Client-side Validation for Efficient Model Poisoning Detection in Federated Learning

Master thesis by Benedikt Völker (Student ID: 2699765)
Date of submission: November 29, 2021

1. Review: Prof. Dr. Carsten Binnig
2. Review: Dr. Zheguang Zhao
Darmstadt

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Computer Science
Department

Data Management Lab

Erklärung zur Abschlussarbeit gemäß §22 Abs. 7 und §23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Benedikt Völker, die vorliegende Masterarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Fall eines Plagiats (§38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß §23 Abs. 7 APB überein.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, 29. November 2021

B. Völker

Abstract

Federated Learning has emerged as a common paradigm for privacy preserving machine learning, where multiple parties collectively train a Machine Learning model on their private data, i.e. without sharing the data with the other parties. However, recently several vulnerabilities have been found within the setting that can lead to the compromise of the integrity of the global model and the data privacy. Model poisoning attacks are among the most severe threat to the model integrity in Federated Learning, where the attacker seeks to tamper the global model by altering parts of the training process. Several defense mechanisms have been introduced to make Federated Learning resilient against these attacks. However, in various specific scenarios, like with heterogeneous data, these defense mechanisms remain less effective. This thesis instead introduces a general framework for secure Federated Learning that applies client-side training validation to reliably detect model poisoning attacks. Several randomized validation algorithms are proposed and analyzed for varying trade-off of efficiency, scalability and security guarantees. The evaluation shows that the framework is able to robustly detect model poisoning, even for tampering at the lowest level close to the numerical limit, while only incurring low overhead over the training time.

Contents

List of Figures

List of Abbreviations

API	Application Programming Interface
ML	Machine Learning
FL	Federated Learning
FedAvg	Federated Averaging
ANN	Artificial Neural Network
DNN	Deep Neural Network
FFNN	Feed-forward Neural Network
NN	Neural Network
MLP	Multi-layer Perceptron
SGD	Stochastic Gradient Decent
TEE	Trusted Execution Environment
SGX	Software Guard Extension
iid	independent and identically distributed
CPU	Central Processing Unit
GPU	Graphics Processing Unit
RAM	Random Access Memory
TPU	Tensor Processing Unit
TLS	Transport Layer Security
FA	Freivalds' Algorithm
GVFA	Gaussian Variant of Freivalds' Algorithm
SubMul	Sub-Matrix Multiplication
GDPR	General Data Protection Regulation
EU	European Union
RSS	Resident Set Size

1. Introduction

1.1. Context and Motivation

Moores' Law and the increase of computational power in computers has given rise to the application of Neural Network, a new class of Machine Learning (ML) method with general applicability. Neural Networks (NNs) have achieved astonishing results that have previously said to be impossible. Prominent examples are AlphaGo, which is the first computer program that was able to defeat the human world champion in the game Go [46], or self-driving cars of the automotive manufacturer Tesla, which are able to navigate crowded cities or run on highways with minimal to zero human interaction [61].

The successful application of Machine Learning (ML) is strongly dependent on the available data. In highly regulated markets or if data contains personal information, data of multiple parties can not be collected centrally for training a ML model. The handling of personal information is for example also been regulated by the General Data Protection Regulation (GDPR) of the European Union [38]. Federated Learning is a new collaborative machine learning scheme, that addresses this issue. It offers the possibility for multiple parties to train a Machine Learning model together, without disclosing the training data to the other parties.

However, it turns out that Federated Learning is not able to protect fully against data leakage and introduces several vulnerabilities that make the global model prone for attacks. Research has shown, that it is possible to infer training data from the gradients that are communicated during Federated Learning. Furthermore, the collectively trained Machine Learning model can be manipulated by the participants, so that it performs poorly, becomes biased to provide wrong predictions or learns unwanted behavior [41]. These attack are called poisoning attacks and are considered to be the most severe threat for the application of Federated Learning.

The security threats in Federated Learning are a great hurdle for its application in real-world scenarios. However, by implementing techniques that protect the privacy of the participants' data and ensure robustness of the learning process against poisoning attacks, Federated Learning can pave the way for the application of Machine Learning in many areas.

1.2. Problem Statement

Improving the privacy and robustness of Federated Learning is still an active field of research. The work on defense mechanisms against privacy issues has introduced several promising approaches that make use of for example differential privacy or multi-party computation. A greater challenge however is the defense against adversarial participants that perform poisoning attack on the global machine learning model.

Poisoning attacks take influence on the behavior of the Machine Learning model by training on malicious data samples or tampering with activations and gradients during model training. This can prevent the model from converging to a good performance, lead to miss-predictions or hidden functionality in the model. A poisoned Machine Learning model with malicious behavior can cause substantial harm, if it is applied in real-world scenarios. And most often, it is hard or even impossible to identify, if a Machine Learning model has been poisoned.

For the application of Federated Learning in many scenarios, it is absolutely necessary to apply defense techniques, that can reliably defend against poisoning techniques. Current defenses in Federated Learning are mainly implemented on a central server and use statistical measures on gradients to detect attacks. The analysis of gradients as the only resource to distinguish between malicious and honest model updates, is often not sufficient. Especially in cases, where the model is trained on heterogeneous data or data is changing over time, basing this decision on gradients alone has its flaws. Also, when attackers team up to pursue a malicious goal, gradient-based defense techniques have difficulties in detecting this.

It is therefore an open research problem to introduce a robust defense mechanism that is able to defend against poisoning attacks of various forms, independent of the data or model that is used for training.

1.3. Goals and Contributions

To make Federated Learning robust against malicious participants, this thesis introduces a new defense mechanism that is able to eliminate model poisoning attacks in the Federated Learning setting. In contrast to previous defense mechanisms, the detection of attacks does not take place on a central server. Instead, a trusted hardware-based validation component is applied locally at each participant. This way, the defense mechanism can take into account a wide variety of training information, and is able to reliably detect model poisoning attacks during training. While not only providing strong protection against model poisoning attacks, this thesis further investigates how client-side training validation can be implemented efficiently and what security guarantees can be derived from the defense mechanism. As a result, it is possible to achieve a guarantee for correct model training of 99%, while introducing only a minor training overhead. A thorough evaluation of the approach shows the great detection capabilities for even small injected errors. Furthermore, client-side training validation proves to perform efficiently in comparison to model training and has a feasible memory footprint for the application in trusted hardware modules.

1.4. Thesis Outline

The thesis is structured in the following way. First, the technological background and related work which is necessary to understand the content of this thesis is explained in chapter 2. This includes basics on Machine Learning, Federated Learning, Attacks on Federated Learning and Trusted Execution Environment (TEE). Section 3 continues by analyzing current defense mechanisms against model poisoning attacks and promising directions for robust federated learning. The core contribution of this work, the concept of client-side training validation using TEEs, is explained in chapter 4. Details about the implementation of the concept are provided in chapter 5. The evaluation in chapter 6 analyzes the proposed concept from different angles and points out the strengths and weaknesses of the concept. Finally, chapter 7 concludes on the presented work and points out promising avenues for future work.

2. Background & Related Work

This chapter provides all fundamentals and background information that are required to understand the content of this work. The main elements include the fundamental workings of Machine Learning in section 2.1 and Federated Learning in section 2.2. Furthermore section 2.3 gives an overview over attacks on Federated Learning and section 2.4 provides details about the concepts and technical capabilities of Trusted Execution Environments (TEEs).

2.1. Machine Learning Fundamentals

2.1.1. Definition of Machine Learning

The term Machine Learning (ML) describes computer programs that are able to improve on their own. A formal and widely used definition of Machine Learning (ML) was published by Tom Mitchell in his book Machine Learning in 1997 [47].

> **Definition:** A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks T, as measured by P, improves with experience E.

The definition is broad enough to include any computer program that is designated as learning but does not make any statement about how the program learns. However, it mentions three aspects that are essential for a learning program.

First of all the program has the goal of learning a certain task T. The task T is the problem that the program is trying to solve. Examples might be classifying emails as spam or automatically driving a car into a parking lot.

To be able to specify the performance of a learning program on a task T a measure of performance P is required. Related to the before stated examples, this could be the classification accuracy of spam emails or a score for how well the car was parked.

For improving on a certain task T the program further needs some kind of observation or feedback called experience E, which the program learns from. Experience E can be direct or indirect feedback regarding the programs actions. Direct feedback for example would be, if an email is actually spam or not. Indirect feedback would be the moving sequence and final position of the car.

2.1.2. Types of Machine Learning

In mathematical terms in Machine Learning a computer program is trying to learn a target function f that maps input variable X to output variable Y [50].

$$f : X \rightarrow Y \tag{2.1}$$

The approximation of the target function f is captured in a Machine Learnings model. Finding the most optimal function f is an optimization problem. Searching for an optimal solution means trying to improve upon the performance measure P (also called objective function). Determined by the performance measure, the optimization problem can be a minimization or maximization problem. In the case of minimization, the objective function is also called the loss function or error function [29].

Machine Learning literature usually distinguishes between three main types of learning: Supervised Learning, Unsupervised Learning and Reinforcement Learning.

Supervised Learning describes a setting, where a model learns a direct mapping from inputs x to outputs y from a training set of inputs and outputs $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$. Input x can be any type of information represented in a data structure like images, sentences or time series data. Output y is usually a categorical or real-valued variable. Supervised learning with categorical output is usually referred to as classification, whereas supervised learning with real-valued output is call regression. [50]

Unsupervised Learning is applied when a model is trained on a set of unlabeled samples $\mathcal{D} = \{x_i\}_{i=1}^{N}$. The model is intended to find interesting patterns in the data set like clustering users into user groups or learning similarities between words. Without provided patterns to look for and lack of a clear error metric, Unsupervised Learning is a less specific learning task than Supervised Learning. [50]

Reinforcement Learning refers to a learning task where the model is supposed to learn by interacting with a certain environment. This means mapping situations to actions in a way to maximize the performance or reward. The model is not told what action results in the highest reward and has to discover that itself by acting in the environment [60]. Reinforcement Learning can for example be applied in a robot vacuum cleaner, that over time learns the terrain and optimal route to clean a room.

2.1.3. Generalization in Machine Learning

An important aspect when training a machine learning model is the models ability to perform well on unseen input data. This means a model is not supposed to simply memorize the training data but should be able to generalize to new data.

As an example, assume an image classifier which is used to detect if the input image shows a dog. If the classifier can only detect dogs on images it has been trained on, the classifier is limited in application and practicability. However, by learning the aspects of how a dog looks like, it is capable of detecting dogs in previously unseen images.

Thus, it is important to ensure generalization of a machine learning model when it is trained. To do so, the model performance is usually tested on two distinct data sets. The data set used for model training is called training set and the loss value on the training set is called training error. In addition

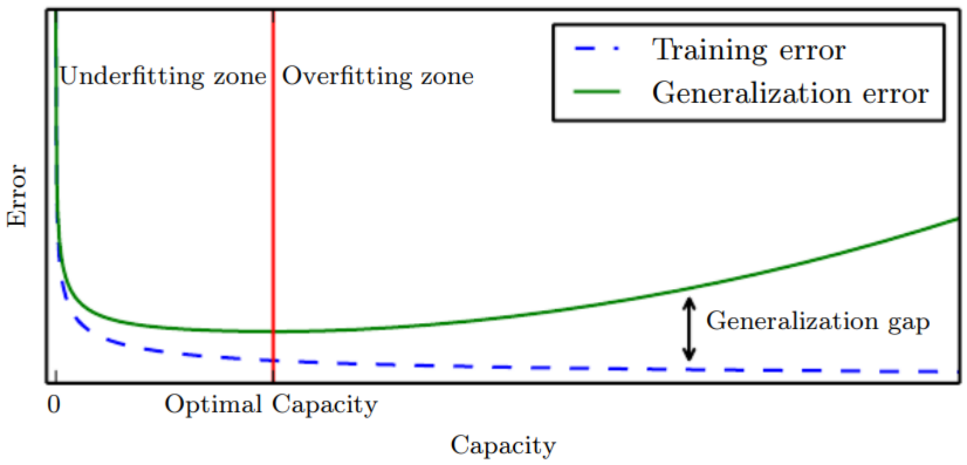


Figure 2.1.: Overfitting & Underfitting of a Machine Learning model relative to the model capacity. [29]

to the training set, the model is further evaluated on the test set, which gives the test error. The training set and test set contain data samples which have been drawn from the same data generation process. However, the data sets have no intersection of data samples $\mathcal{D}_{train} \cap \mathcal{D}_{test} = \varnothing$. [29]

The degree to which the Machine Learning model performs well on the task is determined by its capability to reduce the training error and to reduce the difference between training error and test error. If the training error is too large, the model is not adapting enough to the training data. This is called underfitting. If the difference between training error and test error grows too large, the model is adapting too much to the training data. This is called overfitting.

The ability of a model to adapt to training data is influenced by its capacity. The model capacity describes the ability of the model to represent a wide range of functions. For example, a polynomial of degree 2 ($y = w_1x^2 + w_2x + b$) has a higher capacity than a polynomial of degree 1 ($y = wx + b$), because it is able to represent a larger variety of functions. Hence, the number of input features with associated parameters (weights) influences the capacity of the model. The relation of model capacity and generalization capability of a model is illustrated in Figure 2.1.

Besides the model capacity, the convergence of a machine learning model can further be influenced by so called regularization techniques. Regularization means the modification of a machine learning model to reduce its test error but not its training error [29].

2.1.4. Artificial Neural Networks

One popular approach for machine learning is the application of Artificial Neural Networks (ANNs) (often referred to as Neural Networks). Because of their high capacity, Neural Networks (NNs) and especially Deep Neural Networks (DNNs) can be applied to learn very complex functions. In theory they are able to represent any function and can be optimized in a relatively stable way to perform a certain task [29]. This subsection provides details about the fundamental concepts behind Neural Networks. These basics are needed to understand the attacks and defense mechanisms for Federated Learning, which are introduced and discussed in this work.

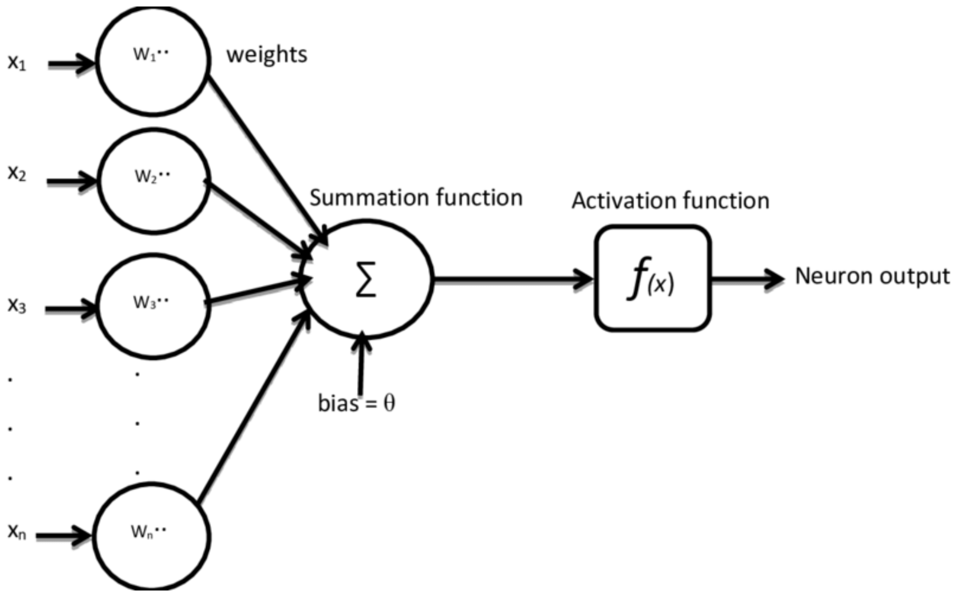


Figure 2.2.: Illustration of an artificial neuron. All inputs are weighted with an individual weight and summed up with the bias. The weighted sum of inputs is fed into the non-linear activation function, which produces the output of the artificial neuron. (image source: [67])

Artificial Neuron

Neural Networks are inspired by the observation that biological learning systems, like the human brain, are constructed of complex webs of interconnected neurons. To mimic the biological systems, Artificial Neural Networks are constructed in a similar way. A neuron is a small unit that has several inputs and maps them to a single real-valued output [47]. Figure 2.2 depicts a single neuron and its components.

Each input of an artificial neuron is multiplied with a weight. The weight regulates the influence that an input has on the neuron's output and should be adjusted according to the importance of the input. During neural network training, the weight values are updated in order to improve the performance of the model.

In the next step, all weighted inputs are summed up with an additional weight, the bias b. The result of the summation is the pre-activation. The pre-activation is the input of the activation function $\sigma(\cdot)$ which is a differentiable, non-linear transformation [32]. In mathematical terms, a neuron can be described with

$$a = \sigma\left(\sum_{i=1}^{N} w_i x_i + b\right) \tag{2.2}$$

When multiple artificial neurons are stacked together they form a layer. Computations of the activations for all neurons in a layer are combined and performed as a matrix vector multiplication. The weights of all neurons are stacked together and form a matrix whereas the collections of all inputs and biases each result in a vector. The computation of the output vector of a layer can be written as

$$\vec{a} = \sigma(W\vec{x} + \vec{b}) \tag{2.3}$$

In a similar way, multiple training samples can be combined to a batch. To compute the activations for all elements of a batch collectively, the input vectors of samples are stacked together and form a matrix. As a result, the computation of layer outputs for the whole batch involves a matrix-matrix multiplication. The output also has the form of a matrix.

$$A = \sigma(WX + \vec{b}) \tag{2.4}$$

Feed-forward neural networks

Feed-forward Neural Networks are a combination of n layers of interconnected artificial neurons. The special case of a Neural Network with only one layer is called perceptron, which is equivalent to equation 2.3. If a Feed-forward Neural Network has more than one layer it is often referred to as Multi-layer Perceptron (MLP). Layers are stacked together such that the output of one layer is the input of the subsequent layer. An MLP with n layers is said to have $n - 1$ hidden layers and one output layer. Inputs are not counted as a layer. Figure 2.3 shows a neural network with 2 layers, one hidden layer and one output layer.

