UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Attack-Resilient Federated Machine Learning at Nodes

Filipe Dinis Ferreira Rodrigues

Mestrado em Segurança Informática

Dissertação orientada por:
Nuno Fuentecilla Maia Ferreira Neves

2024

Agradecimentos

Em primeiro lugar quero agradecer ao meu orientador, Nuno Neves, por toda a sua dedicação, paciência e orientação durante esta importante etapa. Pessoalmente, foi um privilégio enorme ter tido a oportunidade de trabalhar ao seu lado, superando todos os desafios e obstáculos. Nunca será possível retribuir o apoio que me deu nos momentos mais difíceis, um profundo obrigado.

Ao meu colega de mestrado, Rodrigo Simões, por ter partilhado comigo parte desta jornada, sendo o seu contributo indispensável para esta minha conquista. Foi um prazer ter tido a oportunidade de discutir ideias e de colaborar a seu lado a fim de ultrapassarmos os diversos desafios a que fomos submetidos. Mais do que um colega, um amigo, que mais do que ninguém percebia as frustrações e as dificuldades nos momentos mais difíceis da realização deste trabalho, justamente por estar no mesmo lugar que eu.

À minha namorada, que ao longo desta jornada foi a minha luz nos dias mais escuros e o meu porto de abrigo nos momentos mais turbulentos. Não há palavras suficientes para expressar a minha gratidão pelo teu apoio incondicional.

Aos meus queridos pais, por terem tornado possível todo este percurso académico, apoiando-me desde o início até ao fim. Palavras não chegam para expressar esta gratidão, especialmente tendo em conta todos os sacrifícios que fizeram para que me tornasse o homem que sou hoje. Nas fases menos boas, pensei em desistir, mas os valores que eles me transmitiram, como a persistência e resiliência, fizeram com que enfrentasse os desafios que tinha pela frente e conseguisse ter sucesso. Obrigado por tudo.

Aos meus irmãos mais velhos, cujos percursos académicos me motivaram e me influenciaram a seguir o caminho da ciência. Guardo com carinho todos os momentos que passámos na infância e na adolescência, que foram um pilar para que me formasse na pessoa que sou hoje. Vocês serão sempre a minha maior inspiração.

A todas as restantes pessoas que fazem parte da minha vida, resta agradecer pelo apoio que me deram ao longo de todos estes anos, acompanhando-me e celebrando comigo cada uma das minhas conquistas.

Este trabalho foi parcialmente suportado pelo Programa Horizonte Europa 2020 sob acordo de subvenção No 871259 (projeto ADMORPH) e pela FCT através da unidade de investigação LASIGE (UIDB/00408/2020 & UIDP/00408/2020).

Aos meus queridos pais.

Resumo

Nos últimos anos, a aprendizagem automática tem contribuído significativamente para que várias indústrias possam desenvolver sistemas mais complexos e inteligentes, com aplicações úteis para a sociedade, como a condução autónoma e o reconhecimento de voz. A aprendizagem profunda, um tipo específico de aprendizagem automática, permite a criação de modelos complexos capazes de extrair características de dados, produzindo resultados relevantes sobre estes mesmos.

Estes modelos são compostos por estruturas matemáticas que efetuam cálculos nos dados, tais como imagens, frases ou amostras de áudio. O processo de criação destes modelos, também designado por treino, consiste em encontrar valores adequados para os parâmetros dessas estruturas, de modo a que os resultados sejam suficientemente precisos para que os utilizadores possam beneficiar deles. Os requisitos para treinar modelos são grandes quantidades de dados, para que os modelos possam *aprender* uma grande variedade de características/representações, e capacidade computacional para *afinar* os milhões de parâmetros que compõem os modelos mais avançados.

As soluções de aprendizagem automática distribuída resolvem o problema de escalabilidada criado aquando do treino de modelos complexos. Por exemplo, a utilização dos dados gerados por telemóveis (smartphones) e dispositivos IoT é extremamente interessante, uma vez que muitos deles seriam úteis e valiosos para o treino de modelos. No entanto, questões relacionadas com a privacidade surgem, associadas às informações contidas nesses dados, uma vez que a aprendizagem automática distribuída tradicional exige o acesso externo a esses dados. Além disso, mesmo considerando uma situação hipotética em que a privacidade dos dados é facultativa e estes podem ser partilhados com uma entidade central de forma livre, fazê-lo com dispositivos com baixo poder de processamento, como dispositivos IoT, é extremamente ineficiente, uma vez que os custos de comunicação aumentarão consideravelmente.

A Aprendizagem Federada, ou *Federated Learning* (FL), é uma abordagem de aprendizagem distribuída que tem como objetivo treinar modelos sobre dados decentralizados. Esta abordagem resolve as limitações da aprendizagem automática distribuída tradicional, que assume que um servidor central recebe e armazena todos os dados – uma possibilidade que pode ser irrealista tanto em termos da quantidade de dados gerados pelos dispositivos como em termos de preservação da privacidade, como explicado anteriormente. Além disso, o FL também contribui para a redução dos custos de comunicação em cenários de aprendizagem automática distribuída, uma vez que os dados não são transferidos entre os dispositivos e o servidor central. Esta propriedade é especialmente relevante em cenários em que os dispositivos que estão a participar no processo de FL têm uma largura de banda e uma capacidade de computação reduzidas (dispositivos IoT).

O FL baseia-se num servidor central que coordena vários clientes (proprietários de dados). O treino é organizado em rondas - em cada ronda, o servidor selecciona um subconjunto de clientes e pede-lhes que treinem a versão atual do modelo com os seus dados; quando o treino termina, os clientes enviam as suas actualizações do modelo de volta para o servidor, que as agrega com base num algoritmo de agregação (tipicamente, a média entre todas as atualizações); o resultado é uma nova versão do modelo global, que é enviada aos clientes na ronda de treino seguinte.

O interesse em treinar modelos sobre dados descentralizados está associado à diversidade de dados armazenados nos dispositivos dos clientes, que podem ser utilizadas para melhorar as aplicações existentes, como teclados inteligentes (por exemplo, Google GBoard), a condução autónoma e o reconhecimento automático por voz (por exemplo, Siri). A tipologia dos dados processados por essas aplicações pode ser altamente sensível e um utilizador normal pode não ser conivente em que estes sejam cedidos, devido ao risco da sua exposição em bruto. Esses riscos são extremamente reduzidos quando se utiliza FL, uma vez que os dados nunca são enviados dos clientes para o servidor.

Outras aplicações em que o FL é útil são os setores da saúde e o financeiro. No que diz respeito à saúde, os casos de utilização do FL incluem a previsão de doenças, a análise de imagens médicas e a monitorização de pacientes, preservando a privacidade dos pacientes e cumprindo a regulamentação em matéria de proteção de dados. No que diz respeito ao setor financeiro, o FL pode auxiliar os bancos e as instituições financeiras a desenvolver modelos de avaliação de riscos mais precisos, algoritmos de deteção de fraude e sistemas de classificação de crédito de clientes sem comprometer a privacidade da informação financeira dos clientes.

Embora o FL seja uma tecnologia que traz benefícios relacionados com a privacidade, os ataques a estes sistemas continuam a ser possíveis devido ao facto de ser uma tecnologia distribuída com dispositivos que podem ser comprometidos e cujo controlo é limitado. O estudo atual dos mecanismos de segurança em FL está ainda num estado prematuro, com novos ataques e defesas a serem desenvolvidos a um ritmo crescente. Um ator malicioso com capacidade de comprometer alguns dos clientes pode criar um ataque que interfira com o processo de treino, alterando o comportamento do modelo final e, dependendo da aplicação, as consequências podem ser críticas. Por exemplo, suponhamos que está a ser desenvolvido um modelo de condução autónoma para automóveis. Um atacante poderia manipular o modelo de modo a que os sinais de paragem fossem interpretados como sinais de prioridade, fazendo com que os carros avançassem imediatamente numa interceção de estrada, possivelmente provocando um acidente.

Por esta razão, é essencial compreender como estes sistemas podem ser atacados e como protegê-los, municiando as soluções de FL com mecanismos mais robustos e resilientes que possam suportar atores maliciosos cujo objetivo é alterar o comportamento do modelo global.

Ao analisar os ataques aos nós na literatura atual e as suas implementações, deparamo-nos com aplicações complexas, não modulares e com extensibilidade limitada. Além disso, as avaliações atuais de ataques e defesas em FL não são realistas, pois não utilizam os ambientes de rede – por exemplo, o treino de vários clientes no processo de FL é realizado numa única máquina.

Além disso, o desenvolvimento de novos ataques e defesas exige modificações complexas nas implementações existentes – o que constitui um desafio para o desenvolvimento de novos mecanismos de segurança de forma rápida e flexível.

