds, with five nodes in a fully connected topology, use the MLP model, the MNIST IID distributed dataset, run an untargeted label flipping attack on 40% of the nodes, change all labels, and use the TrimmedMean as the aggregation rule. After running the scenario, find a way to check if this attack has any impact on performance. As a given information, the local accuracy of the nodes of such a scenario without the attack lies between 0.94 and 0.96.

Questions to User Experience

This section summarizes the persons' answers to each question. The detailed transcribed answers are in Appendix C.

How easy was it to understand and find the specific parameters for the given scenario on the "scenario deployment" page?

For the first-time user, finding the specific settings to set up the scenario task took a bit of effort. Some specific settings, like the aggregation rule "TrimmedMean", were confusing for the user, as there was no information on what this setting is about, but as it was given, he just chose it. And it was also confusing for the user that some setting can be chosen in two different places, for example, the topology and number of nodes can be defined in the "Network Topology" setting but also by clicking "scenario generation".

Did you find the background information (provided in Section 2) clear and easy to follow? Were there any parts that felt ambiguous or confusing?

The background section was good to give the user an overview of the topic in general, however, some terms like "IID distributed dataset" given in the task description were not clearly explained in the background section. This confused the user a bit as he did not understand the meaning of it and went to look it up differently.

Did you face any challenges during setting up and executing the scenario? If yes, what kind of challenges?

One of the challenges that the user faced is as mentioned before in question one, that it was not always clear where to set up the specific settings, as it is possible in more than one place. Another challenge the user faced was that after running the scenario, one could not look up the settings again, so it was difficult to know if the right settings were applied. As a last challenge,

the user mentioned, that by waiting for the scenario to end it is not clear how long it will take, there is no specific information about the running scenario.

How confident were you that the configuration is correct after running the task?

The user was only about 70% sure that the right settings were applied as there is no confirmation step after running the scenario that shows all the important settings in a summary.

How straightforward was it to analyze the impact of the label-flipping attack on the model's performance? Was it straightforward where to find the performance metrics?

Finding the model's performance was easy for the user. But by first clicking on the model's performances during the scenario, it was confusing that, at the beginning, there was no data. It would have been helpful for the user to have some information about the scenario's status. Understanding the attack's impact on the model's accuracy took more effort for the user, as it was difficult to determine which metric to take. There was more than one metric for the model. Also, the final result of reading the exact accuracy of the graph was not easy, as it was not fully readable.

How would you rate the overall user experience of the Nebula platform on a scale of 1 to 5 (1 being very poor and 5 being excellent)?

The user would rate the user experience of the Nebula Platform as a 3.5. It has potential and a lot of options, but it is not very straightforward for a beginner to use as, at some points, it lacks proper communication with the user.

Based on your first interaction, do you have any suggestions for improving the Nebula platform to make tasks like this easier for new users?

The user mentioned multiple suggestions to improve the platform's usability. One suggestion is to add tooltips or short explanations for all parameters on the development page to make things more straightforward. Another is adding a confirmation or summary step after clicking the button to run the scenario. Moreover, the user mentioned that adding a real-time progress indicator during the run would make it easier to follow the scenario run.

Usability Conclusion

The Nebula platform has strong potential as a tool for decentralized federated learning. However, evaluating its usability for first-time users revealed some challenges. The current front-end settings to set up a scenario lack user guidance in several areas. Overlapping configuration options, the absence of a confirmation step after running the scenario and the lack of a real-time progress indicator during the scenario run affect the user experience. Moreover, interpreting the label-flipping attacks' performance and understanding the impact requires extra effort and is not straightforward.

For improvement, the parameter descriptions on the scenario development page can be expanded and completed, a confirmation summarizing screen can be implemented after running the scenario, and a real-time progressing bar during scenario execution can be added.

These challenges and improvements show what areas the Nebula platform can develop to improve user usability for beginners and advanced users. While the platform's backend is the main focus of improvements and developments, this user feedback shows that a user-centric design as a future work should also be considered to complete the platform.

Chapter 7

Summary and Conclusions

Federated Learning (FL) has emerged as a promising paradigm for collaborative machine learning, where multiple devices or nodes train a global model while keeping their data decentralized. This approach addresses critical challenges such as data privacy, regulatory compliance, and bandwidth constraints. However, FL is not without its unique challenges, including issues of communication efficiency, model heterogeneity, and vulnerability to adversarial attacks.

7.1 Summary

In this thesis, various parts of the Nebula framework[3] have been extended upon. This is divided into parts of 4 different authors.

For the first task, a number of node selection strategies from [4] have been implemented. Here, various selectors such as selecting all available neighbors (`AllSelector`), a random subset of neighbors always including itself (`RandomSelector`), a selector that chooses based on various telemetry data such as CPU usage, data size, bytes I/O, packet loss, latency and node age (`PrioritySelector`) have been implemented. This includes the evaluation and confirmation of correctness of said selectors.

In the second task, various poisoning attacks from [5] were introduced to Nebula. These poisoning attacks are broadly categorized into two kinds, data manipulation attacks and update manipulation attacks. In the data manipulation category are attacks such as labelflipping (both untargeted and targeted - with a specific target and without (unspecific)), as well as the FANG [14] labelflipping attack. In the update manipulation category falls the LIE [11] attack, which applies minimal updates instead of big changes. This is particularly useful in large-scale systems, where promille of accuracy often make a commercial difference. Some more aggregation rules were also implemented, such as Bulyan (a method that combines a byzantine-resilient aggregation rule with TrimmedMean.

Next in the third task, a moving target defense given by [6] was implemented. This includes a dynamic aggregator, which dynamically changes the aggregation rule each round. Another aggregator that has been implemented is the reactive aggregator, which dynamically and reactively changes the aggregation rule if malicious model updates have been detected.

Finally, in the fourth and final task, a privacy auditing component from [7] is introduced to Nebula. To analyze a scenario in which the Shadow Model Based and Metric Based Attacks work best, overfitting has been induced by increasing the number of nodes by a factor of 5, thus leading to less training data for each node.

7.1.1 Key Insights

- Using a different selection strategy is a valid choice and may be useful in increasing the system's efficiency by selecting nodes that have good performance metrics, such as computational power, latency, availability, etc.
- Most data manipulation attack gradually impact the model's performance, up until the amount of malicious nodes exceeds majority, in which case the global model accuracy drops significantly. However, despite not having a larger impact on global accuracy overall, targeting single classes is more effective with less malicious nodes. Other patterns such as the inverted confusion matrix in the FANG attack also appear once the majority is malicious.
- Minimal resource setups may lead to less-than-ideal precision, as seen in the Reference Scenario.
- Overfitting in high-resource setups improves precision and recall but increases vulnerability to targeted adversarial attacks and especially Membership Inference Attacks.

Vulnerability to Attacks

- **Targeted and Untargeted Labelflipping Attacks:** Models are mostly robust against both attacks until malicious nodes reach majority share. In this case, the global accuracy drops sharply and bottoms out at random choice once all nodes are malicious. When targeting just a single class, the accuracy on this class is reduced noticeably even before reaching majority.
- **FANG Labelflipping Attack:** The global model again exhibits the same behaviour, slightly decreasing in accuracy until the majority is malicious. Then, the confusion matrix also begins inverting.
- **Metric-Based Attacks:** These exploit inconsistencies in the model's precision and recall. For example, a high recall but low precision model is more prone to adversarial perturbations, where false positives can be easily induced.
- **Shadow Model-Based Attacks:** Shadow models mimic the behavior of the target model to infer sensitive information or to develop attack strategies. Such attacks are especially effective in overfitted models that rely too heavily on specific patterns from the training data.
- **Overfitting and Generalization:** Overfitting, while improving certain metrics, poses a significant threat to model robustness in FL. Generalization remains a key challenge in ensuring that FL models perform well across diverse and unseen data distributions.

Practical Challenges

- Communication overhead due to frequent updates between nodes.
- Node heterogeneity, where differences in computation power or data distribution among nodes can hinder convergence.
- Ensuring model robustness against adversarial samples and attacks.

7.2 Conclusion

Federated Learning represents a transformative approach to distributed machine learning, enabling collaborative intelligence while safeguarding data privacy. However, this paradigm also brings to light critical challenges that must be addressed to unlock its full potential.