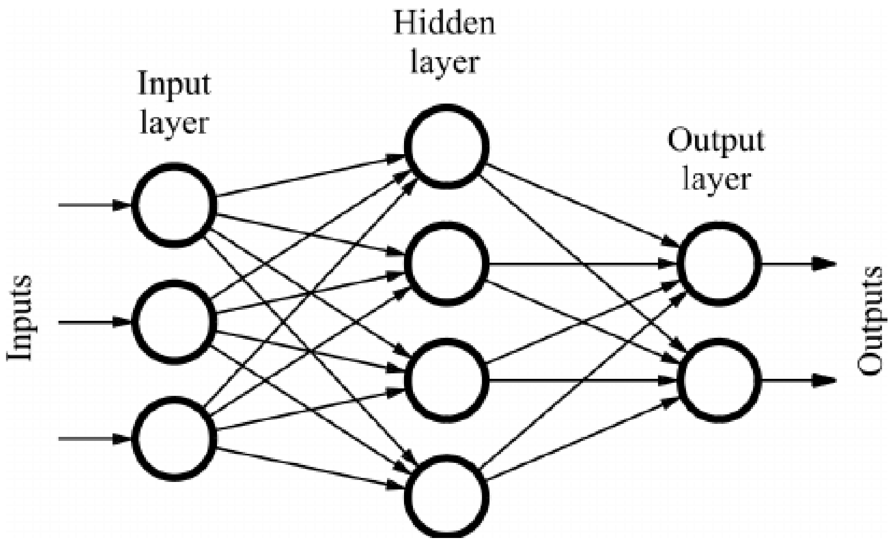


Figure 2.3.: Illustration of a 2-layer feed-forward neural network. It has one hidden layer and one output layer. The layer dimensions are $(3, 4)$ and $(4, 2)$. The inputs of the neural network do not count as a layer. (image source: [54])

The process of computing the output of a Feed-forward Neural Network based on an input vector is called the forward-pass. The output is the prediction of the Neural Network and is supposed to match the label which belongs to the input vector. During creation of the Feed-forward Neural Network, all weights are assigned a random value. Of course, these initial weights produce arbitrary outputs and the Neural Network is not able to make correct predictions. To improve the prediction capabilities of the network and make it learn a certain task, the Neural Network needs to be trained. How the training of weights in Neural Networks works is explained below.

Training of Neural Networks

Training a Neural Network is often a supervised learning task and requires a data set consisting of a collection of data samples and their respective labels. Labels define the expected output of the neural network. The error of the neural networks prediction is calculated by the error function $E(\cdot)$ (also called loss function). The error function is the performance measure P which determines how well the Neural Network is able to predict the labels.

The goal during training is to minimize the prediction error of the model which is equivalent to finding a minimum of the error function. To find a solution for this optimization problem, gradient based methods are applied. The most prominent of which is Stochastic Gradient Decent (SGD). Gradient based methods minimize the error function by repeatedly taking small steps in the direction of the negative gradient [4].

Training of a Feed-forward Neural Network with SGD involves several steps: Forward propagation (forward-pass), loss computation, backward propagation (backward-pass) and gradient update [4].

Forward propagation means computing the neural network output for a given input. The output of the last layer (layer n) is the prediction of the neural network and is supposed to match the label of the input. The error function $E(\cdot)$ determines the error between the prediction and the label.

To determine the direction for optimization, the derivative of the error function with respect to the output of the Neural Network $\frac{\delta E}{\delta A_n}$ is computed. A_n refers to the output of layer n (last layer). The gradient gives the direction of the steepest ascent.

For learning it is necessary to determine the changes that need to be done to the weights to reduce the prediction error. This means to obtain the gradients of the error function with respect to the weights $\frac{\delta E}{\delta W}$. By applying the chain rule, the gradient at the Neural Networks output can be propagated backwards through the network. The chain rule makes a statement about the derivative of a combination of two functions, whose derivatives are known. Assuming two differentiable functions f and g where $y = g(x)$ and $z = f(g(x)) = f(y)$. Then the chain rule states that [29]

$$\frac{\delta z}{\delta x} = \frac{\delta z}{\delta y} \cdot \frac{\delta y}{\delta x} \tag{2.5}$$

The chain rule can also be applied on Equation 2.4. This makes it possible to compute the gradient of the error function with respect to the layer input $\frac{\delta E}{\delta X}$

$$\frac{\delta E}{\delta X} = \frac{\delta E}{\delta A} \cdot \frac{\delta A}{\delta P} \cdot \frac{\delta P}{\delta X} \tag{2.6}$$

where

$$\frac{\delta A}{\delta P} = \sigma'(P) \text{ and } \frac{\delta P}{\delta X} = W \tag{2.7}$$

together this results in

$$\frac{\delta E}{\delta X} = \frac{\delta E}{\delta A} \cdot \sigma'(P) \cdot W \tag{2.8}$$

In a similar way, the gradient with respect to the weight is obtained

$$\frac{\delta E}{\delta W} = \frac{\delta E}{\delta A} \cdot \sigma'(P) \cdot X \tag{2.9}$$

Because the input of a layer is the output of the previous layer, the gradient at the input of layer i $\left(\frac{\delta E}{\delta X_i}\right)$ is also the gradient at the output of layer $i-1$ $\left(\frac{\delta E}{\delta A_{i-1}}\right)$. This way, the gradients of the error function with respect of the weights of each layer can be derived.

In the last step, the weights are updated using the gradients $\frac{\delta E}{\delta W}$. This is done my subtracting a fraction of the gradients from the previous weights. The degree of change is determined by the learning rate η.

$$W' = W - \eta \cdot \frac{\delta E}{\delta W} \tag{2.10}$$

Subtraction is used because the gradient points in the direction of the steepest ascent and optimizing for the loss function is a minimization problem [29]. The direction of steepest descent is the inverse of the direction of steepest ascent.

Computational Complexity of Neural Networks

The training and application of Neural Networks is known to have a high demand in computational resources. Because the training of large models on Central Processing Units (CPUs) is not feasible, computations are often accelerated using GPUs or Tensor Processing Units (TPUs) [62]. The reason for the high computational demand are the numerous matrix operations during model training.

The execution of element-wise operations, like applying the activation function, has a quadratic complexity $\mathcal{O}(n^2)$. For example the summation of two matrices $A, B \in \mathcal{R}^{n \times m}$ involves $n \times m$ additions. The computational complexity of Neural Network is however greater influence by the computation of matrix-matrix multiplications. To compute the matrix product of two matrices $A \in \mathcal{R}^{n \times m}, B \in \mathcal{R}^{m \times k}$, $n \times m \times k$ multiplications and $(n-1) \times m \times k$ additions are required. It is computed like

$$c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj} \tag{2.11}$$

This results in cubic computational complexity $\mathcal{O}(n^3)$ for the computation of matrix products. Researchers have worked for decades to reduce the complexity of matrix product calculation. So far the best know method to compute matrix products has an upper bound of $\mathcal{O}(n^{2.3728639})$. It is an open question, if an algorithm for matrix-matrix multiplication with the complexity of $\mathcal{O}(n^2)$ can be found [59, 27]. So far however, matrix-matrix multiplication remains to be the most compute intensive operation in Neural Networks.

This work presents an approach to detect model poisoning through client-side training validation, that requires the verification of matrix operations during Neural Network training. For the efficiency of this approach, it is necessary to find a way to verify matrix-matrix multiplications with low computational complexity. Section 4.3 elaborates on suitable methods to implement a validation that does not introduce a huge computational overhead.

2.2. Federated Learning Fundamentals

Federated Learning is a method for communication-efficient learning of Deep Neural Networks from decentralized data. It was introduced in 2017 by McMahan et al. from Google Research [45]. Federated Learning addresses privacy issues in scenarios where multiple parties want to train a Neural Network together. It has the advantage that several client machines can collectively train a Neural Network without disclosing their local private data to other participants. This is important, especially if the data is critical for business or contains personal information.

Prior to Federated Learning, data of all participating clients would be collected on a central server. The server would train the neural network with the collected data and distribute the final model. This exposes the raw data of all clients to the server, which needs to be fully trusted by the clients.

In Federated Learning the training is coordinated by the server, but training is executed on the client machines. The training process is synchronized and performed in rounds. There is a fixed number of clients K that are participating in Federated Learning. Figure 2.4 visualizes the federated learning setup.

At the beginning of each round, the server randomly selects a fraction $C \subseteq K$ of the clients and sends each of the clients the current global model. Each client trains the model on his own private data and sends the model updates (gradients) back to the server. Finally the server updates the global model with the aggregated gradients of the clients. The individual steps involved in Federated Learning are shown in figure 2.5.

For aggregating the client updates into a new global model, the original paper uses a method called Federated Averaging (FedAvg). Followup research has investigated different aggregation schemes which are partly discussed in section 3.2. FedAvg has three key parameters

- C - the fraction of clients that perform computation on each round
- E - the number of training passes each client makes over its local data set on each round
- B - the local mini-batch size used for the clients updates

FedAvg is computed as the weighted mean of client weight updates w^c_{t+1}, as defined in the following equation. Thereby, n_c is the number of data samples used for training at client c and n is the total number of data samples.

$$w_{t+1} = \sum_{c=1}^{C} \frac{n_c}{n} w^c_{t+1} \tag{2.12}$$

2.3. Attacks on Federated Learning

Moving the responsibility for model training to the clients, increases opportunities for malicious clients to observe or manipulate the training process, especially, as clients have direct influence on the model updates. In the central model training approach, malicious clients can only manipulate the training data (data poisoning). Incorrect labels or unusual data samples can be detected by applying statistical methods on the server. With Federated Learning, however, data poisoning is harder to detect as the

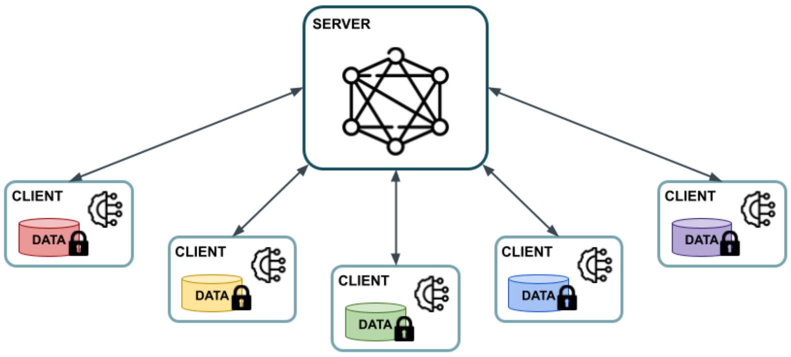


Figure 2.4.: Illustration of a Federated Learning setup with 5 clients. Each client has a private database, which he uses for training on its local hardware. The server maintains the global model and coordinates the Federated Learning process.

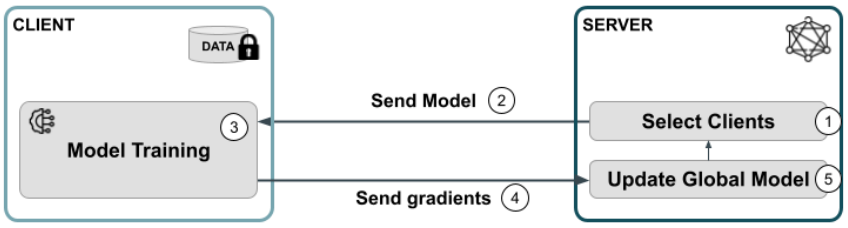


Figure 2.5.: Federated Learning is performed in 5 steps. The server ① selects a number of clients for a training round and ② sends them the current model. The clients ③ train the received model on their local data and ④ send their model update back to the server. The server then ⑤ aggregates the model updates from the clients and starts a new round based on the updated model.

server has no access to the clients data. In addition, clients can tamper with the training process (model poisoning). Model poisoning also increases the amount of damage that can be achieved through poisoning attacks [3].

This thesis presents a novel approach to defend against model poisoning attacks, which are a severe threat in the application of Federated Learning. To understand the danger of these attacks and how the approach of this thesis (see chapter 4) defends against it, it is critical to gain a decent understanding of attacks in Federated Learning. The following sections introduce the reader to the basic concepts of attacks in Federated Learning and describe the procedures and techniques behind them.

2.3.1. General Attack Types and Attack Surface

Federated Learning involves more participants in the model training process compared to the centralized learning approach. This means, that more actors have direct influence on the model training procedure and can abuse this to attack the Federated Learning system. It also accounts for an increased transfer of information between participants which can lead to leakage of private information. This section gives an overview over malicious agents, attacks types and the attack surface in Federated Learning.

Types of Attackers

Attackers can be distinguished between insiders and outsiders. Insider attacks are carried out by participants of the Federated Learning system. Both, clients and server, can act maliciously and disturb the training process or infer private information. Outsiders are foreign entities, which do not play any role in the Federated Learning system. They try to manipulate messages or retrieve information for example by tapping communication channels. In general insider attacks are more severe than outsider attacks, due to the superior capabilities of the attacker [41].

Federated Learning needs to be robust against failure or attacks of a single or multiple clients. A single attacker can act maliciously in his normal radius of influence or join the Federated Learning setting multiple times by impersonating multiple clients. The latter, called a sybil attack [23], increases the influence of the client on model training and results in stronger attacks. By working together, the participants can also gain significantly greater influence [26]. The cooperation of attackers is called byzantine attack and has a similar effect like sybil attack. Both attacks are easier to carry out when the number of participants is small, because the number of malicious clients required to succeed in the adversarial goal is smaller as well.

Attackers can act in a semi-honest/passive or malicious way. Being semi-honest, attackers try to retrieve private information about other clients without deviating from the Federated Learning protocol. Passive attackers use the information which they receive during the normal learning process. A malicious client is actively trying to manipulate the training process or the Federated Learning system to pursue his adversarial goal. An insider attacker for example could deviate from the Federated Learning protocol by changing hyper-parameters or changing the learning behavior [41]. An outsider attack could perform a man-in-the-middle attack.

This thesis focuses on dealing with malicious insider attacks performed by single or multiple adversarial clients. These attacks are the most severe threat to Federated Learning and can have a profound impact on the success of collaborative model training.

Types of Attacks

The attacks on Federated Learning can be classified into two main classes, poisoning attacks and inference attacks. The goal of poisoning attacks is to bias the model such that it adopts a behavior intended by the adversarial client or preventing the model to learn at all. Inference attacks target the privacy of participants with the goal of retrieving information that is not supposed to be shared like class representative, properties of the training data and even labels and training samples [41].

An attack can occur during training and prediction phase. While training, the participants have access to gradient information and can influence the training process. When predicting with a deployed model, attackers can try to cause the model to produce wrong outputs or collect information about the model. Attacks during prediction phase do not tamper with the model, but explore its functionality and characteristics [41].

Poisoning attacks during training phase can create the most damage to the Machine Learning model and make the whole model training obsolete. The approach presented in this thesis therefore aims to defend against these attacks. How these poisoning attacks work is explained in the following.

Poisoning Attacks

Having control over the model training procedure in FL gives malicious clients a range of possibilities to harm the collaboratively trained neural network. This section gives an overview about the types of poisoning attacks. In general poisoning attacks can be divided into targeted an untargeted poisoning attacks [33].

Untargeted Poisoning Attacks
Untargeted / Indiscriminate poisoning attacks aim for a rather broad and general goal like harming the model performance [33]. These attacks do not try to provoke a specific model behavior, but prevent the models convergence or make it converge to a bad optimum. This type of attack can often be detected with model update statistics or model performance evaluation on the server [72].

Targeted Poisoning Attacks
In contrast to a general destructive behavior, targeted model poisoning aims at producing a specific outcome. Here, the malicious agent has an adversarial objective which it tries to optimize [41]. The goal of targeted poisoning attacks is to make the model learn unwanted behavior that is not intended by the main objective. On the one hand, this can include the models behavior according to the main task, for example causing misclassification or misprediction for certain data samples. On the other hand, an attacker might want to implement a hidden functionality by inducing a backdoor into the model [1].

2.3.2. Functionality of Poisoning Attacks

In federated Machine Learning, clients have a larger attack surface than in centralized Machine Learning. This is caused by the fact, that responsibility for model training is transferred to the clients. Furthermore, in Federated Learning, the central server has no option to analyze the training data as it is located on the clients machines.

As a result, there are two ways to manipulate the ML model. First of all, a client can tinker with the data that he is using for training. This is called data poisoning. Secondly, the client can also tamper with the model training procedure or model update creation, which is called model poisoning. This section gives a brief overview of the attack surface and techniques which can be used for adversarial actions and discusses the goal of staying undetected.

Data Poisoning

Data poisoning describes the manipulation of training data. In general, data poisoning attacks can be distinguished into two types. Malicious agents can manipulate the data samples (e.g. images) if they have no control over the labeling of the data, which is called clean-label data poisoning [57]. If labeling of the data samples is controlled by the malicious agent, it can use incorrect labels for data samples. As an example, imagine labeling a stop sign as a speed limit for a traffic sign recognition task. This method is called dirty-label data poisoning. Both approaches can be applied to reduce model performance, achieve targeted misclassification or introduce a backdoor into the model [10]. However, data poisoning attacks happen to be less effective than model poisoning attacks in FL [3].