Para tratar destas limitações, este trabalho propõe uma ferramenta chamada FADO (Federated Attack and Defense Orchestrator) que possibilita aos utilizadores a interação com uma plataforma que permite executar e avaliar rapidamente cenários de FL sob vários tipos de ataques e defesas. O FADO é escalável, com a capacidade de executar cenários de FL com milhares de clientes, garantindo realismo e um alto grau de personalização. Os utilizadores têm a possibilidade de integrar as suas próprias implementações de diferentes componentes de FL no FADO com pouca complexidade graças aos seus mecanismos de extensibilidade.

Além disso, este trabalho também propôs um novo ataque *backdoor* a que chamámos *BLARE*, que foi concebido para melhorar a persistência do *backdoor* no modelo global, mesmo depois de o atacante deixar de participar no processo de FL. Através da manipulação dos parâmetros do modelo produzido aquando do treino do *backdoor*, o BLARE possibilita que o *backdoor* permaneça eficaz por mais tempo do que os ataques de *backdoor* observados até ao momento. Para avaliar o BLARE, implementamo-lo no FADO, com os conjuntos de dados CIFAR10 e CIFAR100, efetuando várias experiências para avaliar o impacto dos vários parâmetros na eficácia do ataque. Os resultados indicam que o BLARE injecta com sucesso uma *backdoor* com elevada durabilidade no processo de FL. Ao implementar o BLARE no FADO, demonstrámos também a utilidade do FADO na avaliação de cenários de FL, em particular, aqueles que envolvam ataques aos nós.

Palavras-chave: aprendizagem federada, privacidade, ataque backdoor, segurança, inteligência artifical

Abstract

Federated learning (FL) emerged as a solution for distributed machine learning, enabling the training of models from decentralized data (e.g., stored in clients' devices) without compromising their privacy. Its distributed nature makes it vulnerable to attacks from adversaries, resulting in possible degraded inference behavior for targeted examples or general misclassification of dataset instances. Hence, the implementation of security mechanisms to prevent nefarious actions from impacting the global model is critical to ensure that FL is a safe environment to train a model collaboratively while assuring clients' data privacy.

Evaluations of node attacks in current FL literature require complex modifications to create or integrate new attacks, which entail considerable effort. Therefore, in this work, we propose a tool called FADO that provides researchers and industry professionals with a platform to quickly evaluate FL scenarios under various types of attacks. FADO is scalable, capable of executing FL scenarios with thousands of clients while ensuring realism and a high degree of customizability. Users can integrate their implementations of different FL components into FADO with minimal complexity, thanks to its interfaces that facilitate the loading of external modules.

Additionally, we introduce a novel backdoor attack called BLARE, which manipulates the attacker's model parameters during training to make the backdoor more persistent in the global model, even after the attacker stops injecting malicious updates. We evaluated this backdoor attack using FADO with two datasets: CIFAR10 and CIFAR100. Several experiments were conducted in which we varied specific hyperparameters to understand their impact on BLARE's effectiveness.

We conclude that FADO and BLARE significantly contribute to the understanding of node attacks in FL, offering valuable insights to improve the robustness and security of FL systems.

Keywords: federated learning, security, framework, node attacks, defenses

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Machine learning is helping several industries develop more complex and smart systems with useful applications for society, such as autonomous driving [14] and voice recognition [36]. Deep learning, a specific type of machine learning, enables the training of complex models able to extract meaningful representations from data and output relevant results.

These models consist of structures that perform computations on data inputs, such as images, phrases, or audio samples. The training of these structures is the process of finding appropriate values for their parameters so that the results are accurate enough for users to benefit from them. The requirements to train models are large quantities of data samples (so models can learn a diversity of representations) and computational power (to fine-tune the millions of parameters that compose state-of-the-art models).

Distributed machine learning solutions appeared to address scalability problems created when training complex models [12]. For example, it would be attractive to use the data generated by mobile (smartphones) and IoT devices, since much of it would be useful for the training of large scale models. However, questions arise regarding privacy issues associated with the information contained in such data, as traditional distributed machine learning requires external access to that data. Moreover, even considering the situation where privacy guarantees are not mandatory and data can be shared with a central entity, doing so with edge devices is extremely inefficient, as communication costs will increase considerably.

Federated Learning (FL) [32, 23] is a distributed learning approach that aims to train models over decentralized data. This approach tackles the limitations of traditional distributed machine learning, which assumes that a central server stores all the examples – a possibility that can be unrealistic both in terms of the quantity of the data generated by devices and in terms of preserving privacy, as described previously. Moreover, FL also contributes to a decrease of communication costs in distributed machine learning scenarios, as data is not transferred between the devices and the central server. This property is specially useful in scenarios where the devices that are participating in the FL process have low bandwidth and computing capacity (edge devices).

Applications

The interest in training over decentralized data is tied to the variety of information collected and stored on end-user devices, which can be utilized to improve existing applications, such as next-word prediction (e.g., Google GBoard) [52], autonomous driving [14], and automatic speech recognition (e.g., Siri) [36]. The nature of the data processed by such applications can be highly sensitive, and a regular user may not be interested in giving it away because of the risk of the raw data being exposed. Such risks become unlikely when resorting to FL, as the data never leaves the devices.

More applications in which FL is extremely useful are healthcare and finance sectors. Regarding healthcare, use cases of FL include disease prediction, medical image analysis, drug discovery, and patient monitoring, while preserving patient privacy and adhering to data protection regulation. When it comes to the financial sector, FL can help banks and financial institutions develop more accurate risk assessment models, fraud detection algorithms, and customer credit scoring systems without compromising customers financial data.

FL is based on a central server coordinating several clients (data owners). The training is organized in rounds – in each round, the server selects a subset of the clients and requests them to train the current version of the model with their data; when the training ends, clients send their model updates back to the server, which merges them all based on an aggregation algorithm; the result is a new version of the global model, which is pushed to the clients in the next round of training. This form of operation is called *cross-device*. The alternative, called *cross-silo*, mainly differs by engaging all clients in every round, and therefore it is more appropriate for smaller settings (up to 100 devices) [22].

Security issues

Even though FL is a technology that brings benefits related to privacy, attacks on these systems are still possible [50, 40, 45, 47, 2, 55] due to the reliance on many devices that are usually dispersed over a wide area, in which control is limited. In fact, the study of security mechanisms in FL is still in a premature state, with novel attacks and defenses being developed at an increased rate. This indicates that it is admissible that a malicious actor could create an original attack compromising the behavior of the final model, and depending on the application, the consequences could be critical. For example, suppose an autonomous driving model is being developed in cars. An attacker could manipulate the model so that stop signs are seen as priority signs [47, 2], causing cars to advance immediately in a road interception.

For this reason, it is vital to understand how these systems can be attacked and how to protect them, equipping FL solutions with more robust and resilient mechanisms that can endure malicious actors which goal is to change the behavior of the deep learning model.

1.2 Tackling limitations of node attacks evaluation in FL

The contribution of this work is composed of two lines: the design and development of a tool that tackles limitations of current literature's implementations by offering a platform where users can evaluate security mechanisms with accessibility and flexibility; and a novel backdoor attack that aims to subsist in the global model for an increased number of rounds after the attacker stops contributing maliciously to the global model, when compared to previous backdoor attacks on the literature. These two lines of work are crossed eventually, as the implementation of the proposed backdoor attack is made on the proposed tool.

In the following sections, we present a brief overview of these contributions, where details about both of these are presented in Chapter 3 and Chapter 4.

1.2.1 FADO – Federated Learning Attack and Defense Orchestrator

This section aims to provide an introduction of the contribution of this work, when it comes to the a FL orchestration tool, which is named FADO (Federated Learning Attack and Defense Orchestrator) [39].

Challenges

When analyzing node attacks in current literature and their implementations, we are faced with complex and unmodular applications where extensibility is limited. Furthermore, our observation of current benchmarks of attacks and defenses in FL is that they are not realistic, as they do not leverage network environments – e.g., the training of multiple clients is performed on a single machine. Additionally, developing new attacks and defenses requires complex modifications to existing implementations – which represents itself a challenge to build new security mechanisms in a fast and flexible way.

Goals

We aim to address these limitations with our tool by providing interfaces that are accessible, enabling a flexible and agile configuration of a variety of FL scenarios. Those interfaces are designed to support the implementation of security mechanisms, such as model poisoning attacks or server-side defenses, for instance.

Furthermore, the tool is equipped with two execution modes (Network and Simulation) that can facilitate experiments with hundreds to thousands of clients without performance issues. It features a network execution mode which tackles the absence of network environments in current literature. This mode leverages a distributed environment in which different devices in the FL scenario communicate through an emulated network (hence supporting network attacks in FL [40]). Although a innovative feature in our tool, the network execution mode is not extensively discussed in this study, which primarily focuses on node attacks rather than network attacks.

In the following paragraphs, we shed some light on the most distinctive properties of FADO.