With Federated Learning, many of the benefits of all the available edge devices available may be reaped, enabling collaborative intelligence while safeguarding data privacy. Yet this new paradigm does not come without its own set of challenges.

7.2.1 Key Takeaways

- Effective deployment of FL systems requires careful tuning of model configurations to achieve a balance between performance metrics like precision, recall, and F1 score.
- Robustness against adversarial and metric-based attacks should be a cornerstone of FL model development. Strategies like adversarial training, differential privacy, and regularization techniques are essential.
- Addressing practical challenges such as communication efficiency, node heterogeneity, and data imbalance will enhance the scalability and effectiveness of FL.

7.2.2 Future Directions

- Improved Defense Mechanisms: Develop adaptive strategies to counter metric-based and shadow model-based attacks.
- Add additional, more complex attacks
- Fairness and Generalization: Ensuring FL models are equitable and robust across diverse node environments and data distributions.

In summary, while Federated Learning holds significant promise, addressing its inherent challenges will determine its success in real-world applications. Through continued research and development, FL has the potential to redefine the boundaries of privacy-preserving collaborative intelligence. This thesis extends the Nebula platform, a cornerstone of DFL research by introducing various selection mechanisms, data and update manipulation attacks as well as aggregators, metric and shadow model based attacks. These components have been evaluated for correctness and are all valid.

Bibliography

[1] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y. Arcas, "Communication-Efficient Learning of Deep Networks from Decentralized Data", in *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, A. Singh and J. Zhu, Eds., ser. Proceedings of Machine Learning Research, vol. 54, PMLR, 20–22 Apr 2017, pp. 1273–1282. [Online]. Available: https://proceedings.mlr.press/v54/mcmahan17a.html.

[2] E. T. M. Beltrán, M. Q. Pérez, P. M. S. Sánchez, *et al.*, "Decentralized federated learning: Fundamentals, state of the art, frameworks, trends, and challenges", *IEEE Communications Surveys & Tutorials*, 2023.

[3] E. T. M. Beltrán, *Nebula: Framework for decentralized federated learning*, 2024. [Online]. Available: https://federatedlearning.inf.um.es/.

[4] C. Ma, "Design and prototypical implementation of the node selection strategy in federated learning", Bachelor Thesis, Sep. 2023.

[5] T.-T. Näscher, "Poisoning attack behavior detection in federated learning", Bachelor Thesis, May 2023.

[6] Z. Ye, "Mitigating poisoning attacks in decentralized federated learning through moving target defense", Master Thesis, Mar. 2024.

[7] Y. Gao, "Design and implementation of a privacy auditing component for the decentralized federated learning framework", Independent Study, Jun. 2024.

[8] W. Y. B. Lim, N. C. Luong, D. T. Hoang, *et al.*, "Federated learning in mobile edge networks: A comprehensive survey", *IEEE communications surveys & tutorials*, vol. 22, no. 3, pp. 2031–2063, 2020.

[9] M. Aledhari, R. Razzak, R. M. Parizi, and F. Saeed, "Federated learning: A survey on enabling technologies, protocols, and applications", *IEEE Access*, vol. 8, pp. 140 699–140 725, 2020.

[10] M. Moshawrab, M. Adda, A. Bouzouane, H. Ibrahim, and A. Raad, "Reviewing federated learning aggregation algorithms; strategies, contributions, limitations and future perspectives", *Electronics*, vol. 12, no. 10, p. 2287, 2023.

[11] M. Baruch, G. Baruch, and Y. Goldberg, "A little is enough: Circumventing defenses for distributed learning", *CoRR*, vol. abs/1902.06156, 2019. arXiv: 1902.06156. [Online]. Available: http://arxiv.org/abs/1902.06156.

[12] P. Blanchard, E. M. E. Mhamdi, R. Guerraoui, and J. Stainer, "Byzantine-tolerant machine learning", *CoRR*, vol. abs/1703.02757, 2017. arXiv: 1703.02757. [Online]. Available: http://arxiv.org/abs/1703.02757.

[13] E. M. E. Mhamdi, R. Guerraoui, and S. Rouault, *The hidden vulnerability of distributed learning in byzantium*, 2018. arXiv: 1802.07927 [stat.ML].

[14] M. Fang, X. Cao, J. Jia, and N. Z. Gong, "Local model poisoning attacks to byzantine-robust federated learning", *CoRR*, vol. abs/1911.11815, 2019. arXiv: 1911.11815. [Online]. Available: http://arxiv.org/abs/1911.11815.

[15] G. Rodola, *Psutil documentation*, Accessed: 2024-10-25, 2023. [Online]. Available: https://psutil.readthedocs.io/en/stable/.

[16] Google, *Protocol buffers overview*, Accessed: 2024-10-25, 2024. [Online]. Available: https://protobuf.dev/overview/.

[17] Docker, Inc., *Compose file reference - deploy*, Accessed: 2024-10-25, 2024. [Online]. Available: https://docs.docker.com/reference/compose-file/deploy/.

[18] Python Software Foundation, *The python standard library: Os — miscellaneous operating system interfaces*, Accessed: 2024-10-25, 2024. [Online]. Available: https://docs.python.org/3/library/os.html.

[19] tcconfig, *Tcconfig documentation: Tcset usage*, Accessed: 2024-10-25, 2024. [Online]. Available: https://tcconfig.readthedocs.io/en/stable/pages/usage/tcset/.

[20] S. Yeom, I. Giacomelli, A. Menaged, M. Fredrikson, and S. Jha, "Overfitting, robustness, and malicious algorithms: A study of potential causes of privacy risk in machine learning", *Journal of Computer Security*, vol. 28, no. 1, pp. 35–70, 2020. DOI: 10.3233/JCS-191362. [Online]. Available: https://par.nsf.gov/servlets/purl/10165386.

Abbreviations

AGR	Aggregation Rule
CFL	Centralized Federated Learning
DFL	Decentralized Federated Learning
FL	Federated Learning
MIA	Membership Inference Attack
MTD	Moving Target Defense

List of Figures

2.1 Centralized Federated Learning Process [10] 7
2.2 Network Topologies Overview [5] 8
2.3 User Mode setting in Nebula Scenario Deployment 10
2.4 User Mode setting in Nebula Scenario Deployment 11

4.1 An overview of the Nebula Architecture. 18

5.1 Choose the Node Selection Strategy 28
5.2 Scenario Participants 29
5.3 Add Resource Constraints to Participant 29
5.4 Frontend Attack Setup 39
5.5 Label Flipping Attack (from [14]) 39
5.6 Update Manipulation Attack (from [11]) 39
5.7 Label Flipping Attack (targeted, specific) 40
5.8 Label Flipping Attack (targeted, unspecific) 40
5.9 Label Flipping Attack (untargeted) 40
5.10 Selection of Bulyan in the Fronend 40
5.11 Frontend Configuration of the MTD Aggregators 43
5.12 Frontend Configuration of MIAs 51

6.1 TensorBoard Logs AllSelector 54
6.2 TensorBoard Logs RandomSelector 55
6.3 TensorBoard Logs PrioritySelector (Nodes without latency constraint) 57
6.4 TensorBoard Logs PrioritySelector (Nodes with latency constraint) 58

6.5 Confusion Matrices Label Flipping Attack (targeted, specific) 60
6.6 Local Accuracies Label Flipping Attack (targeted, specific) 60
6.7 Confusion Matrices Label Flipping Attack (targeted, unspecific) 62
6.8 Local Accuracies Label Flipping Attack (targeted, unspecific) 63
6.9 Confusion Matrices Label Flipping Attack (untargeted, scenario 0-5) 65
6.10 Confusion Matrices Label Flipping Attack (untargeted, scenario 5-11) 66
6.11 Local Accuracies Label Flipping Attack (untargeted, scenario 0-5) 66
6.12 Local Accuracies Label Flipping Attack (untargeted, scenario 5-11) 67
6.13 Confusion Matrices Label Flipping Attack (by [14]) 68
6.14 Local Accuracies Label Flipping Attack (by [14]) 69
6.15 TensorBoard Logs of DynamicAggregator Scenario 70
6.16 TensorBoard Logs of ReactiveAggregator Scenario 71

List of Tables

6.1 Mapping of participant name to IP-Address 53

6.2 Global model accuracy in a targeted label flipping attack with specific target, depending on the number of malicious nodes. 59