Model Poisoning

Model poisoning describes the manipulation of the training process. A client can manipulate the overall training process by changing the hyperparameters for model training. This includes for example the number of epochs, number and size of batches, the learning rate or even exchanging optimizer or loss-function. Training with different hyperparameters can lead to clearly different results.

Besides changing hyperparameters, a malicious client might also manipulate model gradients in a way that it helps to achieve the adversarial objective. One option is for example gradient boosting [3], where gradients for the adversarial objective are scaled, so that they have a larger influence on the overall model. Gradient boosting works in combination with data poisoning, where the gradients obtained from malicious data samples are boosted for a successful targeted poisoning attack. In contrast, an example for untargeted model poisoning would be to simply add random noise to all gradients such that models convergence is harmed or prevented.

Stealth of Attacks

Besides the main goal of a poisoning attack, a malicious agent has to act stealthy. If the agent is detected performing a poisoning attack, the agent will be excluded from the Federated Learning setting and is not able to participate anymore.

One method of detecting malicious agents is to check model accuracy on the test/validation set. If the accuracy is decreased by an update, it is a sign for a poisoning attack. Therefore, in the case of targeted model poisoning, adversaries try to maintain high accuracy on the main task while optimizing for the adversarial objective.

Poisoning attacks can also be detected by comparing new model updates against model update statistics. To increase stealth for poisoning attacks, clients strive to keep the weight update distribution similar to the weight updates of other clients. An attacker can for example alternate between training with normal and malicious data samples, whereby the model updates become similar [3].

2.4. Trusted Execution Environment (TEE)

Data can exist in three different states: in transit, at rest or in use. The encryption of data, which protects it during transit and at rest, is already common in many applications. Available cryptographic libraries make encryption applicable in any use-case and accessible for every developer. The protection of data during use is, however, a new paradigm that has come up in recent years [13].

A common approach to create security in an untrusted environment is to isolate software execution at run-time. This way, sensitive data can be processed and protected against unauthorized use. Isolation is often achieved through virtualization with the help of hypervisors. However, hypervisor often have a large code base, which is vulnerable to attacks [49].

Recently, new alternatives to hypervisors have been introduced. Trusted Execution Environments (TEEs) pair the concept of isolated execution with hardware-assisted technologies. The secure hardware modules ensure performance and security while exposing a smaller attack surface. TEEs are often general purpose and can run any code [49]. The most prominent TEEs are currently Intel Software Guard Extension (SGX) [44, 19] and AMD Memory Encryption Technology [36].

TEEs provide a level of assurance for data integrity, data confidentiality and code integrity. Data integrity means that while data is processed by the TEE, no unauthorized entity is able to alter the data. Data confidentiality prevents unauthorized entities to view data while it is processed by the TEE. Assuring code integrity means that code inside the TEE can not be replaced or modified by unauthorized entities [13].

Besides these core features, TEEs may also provide additional benefits [13] including:

- **Code Confidentiality:** Preventing unauthorized entities to view the source code that is executed inside the TEE.
- **Authenticated Launch:** Enforcing authorization or authentication checks before launching a process request. Requests that fail the initial check are refused.
- **Programmability:** The ability to run arbitrary code inside the TEE. Some TEEs only support a limited set of operations or even just fixed code that was set during manufacturing.
- **Recoverability:** Presence of a mechanism to recover from a non-compliant or potentially-compromised state. This might include retrying execution with updated components if the use of outdated software components has been rejected.
- **Attestability:** Providing evidence for the state a TEE is currently in. The evidence is verifiable by other parties.

The great security benefits of TEEs also have their cost. Depending on the technology and the related security guarantees a TEE is limited in terms of memory and execution time. AMD Memory Encryption Technology only introduces a minor overhead and can use up to available system memory. However, it does not provide memory integrity protection. Intel SGX on the other side, is limited to 128 MB of memory and introduces a larger overhead of 3-5 times the execution duration. Nonetheless, it is suitable for security-critical applications because it provides a slim and protected enclave gateway, imposes memory access control and provides memory integrity protection [49]. An important aspect to consider when making use of TEEs, is the overhead that is introduced by the context switch, when a program enters the TEE [64].

The application of TEEs is a promising method to build trust in an untrusted environment. The concept proposed in this thesis makes use of the security guarantees of TEEs to make Federated Learning more robust against model poisoning attacks.

3. Analysis

This section addresses the threats to Federated Learning posed by the various types of attacks. It also looks at different kinds of countermeasures that have been proposed in research literature and analyses their effectiveness to prevent these attacks. Finally, this section concludes in a promising avenue that lays the basis for the concept described in chapter 4.

3.1. Threads to Federated Learning

Federated learning can be the target of various attacks. Their basic functionality is described in the background section 2.3. All of the attacks have an unwanted effect, however, some are more severe than others. Insider attacks have already be identified as more critical than outsider attacks because of the superior influence on the Federated Learning system.

In a similar way, attacks performed during model training have greater influence on and insights into the model. Only by attacking during training phase, the client can access and poison the gradients, which can lead to leakage of private information or manipulation of the global model. The adversary also has access to the model architecture and structure, that could potentially be used for attacks during inference time.

Insider attacks can originate from the server or the participating clients. The server is selecting the clients for each training round and aggregates the gradients of the clients into the new model. By playing a central role, the server has detailed insights on each clients gradients and can incorrectly aggregate the gradients. Clients train the model locally on their data and send the gradients back to the server. The clients can tamper with the training data and learning process and change the model behavior or infer information from the model changes over time. Both roles are critical for the Federated Learning system, however, the actions performed by the clients are more complex and harder to review. Furthermore, there are always multiple clients and one server where it is easier to implement harsh surveillance for one server than all the clients. The rather simple operations performed by the server could also be performed inside a TEE.

Attacks that are performed by malicious clients are more likely to succeed, because a malicious client has more options to retrieve information or influence the model performance by not sticking to the Federated Learning protocol. The semi-honest client sticks to the protocol and can therefore just observe what is exposed to him during the model training process.

Clients that act maliciously can perform both poisoning and inference attacks. For poisoning attacks, clients are able to use malicious data sets for training or poison model updates before sending them to the server. Also in the case of active inference attacks, the client would tamper with the model updates to receive more information about other clients in the next round. Research has shown that data poisoning alone turns out to be ineffective in the Federated Learning setup [3]. Thus for active poisoning and inference attacks, a client would tamper with the model training process.

The above discussion shows that a maliciously acting client which performs an active model poisoning or inference attack is the most severe threat to the Federated Learning system. The same conclusion holds for the application of byzantine or sybil attacks that are performed by multiple cooperating clients. For a secure application of Federated Learning it is therefore crucial to be able to detect and eliminate model poisoning attacks.

The following section sheds light on existing defense mechanisms that aim to protect against model poisoning attacks and describes how the defense mechanisms work.

3.2. Defense Mechanisms for Federated Learning

Faced with a variety of potential attacks on Federated Learning, researchers have investigated possible defense strategies to make Federated Learning more secure and robust. This section examines several of these defense strategies and discusses their capabilities and limitations to identify the requirements for a novel approach to defend against model poisoning.

3.2.1. Requirements for Defense Techniques

Since the introduction of Federated Learning in 2017 [45], a number of defense techniques that counter poisoning attacks on Federated Learning have been proposed. Most of these defense techniques are specialized for certain attacks and scenarios, in which they function well. To achieve a better understanding on different aspects that are required for private and robust Federated Learning, this section introduces defense goals and parameters that need to be taken into account when designing defense mechanisms.

Lyu et al. [42] identify six goals for Federated Learning defense mechanisms, which are necessary to preserve privacy and robustness. In the ideal case, defense mechanisms cover multiple or all defense goals. Although it might be difficult to address all objectives and aspects in one defense technique, a secure Federated Learning system could be achieved by combining multiple defense techniques. The goals for private and robust Federated Learning are:

1. fast algorithmic convergence
2. good generalisation performance
3. communication efficiency
4. fault tolerance
5. privacy preservation
6. robustness to targeted, untargeted poisoning attacks, and free-riders

All of these goals should be achieved under various circumstances. Defense techniques have to work well for model training on both, independent and identically distributed (iid) and non-iid data. In addition, the six goals need to be achieved in scenarios with one and also multiple cooperating attackers.

The following section gives an overview over existing defense techniques and the concepts behind them.

3.2.2. Popular Defense Techniques

In the last years a number of techniques have been proposed, which are supposed to make Federated Learning more secure and robust. The main goal of these defense mechanisms is to avoid poisoning attacks on the collectively trained model or increase privacy by countering active inference attacks. This section introduces a selection of noticeable defense techniques against model poisoning and describes how they function.

Recent attempts to mitigate model poisoning attacks in Federated Learning can mainly be divided into Coding-theoretic Defenses, Statistical Defenses and System-based Defenses [66].

Coding-theoretic Defenses

Coding-theoretic defenses neglect the need for statistical assumptions by introducing redundancy across multiple clients [66].

DRACO [9] is a system that distributes data redundantly across clients. The clients compute gradients for their data set and encode the data. The encoded gradients are sent to the server. The server decodes the gradients, which reveals adversarial gradients because of redundancy.

DETOX [55] is also a redundancy based framework that uses a hierarchical structure. It works in two steps. In the first filtering step, the server randomly assigns clients to groups and provides all clients in a group with the same data. Clients compute the gradients for their data set and send them to the parameter server. The server takes the majority vote of the gradients, which filters out a large fraction of malicious gradients. In the hierarchical aggregation step, the server assigns the filtered gradients to new groups and applies an aggregation method to each group. The aggregated group gradients are again accumulated with a robust aggregation method, resulting in the final gradients.

Both methods are able to significantly reduce the possibilities for byzantine attacks on the global model. However, both methods require the distribution of data which violates the goal of privacy preservation and introduces additional communication overhead.

Statistical Defenses

The majority of statistical defense mechanisms are robust aggregation methods that leave out or mitigate malicious updates by analyzing gradient update similarity [66].

KRUM [7] analyses the clients gradients using the euclidean norm. It selects the one gradient that minimizes the sum of squared distances to its $n - f$ closest gradients $\sum_i ||g_i - g||_2$. Where n is the total number of clients and f the number of adversarial clients. Krum assumes that $2f + 2 < n$. Where

Krum only selects the one gradient vector with the best score, Multi-Krum selects the m best gradient vectors and computes the average of these.

Robust Federated Aggregation (RFA) [53] implements robust gradient aggregation by computing the geometric median for the gradients. The geometric median is the minimum of the sum of weighted euclidean norm $\sum_i a_i||g_i - g||_2$, where weight $a_i > 0$ and $\sum_i a_i = 1$. Compared with vanilla federated learning, RFA introduces thrice the communication cost.

Coordinate-wise Median (CM) & Coordinate-wise Trimmed Mean (CTM) [68] are two robust distributed gradient descent algorithms based on median and trimmed mean operations. Both methods aim at achieving optimal statistical performance, which they achieve under mild conditions.

FoolsGold [26] is a defense technique that adapts the learning rate of individual clients based on their contribution similarity. The method is able to cope with several sybil attackers and makes less assumptions about the clients and their data. The idea behind FoolsGold is that cooperating attackers that aim at achieving a specific adversarial goal, submit more similar gradients.

All of the statistical defense techniques are server-side techniques that require the parameter server to have access to the plain gradients submitted by the clients. This design aspect requires an honest server and limits the application of multi-party computation or differential privacy techniques to preserve clients privacy.

Furthermore, most statistical techniques usually require expensive computation and scale super-linearly with the number of clients [55]. They further make the assumption, that the training data is independent and identically distributed (iid) across clients [2].

System-based Defenses

More recently, defenses have been proposed that aim to eliminate model poisoning attacks by detecting or preventing them using system-based approaches.

Ensemble Federated Learning (EFL) [8] trains an ensemble of models on subsets of the clients. For that EFL randomly assigns clients to groups of size k without replacement. Each group trains a separate neural network with the standard Federated Learning protocol. For predictions with the model ensemble, EFL uses the majority vote of all models. The authors show that EFL is provably secure against a small number of malicious clients.

Client-side Cross-Validation [70] is an approach were clients evaluate each others gradient updates on their local data. Model updates and evaluation results are collected by the server. The global model is updated with the local updates weighted by their evaluation results. To cope with non-iid data, a dynamic client allocation mechanism selects the most suitable clients for evaluation. The approach also protects the privacy of clients by integrating differential privacy. Client-side Cross-Validation tries to reduce communication by randomly selecting a small set of clients to evaluate each model. However, it still introduces a major communication overhead compared to vanilla Federated Learning.

Privacy-preserving Federated Learning (PPFL) [48] uses secure aggregation and layer-wise model training inside a Trusted Execution Environment (TEE). Layer-wise training limits the overhead introduced by the TEE and enables early stopping, when great model performance is already achieved by training only a subset of layers.

TrustFL [69] uses a Trusted Execution Environment (TEE) to efficiently enforce that clients are training and not submitting arbitrary gradients. After the client has performed model training, it is supposed to build a Merkle Hash Tree (MHT) from the hash values of the model parameters in order. The MHT is passed over to the TEE, which randomly requests input parameters and hash of output parameters for a specific training step. By recomputing gradients within the TEE, the training step is validated.

Client-side defenses that build on TEEs can ensure correct and trusted code execution on an untrusted host, without introducing high communication cost. PPFL can eliminate model poisoning by performing layer-wise training completely within the TEE and improves training duration through early-stopping. Without the application of early-stopping, this approach, however, has a huge training overhead. TrustFL can ensure that a client is training, but is not able to detect poisoning attacks during training phase.

Based on the insights about current defense strategies in Federated Learning, the following section derives a promising avenue for robust Federated Learning and lies the basis for the proposed concept in chapter 4.

3.3. Discussion of Defense Mechanisms

Most of the proposed defense techniques against model poisoning in Federated Learning are server-side approaches. The approaches either leverage redundant gradient computation (coding-theory) or statistical methods to detect adversarial updates. All of these methods are based on the gradients submitted by the clients.

Gradients are largely dependent on the loss function and the data that is used for training. Previous statistical defenses worked well on homogeneous iid data, but are vulnerable in edge cases of data that is unlikely to be part of training or test set [65]. They struggle to prevent model poisoning attacks of multiple adversarial clients [26] and are not able to cope with non-iid data or data that is changing over time [66]. Even by combining clustering techniques with robust aggregation methods, they achieve mediocre performance. The gradient value also says nothing about the actions performed by a client. For example, a client could submit random gradients which are sampled from the distribution of the last model updates.

Client-side defense mechanisms have the potential to solve the robustness issue in Federated Learning. The application of TEEs on the client machine enables a trusted validation of the training process without compromising privacy or introducing additional communication cost. The client machine has access to a greater amount of information (like activations, gradients and loss), that can be leveraged to identify model poisoning more effectively. On the client side it is also possible to proof that a client has performed training and does not submit arbitrary gradients that are randomly sampled from the gradient distribution. Training validation on the client also fits seamlessly into the vanilla Federated Learning protocol. Therefore, it can be easily combined with existing techniques for privacy preserving Federated Learning like multi-party computation or differential privacy.

Existing client-side defense mechanisms that leverage TEEs compensate for the slower execution speed of TEEs by applying either early-stopping or sample based approaches. They either train within the TEE and provide great protection with large training overhead or use the full computational resources of the client, but are not able to effectively defend against model poisoning. An approach that could combine the best of both worlds would be able to provide great protection against malicious updates without introducing a large training overhead.

The following chapter further investigates this promising direction and proposes a new client-side defense strategy against model poisoning in Federated Learning.

4. Client-side Training Validation

The examination of defenses against model poisoning attacks in Federated Learning in chapter 3 has shown that client-side defense mechanisms offer promising possibilities to drastically reduce or prevent model poisoning. This chapter further investigates these possibilities and results in a novel concept to defend against model poisoning in Federated Learning.

4.1. Client-side Model Poisoning Detection

4.1.1. Motivation behind Client-side Training Validation with TEE

The approach to defend against model poisoning attacks on the client-side instead of the server-side, yields a number of advantages to detect attacks during the training phase.

By performing validation on the client-side, more information is available to evaluate the correctness of training execution. This includes all information that is part of the training process like training data, labels, model states, activations, gradients and loss. Also, the hyperparameters of the training process like leaning rate, number and size of batches, number of epochs or loss function can be checked. Client-side validation also brings a locality advantage as the rich information available for training validation does not have to be transferred to other instances. This preserves the privacy of the clients.

To make use of the advantages of client-side training validation, the execution of the validation needs to be performed by a trusted unit. A suitable technology for this application is a Trusted Execution Environment (TEE). A TEE is a tamper-resistant processing environment, which guarantees the correct execution of arbitrary code on an untrusted host. With remote attestation it can prove its trustworthiness against third parties. TEEs make it possible to apply client-side defense mechanisms in Federated Learning, without the need to trust the participants. Two different ways exist to use TEEs to defend against model poisoning attacks in Federated Learning.

4.1.2. Training in TEE

The most straight forward way to eliminate model poisoning using client-side TEEs is to perform the model training inside the TEE. This way, model poisoning can be eliminated completely, because of the TEEs guarantees for correct code execution. It also eliminates the need for any verification or model poisoning defense mechanism.