Realism

We are interested in making sure that the experiments conducted in this framework closely resemble real-world situations. The platform should mimic the complexities of FL processes, producing results that are similar to what we would observe in real FL scenarios. This can be achieved by isolating the different entities that participate in the FL process through virtualization or containerization, while using a virtual network to support the data exchanges. Although fine control over the network in FL security context is more useful to test network attacks, instrumentation of node attacks benefits from the isolation that is attained with the separation among the various elements – particularly, coordinated attacks executed by a strong adversary that controls multiple clients in the FL process. Moreover, experimenting coordinated attacks with client isolation and a network is more challenging – forcing security researchers to come up with new strategies to enforce collaboration among the nodes to maximize the attack success.

Comparable Results

The platform should make it easy for researchers to compare different attack and defense strategies. This comparability should work well across different system setups, including scenarios with various types of devices and data repositories. Researchers should be able to assess how effective different approaches are without much difficulty.

To achieve this property, FADO exports the results of the experiment in a file that packs standard metrics in machine learning contexts, such as model accuracy and loss. Moreover, other metrics can be defined by the user through a customizable interface, enabling flexibility to define new metrics in order to benchmark a variety of FL scenarios. Users can also customize the name of the file exported by FADO with the parameters of the experiment, enabling an agile and flexible differentiation between multiple experiments.

Scalability

Managing the platform's resources efficiently is crucial for emulating large-scale test scenarios. It should handle scenarios with many clients or other resource-intensive setups effectively, as some FL scenarios have hundreds to thousands of clients. Therefore, we argue that the tool should be scalable in terms of the number of clients. This property can be ensured by equipping the tool with the capabilities that enables it to execute in an environment that is able to scale out without performance issues – possibly inside a cluster. We are aware that scalability is a real challenge, as the possibility of scaling is always dependent on the hardware resources – nevertheless, the tool is designed to avoid bottlenecks and manage these different hardware resources in an effective manner.

In the current version of FADO, we tackle this issue by including two execution modes into the tool: the first leverages a network environment to support realistic network attacks (network mode), and the second aligns with the approaches found in current literature's implementations, as nodes do not communicate over a network (simulation mode). With these two execution modes,

users assessing node attacks can avoid the slightly more complex network environment setup, while those focusing on network attacks have the execution mode tailored to their objective. Moreover, it should be noted that: (i) switching between execution modes does not require effort from the user; (ii) in simulation mode, the user still benefits from all FADO capabilities, such as the ability to customize multiple parts of the FL scenario (e.g., dataset distribution, aggregation process, security mechanisms,...).

Ease-of-Use and Customizability

The tool is designed for a wide range of users, from beginners to experts. Beginners should find it simple to start experiments using default settings with minimal setup, using the already provided configuration files. As users become more experienced, they should have the option to explore the platform's finer details and customize its settings to suit their specific needs. This flexibility empowers users to experiment with various setups and adapt the platform to their changing research goals.

1.2.2 BLARE – a more durable backdoor attack

This contribution consists of an implementation of a novel backdoor attack on FADO, which we call BLARE. The goal of this new backdoor attack is to further increase the attack durability, which is defined as the amount of rounds the backdoor persists in the global model after the attacker stops injecting malicious updates into the FL process. To accomplish this, BLARE creates a history of the most modified model parameters by benign participants in the past, and crafts the malicious model update with the backdoor in a special manner in order to prevent those parameters from being modified. This ensures that the attacker's model update do not deviate heavily from benign clients model updates, resisting to norm clipping defenses [44, 42] and guaranteeing a backdoor injection with an increased attack durability. The implementation details referring to BLARE will be discussed later in Chapter 4.

1.3 Document Organization

In Chapter 2 (Context and Related Work), we will present the context required to demonstrate our contributions, as well as the relevant related work.

In Chapter 3 (FADO – Federated Attack and Defense Orchestrator), we will explain FADO's design and implementation details.

In Chapter 4 (BLARE) describes the novel backdoor attack we propose, providing intuitions on its design and its implementations details on FADO.

In Chapter 5 (Evaluation), we will present the evaluation methodology and a description of the experiments we carried out to demonstrate BLARE's effectiveness in different scenarios, as well as the utility of FADO.

Finally, in Chapter 6 (Conclusions and Future Work), we will disclose our primary conclusions and future work ideas to further improve the work in order to better understand node attacks on federated learning systems.

Chapter 2

Context and Related Work

2.1 Deep Learning

Deep learning is a sub-field of machine learning that tries to learn meaningful representations from data through a model that consists of a deep neural network (DNN) [10]. A DNN is formalized as a function $F_\theta : \mathbb{R}^N \rightarrow \mathbb{R}^M$ where θ are the function's parameters. Function F maps an input $x \in \mathbb{R}^N$ to an output $y \in \mathbb{R}^M$. It is made of L layers, where each layer $i \in [1, L]$ has nodes with parameters that define the computation performed to its inputs, which normally are the outputs of the preceding layer. The goal of the training process is to find the parameters such that the model behaves well for the task it was designed to. Typically, these parameters are found through an iterative process called *stochastic gradient descent* (SGD) [6], which operates as follows:

Consider a training data set $D = ((x_1, y_1), ..., (x_n, y_n))$ with $|D|$ training examples. Each pair $(x_i, y_i) \in D$ represents a data sample (x_i) and its corresponding label (y_i) [1], such that $0 < i \leq |D|$. Consider a loss function $L(y_a, y_b)$, which measures the distance between y_a and y_b. The objective of SGD is, for each $(x_i, y_i) \in D$, to minimize $L(F_\theta(x_i), y_i)$ by adjusting θ. The way θ is altered is based on the computation of the gradient of the loss with regard to the model parameters. Therefore, SGD updates θ in the direction of $-\nabla_\theta L(F_\theta(x_i), y_i)$.

2.2 Federated Learning

2.2.1 Motivation

Federated Learning (FL) [32] [23] is a distributed learning approach where models are trained across multiple devices (clients) without exchanging raw data. It addresses privacy concerns, as data never leaves the devices, allowing them to collaboratively contribute to a global deep learning model. Furthermore, this approach improves model generalization, as the decentralized nature of the data that is used implies more diverse examples. Communication costs are also reduced with FL, making it a valuable property in situations where privacy guarantees are not mandatory.

[1] Considering a classification task – for instance, x_i could be an image of an horse in a mountain, where y_i could be a label that indicates that the corresponding input is an horse

FL has a broad range of applications that could benefit from the properties it offers. Such applications are next word prediction models (e.g., Google GBoard) [52], autonomous driving [14], automatic speech recognition (e.g., Siri) [36] or healthcare [38]. Specifically, healthcare applications could benefit from the fact that FL allows training over sensitive patient data, which would allow for better generalization when detecting various types of diseases. Furthermore, when it comes to autonomous driving, FL allows multiple vehicles to collaboratively contribute to a global model that is more aware of different traffic situations, without spending too much resources (communication efficiency).

FL is based on a central server that coordinates several clients (data owners). The training happens in rounds – in each round, the server selects a subset of clients and requests them to train the current version of the model over their own data; when the training ends, clients send their model updates back to the server, which merges them all based on an aggregation algorithm; the result is a new version of the global model, which is pushed to the clients in the next round of training. The FL protocol will be detailed in Section 2.2.3.

The distributed nature of FL represents itself as a suitable environment for adversaries, which can conduct attacks in order to change the expected behavior of the system. For this same reason, there is also a motivation to develop aggregation algorithms (and other techniques) that can detect attacks and prevent them from having significant impact on the global model [37, 49, 34, 19, 44, 4, 8, 28].

As FL applicability rises, it is important to explore and evaluate attacks and defenses in this context in order to understand which approaches lead to solutions that are more resilient and robust. Thus, this work aims to build a tool that is able to simulate attacks where the adversary compromises the nodes (data owners). More specifically, the project will focus on model poisoning attacks, which according to the literature can be more impactful in FL deployments than traditional data poisoning attacks [41]. Section 2.3 will provide a categorization of the diverse types of attacks.

2.2.2 Architecture

The FL architecture is composed by a server and multiple clients (data owners) - as shown in Figure 2.1. Each client has a local model and a database that contains examples to perform a local training process using SGD. The server and the clients exchange messages in the context of the FL protocol (explained in detail in the next section) through the network. The data stored in each client does not leave the device, as it should be expected – such property contributes to an increased assurance of data privacy.