6.3 Evaluation Scenarios Label Flipping Attack (targeted, specific) 59

6.4 Global model accuracy in a targeted label flipping attack without a specific target (unspecific), depending on the number of malicious nodes. 61

6.5 Evaluation Scenarios Label Flipping Attack (targeted, unspecific) 62

6.6 Global model accuracy in a targeted labelflipping attack without a specific target (unspecific), depending on the number of malicious nodes. 64

6.7 Evaluation Scenarios Label Flipping Attack (untargeted) 64

6.8 Global model accuracy in a FANG [14] labelflipping attack, depending on the number of malicious nodes. 68

6.9 Evaluation Scenarios Label Flipping Attack (by [14]) 68

6.10 Comparison of F1 scores, recall and precision for a well-configured scenario and a scenario in which overfitting is occuring. 71

A.1 Model Summary (MNIST, MLP) 89

Listings

5.1 NSS Features extraction 22
5.2 NSS Features extraction (CPU) 22
5.3 NSS Features extraction (Networking) 22
5.4 NSS Features extraction (Loss) 22
5.5 NSS Features extraction (Data Size) 23
5.6 NSS Features extraction (Data Size) 23
5.7 NSS Features extraction (Latency) 23
5.8 Protobuf Features Message 24
5.9 Protobuf Message Wrapper 24
5.10 Sending and Receiving NSS Features Messages 24
5.11 Generating the Protobuf Message 25
5.12 NSS Features Message Handler 25
5.13 NSS Features Message Event Handler 25
5.14 NSS Features extraction 26
5.15 NSS Resource Constraints Setup (CPU) 27
5.16 NSS Resource Constraints Setup (Network) 28
5.17 Labelflipping Attack (Fang) 31
5.18 Labelflipping Attack (Targeted, Specific) 31
5.19 Labelflipping Attack (Targeted, Unspecific) 32
5.20 Labelflipping Attack (Fang) 32
5.21 Update Manipulation Attack (from [11]) 35
5.22 Executing Update Manipulation Attacks 35
5.23 Calculation of Z for Update Manipulation Attack from [11] 36
5.24 Bulyan Aggregation Rule 37
5.25 MTD DynamicAggregator 41
5.26 MTD ReactiveAggregator 42
5.27 ”Base Class MembershipInferenceAttack” 44
5.28 ”Shadow Model Based Attack” 46
5.29 ”MIA Class Metric Based” 48
6.1 Changes to PrioritySelector for Evaluations 56
6.2 Labelflipping Attack: targeted; unspecific 69
B.1 NSS Selector (Superclass) 91
B.2 NSS AllSelector 92
B.3 NSS RandomSelector Implementation from [5] 92
B.4 NSS RandomSelector 92
B.5 NSS PrioritySelector 93
B.6 Resource Constraints in Dockerfile 95
B.7 Dockerfile with CPU / Network constraints (example) 97
B.8 DataModule 98

B.9 ChangeableSubset 99
B.10 Node 100
B.11 Logs of PrioritySelector Evaluation Scenario 102
B.12 Logs of DynamicAggregator (Reactive) Evaluation Scenario Participant 0 105
B.13 Logs of DynamicAggregator (Reactive) Evaluation Scenario Participant 3 106
B.14 ”MIA Class Metric Based” 108

Appendix A

Model Summaries

Table A.1: Model Summary (MNIST, MLP)

MNIST: MLP				
	Name	Type	Output Shape	Params
0	metric	MulticlassAccuracy	[1, 10]	
1	l1	Linear	[1, 256]	200'960
2	l2	Linear	[1, 128]	32'896
3	l3	Linear	[1, 10]	1'290

235146 (235146) Total (Trainable) params
0.95 MB Model params size (estimate)
Optimizer: Adam

Appendix B

Additional Resources

nebula/core/selectors/selector.py

```python
class Selector():
    def __init__(self, config = None):
        self.config = config
        self.neighbors_list = []
        self.selected_nodes = []
        self.features = {}
        self.ages = {}

    def add_node_features(self, node, features):
        self.features[node] = features
        self.features[node]["availability"] = 1
        # ... (logging omitted for brevity)

    def get_neighbors(self):
        return self.neighbors_list

    def add_neighbor(self, neighbor):
        logging.info("[Selector] Adding Neighbor: {}".format(neighbor))
        if neighbor not in self.neighbors_list:
            self.neighbors_list.append(neighbor)

    def reset_neighbors(self):
        self.neighbors_list = []

    def node_selection(self, node):
        """To be overridden by the subclasses (selectors)"""
        pass

    def clear_selector_features(self):
        self.features = {}

    def init_age(self):
        for i in self.neighbors_list:
            self.ages[i] = 1
```

Listing B.1: NSS Selector (Superclass)

nebula/core/selectors/all_selector.py

```python
class AllSelector(Selector):
    def __init__(self, config = None):
        super().__init__(config)
        self.config = config
        logging.info("[AllSelector] Initialized")

    def node_selection(self, node):
        neighbors = self.neighbors_list.copy()
        logging.info(f"[AllSelector] available neighbors: {neighbors}")
        if len(neighbors) == 0:
            logging.error(
                "[AllSelector] Trying to select neighbors when there are no neighbors - aggregating itself only"
            )
            self.selected_nodes = [node.addr]
        else:
            self.selected_nodes = neighbors + [node.addr]
            logging.info(f"[AllSelector] selection finished - selected_nodes: {self.selected_nodes}")
        return self.selected_nodes
```

Listing B.2: NSS AllSelector

```python
class RandomSelector(Selector):
    def __init__(self, node_name="unknown", config=None):
        super().__init__(node_name, config)
        self.config = config
        self.role = self.config.participant["device_args"]["role"]

    def node_selection(self, node):
        neighbors = self.neighbors_list.copy()
        if len(neighbors) == 0:
            return None
        num_selected = max(1, int(len(neighbors) * 0.8 // 1))
        selected_nodes = np.random.choice(
            neighbors, num_selected, replace=False
        ).tolist()
        selected_nodes.append(self.node_name)
        return selected_nodes
```

Listing B.3: NSS RandomSelector Implementation from [5]

nebula/core/selectors/random_selector.py

```python
class RandomSelector(Selector):
    MIN_AMOUNT_OF_SELECTED_NEIGHBORS = 1
    MAX_PERCENT_SELECTABLE_NEIGHBORS = 0.7

    def __init__(self, config = None):
        super().__init__(config)
        self.config = config
        logging.info("[RandomSelector] Initialized")

    def node_selection(self, node):
        neighbors = self.neighbors_list.copy()
```

```
12        if len(neighbors) == 0:
13            logging.error(
14                "[RandomSelector] Trying to select neighbors when there
    are no neighbors - aggregating itself only"
15            )
16            self.selected_nodes = [node.addr]
17            return self.selected_nodes
18        logging.info(f"[RandomSelector] available neighbors: {neighbors}
    ")
19        max_selectable = math.floor(len(neighbors) * self.
    MAX_PERCENT_SELECTABLE_NEIGHBORS)
20        num_selected = np.random.randint(
21            self.MIN_AMOUNT_OF_SELECTED_NEIGHBORS,
22            max(max_selectable, self.MIN_AMOUNT_OF_SELECTED_NEIGHBORS) +
     1
23        )
24
25        selected_nodes = np.random.choice(neighbors, num_selected,
    replace = False).tolist()
26        self.selected_nodes = selected_nodes + [node.addr]
27        logging.info(f"[RandomSelector] selection finished,
    selected_nodes: {self.selected_nodes}")
28        return self.selected_nodes
```