Performing computation in TEEs is significantly slower than untrusted CPUs or even GPUs. In addition, the memory, which is available for secure computations, is typically very small compared to usual system memory. These constraints make TEEs unsuitable for compute and memory heavy machine learning tasks. Previous work with the goal to reduced the amount of computation and memory space, has encountered this issue by combining layer-wise neural network training with early stopping [48]. Nonetheless, this still provides a large overhead and limited flexibility.

4.1.3. Validation in TEE

The second option is to leverage TEEs for client-side training validation. This way, model training can benefit from the full system of the client. Training can take place on CPUs and GPUs and make use of the complete system memory. The TEE can be used to not only proof that the client has actually trained (like TrustFL does), but verify that the client has trained correctly. Hence, this thesis proposes a novel approach that leverages TEEs as validation units that check the correctness of model training. How this can be efficiently done, is further investigated in section 4.3.

Performing model training on normal hardware and validating its correctness in a TEE provides a flexible trade-off between security guarantees and training performance. Although poisoning attacks during training time can not be prevented in the first place, this approach can provide solid guarantees for detecting such attacks.

4.2. Federated Learning with Client-side Validation

Client-side training validation can be implemented into the standard Federated Learning setup. This section describes prerequisites, system architecture and the training process with the new validation scheme.

4.2.1. System Architecture

Like in the normal Federated Learning setup, the system consists of one central server and n clients with their individual data set. The server coordinates the training process and aggregates the model updates, whereas the clients perform model training on their local machines. For this concept to work, each client requires to have a local validator, which is a trusted unit that reliably executes model training validation. The abstract validator can be implemented with a trusted device, server or TEE. This work assumes a TEE as it does not require any other participants, does not introduce network cost and provides a good level of trust.

Like previous works that make use of TEEs [48], this concept builds on the assumption that a secure way to build trust between client TEEs and the server exists. Possible solutions for this could include SIGMA key exchange protocol [39, 71] or attested Transport Layer Security (TLS) [37]. A further assumption is the presence of a key management mechanism that can be used to update or revoke keys, if required [52]. The initial exchange to build trust between TEE and server is referred to as TEE registration.

4.2.2. Training Process

The training process is similar to the standard Federated Learning protocol with a few additional steps. Figure 4.1 gives an overview over the individual steps in the training process.

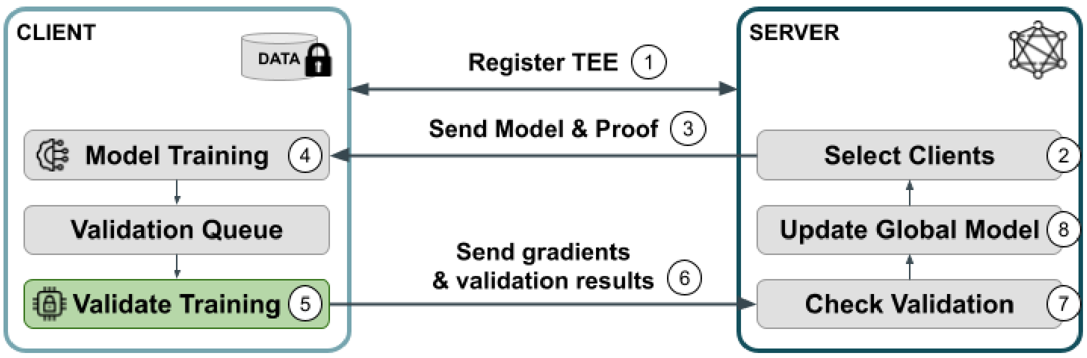


Figure 4.1.: The training process begins with ① the registration of the clients TEE. Like in standard Federated Learning, the server ② randomly selects clients, ③ sends them the current model and the clients ④ train the model on their local data. Before sending the model back, the validator component ⑤ validates, if model training was performed correctly. The validator creates a validation result, that is ⑥ send to the server with the gradients. The server ⑦ checks the validation result and uses the gradients for ⑧ model update, if the validation was successful.

Prior to training, the TEE registration takes place and builds trust between the server and client TEEs. This step only has to be performed once per client.

Equivalent to the vanilla Federated Learning protocol, the server selects a subset of clients for a training round. Then, the server sends the current model to these clients. Alongside the model, the server sends a model proof for the validator that contains a hash of the current model and a unique serial number. The proof is signed by the servers private key and encrypted by the TEEs public key. The serial number starts with 1 after TEE registration and is incremented each time, the client is selected for training. This ensures, that the validator receives the correct initial model for validation and prevents replay attacks.

The selected clients perform the model training with their local data. During training, the client collects the information which is necessary for the validation process and temporarily stores them in a validation queue. The TEE validator reads the training information from the queue and performs the validation. Because the validation of training steps is done separately, it can be performed simultaneously to model training and could potentially, if multiple TEEs are available, be parallelized. By validating all training steps, the validator(s) can proof that the resulting model updates are correct and no model poisoning has been applied.

If the validation of the training process was successful, the validator creates a validation result for the model update. The message includes the hash of the verified model update and the serial number, that was send by the server. The validation result is signed by the TEEs private key and encrypted with the servers public key.

The client sends the signed validation result and the model updates to the server. The server, who has built trust with the TEEs through registration, validates the gradients correctness by verifying the signature of the validation message and the gradients hash value. By checking the serial number, the server knows if it received the correct model update and can prevent replay attacks. If the validation method turns out to be valid, the server uses the gradients for the model update. The new model is used in the next round.

The following section 4.3 goes into details on how the validator works and discusses several design choices.

4.3. Client-side Validator

This section explores how efficient model training validation can be implemented inside a TEE. After identifying the main objectives for efficient training validation in subsection 4.3.1, a number of potential validation methods are explored in subsection 4.3.2.

4.3.1. Objectives & Constraints for Efficient Training Validation

For the design of efficient training validation three main objectives have to be considered:

- **Validation Time:** Training validation is supposed to introduce the least overhead necessary to the overall training time. In the best case it introduces no overhead to the model training. In the worst case it introduces an overhead, so that the overall training time matches secure training in the TEE.
- **Validation Quality:** Training validation aims at achieving the highest possible validation quality. This means that all model poisoning attacks are reliably detected and no false positives occur.
- **Validation Memory Usage:** The execution of training validation is limited by the amount of secure memory, which the TEE provides. Therefore it is desirable to require the least amount of memory for validation.

To make a statement about the correctness of model updates, each phase of neural network training has to be validated. The training process consists of four phases: Forward pass, loss computation, backward pass and weight update. The loss computation and weight update involve relatively simple computation steps. The forward pass and backward pass however require complex matrix-matrix multiplication and have high demand in memory and computational resources. As a result, the most important lever to implement efficient training validation is to find a fast way to validate, recompute or approximate the forward pass and backward pass.

The validation method can work on two different levels of granularity. It can approximate or accelerate the forward pass and backward pass as a whole or verify each matrix-matrix multiplication individually. The following subsection explores several potential validation methods.

4.3.2. Validation Methods

The acceleration of neural networks and validation or approximation of matrix-matrix multiplications is an active field of research. This subsection introduces and discusses several techniques and concludes in a selection of the most promising validation methods. The techniques are divided into methods that focus on the whole forward and backward pass and methods that focus on matrix-matrix multiplication.

Validate Forward-pass & Backward-pass

Fast validation of the whole forward pass and backward pass can be accomplished with model compression, acceleration or sampling techniques.

Knowledge Distillation exploits knowledge transfer learning, which is the training of a compressed neural network with pseudo-data to reproduce the output of the original larger model. While this technique achieves a reduction of model size and improved inference speed, the creation of the compressed model requires a larger training overhead [11]. This technique would be applicable for forward pass validation, but not for backward pass validation. The reproduction capability of the compressed model is strongly dependant on the convergence of the compressed model. Therefore it is hard to derive a correctness guarantee for the validation result.

Low-rank Factorization has similarities to knowledge distillation. It builds on the intuition that a structure spacity exists in a convolutional or fully connected layer. To speed up inference, the original layer is replaced by a low-rank filter. The low-rank filter is tuned layer wise [11]. Like knowledge distillation, low-rank factorization involves computational expensive training to receive a compressed model. Also in terms of reproduction capability it shares the disadvantages of knowledge distillation as it is only applicable to forward pass and the derivation of guarantees is hard.

Parameter Pruning & Quantization are effective in reducing the complexity or neural networks and thus improve computation speed and memory requirements.

Quantization & Binarization compresses the original network by reducing the number of digits that are used to represent the weights. This can significantly reduce the memory usage and demand for computational resources. The extreme case of reducing the weights to 1-bit representations is called binarization. However binarization lowers the accuracy for the network significantly [11]. In general, decreasing the number of digits increases the rounding error for computations. The attempt to train neural networks with unified 8-bit (INT8) weights can make the training unstable or even crash. Training with 16-bit floating-point (FP16) is supposed to be 2x faster than default FP32 [73]. FP16 model training support is available in common machine learning frameworks like TensorFlow [21] and PyTorch [15], however only for the use on GPUs [20]. Therefore quantized operations are not easily applied for validation in TEEs.

Network Pruning reduces the complexity of a neural network by removing connections in the network which do not contribute to the model performance. A reduced number of parameters results in fewer operations and therefore a faster network. The compact network can also be trained however converges slower than the original network [11]. This makes it again hard to provide correctness guarantees for the validation method. Moreover, the pruning criteria often requires manual tuning, which is not practical for validation.

Batch Sampling can also be applied to reduce the number of operations for model training validation. By randomly selecting, if a batch is validated or not, the decision probability provides a trade-off between security guarantee and validation performance. The gradient computation for selected batches is recomputed or validated with another validation method, while the other batches are skipped.

Validate Matrix-Matrix Multiplication

Validation methods can also be applied at the level of matrix-matrix multiplication, because forward-pass and backward-pass are both sequences of matrix-matrix multiplications. The goal of the validation method is to verify, given matrix A, B and C, if $A \times B = C$. Methods applicable to this problem are Matrix Product Verification, Matrix Product Approximation and Sub-matrix Sampling.

Matrix Product Verification is the algorithmic check if $A \times B = C$. The fastest known algorithm for this is Freivalds' Algorithm [25]. By taking advantage of randomness, Freivalds' Algorithm can check the correctness of matrix-matrix multiplication in $O(n^2)$. The probability of detecting an error is $(\frac{1}{2})^k$, where k is the number of iterations. Freivalds' Algorithm uses a random vector $r \in \{0, 1\}^n$ and calculates if $ABr = Cr$. If the results are equal the output is `True` and if the results do not match, the output is `False`. Freivalds' Algorithm has already been successfully applied in the area of secure ML outsourcing [63]. A recent variation of Freivalds' Algorithm is GVFA. Instead of a random vector containing zeros and ones, GVFA uses a Gaussian random vector, where each element is an independent distributed normal random variable with finite mean and variance. The advantage is, that one single iteration is enough to verify the matrix-matrix multiplication [35].

Sub-matrix Sampling / Sub-Matrix Multiplication (SubMul) also makes use of randomness to efficiently verify the correctness of a matrix product. For that, $\hat{A}$ is sampled as a subset of dimension 0 of A. The proportion of sampled rows and total rows $\frac{\hat{A}.shape[0]}{A.shape[0]}$ is the correctness guarantee of the validation. The validation works by computing $\hat{A} \times B = \hat{C}$ and checking if $\hat{C}$ is a sub-matrix of C. This reduces the matrix product complexity depending on the sample probability.

Matrix Product Approximation is a class of techniques that compute an approximation of matrix products in less time than an actual matrix product would need. Given two matrices A and B, the matrix product approximation returns matrix C' which approximates $C = AB$. The objective is to make the approximation error $C - C'$ as small as possible [43]. After the first approximation method for matrix product [12], several efficient methods like MADDNESS [5], Bolt [6], PQ [34] and OPQ [28] have been proposed and promise good approximation capability. For none of the approximation methods, a functioning implementation is publicly available.

Of the methods presented, the most promising are the ones that validate matrix-matrix multiplications. Freivalds' Algorithm and GVFA convince by their simple and straight forward implementation. Also Sub-matrix Sampling can be easily implemented and applied. The use of Matrix Product Approximation could also have great potential, but there is a lack of publicly available and functional software implementations. Therefore this is declared as future work.

The following section derives security guarantees for the application of matrix product verification and takes a look at the computation cost for Freivalds' Algorithm, GVFA and Sub-matrix Sampling.

4.4. Guarantees & Computational Complexity

This section reasons about the guarantees and computational costs of Feivalds' Algorithm, GVFA and Sub-matrix Sampling.

4.4.1. Detection Guarantees

This subsection derives guarantees for model training validation through matrix product verification. Furthermore, the resulting guarantees of the three validation methods are presented. The guarantees refer to an MLP with L layers.

The derivations are stated in mathematical terms and do not account for numerical error and do not consider the relative and absolute tolerance of using the `torch.allclose(t1,t2)` method to check the equality of tensors `t1` and `t2`. However, it can be said that the derived guarantees hold for errors that are above the tolerance introduced by `torch.allclose()`.

Forward-pass Guarantees

The forward-pass is a sequence of matrix-matrix multiplications. Figure 4.2 visualizes the forward pass and the containing matrix-matrix multiplications.

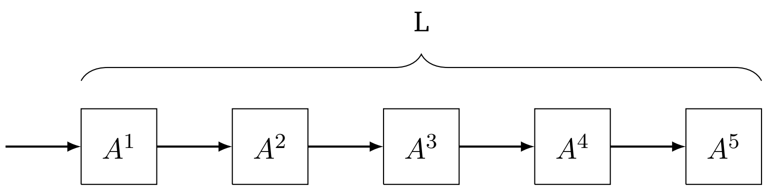


Figure 4.2.: Visualization of the forward-pass of a MLP with $L = 5$ layers. Each arrow $\rightarrow$ represents a matrix-matrix multiplication. A^i is the activation matrix at the output of layer i.

Let p be the probability of not detecting an error when validating a tampered matrix-matrix multiplication. If the validation of a matrix-matrix multiplication is positive, the probability for it to be actually correct is $1 - p$.

The validation of each layer $i > 1$ is dependent on the output of the previous layer A^{i-1}. Assuming that each matrix-matrix multiplication is validated independently, the probability of the forward pass being correct is

$$P(A^1)P(A^2|A^1)\cdots P(A^L|A^{L-1},\cdots,A^1) = P(A^1,\cdots,A^L) = (1-p)^L \tag{4.1}$$

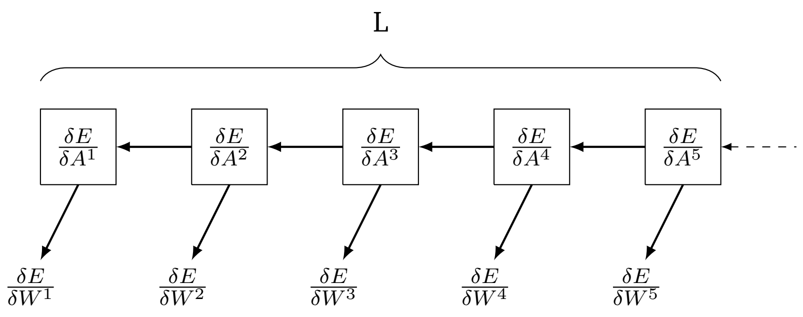

L
$\frac{\delta E}{\delta A^1}$
$\frac{\delta E}{\delta A^2}$
$\frac{\delta E}{\delta A^3}$
$\frac{\delta E}{\delta A^4}$
$\frac{\delta E}{\delta A^5}$
$\frac{\delta E}{\delta W^1}$
$\frac{\delta E}{\delta W^2}$
$\frac{\delta E}{\delta W^3}$
$\frac{\delta E}{\delta W^4}$
$\frac{\delta E}{\delta W^5}$

Figure 4.3.: Visualization of the backward-pass of a MLP with $L = 5$ layers. Each solid arrow $\rightarrow$ represents a matrix-matrix multiplication. The dashed arrow $\dashrightarrow$ shows the dependency on the forward pass. $\frac{\delta E}{\delta A^i}$ is the gradient of the loss function with respect to the activation matrix at the output of layer i. Similarly $\frac{\delta E}{\delta W^i}$ is the gradient of the loss function with respect to the weight matrix at the output of layer i.

Backward-pass Guarantees

The backward-pass also consists of a sequence of matrix-matrix multiplications, with the difference of additionally computing the gradient with respect to the weight for each layer. Figure 4.3 visualizes the backward-pass and the containing matrix-matrix multiplications.

