



University of  
Zurich<sup>UZH</sup>

# Nove Poisoning Attacks on Decentralized Federated Learning

*Runxi Cui*

*Zurich, Switzerland*

*Student ID:22-736-714*

*YunLong Li*

*Zurich, Switzerland*

*Student ID: 22-736-367*

Supervisor: Chao Feng, Dr. Alberto Huertas Celdrán

Date of Submission: July 25, 2024



# 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 All passages taken verbatim or in sense from published or unpublished writings are identified as such.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.

Yunlong Li Runxi Guo

Zürich,

---

Signature of student

25.07.2024

# Zusammenfassung

Dieser Masterprojektbericht untersucht das Design und die prototypische Implementierung von Modellvergiftungsangriffen in Distributed Federated Learning (DFL)-Systemen. Federated Learning ist ein verteiltes maschinelles Lernframework, das es mehreren Datenbesitzern ermöglicht, gemeinsam Modelle zu trainieren, ohne ihre privaten Daten zu teilen. Dieses System ist jedoch anfällig für Modellvergiftungsangriffe, bei denen böswillige Teilnehmer schädliche Aktualisierungen einführen, um die Leistung des Modells zu verschlechtern.

Der Schwerpunkt dieser Forschung liegt darin, zu untersuchen, wie die Effektivität von Vergiftungsangriffen intelligent gesteigert werden kann. Dies erfordert ein tiefes Verständnis der Arbeitsmechanismen von DFL, um potenzielle Schwachstellen zu identifizieren und auszunutzen. Das Projekt verfolgt einen vielseitigen Forschungsansatz, der die Bewertung der Sicherheit von DFL-Systemen, den Aufbau und die Validierung von Angriffsmodellen sowie die Prüfung ihrer Wirksamkeit in simulierten und realen Umgebungen umfasst.

Die Hauptbeiträge dieses Projekts umfassen die Entwicklung neuer Angriffsstrategien, die auf Kosinus-Ähnlichkeit, maximalem Eigenwert und Fisher-Winkel basieren. Diese Strategien zielen darauf ab, fortschrittliche Aggregationsalgorithmen zu umgehen und die Qualität des globalen Modells zu verschlechtern. Darüber hinaus integriert das Projekt diese Strategien in die Fedstellar-Plattform und bewertet ihre Leistung anhand verschiedener Metriken.

Die Forschungsergebnisse zeigen Schwachstellen in aktuellen DFL-Systemen auf und bieten theoretische und praktische Einblicke in den Aufbau sichererer und zuverlässigerer Modelle.

# Abstract

This master's project report explores the design and prototypical implementation of model poisoning attacks in Distributed Federated Learning (DFL) systems. Federated Learning is a distributed machine learning framework that enables multiple data owners to collaboratively train models without sharing their private data. However, this system is vulnerable to model poisoning attacks, where malicious participants introduce harmful updates to degrade the model's performance.

The focus of this research is to explore how to intelligently enhance the effectiveness of poisoning attacks. It requires a deep understanding of the working mechanisms of DFL to identify and exploit potential weaknesses. The project adopts a multi-faceted research approach, including assessing the security of DFL systems, constructing and validating attack models, and testing their effectiveness in both simulated and real environments.

The main contributions of this project include the development of new attack strategies based on cosine similarity, maximum eigenvalue, and Fisher angle. These strategies aim to bypass advanced aggregation algorithms and degrade the quality of the global model. Additionally, the project integrates these strategies into the Fedstellar platform and evaluates their performance using various metrics.

The research results reveal vulnerabilities in current DFL systems and provide theoretical and practical insights for building more secure and reliable models.

# Acknowledgments

We would like to express our sincerest gratitude to everyone who supported and assisted us throughout the process of my master's project.

First and foremost, we want to thank our supervisor Chao Feng , for his meticulous guidance and valuable advice throughout the research process. Your expertise and insights have greatly contributed to our research, and your patience and encouragement have kept us confident in the face of challenges. This thesis would not have been possible without your support.

Secondly, we would like to thank Enrique for his assistance with the Fedstellar platform. Your technical support and expertise significantly accelerated our research progress and enabled us to successfully complete the experimental part of our work.

Lastly, we would like to thank the Communication Systems Group at the Department of Informatics, University of Zurich, for their support and assistance. The hardware platform provided by the department played a crucial role in our project, allowing us to conduct our research and experiments efficiently.

# Contents

<b>Declaration of Independence</b>	<b>i</b>
<b>Abstract</b>	<b>ii</b>
<b>Acknowledgments</b>	<b>iv</b>
<b>1 introduction</b>	<b>1</b>
1.1 Introduction . . . . .	1
1.2 Motivation . . . . .	2
1.3 Description of Work . . . . .	3
1.4 Thesis Outline . . . . .	3
<b>2 Background</b>	<b>5</b>
2.1 Federated Learning Paradigms . . . . .	5
2.1.1 Federated Learning . . . . .	5
2.1.2 Decentralized Federated Learning . . . . .	6
2.1.3 The Comparison between CFL and DFL . . . . .	7
2.2 Poisoning Attack in DFL . . . . .	9
2.2.1 Attack Objectives . . . . .	9
2.2.2 Attack Methods . . . . .	10
2.2.3 Attack Scope . . . . .	11
2.2.4 Attack Timing . . . . .	12
2.3 Aggregation Methods in DFL . . . . .	13

2.3.1	Aggregation Efficiency . . . . .	13
2.3.2	Robustness . . . . .	15
2.3.3	Privacy Protection . . . . .	16
<b>3</b>	<b>Related Work</b>	<b>19</b>
3.1	Applications of Feature Metrics . . . . .	19
3.1.1	Applications of Euclidean Distance . . . . .	19
3.1.2	Applications of Cosine Distance . . . . .	20
3.2	High Computational Power in Model Attacks . . . . .	20
3.3	The Importance of Knowing Model Parameters . . . . .	21
3.4	A Min-Max Disturbance Strategy for Counteracting Attacks . . . . .	21
3.4.1	Strategy Overview . . . . .	21
3.4.2	Algorithm Implementation . . . . .	21
3.5	Comprehensive Analysis . . . . .	22
<b>4</b>	<b>Attack Design</b>	<b>24</b>
4.1	Prototype 1: Cosine Similarity-based Attack . . . . .	24
4.2	Prototype 2: Max Eigenvalue Attack . . . . .	27
4.3	Prototype 3: Fisher Angle Attack . . . . .	29
<b>5</b>	<b>Implementation</b>	<b>31</b>
5.1	Fedstellar Platform integration . . . . .	31
5.1.1	Fedstellar Platform Introduction . . . . .	31
5.1.2	MKrum Aggregator . . . . .	33
5.1.3	Fang-Krum Aggregator . . . . .	35
5.1.4	Bulyan Aggregator . . . . .	36
5.1.5	AFA Aggregator . . . . .	36
5.2	Data processing . . . . .	36
5.2.1	Dataset Selection . . . . .	36

<i>CONTENTS</i>	vii
5.2.2 Data Preprocessing . . . . .	36
5.2.3 Data Randomization Process . . . . .	39
5.2.4 Data Segmentation Strategy . . . . .	39
5.2.5 User-Specific Data Allocation . . . . .	40
5.3 Model Training . . . . .	40
5.3.1 Parameter and Environment Initialization . . . . .	41
5.3.2 Data Preparation . . . . .	41
5.3.3 Training Loop . . . . .	41
5.3.4 Performance Evaluation and Model Saving . . . . .	42
5.4 Model Attacks . . . . .	42
5.4.1 Definition of Attack and Defense Types . . . . .	42
5.4.2 Data Loading and Attack Setup . . . . .	42
5.4.3 Application of Attack Strategies . . . . .	43
5.5 Evaluation Setup . . . . .	46
<b>6 Evaluation</b>	<b>48</b>
6.1 Prototype 1: Cosine Similarity-based Attack . . . . .	48
6.2 Prototype 2: Max Eigenvalue Attack . . . . .	51
6.3 Prototype 3: Fisher Angle Attack . . . . .	51
<b>7 Summary, Conclusions and Future Work</b>	<b>54</b>
7.1 Summary . . . . .	54
7.2 Conclusion . . . . .	55
7.3 Future Work . . . . .	56
<b>Bibliography</b>	<b>57</b>
<b>List of Figures</b>	<b>59</b>
<b>List of Tables</b>	<b>60</b>



# Chapter 1

## introduction

### 1.1 Introduction

Federated Learning (FL) is an innovative distributed machine learning framework that allows multiple data owners to collaboratively train machine learning models without sharing private data. FL has been widely applied in various fields, from early smart keyboard predictions and mobile device personalization recommendations to high-value areas such as healthcare, financial fraud detection, intelligent manufacturing, and autonomous driving. In healthcare, FL enables different medical institutions to collaborate on improving disease diagnosis models without sharing sensitive patient medical records. In the financial sector, it assists banks and credit institutions in sharing the results of fraud detection models while protecting customer privacy. Additionally, with the rise of the Internet of Things (IoT) and edge computing, FL also shows great potential in handling the massive data generated by edge devices.

As technology progresses and application demands grow, FL has evolved into various forms, including but not limited to Centralized FL, Decentralized FL, and Cross-Silo FL. Each type provides customized solutions tailored to different network structures, data distribution characteristics, and privacy protection needs. For instance, Centralized FL relies on a central server to aggregate model updates, suitable for scenarios with relatively uniform data distribution; Decentralized FL eliminates the central server, instead directly exchanging information between nodes to jointly train the model, enhancing system robustness and privacy protection; Cross-Silo FL is primarily used for collaboration among organizations, capable of handling data sharing issues under different legal and policy constraints.

However, this system is susceptible to model poisoning attacks, as FL allows for the construction of a global model by locally computing model updates and sharing only these updates, not the raw data. Consequently, malicious participants can introduce harmful updates to degrade the model's performance. This paper focuses on the issue of

model poisoning attacks in the FL environment. Model poisoning attacks are a major security threat to FL, where attackers deliberately modify their model updates to reduce the performance of the global model or introduce erroneous decision-making. The covert and destructive nature of these attacks makes them particularly dangerous in real-world applications, especially in areas relying on FL for critical decision-making, such as medical diagnosis and financial fraud detection. Attackers can use a small number of malicious updates to profoundly affect the entire model's learning process, leading not only to performance degradation but also potentially to incorrect decisions, posing significant risks to user safety and business operations.

Based on the above environment, the motivation for this research emerges.

## 1.2 Motivation

Currently, the research and practical application of poisoning attacks in DFL systems is significantly lacking, especially compared to CFL systems. Existing poisoning attacks, particularly model poisoning attacks, often focus on introducing noise into the model, a method that tends to be easily identified and defended against using techniques based on similarity or loss measurement. In contrast, a more effective strategy is to inject malicious noise in specific directions determined by gradients, which can minimize changes to the model while maximizing the injection of malicious intent, effectively circumventing defenses based on similarity or loss measurements.

Facing these security challenges in DFL systems, enhancing the effectiveness of attacks presents significant challenges and offers new avenues for research. Therefore, the motivation for this project is to explore how to intelligently enhance the effectiveness of poisoning attacks in DFL systems from the attacker's perspective. This requires a deep understanding of the working mechanisms and communication protocols of DFL, as well as the precise identification and exploitation of potential weaknesses in the system.

To achieve this goal, the project adopts a multi-faceted research approach, including but not limited to the assessment of the existing DFL system's security, the construction and verification of attack models, and testing the effectiveness of attacks in both simulated and real environments. Through these research activities, it not only realize the goal of intelligently enhancing the effectiveness of poisoning attacks in DFL systems, but also help to reveal potential security vulnerabilities within DFL systems, providing a theoretical basis and practical guidance for building more secure and reliable DFL systems in the future.

Moreover, the project will also focus on the development of attack detection and defense mechanisms, providing important clues for developing a new generation of security protection measures for DFL systems. This will not only help enhance the system's security but also contribute to the healthy development of the federated learning field. Throughout this process, the project will emphasize research under ethical and legal frameworks, ensuring that the research outcomes enhance system security while also considering user privacy and data protection needs.

## 1.3 Description of Work

This paper builds on an advanced attack strategy proposed in the literature, which involves participants uploading harmful model updates to disrupt or manipulate the entire learning process[1]. During local model training, we design specific algorithms to obfuscate and bypass aggregation algorithms, thereby compromising the model's integrity. Our approach ensures that even in the presence of advanced aggregation algorithms, malicious participants can effectively degrade the quality of the global model. This methodology can provide a reference for future research aimed at enhancing the security and reliability of models, ultimately improving the robustness of federated learning systems against complex poisoning attacks.

## 1.4 Thesis Outline

Chapter 1 introduces the basic concepts of FL and its widespread applications across various fields, along with the evolution of multiple FL systems. This chapter also emphasizes the vulnerability of FL systems to model poisoning attacks and discusses the importance and motivation for studying such security issues.

Chapter 2 delves into the security challenges and background knowledge of FL, including different architectures of FL and common security threats, particularly focusing on poisoning attacks against DFL systems. Additionally, this chapter provides a detailed introduction to various data aggregation methods used in FL systems, analyzing their performance in handling heterogeneous data and resisting malicious attacks.

Chapter 3 reviews research related to model poisoning attacks in FL and DFL systems, discussing attack strategies under low and high computational power, and introduces an innovative counterattack strategy based on the min-max disturbance principle, aimed at maximizing disruption while avoiding detection.

Chapter 4 thoroughly describes the design and implementation of attack strategies, including attacks based on cosine similarity, maximum eigenvalue, and Fisher angle. Each strategy uses different mathematical and computational methods to optimize attack effectiveness while maintaining stealth and discusses specific solutions during the implementation process.

Chapter 5 showcases the integration of new defense strategies on the Fedstellar platform, detailing various stages of data processing, including data preprocessing, randomization, segmentation, and model training. This chapter also explores different types of attacks and defenses used during model training, as well as how to evaluate these attacks in real-world settings.

Chapter 6 systematically evaluates the experimental data from previous chapters, using various metrics to measure their effectiveness and compares the impact of different attacks on security. Through detailed data analysis and experimental results, this chapter discusses how to improve current attack methods and defense strategies.

Chapter 7 summarizes the findings of the research and draws conclusions on the security of federated learning systems and the effectiveness of attack strategies. This chapter also outlines future research directions, including developing more efficient attack strategies and constructing stronger defense mechanisms to enhance data security and system robustness in the federated learning environment.

# Chapter 2

## Background

In Chapter 2, this work will delve into the foundational knowledge and security challenges of DFL. This project begins by introducing the basic concepts of FL and the architecture of DFL, discussing how these systems allow multiple participants to jointly train models while protecting personal data privacy. Next, this work will analyze the various types of attacks that DFL systems may face, with a focus on poisoning attacks. This chapter also includes detailed introductions to these attacks, exploring how they specifically compromise the performance and integrity of DFL systems. Finally, this project will examine different aggregation methods, including their fundamental principles, application scenarios, strengths, and weaknesses, and their performance in handling heterogeneous data and malicious attacks. Through these analyses, this project aims to understand the current mainstream attacks and defense mechanisms, providing references for the designs in subsequent sections.

### 2.1 Federated Learning Paradigms

#### 2.1.1 Federated Learning

In recent years, as awareness of data privacy protection has increased and regulations like the General Data Protection Regulation (GDPR) in Europe have been strictly enforced, centralized data processing and storage methods have faced unprecedented challenges. This backdrop has facilitated the rise of FL, an innovative machine learning paradigm that allows multiple parties to collaboratively train a robust model while maintaining data localization. The core advantage of FL lies in its emphasis on data privacy and security; it retains data on local devices and only shares model updates (such as gradients or model parameters) instead of raw data, thus mitigating concerns about data privacy and security to some extent[2].

The process of FL typically involves several key steps: initially, a central server distributes the current global model to the participating devices. Then, these devices update the

model using their own data and send the updated portion of the model back to the central server. The central server then aggregates these updates to improve the global model[3]. This cycle continues until the model performance meets a predefined standard or specific stopping criteria. The steps include:

- **Model Initialization:** The central server initializes a global model and distributes it to all participating devices.
- **Local Training:** Each device independently trains the model using its own data and computes updates.
- **Uploading Updates:** Devices upload their model updates to the central server.
- **Model Aggregation:** The central server aggregates updates from all devices to update the global model.
- **Model Distribution:** The updated global model is redistributed to all devices, starting a new round of training.