Listing B.4: NSS RandomSelector

nebula/core/selectors/priority_selector.py

```
1 class PrioritySelector(Selector):
2     MIN_AMOUNT_OF_SELECTED_NEIGHBORS = 1
3     MAX_PERCENT_SELECTABLE_NEIGHBORS = 0.8
4     # Original Feature Weights provided in Report / Thesis
5     FEATURE_WEIGHTS = [1.0, 1.0, 1.0, 0.5, 0.5, 10.0, 3.0]
6     # Feature Weights for Testing (Latency can be changed reliably by
    virtual constraints)
7     #FEATURE_WEIGHTS = [0, 0, 0, 0, 0, 100, 0]
8
9     def __init__(self, config = None):
10        super().__init__(config)
11        self.config = config
12        FeatureWeights = namedtuple(
13            'FeatureWeights',
14            ['loss', 'cpu_percent', 'data_size', 'bytes_received', '
    bytes_sent', 'latency', 'age']
15        )
16        self.feature_weights = FeatureWeights(*self.FEATURE_WEIGHTS)
17        logging.info("[PrioritySelector] Initialized")
18
19    def node_selection(self, node):
20        neighbors = self.neighbors_list.copy()
21
22        if len(neighbors) == 0:
23            logging.error(
24                "[PrioritySelector] Trying to select neighbors when
    there are no neighbors - aggregating itself only"
25            )
26            self.selected_nodes = [node.addr]
```

```python
            return self.selected_nodes

        num_selected = max(
            self.MIN_AMOUNT_OF_SELECTED_NEIGHBORS,
            math.floor(len(neighbors) * self.MAX_PERCENT_SELECTABLE_NEIGHBORS)
        )

        availability = []
        feature_array = np.empty((7, 0))

        for neighbor in neighbors:
            if neighbor not in self.ages.keys():
                self.ages[neighbor] = 1

            # Invert CPU Percent/Latency, 0.000001 is added to avoid division by zero
            feature_list = list((self.features[neighbor]["loss"],
                                 1/(self.features[neighbor]["cpu_percent"] + 0.000001),
                                 self.features[neighbor]["data_size"],
                                 self.features[neighbor]["bytes_received"],
                                 self.features[neighbor]["bytes_sent"],
                                 1/(self.features[neighbor]["latency"] + 0.000001),
                                 self.ages[neighbor]))

            # Set loss to 100 if loss metric is unavailable
            if feature_list[0] == -1:
                feature_list[0] = 100

            logging.info(f"[PrioritySelector] Features for node {neighbor}: {feature_list}")

            availability.append(self.features[neighbor]["availability"])

            feature = np.array(feature_list).reshape(-1, 1).astype(np.float64)
            feature_array = np.append(feature_array, feature, axis = 1)

        # Normalized features
        feature_array_normed = normalize(feature_array, axis = 1, norm = 'l1')

        # Add weight to features
        weight = np.array(self.FEATURE_WEIGHTS).reshape(-1, 1)
        feature_array_weighted = np.multiply(feature_array_normed, weight)

        # Before availability
        scores = np.sum(feature_array_weighted, axis = 0)

        print_msg_box(msg=f"Scores: {dict(zip(neighbors, scores))}", title="Final NSS Scores")

```

```
        # Add availability
        final_scores = np.multiply(scores, np.array(availability))

        # Probability selection
        p = normalize([final_scores], axis = 1, norm = 'l1')

        logging.info(f"[PrioritySelector] scores: {scores}")

        # Select nodes according to thesis (weighted probability)
        selected_nodes = np.random.choice(
            neighbors, num_selected, replace = False, p = p[0]
        ).tolist()

        # Select num_selected nodes with the highest score (or the
    derived probability) for easier evaluation
        #selected_nodes = [neighbors[i] for i in np.argsort(scores)[-
    num_selected:]]

        # Update ages
        for neighbor in neighbors:
            if neighbor not in selected_nodes:
                self.ages[neighbor] = self.ages[neighbor] + 2

        # Add own node
        self.selected_nodes = selected_nodes + [node.addr]

        logging.info(f"[PrioritySelector] selection finished,
    selected_nodes: {self.selected_nodes}")

        return self.selected_nodes
```

Listing B.5: NSS PrioritySelector

nebula/scenarios.py

```
        participant_template = textwrap.dedent(
            """
            participant{}:
                image: nebula-core
                restart: no
                volumes:
                    - {}:/nebula
                    - /var/run/docker.sock:/var/run/docker.sock
                extra_hosts:
                    - "host.docker.internal:host-gateway"
                ipc: host
                privileged: true
                deploy:
                    resources:
                        limits:
                            cpus: '{}'
                command:
                    - /bin/bash
                    - -c
                    - |
                        ifconfig && echo '{} host.docker.internal' >> /
    etc/hosts {} && python3.11 /nebula/nebula/node.py {}
```

```
22                  networks:
23                      nebula-net-scenario:
24                          ipv4_address: {}
25                      nebula-net-base:
26                      {}
27          """
28          )
29          participant_template = textwrap.indent(participant_template, " "
     * 4)
30          network_template = textwrap.dedent(
31              """
32              networks:
33                  nebula-net-scenario:
34                      name: nebula-net-scenario
35                      driver: bridge
36                      ipam:
37                          config:
38                              - subnet: {}
39                                gateway: {}
40                  nebula-net-base:
41                      name: nebula-net-base
42                      external: true
43                  {}
44                      {}
45                      {}
46          """
47          )
48          ...
49          # Generate the Docker Compose file dynamically
50          services = ""
51          self.config.participants.sort(key=lambda x: x["device_args"]["
    idx"])
52          for node in self.config.participants:
53              idx = node["device_args"]["idx"]
54              path = f"/nebula/app/config/{self.scenario_name}/
    participant_{idx}.json"
55
56              tcset_cmd = ""
57              if node["resource_args"]["resource_constraint_latency"] !=
    0:
58                  tcset_cmd = f"&& tcset eth1 --delay {node['resource_args
    ']['resource_constraint_latency']} && sleep 2"
59              if node["resource_args"]["resource_constraint_cpu"] == 0:
60                  # If 0, the node shall have no CPU constraints
61                  resource_constraint_cpu = os.cpu_count()
62                  logging.info("Node has no Resource Constraint on CPU")
63              else:
64                  resource_constraint_cpu = node["resource_args"]["
    resource_constraint_cpu"]
65                  logging.info(f"Node has the following Resource
    Constraint on CPU :{resource_constraint_cpu}")
66
67              logging.info("Starting node {} with configuration {}".format
    (idx, path))
68              logging.info("Node {} is listening on ip {}".format(idx,
    node["network_args"]["ip"]))
```

```
69            # Add one service for each participant
70            if node["device_args"]["accelerator"] == "gpu":
71                ...
72            else:
73                logging.info("Node {} is using CPU".format(idx))
74                services += participant_template.format(
75                    idx,
76                    self.root_path,
77                    resource_constraint_cpu,
78                    self.scenario.network_gateway,
79                    tcset_cmd,
80                    path,
81                    node["network_args"]["ip"],
82                    "proxy:" if self.scenario.simulation and self.
    use_blockchain else "",
83                )
84        docker_compose_file = docker_compose_template.format(services)
85        docker_compose_file += network_template.format(
86            self.scenario.network_subnet, self.scenario.network_gateway,
     "proxy:" if self.scenario.simulation and self.use_blockchain else ""
    , "name: chainnet" if self.scenario.simulation and self.
    use_blockchain else "", "external: true" if self.scenario.simulation
    and self.use_blockchain else ""
87        )
88        # Write the Docker Compose file in config directory
89        with open(f"{self.config_dir}/docker-compose.yml", "w") as f:
90            f.write(docker_compose_file)
```

Listing B.6: Resource Constraints in Dockerfile

```
1 services:
2     participant0:
3         image: nebula-core
4         restart: no
5         volumes:
6             - /Users/user/Software/nebula:/nebula
7             - /var/run/docker.sock:/var/run/docker.sock
8         extra_hosts:
9             - "host.docker.internal:host-gateway"
10        ipc: host
11        privileged: true
12        deploy:
13            resources:
14                limits:
15                    cpus: '0.3'
16        command:
17            - /bin/bash
18            - -c
19            - |
20                ifconfig && echo '192.168.50.1 host.docker.internal' >>
    /etc/hosts && tcset eth1 --delay 50 && sleep 2 && python3.11 /nebula/
    nebula/node.py /nebula/app/config/nebula_DFL_02_11_2024_18_01_21/
    participant_9.json
21        networks:
22            nebula-net-scenario:
23                ipv4_address: 192.168.50.2
24            nebula-net-base:
```

```
25
26      participant1:
27          image: nebula-core
28          restart: no
29          volumes:
30              - /Users/user/Software/nebula:/nebula
31              - /var/run/docker.sock:/var/run/docker.sock
32          extra_hosts:
33              - "host.docker.internal:host-gateway"
34          ipc: host
35          privileged: true
36          deploy:
37              resources:
38                  limits:
39                      cpus: '10'
40          command:
41              - /bin/bash
42              - -c
43              - |
44                  ifconfig && echo '192.168.50.1 host.docker.internal' >>
     /etc/hosts && python3.11 /nebula/nebula/node.py /nebula/app/config/
     nebula_demo_scenario_dir/participant_1.json
45          networks:
46              nebula-net-scenario:
47                  ipv4_address: 192.168.50.3
48              nebula-net-base:
49
50  networks:
51      nebula-net-scenario:
52          name: nebula-net-scenario
53          driver: bridge
54          ipam:
55              config:
56                  - subnet: 192.168.50.0/24
57                    gateway: 192.168.50.1
58      nebula-net-base:
59          name: nebula-net-base
60          external: true
```