Let g_A^i be the gradient of the error function with respect to the activation matrix $\frac{\delta E}{\delta A^i}$ at layer i. Further, let g_W^i be the gradient of the error function with respect to the weight matrix $\frac{\delta E}{\delta W^i}$ at layer i. The correctness guarantee of the gradient at the output of the last layer equals the guarantee of the forward-pass $P(g_A^L) = P(A^1, \cdots, A^L) = (1-p)^L$. Assuming that matrix products are validated independently, the probability of gradient g_W^i being correct is

$$P(g_W^L | g_A^L) = (1-p) \tag{4.2}$$

$$P(g_W^{L-1} | g_A^{L-1}, g_A^L) P(g_A^{L-1} | g_A^L) = P(g_W^{L-1}, g_A^{L-1} | g_A^L) = (1-p)^2 \tag{4.3}$$

$$\vdots$$

$$\begin{aligned} P(g_W^1 | g_A^1, \cdots, g_A^L) P(g_A^1 | g_A^2, \cdots, g_A^L) \cdots P(g_A^{L-1} | g_A^L) = \\ P(g_W^1, g_A^1, \cdots, g_A^{L-1} | g_A^L) = (1-p)^L \end{aligned} \tag{4.4}$$

For the case that the whole backward-pass is correct, all gradients have to be correct. Hence, the probability of correctness of the backward-pass is the joint probability of the individual correctness of the gradients with respect to the weights g_W^i.

$$\begin{aligned} P(g_W^1, g_A^1, \cdots, g_A^{L-1} | g_A^L) P(g_W^2, g_A^2, \cdots, g_A^{L-1} | g_A^L) \cdots P(g_W^{L-1}, g_A^{L-1} | g_A^L) P(g_W^L | g_A^L) = \\ P(g_W^1 | g_A^1, \cdots, g_A^L) P(g_W^2 | g_A^2, \cdots, g_A^L) \cdots P(g_A^L | g_A^L) \cdot \\ P(g_A^1 | g_A^2, \cdots, g_A^L) P(g_A^2 | g_A^3, \cdots, g_A^L) P(g_A^{L-1} | g_A^L) = (1-p)^{2L-1} \end{aligned} \tag{4.5}$$

Training-step Guarantee

Combining the probabilities for the correctness of forward-pass $P(FP)$ and backward-pass $P(BP|FP)$ results in the correctness probability of the whole training step. Again assuming that the validation of forward-pass and backward-pass is performed independently.

$$P(\text{Training-step} = correct) = P(BP|FP)P(FP) = (1-p)^{3L-1} \tag{4.6}$$

The correctness guarantee for the validation of one training step is influenced by the number of layers L and the probability of not detecting an error in a matrix product check p. Given a desired guarantee for the correctness of the training step q, the upper bound for p is justified in Theorem 1.

Theorem 1. *Let L be the number of layers of an MLP. Further, let q be the desired correctness guarantee for the validation of a training step with the MLP. Then, the probability p of not detecting an error during matrix product validation needs to be*

$$1 - \sqrt[3L-1]{q} \geq p$$

Proof. To meet the desired guarantee q, the probability of the training step verification being correct needs to be greater or equal to q. From that, the maximum value for p can be derived

$$\begin{aligned}
(1-p)^{3L-1} &\geq q \\
\ln(1-p)(3L-1) &\geq \ln(q) \\
\ln(1-p) &\geq \frac{\ln(q)}{3L-1} \\
e^{\ln(1-p)} &\geq e^{\frac{\ln(q)}{3L-1}} \\
1-p &\geq \sqrt[3L-1]{q} \\
1-\sqrt[3L-1]{q} &\geq p
\end{aligned} \tag{4.7}$$

A lower bound for this can also be derived, which is shown in Appendix A. □

Validation Method Guarantees

The guarantees derived for a whole training step of an MLP can be applied on the individual validation methods.

Freivalds' Algorithm

The probability p of not detecting an error during matrix product verification with Freivalds' Algorithm is $p = (\frac{1}{2})^k$. Where k is the number of times Freivalds' Alrogithm is applied. By using Theorem 1, it is possible to derive the number of times Freivalds' Algorithm needs to be applied to meet the desired correctness guarantee q.

$$\begin{aligned} 1 - \sqrt[3L-1]{q} &\geq \frac{1}{2}^k \\ \log_2(1 - \sqrt[3L-1]{q}) &\geq \log_2\left(\frac{1}{2}\right)k \\ \log_{\frac{1}{2}}(1 - \sqrt[3L-1]{q}) &\leq k \end{aligned} \tag{4.8}$$

It is an advantage that the number of necessary iteration for achieving a certain guarantee grows logarithmic with the number of layers. The number of iterations is also closely related to the validation time of Freivalds' Algorithm. Figure 4.4 show the development of Freivald iterations to achieve 99% correctness guarantee per number of layers.

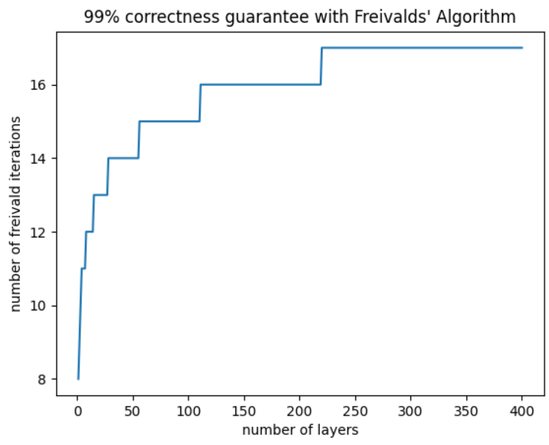


Figure 4.4.: Number of times Freivalds' Algorithm needs to be applied, to achieve a 99% correctness guarantee for validating a specific number of layers.

Gaussian Variant of Freivalds' Algorithm (GVFA)

For GVFA, the validation error is small and depends on the floating point digits during computation. In general, the larger the error in the matrix computation, the more likely it is detected with GVFA. Assuming no computation error p is very small $1 - \sqrt[3L-1]{q} \gg p$. Equivalent to Freivalds' Algorithm, the guarantee of the method can be increased by applying GVFA multiple times. However, a concrete guarantee like for Freivalds' Algorithm can not be derived.

Sub-matrix Sampling

Assuming s to be the fraction of rows sampled from the matrix in Sub-matrix Sampling. Then, the probability of not detecting an error is $p = 1 - s$. To satisfy a desired guarantee q in a scenario with L layers, the sampling fraction needs to be

$$\begin{aligned} 1 - \sqrt[3L-1]{q} &\geq p \\ 1 - \sqrt[3L-1]{q} &\geq 1 - s \\ \sqrt[3L-1]{q} &\leq s \end{aligned} \tag{4.9}$$

For a MLP with $L = 3$ and a desired guarantee $q = 0.5$, the sample fraction s needs to be $s = \sqrt[3\cdot 3-1]{0.5} \approx 0.917$. However reducing computation by only 8.3% for a drop in guarantee of 50% is not reasonable.

It is better to sample once from the inputs and then perform normal forward-pass and backward-pass with the reduced batch. Then $P(Training = correct) = s$. This further saves some sampling and matrix transformation operations in the implementation. This however, was not implemented in this work.

4.4.2. Computational Complexity

As previously stated in the background section 2.1.4, the basic algorithm to perform matrix-matrix multiplication of two matrices $A, B \in \mathbb{R}^{n\times n}$ has a complexity of $\mathcal{O}(n^3)$ and the fastest known algorithm has a complexity of $\mathcal{O}(n^{2.3728639})$. Generalizing to matrices with unequal size $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times k}$, a complexity of $\mathcal{O}(mnk)$ can be assumed. An efficient validation method for matrix-matrix multiplications is supposed to have a lower computational complexity.

Sub-matrix sampling reduces the first dimension of one matrix by a factor of s. As a result, the computational complexity of Sub-matrix Sampling is $\mathcal{O}(\frac{mnk}{s})$, which also reduces to $\mathcal{O}(mnk)$. Hence, it does not have a benefit in terms of computational complexity over MatMul.

Freivalds' Algorithm and GVFA reduce the computation to three matrix-vector multiplications. The complexity for the matrix-vector product of matrix $A \in \mathbb{R}^{n\times m}$ and vector $r \in \mathbb{R}^m$ is $\mathcal{O}(mn)$. Taking all three matrix-vector multiplications together ($Br = B \times r$, $ABr = A \times Br$ and $Cr = C \times r$), the computational complexity for one round of Freivalds' Algorithm or GVFA is $\mathcal{O}(mn + nk + mk)$.

5. Implementation

After the introduction of the concept for client-side training validation in Federated Learning in the previous chapter 4, this chapter provides details about the implementation of the concept. The complete source code for this thesis is further available in a Github repository at `https://github.com/DataManagementLab/thesis-fl_client-side_validation`.

5.1. Technological Foundation

The code for this work is implemented in Python. Python is among the most popular programming languages nowadays [51] and is known for its simple and readable syntax. It is an interpreted language that is platform independent. Python is widely used in the application of ML as the majority of ML frameworks provide a python Application Programming Interface (API). Notably there are the open-source frameworks TensorFlow [30], developed by Google, PyTorch [58], developed by Facebook, and Scikit-learn [22]. This work makes use of PyTorch which is often chosen in research context.

This section describes the implementation of the training process for the concept of client-side training validation. To reduce complexity, the implementation focuses only on the federated learning client. This means that training in rounds with a central server and communication between server and client is neglected. These aspects of Federated Learning do not have any influence on the efficacy of the approach and do not benefit the evaluation.

The validator is not implemented within an actual TEE but in normal python code. It also builds on the standard API of PyTorch. This abstraction is valid as the computation performance of TEEs is proportional to the ones of CPUs. The use of the PyTorch framework is also possible in a real TEE with the help of Graphene [14].

Based on the fact that the validations of training steps are independent of each other, a potential for parallelization of the training validation among multiple validators exists. Eventhough the implementation is able to create multiple concurrent validation processes in the asynchronous validation setting, this is not further investigated in this work.

5.2. Training Process

Neural network training on the client-side has similarities with the vanilla PyTorch model training but requires some additional steps. After training preparation the local training itself is executed and requires post-processing. Finally the information required for validation is transferred to the validator.

5.2.1. Training Preparation

The validation of the training process requires information about activations and gradients that are usually not stored when training a neural network. To retrieve this information during model training in PyTorch, it is required to register hooks that receive and store the information.

To save the activations a `forward_hook` is registered at each layer, which is called every time after the `forward()` method of the layer has computed an output [17]. The `forward_hook` contains logic to store the computed output of the layer in a data structure.

In a similar way computed gradients with respect to the layers inputs, weights and bias have to be saved. This is possible by registering a `full_backward_hook` at each layer [18] which is executed every time the gradients have been computed in a backward pass. The gradients are also saved in a suitable data structure.

Another important preprocessing step is saving the current status of model parameters and optimizer (if stateful) before starting the actual training. This way the validation of the gradients starting from the initial model is possible.

After the training preparation steps, the actual model training can start.

5.2.2. Training Execution

The execution of the neural network training is implemented in a training function (`training_fn`). The training function receives the current model, optimizer, loss function, data and the corresponding label of one batch. In a normal setup, the training function simply trains the PyTorch model in the usual way. This corresponds to the honest clients training. However the training function can be exchanged arbitrarily and can contain model poisoning attacks. Several implementations of model poisoning attacks are discussed in section 5.5.

Normal model training in PyTorch involves four steps. The following code snippet shows, how they are applied.

```
1 output = model(batch)
2 loss = loss_fn(output, target)
3 loss.backward()
4 optimizer.step()
```

Line 1 is the forward pass, where the model makes a prediction depending on the input data. The model processes the batch of data samples and generates an output with a prediction for each data sample. In line 2 the output (predictions) is compared to the real target (labels) of the data samples using a loss function. In line 3 the loss function generates a gradient which is propagated backwards through the models. The result of the backward pass is a gradient at each weight, which represents the necessary changes for this weight. Finally in line 4 the optimizer performs the weight update.

Depending on the available hardware, the training can be executed on the CPU or GPU. In comparison the GPU achieves a significantly higher training speed than the CPU. Because of its highly parallel architecture, the GPU is well suited for computing matrix operations.

After the model has been trained on one batch, the post-processing of the training takes place.

5.2.3. Training Post-processing

The task of the post-processing step is to collect all necessary data for the validation of a training step. This includes the state of the model after the training step, data and corresponding labels which were used for training, the state of the optimizer and the computed activations, gradients and loss. For improved data handling, the information is collected in a `ValidationSet`. Within a `ValidationSet` the validator has all required information to validate one training step.

5.2.4. Transfer Validation Information

Switching between application and TEE introduces heavy performance penalty [64]. Therefore it is not advisable to pass each `ValidationSet` individually to the TEE. As a result, `ValidationSets` are buffered and passed to the validator in one chunk. The size of the buffer is a hyperparameter which can be adapted as needed.

If the buffer has reached its maximum capacity or if the last batch of an epoch was added to the buffer, it is marked as complete. Then the buffer is passed over to the validator. The validation can take place synchronous, alternating between model training and validation, or asynchronous to the training in a parallel process. The following section describes the inner workings of the validation process and how the synchronous and asynchronous validation settings work.

5.3. Validation Process

The buffer of `ValidationSets` is the only information which the validation process receives from the training process. This section describes how the content of the buffer is validated (5.3.1) and how the synchronous (5.3.2) and asynchronous (5.3.3) validation is implemented.

5.3.1. Buffer Validation

A buffer is a collection of the information about subsequent model training steps. The validation process iterates over the individual `ValidationSets` in the buffer and validates each of them separately.

The validation of a training step starts with the initial state of the model at the beginning of the step and the batch training data. For each layer, the forward pass is validated, which is a matrix-matrix multiplication with the subsequent application of an activation function. The validation is performed with the help of a selected validation method from the concept section (4.3.2). Their implementation is further described in section 5.4.

If the forward pass was validated successfully, the validation process has a proof for the model output. Together with the labels associated with the batch data, the loss can be computed. This operation is in comparison to forward pass and backward pass not costly and is simply recomputed with the loss function. The gradient of the loss function with respect to the output of the neural network provides the initial gradient that is used to validate the backward pass.

To validate the backward pass, it is required to validate two matrix-matrix multiplications at each layer. The first matrix-matrix multiplication is the computation of the gradients with respect to the weights of the layer and the second one is the computation of gradients with respect to the output of the previous layer. Both matrix-matrix multiplications are again validated with the selected validation method. Only for the first layer of the network, there is only one matrix product to validate. The gradient at the bias is computed by summing the dimension 0 of the gradient with respect to the pre-activation of the layer.

As previous procedures have successfully validated the correctness of the weight gradients of the model, the last step is validating the model update. This step involves matrix-matrix additions, which are again less computational expensive than matrix-matrix multiplications. Hence, this step is also recomputed like in the original training process. If the newly computed weights match the state of the model at the end of the training step, the validation of the training step was successful.

The validation process can be performed alternating with the training process, or run in parallel. Both approaches and how the communication between training and validation process works for both approaches, is described in the following subsections.

5.3.2. Synchronous Validation

In the case of synchronous validation, the client is alternating between the training and validation phase. If the buffer reaches his maximum capacity, the client stops training and enters the validation. The buffer with its `ValidationSets` is directly transferred to the validator. Validation takes place as described in the previous subsection until all `ValidationSets` in the buffer are validated. Finally the program returns to the training function and continues the model training, until the buffer has filled up again.

5.3.3. Asynchronous Validation

Interaction between training and validation is more complex when it comes to asynchronous validation. Unlike the synchronous validation, training and validation have to be executed in separate processes to achieve real concurrent execution in the asynchronous setting. The model training, which might make use of GPUs remains in the main python process. For the validation task, a python sub-process is spawned at the beginning of the training.

For the transfer of the validation buffers, both processes are connected by a multiprocessing safe queue. If a buffer is filled with `ValidationSets`, the training process writes it to the queue and continues training. The validation process constantly listens at the queue. If a new buffer is added to the queue, the validation process removes it from the queue and validates it.

If the time for validating a buffer is larger than the time for producing a buffer, the number of items in the queue might grow over time. In an extreme case, this can lead to the size of the queue exceeding the clients memory. As an alternative to writing the buffer object to the queue, the buffer can be saved on the file system with Python object serialization and the path to the buffers location is written to the queue. This greatly reduces memory overhead. As TEEs are able to read data from untrusted locations, this approach is also applicable with a real TEE in place.

5.4. Validation Methods

The abstraction of the client-side training validator provides the possibility to make use of different validation methods in an interchangeable way. The concept section (4.3.2) has discussed several approaches and selected few of them. This section shows how the validation methods have been implemented using PyTorch.

5.4.1. Retrain (Baseline)

As a baseline, the Retrain validation method refers to the secure retraining in the TEE. It retrains the initial model with the data and captures activations and gradients. The information is compared against the original observations from the `ValidationSet`. It is expected that other validation methods improve upon the Retrain baseline in terms of memory usage and validation time.