2.2.3 Protocol

We define a series of terminology that will be useful to describe the FL protocol: C is the set of all clients that are eligible to participate in the FL round; θ_g^t is the global model parameters in round t; in the first round, the θ_g^1 values are randomly assigned by the server; D_i is the training dataset

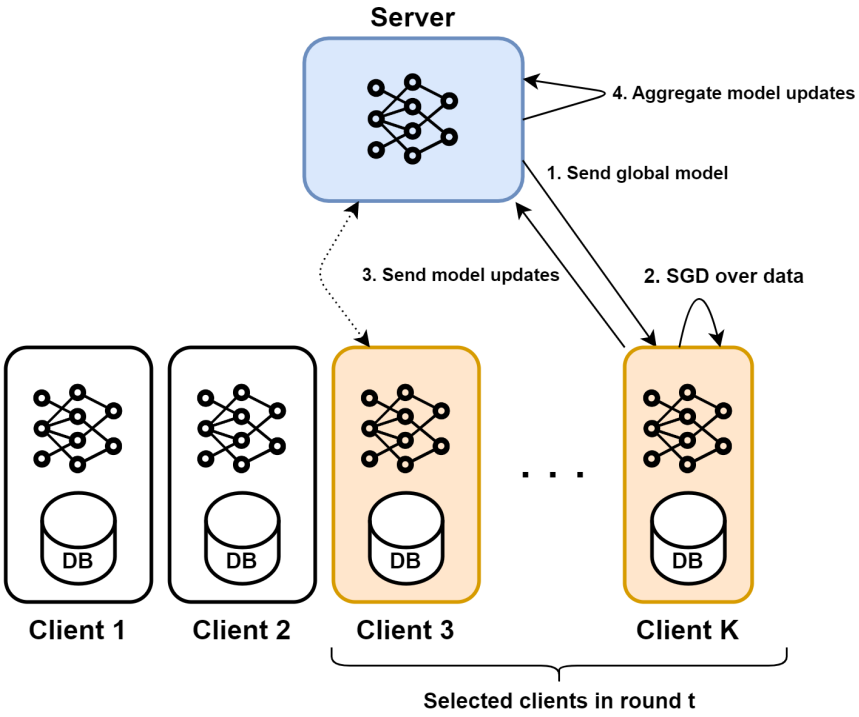

Figure 2.1: FL architecture representing up to K clients and an aggregation server. Each client locally stores a model used for the FL process as well as a database that contains training samples - such data is collected locally by the device through sensors or applications.

of client i, and D is the union of all client's datasets, i.e., $D = \bigcup_{i=1}^{|C|} D_i$; and T is the number of rounds the server will execute the FL process.

The FL process happens in rounds, whereas in each round $t \in \{1, ..., T\}$, the server S selects a set of clients $C_t \subset C$ and sends them the current global model θ_g^t (Figure 2.1, step 1). Upon receiving θ_g^t, each client $i \in C_t$ performs *stochastic gradient descent* (see Section 2.1) over their own data $D_i \subset D$ (Figure 2.1, step 2), obtaining their *stochastic gradients* which are used to update the local model parameters θ_i^t. The local model update is computed as $\Delta^t w_i \leftarrow \theta_i^t - \theta_g^t$, and the client i sends it to S (Figure 2.1, step 3). When the server S receives all clients local model updates, it aggregates them using a function f_{agg}, i.e., $\Delta_{agg}^t = f_{agg}(\Delta_{i \in C_t}^t w_i)$ (Figure 2.1, step 4). The new global model is built by applying the aggregated model update to the current global model under a learning rate η predefined by the server: $\theta_g^{t+1} \leftarrow \theta_g^t + \eta \Delta_{agg}^t$. The server increments the round counter and repeats the process, until $t = T$.

The most referenced implementation of the FL protocol is the *FedAvg* protocol, proposed by [32], in which f_{agg} consists of an average update of the clients gradients. Nevertheless, other implementations were described in the literature [30] – those implementations have different approaches when it comes to data partitioning, model implementations and communication architectures.

2.3 Threat Model

The attack surface in the FL context is composed by three components: the server, the nodes (clients) and the network. In this work, we will focus on attacks in the clients.

The threat modelling described next analyzes the nature of poisoning attacks under three dimensions: the attacker's objective, knowledge and capability. This framework was originally proposed by Shejwalkar et al. [42], however, we argue that the attacker's knowledge and attack mode have limitations that need to be tackled when describing the FL threat model. We aim to tackle these limitations in order to provide a full and realistic overview of the potential threats that FL faces.

2.3.1 Attacker's Objective

The attacker's objective can be broken down in three attributes:

Security violation: Security violations can be of two types in FL's context, *integrity* violations or *availability* violations. If an attacker wants to completely disrupt the service for all users (i.e., misclassify all inputs), it is considered an availability violation; if the attacker evades detection while causing a specific test input to be misclassified, it is categorized as an integrity violation.

Targeted attacks, for example, fit into integrity security violations – the goal is to attack specific inputs and evade detection without degrading the global model's accuracy [2, 44, 50, 47]. On the contrary, untargeted attacks are availability violations – the goal is to degrade the global model's accuracy and consequently compromise the service utility for users [41, 15].

Attack specificity: Attack specificity describes if an attack discriminates examples at the test time. If the specificity is *discriminate*, the attack is aimed to misclassify a specific set of examples; and is *indiscriminate* otherwise – all (or most) inputs are attacked.

For instance, considering an image classification task for pictures of cars, an example of a discriminate attack is a backdoor attack which only misclassifies pictures of cars with orange stripes. One could also translate this example into next word prediction tasks, where an attacker is interested in tricking the model into suggesting words that resemble hate when a specific ethnic group is mentioned in a phrase [55]. On the other hand, an indiscriminate attack will misclassify any picture of any car [42] or will suggest a specific word given any phrase.

Error specificity: The error specificity describes the nature of the misclassification that is to be performed by the attack. It can be either *specific* or *generic*: specific error specificity results in a given test input being classified to a specific class; a generic error specificity results in a given example being classified to any class.

The nature of the misclassification depends entirely on the attackers objective. If an attacker wants the model to classify all pink flowers into green flowers, it creates a specific error specificity; on the other hand, an attacker may be interested in misclassifying all pink flowers without

bothering with the result of the classification – thus the error specificity is generic. In both these cases, all pictures with pink flowers are attacked, but in different ways.

2.3.2 Attacker's Knowledge

The attacker's knowledge is divided into three domains: knowledge of the global model, knowledge of the data distribution of benign clients data and knowledge of the aggregator employed in the server to merge clients gradients.

Global model knowledge:

Under this model, the knowledge can be either: *whitebox*, when the adversary can access the global model parameters; or *blackbox*, when the adversary cannot access the global model parameters. An adversary with whitebox knowledge has way more capacity, as it can develop more sophisticated attacks that exploit particular properties of the model that is being trained, such as directly manipulating the model parameters to perform a certain attack [41, 15].

Data distribution:

The attacker has *full knowledge* of the data from the distribution of benign client's datasets when it has access to data (or is able to infer it) from non–compromised and compromised clients: the attacker knows D entirely. On the other hand, it has *partial knowledge* when it only has knowledge of the data from the distribution of compromised clients: the attacker knows D_k such that $D_k \subset D$. Full knowledge of the data is possible if the clients data is stored under particular scenarios – e.g., all the clients send their data to an external server that happens to be compromised by an attacker. However, given that partial knowledge is the most practical case in production FL, we will only consider it throughout this work, discarding full knowledge.

Aggregator:

Knowledge of the aggregation algorithm enables the development of more sophisticated attacks – the expected behaviour of this algorithm can be explored by the attacker to further increase the attack's success [41].

Consider an aggregation algorithm that rejects updates whose norms are higher than a static threshold value λ [44] – if the attacker has knowledge of f_{agg}, it can tailor its updates in such a way that they will be accepted by f_{agg}, i.e., guarantee that the norms of the updates are lower that λ.

Therefore, if the adversary does not know f_{agg}, it will be way more difficult to develop an attack which impacts the global model according to its objective.

2.3.3 Attacker's Capability

The attacker's capability is analyzed in terms of access to the different components in the FL architecture. However, we first need to define what are the mechanisms the attacker employs to

conduct an attack because these are strongly related with the ability to carry out certain actions.

Model poisoning attack

In a model poisoning attack, the attacker can modify the training process conducted in the client while participating in the FL protocol. This includes tampering with the training data at training time, changing hyperparameters or modifying the loss function. To get to a point where it can perform a model poisoning attack, the attacker has to compromise the client's device, which is very dependent on the security configurations of the underlying operating system and applications.

Data poisoning attack

In a data poisoning attack, the attacker is limited to changing the training data that is used by the clients. In practice, the attacker tampers the dataset by either adding manipulated examples or modifying existing ones to accomplish a certain goal [9, 45].

Machine learning defenses against data poisoning [43] are unpractical in FL configurations, as these defenses can imply access to clients data. Furthermore, data poisoning attacks are easier to perform compared to model poisoning attacks, as in the latter case the attacker needs to compromise the device – this results in an increased likelihood that less capable (and more common) attackers will attempt to attack the FL process.