Listing B.7: Dockerfile with CPU / Network constraints (example)

nebula/core/datasets/datamodule.py

```
1  class DataModule(LightningDataModule):
2      def __init__(
3          self,
4          train_set,
5          train_set_indices,
6          test_set,
7          test_set_indices,
8          local_test_set_indices,
9          partition_id=0,
10         partitions_number=1,
11         batch_size=32,
12         num_workers=0,
13         val_percent=0.1,
14         label_flipping=False,
```

```
        label_flipping_config=None,
        data_poisoning=False,
        poisoned_persent=0,
        poisoned_ratio=0,
        targeted=False,
        target_label=0,
        target_changed_label=0,
        noise_type="salt",
    ):
        ...

        # Training / validation set
        tr_subset = ChangeableSubset(
            train_set,
            train_set_indices,
            label_flipping=self.label_flipping,
            label_flipping_config = self.label_flipping_config,
            data_poisoning=self.data_poisoning,
            poisoned_persent=self.poisoned_percent,
            poisoned_ratio=self.poisoned_ratio,
            targeted=self.targeted,
            target_label=self.target_label,
            target_changed_label=self.target_changed_label,
            noise_type=self.noise_type,
        )

        train_size = round(len(tr_subset) * (1 - self.val_percent))
        val_size = len(tr_subset) - train_size

        data_train, data_val = random_split(
            tr_subset,
            [
                train_size,
                val_size,
            ],
        )

        # Test set
        global_te_subset = ChangeableSubset(test_set, test_set_indices)

        # Local test set
        local_te_subset = ChangeableSubset(test_set,
    local_test_set_indices)
```

Listing B.8: DataModule

nebula/core/datasets/changeablesubset.py

```
class ChangeableSubset(Subset):
    def __init__(self,
    dataset,
    indices,
    label_flipping=False,
    label_flipping_config=None,
    data_poisoning=False,
    poisoned_persent=0,
    poisoned_ratio=0,
```

```
10     targeted=False,
11     target_label=0,
12     target_changed_label=0,
13     noise_type="salt"):
14         super().__init__(dataset, indices)
15         new_dataset = copy.copy(dataset)
16         ...
17         if self.label_flipping:
18             logging.info("[Labelflipping] Received attack: {}".format(
   self.label_flipping_config["attack"]))
19             if self.label_flipping_config["attack"] == "
   label_flipping_targeted_specific":
20                 self.dataset = labelflipping_targeted_specific(
21                     self.dataset,
22                     self.indices,
23                     self.label_flipping_config["label_og"],
24                     self.label_flipping_config["label_goal"]
25                 )
26             elif self.label_flipping_config["attack"] == "
   label_flipping_targeted_unspecific":
27                 self.dataset = labelflipping_targeted_unspecific(
28                     self.dataset,
29                     self.indices,
30                     self.label_flipping_config["label_og"]
31                 )
32             elif self.label_flipping_config["attack"] == "
   label_flipping_untargeted":
33                 self.dataset = labelflipping_untargeted(
34                     self.dataset,
35                     self.indices,
36                     self.label_flipping_config["sample_percent"]
37                 )
38             elif self.label_flipping_config["attack"] == "
   label_flipping_fang":
39                 self.dataset = labelflipping_fang(self.dataset)
40             logging.info("[Labelflipping] Dataset manipulated (attack:
   {})".format(self.label_flipping_config["attack"]))
41
42         if self.data_poisoning:
43             self.dataset = datapoison(self.dataset, self.indices, self.
   poisoned_persent, self.poisoned_ratio, self.targeted, self.
   target_label, self.noise_type)
44
45     def __getitem__(self, idx):
46         if isinstance(idx, list):
47             return self.dataset[[self.indices[i] for i in idx]]
48         return self.dataset[self.indices[idx]]
49
50     def __len__(self):
51         return len(self.indices)
```

Listing B.9: ChangeableSubset

nebula/nebula.py

```
1 async def main():
2     config_path = str(sys.argv[1])
```

```
    config = Config(entity="participant", participant_config_file=config_path)

    n_nodes = config.participant["scenario_args"]["n_nodes"]
    model_name = config.participant["model_args"]["model"]
    idx = config.participant["device_args"]["idx"]

    additional_node_status = config.participant["mobility_args"]["additional_node"]["status"]
    additional_node_round = config.participant["mobility_args"]["additional_node"]["round_start"]

    attacks = config.participant["adversarial_args"]["attacks"]
    label_flipping_config = config.participant["adversarial_args"]["label_flipping_config"]
    poisoned_persent = config.participant["adversarial_args"]["poisoned_sample_percent"]
    poisoned_ratio = config.participant["adversarial_args"]["poisoned_ratio"]
    targeted = str(config.participant["adversarial_args"]["targeted"])
    target_label = config.participant["adversarial_args"]["target_label"]
    target_changed_label = config.participant["adversarial_args"]["target_changed_label"]
    noise_type = config.participant["adversarial_args"]["noise_type"]
    iid = config.participant["data_args"]["iid"]
    partition_selection = config.participant["data_args"]["partition_selection"]
    partition_parameter = np.array(config.participant["data_args"]["partition_parameter"], dtype=np.float64)
    label_flipping = False
    data_poisoning = False
    model_poisoning = False
    if "label_flipping" in attacks:
        label_flipping = True
        poisoned_ratio = 0
        if "_targeted" in attacks:
            targeted = True
        else:
            targeted = False
    elif attacks == "Sample Poisoning":
        data_poisoning = True
        if targeted == "true" or targeted == "True":
            targeted = True
        else:
            targeted = False
    elif attacks == "Model Poisoning":
        model_poisoning = True
    else:
        label_flipping = False
        data_poisoning = False
        targeted = False
        poisoned_persent = 0
        poisoned_ratio = 0
    ...
    dataset = DataModule(
```

```
        train_set=dataset.train_set,
        train_set_indices=dataset.train_indices_map,
        test_set=dataset.test_set,
        test_set_indices=dataset.test_indices_map,
        local_test_set_indices=dataset.local_test_indices_map,
        num_workers=num_workers,
        partition_id=idx,
        partitions_number=n_nodes,
        batch_size=dataset.batch_size,
        label_flipping=label_flipping,
        label_flipping_config=label_flipping_config,
        data_poisoning=data_poisoning,
        poisoned_persent=poisoned_persent,
        poisoned_ratio=poisoned_ratio,
        targeted=targeted,
        target_label=target_label,
        target_changed_label=target_changed_label,
        noise_type=noise_type,
    )
    ...
```