5.4.2. MatMul (Baseline)

Another baseline which has strong similarities with the Retrain validation method is the MatMul validation method. MatMul also recomputes the matrix-matrix multiplications of the forward pass and backward pass. In contrast to Retrain, MatMul computes the matrix product of each layer individually and does not use the model training functionality of PyTorch. Therefore it is a drop-in replacement for the remaining validation methods and shares, except for its execution time, the same validation overhead. The following code shows the implementation of the MatMul validation method:

```
def matmul(A, B, C, bias):
    C_ = torch.matmul(A,B) + bias
    return tensors_close(C, C_)
```

5.4.3. Sub-matrix Sampling

Sub-matrix sampling is a validation method that improves upon the MatMul baseline. Instead of fully recomputing the matrix-matrix multiplication, Sub-Matrix Multiplication (SubMul) randomly samples rows from one matrix and then computes the matrix-matrix multiplication with the sub-matrix. The guarantee of the validation correlates positive and the execution speed correlates negative with the sampling ratio s. The following code shows how sub-matrix sampling is implemented:

```
def submul(A, B, C, bias, s):
    permut = torch.randperm(A.shape[0])[0:int(A.shape[0]*s)]
    Ap = A[permut]
    Cp = C[permut]
    return tensors_close(torch.matmul(Ap, B) + bias, Cp)
```

5.4.4. Freivalds’ Algorithm

Freivalds’ Algorithm makes use of randomness to verify the correctness of a matrix-matrix multiplication. The guarantee of the validation can be improved by executing Freivalds’ Algorithm several times. The random 0-1-vector is created by sampling a random vector in $[0, 1)$ and round it to integer values. Executing freivalds multiple times can be speeded up by using a 0-1-matrix instead. This way, multiple rounds of Freivalds’ Algorithm can be computed at once. Following code shows the implementation of Freivalds’ Algorithm:

```
def freivald(A, B, C, bias, n_check):
    R = torch.round(torch.rand(B.shape[1], n_check))
    ABr = torch.matmul(A, torch.matmul(B, R))
    Cr = torch.matmul(torch.sub(C, bias), R)
    return tensors_close(ABr, Cr)
```

5.4.5. Gaussian Variant of Freivalds’ Algorithm (GVFA)

The implementation of the GVFA has many similarities with Freivalds’ algorithm. However GVFA does not require repeated execution, because one execution can already validate the correctness of the matrix-matrix multiplication. The random vector is filled with values that are sampled from independent random variables that have a normal distribution with mean 0 and standard deviation of 1. The implementation of GVFA is as follows:

```
def gvfa(A, B, C, bias, n_check):
    R = torch.normal(
        mean=torch.zeros(B.shape[1], n_check),
        std=torch.ones(B.shape[1], n_check))
    ABr = torch.matmul(A,torch.matmul(B, R))
    Cr = torch.matmul(torch.sub(C, bias), r)
    return tensors_close(ABr, Cr)
```

5.5. Poisoning Attacks

The aim of client-side training validation is to prevent model poisoning attacks in the Federated Learning setting. To test the effectivity of the contributed approach, two types of model poisoning attacks have been implemented. Both model poisoning attacks replace the `training_fn` in the training execution. The following subsections describe the implementation for untargeted model poisoning 5.5.1 and targeted model poisoning 5.5.2.

5.5.1. Untargeted Model Poisoning

Untargeted model poisoning is implemented by inserting random noise during the training process. Noise can be added to activations, gradients or model updates. The goal of this attack is to prevent the model from converging to an optimal state. Alternatively adding noise can also be viewed as a generalization of any form of model poisoning attack. The reason is that any model poisoning attack would add arbitrary changes some were throughout the model training process. For poisoning the model gradients, the client randomly adds noise to the gradients of a layer before the optimizer updates the weights. The following code shows how this is done:

```
output = model(batch)
loss = loss_fn(output, target)
loss.backward()
for l, weight in model.named_parameters():
    if torch.rand(1).item() <= frequency:
        weight.grad += gradient_noise(weight.shape, scale=scale)
optimizer.step()
```

The noise is generated with the`gradient_noise` function that returns a uniform noise in range $]0, 1[$ which is scaled by the `scale` parameter. The `frequency` variable controls how often noise is added to the layers gradients.

For the simulation of attacks that do not add changes to all elements of the gradient vector, an alternative implementation looks like this:

```
output = model(batch)
loss = loss_fn(output, target)
loss.backward()
for l, weight in model.named_parameters():
    if torch.rand(1).item() <= frequency:
        noise = torch.zeros(weight.shape)
        for n in range(corrupt_neurons):
            noise[rand_item(noise.shape)] = neuron_noise
        weight.grad += noise
optimizer.step()
```

This implementation adds a fixed noise of `neuron_noise` to `corrupt_neurons` number of neurons. The neurons are randomly selected with the `rand_item()` function.

Creating noise for the activations during forward pass or the gradients during backward pass can be done in the same way. The insertion of noise is then applied by forward hooks and full backward hooks. However the implementation does not inclue this.

5.5.2. Targeted Model Poisoning

For the targeted model poisoning attack, the model is trained on normal and malicious data samples in an alternating fashion. To increase the effect of the malicious data samples on the overall model, weight updates for malicious data samples are boosted by increasing the learning rate of the optimizer for these samples. The training data consists of batches that either contain only normal or malicious data samples. The batches are in random order.

```
if malicious_data: set_malicious_lr(optimizer)
output = model(batch)
loss = loss_fn(output, target)
loss.backward()
optimizer.step()
if malicious_data: set_original_lr(optimizer)
```

Before training the model with a batch, the malicious client checks, if this batch contains malicious data samples or not. If so, the learning rate of the optimizer is increased. Further training with the batch is performed according to protocol. After the model weights are updated, the learning rate needs to be reset to the original value.

Creation of Dirty-label Data Sets

To create a dirty-label data set for targeted misclassification in PyTorch, original data samples have to be extracted from the training set of the original data set. The data samples are stored with the corresponding dirty-labels in any format of choice.

To use the malicious data samples for model training in PyTorch, it is necessary to implement a custom PyTorch data set, which inherits from the `torch.utils.data.Dataset` class [16]. After implementing the required methods `__init__()`, `__len__()` and `__getitem__()`, the malicious data set can be used for model training in PyTorch.

6. Evaluation

Previously this work has introduced the concept of client-side training validation in chapter 4 and details about the implementation of this concept in chapter 5. To gain a better understanding about the benefits and disadvantages of the proposed concept, this chapter evaluates the most important aspects of the concept by conducting experiments. Section 6.1 provides details about the experimental setup, evaluation criteria and methodology for measurements. The experimental results are presented in section 6.2 and discussed in section 6.3.

6.1. Experimental Setup

This section describes the experimental setup for the evaluation. Information about the evaluation criteria, execution of the experiments and the system hardware is provided here.

6.1.1. Selection of Evaluation Criteria

In the analysis of current model poisoning defense mechanisms, six goals for private and robust Federated Learning have been mentioned. These are (1) fast algorithmic convergence, (2) good generalisation performance, (3) communication efficiency, (4) fault tolerance, (5) privacy preservation, (6) robustness to targeted, untargeted poisoning attacks, and free-riders.

The concept that is proposed by this thesis is designed in a way that it provides security against model poisoning attacks without interfering with the standard Federated Learning system. Therefore it has no effect on the (1) algorithmic convergence and (2) generalization performance of the model. The (3) communication efficiency and (4) fault tolerance of the Federated Learning system remain unchanged. Even though, the concept does not address (5) privacy preservation per se, it can be combined with most existing privacy preservation techniques like the ones building on Differential Privacy or Multi Party Computation.

The main goal addressed by the proposed concept is the robustness to targeted & untargeted model poisoning attacks and free-riders. Therefore, one evaluation criterion is the quality of detecting model poisoning attacks.

An important component of the concept is the client-side TEE that serves as a trusted validation component. The background section has stated memory and execution time to be the two limiting factors for the application of TEEs. Hence, it is desirable to have a minimal memory footprint and efficient execution if the validation is performed within the TEE. Hence, validation time and memory consumption are two further evaluation criteria.

Finally, the application of the proposed concept should introduce the least amount of overhead to the total training time of the client. Therefore, the fourth and last evaluation criterion is the total time required for executing model training and training validation.

6.1.2. Measurement of Evaluation Criteria

This subsection describes, how the four evaluation criteria Quality of Detection, Memory Consumption, Validation Time and Total Training Time are measured during the experiments.

Quality of Detection

Every model poisoning attack injects some changes to either activations or gradients during model training. Therefore, an arbitrary model poisoning attack can be generalized as the insertion of noise to activations or gradients. Hence, the targeted model poisoning attack is not used in the evaluation. Further, when applying untargeted model poisoning attacks in the experiments, noise is only injected at the gradients. Since the attack detection is equivalent for activations and gradients, this does not have an effect on the significance of the experiment.

The gradients that become poisoned are randomly selected with a frequency of 0.1. For a gradient that is supposed to be poisoned, a fixed error ϵ (i.e., noise) is added to n randomly selected elements of the matrix. Both parameters have an influence on the degree of which validation methods are able to detect the errors.

If an error is added to a gradient, this information is logged to the experiments log directory. During validation, if an incorrect matrix-matrix multiplication is determined, this event is logged as a detected attack. Afterwards, both information can be matched against each other, which reveals the True Positives, False Positives and False Negatives. Based on this information, several metrics like f1-score, precision, recall and accuracy can be computed [31].

Memory Consumption

The memory consumption is measured in the asynchronous setting, where the validation process is located in a separate process. At the beginning of the process, a thread is spawned within the validation process, that checks the current memory footprint of the process with an interval of 1 millisecond.

This is done with the Python library `psutil`. Given the process id `pid`, the library can retrieve memory information about the process with `psutil.Process(pid).memory_info()`. Among others, the library provides the Resident Set Size (RSS) metric, which represents the non-swapped physical memory a process has used [56].

Memory measurements begin 5 seconds after the validation process has started to be able to measure the base memory usage of the python process. Each time, the threat measures a new data point, it is compared against two values. `base` is the lowest value of memory usage, the monitor threat has seen so far. `peak` is the maximum value of memory usage, that was measured. If the new data point exceeds the `peak` or falls below the `base`, it replaces them respectively. The difference between `base` and `peak` represents the minimum amount of memory that is required for validation.

The measurement continues during all epochs of the experiment. When the validation process has processed all validation buffers from the queue and has received the shut-down notice, the final `base` and `peak` is saved to the experiments log directory and the process shuts down.

Validation Time

The validation time is tracked using a custom `TimeTracker` class that builds upon the python `time` library [24]. The `time.time()` method reports the current Unix time (time in seconds since January 1, 1970, 00:00:00 (UTC)). `TimeTracker` uses this and provides an interface to start and stop time tracking for a specified `id`, like a stopwatch.

The validation of a buffer takes place in the `validate_buffer()` function. Right at the beginning of the function, tracking of the validation time is started. And right before the `validate_buffer()` ends, time tracking is stopped. The validation duration for all buffers is stored in the `TimeTracker` object and saved to the log directory at the end of each epoch. For evaluation, the measurements for buffer validation are summed up per epoch and the total validation time for all epochs in multiple experiments are averaged.

Total Training Time

Similar to the validation time, the total training time is measured with the `TimeTracker` class. The time frame is started once, before model training has started, and is stopped right after model training has ended. In the synchronous setting, this includes the validation time, because the main process alternates between training and validation. In the asynchronous setting, the validation process measures its own run-time. For that, a time frame is stared as soon as the validation process was started and is stopped right before the process ends. The total training time includes both, the time frame of the training process and the time frame of the validation process.

6.1.3. Experiment Execution

Experiments are executed on a desktop computer with an AMD Ryzen 5 2600 (6 cores, 3.40GHz, 12 threads) CPU and a NVIDIA GeForce GTX 1660 (6GB GPU RAM, DDR5) GPU. The system has 16 GB of Random Access Memory (RAM) and runs Ubuntu 20.04.3 LTS. The code is executed with Python 3.8.10 and uses PyTorch 1.8.1 with CUDA 10.1.

For neural network training, the MNIST database is used, which contains a training set with 60.000 data samples [40]. MNIST is a collection of images of handwritten digits, which the neural network learns to classify.

Experiment Parameters

The following table 6.1 summarizes all parameters of the experiments and what they stand for.

Parameter	**Description**
validation method	The method used for validation (Freivalds' Algorithm, GVFA, SubMul, MatMul, Retrain)
model size	The number of hidden layers of the MLP
layer size	The number of neurons of the hidden layers
execution mode	Mode for the execution of the training validation: training without validation (train-only), synchronous (sync) or asynchronous (async)
device	The device on which the model training is performed (can be CPU or GPU)
batch size	The batch size used for model training
buffer size	The number of ValidationSets that are collected in a buffer, before it is send to the validator
guarantee	The desired correctness guarantee for validation (applicable for Freivalds' Algorithm and SubMul)
n_check	The number of checks performed to validate a matrix-matrix multiplication (applicable for Freivalds' Algorithm and GVFA)
attack probability	The probability to which the result of a matrix-matrix multiplication becomes poisoned
attack noise	The attack noise ϵ, which is used to poison individual neurons
n_neurons	The number of neurons in a matrix that are tampered with, if a matrix product is poisoned (if poisoning is applied)
absolute tolerance	The absolute tolerance for checking matrix-matrix equality in `tensors_close`
relative tolerance	The relative tolerance for checking matrix-matrix equality in `tensors_close`
repetitions	How often an experiment is repeated
epochs	The number of epochs, a model is trained for within an experiment

Table 6.1.: Experiment parameters and their meaning.

In the following, the values for all common experiment parameters (e.g., layer size) and specific parameter values per experiment will be outlined.

6.2. Experimental Results

The experimental results in this section show the performance of the proposed concept with regard to the four identified evaluation criteria: Validation Time, Total Training Time, Memory Consumption and Quality of Detection.

All experiments within this evaluation are repeated 5 times and each experiment consists of 5 epochs. Model training is performed with batch size 64. The experimental results are averages over all 5 experiment repetitions. The layer size of hidden layers remains constant with 512 neurons per layer. For matrix similarity checking, the absolute tolerance is 10^{-4} and the relative tolerance is 10^{-5}. Both tolerances have been determined empirically and provide the best trade-off between the number of False Positives and successful detection of attacks. For experiments that include model poisoning, noise insertion is applied on the weight gradients of the model, with an attack probability of 0.1. In the case of an attack, a single neuron in the matrix-matrix multiplication is tampered with. The MLP is trained with SGD using a learning rate of 0.01 and a momentum of 0.9. CrossEntropyLoss is used as the error function.

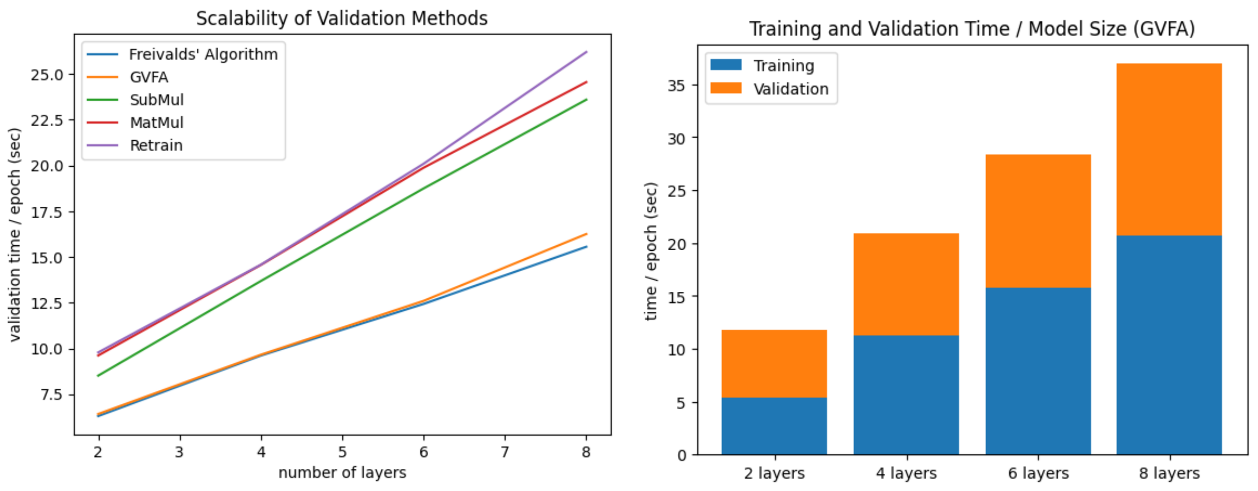
The most important parameters for experiments are mentioned in their description. However, the full parameter setting for the experiments can be found in Appendix B.

6.2.1. Validation Time

The validation time of all validation methods is expected to scale linearly with the number of layers in the model, because the number of matrix-matrix multiplications depends on it. Figure 6.1a shows the scalability of all validation methods. Freivalds' Algorithm and GVFA were both only executed once (i.e., $n\_check = 1$) and the scale factor for Sub-Matrix Multiplication (SubMul) was $s = 0.5$. It can be seen, that Retrain and MatMul have similar performance for a smaller number of layers, but diverge if more layers are used. SubMul scales parallel to MatMul, with slightly faster validation time. Freivalds' Algorithm and GVFA scale similarly by nature and provide significantly lower validation time compared to the other validation methods.

Figure 6.1b shows for GVFA, how the training time and validation time evolves for a rising number of layers. As the computational complexity of GVFA (and Freivalds' Algorithm) is smaller than for matrix-matrix multiplication, the validation time grows slower than the training time. This means that the validation with GVFA and Freivalds' Algorithm becomes more efficient relative to model training with a larger Neural Network.

The validation of a training step contains the validation of activations, gradients, loss and new weights. The proportion to which the individual validation times contribute to the total validation time is visualized in figure 6.3. The data is based on a model with 2 hidden layers. For Freivalds' Algorithm, GVFA and SubMul, the validation of the new weights and the loss is computed in the same way. This can also be seen in the figure, as the respective blocks have equal size. The difference lies in the validation of activation and gradient computation. Here, Freivalds' Algorithm and GVFA provide clearly better performance than SubMul. As a result validating activations and gradients only makes up for half of the needed validation time for Freivalds' Algorithm and GVFA.



(a) Validation time per epoch for all validation methods for different numbers of hidden layers.

(b) Validation and training time of GVFA for different numbers of layers.

Figure 6.1.: Scalability of Validation Methods

This experiment shows that Freivalds' Algorithm and GVFA can successfully reduce the validation time. However, it is important that the observed overhead for the actual training (i.e., the total training time) is minimized, this is evaluated in the next experiment.

6.2.2. Total Training Time

Client-side training validation can be performed in different settings. The training process can be performed on the clients CPU or GPU and the training validation can be executed synchronous (alternating between training and validation) or asynchronous (concurrent training and validation). Figure 6.2 shows, the evaluation of these dimensions, alone and in combination, in terms of total training time. For the synchronous and asynchronous execution, the validation buffer is transferred to the validator through a multiprocessing queue. Reading from the queue introduces some latency that is meant to simulate the overhead introduced by the context switch when a program enters a TEE. As a baseline, the figure also shows the total training time without validation (Training-only). To ensure comparability, training without validation also collects the activations and gradients of the model and stores them in the buffer. Yet, instead of initiating validation, the buffer is discarded. The experiment is run with GVFA and 2 hidden layers. Only one iteration of GVFA was applied (i.e., $n_check = 1$)).