Attack surface

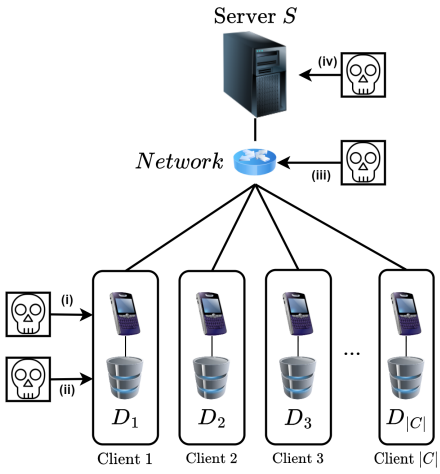

Figure 2.2: Attack surface of the FL architecture – the attacker can either compromise the client's device (i), it's data (ii), a network device in the middle of the communication (iii) and the server (iv)

We can also analyze the attacker's capability in terms of access to devices in the FL architecture (see Figure 2.2):

Device (i) If an attacker breaks into a client i (for example, by exploiting security vulnerabilities in the operating system), it is capable of performing both model or data poisoning attacks – as it access to the code that is executed to perform its part in the FL process as well as manipulating D_i. We argue that model poisoning attacks are way more dangerous than data poisoning attacks, as the first can result in changing the training process, e.g., altering the loss function L, hyperparameters, manipulating the gradients directly (θ_i^t) or introducing backdoors during the FL process. Therefore, compromising the device results in a more customized and flexible playground for an attacker to develop attacks.

Dataset (ii) When an attacker can only manipulate the local data D_i of client i, it is capable of performing data poisoning attacks. Moreover, there are some applications where the training dataset is built upon information contained outside the device (e.g., an external database) – for this reason, one could also consider the possibility of the attacker altering this information and thus being able to perform data poisoning attacks by manipulating the local data **indirectly**.

Server (iii) Although not considered in most of the literature, there is a possibility that the attacker has the ability of compromising the server, consequently poisoning the global model θ_g. Since the server sends θ_g to each client $i \in C_t$ on the beginning of each round, those clients will now work over a malicious model. Furthermore, the attacker can also interfere with the algorithm that selects the participants of each round, e.g., excluding clients that are more prone to improve the global model accuracy.

Network (iv) Network adversaries can also attack the FL process by observing, modifying or dropping the packets that are sent between the server and the clients. Typically, to conduct this type of attack, the adversary has to compromise a device that intermediates the communication of the FL process – for example, a router or an Wi-Fi access point.

2.3.4 Attack mode

The two capabilities in terms of the attack mode are: online, when the adversary can attack every round thus provoking significant impact over the entire FL training; and offline, when the adversary can attack only once at the beginning of the FL process, hence the impact of the attack may quickly disappear as rounds increase.

Typically, an attacker can only conduct an online attack if the following condition holds for every round: $C_t \subseteq C_m$ (C_m is the set of compromised clients) – such condition is unlikely, as the attacker would need to put a lot of efforts into compromising each client $i \in C_m$. Thus, we consider that an attacker can participate in some of the rounds, sitting between a grey zone attack mode between online and offline mode.

It should be noted that if the attacker is able to compromise the server and further modify the algorithm that selects the participants in each round, it can force that the previous condition is always true, therefore attacking every round.

2.4 Data characteristics

2.4.1 Distribution

Data to be used in a FL process typically is collected from applications stored in the clients devices. Depending on the nature of the data, its distribution over the clients can be either *Independent and Identically Distributed* (IID); or *Non Independent and Identically Distributed* (Non-IID). For example, if the data is based on different telemetry that is collected and stored periodically on the device, the distribution is likely to be IID; on the other hand, if the data consists of images of birds and different users have distinct tastes (for example, Mary may like to take a lot of photos of pigeons and John not), then we face an Non-IID distribution.

One of the most challenging aspects of FL is the data distribution nature, which in general is Non-IID [32] – considering an FL scenario with a great number of clients, most likely they do not collect and store the same kind and quantity of data. Therefore, the unbalanced and heterogeneity associated with the data distribution of the FL clients should be taken into account in the development of both attack or defense strategies [54].

The data distribution is important as it is appointed as one of the main challenges of FL [32]. Furthermore, in traditional machine learning, the IID property is leveraged to tackle malicious (byzantine) clients who contribute with malign updates [4], which normally is not applicable in FL.

Hence, to perform experimental evaluations under realistic assumptions, the datasets used should have representatives of both IID and Non-IID data distributions. LEAF [7] is a framework to generate federated datasets, with the possibility of them being IID or Non-IID. Data heterogeneity plays a role in FL security [54] – although high data heterogeneity correlates with a decrease of attack success rate, it also enables attackers to inject backdoors in a stealthy way, evading detection.

2.4.2 Partitioning

There is a categorization to define the partitioning of the data across different FL architectures. Data in FL can be distributed over the sample and feature spaces – in the literature, there are three possible approaches: horizontal FL, vertical FL and hybrid FL [30]. In this work, we will only describe horizontal and vertical FL, as they are the most common in the literature.

Horizontal FL:

In an horizontal FL scenario, it is expected that the clients share the same feature space but differ in the sample space. As an example, consider a FL scenario where several hospitals collaborate with the goal of training a global model. Hospitals can infer if a given radiology image has a potential health issue, without sharing their data. Each hospital i has a training dataset with examples $D_i = \{(x_0, y_0), ...(x_n, y_n)\}$ where each example consists of an image and a label associated with a (possible) health issue – the images have the same feature space (the images have similar

characteristics), label space (the possible health issues are the same), but differ on the sample space (different hospitals have distinct patients).

Formally, considering that the two clients (i, j) have the feature space (X_i, X_j), label space (Y_i, Y_j), datasets (D_i, D_j) and sample space (I_i, I_j), horizontal FL is described as follows:

$$X_i = X_j, Y_i = Y_j, I_i \neq I_j, \forall D_i, D_j, i \neq j$$

Vertical FL:

Vertical FL [51] scenarios assume that the clients that participate in the process share the same sample space, but not the feature space. For example, multiple hospitals that participate in a vertical FL scenario share different types of information related to the same patients (e.g., hospital A stores information about blood analysis of a patient P and hospital B stores radiology images of the same patient P). In the vertical FL framework proposed by Liu et. al [31], one of the clients is considered the FL manager, which is the only participant that owns the labels while coordinating the FL protocol.

Formally, considering that the two clients (i, j) have the feature space (X_i, X_j), label space (Y_i, Y_j), datasets (D_i, D_j) and sample space (I_i, I_j), vertical FL is described as follows:

$$X_i \neq X_j, Y_i \neq Y_j, I_i = I_j, \forall D_i, D_j, i \neq j$$

2.5 Attacks in FL

In this work, we will give particular focus to two categories of poisoning attacks: targeted attacks and untargeted attacks. Attacks described in the literature generally fit into one of these categories. However, there are also contributions to provide a general framework and techniques that can be both useful for targeted and untargeted attack objectives. This is the case of the work made by Baruch et al. [3], which introduced an attack paradigm that has the capability of either preventing convergence (untargeted attack) or backdoor distributed learning (targeted attack) in a stealthy way – the authors study a method that applies small modifications to the model parameters under a perturbation range, consequently evading defenses.

In the next sections, we will explain both targeted and untargeted attacks, identifying the most relevant works in each of the categories.

2.5.1 Targeted attacks

Targeted attacks aim at attacking specific sub-tasks of the model without degrading the global model's accuracy. For this reason, they are harder to detect and thus more dangerous for a FL system. For example, if we consider a model that is classifying pictures of birds based on their color, a sub-task could be the classification of yellow birds. Consequently, an attacker could be interested in attacking the resulting classification of yellow birds, while not changing the model's

behavior for every other bird that is not yellow. We further divide targeted attacks into three groups:

Backdoor attacks: Backdoor attacks in FL aim to control how a model performs on a certain chosen backdoor sub-task, hence, it is categorized as a targeted attack. Backdoors were already studied in traditional machine learning before the insurgence of FL. However, the properties of FL make backdoors easier to inject. Bagdasaryan et al. proposed a backdoor attack that introduces the concept of model replacement, in which the attacker attempts to replace the global model with a malicious model specially crafted [2]. Moreover, this attack incorporates an evasion technique called *constrain-and-scale* – in which the loss function is modified, so that it penalizes the model if it deviates from benign behavior.

These attacks work because the model is trained on a poisoned dataset, where examples have certain features that map into an attack-chosen label. These features can be pixel-patterns inserted by the attacker or simply semantic features such as green cars or yellow trucks. Semantic backdoors are often considered more dangerous, as the backdoor triggers are natural features – for instance, if an attacker successfully injects a semantic backdoor into an FL model based on purple cars, future model inferences of inputs containing purple cars are likely to be classified as the attacker-chosen label.