Listing B.10: Node

```
18:05:02,342 - participant_2_192.168.50.4_45000 - [functions.py:22]
Selector: Received NSS Features
------------------------------
Node: 192.168.50.2:45000
CPU Usage (%): 13.6%
Bytes Sent: 23810006
Bytes Received: 16007398
Loss: 0.209886372089386
Data Size: 4878
Latency (ms): 0.13
Availability: 1

...

18:05:02,441 - participant_2_192.168.50.4_45000 - [functions.py:22]
Selector: Received NSS Features
------------------------------
Node: 192.168.50.3:45000
CPU Usage (%): 17.0%
Bytes Sent: 22898942
Bytes Received: 23751658
Loss: 0.3206459879875183
Data Size: 4878
Latency (ms): 0.12
Availability: 1

...

18:05:01,953 - participant_2_192.168.50.4_45000 - [functions.py:22]
Selector: Received NSS Features
------------------------------
Node: 192.168.50.5:45000
CPU Usage (%): 12.2%
Bytes Sent: 22911453
```

```
Bytes Received: 23805963
Loss: 0.5052706003189087
Data Size: 4878
Latency (ms): 0.48
Availability: 1

...

18:05:01,947 - participant_2_192.168.50.4_45000 - [functions.py:22]
Selector: Received NSS Features
------------------------------
Node: 192.168.50.6:45000
CPU Usage (%): 11.8%
Bytes Sent: 22896745
Bytes Received: 23766526
Loss: 0.7437791228294373
Data Size: 4878
Latency (ms): 0.05
Availability: 1

...

18:05:02,175 - participant_2_192.168.50.4_45000 - [functions.py:22]
Selector: Received NSS Features
------------------------------
Node: 192.168.50.7:45000
CPU Usage (%): 12.6%
Bytes Sent: 22932623
Bytes Received: 23796569
Loss: 0.32877373695373535
Data Size: 4878
Latency (ms): 0.06
Availability: 1

...

18:05:02,249 - participant_2_192.168.50.4_45000 - [functions.py:22]
Selector: Received NSS Features
------------------------------
Node: 192.168.50.8:45000
CPU Usage (%): 17.1%
Bytes Sent: 14066905
Bytes Received: 14040679
Loss: 0.253036767244339
Data Size: 4878
Latency (ms): 0.08
Availability: 1

...

18:05:01,950 - participant_2_192.168.50.4_45000 - [functions.py:22]
Selector: Received NSS Features
------------------------------
Node: 192.168.50.9:45000
CPU Usage (%): 12.1%
Bytes Sent: 22909055
```

```
Bytes Received: 23745516
Loss: 0.06718137860298157
Data Size: 4878
Latency (ms): 0.16
Availability: 1

...

18:05:06,027 - participant_2_192.168.50.4_45000 - [functions.py:22]
Selector: Received NSS Features
------------------------------
Node: 192.168.50.10:45000
CPU Usage (%): 57.5%
Bytes Sent: 23658203
Bytes Received: 24223180
Loss: 0.19160529971122742
Data Size: 4878
Latency (ms): 151.17
Availability: 1

...

18:05:21,530 - participant_2_192.168.50.4_45000 - [functions.py:22]
Selector: Received NSS Features
------------------------------
Node: 192.168.50.11:45000
CPU Usage (%): 74.6%
Bytes Sent: 23149186
Bytes Received: 24229371
Loss: 0.3143141269683838
Data Size: 4878
Latency (ms): 154.56
Availability: 1

...

18:05:22,268 - participant_2_192.168.50.4_45000 - [functions.py:22]
NSS features (this node)
------------------------
NSS features for round 2:
CPU Usage (%): 12.3%
Bytes Sent: 22905109
Bytes Received: 23749291
Loss: 0.42930173873901367
Data Size: 4878

...
18:05:22,271 - participant_2_192.168.50.4_45000 - [functions.py:22]
Final NSS Scores
----------------
Scores: {'192.168.50.2:45000': np.float64(10.073334904398028),
         '192.168.50.9:45000': np.float64(8.122686362567519),
         '192.168.50.5:45000': np.float64(2.661932865501917),
         '192.168.50.7:45000': np.float64(20.49253359831065),
         '192.168.50.3:45000': np.float64(10.507016996183202),
         '192.168.50.6:45000': np.float64(25.56677390405228),
```

```
147         '192.168.50.8:45000': np.float64(22.558971157521302),
148         '192.168.50.10:45000': np.float64(0.008468057817907913),
149         '192.168.50.11:45000': np.float64(0.008282153647183474)}
150
151 ...
152
153 18:05:22,271 - participant_2_192.168.50.4_45000 - [priority_selector.py
       :102]
154 [PrioritySelector] scores:
155 [2.24023557e+01 4.99629637e+00 1.04959398e+01
156 1.41730363e+01 2.68827056e+01 7.89185422e+00
157 1.31447224e+01 6.58050248e-03 6.50901092e-03]
158
159 ...
160
161 18:05:22,271 - participant_2_192.168.50.4_45000 - [priority_selector.py
       :120]
162 [PrioritySelector] selection finished, selected_nodes:
163 ['192.168.50.9:45000', '192.168.50.6:45000', '192.168.50.5:45000',
       '192.168.50.8:45000', '192.168.50.7:45000', '192.168.50.2:45000',
       '192.168.50.3:45000', '192.168.50.4:45000']
```

Listing B.11: Logs of PrioritySelector Evaluation Scenario

```
1  ...
2    Scenario information
3    --------------------
4    Trainer: Lightning
5    Dataset: MNIST
6    IID: True
7    Model: MNISTModelMLP
8    Aggregation algorithm: ReactiveAggregator
9    Node behavior: benign
10 ...
11   Defense information
12   -------------------
13   Reputation system: True
14   Dynamic topology: False
15   Dynamic aggregation: True
16   Target aggregation: FedAvg
17 ...
18 [aggregator.py:205] get_aggregation | All models accounted for,
      proceeding with aggregation.
19 [aggregator.py:213] get_aggregation | Final nodes for aggregation:
      dict_keys(['192.168.50.2:45000', '192.168.50.5:45000',
      '192.168.50.4:45000', '192.168.50.6:45000', '192.168.50.3:45000'])
20 [reactiveAggregator.py:12] [ReactiveAggregator] Initializing Aggregation
21 [engine.py:544] reputation_calculation untrusted_nodes at round 0:
      ['192.168.50.2:45000', '192.168.50.5:45000', '192.168.50.4:45000',
      '192.168.50.6:45000', '192.168.50.3:45000']
22 [engine.py:547] reputation_calculation untrusted_node at round 0:
      192.168.50.2:45000
23 [engine.py:552] reputation_calculation cossim at round 0:
      192.168.50.2:45000: 1.0
24 [lightning.py:100] Computed neighbor loss over 361 data samples
25 [engine.py:556] reputation_calculation avg_loss at round 0
      192.168.50.2:45000: 0.4594580071667830З
```

```
[engine.py:547] reputation_calculation untrusted_node at round 0:
    192.168.50.5:45000
[engine.py:552] reputation_calculation cossim at round 0:
    192.168.50.5:45000: 0.8904288411140442
[lightning.py:100] Computed neighbor loss over 361 data samples
[engine.py:556] reputation_calculation avg_loss at round 0
    192.168.50.5:45000: 8.382751703262329
[engine.py:547] reputation_calculation untrusted_node at round 0:
    192.168.50.4:45000
[engine.py:552] reputation_calculation cossim at round 0:
    192.168.50.4:45000: 0.8922047019004822
[lightning.py:100] Computed neighbor loss over 361 data samples
[engine.py:556] reputation_calculation avg_loss at round 0
    192.168.50.4:45000: 0.4201909552017848
[engine.py:547] reputation_calculation untrusted_node at round 0:
    192.168.50.6:45000
[engine.py:552] reputation_calculation cossim at round 0:
    192.168.50.6:45000: 0.9497191309928894
[lightning.py:100] Computed neighbor loss over 361 data samples
[engine.py:556] reputation_calculation avg_loss at round 0
    192.168.50.6:45000: 0.36382036035259563
[engine.py:547] reputation_calculation untrusted_node at round 0:
    192.168.50.3:45000
[engine.py:552] reputation_calculation cossim at round 0:
    192.168.50.3:45000: 0.9434241652488708
[lightning.py:100] Computed neighbor loss over 361 data samples
[engine.py:556] reputation_calculation avg_loss at round 0
    192.168.50.3:45000: 0.43781481434901554
[reactiveAggregator.py:16] [ReactiveAggregator] Detected Malicious Nodes
    : ['192.168.50.5:45000']
[reactiveAggregator.py:19] [ReactiveAggregator] Malicious Node - Using
    Dynamic Aggregator
[aggregator.py:73] [DynamicAggregator] Starting Aggregator
[dynamicAggregator.py:11] [DynamicAggregator] Initializing Aggregation
[dynamicAggregator.py:24] [DynamicAggregator] Chosen Aggregator: <class
    'nebula.core.aggregation.fedavg.FedAvg'>
[aggregator.py:73] [FedAvg] Starting Aggregator
[engine.py:454] _waiting_model_updates | Aggregation done for round 1,
    including parameters in local model.
...
```