The figure shows, that the total training time for the synchronous setting is almost twice as long as the execution without validation (for both CPU and GPU). The time difference represents the validation time, which is included in the synchronous setting. For the asynchronous execution, the validation introduces almost no overhead to the training time. The overhead for GPU is slightly larger than the one for CPU. The total training time for the asynchronous setting improves upon the synchronous setting as expected. However, it does not match the total training time for the execution without validation. A possible explanation for this could be the overhead introduced by multiprocessing in Python or the overhead of cashing data for both, training and validation. By investigating this issue further in future work, it might be possible to bring asynchronous execution time close to training without validation.

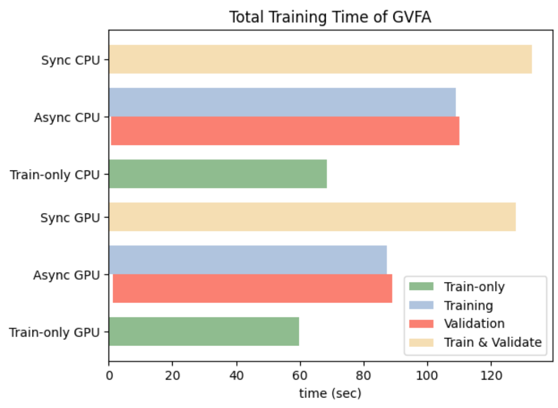


Figure 6.2.: Total Training Time for GVFA with asynchronous (async) execution and GPU usage.

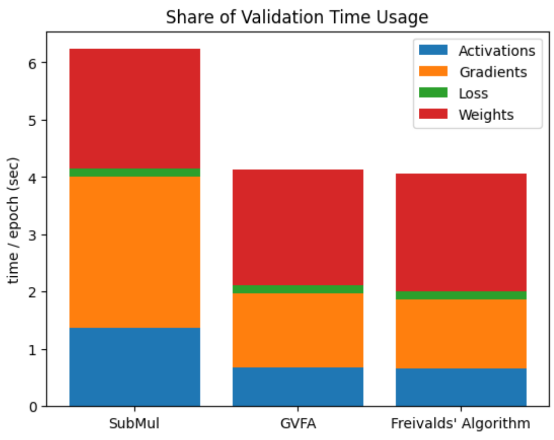


Figure 6.3.: The share of validation time used for validating activations, gradients, loss and new weights.

6.2.3. Memory Consumption

For the application of TEEs, the use of memory is typically limited. As a popular example, Intel Software Guard Extension (SGX) has a total memory capacity of 128 MB [19]. Although the size of newer enclaves is increasing, it is desirable to use as little memory as possible for the validation process. Figure 6.4 shows the memory usage of the validation methods for various model sizes. Measurements were conducted in the asynchronous setting, with a buffer size of 8. The reported memory is the difference between the total memory `peak` and the `base` memory of the python process.

It can be seen, that the dominant factor for memory consumption is the amount of information passed to the validation process. With a growing number of layers, the amount of weight, activation and gradient matrices passed to the validator increases as well. This also causes the memory consumption to go up. Among the validation methods, memory consumption is fairly similar for the same number of layers. However, with a larger model, the baselines Retrain and MatMul have a slightly bigger memory footprint compared to the other validation methods. This increase can probably be attributed to the fact that Retrain and MatMul compute full matrix-matrix multiplications. The validation for small models would fit within the memory of a SGX, whereas the validation of larger models would require a reduced buffer size or further memory optimizations. This is, however, an orthogonal topic, that can be addressed in future work.

6.2.4. Quality of Detection

It is a major goal for the development of defense mechanisms against model poisoning attacks in Federated Learning, to provide reliable protection against these attacks. The effectiveness of the validation methods to detect model poisoning attacks is tested for the worst case scenario, where only one element per matrix product is poisoned. Figure 6.5 shows the recall of Freivalds' Algorithm, GVFA and SubMul for the insertion of noise of different sizes. Only one iteration is performed for each validation method. It is worth mentioning that none of the validation methods produced false positives during the experiments.

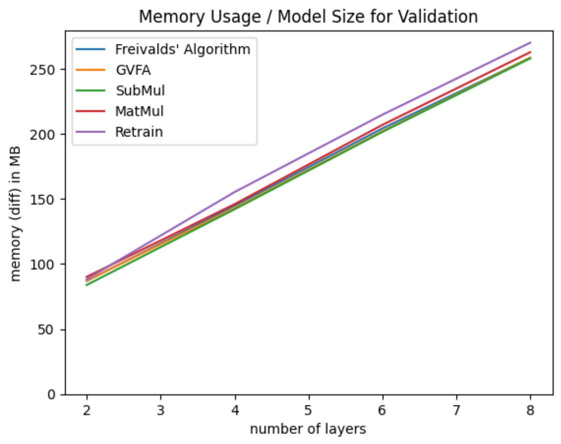


Figure 6.4.: Memory usage of the validation process in dependence of the number of hidden layers.

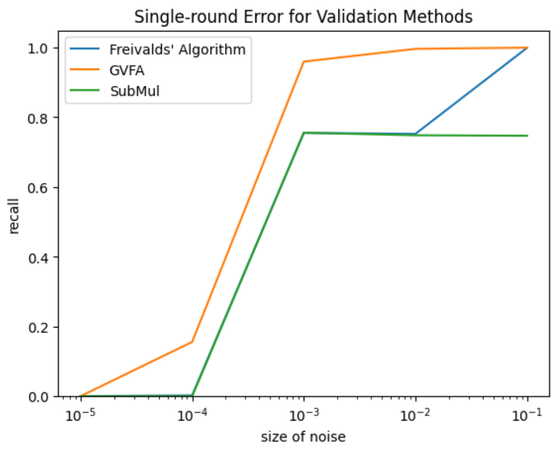
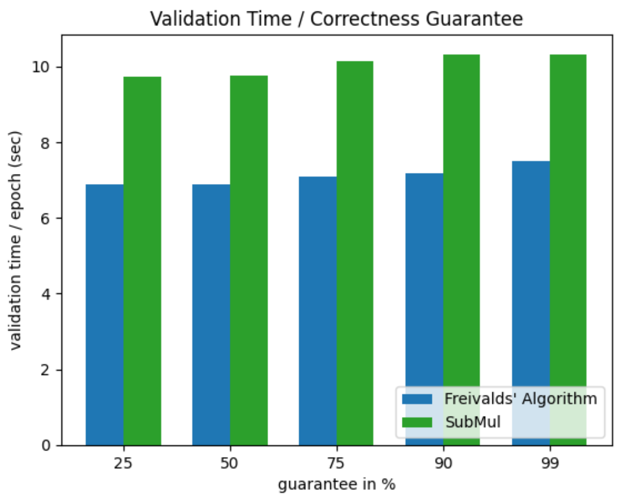


Figure 6.5.: Worst-case detection capabilities of Freivalds' Algorithm, GVFA and SubMul for different noise sizes. One neuron per matrix is poisoned and all validation methods are executed for one iteration.

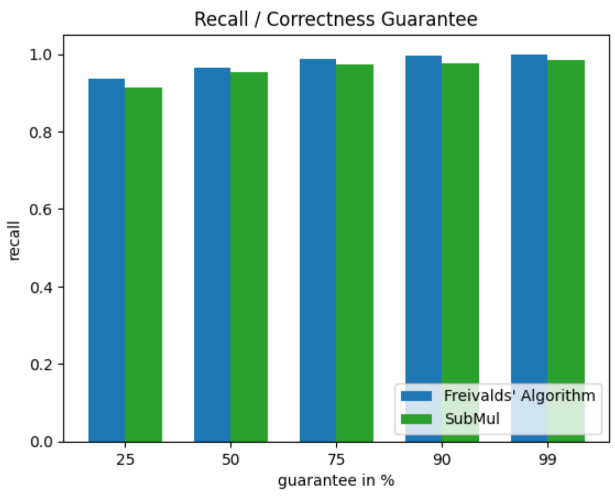
For the insertion of very small noise (10^{-5}, 10^{-4}), the detection capabilities of all validation methods is quite low. This can be attributed to the absolute tolerance for the matrix equality check, which is 10^{-4}. If the size of the inserted error is larger, the capability of the validation methods to detect the attacks grows substantially. As an example, for the insertion of noise of 10^{-1}, Freivalds' Algorithm and GVFA achieve both a recall of 1. The experiment only depicts the worst case scenario. By increasing the number of poisoned elements in the matrix or the number of iterations for the validation methods, the detection capabilities of all validation methods would increase drastically.

The advantage of Freivalds' Algorithm and SubMul is, that these validation methods are able to provide a correctness guarantee for training validation. The last experiment aims at comparing both methods in terms of validation time and detection quality, depending on the desired correctness guarantee. The results are visualized in figure 6.6a for validation time and figure 6.6b for quality of detection. The insertion of smaller noise is harder to detect. Hence, the experiment uses an attack noise of $5 * 10^{-4}$, which is quite small, but still above the absolute tolerance for matrix similarity checking. Freivalds' Algorithm and SubMul both achieve high recall, even for lower levels of guarantee. However, Freivalds' Algorithm always performs slightly better and achieves a recall close to 1 for a guarantee of 99%.

In terms of validation time, Freivalds' Algorithm (FA) outperforms SubMul by 30%. This gap in performance can be attributed to the fact that Freivalds' Algorithm has a better run-time complexity than SubMul and is supposed to increase with a larger model. Providing better security in combination with a significantly lower validation time makes Freivalds' Algorithm clearly superior to SubMul for efficient client-side training validation.



(a) Validation time with respect to the desired correctness guarantee for Freivalds' Algorithm and SubMul.



(b) Recall for model poisoning detection of 1 poisoned neuron with noise $5 * 10^{-4}$ and varying correctness guarantee.

Figure 6.6.: Validation Time and Recall for different Correctness Guarantees

6.3. Discussion of Experimental Results

The experiments presented in the previous section have shown evidence on how well the proposed concept performs according to the four evaluation criteria.

With the application of Freivalds' Algorithm and GVFA for matrix product verification, client-side training validation turns out to work efficiently. For larger networks, the wall-clock time for validation is faster than for training. This indicates, that the validation time with Freivalds' Algorithm and GVFA executed inside a TEE, would be faster than simply training inside the TEE. Analyzing the validation time for the individual components of the validation, shows that validating activations and gradients becomes quite efficient with the application of Freivalds' Algorithm and GVFA. As a result, validating the new weights makes up for almost half of the validation time. Upon further investigation it was found out that the tensor similarity check is the dominating factor for weight validation. Assumably, this can further be optimized, which would make the validation even more efficient. The conclusion can be drawn, that the application of Freivalds' Algorithm and GVFA should be preferred compared to the other validation methods.

The total validation time is shown to benefit from the application of GPUs and asynchronous execution of the validation. The use of GPU can only improve the validation time, if validation performs faster than model training. This effect, which can be seen in the experiments, will probably not occur, if validation is implemented in a real TEE. The main reasons behind this are the overhead of context switches when entering the TEE and the slower execution inside the TEE. However, the asynchronous execution is able to hide some of the validation time, which reduces the training overhead introduced by the validation. This is also beneficial for the application in real TEE.

The evaluation of the memory usage indicates, that for small models, the memory of e.g. Intel SGX would be sufficient to execute the client-side training validation. Here, it is beneficial, that the memory of more recent TEEs is rising rapidly. However, it is worth mentioning, that the data set used for training (MNIST) is a quite small data set and the buffer size of 8 is also quite small. The memory usage is further influenced by the batch size and layer size and can be optimized for individual applications. Clearly, the optimization of the proposed concept in terms of memory consumption was not the primary goal of this thesis, but improvement in this direction is a promising avenue for future work.

The validation methods have shown great performance in detecting a single poisoned neuron with small noise within matrix-matrix multiplications. Even with only one iteration, Freivalds' Algorithm and GVFA provide good detection. SubMul is neglected here, because of its inferiority in comparison to Freivalds' Algorithm and GVFA. If the number of iterations increases, the detection capability of both validation methods would increase exponentially. With an increased number of poisoned neurons per matrix, the detection capabilities would also increase substantially. Therefore, it can be said that the proposed defense mechanism against model poisoning attacks provides great security against these attacks. Further, the protection does not depend on, if the data used for model training is non-iid or changing over time. Moreover, the approach provides the same protection for one and multiple attackers. Even if the number of attackers is more than 50% of the total clients in the Federated Learning (FL) environment. Besides the detection of model poisoning, client-side training validation can also detect or prevent clients, that want to submit sampled gradients.

Freivalds' Algorithm and GVFA are both well suited for the implementation of client-side training validation. The question, which of these methods should be preferred can not be answered completely. While GVFA had a small edge in the case of detecting a single, very small noise, the gap between Freivalds' Algorithm and GVFA would close quickly for an increased number of poisoned neurons or performed iterations. All experiments have been executed with a hidden layer size of 512. However, it requires further investigation, how both algorithms work for larger layer sizes and if increasing numerical error makes a difference here. One benefit that Freivalds' Algorithm has over GVFA is that the correctness guarantee for the application of Freivalds' Algorithm is actually known. For GVFA, the guarantee depends on the machine error during application and is therefore unknown. If a concrete correctness guarantee is required, it is advisable to make use of Freivalds' Algorithm. If this is not the case, both validation methods can be considered.

7. Conclusion and Future Work

7.1. Conclusion

The goal of this thesis was to make Federated Learning more robust against malicious actions. Analyzing the weak points of Federated Learning and several well known attacks showed that model poisoning attacks are the most severe threat to the application of Federated Learning. They are capable of completely manipulating the Machine Learning model and can induce targeted misclassification or hidden functionality.

Several existing defense mechanisms have been investigated and shown to provide limited protection. They either depend heavily on the data that is used for training, or introducing huge communication cost or other kinds of limitation. This thesis proposed a novel defense mechanism against model poisoning attacks, that leverages client-side training validation to achieve efficient and reliable attack detection. The approach is based on the efficient validation of matrix-matrix multiplications in a trusted validator component, that can be implemented with a client-side Trusted Execution Environment (TEE). Client-side training validation allows the derivation of correctness guarantees that can be adjusted by the underlying validation method.

The evaluation of the client-side training validation approach shows the efficiency of the selected validation scheme and a minor overhead in terms of total training time, even for a high guarantee of 99%. Finally, the approach proves to reliably detect model poisoning attacks and to have good scalability, which makes it also applicable for larger models. The evaluation has also uncovered further steps that can be taken to improve the applicability of the presented concept.

7.2. Future Work

Evaluation has shown that client-side training validation has room for improvement to reduce the amount of memory required for validating model training. Furthermore, by investigating inefficiencies in the asynchronous execution, it is likely that the total training time with validation can further be reduced. Also, the weight comparison during validation has hidden potential for run-time improvements. While these are more implementation specific details, further possibilities for improvement at the conceptual level are possible. An interesting avenue might be the application of Matrix Product Approximation as a further validation method. This is also expected to yield great performance and it would be interesting to see its performance and guarantees compared to Freivalds' Algorithm and GVFA. While currently only supporting validation for Multi-layer Perceptrons (MLPs), the concept can be further generalized to other layer types like convolutional layers. If the participants have sufficient hardware resources, like multiple TEEs, it is also conceivable to further parallelize the validation component. Apart from improving the presented approach itself, it remains open to see how the proposed concept performs when implemented within a real TEE. While the concept proposed in this thesis was focused on defending against model poisoning attacks, it is an interesting route to extend the approach by functionality to also detect data poisoning attacks by for example sharing data statistics among validation components.

Bibliography

[1] Eugene Bagdasaryan et al. “How To Backdoor Federated Learning”. In: *CoRR* abs/1807.00459 (2018). arXiv: 1807.00459. URL: http://arxiv.org/abs/1807.00459.

[2] Moran Baruch, Gilad Baruch, and Yoav Goldberg. *A Little Is Enough: Circumventing Defenses For Distributed Learning*. 2019. arXiv: 1902.06156 [cs.LG].

[3] Arjun Nitin Bhagoji et al. *Analyzing Federated Learning through an Adversarial Lens*. 2019. arXiv: 1811.12470 [cs.LG].

[4] Christopher M Bishop. *Pattern Recognition and Machine Learning*. Ed. by M Jordan, J Kleinberg, and B Schölkopf. Vol. 4. Information science and statistics 4. Springer, 2006. Chap. Graphical, p. 738. ISBN: 9780387310732. DOI: 10.1117/1.2819119. arXiv: 0-387-31073-8. URL: http://www.library.wisc.edu/selectedtocs/bg0137.pdf.

[5] Davis Blalock and John Guttag. *Multiplying Matrices Without Multiplying*. 2021. arXiv: 2106.10860 [cs.LG].

[6] Davis W. Blalock and John V. Guttag. “Bolt: Accelerated Data Mining with Fast Vector Compression”. In: *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Aug. 2017). DOI: 10.1145/3097983.3098195. URL: http://dx.doi.org/10.1145/3097983.3098195.

[7] Peva Blanchard et al. “Machine Learning with Adversaries: Byzantine Tolerant Gradient Descent”. In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS’17. Long Beach, California, USA: Curran Associates Inc., 2017, pp. 118–128. ISBN: 9781510860964.