Although FL significantly enhances the flexibility of model training and the protection of data privacy, it also introduces a series of challenges. Firstly, from a communication perspective, FL requires efficient communication protocols to handle the model updates uploaded by a large number of devices. This is particularly important in mobile network environments, where bandwidth may be limited and the stability of connections might not match that of fixed networks. Secondly, from the perspective of model aggregation, designing an efficient and fair aggregation algorithm to ensure that each device's contribution is appropriately considered, while avoiding biases caused by uneven data distribution among devices, is another important research direction. Furthermore, despite the enhanced privacy protection afforded by localized data processing in FL, there is still a certain risk of privacy leakage, such as through inference attacks using model updates.

### 2.1.2 Decentralized Federated Learning

In response to the limitations of FL, DFL has emerged as an innovative solution. It is clearly visible in Figure 2.1 the differences in system architecture between these two systems. Traditional FL systems rely on a central server to aggregate model updates from various clients, which could potentially become a point of data privacy leakage and makes the entire system vulnerable to central server failures. Moreover, the centralized architecture may lead to communication bottlenecks [4], especially when the number of participating clients is large or globally distributed.

By adopting a decentralized architecture, DFL effectively enhances system robustness and reduces dependence on a single point of failure. In this architecture, data does not need to be centralized for storage or processing. Each participating node (e.g., mobile devices or edge computing nodes) can train models locally and then exchange model updates directly with other nodes through peer-to-peer communication. This method significantly

improves data privacy protection since original data no longer needs to leave the local device.

Furthermore, DFL employs efficient peer-to-peer communication mechanisms to reduce the communication delays present in traditional FL. Through optimizing network protocols and employing data compression technologies, DFL can reduce the amount of data that needs to be transmitted while maintaining data integrity, thus accelerating the propagation speed of model updates. This efficient data transmission method is not only suitable for bandwidth-limited environments, such as mobile networks, but also enables DFL to be deployed in a wider range of application scenarios, including but not limited to healthcare, smart manufacturing, and intelligent transportation[5].

Additionally, DFL mitigates data bias issues through distributed model aggregation strategies. In traditional FL, due to the non-independent and identically distributed (Non-IID) nature of data, updates from different clients may cause the global model to be biased towards certain data patterns. DFL allows for more flexible aggregation strategies, such as weighted averaging or trust-based dynamic aggregation, by directly exchanging and aggregating model updates among devices, thus reducing the negative impact of Non-IID data.

In terms of security, DFL also faces threats such as model poisoning attacks and data tampering. Consequently, researchers have proposed various security mechanisms, including but not limited to encrypted communication, secure multiparty computation (SMC), differential privacy (DP), and blockchain technology, to ensure the security and privacy of data during transmission and processing. Through these mechanisms, DFL not only enhances the system's security but also ensures the feasibility and effectiveness of machine learning in a decentralized environment.

In summary, DFL represents a significant advancement in the field of machine learning. By overcoming several limitations of traditional FL with its decentralized architecture and efficient communication mechanisms, it not only opens up new possibilities for cross-domain applications involving sensitive data but is also expected to play an increasingly important role in many key areas as technology evolves.

### 2.1.3 The Comparison between CFL and DFL

While exploring the differences between FL and DFL, we not only observe the distinctions in their technical implementations but also understand how they optimize real-world issues in specific application domains leveraging their respective strengths.

In practical applications, although the centralized structure of FL brings convenience in management and synchronization, it may become a bottleneck in performance and expose the entire network to risks due to vulnerabilities, especially in scenarios involving large data volumes and widely dispersed participating devices. This structure requires additional protective measures, particularly in tasks involving sensitive data such as financial services and personal health information processing.

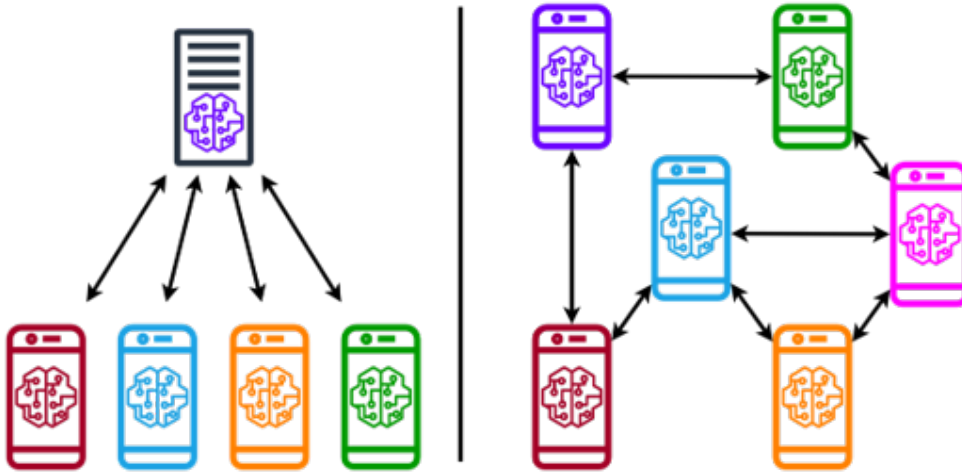


Figure 2.1: Architecture Comparison between CFL and DFL [6]

In contrast, the decentralized nature of DFL inherently offers advantages in protecting data privacy and enhancing system security. The independent operation of each node reduces centralized risks and minimizes potential attack surfaces through direct peer-to-peer communications. This model is particularly suitable for applications requiring high levels of data privacy protection, such as data cooperation between multinational corporations and cross-border medical information sharing.

Moreover, improvements in communication efficiency in DFL, achieved through advanced data compression technologies and optimized network protocols, significantly enhance data processing capabilities in bandwidth-limited environments. For example, in disaster response and resource management in remote areas, it can quickly and effectively process and analyze data from multiple sources to provide real-time decision support.

Regarding data bias issues, DFL effectively mitigates the impact of non-independent and identically distributed (Non-IID) data through more flexible model aggregation strategies. This is particularly important in fields like urban planning and traffic flow management, where data sources are diverse and complex.

In terms of security, the advancements in DFL are not only reflected in traditional encryption and data protection technologies but also include the use of blockchain technology to ensure the immutability and transparency of data and model updates. This provides possibilities for DFL applications requiring very high levels of trust, such as electoral data processing and intergovernmental information sharing.

Overall, decentralized FL represents not just a technological advancement but also a new perspective on the application models of machine learning, indicating that DFL will play a more critical role in future application scenarios sensitive to privacy and high security. As technology progresses and applications deepen, DFL is expected to demonstrate greater potential and impact in various fields such as smart manufacturing, smart cities, and environmental monitoring.

## 2.2 Poisoning Attack in DFL

In DFL, Poisoning attacks can be categorized based on different criteria. The main criteria include attack goals and attack methods. Each classification criterion has its unique characteristics and specific attack examples. Table 2.1 provides an overview of these contents.

### 2.2.1 Attack Objectives

According to the different goals of attacks, they can be divided into untargeted attacks and targeted attacks.

Untargeted attacks aim to broadly degrade the overall performance of a model without specifically targeting any particular model outputs. This type of attack disrupts the training process, rendering the final model unreliable across various tasks. For example, attackers might introduce random noise or perturbations into the training data, making it difficult for the model to effectively learn useful information, thus performing poorly during the testing phase. As discussed by Rathore and Basak [7], untargeted attacks involve introducing random disturbances such as noise into the training data. These attacks do not require knowledge of the model's structure but can significantly degrade its performance. This type of attack typically exploits the model's sensitivity to input data, using slight but precise modifications to greatly disturb the model's learning and generalization capabilities.

Another example of an untargeted attack is the generation of adversarial samples. These samples are visually or statistically similar to normal samples but are designed to cause the model to make incorrect predictions. Adversarial samples can be used not only during the testing phase but also during the training phase to disrupt the model's learning process. For instance, in natural language processing tasks, attackers can generate adversarial samples that are semantically similar to normal text but are meticulously designed to mislead the model into producing incorrect classification results. The challenge with this method lies in creating samples that are both subtle enough to go unnoticed and effective enough to activate misleading pathways during model training or testing.

Targeted attacks, on the other hand, aim to induce the model to make incorrect decisions under specific conditions or inputs, with a clear target in mind. For example, a backdoor attack is a typical example of a targeted attack. In a backdoor attack, the attacker embeds specific triggers in the training data. When the model encounters these triggers during the inference phase, it produces predefined incorrect outputs, while functioning normally under other conditions. For instance, attackers might add a small, nearly invisible mark to some images in the training data as a trigger. When the model encounters images with the same mark during the inference phase, it produces the attacker's predefined classification results, while performing normally under other conditions. Backdoor attacks are highly covert because they are only triggered under specific conditions, making them difficult to detect and defend against during the training phase, as described by Gu et al. [8].

Another example of a targeted attack is a targeted classification attack, where the attacker has a clear goal, such as causing the model to misclassify images of a specific person. During training, the attacker can add subtle perturbations to images of the target person, causing the model to consistently misclassify that person. For instance, in a facial recognition system, an attacker can add subtle noise to images of the target person. This noise may not significantly alter the visual appearance of the images but can affect the model's recognition results, preventing the model from correctly identifying the target person. These precise attacks not only pose technical challenges but may also involve more complex strategies, including precise control over the timing and manner of interference, to achieve exact manipulation of the model.

### 2.2.2 Attack Methods

According to the different methods of attack, they can be divided into data poisoning and model poisoning.

Data poisoning attacks disrupt the model's learning process by injecting erroneous or malicious samples into the training data, leading to degraded performance or incorrect outputs under specific inputs. Barreno et al. [9] early discussed how adversaries could manipulate machine learning models through malicious inputs. For example, attackers can add specific noise samples to the dataset, causing the model to be disrupted during learning, unable to correctly learn the patterns in the data, thus performing poorly during the testing phase. Biggio and Roli [10] further explored the complexities of these attacks and their impact on model robustness.

Specifically, attackers can add samples with incorrect labels to the training data, causing the model to learn incorrect patterns during training. Gu et al. [11] demonstrated the effectiveness of backdooring deep neural networks through data poisoning in their research. For example, in a spam email classification system, attackers can add normal emails that are incorrectly labeled as spam, leading to confusion in classification. This type of attack is characterized by manipulating the data to indirectly affect the model's performance, rather than directly modifying the model's parameters or updates. For example, in a medical diagnosis system, attackers can add erroneous medical records to the training data, disrupting the model's learning process, resulting in incorrect diagnoses.

Another example of a data poisoning attack is sample contamination, where attackers introduce seemingly normal but actually malicious samples into the training data, causing the model to learn these malicious features during training, thus affecting its performance during testing. For instance, in an image recognition task, attackers can add subtle perturbations to normal images, causing the model to produce incorrect results when recognizing these images. For example, in a traffic sign recognition system, attackers can add subtle noise to images of traffic signs in the training data, causing the model to produce incorrect classification results during recognition, posing a safety hazard to autonomous driving systems.

Model poisoning, on the other hand, involves directly modifying the model parameters or updates uploaded to the server to introduce a decline in performance. For instance,

in FL, each client uploads locally trained model updates to the server for aggregation. Attackers can modify these updates during this process, affecting the performance of the final global model.

Specifically, attackers can upload maliciously modified model updates to the server after locally training the model. For example, attackers can add subtle perturbations to the updates, causing the global model to perform poorly after aggregating these updates. Another example of model poisoning is parameter tampering, where attackers directly tamper with the aggregated global model parameters on the server. For instance, attackers can modify the weights of the model, causing it to produce incorrect outputs under specific inputs. A notable characteristic of this attack is that it does not require controlling multiple participants; it only requires operations on the server. For example, in a distributed financial prediction system, attackers can manipulate the model updates uploaded to the server, causing the global model to produce systematic errors in stock price predictions, thus affecting the stability of the entire financial market.

### 2.2.3 Attack Scope

Based on the scope of attacks, they can be divided into local attacks and global attacks.

Local attacks are common in DFL systems and are characterized by their limited scope, usually involving a few participants or a single client. These attacks primarily disrupt the global model through the malicious updates uploaded by the affected nodes. As described by Zhu and colleagues, although the direct impact of these attacks is limited, if an attacker can continuously or at critical moments upload a large number of malicious updates, local attacks can still have a significant impact on the global model [12]. In practical applications, for example, in a distributed FL system with multiple participants, if an attacker controls one or several key nodes, uploading tampered model updates, it will directly affect the performance of the global model. In sensitive areas such as distributed medical diagnostic systems, the nodes attacked uploading model updates with incorrect parameters could lead to severe diagnostic inaccuracies, posing a threat to patient health [13].

Furthermore, local attacks also demonstrate their destructive power in image classification and speech recognition tasks. For instance, attackers might manipulate a few nodes to upload image data with specific biases, significantly reducing the global model's accuracy in recognizing certain categories. In distributed speech recognition systems, by uploading noisy voice data, the clients controlled by attackers could cause the model to erroneously recognize certain accents or speech speeds.

Global attacks affect a broader range, involving more nodes or participants, and their coordinated attack mode causes more severe damage to the global model. Such attacks pose a greater threat to the entire distributed FL system because they can affect the overall operation of the system [14]. For example, attackers can manipulate multiple nodes to introduce systematic biases, which during the model aggregation process would lead to a significant decline in performance, making the model unreliable across various tasks. In distributed financial prediction systems, a globally coordinated attack manipulating

multiple nodes to upload malicious updates could result in systematic errors in stock price predictions, having a significant impact on the stability of the entire financial market.

The autonomous driving system is another example where global attacks have severe consequences. By controlling multiple sensor nodes to upload incorrect data, attackers could cause the central control system to develop biases in perceiving the surrounding environment, leading to erroneous or dangerous vehicle operations. Similarly, in large-scale smart home systems, global attacks might involve coordinating multiple devices to upload incorrect status information, disrupting the correct management by the central control system, thereby affecting home security and the overall user experience [15].

### 2.2.4 Attack Timing

According to the timing of attacks, they can be classified into training-time attacks and inference-time attacks.

Training-time attacks aim to manipulate the learning process during the model's training phase through data or model poisoning operations, impacting the training outcomes. A significant feature of training-time attacks is their persistent impact. Once the model is compromised during training, the final model will always carry this bias, making it difficult to correct in later stages [16]. For instance, malicious attackers might systemically shift the algorithm's decision boundaries by tampering with the training data, such as inserting fake news or misinformation, thus gradually guiding the model to develop biases unnoticed.

For example, attackers can inject malicious samples into the training data or tamper with model updates during upload, causing the final model to carry the attacker's preset biases. In natural language processing tasks, attackers can add misleading text samples to the training data, disrupting the model's learning process and causing it to produce incorrect results in classification or generation tasks [17]. The persistent impact of training-time attacks is significant: once the attack succeeds, the model will always carry these biases, and it will be extremely difficult to eliminate these effects through subsequent training or adjustments. In image processing tasks, attackers can insert specific noise images into the training dataset, causing the model to gradually deviate from the correct recognition path, thereby producing incorrect classification results in practical applications [18].

Inference-time attacks manipulate the model's prediction or decision outputs, causing the model to produce specific erroneous outputs. A notable feature of inference-time attacks is their immediacy. These types of attacks are only triggered under specific conditions, allowing the model to operate normally under other circumstances [17]. Such attacks exploit the model's uncertainties or insufficiently trained parts by using carefully designed inputs to trigger erroneous decisions. For example, introducing perturbations with specific patterns can cause the model to fail in particular scenarios, while it appears normal in routine tests.

For instance, in image recognition tasks, attackers can enter specific trigger conditions at the inference stage, causing the model to make erroneous predictions. In autonomous driving systems, attackers can add specific noise or markings on road signs, causing the vehicle

recognition system to misidentify these signs, thus leading to incorrect vehicle decisions. The stealth of inference-time attacks is significant, as these attacks only trigger under specific conditions, allowing the model to operate normally under other circumstances. In financial prediction systems, attackers can input abnormal data during specific periods, causing the model to make incorrect predictions during that time, adversely affecting market transactions. For example, in smart assistant systems, attackers can input forged voice commands under certain conditions, causing the smart assistant to perform erroneous actions, such as sending incorrect messages or making unauthorized purchases [16].

## 2.3 Aggregation Methods in DFL

In distributed FL, aggregation methods are crucial for ensuring the performance and robustness of the global model. Each participating node independently trains a local model and sends updates to a central server or aggregates them in a decentralized manner, ultimately forming the global model. An appropriate aggregation strategy can fully utilize the data from each node, enhancing the model’s accuracy and robustness while effectively mitigating the negative impact of data heterogeneity on training. Additionally, aggregation methods are vital for protecting data privacy and system security. With suitable strategies, effective training can be achieved without disclosing raw data, while also addressing attacks from malicious nodes. Different aggregation methods have varying demands on computational and communication resources, and efficient aggregation methods can minimize these costs. In FL, a reasonable aggregation strategy can ensure that each node’s contribution is fairly recognized, encouraging more nodes to participate and forming a healthy ecosystem. Therefore, the following sections will explore various aggregation methods in detail.