The success of backdoor attacks is dependent on a number of factors: (i) a higher fraction of compromised clients will result in a more successful backdoor attack [44]; (ii) an attack injected in later rounds of the FL process is likely to persist, as in early rounds the global model will represent common features from clients and not individual features (such as a backdoor) [50, 2]; (iii) the parameters from the model update after a backdoor is inserted – Zhang et al. proposed a new attack method that improves the backdoor durability by only updating the parameters that are less used by benign model updates [55]; and (iv) the features that are used by the backdoor – Wang et al. showed that backdoors based on rare features, i.e., data present in the tails of the data distribution, are stealthier than backdoors based on very common features across the feature space [47].

Label-flipping attacks: A label-flipping attack tampers with the training examples by changing the labels associated with data samples. The first label-flipping attack in FL was proposed as a data poisoning attack [45]. Here, the attacker modifies D_k (compromises datasets) such that, for all examples with the label l_{src}, change it to l_{target} (i.e., $l_{src} \rightarrow l_{target}$) – which results in the global model misclassifying examples associated with the specific classes. The nature of the label-flipping can be either targeted or untargeted: an attacker which randomly changes the labels in all examples of the training dataset attempts to compromise the availability of the model (untargeted attack); and an attacker which carefully changes the label of specific examples most likely is performing a targeted attack. For instance, if in a image classification task build to infer pictures of fruits and classify them accordingly, an attacker could perform the label alteration "$banana \rightarrow apple$" in all examples of the compromised dataset – the result is a model that

theoretically will categorize bananas as apples.

The severe consequences of this attack can be better understood if we consider a critical scenario in which autonomous vehicles need to perform image classification of road signs to adjust their current speed. For instance, a label-flipping attack in this scenario could attack examples containing maximum speed road signs and change their label to a higher value, e.g., "$30 \rightarrow 120$ (kph/mph)".

Solutions have been proposed in order to mitigate label-flipping attacks [17, 21] – however, these works assume attackers just tamper with the dataset used for the training process, not taking into account the possibility of attackers employing label-flipping in model poisoning scenarios. Attackers who are able to perform model poisoning scenarios have a more sophisticated capability to avoid defenses, as they are in total control of the training process – for instance, attackers who employ the *constrain-and-scale* technique proposed by Bagdasaryan et al. [2] can prevent malign gradients from deviating too much from the benign ones, thus avoiding *anomaly detection* mechanisms that use clustering to separate malign from benign updates.

Distributed backdoor attacks: Although the above backdoor attacks can be dangerous, there are approaches able to backdoor a FL model in a more stealthy and effective way. These approaches aim to decentralize the process of backdooring a FL model, by decomposing the backdoor trigger (in this case, we consider pixel-pattern backdoors) across several compromised clients – a distributed backdoor attack [50]. This attack is proven to be more persistent and effective than centralized backdoors [2, 44].

Moreover, state of art attacks should have this distributed advantage in consideration, as it is based on a realistic threat model and its powers seem to be superior than current FL backdoors.

Neurotoxin: One challenge that comes with trying to inject a backdoor into a FL model is that the attacker has to perform model poisoning every round in order to make sure that the backdoor remains in the model.

Neurotoxin [55] is a model poisoning attack proposed by Zhang et al., which is designed to inject a backdoor into a FL model by only updating model parameters that are not frequently updated by the rest of the benign users. By making sure that the model poisoning only updates coordinates that benign clients do not frequently update, the attacker can expect the backdoor to persist in the global model for a longer number of rounds, after it stops attacking. The algorithm of this attack works as follows:

1. When selected by the server to participate in the FL round, the attacker expects to receive the global model g from the server, which is used to update its local model, according to the FL protocol;

2. Next, it uses g to calculate an approximation S of the aggregated benign model update, for the next round, using a benign dataset;

3. S is used to further infer what are the model parameters that will probably be more affected by the benign contributions, this inference will result in a mask M;

4. During the training, the attacker prevents the model parameters included in M from being modified on the local model update (by setting those parameters to zero), which is generated by the SGD over the malicious dataset;

This approach ensures that, regardless of what the values contained in the model parameters from the malicious update, the most updated parameters from benign contributions are not heavily changed. Therefore, it increases the persistence of the backdoor, as after the attacker stops injecting the backdoor, future benign contributions will not alter significantly parameters that were used by the model poisoning attack to backdoor the model.

This attack will be further discussed in Section 4, when our novel model poisoning attack, BLARE, will be introduced and explained.

2.5.2 Untargeted attacks

Untargeted attacks have the goal of degrading the performance of the global model on the main task. In practice, attackers want the global model's accuracy to have a lower value and/or prevent the model from converging. Following the example that was presented in the targeted attacks section (Section 2.5.1), an untargeted attack would result in an increased likelihood of the model misclassifying the birds pictures regardless of their color.

Untargeted attacks are less stealthy as malign updates can be significantly different from the benign updates – because attackers sometimes just send random parameters to the server or modify data in a random way (as referred to in the label-flipping attack previously). Nevertheless, attackers who have knowledge of the server aggregation rule can further develop attacks that can circumvent existing defenses, tailoring their updates carefully to accomplish that objective [15]. Sometimes, however, the attack does not need to know the server aggregation rule because it adds a small amount of noise to benign parameters in order to corrupt the model's performance in a stealthy manner [3].

Shejwalkar et al. presented a framework for untargeted attacks that covers a range of important aspects – from the types of perturbation vectors to the different scenarios related to the knowledge of the adversary with respect to the server aggregator [41]. In addition to that, the authors also point out several limitations of the *Fang* attack [15] and address them by developing a novel attack which is able to defeat all robust aggregators developed up to that time.

2.6 Defenses in FL

Defenses have been developed to detect attacks and mitigate their effects on a FL system. Although literature has tried to categorize defenses accordingly, such goal is difficult to attain as state-of-art defenses frequently combine multiple mechanisms and the same defense can be effective against

both untargeted or targeted attacks. Thus, we will discuss these mechanisms, categorizing the defenses under their most interesting properties.

2.6.1 Robust aggregators

The *FedAvg* aggregation algorithm uses the mean as the statistical operator to aggregate client's gradients – which is susceptible to outliers. To tackle this limitation, robust aggregation algorithms that consider other statistical operators were proposed: *median*, which replaces the mean operator in the aggregation algorithm by the median, choosing the value in the center of the distribution; *trimmed-mean*, which filters extreme values below and above the data distribution by k-% and then performs the mean of remaining values [53]; and *geometric-median* [37], which computes a value that represents the central tendency of the product of the all model updates.

Krum [4] is proposed under the intuition that benign gradients are close to each other, whereas malign gradients will be far from them. Therefore, *Krum* consists of a function that, given a set of gradients from n clients, returns the gradient which is least distant from their neighbours and uses it as the global update Δ_{agg}^{t} (see Section 2.2.3). *Multi-Krum* executes *Krum* i times to build a set that does not contain distant (and hopefully, malign) gradients. *Krum* was the first solution to provide Byzantine tolerance in distributed learning, but it has a few critical limitations: it assumes that the data distribution across nodes is IID (this limitation may result in *Krum* rejecting benign gradients) and requires prior knowledge of the number of sybils in a FL system – both these assumptions are not suited for a FL scenario. We still argue that this defense has to be included in our tool, as *Krum* provides a first level of defense in distributed learning.

2.6.2 Norm clipping

Sun et al. proposed a norm clipping approach to tackle backdoor attacks based on the fact that attackers may increase the norm of their updates in order to cause more impact [44]. This approach ignores client updates which are above some pre-defined threshold value – which introduces two limitations: a strong attacker can know the threshold and thus adjust the norm of the updates accordingly to evade the norm clipping defense; the solution assumes a fixed threshold value, which can be exploited by an adversary to conduct distributed backdoor attacks [50]. Furthermore, Guo et al. proposed a dynamic norm clipping approach which avoids fixed thresholds and thus can defend against distributed backdoor attacks successfully [19].

2.6.3 Differential Privacy

Differential privacy is a mechanism that limits information disclosure about individuals given some database – it guarantees that no third party can infer if any individual's data is present in a database given some statistical analysis of that same database. For example, differential privacy was employed by U.S. Census Bureau to prevent malicious parties from tracing information back to the citizens through re-identification methods [18].

In the context of FL, we are interested in a type of differential privacy introduced by [13] which is based on two parameters (ε, δ) that quantify the amount of privacy to be applied. Formally, a random algorithm $A : \mathcal{D} \rightarrow \mathcal{R}$ ($\mathcal{D}$ is the domain and $\mathcal{R}$ is the range) is (ε, δ)-*differentially private* if for all adjacent datasets $d, d' \in \mathcal{D}$ and $S \in \mathcal{R}$, the following equation is valid:

$$Pr(A(d) \in S) \leq e^{\varepsilon} Pr(A(d') \in S) + \delta \tag{2.1}$$

d and d' correspond to adjacent decentralized datasets, and d' can be generated by a slightly modification (addition or removal) of the records of a single client in d.