Listing B.12: Logs of DynamicAggregator (Reactive) Evaluation Scenario Participant 0

```
...
  Scenario information
  --------------------
  Trainer: Lightning
  Dataset: MNIST
  IID: True
  Model: MNISTModelMLP
  Aggregation algorithm: ReactiveAggregator
  Node behavior: malicious
...
  Defense information
  -------------------
  Reputation system: True
  Dynamic topology: False
```

```
15   Dynamic aggregation: True
16   Target aggregation: FedAvg
17 ...
18 [aggregator.py:205] get_aggregation | All models accounted for,
     proceeding with aggregation.
19 [aggregator.py:213] get_aggregation | Final nodes for aggregation:
     dict_keys(['192.168.50.5:45000', '192.168.50.2:45000',
     '192.168.50.4:45000', '192.168.50.6:45000', '192.168.50.3:45000'])
20 [reactiveAggregator.py:12] [ReactiveAggregator] Initializing Aggregation
21 [engine.py:544] reputation_calculation untrusted_nodes at round 0:
     ['192.168.50.5:45000', '192.168.50.2:45000', '192.168.50.4:45000',
     '192.168.50.6:45000', '192.168.50.3:45000']
22 [engine.py:547] reputation_calculation untrusted_node at round 0:
     192.168.50.5:45000
23 [engine.py:552] reputation_calculation cossim at round 0:
     192.168.50.5:45000: 1.0
24 [lightning.py:100] Computed neighbor loss over 361 data samples
25 [engine.py:556] reputation_calculation avg_loss at round 0
     192.168.50.5:45000: 0.3141826655094822
26 [engine.py:547] reputation_calculation untrusted_node at round 0:
     192.168.50.2:45000
27 [engine.py:552] reputation_calculation cossim at round 0:
     192.168.50.2:45000: 0.8904288411140442
28 [lightning.py:100] Computed neighbor loss over 361 data samples
29 [engine.py:556] reputation_calculation avg_loss at round 0
     192.168.50.2:45000: 8.77032987276713
30 [engine.py:547] reputation_calculation untrusted_node at round 0:
     192.168.50.4:45000
31 [engine.py:552] reputation_calculation cossim at round 0:
     192.168.50.4:45000: 0.9467437863349915
32 [lightning.py:100] Computed neighbor loss over 361 data samples
33 [engine.py:556] reputation_calculation avg_loss at round 0
     192.168.50.4:45000: 9.669642527898153
34 [engine.py:547] reputation_calculation untrusted_node at round 0:
     192.168.50.6:45000
35 [engine.py:552] reputation_calculation cossim at round 0:
     192.168.50.6:45000: 0.9497191309928894
36 [lightning.py:100] Computed neighbor loss over 361 data samples
37 [engine.py:556] reputation_calculation avg_loss at round 0
     192.168.50.6:45000: 8.483981172243753
38 [engine.py:547] reputation_calculation untrusted_node at round 0:
     192.168.50.3:45000
39 [engine.py:552] reputation_calculation cossim at round 0:
     192.168.50.3:45000: 0.9434241652488708
40 [lightning.py:100] Computed neighbor loss over 361 data samples
41 [engine.py:556] reputation_calculation avg_loss at round 0
     192.168.50.3:45000: 8.887668450673422
42 [reactiveAggregator.py:16] [ReactiveAggregator] Detected Malicious Nodes
     : ['192.168.50.2:45000', '192.168.50.4:45000', '192.168.50.6:45000',
     '192.168.50.3:45000']
43 [reactiveAggregator.py:19] [ReactiveAggregator] Malicious Node - Using
     Dynamic Aggregator
44 [aggregator.py:73] [DynamicAggregator] Starting Aggregator
45 [dynamicAggregator.py:11] [DynamicAggregator] Initializing Aggregation
46 [dynamicAggregator.py:24] [DynamicAggregator] Chosen Aggregator: <class
     'nebula.core.aggregation.fedavg.FedAvg'>
```

```
[aggregator.py:73] [FedAvg] Starting Aggregator
[engine.py:454] _waiting_model_updates | Aggregation done for round 0,
    including parameters in local model.
...
```

Listing B.13: Logs of DynamicAggregator (Reactive) Evaluation Scenario Participant 3

nebula/addons/attacks/mia/MetricMIA.py

```
class MetricBasedAttack(MembershipInferenceAttack):
    def __init__(self, model, global_dataset, in_eval, out_eval,
    indexing_map, train_result, method_name):
        super().__init__(model, global_dataset, in_eval, out_eval,
    indexing_map)
        self.train_result = train_result
        self.method_name = method_name

    def execute_all_attack(self):
        for attr_name in dir(self):
            if attr_name.startswith("MIA") and callable(getattr(self,
    attr_name)):
                method = getattr(self, attr_name)
                method()

    def execute_specific_attack(self):
        if self.method_name == "Prediction Correctness":
            return self.MIA_correctness_attack()
        elif self.method_name == "Prediction Loss":
            return self.MIA_loss_attack()
        elif self.method_name == "Prediction Maximal Confidence":
            return self.MIA_maximal_confidence_attack()
        elif self.method_name == "Prediction Entropy":
            return self.MIA_entropy_attack()
        elif self.method_name == "Prediction Sensitivity (Jacobian
    Matrix)":
            return self.MIA_sensitivity_attack()
        else:
            raise NotImplementedError("This kind of attack is still not
    implemented.")

    def MIA_correctness_attack(self):
        def correctness_check(dataset):
            predictions, labels = dataset
            _, predicted_labels = torch.max(predictions, dim=1)
            correct_predictions = predicted_labels == labels

            return correct_predictions

        in_predictions = correctness_check(self.in_eval_pre)
        out_predictions = correctness_check(self.out_eval_pre)

        true_positives = in_predictions.sum().item()
        false_positives = out_predictions.sum().item()

        print(true_positives)
        print(false_positives)

```

```
        precision, recall, f1 = self.evaluate_metrics(true_positives,
false_positives)

        # If you want to get a micro view of in evaluation group:
        # nodes_tp_dict = self.evaluate_tp_for_each_node(in_predictions)

        return precision, recall, f1

    def MIA_loss_attack(self):
        loss_threshold = self.train_result

        self.model.eval()
        with torch.no_grad():
            for inputs, labels in self.in_eval:
                inputs = inputs.to(self.device)
                labels = labels.to(self.device)

                logits = self.model(inputs)
                losses = F.cross_entropy(logits, labels, reduction='none
')
                in_predictions = losses < loss_threshold

            for inputs, labels in self.out_eval:
                inputs = inputs.to(self.device)
                labels = labels.to(self.device)

                logits = self.model(inputs)
                losses = F.cross_entropy(logits, labels, reduction='none
')
                out_predictions = losses < loss_threshold

        true_positives = in_predictions.sum().item()
        false_positives = out_predictions.sum().item()

        precision, recall, f1 = self.evaluate_metrics(true_positives,
false_positives)

        # If you want to get a micro view of in evaluation group:
        # nodes_tp_dict = self.evaluate_tp_for_each_node(in_predictions)

        return precision, recall, f1

    def _generate_random_images(self, batch_size):
        images = []
        data_shape = self.global_dataset.train_set[0][0].shape

        if data_shape == (3, 32, 32):  # CIFAR-10 case
            height, width, channels = 32, 32, 3
            mean, std = [0.4914, 0.4822, 0.4465], [0.2471, 0.2435,
0.2616]

            transform = T.Compose([
                T.RandomCrop(32, padding=4),
                T.RandomHorizontalFlip(),
                T.ToTensor(),
                T.Normalize(mean=mean, std=std),
```

```
            ])
        else:  # gray scale images (FMNIST and MNIST)
            height, width, channels = 28, 28, 1
            transform = T.Compose([
                T.ToTensor(),
                T.Normalize((0.5,), (0.5,))
            ])

        # Generate random images
        for _ in range(batch_size):
            data = np.random.randint(0, 256, (height, width, channels), dtype=np.uint8)
            img = Image.fromarray(data.squeeze() if channels == 1 else data)
            images.append(img)

        # Apply transformations
        transformed_images = [transform(img) for img in images]

        return torch.stack(transformed_images)

    def _threshold_choosing(self, m_name):
        random_images = self._generate_random_images(batch_size=len(self.out_eval_pre[0]))
        random_dataloader = DataLoader(TensorDataset(random_images), batch_size=128, shuffle=False, num_workers=0)

        threshold = []

        self.model.eval()
        with torch.no_grad():
            for batch in random_dataloader:
                inputs = batch[0].to(self.device)

                outputs = self.model(inputs)
                probs = torch.softmax(outputs, dim=1)

                if m_name == "confidence":
                    confidences, _ = torch.max(probs, dim=1)
                    threshold.append(confidences)
                else:
                    entropies = self._compute_entropy(probs)
                    threshold.append(entropies)

        threshold_tensor = torch.cat(threshold)

        sequence = list(range(10, 100, 10)) + [95]
        threshold_percentiles = [np.percentile(threshold_tensor.cpu().detach().numpy(), i) for i in sequence]

        return threshold_percentiles  # it contains 10 percentiles as the backup thresholds

    def MIA_maximal_confidence_attack(self):
        threshold = self._threshold_choosing("confidence")