### 2.3.1 Aggregation Efficiency

Based on aggregation efficiency, aggregation methods can be divided into simple aggregation and complex aggregation. Simple aggregation methods typically include basic algorithms such as Federated Averaging (FedAvg). These methods are characterized by their computational simplicity and low communication cost, making them suitable for large-scale distributed systems. In the FedAvg method, each node independently trains a local model and sends the model updates to a central server. The central server performs a simple average of all node updates to generate the new global model. McMahan et al. validated the effectiveness of FedAvg under various data distributions and network conditions through experiments [14]. Although simple aggregation methods have lower computational and communication overhead, their ability to handle data heterogeneity and defend against malicious node attacks is limited.

To improve the accuracy and robustness of aggregation, complex aggregation methods introduce more computational and communication steps. For example, weighted averaging methods assign different weights to each update based on the amount of data and training performance of each node, resulting in a more accurate global model. Li et al. proposed

Table 2.1: Categories of Attacks in Distributed FL

<b>Classification Criterion</b>	<b>Attack Type</b>	<b>Description</b>	<b>Example</b>
Attack Goals	Untargeted Attacks	Disrupt the training process, degrading overall model performance	Introducing random noise into the training data [7]
	Targeted Attacks	Induce the model to make incorrect decisions under specific conditions	Backdoor attack: embedding specific triggers in the training data [8]
Attack Methods	Data Poisoning	Injecting erroneous or malicious samples into the training data, disrupting learning	Adding normal emails with incorrect labels in a spam email classification system [9]
	Model Poisoning	Directly modifying model parameters or updates uploaded to the server, degrading performance	Uploading locally trained model updates with subtle perturbations [19]
Attack Scope	Local Attacks	Involve a few participants or a single client, limited impact but can be significant if persistent or timely	Attacking key nodes in a distributed medical diagnostic system to upload incorrect parameters [13]
	Global Attacks	Involve multiple nodes, causing more severe damage through coordinated efforts	Controlling multiple sensor nodes in an autonomous driving system to upload incorrect data [15]
Attack Timing	Inference-Time Attacks	Manipulate model outputs during inference, causing specific erroneous outputs	Adding specific noise to mislead vehicle recognition systems in autonomous driving [16]
	Training-Time Attacks	Manipulate the learning process during model training through data or model poisoning	Injecting misleading samples into the training data to disrupt NLP model learning [17]

a weighted averaging aggregation method in their research, demonstrating its advantages in handling data heterogeneity [20]. This method effectively improves the robustness and accuracy of the global model by considering the quality and quantity differences of data across nodes. However, complex aggregation methods often require more computational resources and communication overhead.

Additionally, there are more advanced complex aggregation methods, such as consensus-based aggregation methods and graph-based aggregation methods. Consensus-based aggregation methods improve the aggregation effect by allowing nodes to reach a consensus locally. This approach can effectively defend against malicious node attacks but comes with higher communication and computational complexity. Xu et al. explored the application and potential advantages of blockchain technology in FL [21]. Graph-based aggregation methods utilize graph theory to achieve global model consistency through mutual communication and updates between nodes. This method is suitable for decentralized FL systems and can achieve model aggregation without relying on a central server. Li et al. demonstrated the effectiveness of graph-based aggregation methods under various network topologies in their research [22]. Although this method has advantages in decentralized scenarios, it also faces challenges related to high computational and communication costs.

Other examples include compression-based aggregation methods and dynamic aggregation methods. Compression-based aggregation methods aim to reduce communication overhead while maintaining model performance. By compressing model updates, the data transmission volume can be reduced significantly, lowering communication costs without notably impacting the global model's performance. Alistarh et al. proposed a compression-based FL aggregation method, demonstrating its effectiveness in practical applications [23]. Dynamic aggregation methods adjust aggregation strategies dynamically based on real-time node conditions (such as network latency and computational resources) to achieve optimal aggregation results. Mishchenko et al. proposed a dynamic adjustment aggregation strategy method, which can adaptively adjust under different node conditions, thereby improving the system's overall efficiency [24].

### 2.3.2 Robustness

Based on the robustness of aggregation methods, they can be divided into anti-malicious attack aggregation and fault-tolerant aggregation.

Anti-malicious attack aggregation aims to defend against erroneous updates introduced by malicious nodes. Median and Trimmed Mean are two common methods that reduce the impact of malicious updates by removing extreme values. In these methods, the server sorts the received model updates and then uses the median or the average after excluding extreme values for aggregation. This approach effectively minimizes the disruption caused by a few malicious nodes. Yin et al. analyzed the attack resistance of these methods in their study [25]. These methods are particularly useful in environments where some participating nodes may be compromised or unreliable, as they help ensure that the global model remains accurate and robust despite the presence of faulty or malicious data.

Furthermore, other advanced methods such as the FoolsGold algorithm have been proposed to enhance robustness against Sybil attacks, where multiple fake identities are used to subvert the learning process. FoolsGold reduces the weight of updates from nodes that appear to be overly similar, thus mitigating the risk of coordinated attacks by multiple malicious nodes. Fung et al. demonstrated the effectiveness of FoolsGold in FL scenarios with potential Sybil attacks [26].

Fault-tolerant aggregation focuses on generating a reliable global model in the presence of data anomalies or node failures. For instance, the Krum aggregation method selects the update that best represents the overall trend, thus enhancing the stability and security of the global model. Blanchard et al. demonstrated the effectiveness of the Krum algorithm in dealing with Byzantine faults [19]. Krum works by calculating the distance between each update and every other update, and then selecting the update with the smallest sum of distances, effectively filtering out outliers.

In addition to Krum, other fault-tolerant methods like Bulyan and Multi-Krum have been developed to further enhance robustness. Bulyan combines the strengths of Krum and median-based methods by first selecting multiple candidate updates using Krum, and then performing a median-based aggregation on these candidates to produce the final update. This layered approach provides a higher level of security and robustness, particularly in highly adversarial environments. Multi-Krum extends the basic Krum algorithm by selecting multiple updates that are close to each other in the parameter space, further mitigating the impact of isolated malicious updates.

Moreover, some fault-tolerant methods incorporate redundancy and diversity into the aggregation process. For example, Federated Averaging with Error Feedback (FedAvg-EF) introduces redundancy by sending multiple updates from each node and using error feedback to correct deviations, thereby enhancing fault tolerance. Karimireddy et al. highlighted the benefits of error feedback in improving the robustness and convergence speed of FL [27].

In summary, robust aggregation methods are essential for maintaining the integrity and reliability of FL systems, especially in environments with potential malicious attacks or node failures.

### 2.3.3 Privacy Protection

Based on the needs of privacy protection, aggregation methods can be divided into differential privacy aggregation and encrypted aggregation.

Differential Privacy Aggregation methods protect participant nodes' data privacy by introducing noise during the aggregation process. Specifically, the server adds a certain amount of random noise to each node's model updates to ensure that individual update information cannot be reverse engineered. This method effectively protects data privacy while maintaining model performance. Abadi and others have demonstrated the effectiveness of differential privacy technology in FL [28]. Differential privacy helps prevent data leakage and allows for flexible control over the balance between privacy protection strength and model accuracy by adjusting the amount of noise.

Encrypted Aggregation methods use encryption technology to protect the transmission of model updates, ensuring that data is not stolen or tampered with during transmission. Typical encrypted aggregation methods include homomorphic encryption and secure multi-party computation, which allow data to be computed on without decryption, thus ensuring data privacy. Bonawitz and others proposed an aggregation protocol based

on secure multi-party computation, demonstrating its feasibility in practical applications [29]. Moreover, homomorphic encryption technology enables direct mathematical operations on encrypted data, with the results upon decryption being the same as if operations had been performed on the plaintext data, providing a powerful tool for protecting data security in environments like cloud computing.

In addition to these methods, privacy enhancement technologies continue to evolve, such as using blockchain technology to ensure the integrity and immutability of data and model updates. The decentralized and tamper-proof nature of blockchain can be used to create a transparent and trustworthy aggregation environment, reducing centralized trust issues. Furthermore, advanced privacy protection technologies such as Multi-Party Computation (MPC) allow multiple participants to compute functions together without revealing their inputs, providing further guarantees for data privacy and security.

Aggregation methods are crucial for protecting data privacy and system security. Through an analysis of different aggregation methods, it is evident that each method has its unique advantages and applicable scenarios. Appropriate aggregation strategies can effectively train models without disclosing the original data of participating nodes. Different aggregation methods also have varying demands on computational and communication resources. Efficient aggregation methods can minimize computational and communication overhead while ensuring model quality. By employing reasonable aggregation strategies, contributions from each node can be fairly recognized, incorporating healthy nodes into FL and excluding those affected by malicious attacks.

In previous sections, this project have thoroughly explored various aggregation methods, including their basic principles, application scenarios, strengths, and weaknesses, as well as their performance in handling heterogeneous data and malicious attacks, Table 2.2 is a summary of these contents. Through these analyses, this project can provide valuable references for the subsequent design of attacks.

Table 2.2: Overview of Aggregation Methods in FL

Category	Method	Characteristics	Reference
<b>Simple Aggregation</b>	Federated Averaging (FedAvg)	Low computational and communication overhead. Suitable for large-scale systems.	[14]
<b>Complex Aggregation</b>	Weighted Averaging	Assigns weights to updates based on node data volume and training performance. Improves accuracy and robustness.	[20]
<b>Consensus-Based</b>	Consensus Methods	Allows nodes to reach a consensus locally. Defends against malicious attacks but increases computational complexity.	[21]
<b>Graph-Based</b>	Graph Theoretical Methods	Achieves model consistency through node communication. Suitable for decentralized systems.	[22]
<b>Compression-Based</b>	Data Compression	Reduces communication costs by compressing updates, maintaining performance.	[23]
<b>Dynamic</b>	Dynamic Aggregation	Adapts strategies based on real-time node conditions. Enhances system efficiency.	[24]
<b>Anti-Malicious Attack</b>	Krum, Bulyan, Median, Trimmed Mean, FoolsGold	Focus on robustness against attacks. Methods like Krum and Bulyan provide strong defenses against Byzantine faults.	[19], [26]
<b>Fault-Tolerant</b>	Multi-Krum, Adaptive Federated Averaging (AFA)	Enhances fault tolerance by selecting multiple updates or adjusting aggregation dynamically.	[19], [27]
<b>Privacy Protection</b>	Differential Privacy, Encrypted Aggregation	Protects data privacy by adding noise or using encryption. Ensures data cannot be reverse engineered.	[28], [29]

# Chapter 3

## Related Work

In past research, various methods have been utilized for model poisoning attacks on FL and DFL systems. This chapter discusses four main attack strategies. The initial strategy involves using low computational power to identify attack targets, primarily focusing on analyzing the target’s characteristics, such as Euclidean distance and cosine similarity. The second strategy uses high computational power for simulated attacks, typically employing deep learning methods for simulation, which are highly effective but consume substantial resources, thus they are less commonly used in regular attacks. The third strategy involves conducting large-scale attacks using computational power when the model parameters are known; within the deep learning framework, attackers can quickly calculate the modifications needed to implement the attack by understanding the model parameters and related information. Additionally, there is also a strategy using brute force attacks.

This chapter also details a novel counterattack strategy using the min-max disturbance principle, aimed at maximizing disruption while avoiding detection in distributed learning environments.

Finally, the document proposes several research directions, focusing on developing more effective attack strategies using minimal information and computational resources. It suggests transitioning from parameter-based to gradient-based communication in FL to enhance attack effectiveness.

### 3.1 Applications of Feature Metrics

#### 3.1.1 Applications of Euclidean Distance

In the field of machine learning, calculating distances and similarities is a common requirement. For example, in classification tasks, it is often necessary to measure the similarity between different samples. A common method includes calculating the distance

between samples. For instance, in the k-means clustering algorithm, the distance formula is required to determine the distance between a sample and the cluster center; in kNN classification, the similarity between a sample and known categories is calculated to determine the sample's category affiliation.

In model attack scenarios, distance metrics are often used as a reference index to assess differences between models. Due to structural differences between FL and DFL, often modifications in attack strategies are required. For example, in the study published by Virat Shejwalkar at the NDSS conference in 2021[1], Euclidean distance was utilized for attacks, a method widely used in FL attacks due to its low computational complexity. The mathematical expression for Euclidean distance is:

$$d(p, q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \dots + (p_n - q_n)^2}$$

In model poisoning attacks, malicious clients directly manipulate the model updates shared with the central server. The comprehensive model poisoning attack models proposed in utilize the knowledge shared by benign clients and the AGR algorithm used by the server. The attack process includes: calculating a benign reference set using leaked benign model samples, calculating a malignant disturbance vector, and generating malicious model updates in the most disruptive direction to evade detection by robust aggregation algorithms.

### 3.1.2 Applications of Cosine Distance

Cosine distance is a method used in machine learning to measure the difference between two vectors by calculating the cosine of the angle between them. The cosine ranges from [-1,1], with higher values indicating a smaller angle between the vectors and lower values indicating a larger angle. When two vectors are in the same direction, the cosine reaches its maximum value of 1; when the directions are completely opposite, it reaches its minimum value of -1. Compared to Euclidean distance, cosine distance has different calculation methods and characteristics, making it suitable for different data analysis scenarios. For example, Euclidean distance is suitable for analyses that need to reflect differences from the numerical size of dimensions, while cosine distance is more applicable for distinguishing vector differences based on direction, important in analyses such as user content rating similarity, as it can correct for potential differences in metrics among users.

In existing model poisoning attacks, cosine similarity is often used to attack local models with the aim of reducing performance while maintaining a high cosine similarity. As described in [30], the approach involves generating model updates in the opposite direction of good model updates but maintaining a similar cosine value, thus executing the attack.

## 3.2 High Computational Power in Model Attacks

In model attacks, attacks requiring high computational resources generally rely on deep learning techniques. The resource consumption of these attacks is similar to that involved

in training and is easily detectable. These attacks can not only achieve denial-of-service (DOS) but also execute fine-grained control attacks. As described in [31], this method allows for fine-tuning of the malicious model to achieve minimal disturbance without relying on the knowledge of the aggregation methods used by the main server, improving the stealth and durability of the attack method.

### 3.3 The Importance of Knowing Model Parameters

If attackers understand model parameters, they can perform reverse engineering through mathematical deduction. For instance, the [32] paper mentions that by knowing all model parameters, other characteristics of model parameters can be calculated, thus increasing the transparency of the model. The attack process includes evaluating the importance of parameters, calculating the malicious enhancement coefficient, and generating the malicious update. Initially, attackers use Fisher information obtained from the local datasets of compromised participants to evaluate the importance of each model parameter. Then, based on the importance of the parameters, key parameters are chosen for poisoning, and an enhancement coefficient is calculated to amplify the impact of the malicious update, ultimately generating the malicious update and uploading it to the parameter server to deviate the global model from its optimal path and evade server defense mechanisms.

### 3.4 A Min-Max Disturbance Strategy for Counteracting Attacks

In DFL systems, Virat Shejwalkar and colleagues proposed an innovative counterattack strategy [1], which uses the principle of min-max to design malicious updates to maximize disruption to the model without triggering detection mechanisms. This strategy has now become the SOTA in defending against attacks in DFL systems.

#### 3.4.1 Strategy Overview

The main objective of this method is to maximize the disruption of model parameters within distributed learning environments, ensuring that the disturbance does not cause the maximum deviation of model updates to exceed that of normal operations. The core of this strategy is to degrade the model's performance through carefully crafted malicious updates without activating the system's anomaly detection mechanisms.

#### 3.4.2 Algorithm Implementation

The specific implementation of the algorithm involves several key steps:

1. **Input Definition:** The algorithm takes multiple inputs, including all normal updates (`all_updates`), reference model parameters (`model_re`), the number of attackers (`n_attackers`), and the type of disturbance (`dev_type`). These inputs define the operating environment and basic direction of the disturbance.
2. **Disturbance Calculation:** Depending on the chosen type of disturbance, the algorithm employs different strategies to compute the disturbance vector.
  - `unit_vec`: Calculation based on the unit vector of the reference model parameters to ensure that the direction of disturbance is opposite to that of normal updates.
  - `sign`: Utilizes the sign of the reference parameters to determine the direction of the disturbance.
  - `std`: Adjusts the intensity of the disturbance based on the standard deviation of all updates.
3. **Optimal Disturbance Factor Search:** The algorithm iteratively adjusts the size of the disturbance factor (`lamda`) and sets a threshold (`threshold_diff`) to determine when to stop iterating. The goal is to find a disturbance factor such that the maximum deviation with malicious updates does not exceed the maximum deviation without disturbances.
4. **Malicious Update Output:** Upon achieving optimal disturbance conditions, the algorithm generates and outputs malicious updates that maximize model disruption without increasing the detection threshold.