A typical approach to enforce differential privacy in FL is to add random Gaussian noise $\mathcal{N}(0, \sigma^2)$ into model updates [1], which results in a reduction of the influence of specific data points. Such mechanism was already studied in several works to provide robust aggregators with better mechanisms to defend against backdoors attacks [49, 34, 44], as well as in clients to avoid attacks whose objective is to infer information from the observed model updates [48].

Sophisticated backdoors are unlikely to be detected by a defense that uses statistical operators – this happens because model replacement techniques [2] are crafted to not deviate largely from benign behavior. Furthermore, many aggregation algorithms in the literature make unrealistic assumptions about the adversary in FL scenarios. For this reason, alternative solutions based on differential privacy (see Section 2.6.3) were developed, which in fact can eliminate backdoors – the main challenge behind DP is that it can deteriorate the performance of the model, as the noise added has to be large (enough) to eliminate malign behavior.

FLAME[34] was proposed as a defense framework against backdoor attacks – it is applicable to generic scenarios and demonstrates that the amount of required noise can be reduced by first applying clustering (based on the cosine distance between the updates), followed by norm clipping of the weights and then adding a small amount of noise. However, *FLAME* has a complex implementation which was not made public. For this reason, it was not possible to include *FLAME* in our tool.

2.6.4 Adjusting the learning rate

FoolsGold [17] was proposed as a novel defense that adapts the learning rate of clients based on the similarity of their updates. The intuition behind *FoolsGold* is that in a setting with no sybils, model updates are slightly different (e.g., they have different directions) – thus, malicious clients can be detected as their model updates are expected to be similar. This defense was designed to address a specific type of targeted attack: a label-flipping attack that aims to corrupt examples with specific features.

Ozdayi et al. proposed *Robust Learning Rate (RLR)*, which adjusts the global learning rate (used by the server to update the global model) accordingly to whether the clients are updating the model towards the same direction. If not, the learning rate is modified to defend against a possible attack [35].

2.6.5 Spectral methods

Divide-and-conquer (*DnC*) [41] uses singular value decomposition (SVD) spectral methods to detect model updates that are outliers and remove them before the aggregation process. As SVD-based defenses require a lot of memory and computation, *DnC* performs dimension reduction through random sampling. The downside of such an approach is that information can get lost if the model updates consist of many parameters. Although resilient, *DnC* was unsuccessful at defending against adversaries in real world Non-IID FL settings.

2.6.6 Trust Bootstrapping

Typical defenses were based on analyzing client updates or by employing robust aggregation operators. A new paradigm was proposed which consists of bootstrapping trust into the FL process [8]. The server (called *service provider*) maintains a validation dataset which is used to assign trust scores to the local model updates received from the clients. To assign these scores, in each round, the server trains over its validation data to obtain a global update and measures the cosine similarity between the global update and the clients updates. Client updates that have a negative score will be rejected and the accepted updates are normalized according to the norm of the global update. Finally, the client updates are aggregated.

Chapter 3

FADO – Federated Attack and Defense Orchestrator

In this chapter, we present FADO, a novel FL orchestration tool that simplifies the evaluation of security methods while reducing complexity. FADO is created to tackle the fact that current literature's implementations have several accessibility flaws, making their extensibility a difficult task.

In the first part of the chapter, we examine FADO's architecture and components, which are crucial for understanding the different elements of the tool and their roles. Upon examination of those elements, we highlight how these parts lead to more realistic and accessible attack evaluation on FL nodes. Next, we dive into the alternative execution mode, which was created to tackle the startup performance impact of FADO under a network setup, which is not critical in scenarios where users are only keen to evaluate node attacks. Furthermore, we provide an in-depth inspection of each component of the tool, showing examples on how they enable quick and easy FL orchestration while also presenting the code that enables FADO to operate with such a high degree of flexibility.

We finish the chapter by delving into dataset integration, providing insight into some of the most used datasets in the FL literature and what datasets FADO supports out-of-the-box.

3.1 Architecture and Components

The architecture and components of FADO are depicted in Figure 3.1. FADO's architecture is divided into two modules, the *Builder* and the *Runner*, which will be described in the following sections.

3.1.1 Builder

The primary objective of the *Builder* module is to generate the datasets that will be utilized in FL experiments. Specifically, it undertakes the task of partitioning data examples across various client nodes, enabling them to train their respective local models. It also distributes the validation and test data to the central server in order to facilitate the evaluation of the global (aggregated) model.

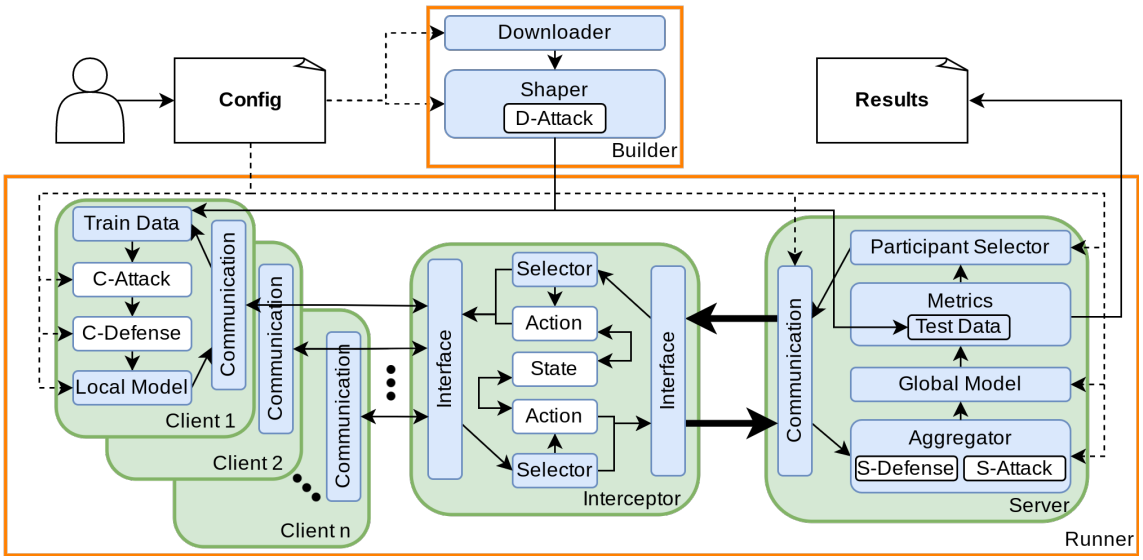

Figure 3.1: FADO Architecture

The execution of this task is carried out by two components, entailing a two-step process that is described next:

Downloader (Data Acquisition): This component is responsible for performing data acquisition, retrieving data from either a web-based repository or from a local file source. It is assumed that the output data of this component is not yet suitable for being used in FADO because it is in its raw form.

Shaper (Data Preprocessing): The succeeding step transforms the acquired data into a format suitable for training and testing purposes. This phase encompasses a set of operations taking into account configuration parameters such as the number of clients of the FL experiment, the target class being evaluated (suitable for label-flipping attacks, for instance), the data distribution, or any attack strategies intended for experimentation.

The *Builder* module offers a user-friendly interface that abstracts the underlying complexity, granting a high degree of flexibility. This flexibility enables users to seamlessly integrate their own datasets and attack experiments. In practical terms, the two aforementioned steps are entirely customizable, empowering users to implement (as per their research requirements) custom *downloaders* for alternate dataset sources or custom *shapers* with the goal of modifying properties, such as data distribution characteristics of existing datasets across distinct clients.

3.1.2 Runner

The Runner is responsible for training the model according to the traditional FL protocol and is composed, in its default execution mode (see Section 3.2), of three components:

Server: The primary responsibility is to orchestrate the FL scenario throughout the execution of the protocol. Initially, it selects the clients that will participate in the next round of training through the *ParticipantSelector* interface, and sends the initial global model to these clients. The server then awaits for the arrival of local model updates from the clients. These updates are aggregated to form a new iteration of the global model, using the *Aggregator* interface. Finally, the model undergoes evaluation against a test dataset, and the results are recorded in a file. This sequence of steps is iterated for a specified number of rounds, which is indicated in the configuration file, until the final global model is produced.

Some of the steps described above are entirely customizable by the user, without much effort, through a configuration file that is ingested by FADO. This configuration file will be reviewed more in depth in Section 3.1.3.

To give some examples on how this component can be customized, one could modify the selection of clients in each round through the *ParticipantSelector* interface. Moreover, the aggregation procedure can also be personalized through the *Aggregator* interface. Thus, the customization versatility inherent to these interfaces facilitates the implementation of diverse enhanced security mechanisms (*S-Defense* or *S-Attack*). With the goal of simplifying usage, FADO is already equipped with an example of a *ParticipantSelector*, which chooses a subset of clients, and an *Aggregator*, which executes the traditional FL aggregation strategy ***FedAvg***.