[8] Xiaoyu Cao, Jinyuan Jia, and Neil Zhenqiang Gong. *Provably Secure Federated Learning against Malicious Clients*. 2021. arXiv: 2102.01854 [cs.CR].

[9] Lingjiao Chen et al. *DRACO: Byzantine-resilient Distributed Training via Redundant Gradients*. 2018. arXiv: 1803.09877 [stat.ML].

[10] Xinyun Chen et al. *Targeted Backdoor Attacks on Deep Learning Systems Using Data Poisoning*. 2017. arXiv: 1712.05526 [cs.CR].

[11] Yu Cheng et al. *A Survey of Model Compression and Acceleration for Deep Neural Networks*. 2020. arXiv: 1710.09282 [cs.LG].

[12] Edith Cohen and David D. Lewis. “Approximating Matrix Multiplication for Pattern Recognition Tasks”. In: *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA ’97. New Orleans, Louisiana, USA: Society for Industrial and Applied Mathematics, 1997, pp. 682–691. ISBN: 0898713900.

[13] Confidential Computing Consortium. *Confidential Computing: Hardware-Based Trusted Execution for Applications and Data*. 2021. URL: https://confidentialcomputing.io/wp-content/uploads/sites/85/2021/03/confidentialcomputing_outreach_whitepaper-8-5x11-1.pdf (visited on 11/18/2021).

[14] Graphene Contributors. *PyTorch PPML Framework Tutorial*. 2021. URL: https://graphene.readthedocs.io/en/latest/tutorials/pytorch/index.html (visited on 11/25/2021).

[15] Torch Contributors. *AUTOMATIC MIXED PRECISION PACKAGE - TORCH.CUDA.AMP*. 2019. URL: https://pytorch.org/docs/stable/amp.html (visited on 11/11/2021).

[16] Torch Contributors. *Dataset*. 2021. URL: https://pytorch.org/docs/stable/data.html#torch.utils.data.Dataset (visited on 11/25/2021).

[17] Torch Contributors. *Register Forward Hook*. 2021. URL: https://pytorch.org/docs/stable/generated/torch.nn.Module.html?highlight=register_forward_hook#torch.nn.Module.register_forward_hook (visited on 11/25/2021).

[18] Torch Contributors. *Register Full Backward Hook*. 2021. URL: https://pytorch.org/docs/stable/generated/torch.nn.Module.html?highlight=register_forward_hook#torch.nn.Module.register_full_backward_hook (visited on 11/25/2021).

[19] Victor Costan and Srinivas Devadas. *Intel SGX Explained*. Tech. rep. 086. 2016. URL: http://eprint.iacr.org/2016/086 (visited on 11/18/2021).

[20] NVIDIA DEVELOPER. *Automatic Mixed Precision for Deep Learning*. URL: https://developer.nvidia.com/automatic-mixed-precision (visited on 11/11/2021).

[21] Google Developers. *Mixed precision*. 2021. URL: https://www.tensorflow.org/guide/mixed_precision (visited on 11/11/2021).

[22] scikit-learn developers. *scikit-learn*. 2021. URL: https://scikit-learn.org/stable/ (visited on 11/14/2021).

[23] John R. Douceur. “The Sybil Attack”. In: *Revised Papers from the First International Workshop on Peer-to-Peer Systems*. IPTPS ’01. 2002, pp. 251–260. ISBN: 3540441794.

[24] Python Software Foundation. *Python Time*. 2021. URL: https://docs.python.org/3/library/time.html#time.time (visited on 11/26/2021).

[25] Rūsiņš Freivalds. “Fast probabilistic algorithms”. en. In: *Mathematical Foundations of Computer Science 1979*. Ed. by G. Goos et al. Vol. 74. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1979, pp. 57–69. ISBN: 978-3-540-09526-2 978-3-540-35088-0. DOI: 10.1007/3-540-09526-8_5. URL: http://link.springer.com/10.1007/3-540-09526-8_5 (visited on 03/25/2021).

[26] Clement Fung, Chris J. M. Yoon, and Ivan Beschastnikh. *Mitigating Sybils in Federated Learning Poisoning*. 2020. arXiv: 1808.04866 [cs.LG].

[27] François Le Gall. “Powers of Tensors and Fast Matrix Multiplication”. In: *CoRR* abs/1401.7714 (2014). arXiv: 1401.7714. URL: http://arxiv.org/abs/1401.7714.

[28] Tiezheng Ge et al. “Optimized Product Quantization”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 36.4 (2014), pp. 744–755. DOI: 10.1109/TPAMI.2013.240.

[29] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL: http://www.deeplearningbook.org.

[30] Google. *TensorFlow*. 2021. URL: https://www.tensorflow.org/ (visited on 11/14/2021).

[31] Margherita Grandini, Enrico Bagli, and Giorgio Visani. *Metrics for Multi-Class Classification: an Overview*. 2020. arXiv: 2008.05756 [stat.ML].

[32] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The elements of statistical learning: data mining, inference and prediction*. 2nd ed. Springer, 2009. URL: http://www-stat.stanford.edu/~tibs/ElemStatLearn/.

[33] Ling Huang et al. “Adversarial Machine Learning”. en. In: (2011), p. 15.

[34] Herve Jégou, Matthijs Douze, and Cordelia Schmid. “Product Quantization for Nearest Neighbor Search”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33.1 (2011), pp. 117–128. DOI: 10.1109/TPAMI.2010.57.

[35] Hao Ji, Michael Mascagni, and Yaohang Li. *Gaussian Variant of Freivalds’ Algorithm for Efficient and Reliable Matrix Product Verification*. 2017. arXiv: 1705.10449 [cs.DS].

[36] David Kaplan, Jeremy Powell, and Tom Woller. *AMD MEMORY ENCRYPTION*. 2016. URL: https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf (visited on 11/18/2021).

[37] Thomas Knauth et al. *Integrating Remote Attestation with Transport Layer Security*. 2019. arXiv: 1801.05863 [cs.CR].

[38] Europäische Kommission. *Datenschutz*. 2021. URL: https://ec.europa.eu/info/law/law-topic/data-protection_de (visited on 11/16/2021).

[39] Hugo Krawczyk. “SIGMA: The ‘SIGn-and-MAc’ Approach to Authenticated Diffie-Hellman and Its Use in the IKE Protocols”. en. In: *Advances in Cryptology - CRYPTO 2003*. Ed. by Dan Boneh. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2003, pp. 400–425. ISBN: 978-3-540-45146-4. DOI: 10.1007/978-3-540-45146-4_24.

[40] Yann LeCun. *THE MNIST DATABASE*. 2021. URL: http://yann.lecun.com/exdb/mnist/ (visited on 11/27/2021).

[41] Lingjuan Lyu, Han Yu, and Qiang Yang. “Threats to Federated Learning: A Survey”. In: *arXiv:2003.02133 [cs, stat]* (Mar. 2020). arXiv: 2003.02133. URL: http://arxiv.org/abs/2003.02133 (visited on 12/21/2020).

[42] Lingjuan Lyu et al. *Privacy and Robustness in Federated Learning: Attacks and Defenses*. 2020. arXiv: 2012.06337 [cs.CR].

[43] Shiva Manne and Manjish Pal. *Fast Approximate Matrix Multiplication by Solving Linear Systems*. 2014. arXiv: 1408.4230 [cs.DS].

[44] Frank McKeen et al. “Innovative Instructions and Software Model for Isolated Execution”. In: *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. HASP ’13. Tel-Aviv, Israel: Association for Computing Machinery, 2013. ISBN: 9781450321181. DOI: 10.1145/2487726.2488368. URL: https://doi.org/10.1145/2487726.2488368.

[45] H. Brendan McMahan et al. “Communication-Efficient Learning of Deep Networks from Decentralized Data”. In: *arXiv:1602.05629 [cs]* (Feb. 2017). arXiv: 1602.05629. URL: http://arxiv.org/abs/1602.05629 (visited on 12/04/2020).

[46] Deep Mind. *AlphaGo*. 2021. URL: https://deepmind.com/research/case-studies/alphago-the-story-so-far (visited on 11/16/2021).

[47] Tom M. Mitchell. *Machine Learning*. New York: McGraw-Hill, 1997. ISBN: 978-0-07-042807-2.

[48] Fan Mo et al. *PPFL: Privacy-preserving Federated Learning with Trusted Execution Environments*. 2021. arXiv: 2104.14380 [cs.CR].

[49] Saeid Mofrad et al. “A Comparison Study of Intel SGX and AMD Memory Encryption Technology”. In: *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*. HASP ’18. Los Angeles, California: Association for Computing Machinery, 2018. ISBN: 9781450365000. DOI: 10.1145/3214292.3214301. URL: https://doi.org/10.1145/3214292.3214301.

[50] Kevin P. Murphy. *Machine learning : a probabilistic perspective*. Cambridge, Mass. [u.a.]: MIT Press, 2013. ISBN: 9780262018029 0262018020.

[51] Stack Overflow. *2021 Developer Survey*. 2021. URL: https://insights.stackoverflow.com/survey/2021 (visited on 11/13/2021).

[52] Nicolae Paladi, Linus Karlsson, and Khalid Elbashir. “Trust Anchors in Software Defined Networks”. en. In: *Computer Security*. Ed. by Javier Lopez, Jianying Zhou, and Miguel Soriano. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 485–504. ISBN: 978-3-319-98989-1. DOI: 10.1007/978-3-319-98989-1_24.

[53] Krishna Pillutla, Sham M. Kakade, and Zaid Harchaoui. *Robust Aggregation for Federated Learning*. 2019. arXiv: 1912.13445 [stat.ML].

[54] Ramon Quiza and J. Davim. “Computational Methods and Optimization”. In: Jan. 2011, pp. 177–208. ISBN: 978-1-84996-449-4. DOI: 10.1007/978-1-84996-450-0.

[55] Shashank Rajput et al. *DETOX: A Redundancy-based Framework for Faster and More Robust Gradient Aggregation*. 2020. arXiv: 1907.12205 [cs.LG].

[56] Giampaolo Rodola. *PsUtil Process Memory Info*. 2021. URL: https://psutil.readthedocs.io/en/latest/index.html?highlight=memory_info#psutil.Process.memory_info (visited on 11/26/2021).

[57] Ali Shafahi et al. “Poison Frogs! Targeted Clean-Label Poisoning Attacks on Neural Networks”. In: *arXiv:1804.00792 [cs, stat]* (Nov. 2018). arXiv: 1804.00792 version: 2. URL: http://arxiv.org/abs/1804.00792 (visited on 08/19/2021).

[58] Facebook Open Source. *PyTorch*. 2021. URL: https://pytorch.org/ (visited on 11/14/2021).

[59] V. STRASSEN. “Gaussian Elimination is not Optimal.” In: *Numerische Mathematik* 13 (1969), pp. 354–356. URL: http://eudml.org/doc/131927.

[60] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. URL: http://incompleteideas.net/book/the-book-2nd.html.

[61] Tesla. *Future of Driving*. 2021. URL: https://www.tesla.com/autopilot (visited on 11/16/2021).

[62] Neil C. Thompson et al. *The Computational Limits of Deep Learning*. 2020. arXiv: 2007.05558 [cs.LG].

[63] Florian Tramèr and Dan Boneh. *Slalom: Fast, Verifiable and Private Execution of Neural Networks in Trusted Hardware*. 2019. arXiv: 1806.03287 [stat.ML].

[64] Dinh Ngoc Tu et al. “Everything You Should Know about Intel SGX Performance on Virtualized Systems”. In: *ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2019)*. Phoenix, United States, June 2019, pp. 1–21. URL: https://hal.archives-ouvertes.fr/hal-02947792.

[65] Hongyi Wang et al. *Attack of the Tails: Yes, You Really Can Backdoor Federated Learning*. 2020. arXiv: 2007.05084 [cs.LG].

[66] Jianyu Wang et al. *A Field Guide to Federated Optimization*. 2021. arXiv: 2107.06917 [cs.LG].

[67] Joseph Yacim and Douw Boshoff. "Impact of Artificial Neural Networks Training Algorithms on Accurate Prediction of Property Values". In: *Journal of Real Estate Research* 40 (Nov. 2018), pp. 375–418. DOI: 10.1080/10835547.2018.12091505.

[68] Dong Yin et al. *Byzantine-Robust Distributed Learning: Towards Optimal Statistical Rates*. 2021. arXiv: 1803.01498 [cs.LG].

[69] Xiaoli Zhang et al. *Enabling Execution Assurance of Federated Learning at Untrusted Participants*. 2020. DOI: 10.1109/INFOCOM41043.2020.9155414.

[70] Lingchen Zhao et al. *Shielding Collaborative Learning: Mitigating Poisoning Attacks through Client-Side Detection*. 2020. arXiv: 1910.13111 [cs.CR].

[71] Shijun Zhao et al. "SecTEE: A Software-Based Approach to Secure Enclave Architecture Using TEE". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS '19. London, United Kingdom: Association for Computing Machinery, 2019, pp. 1723–1740. ISBN: 9781450367479. DOI: 10.1145/3319535.3363205. URL: https://doi.org/10.1145/3319535.3363205.

[72] Xingchen Zhou et al. "Deep Model Poisoning Attack on Federated Learning". In: *Future Internet* 13.3 (2021). ISSN: 1999-5903. DOI: 10.3390/fi13030073. URL: https://www.mdpi.com/1999-5903/13/3/73.

[73] Feng Zhu et al. *Towards Unified INT8 Training for Convolutional Neural Network*. 2019. arXiv: 1912.12607 [cs.LG].

A. Lower Bound for Training-step Guarantee

For the Training-step guarantee, a lower bound can be derived. Let L be the number of layers in an MLP. One training step is validated by validating all matrix-matrix multiplications in the forward-pass and backward-pass with a validation method with a correctness guarantee p. Assuming q to be the desired correctness guarantee for the validation of the training step. A lower bound can be derived for the maximum probability p to match the desired guarantee q.

Starting with the correctness guarantee of validating the training step $(1-p)^{3L-1} \geq q$. From equation

$$(1-p)^2 = 1 - 2p + p^2$$

a lower bound can be derived by dropping p^2

$$(1-p)^2 \geq 1 - 2p$$

This can be done multiple times

$$\begin{aligned}(1-p)^3 &\geq (1-2p)(1-p)\\(1-p)^3 &\geq 1 - 3p + 2p^2\\(1-p)^3 &\geq 1 - 3p\end{aligned}$$

So for L layers it generalizes to

$$(1-p)^L \geq 1 - Lp_{low}$$

$1 - Lp_{low}$ is the lower bound for $(1-p)^L$, so if

$$1 - Lp_{low} = q \Rightarrow p_{low} = (1-q)/L$$

then this holds

$$(1-p)^L \geq q$$

Therefore it is safe to say that

$$\begin{aligned}p &\leq p_low\\p &\leq (1-q)/L\end{aligned}$$

B. Experiment Parameters

The experiments conducted in the Evaluation Chapter were executed with the parameters that are stated in the following tables. The values for parameters that remain constant for all experiments, are presented in table B.1. The individual values for parameters that are not consistent among the experiments, are individually shown in separate tables per experiment.

Parameter	**Value**
repetitions	5
epochs	5
layer size	512
batch size	64
attack probability	10%
n_neurons	1
absolute tolerance	10^{-4}
relative tolerance	10^{-5}
data set	MNIST
optimizer	Stochastic Gradient Decent (SGD) with learning rate 0.01 and momentum 0.9
error function	CrossEntropyLoss

Table B.1.: Experiment parameters that are fixed for all experiments.

B.1. Validation Method Scalability

Parameter	Values
validation method	Freivalds' Algorithm, GVFA, SubMul, MatMul, Retrain
model size	2, 4, 6, 8
execution mode	synchronous (sync)
device	CPU
buffer size	32
n_check	1

Table B.2.: Parameters for Experiment *exp_time_attack_scalability*

B.2. Optimization with GPU and Asynchronous Validation

Parameter	Values
validation method	GVFA
model size	2
execution mode	training without validation, synchronous (sync), asynchronous (async)
device	CPU, GPU
buffer size	32
n_check	1
Notes	For the sync and async execution setting, the validation buffer is transferred to the validator with a multiprocessing queue. At this point, the queue simulates the context switch, that is introduced, when a program enters a TEE. In the no-validation setting, the data for activations and gradients is also collected during training. However, instead of validating the validation buffer, it is discarded.

Table B.3.: Parameters for Experiment *exp_time_device_concurrency*

B.3. Memory Consumption per Model Size

Parameter	Values
validation method	Freivalds' Algorithm, GVFA, SubMul, MatMul, Retrain
model size	2, 4, 6, 8
execution mode	asynchronous (async)
device	CPU
buffer size	8
n_check	1

Table B.4.: Parameters for Experiment *exp_mem_model_size*

B.4. Noise Detection Capability of Validation Methods

Parameter	Values
validation method	Freivalds' Algorithm, GVFA, SubMul
model size	2
execution mode	synchronous (sync)
device	CPU
buffer size	32
n_check	1
attack noise	10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}

Table B.5.: Parameters for Experiment *exp_attack_varying_noise*

B.5. Noise Detection Capability and Validation Time per Guarantee Level

Parameter	Values
validation method	Freivalds' Algorithm, SubMul
model size	2
execution mode	synchronous (sync)
device	CPU
buffer size	32
guarantee	25%, 50%, 75%, 90%, 99%
attack noise	$5 * 10^{-4}$

Table B.6.: Parameters for Experiment *exp_time_attack_varying_guarantee*