### 3.5 Comprehensive Analysis

This section reviews the literature on model poisoning attacks and summarizes key insights from the following perspectives:

- In the field of deep learning, the deeper the understanding of model parameters and related information, the richer the computational information available to assist in attacks.
- In conducting model attacks, there may be various deep learning evaluation parameters, all of which can serve as effective references.
- The current industry need is to develop an algorithm that can effectively enhance attack effects without relying on substantial computational resources and information.
- The algorithm mentioned in section 3.4 offers a method for effectively identifying and defending against potential malicious attacks while maintaining the security of machine learning models, particularly in distributed learning environments where it is crucial to preserve data integrity and model robustness.

Based on these insights, this paper proposes the following research directions:

- How to effectively attack using minimal information in the transfer of federal learning is a research topic worth exploring. Therefore, this study suggests using model gradient information instead of model parameters for communication in FL, because model parameters can directly derive the corresponding Hessian matrix, as mentioned in Section 3. Utilizing model gradient information for communication can achieve attacks with less information.
- Most current defense measures are based on Euclidean distance, with less utilization of cosine similarity. If attacks are initiated from the perspective of cosine similarity, it may circumvent existing aggregation and detection mechanisms.
- Currently, the strategies for attacks with different characteristics mainly rely on self-developed methods. In light of this, this study aims to explore a new attack method that can realize aimless model poisoning attacks around different aggregation functions without relying on extensive computational resources.

# Chapter 4

## Attack Design

In this chapter, in designing the Architecture of the attack system, we will focus on analyzing the SOTA paper with the best effect at present, and then explain how to design an idea that can use mathematics to explain why such an attack will have an impact. First, we will use COS similarity to modify the discovery of the attack, second, we will use the discovery of numerical summary, and finally, we will use mathematics to deduce and calculate. Finally, a set of methods that can be used to explain why to attack in this way is constructed.

### 4.1 Prototype 1: Cosine Similarity-based Attack

From the analysis in previous chapters, it is evident that manipulating model characteristics and their behavior in attack scenarios can significantly influence the effectiveness of attacks. Based on this feature, we can design a new cosine similarity attack strategy that revolves around the concept of leveraging model characteristics to enhance the impact of attacks, as shown in Algorithm 1. This algorithm implements the attack by adjusting different target thresholds. It minimizes the cosine angle difference from the average update direction by adjusting malicious updates. Here is a detailed introduction to these two functions:

- **calculate\_cos\_angle(a, b)**: This function is used to calculate the cosine angle between two tensors  $A$  and  $B$ . The function first flattens the input tensor (using `view(-1)`), then uses `torch.dot()` to calculate the dot product of the two tensors, and then divides it by the product of the norms of the two tensors to get the cosine angle. This cosine value reflects the similarity of two vector directions, and the closer the value is to 1, the more similar the directions are.
- **attack\_cos\_value(all\_updates, model\_re, n\_attackers, dev\_type='unit\_vec')**: This function realizes the attack strategy against the model, and its goal is to find a malicious update, so that the cosine angle between the update and the average direction of all updates is close to a certain `target_angle`. The parameters are as follows:

- `all_updates`: updates for all participants.
- `model_re`: reference model parameters.
- `n_attackers`: number of attackers.
- `dev_type`: Specifies the type of disturbance (unit vector, sign, or standard deviation).

According to the selection of `dev_type`, the calculation method of disturbance deviation is different:

- `unit_vec`: Use the unit vector of the reference model parameters as the perturbation.
- `sign`: Use the symbol of the reference model parameter as the perturbation.
- `std`: Use all updated standard deviations as perturbations.

Firstly, the algorithm calculates all updated average vectors and normalizes them into `average_direction`. Then, the algorithm calculates the cosine angle of each update and the average direction, and sorts these angles. According to the sorted angle differences, the algorithm sets a target angle `target_angle`. This target angle is selected based on the specific logic of angle difference to ensure that the malicious update and the average update direction are as close as possible. Then, the algorithm tries to find malicious updates that make the angle of malicious updates as close as possible to the target angle by iteratively adjusting the parameter `lamda`. During the iteration process, the algorithm will continuously calculate the cosine angle of the current malicious update and the average direction, and adjust the value of `lamda` according to the comparison result with the target angle. Finally, the algorithm outputs the adjusted malicious update. This update tries to destroy the performance of the model without significantly increasing the deviation from the average update direction.

Generally speaking, this code shows how to carry out antagonistic attacks by carefully controlling the update direction, aiming at maintaining the integrity of data and the robustness of the model in the distributed learning environment. From the code's `target_angle` value, it is adjusted from different results, and different results have different performances (good and bad). From the performance of these results, it can be seen that different cos target values will lead to different effects, and different aggregation functions will also have different performances. The specific discussion is discussed in the Evaluation chapter. However, based on this algorithm, we derive the following questions:

1. Why do different target angles lead to this different effect?
2. Why is the angle difference between each vector and the average vector different in a certain area, instead of the opposite, the better, but a good effect will appear under a certain threshold?
3. Is there a way to explain the above phenomenon?

**Algorithm 1** Cosine Similarity-based Attack

---

```

1: function CALCULATE_COS_ANGLE( $a, b$ )
2:    $a\_flat \leftarrow a.view(-1)$ 
3:    $b\_flat \leftarrow b.view(-1)$ 
4:    $cos\_angle \leftarrow torch.dot(a\_flat, b\_flat) / (torch.norm(a\_flat) * torch.norm(b\_flat))$ 
5:   return  $cos\_angle$ 
6: end function
7: function ATTACK_COS_VALUE( $all\_updates, model\_re, n\_attackers, dev\_type$ )
8:   if  $dev\_type = 'unit\_vec'$  then
9:      $deviation \leftarrow model\_re / torch.norm(model\_re)$ 
10:  else if  $dev\_type = 'sign'$  then
11:     $deviation \leftarrow torch.sign(model\_re)$ 
12:  else if  $dev\_type = 'std'$  then
13:     $deviation \leftarrow torch.std(all\_updates, 0)$ 
14:  end if
15:   $lamda \leftarrow torch.Tensor([10.0]).float().cuda()$ 
16:   $threshold\_diff \leftarrow 1e - 5$ 
17:   $lamda\_fail \leftarrow lamda$ 
18:   $lamda\_succ \leftarrow 0$ 
19:   $angles \leftarrow []$ 
20:   $mean\_vector \leftarrow torch.mean(all\_updates, dim = 0)$ 
21:   $average\_direction \leftarrow mean\_vector / torch.norm(mean\_vector)$ 
22:  for  $update$  in  $all\_updates$  do
23:     $angle \leftarrow CALCULATE\_COS\_ANGLE(update, average\_direction)$ 
24:     $angles.append(angle)$ 
25:  end for
26:   $sorted\_angles, indices \leftarrow torch.sort(torch.tensor(angles), descending = True)$ 
27:  if  $(sorted\_angles[0] - sorted\_angles[-1]) > 0.5$  then
28:     $target\_angle \leftarrow sorted\_angles[4]$ 
29:  else if  $(sorted\_angles[0] - sorted\_angles[-1]) \leq 0.15$  and  $(sorted\_angles[0] - sorted\_angles[-1]) \geq 0.05$  then
30:     $target\_angle \leftarrow sorted\_angles[-1] - 0.1$ 
31:  else if  $(sorted\_angles[0] - sorted\_angles[-1]) < 0.05$  then
32:     $target\_angle \leftarrow sorted\_angles[1] - 0.2$ 
33:  else
34:     $target\_angle \leftarrow sorted\_angles[1]$ 
35:  end if
36:  while  $torch.abs(lamda\_succ - lamda) > threshold\_diff$  do
37:     $mal\_update \leftarrow model\_re - lamda * deviation$ 
38:     $current\_angle \leftarrow CALCULATE\_COS\_ANGLE(mal\_update, average\_direction)$ 
39:    if  $current\_angle > target\_angle$  then
40:       $lamda\_succ \leftarrow lamda$ 
41:       $lamda \leftarrow lamda + lamda\_fail / 2$ 
42:    else
43:       $lamda \leftarrow lamda - lamda\_fail / 2$ 
44:    end if
45:     $lamda\_fail \leftarrow lamda\_fail / 2$ 
46:  end while
47:   $mal\_update \leftarrow model\_re - lamda\_succ * deviation$ 
48:  return  $mal\_update$ 
49: end function

```

---

## 4.2 Prototype 2: Max Eigenvalue Attack

In attempting to determine a cosine target value, we utilize a mathematical method to explain the phenomenon observed during parameter debugging. We extract the angle value using the eigenvector associated with the maximum eigenvalue 3. In data science, *feature vectors* are vectors that describe the direction of data set changes, and *feature values* indicate the magnitude or importance of these changes. The eigenvector corresponding to the maximum eigenvalue points to the direction of maximum variance within the data set. In statistics and machine learning, variance often correlates directly with information content; higher variance generally signifies greater information content, making the data characteristics in this direction most crucial.

In the context of an adversarial attack, the eigenvector corresponding to the largest eigenvalue represents the direction of largest variance. Attacking this direction can significantly disrupt the decision-making process of the model. For example, an attacker might slightly alter the input data of an image recognition model along the direction of largest variance to more effectively influence the model's output, thereby increasing the likelihood of a successful attack. Since the eigenvector of the largest eigenvalue indicates where data changes are most significant, disturbances in this direction are more likely to impact model outputs and are harder to detect and counter with simple defensive measures.

To validate the effectiveness of the cosine algorithm, we have designed Algorithms 2 and 3. These algorithms provide the framework and computational steps necessary to demonstrate and assess the impact of attacking along the direction of maximum variance, thus proving the accuracy of the cosine target value method.

---

### Algorithm 2 Adversarial Attack Algorithm

---

Calculate the target angle.

Use `torch.mean` to compute the mean vector (*mean\_vector*) of all updates.

Normalize *mean\_vector* to obtain the unit vector (*average\_direction*).

**for** each update **do**

    Calculate its cosine angle with *average\_direction*.

**end for**

Sort the angles, compute the eigenvalues and eigenvectors of the angle matrix  $C$ , and identify the eigenvectors related to the minimum eigenvalues.

Realize the attack algorithm using a sigmoid function to process the target angle.

Adjust the *model\_re* vector based on the target angle value and direction of disturbance.

**repeat**

    Adjust *lambda* cyclically until the difference between iterations is less than *threshold\_diff*.

    Generate a malicious update according to the current *lambda* value.

    Calculate its cosine angle with *average\_direction* and adjust *lambda* accordingly.

**until** condition met

**return** the last calculated malicious update (*mal\_update*).

---

**Algorithm 3** max eigenvalue Attack Algorithm

---

```

1: Input: all_updates, model_re, n_attackers, dev_type
2: Output: ma_update
3: if dev_type = 'unit_vec' then
4:   deviation  $\leftarrow$  model_re / torch.norm(model_re)
5: else if dev_type = 'sign' then
6:   deviation  $\leftarrow$  torch.sign(model_re)
7: else if dev_type = 'std' then
8:   deviation  $\leftarrow$  torch.std(all_updates, 0)
9: end if
10: lamda  $\leftarrow$  torch.Tensor([10.0]).float().cuda()
11: threshold_diff  $\leftarrow$  1e-5
12: lamda_fail  $\leftarrow$  lamda
13: lamda_succ  $\leftarrow$  0
14: mean_vector  $\leftarrow$  torch.mean(all_updates, dim=0)
15: average_direction  $\leftarrow$  mean_vector / torch.norm(mean_vector)
16: angles  $\leftarrow$  []
17: for update in all_updates do
18:   angle  $\leftarrow$  calculate_cos_angle(update, average_direction)
19:   angles.append(angle)
20: end for
21: sorted_angles, indices  $\leftarrow$  torch.sort(torch.tensor(angles), descending=True)
22: angles  $\leftarrow$  torch.tensor(angles).unsqueeze(0).cuda()
23: C  $\leftarrow$  torch.matmul(angles.T, angles)
24: eigenvalues, eigenvectors  $\leftarrow$  torch.linalg.eig(C)
25: real_eigenvalues  $\leftarrow$  eigenvalues.real
26: eigenvectors  $\leftarrow$  eigenvectors.real
27: max_eigenvalue_index  $\leftarrow$  torch.argmax(real_eigenvalues)
28: max_eigenvector  $\leftarrow$  eigenvectors[:, max_eigenvalue_index]
29: min_eigenvector_index  $\leftarrow$  torch.argmin(max_eigenvector)
30: target_angle_value  $\leftarrow$  F.sigmoid(angles.squeeze()[min_eigenvector_index])
31: model_re  $\leftarrow$  model_re - target_angle_value * deviation
32: target_angle  $\leftarrow$  sorted_angles[0]
33: while torch.abs(lamda_succ - lamda) > threshold_diff do
34:   ma_update  $\leftarrow$  model_re - lamda * deviation
35:   current_angle  $\leftarrow$  calculate_cos_angle(ma_update, average_direction)
36:   if current_angle > target_angle then
37:     lamda_succ  $\leftarrow$  lamda
38:     lamda  $\leftarrow$  lamda + lamda_fail / 2
39:   else
40:     lamda  $\leftarrow$  lamda - lamda_fail / 2
41:   end if
42:   lamda_fail  $\leftarrow$  lamda_fail / 2
43: end while
44: ma_update  $\leftarrow$  model_re - lamda_succ * deviation
45: return ma_update

```

---

### 4.3 Prototype 3: Fisher Angle Attack

However, the effect obtained by using eigenvalue calculation cannot reproduce the similarity with cosine. Therefore, we continue to explore new methods. We retrieved many model parameters from the referenced paper to calculate the Hessian Matrix. However, because we use gradients, and the gradient only represents the first derivative of  $X$ , we do not have the conditions for model parameters needed to calculate the second derivative of  $X$ . Therefore, we use the Fisher information matrix instead. The Fisher information matrix is used to measure the information that the unknown parameter  $\theta$  of the random variable  $X$  conveys about its own random distribution function. The greater the Fisher information, the greater the variance of the Score function, and thus, it represents more information and increases the accuracy of parameter estimation.

Fisher information matrix is a key concept in statistical estimation, which is used to describe the local curvature of parameter space or the sensitivity of model parameters. Suppose we have a probability model  $p(x|\theta)$  that depends on the parameter  $\theta$ , and our goal is to estimate the parameter  $\theta$ :

$$\max_{\theta} p(x|\theta)$$

by maximizing the likelihood function, and define the score function  $s(\theta)$  The expected value of nabla

$$s(\theta) = \nabla_{\theta} \log p(x|\theta)$$

score function is 0:

$$\mathbb{E}_{p(x|\theta)}[s(\theta)] = 0$$

Fisher information matrix

$$F(\theta) = \mathbb{E}_{p(x|\theta)} [(s(\theta) - 0)(s(\theta) - 0)^T] = \mathbb{E}_{p(x|\theta)} [\nabla_{\theta} \log p(x|\theta) \nabla_{\theta} \log p(x|\theta)^T]$$

This means that Fisher information matrix is a sensitivity measure of model parameters, and the influence of parameter adjustment on model prediction can be better understood by estimating this matrix. Therefore, the algorithm uses fisher matrix to calculate, and finally our algorithm uses this matrix to calculate the new gradient optimization logic.

First of all, we adopt the natural gradient method, whose main feature is that it uses the Riemann metric of Fisher information matrix to adjust the gradient direction, which takes into account the manifold structure of parameter space, so it is mathematically established to use this method to derive the new gradient of model attack.