```

```
145         def maximal_confidence_check(dataset):
146             predictions, labels = dataset
147 
148             confidences, _ = torch.max(predictions, dim=1)
149 
150             return confidences
151 
152         best_f1 = 0
153         final_precison = 0
154         final_recall = 0
155 
156         in_confidences = maximal_confidence_check(self.in_eval_pre)
157         out_confidences = maximal_confidence_check(self.out_eval_pre)
158 
159         for i, thre in enumerate(threshold):
160             in_predictions = in_confidences >= thre
161             true_positives = in_predictions.sum().item()
162 
163             out_predictions = out_confidences >= thre
164             false_positives = out_predictions.sum().item()
165 
166             precision, recall, f1 = self.evaluate_metrics(true_positives
    , false_positives)
167 
168             # Update the best threshold based on F1 score
169             if f1 > best_f1:
170                 best_f1 = f1
171                 final_precison = precision
172                 final_recall = recall
173 
174         return final_precison, final_recall, best_f1
175 
176     def _compute_entropy(self, probs):
177         log_probs = torch.log(probs + 1e-6)  # Correctly use log on
    probabilities
178         entropy = -(probs * log_probs).sum(dim=1)
179         return entropy
180 
181     def MIA_entropy_attack(self):
182         threshold = self._threshold_choosing("entropy")
183 
184         def entropy_check(dataset):
185             predictions, labels = dataset
186 
187             entropies = self._compute_entropy(predictions)
188 
189             return entropies
190 
191         best_f1 = 0
192         final_precison = 0
193         final_recall = 0
194 
195         in_entropies = entropy_check(self.in_eval_pre)
196         out_entropies = entropy_check(self.out_eval_pre)
197 
198         for i, thre in enumerate(threshold):
```

```
199             in_predictions = in_entropies <= thre
200             true_positives = in_predictions.sum().item()
201 
202             out_predictions = out_entropies <= thre
203             false_positives = out_predictions.sum().item()
204 
205             precision, recall, f1 = self.evaluate_metrics(true_positives
    , false_positives)
206 
207             # Update the best threshold based on F1 score
208             if f1 > best_f1:
209                 best_f1 = f1
210                 final_precison = precision
211                 final_recall = recall
212 
213         return final_precison, final_recall, best_f1
214 
215     def _compute_jacobian_and_norm_white_box(self, inputs):
216         inputs = inputs.to(self.device)
217         inputs.requires_grad_(True)
218 
219         jacobian_matrix = jacobian(lambda x: self.model(x), inputs)
220 
221         jacobian_reshaped = jacobian_matrix.squeeze().reshape(inputs.
    size(1), -1)  # Reshape to 2D
222         l2_norm = torch.norm(jacobian_reshaped, p=2)
223         return l2_norm.item()
224 
225     def _compute_jacobian_and_norm_black_box(self, inputs, epsilon=1e-5)
    :
226         self.model.eval()
227         inputs = inputs.clone().detach().requires_grad_(True).to(
228             self.device)  # Ensure the inputs require gradients and move
     to device
229 
230         outputs = self.model(inputs)
231         num_outputs = outputs.size(1)
232         num_inputs = inputs.size(1)
233 
234         jacobian = torch.zeros(num_outputs, num_inputs).to(self.device)
235 
236         for i in range(num_inputs):
237             inputs_pos = inputs.clone().detach()
238             inputs_neg = inputs.clone().detach()
239 
240             inputs_pos[:, i] += epsilon
241             inputs_neg[:, i] -= epsilon
242 
243             outputs_pos = self.model(inputs_pos)
244             outputs_neg = self.model(inputs_neg)
245 
246             jacobian[:, i] = (outputs_pos - outputs_neg) / (2 * epsilon)
247 
248         l2_norm = torch.norm(jacobian, p=2)
249         return l2_norm
250 
```

```
 def MIA_sensitivity_attack(self):
     norms = []

     # Compute norms for in_eval_group
     for inputs, _ in self.in_eval:
         l2_norm = self._compute_jacobian_and_norm_black_box(inputs)
         norms.append(l2_norm.cpu().item())

     # Compute norms for out_eval_group
     for inputs, _ in self.out_eval:
         l2_norm = self._compute_jacobian_and_norm_black_box(inputs)
         norms.append(l2_norm.cpu().item())

     norm_array = np.array(norms)

     attack_cluster = SpectralClustering(n_clusters=6, n_jobs=-1,
affinity='nearest_neighbors', n_neighbors=19)
     y_attack_pred = attack_cluster.fit_predict(norm_array.reshape
(-1, 1))
     split = 1

     cluster_1 = np.where(y_attack_pred >= split)[0]
     cluster_0 = np.where(y_attack_pred < split)[0]

     y_attack_pred[cluster_1] = 1
     y_attack_pred[cluster_0] = 0
     cluster_1_mean_norm = norm_array[cluster_1].mean()
     cluster_0_mean_norm = norm_array[cluster_0].mean()
     if cluster_1_mean_norm > cluster_0_mean_norm:
         y_attack_pred = np.abs(y_attack_pred - 1)

     size = len(self.in_eval_pre[0])

     true_positives = np.sum(y_attack_pred[:size] == 1)
     false_positives = np.sum(y_attack_pred[size:] == 1)

     precision, recall, f1 = self.evaluate_metrics(true_positives,
false_positives)

     return precision, recall, f1
```

Listing B.14: ”MIA Class Metric Based”

Appendix C

Usability Evaluation Questions and Answers

How easy was it to understand and find the specific parameters for the given scenario on the "scenario deployment" page?

It wasn't too hard to find the settings, but some of them were a bit confusing. Like the 'Trimmed-Mean' aggregation rule—I had no idea what it was, but I just picked it because it was in the instructions. And it was weird that you could set the topology and number of nodes in two different places. I wasn't sure if I had to do both or just one.

Did you find the background information (provided in Section 2) clear and easy to follow? Were there any parts that felt ambiguous or confusing?

The background was helpful for understanding the overall idea, but some terms weren't explained well. For example, 'Partition Method: Dirichlet' didn't make much sense to me, so I had to look it up online. A quick explanation in the section would've been nice.

Did you face any challenges during setting up and executing the scenario? If yes, what kind of challenges?

Yeah, I did. Like I said earlier, it wasn't clear where to set some of the settings since you could do it in more than one place. After running the scenario, I also couldn't go back to check what I had set up, which was frustrating. And while the scenario was running, I had no idea how long it would take—there wasn't any progress bar or timer, so I was just guessing.

How confident were you that the configuration is correct after running the task?

I'd say about 70% sure. There's no confirmation step or anything to show you what you've set up before it runs, so I just hoped everything was right. A summary of the settings before running would've been really helpful.

How straightforward was it to analyze the impact of the label-flipping attack on the model's performance? Was it straightforward where to find the performance metrics?

Finding the metrics was easy enough, but when I first clicked on it, there was no data showing yet. I didn't know if something was wrong or if I just needed to wait. It would've been helpful to

have a message saying the scenario was still running. Also, figuring out how the attack affected the model wasn't very clear—there were multiple metrics, and I wasn't sure which one to use. And reading the graph to get the accuracy wasn't easy either; it was hard to see the exact value.

How would you rate the overall user experience of the Nebula platform on a scale of 1 to 5 (1 being very poor and 5 being excellent)?

I'd give it a 3.5. The platform has a lot of cool features, but it's not very beginner-friendly. It's missing some things, like explanations and feedback, that would make it easier to use.

Based on your first interaction, do you have any suggestions for improving the Nebula platform to make tasks like this easier for new users?

Yeah, definitely. Add tooltips or little explanations for the settings—that would help a lot. Also, a summary screen before running the scenario would make it easier to check if everything is correct. And having a progress bar or some kind of status update while the scenario is running would be really useful.