**Algorithm 4** Fisher Angle Attack

---

```

1: procedure FISHER ANGLE ATTACK(all_updates, model_re, n_attackers, dev_type)
2:   if dev_type = 'unit_vec' then
3:     deviation  $\leftarrow$  model_re / ||model_re||
4:   else if dev_type = 'sign' then
5:     deviation  $\leftarrow$  sign(model_re)
6:   else if dev_type = 'std' then
7:     deviation  $\leftarrow$  std(all_updates, 0)
8:   end if
9:   lamda  $\leftarrow$  Tensor([10.0]).float().cuda()
10:  threshold_diff  $\leftarrow$  1e - 5
11:  lamda_fail  $\leftarrow$  lamda
12:  lamda_succ  $\leftarrow$  0
13:  mean_vector  $\leftarrow$  mean(all_updates, 0)
14:  fim_diag_mean  $\leftarrow$  mean_vector.2
15:  app  $\leftarrow$  []
16:  for data_e in all_updates do
17:    fim_diag_approx  $\leftarrow$  data_e.2
18:    target_angle  $\leftarrow$  calculate_cos_angle(fim_diag_approx, fim_diag_mean)
19:    app.append(target_angle)
20:  end for
21:  target_angle  $\leftarrow$  calculate_cos_angle(all_updates[app.index(min(app))], mean_vector)
22:  model_re  $\leftarrow$  model_re + 0.01  $\times$  (-fim_diag_mean  $\times$  mean_vector)
23:  while |lamda_succ - lamda| > threshold_diff do
24:    mal_update  $\leftarrow$  model_re - lamda  $\times$  deviation
25:    current_angle  $\leftarrow$  calculate_cos_angle(mal_update, mean_vector)
26:    if current_angle > target_angle then
27:      lamda_succ  $\leftarrow$  lamda
28:      lamda  $\leftarrow$  lamda + lamda_fail / 2
29:    else
30:      lamda  $\leftarrow$  lamda - lamda_fail / 2
31:    end if
32:    lamda_fail  $\leftarrow$  lamda_fail / 2
33:  end while
34:  mal_update  $\leftarrow$  model_re - lamda_succ  $\times$  deviation
35:  return mal_update
36: end procedure

```

---

# Chapter 5

## Implementation

This chapter will use pseudo-code to explain the algorithm described in the fourth chapter, and will introduce how to achieve the corresponding effect in the following four steps: first, system design, introducing the system configuration and the corresponding call library when deploying the system; The second is the preparation of data, which introduces what data to use for training and testing; The third is the description of the algorithm, introducing the principle and design idea of the algorithm. The fourth is about the overall training.

### 5.1 Fedstellar Platform integration

#### 5.1.1 Fedstellar Platform Introduction

Fedstellar is an advanced platform designed to simulate and implement decentralized FL. It allows multiple nodes to collaboratively train machine learning models without centralizing data, thus ensuring data privacy and adopting a decentralized approach. Fedstellar supports a variety of topological structures and aggregation methods, capable of adapting and operating under different network conditions and data distributions.

Within the Fedstellar platform, users can create specific simulation scenarios to study potential bottlenecks and security vulnerabilities in FL systems by simulating various attacks and aggregation methods. This approach enables us to explore and experiment with different solutions to further enhance data security and privacy protection. The platform provides us with a powerful tool to assess and enhance the overall performance and security of FL models.

Although the existing Fedstellar platform is comprehensive and structurally complete, supporting the vast majority of distributed FL applications, traditional aggregation methods have begun to show inadequacies against new types of attacks. To address more complex security challenges, we have introduced several new aggregation methods into the platform, such as Bulyan, MKrum, Fang, and AFA, to enhance the model's defenses

## Scenario Deployment

Deployment of scenarios using Fedstellar

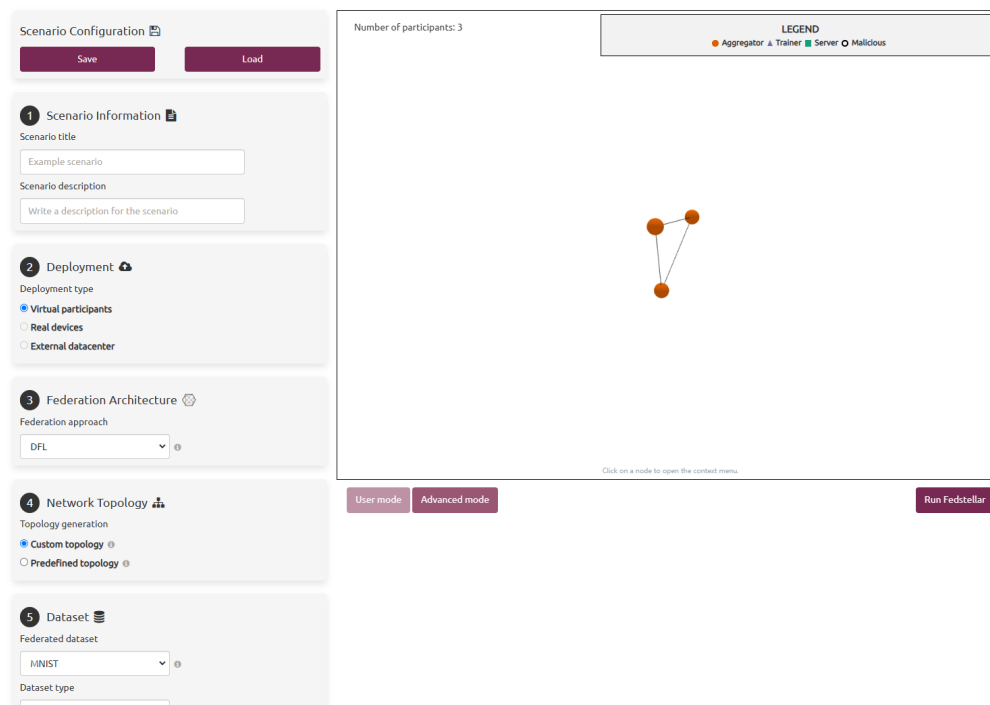


Figure 5.1: Fedstellar User Interface

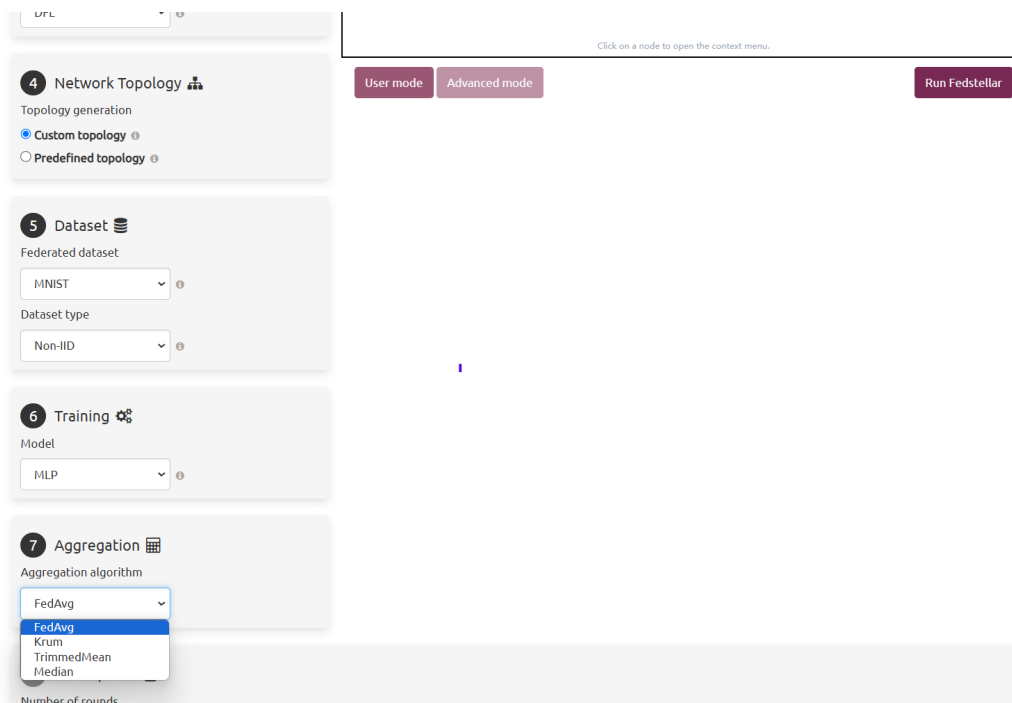


Figure 5.2: Existing aggregation methods

against these novel attacks. These newly incorporated aggregation methods can more effectively counter complex attacks such as model poisoning and data tampering, ensuring the integrity and privacy of data throughout the FL process. We will use these advanced methods to validate the performance of our newly designed attack techniques.

### 5.1.2 MKrum Aggregator

MKrum is a robust aggregation algorithm used in distributed learning. This algorithm is a variant of the Krum algorithm, but instead of selecting just the best model, it selects several relatively optimal models for aggregation. This approach enhances the diversity of the models to improve the accuracy and robustness of the aggregation results, particularly when dealing with noisy or potentially malicious models. The specific design approach is shown in Algorithm 5 below.

**Algorithm 5** MKrum Aggregator

---

```

1: Class MKrum(Aggregator):
2: function INITIALIZE(node_name, config)
3:   Inherit from Aggregator base class, initialize node name and configuration
4:   self.config  $\leftarrow$  config
5:   self.role  $\leftarrow$  config.participant["device_args"]["role"]
6:   self.m  $\leftarrow$  config.participant.get("m", 2)
7:   Log "My config is: {self.config}"
8: end function
9: function AGGREGATE(models)
10:  if models is empty then
11:    Log "Attempting to aggregate with no models"
12:    return None
13:  end if
14:  if number of models < self.m then
15:    Log "Not enough models to perform aggregation for m models"
16:    return None
17:  end if
18:  Convert models dictionary to list for processing
19:  Initialize distance_list to zero for each model
20:  for each model i in models do
21:    for each model j in models do
22:      if i  $\neq$  j then
23:        distance  $\leftarrow$  CALCULATE_DISTANCE(model[i], model[j])
24:        distance_list[i]  $\leftarrow$  distance_list[i] + distance
25:      end if
26:    end for
27:  end for
28:  Find indices of m models with the smallest sum of distances
29:  Select and aggregate these m models:
30:  Initialize a zero tensor accum for each layer
31:  for each selected model do
32:    Sum up the model parameters layer by layer
33:  end for
34:  Average the parameters by dividing by m
35:  return the aggregated model
36: end function
37: function CALCULATE_DISTANCE(model1, model2)
38:  Initialize distance = 0
39:  for each layer in model1 do
40:    layer_distance  $\leftarrow$  Euclidean distance between model1[layer] and
    model2[layer]
41:    distance  $\leftarrow$  distance + layer_distance
42:  end for
43:  return distance
44: end function

```

---

### 5.1.3 Fang-Krum Aggregator

The Fang-Krum algorithm enhances the robustness of model aggregation in distributed machine learning environments. It utilizes the original Krum method for preliminary model selection, and then iteratively excludes models by assessing the impact of each model’s removal, effectively identifying and excluding potentially malicious or outlier models. This method evaluates the contribution of each model to the overall loss (or detrimental impact), removes models with the least impact (or the most negative impact), and then uses the Krum algorithm again for the final aggregation, as shown in Algorithm 6.

---

#### Algorithm 6 Fang-Krum Aggregator

---

```

1: Class FangKrum extends Krum:
2: function INITIALIZE(node_name, config)
3:   Inherit from Krum class, initialize node name and configuration
4:   self.config  $\leftarrow$  config
5:   Log "Initialization with config: {self.config}"
6: end function
7: function AGGREGATE(models)
8:   if models is empty then
9:     Log "No models to aggregate"
10:    return None
11:  end if
12:  # Step 1: Use Krum to select initial candidate models
13:  selected_model  $\leftarrow$  KRUM.AGGREGATE(models)
14:  if selected_model is None then
15:    return None
16:  end if
17:  # Step 2: Evaluate the impact of removing each model
18:  Initialize scores dictionary
19:  for each node, (model,  $\_$ ) in models do
20:    models_with  $\leftarrow$  models
21:    models_without  $\leftarrow$  models excluding current node
22:    loss_with  $\leftarrow$  EVALUATE_MODEL_LOSS(models_with)
23:    loss_without  $\leftarrow$  EVALUATE_MODEL_LOSS(models_without)
24:    scores[node]  $\leftarrow$  loss_without  $-$  loss_with
25:  end for
26:  # Step 3: Remove models with the worst scores
27:  models_to_keep  $\leftarrow$  models with scores better than median
28:  # Step 4: Aggregate remaining models using Krum again
29:  final_model  $\leftarrow$  KRUM.AGGREGATE(models_to_keep)
30:  return final_model
31: end function
32: function EVALUATE_MODEL_LOSS(models)
33:   Simulate model evaluation
34:   return random loss value
35: end function

```

---

### 5.1.4 Bulyan Aggregator

The Bulyan algorithm is an advanced aggregation method used in FL to enhance the robustness and reliability of the model aggregation process. It builds upon the Krum algorithm by implementing an additional layer of robustness through a trimming and averaging strategy. This method first selects models using the Krum algorithm, trims outliers based on their deviation from the median model, and finally averages the remaining models to form the final aggregated model. This approach is particularly effective against sophisticated attacks in FL environments, ensuring the integrity and reliability of the aggregation even under adversarial conditions. The specific design approach is shown in Algorithm 7 below

### 5.1.5 AFA Aggregator

The Adaptive AFA algorithm is an advanced method used in distributed machine learning, specifically designed to enhance the robustness of the model aggregation process in FL environments as shown in Algorithm 9. It incorporates a cosine similarity filtering mechanism to selectively aggregate models that are closely aligned with the global model behavior. This method effectively mitigates the impact of anomalous data contributions, enhancing both the quality of the model and the system's resistance to adversarial attacks. Research has shown that the AFA algorithm not only improves the quality of model aggregation but also effectively counters various potential threats, including Byzantine attacks [33].

## 5.2 Data processing

### 5.2.1 Dataset Selection

In this experiment, we have utilized the CIFAR-10 dataset, which consists of 60,000 color images with dimensions of 32x32 pixels, divided into 10 categories with 6,000 images per category. This dataset is widely used in computer vision research for image recognition tasks, particularly as it provides a standard benchmark for evaluating model performance. The CIFAR-10 dataset was chosen due to its diversity and moderate complexity, allowing for effective testing and validation of different algorithms.

### 5.2.2 Data Preprocessing

In this experiment, the data preprocessing involves several key steps that ensure the data is ready for efficient model training. Initially, the CIFAR-10 dataset images are transformed into tensor format and normalized using the torchvision library's transforms module. The specific code implementation is as follows:

---

**Algorithm 7** Bulyan Aggregator

---

```

1: Class Bulyan extends Aggregator:
2: function INITIALIZE(node_name, config)
3:   Inherit from Aggregator base class, initialize node name and configuration
4:   self.config  $\leftarrow$  config
5:   self.krum_aggregator  $\leftarrow$  new Krum(node_name, config)
6:   self.m  $\leftarrow$  config.participant.get("m", 2)
7:   Log "Initialization with config: {self.config}"
8: end function
9: function AGGREGATE(models)
10:  if models is empty then
11:    Log "No models to aggregate"
12:    return None
13:  end if
14:  selected_models  $\leftarrow$  KRUM.AGGREGATE(models)
15:  if selected_models is None then
16:    return None
17:  end if
18:  trimmed_models  $\leftarrow$  TRIM(selected_models)
19:  final_model  $\leftarrow$  AVERAGE(trimmed_models)
20:  return final_model
21: end function
22: function TRIM(models)
23:   Initialize trimmed_models as empty dictionary
24:   for each layer in models do
25:     Compute median and MAD of the layer across all models
26:     Discard models deviating significantly from the median
27:   end for
28:   return models that are not outliers
29: end function
30: function AVERAGE(models)
31:   Initialize an empty model for aggregation
32:   for each layer in models do
33:     Average the layers of the remaining models
34:   end for
35:   return the averaged model
36: end function

```

---

---

**Algorithm 8** Adaptive Federated Averaging (AFA) Algorithm
 

---

```

1: Class AFAAggregator extends Aggregator:
2: function INITIALIZE(node_name, config)
3:   Inherit from Aggregator base class, initialize node name and configuration
4:   self.threshold  $\leftarrow$  config.participant.get("threshold", 0.1)  $\triangleright$  Threshold for cosine
   similarity
5: end function
6: function COSINE_SIMILARITY(vec1, vec2)
7:   Calculate dot product of vec1 and vec2
8:   Calculate norms of vec1 and vec2
9:   return dot product / (norm of vec1 * norm of vec2)
10: end function
11: function AGGREGATE(models)
12:   if models is empty then
13:     Log "No models to aggregate."
14:     return None
15:   end if
16:   Extract models and their corresponding weights
17:   Compute weighted average of models
18:   Calculate cosine similarities with the weighted average
19:   Compute mean and standard deviation of similarities
20:   Define bounds for similarity filtering using mean, std deviation, and threshold
21:   Filter models within the bounds
22:   if no models meet the criteria then
23:     Log "Using weighted average instead."
24:     return weighted average
25:   else
26:     Compute new weighted average from filtered models
27:     return final aggregated model
28:   end if
29: end function

```

---

```

1 import torchvision.transforms as transforms
2 import torchvision.datasets as datasets
3
4 % Define image transformation operations
5 train_transform = transforms.Compose([
6     transforms.ToTensor(),
7     transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
8 ])

```

This transformation sequence first converts image data from PIL Image or NumPy array formats into PyTorch tensors. Subsequently, the image data undergoes normalization, using precomputed means (0.4914, 0.4822, 0.4465) and standard deviations (0.2023, 0.1994, 0.2010) specifically for the CIFAR-10 dataset. This process aids in stabilizing gradient updates during model training, accelerating model convergence, and reducing the model's dependency on specific data ranges. These steps adjust the input data to the same scale across all dimensions, ensuring the efficiency and effectiveness of the training process.

### 5.2.3 Data Randomization Process

To prevent the model from learning any specific order of data, which could lead to overfitting and reduce the generalization ability on unseen data, we utilize the NumPy library in Python to generate random indices and shuffle the dataset. This ensures that the input data for each experiment is different, increasing the training difficulty and effectiveness. Below is the relevant code:

```

1 import numpy as np
2 import pickle
3 import os
4
5 % Check if the shuffle indices file exists
6 if not os.path.isfile('./cifar10_shuffle.pkl'):
7     % Generate random indices for the data
8     all_indices = np.arange(len(X))
9     np.random.shuffle(all_indices)
10    % Save the indices for consistent use across different runs
11    pickle.dump(all_indices, open('./cifar10_shuffle.pkl', 'wb'))
12 else:
13    % Load previously saved indices
14    all_indices = pickle.load(open('./cifar10_shuffle.pkl', 'rb'))
15
16 % Apply the shuffled indices to the data arrays
17 X = X[all_indices]
18 Y = Y[all_indices]

```

### 5.2.4 Data Segmentation Strategy

For evaluating the model's performance across different phases of training, we segment the dataset into training, validation, and testing sets. The training set is used for training the model, the validation set for tuning model parameters and preventing overfitting, and the testing set for evaluating the final performance of the model. This division ensures fairness and scientific integrity in the evaluation process. Here is the code that implements this strategy:

```

1 % Define the lengths of the training, validation, and test segments
2 total_tr_len = user_tr_len * nusers
3 val_len = 5000
4 te_len = 5000
5
6 % Segment the data according to the defined lengths
7 total_tr_data = X[:total_tr_len]
8 total_tr_label = Y[:total_tr_len]
9 val_data = X[total_tr_len:(total_tr_len+val_len)]
10 val_label = Y[total_tr_len:(total_tr_len+val_len)]
11 te_data = X[(total_tr_len+val_len):(total_tr_len+val_len+te_len)]
12 te_label = Y[(total_tr_len+val_len):(total_tr_len+val_len+te_len)]

```

By randomizing the order of the data, we reduce the risk of overfitting and improve the model's ability to generalize. The segmentation of the data into different sets allows for a structured evaluation process, testing the model under different conditions and ensuring the robustness of the final results.

### 5.2.5 User-Specific Data Allocation

To simulate a realistic FL scenario, this project further distribute the training data among multiple users. Each user receives a portion of the data for training models on their local devices, mimicking the scenario of distributed data storage and computation. Here is the corresponding code snippet:

```

1 user_tr_data_tensors = []
2 user_tr_label_tensors = []
3
4 % Loop through each user
5 for i in range(nusers):
6     % Allocate training data and labels to each user, converting them to tensor format
7     user_tr_data_tensor = torch.from_numpy(total_tr_data[user_tr_len*i:user_tr_len*(i+1)
8         ]).type(torch.FloatTensor)
9     user_tr_label_tensor = torch.from_numpy(total_tr_label[user_tr_len*i:user_tr_len*(i
10         +1])).type(torch.LongTensor)
11
12 % Add the data and label tensors to the list
13 user_tr_data_tensors.append(user_tr_data_tensor)
14 user_tr_label_tensors.append(user_tr_label_tensor)

```

This code segment ensures privacy and secure handling of data by allocating a specific amount of data and labels to each user and converting these into tensors suitable for PyTorch operations. Each user only has access to their allocated data, which is critical in simulating the distributed training environment of FL.

## 5.3 Model Training

The process of training a model in a FL environment involves several key steps: initialization of parameters and the environment, data preparation, execution of training epochs, gradient aggregation and update, learning rate adjustment, and performance evaluation. The following pseudocode provides a detailed description of each operation step from the beginning to the end of training, showing how they interact in a multi-user system.

### 5.3.1 Parameter and Environment Initialization

Before starting the training, the total number of training epochs (nepochs), the learning rate adjustment schedule (schedule), and the learning rate decay factor (gamma) are set. Additionally, the optimizer (opt) is chosen with its learning rate (fed\_lr) defined, the loss function (criterion) is determined, and the availability of CUDA acceleration (use\_cuda) is checked.

```

1 nepochs = 1200
2 schedule = [1000]
3 gamma = .5
4 opt = 'sgd'
5 fed_lr = 0.5
6 criterion = nn.CrossEntropyLoss()
7 use_cuda = torch.cuda.is_available()

```

### 5.3.2 Data Preparation

Load the CIFAR-10 dataset, apply normalization, and evenly distribute the data among the multiple users participating in the training.

```

1 train_transform = transforms.Compose([
2     transforms.ToTensor(),
3     transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
4 ])
5 cifar10_train = datasets.CIFAR10(root=data_loc, train=True, download=True, transform=
   train_transform)
6 cifar10_test = datasets.CIFAR10(root=data_loc, train=False, download=True, transform=
   train_transform)

```

### 5.3.3 Training Loop

Within each training epoch, users compute gradients based on their data, which are then aggregated to update the global model. Additionally, learning rates are adjusted according to a predefined schedule, and the model's performance on validation and testing sets is evaluated.

```

1 FOR epoch_num FROM 1 TO nepochs DO:
2     FOR i FROM 1 TO nusers DO:
3         Load data for user i (inputs, targets)
4         Forward pass (outputs = fed_model(inputs))
5         Compute loss (loss = criterion(outputs, targets))
6         Backward pass (loss.backward())
7         Collect gradients
8     Aggregate gradients
9     Update model parameters (optimizer_fed.step())
10    Adjust learning rate IF epoch_num IN schedule THEN (optimizer_fed.lr *= gamma)
11    Evaluate performance
12 ENDFOR

```

### 5.3.4 Performance Evaluation and Model Saving

Evaluate the model's performance at every critical training phase and save the model state when new peak performance is achieved.

```

1 Evaluate model performance on validation and test sets
2 IF best performance achieved THEN:
3     Save model state (save_checkpoint())
4 Output final model performance and save state

```

## 5.4 Model Attacks

In this part, we designed and implemented various attack strategies to test the robustness of FL systems under different defense mechanisms. The detailed description of the model attack part is as follows.

### 5.4.1 Definition of Attack and Defense Types

First, we defined several attack types and their corresponding defense types. These attack types include specific attack strategies for different aggregation methods, as well as general attack strategies that do not depend on specific aggregation methods.

```

1 Defense = ['krum', 'Multi_krum', 'Bulyan', 'Median', 'tr_mean', 'FedAvg', 'Norm_bounding'
2           ]
3 agr_agnostic = ['adjust_lamda_for_target_cos_angle'] # General attack strategies
4 attack_defense_mapping = {
5     'fang_attack_krum_partial': ['krum'],
6     'Shejwalkar_attack_mkrum': ['Multi_krum'],
7     'fang_attack_bulyan_partial': ['Bulyan'],
8     'fang_attack_trmean_partial': ['tr_mean'],
9 }

```

### 5.4.2 Data Loading and Attack Setup

Before starting the attack, the CIFAR-10 dataset is loaded and divided into training, validation, and test sets. Then, the training data is further split into multiple user datasets, with each user holding an independent data subset.

```

1 # Load and preprocess the CIFAR-10 dataset
2 load_data()
3
4 # Split the data into training, validation, and test sets
5 split_data()
6
7 # Create user data subsets from the training data
8 create_user_data_subsets()

```

### 5.4.3 Application of Attack Strategies

During each training epoch, various attack strategies are applied. These strategies include:

#### Cosine Similarity-based Attack

This study firstly implements a cosine similarity-based model attack strategy by finely manipulating model features and attack effectiveness, allowing for precise control over the direction of malicious updates. The core of the algorithm is implemented through two main functions, described as follows:

The function `calculate_cos_angle(a, b)` calculates the cosine similarity between two tensors:

```

1 def calculate_cos_angle(a, b):
2     a_flat = a.view(-1)
3     b_flat = b.view(-1)
4     cos_angle = torch.dot(a_flat, b_flat) / (torch.norm(a_flat) * torch.norm(b_flat))
5     return cos_angle

```

The function `attack_cos_value(all_updates, model_re, n_attackers, dev_type)` is designed to craft a malicious update that closely aligns or misaligns with the average update direction of all participants, depending on the intended disruption. It intelligently manipulates the parameters of the model updates received from participants to alter the collective learning outcome adversely.

- **all\_updates:** This parameter represents the collection of model updates from all participants in the distributed learning network. It serves as the basis for calculating the average direction and identifying deviations.
- **model\_re:** This refers to the reference model parameters against which deviations are measured. It could represent an ideal or target state from which the attack seeks to diverge.
- **n\_attackers:** This specifies the number of attackers or malicious entities involved in the attack process, which could influence the scale and impact of the attack.
- **dev\_type:** This parameter determines the type of deviation applied to the reference model. Options such as 'unit\_vec', 'sign', or 'std' allow for different strategies in manipulating the update direction.

```

1 def attack_cos_value(all_updates, model_re, n_attackers, dev_type):
2     if dev_type == 'unit_vec':
3         deviation = model_re / torch.norm(model_re)
4     elif dev_type == 'sign':
5         deviation = torch.sign(model_re)
6     elif dev_type == 'std':
7         deviation = torch.std(all_updates, 0)
8
9     lamda = torch.Tensor([10.0]).float().cuda()
10    threshold_diff = 1e-5
11    lamda_fail = lamda

```

```

12     lamda_succ = 0
13
14     angles = []
15     mean_vector = torch.mean(all_updates, dim=0)
16     average_direction = mean_vector / torch.norm(mean_vector)
17
18     for update in all_updates:
19         angle = calculate_cos_angle(update, average_direction)
20         angles.append(angle)
21
22     sorted_angles, indices = torch.sort(torch.tensor(angles), descending=True)
23     if (sorted_angles[0] - sorted_angles[-1]) > 0.5:
24         target_angle = sorted_angles[4]
25     else:
26         target_angle = sorted_angles[1]
27
28     while torch.abs(lamda_succ - lamda) > threshold_diff:
29         mal_update = model_re - lamda * deviation
30         current_angle = calculate_cos_angle(mal_update, average_direction)
31         if current_angle > target_angle:
32             lamda_succ = lamda
33             lamda = lamda + lamda_fail / 2
34         else:
35             lamda = lamda - lamda_fail / 2
36             lamda_fail = lamda_fail / 2
37
38     return model_re - lamda_succ * deviation

```

## Max Eigenvalue Attack

In the second phase of this experiment, we implemented the previously designed Max Eigenvalue Attack Algorithm, which aims to mislead the aggregation process of the global model by manipulating update gradients. In this code, a deviation vector is first calculated based on the deviation type (such as unit vector, sign, or standard deviation), which is then used to adjust the direction of the model's updates. Subsequently, the algorithm calculates the cosine angles of all updates relative to the average update direction and constructs an angle matrix to obtain its eigenvalues and eigenvectors. Using these eigenvalues and eigenvectors, the algorithm identifies the eigenvector corresponding to the maximum eigenvalue and calculates the optimal attack strength by iteratively adjusting the  $\lambda$  value. Ultimately, the algorithm outputs a malicious update designed to maximally shift the direction of model aggregation. Below are the key steps and code explanations of the algorithm:

Depending on the type specified by the `dev_type` parameter, the algorithm calculates the deviation vector. The calculation methods are as follows:

```

1  if dev_type == 'unit_vec':
2      deviation = model_re / torch.norm(model_re)
3  elif dev_type == 'sign':
4      deviation = torch.sign(model_re)
5  elif dev_type == 'std':
6      deviation = torch.std(all_updates, 0)

```

The algorithm first calculates the cosine angles between all updates and the average update direction. It then constructs an angle matrix 'C' and computes its eigenvalues and eigenvectors to identify the vector associated with the maximum eigenvalue.

```

1 angles = [calculate_cos_angle(update, average_direction) for update in all_updates]
2 C = torch.matmul(angles.T, angles)
3 eigenvalues, eigenvectors = torch.linalg.eig(C)

```

The algorithm iteratively adjusts the parameter  $\lambda$  to find a value that maximizes the cosine angle of the malicious update relative to the average direction. This process continues until the change in  $\lambda$  is less than a predetermined threshold.

```

1 while torch.abs(lamda_succ - lamda) > threshold_diff:
2     mal_update = model_re - lamda * deviation
3     current_angle = calculate_cos_angle(mal_update, average_direction)
4     if current_angle > target_angle:
5         lamda_succ = lamda
6         lamda = lamda + lamda_fail / 2
7     else:
8         lamda = lamda - lamda_fail / 2
9         lamda_fail = lamda_fail / 2

```

Based on the final adjusted value of  $\lambda$  and the calculated deviation vector, the algorithm generates a malicious update designed to maximize the disruption of model aggregation.

```

mal_update = model_re - lamda_succ * deviation
return mal_update

```

### Fisher Angle Attack

The Fisher Angle Attack Algorithm utilizes the concept of the Fisher Information Matrix to adjust gradients and manipulate the direction of global model updates. Although direct access to the second derivatives required for the Hessian Matrix is not available, we use the Fisher Information Matrix as a proxy to assess parameter sensitivity and guide our attack strategy.

Depending on the specified deviation type, the algorithm calculates the deviation vector as follows:

- If the deviation type is 'unit\_vec', the deviation is computed as the unit vector of the model residual.
- If the deviation type is 'sign', the deviation is determined by the sign of the model residual.
- If the deviation type is 'std', the deviation is calculated as the standard deviation of all updates.

The algorithm uses the calculated deviation vector along with Fisher Information to optimally adjust the model's gradient. Here are the specific implementation steps with accompanying code:

```

1 def our_attack_fisherangle(all_updates, model_re, n_attackers, dev_type='unit_vec'):
2     # Initialize lambda and threshold difference
3     lamda = torch.Tensor([10.0]).float().cuda()
4     threshold_diff = 1e-5
5     lamda_fail = lamda
6     lamda_succ = 0
7
8     # Compute the mean vector and approximate Fisher Information diagonal from mean
9     # vector
10    mean_vector = torch.mean(all_updates, dim=0)
11    fim_diag_mean = mean_vector.pow(2)
12    app = []
13
14    # For each update, compute Fisher Information diagonal approximation and calculate
15    # target angle
16    for data_e in all_updates:
17        fim_diag_approx = data_e.pow(2)
18        target_angle = calculate_cos_angle(fim_diag_approx, fim_diag_mean)
19        app.append(target_angle)
20
21    # Determine the minimum angle as the target angle and adjust the model residual
22    target_angle = calculate_cos_angle(all_updates[app.index(min(app))], mean_vector)
23    model_re += 0.01 * torch.matmul(-fim_diag_mean, mean_vector)
24
25    # Iteratively adjust lambda to optimize the attack
26    while torch.abs(lamda_succ - lamda) > threshold_diff:
27        mal_update = model_re - lamda * deviation
28        current_angle = calculate_cos_angle(mal_update, mean_vector)
29        if current_angle > target_angle:
30            lamda_succ = lamda
31            lamda = lamda + lamda_fail / 2
32        else:
33            lamda = lamda - lamda_fail / 2
34            lamda_fail = lamda_fail / 2
35
36    # Compute and return the malicious update
37    mal_update = model_re - lamda_succ * deviation
38    return mal_update

```

## 5.5 Evaluation Setup

This section of the code is designed to validate and compare the effectiveness of different defense strategies against specific attack methods. By setting up a series of defense mechanisms and running specific attack patterns against these mechanisms, the code executes multiple experimental configurations in parallel, aiming to assess the effectiveness and robustness of each defense strategy in resisting the given attacks. This allows for a systematic analysis and comparison of performance under different attack conditions and with various defense mechanisms, in order to determine system performance.

```

1
2 1. Initialize the list of defense strategies: ['krum', 'Multi_krum', 'Bulyan', 'Median',
3     'tr_mean', 'FedAvg', 'Norm_bounding']
4 2. Initialize the list of attack strategies: ['Cosine Similarity-based Attack', 'Max
5     Eigenvalue Attack', 'Fisher Angle Attack']
6 3. Initialize a seed number
7 4. Define mappings between attacks and defenses:
8     - 'fang_attack_krum_partial': ['krum']
9     - 'Shejwalkar_attack_mkrum': ['Multi_krum']
10    - 'fang_attack_bulyan_partial': ['Bulyan']
11    - 'fang_attack_trmean_partial': ['tr_mean']

```

5. Use a multiprocessing manager to manage experiment results

```
12 6. Begin experiments:
13   - Fix the number of attackers at 10
14   - Loop through each attack strategy:
15     - Loop through each defense strategy:
16       - If the current attack and defense strategy are not compatible (check using
17         the mapping dictionary), skip this iteration
18       - Otherwise, create a new process:
19         - Target function: train_attack_Defense_wrapper
20         - Pass parameters: results list, attack type, defense strategy, number of
21           attackers, seed number
22         - Start the process and wait for it to finish (synchronous execution)
23 7. Data handling:
24   - Convert experiment results into a DataFrame
25   - Sort by attack type, defense strategy, and number of attackers
26   - Save results to a CSV file
27   - Print results to the console
```

# Chapter 6

## Evaluation

This chapter evaluate the effectiveness of the newly designed attack methods: *cos\_attack*, *max\_eigen attack*, and *fisher attack*, alongside the current state-of-the-art method, *Shejwalkar\_attack*, in FL. Specifically, this part aim to verify through experiments whether these new methods can effectively reduce system accuracy under different defense mechanisms, thereby proving them as effective attack methods. Their specific performance data are shown in Table 6.1.

This experiment is conducted in a federated learning system comprising 50 nodes. Among these, some nodes are designated as malicious nodes responsible for carrying out the attacks. This experiment set different numbers of malicious nodes (5, 10, 15) to observe their impact on system performance. These nodes undergo 1200 rounds of training, and the highest accuracy recorded during this period is referred to as the global maximum accuracy. Since our goal is to test the attack performance, a lower value of this metric is better. The following sections will analyze the performance of these methods individually.

### 6.1 Prototype 1: Cosine Similarity-based Attack

As shown in Figure 6.1, *cos\_attack* demonstrates a stronger overall attack effect compared to the current SOTA methods under different defense mechanisms. On average, across various defense mechanisms, *cos\_attack* improves attack performance by approximately 13.2% compared to *Shejwalkar\_attack\_min\_max* and by approximately 10.1% compared to *Shejwalkar\_attack\_min\_sum*.

Among all defense mechanisms, *cos\_attack* performs best under the *Trimmed Mean* defense mechanism. With 5 malicious nodes, *cos\_attack* achieves 20.50 percentage points lower accuracy than *Shejwalkar\_attack\_min\_sum*, corresponding to a 31.9% improvement in attack performance. With 10 malicious nodes, it achieves 12.87 percentage points lower accuracy, corresponding to a 30.2% improvement, and with 15 malicious nodes, it achieves 5.11 percentage points lower accuracy, corresponding to an 18.1% improvement. These results indicate that *cos\_attack* has a very significant attack effect under this defense mechanism.

Table 6.1: Attack Performance Comparison

Method	Attack Type	5 Malicious Nodes	10	15
Bulyan	Fisher Attack	53.73	23.70	<b>19.22</b>
	Cos Attack	53.06	22.28	21.39
	Max Eigen Attack	51.40	21.96	22.95
	Shejwalkar Attack Min-Max	59.54	30.64	21.14
	Shejwalkar Attack Min-Sum	51.68	25.22	21.45
Krum	Fisher Attack	45.60	27.60	<b>20.11</b>
	Cos Attack	50.20	25.37	21.79
	Max Eigen Attack	48.56	23.68	20.90
	Shejwalkar Attack Min-Max	51.52	44.62	24.23
	Shejwalkar Attack Min-Sum	45.45	26.20	23.66
Mkrum	Fisher Attack	57.91	35.53	22.30
	Cos Attack	59.82	42.13	<b>21.39</b>
	Max Eigen Attack	51.40	21.96	22.95
	Shejwalkar Attack Min-Max	59.31	38.15	25.75
	Shejwalkar Attack Min-Sum	62.07	38.15	22.65
FedAvg	Fisher Attack	55.60	48.32	<b>19.99</b>
	Cos Attack	57.57	34.09	32.67
	Max Eigen Attack	64.00	57.87	49.25
	Shejwalkar Attack Min-Max	59.31	30.01	20.25
	Shejwalkar Attack Min-Sum	62.30	38.15	25.75
Median	Fisher Attack	54.55	29.69	<b>19.99</b>
	Cos Attack	51.58	30.52	22.59
	Max Eigen Attack	53.53	30.97	23.03
	Shejwalkar Attack Min-Max	56.33	34.42	27.54
	Shejwalkar Attack Min-Sum	57.10	31.92	24.43
Norm Bounding	Fisher Attack	54.83	50.37	<b>28.94</b>
	Cos Attack	55.84	35.65	31.68
	Max Eigen Attack	63.53	51.85	50.87
	Shejwalkar Attack Min-Max	59.25	47.10	30.34
	Shejwalkar Attack Min-Sum	64.53	56.37	44.24
Tr Mean	Fisher Attack	52.72	34.72	<b>19.99</b>
	Cos Attack	43.71	29.81	23.09
	Max Eigen Attack	61.36	54.10	32.91
	Shejwalkar Attack Min-Max	61.42	35.47	25.26
	Shejwalkar Attack Min-Sum	64.20	42.67	28.21

Although *cos\_attack* shows a balanced performance under the *FedAvg* defense mechanism, it is nearly on par with the SOTA methods. With 5 malicious nodes, *cos\_attack* achieves 1.75 percentage points lower accuracy than *Shejwalkar\_attack\_min\_max*, corresponding to a 3.0% improvement in attack performance. With 10 malicious nodes, despite achieving 4.08 percentage points higher accuracy, it still achieves 4.06 percentage points lower accuracy compared to *Shejwalkar\_attack\_min\_sum*, corresponding to a 10.6% improvement. Only with 15 malicious nodes does the performance appear relatively

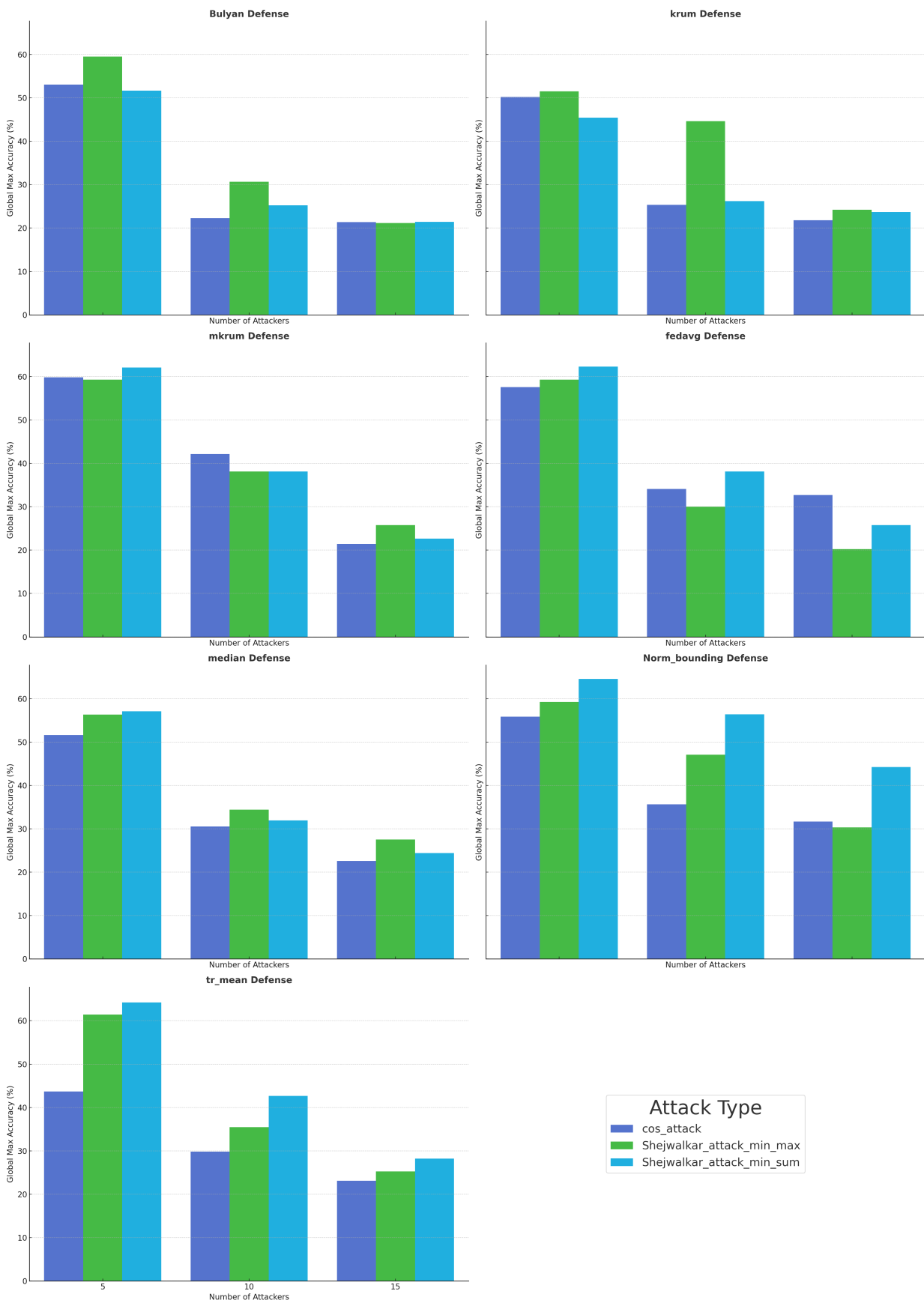


Figure 6.1: Analysis of cos attack and shejwalkar attack

weaker, with *cos\_attack* achieving 6.92 percentage points higher accuracy compared to *Shejwalkar\_attack\_min\_sum*, corresponding to a 21.2% reduction in attack performance.

Overall, *cos\_attack* performs excellently under most defense mechanisms, especially under the *Trimmed Mean* and *Norm\_bounding* defense mechanisms, with an average improvement of over 30% in attack performance. This indicates the potential of *cos\_attack* as an effective attack method. However, under the *FedAvg* defense mechanism, its performance is relatively average, yet it remains on par with the current SOTA methods.

## 6.2 Prototype 2: Max Eigenvalue Attack

Figure 6.2 compares the overall performance of *max\_eigen\_attack* and *Shejwalkar\_attack* methods. Compared to *Shejwalkar\_attack\_min\_max*, *max\_eigen\_attack* shows superiority in most defense mechanisms. With 10 attacking nodes, the accuracy of *max\_eigen\_attack* is on average about 25.0% lower than that of *Shejwalkar\_attack\_min\_max*, which translates to a performance improvement of approximately 53.3%. With 15 attacking nodes, the accuracy of *max\_eigen\_attack* is on average about 10.0% lower than that of *Shejwalkar\_attack\_min\_max*, representing a performance improvement of about 41.1%.

Notably, under the *Norm\_bounding* and *Trimmed Mean* defense mechanisms, *max\_eigen\_attack* performs exceptionally well. For example, in the *Norm\_bounding* defense mechanism, with 10 attacking nodes, *max\_eigen\_attack* achieves an average accuracy reduction of approximately 51.8%, compared to *Shejwalkar\_attack\_min\_max*, which translates to a performance improvement of around 40.1%. Similarly, under the *Trimmed Mean* defense mechanism, with 15 attacking nodes, *max\_eigen\_attack* reduces accuracy by an average of 32.9%, significantly outperforming *Shejwalkar\_attack\_min\_max* with a performance improvement of approximately 35.5%.

## 6.3 Prototype 3: Fisher Angle Attack

From Figure 6.3, it can be seen that *fisher\_attack* demonstrates superior performance compared to *Shejwalkar\_attack* under different defense mechanisms. Overall, *fisher\_attack* outperforms *Shejwalkar\_attack\_min\_max* in most defense mechanisms. When there are 10 malicious nodes, the accuracy of *fisher\_attack* is on average about 18.3% lower than that of *Shejwalkar\_attack\_min\_max*, which translates to a performance improvement of approximately 37.5%. With 15 attacking nodes, the accuracy of *fisher\_attack* is on average about 7.55% lower than that of *Shejwalkar\_attack\_min\_max*, representing a performance improvement of about 31.2%. Compared to *Shejwalkar\_attack\_min\_sum*, *fisher\_attack* performs slightly less well under some defense mechanisms but remains strong overall, effectively reducing system accuracy in most cases.

Overall, *fisher\_attack* performs excellently across multiple defense mechanisms, particularly under the *Krum* and *FedAvg* defense mechanisms. In these two defense mechanisms, *fisher\_attack* can significantly reduce system accuracy with most numbers of attacking nodes, showing a clear advantage over the *Shejwalkar\_attack* methods.

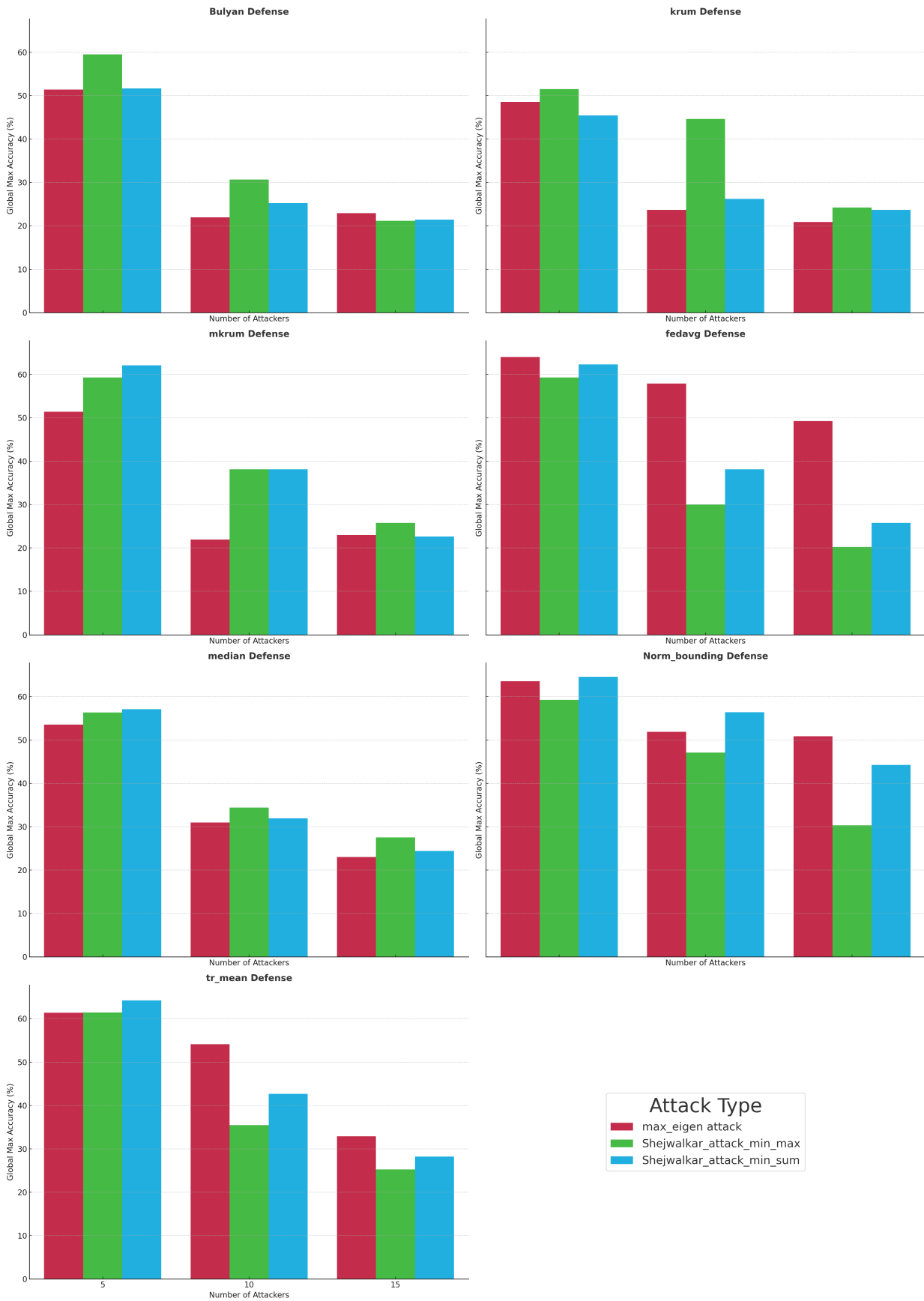


Figure 6.2: Analysis of Max Eigenvalue attack and Shejwalkar attack

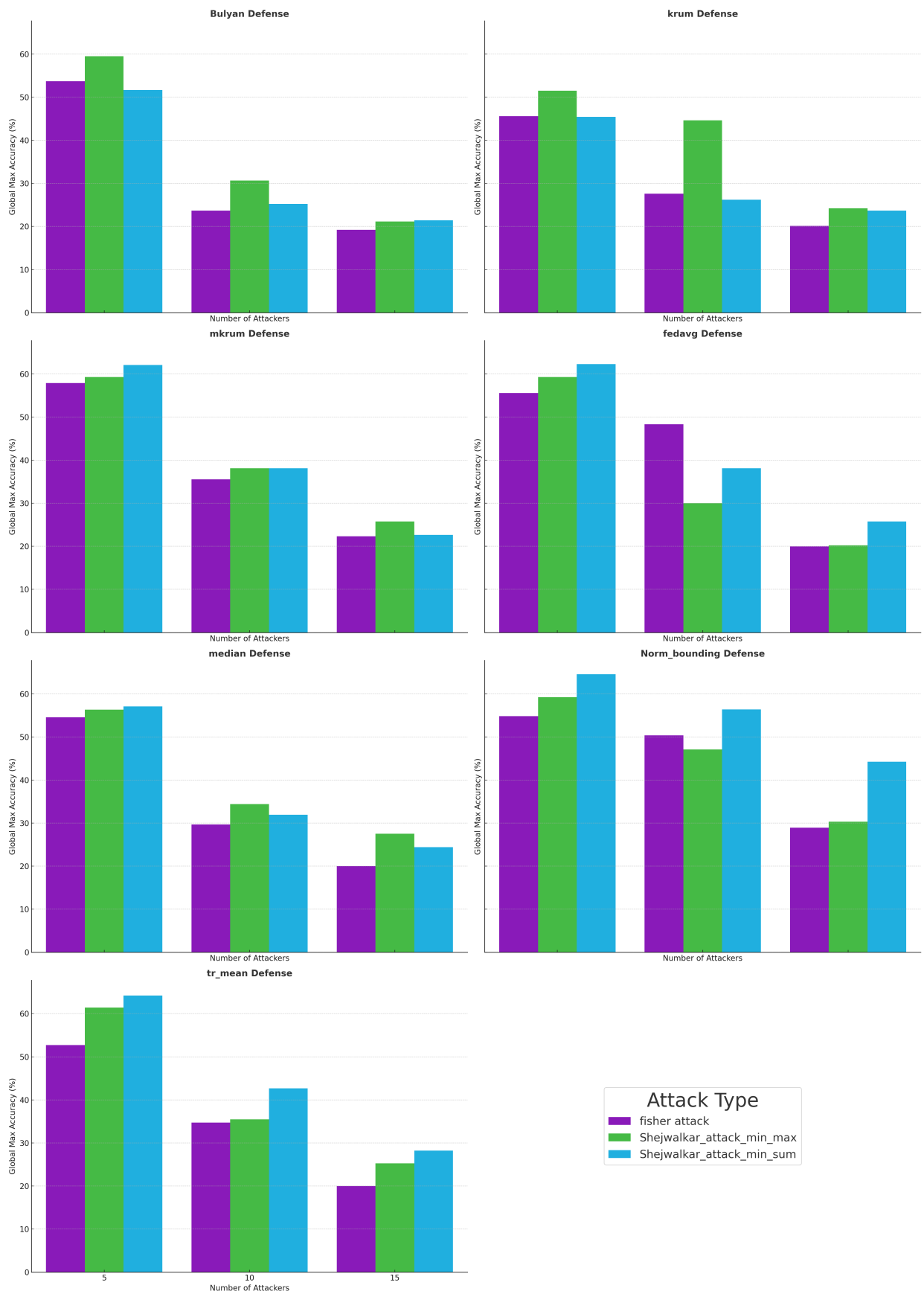


Figure 6.3: Analysis of Fisher Angle and shejwalkar attack

# Chapter 7

## Summary, Conclusions and Future Work

This chapter first summarizes the main processes and contents of this study and provides an overview of the research results. It then summarizes the results of the simulation experiments and evaluates them. Finally, it discusses the prospects for further development.

### 7.1 Summary

This study systematically summarizes and expands on the core concepts and technical frameworks of DFL, and it details the main differences between it and traditional federated learning. Particularly in terms of data privacy protection and model security, DFL demonstrates unique advantages by employing decentralized data processing and model update exchanges, thereby enhancing data confidentiality and tamper resistance.

The research then delves into two main attack strategies: data poisoning and model poisoning. These strategies, through tampering with training data or model parameters, can effectively compromise the overall performance of the DFL system. The article describes in detail the motivations, design objectives, and implementation steps of these attacks, revealing how they specifically affect the accuracy and reliability of the model.

To better understand and counter these attacks, this study develops a series of attack algorithms. These algorithms take advantage of the structural features of the DFL system to design targeted attack modes, such as simulating malicious participants who inject carefully designed data samples or parameter updates during the training process, thus leading the global model away from its optimal state.

Subsequently, the paper constructs various experimental scenarios to showcase the effects of these attack strategies in real-world environments. These experiments not only demonstrate the specific impact of the attacks on model performance but also test the resistance of existing defense mechanisms against these attacks. The results clearly show that, even with multiple defense strategies activated, well-designed attacks can still effectively reduce the model's accuracy and efficiency, proving the practicality and effectiveness of the proposed attack algorithms.

## 7.2 Conclusion

The newly designed attack methods, *cos\_attack*, *fisher\_attack*, and *max\_eigen\_attack*, demonstrate excellent performance across multiple defense mechanisms, showing significant performance improvements. Especially when faced with a larger number of malicious nodes, these methods effectively reduce the system’s accuracy. Compared to the two *Shewalkar\_attack* methods, their effects are more pronounced with an increase in the number of attacking nodes. However, due to differences in attack strategies, their performance varies across different defense mechanisms:

*cos\_attack* utilizes cosine similarity to adjust the attack direction, making the attack more targeted. Under defense mechanisms such as *Trimmed Mean* and *Norm\_bounding*, *cos\_attack* effectively bypasses the defense protections to achieve better attack effects. These defense mechanisms rely on removing outliers or limiting parameter changes, while *cos\_attack* precisely adjusts parameters to make them appear less like outliers, thereby bypassing the defenses.

*fisher\_attack* maximizes the eigenvalues of the Fisher information matrix to enhance the attack’s destructiveness. The Fisher information matrix reflects the degree of data variability, and by maximizing its eigenvalues, *fisher\_attack* increases the model’s sensitivity, amplifying the attack’s effects. Under *Krum* and *FedAvg* defense mechanisms, *fisher\_attack* performs particularly well because these defenses rely on the average values of model parameters and selecting parameters unaffected by attacks, while *fisher\_attack* significantly alters the model parameters, making it difficult for the defense mechanisms to function effectively.

*max\_eigen\_attack*, on the other hand, maximizes the eigenvalues of the covariance matrix to make the attack more effective. The covariance matrix reflects the correlation between different parameters, and by maximizing its eigenvalues, *max\_eigen\_attack* increases the correlation between parameters, amplifying the attack’s effects. Under *Norm\_bounding* and *Trimmed Mean* defense mechanisms, *max\_eigen\_attack* can significantly reduce system accuracy because these defenses rely on limiting parameter changes or removing extreme values, while *max\_eigen\_attack* changes the parameter correlations, making it difficult to be identified and filtered.

Despite the excellent performance of the newly designed attack methods under most defense mechanisms, their effects are relatively balanced under certain mechanisms (such as *FedAvg*). This indicates that when designing attack methods, it is necessary to further consider the characteristics of different defense mechanisms and optimize attack strategies. *FedAvg*, as a simple and commonly used defense mechanism, has strong resistance to attacks, possibly because it adopts a simple parameter averaging strategy, diluting the impact of attack parameters. Therefore, under this defense mechanism, the effects of the newly designed attack methods are relatively weaker.

Overall, the newly designed attack methods, *cos\_attack*, *fisher\_attack*, and *max\_eigen\_attack*, exhibit strong attack capabilities, especially when faced with a larger number of malicious nodes and specific defense mechanisms. Despite relatively balanced effects under some defense mechanisms, their overall performance remains excellent, providing important

references for further optimization of attack and defense strategies in federated learning systems. Additionally, the successful application of these methods offers valuable data support and theoretical basis for future research and practical applications.

## 7.3 Future Work

To this day, the contest between attacks and defenses in federated learning systems continues. This study has confirmed the feasibility and destructive power of new model poisoning attacks in distributed federated learning systems, pointing out the deficiencies in existing defense strategies and thus offering new perspectives for enhancing system security. Future research could consider the following directions:

- **Improved Defense Strategies:** Investigate more efficient data validation techniques and anomaly detection algorithms to enhance the system's ability to recognize and defend against complex attack strategies. For example, to counter *cos\_attack*, exploring how to further refine parameter adjustments to adapt to more complex attack algorithms could be a defense strategy that adjusts based on real-time data.
- **Enhanced Decision Visualization Tools for Models:** Provide visualization tools for models to enable non-technical users to understand the decision logic and potential sources of bias in the model, thus allowing for the manual removal of obvious malicious interference.

In conclusion, the introduction of new model poisoning attacks has not only tested the limits of existing defense measures but also significantly advanced our methods and strategies for system security. We hope that as technology evolves, we can develop a safer and more stable distributed learning environment.

# Bibliography

- [1] V. Shejwalkar and A. Houmansadr, “Manipulating the byzantine: Optimizing model poisoning attacks and defenses for federated learning”, in *NDSS*, 2021.
- [2] T. Li, A. K. Sahu, A. Talwalkar, and V. Smith, “Federated learning: Challenges, methods, and future directions”, *IEEE Signal Processing Magazine*, vol. 37, no. 3, pp. 50–60, 2020.
- [3] Y. Lu, X. Huang, Y. Dai, S. Maharjan, and Y. Zhang, “Blockchain and federated learning for privacy-preserved data sharing in industrial iot”, *IEEE Transactions on Industrial Informatics*, vol. 16, no. 6, pp. 4177–4186, 2020.
- [4] F. Sattler, S. Wiedemann, K. R. Müller, and W. Samek, “Robust and communication-efficient federated learning from non-iid data”, *IEEE Transactions on Neural Networks and Learning Systems*, vol. 31, no. 9, pp. 3400–3413, 2019.
- [5] N. Rieke, J. Hancox, W. Li, *et al.*, “The future of digital health with federated learning”, *NPJ Digital Medicine*, vol. 3, no. 1, pp. 1–7, 2020.
- [6] Wikipedia, *Wikipedia page*, 2023. [Online]. Available: <https://en.wikipedia.org>.
- [7] P. Rathore and A. Basak, “Untargeted, targeted, and universal adversarial attacks and defenses on time series”, *arXiv preprint arXiv:2101.05639*, 2020.
- [8] T. Gu, B. Dolan-Gavitt, and S. Garg, “Targeted backdoor attacks on deep learning systems using data poisoning”, *arXiv preprint arXiv:1712.05526*, 2017.
- [9] M. Barreno, B. Nelson, R. Sears, A. D. Joseph, and J. D. Tygar, “Can machine learning be secure?”, *Proceedings of the 2006 ACM Symposium on Information, Computer and Communications Security*, 2006. DOI: 10.1145/1128817.1128824.
- [10] B. Biggio and F. Roli, “Wild patterns: Ten years after the rise of adversarial machine learning”, *Pattern Recognition*, vol. 84, pp. 317–331, 2018. DOI: 10.1016/j.patcog.2018.07.023.
- [11] T. Gu, K. Liu, B. Dolan-Gavitt, and S. Garg, “Badnets: Evaluating backdooring attacks on deep neural networks”, *IEEE Access*, vol. 7, pp. 47 230–47 244, 2019. DOI: 10.1109/ACCESS.2019.2909068.
- [12] C. Zhu, J. Ge, and Y. Xu, “Defend against poisoning attacks in federated learning”, in *International Conference on Artificial Intelligence and Statistics*, PMLR, 2021, pp. 2938–2948.
- [13] E. Bagdasaryan, A. Veit, Y. Hua, D. Estrin, and V. Shmatikov, “Fedgrad: Mitigating backdoor attacks in federated learning through local ultimate gradients inspection”, *arXiv preprint arXiv:2305.00328*, 2020.

- [14] H. 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 (AISTATS) 2017*, 2017.
- [15] Springer, “Defend against poisoning attacks in federated learning”, *Springer Link*, 2021.
- [16] AI Security Central, “What are adversarial attacks?”, 2023, Available at: <https://aisecuritycentral.com/adversarial-attacks>.
- [17] A. Markov, B. Kim, *et al.*, “Attacks, defenses and evaluations for llm conversation safety: A survey”, *ar5iv*, 2023, Available at: <https://ar5iv.labs.arxiv.org/html/2402.09283>.
- [18] Hidden Layer, “The tactics & techniques of adversarial machine learning”, *Hidden-Layer*, 2023, Available at: <https://hiddenlayer.com/tactics-techniques-adversarial-ml>.
- [19] P. Blanchard, E. M. El-Mhamdi, R. Guerraoui, and J. Stainer, “Machine learning with adversaries: Byzantine tolerant gradient descent”, in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 2017, pp. 119–129.
- [20] T. Li, A. K. Sahu, A. Talwalkar, and V. Smith, “Federated optimization in heterogeneous networks”, in *Proceedings of Machine Learning and Systems*, 2020, pp. 429–450.
- [21] J. Xu, F. Wang, X. Xu, Q. Liu, and W. Shi, “Hybrid blockchain-based privacy-preserving federated learning for smart cities”, *Journal of Network and Computer Applications*, vol. 135, pp. 62–75, 2019.
- [22] Q. Li, B. He, and D. Song, “Decentralized federated learning: A collaborative approach without a central server”, in *Proceedings of the 11th ACM Workshop on Artificial Intelligence and Security*, 2018, pp. 61–68.
- [23] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic, “Qsgd: Communication-efficient sgd via gradient quantization and encoding”, *arXiv preprint arXiv:1610.02132*, 2017.
- [24] K. Mishchenko, E. Gorbunov, and P. Richtarik, “Distributed learning with dynamic node participation”, *arXiv preprint arXiv:1906.04878*, 2019.
- [25] D. Yin, Y. Chen, K. Ramchandran, and P. Bartlett, “Byzantine-robust distributed learning: Towards optimal statistical rates”, *arXiv preprint arXiv:1803.01498*, 2018.
- [26] C. Fung, J. Yoon, and I. Beschastnikh, “Mitigating sybils in federated learning poisoning”, in *Proceedings of the 39th IEEE Symposium on Security and Privacy (S&P)*, 2018.
- [27] S. P. Karimireddy, Q. Rebjock, S. U. Stich, and M. Jaggi, “Error feedback fixes signsgd and other gradient compression schemes”, in *Proceedings of the 36th International Conference on Machine Learning*, 2019, pp. 3252–3261.
- [28] M. Abadi, A. Chu, I. Goodfellow, *et al.*, “Deep learning with differential privacy”, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 308–318, 2016.

- [29] K. Bonawitz, V. Ivanov, B. Kreuter, *et al.*, “Practical secure aggregation for privacy-preserving machine learning”, in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2017, pp. 1175–1191.
- [30] H. Kasyap and S. Tripathy, “Hidden vulnerabilities in cosine similarity based poisoning defense”, in *2022 56th Annual Conference on Information Sciences and Systems (CISS)*, IEEE, 2022, pp. 263–268.
- [31] H. Zhang, Z. Yao, L. Y. Zhang, *et al.*, “Denial-of-service or fine-grained control: Towards flexible model poisoning attacks on federated learning”, *arXiv preprint arXiv:2304.10783*, 2023.
- [32] X. Li, N. Wang, S. Yuan, and Z. Guan, “Fedimp: Parameter importance-based model poisoning attack against federated learning system”, *Computers & Security*, p. 103936, 2024.
- [33] E. C. L. Luis Muñoz-González Kenneth T. Co, “Byzantine-robust federated learning through adaptive model averaging”, *arXiv*, vol. preprint arXiv:1909.05125, 2021. [Online]. Available: <https://arxiv.org/abs/1909.05125>.

# List of Figures

- 2.1 Architecture Comparison between CFL and DFL [6] . . . . . 8
  
- 5.1 Fedstellar User Interface . . . . . 32
- 5.2 Existing aggregation methods . . . . . 32
  
- 6.1 Analysis of cos attack and shejwalkar attack . . . . . 50
- 6.2 Analysis of Max Eigenvalue attack and Shejwalkar attack . . . . . 52
- 6.3 Analysis of Fisher Angle and shejwalkar attack . . . . . 53

# List of Tables

2.1	Categories of Attacks in Distributed FL . . . . .	14
2.2	Overview of Aggregation Methods in FL . . . . .	18
6.1	Attack Performance Comparison . . . . .	49