In Section 3.3, further explanation will be supplied on how to exploit these interfaces for a variety of purposes.

Clients: They function as active participants that are tasked with receiving a global model from the server and train it using the local data, resulting in the creation of a unique local model update.

This step can be further adjusted to implement specific actions, such as an client attack (*C-Attack*) or a client defense (*C-Defense*), with the goal of maliciously altering or protecting the local model update. Following this customization, the modified local model is then sent back to the server.

Much like the server, the behavior of each client is subject to customization through user-written configuration files, offering a level of adaptability and flexibility tailored to the specifics of their individual operations.

Interceptor: This component acts as a middle entity that forwards messages between the server and the clients, being capable of intercepting them to further develop network attacks, as it has direct access to the packets that are being communicated. The Interceptor plays a crucial role in facilitating realistic network attack experiments, as it simulates the occurrence of an attacker compromising a network device situated between clients and the server within FL scenarios.

However, when implementing node attacks, this entity is less relevant, as we are only interested in modifying the learning process or the data on nodes. Additionally, the deployment of the FL job with the Interceptor incurs a substantial performance cost due to the intricate Docker and network setup required to guarantee the realism of the network attacker's role. In order to address this

challenge, this work contributes to FADO by proposing an alternative execution mode, which we call *simulation mode* and explain in Section 3.2.

3.1.3 Configuration and Results

The configuration file is the interface through which FADO users define the FL experiment/scenario parameters, such as the number of clients, dataset, security mechanisms (attacks or defenses), or the percentage/number of malicious clients. As such, this file plays a central role in the coordination of the Builder, the Runner, and all associated components within both modules.

File structure and important options

The configuration file structure is versatile, allowing users to specify custom options (or parameters). However, it must contain certain baseline options so that FADO can orchestrate the FL job without problems.

Figure 3.2 shows a working example of a FADO configuration file:

```yaml
general:
    random_seed: 1                    # RNG seed
    use_gpu: true                     # Or "false"

fl_training_process:
    rounds: 1850                      # Number of rounds
    aggregator: "diff"                # Selected Aggregator (diff = FedAvg)
    agg_learning_rate: 1.0            # Aggregation learning rate
    number_clients: 200               # Number of clients
    clients_per_round: 10             # Clients per round
    participant_selector: "random"    # Selected ParticipantSelector

client_training_process:
    model: "resnet"                   # Machine learning model implementation
    batch_size: 64                    # Batch size of the training
    epochs: 2                         # Number of times the client trains
    learning_rate: 0.001              # Client learning rate
    momentum: 0.9                     # Optimizer momentum
    weight_decay: 0.0005              # Optimizer weight_decay

dataset_spec:
    dataset: "cifar10"                # Dataset - maps to a Shaper object

output:
    logs_file_name: "..."             # Filename of the logs
    results_file_name: "..."          # Filename of the results
```

Figure 3.2: Example of a baseline configuration file, containing the mandatory options (in YAML).

The configuration file is composed of several key-value pairs. The pairs are organized into various categories, namely: `client_training_process`, `fl_training_process`, `general`, `output`, and `dataset_spec`.. However, this structure is not rigid, since FADO strips out each category at the beginning of the processing of the configuration file, with only the second level

key-value pairs remaining. The motivation behind this structure is to encourage users to organize their configuration parameters, as the file becomes significantly more readable.

The example contains a minimal set of configuration options to ensure that FADO runs smoothly, in case the user plans to create a configuration file from scratch. However, there are other configuration options that enable the injection of personalized attacks, defenses, or model checkpoints. These options will be explained further in Section 3.3.

Furthermore, the user is able to specify new configuration options in the configuration file, with these being accessible on all components, including the customizable implementations of the different interfaces that FADO offers.

Multi-experiment list options

Suppose that a user's goal is to run multiple experiments to evaluate the impact of the clients learning rate, in the context of a certain machine learning task. To accomplish this, the user could create multiple FADO configuration files, where the learning rate is different in each one of the files. We argue that there is no advantage in creating multiple files if the difference between each is very small.

To address this, FADO parses the options from the file, enabling users to create a unified configuration that can cover multiple distinct experiments, each with its own set of options. This mechanism is activated when the user specifies the option as a list containing the different values that he/she wants to use in different experiments. For instance, the user could employ a single file to initiate five separate FL jobs, each using a distinct learning rate, as demonstrated in Figure 3.3:

```
client_training_process:
    ...
    learning_rate: [0.001, 0.002, 0.003, 0.004, 0.1]
    ...
```

Figure 3.3: Example snippet of a multi-experiment list configuration declaration.

Rapid experimentation of security mechanisms

In the context of FL security experimentation, the feature described previously (multi-experiment list options) is considerably valuable, as it enables users to evaluate the attack impact of specific parameters, such as the number of epochs a malicious client trains, using a single configuration file. The same argument applies to the implementation and evaluation of a defense mechanism, such as norm clipping, if the user wants to evaluate different norm clipping values and its effects on the effectiveness of the attack.

Moreover, the fact that the configuration file is not rigid and is able to include additional options makes the experimentation of node security mechanisms more versatile. With this property, users can use whatever configuration parameters they want and access them in their custom implementations of FADO interfaces.

Results

The emulation outputs the results that the user requests, such as traditional metrics like the accuracy of the model on the final test dataset, loss values during the learning process, and the accuracy of specific classes when exposed to malicious examples. In FADO, when the user uses multi-experiment list options and specifies a `results_file_name` configuration option with formatting parameters, those values are injected into the filename. This feature facilitates the post-processing of the files, particularly when extracting metrics from different experiments that were executed through multi-experiment list options.

To explain this mechanism with a practical example, suppose that we have a configuration file such as the one shown in Figure 3.4. This configuration file is based on the one illustrated in Figure 3.2, however, the `results_file_name` configuration option has a special string structure, which allows FADO to apply string formatting with configuration options.

```
fl_training_process:
    rounds: 1000
    number_clients: 100
    clients_per_round: 5
...
client_training_process:
    ...
    learning_rate: [0.001, 0.002, 0.003, 0.004, 0.1]
    ...
output:
    logs_file_name: "experiment1_logs"
    results_file_name: "experiment1_rounds_{rounds}_num_clients_{number_clients
    }_client_lr_{learning_rate}"
```

Figure 3.4: Configuration file with string formatting on `results_file_name`, in order to inject parameters into the name of the file that will store the results.

This configuration file will result in five different FADO executions, each with a different client learning rate (`0.001, 0.002, 0.003, 0.004, 0.1`). Moreover, it will also create five different files that will contain the experiment's results, specifically:

- `experiment1_rounds_1000_num_clients_100_client_lr_0.001.npy`
- `experiment1_rounds_1000_num_clients_100_client_lr_0.002.npy`
- `experiment1_rounds_1000_num_clients_100_client_lr_0.003.npy`
- `experiment1_rounds_1000_num_clients_100_client_lr_0.004.npy`
- `experiment1_rounds_1000_num_clients_100_client_lr_0.004.npy`

The results are packed in a *NumPy* binary file (`.npy`), which has a hash map where the key is the name of the metric and the values are lists of numbers.

By default, every experiment will create a file with at least four keys:

- `per_round_model_accuracy` – Global model accuracy

- `per_round_model_loss` – Global model loss
- `per_round_target_accuracy`[1] – Model accuracy on a target class.
- `per_round_target_loss`[1] – Model loss on a target class

In the custom implementations, users can also add new metrics to the results file using the `Results()` singleton, which uses the method `add_round(key, value)` to manipulate the resulting hash map according to the particular needs.

Suppose a user wants to add a new metric called `l2_norm`, to register the L2-norm of a client model update in every round. Figure 3.5 illustrates how the user could use the `Results` interface to achieve this goal.

```
from fado.runner.output.results import Results
...
l2_norm_value = calculate_l2_norm(...)
results = Results()
results.add_round('l2_norm', l2_norm_value)
...
```

Figure 3.5: Example of how the user could use the *Results* interface to export additional metrics to the output file (`results_file_name`).

3.2 Alternative execution mode: *simulation mode*

When evaluating scenarios with a particular focus on node attacks, we found that FADO had a series unnecessary overheads in the initial phase. Those delays appear when FADO interacts heavily with Docker to create a stable environment. For example, it takes some time to build and prepare the Docker images. We argue that it would be beneficial if node attack experimentation would not suffer from this behavior, as the execution of these operations is due primarily to the emulation of a network to support realistic network attack evaluations.

Therefore, we decided to implement an alternative execution mode into FADO, the *simulation mode*. The main difference between the default execution mode and the *simulation mode* is the elimination of the interceptor throughout the entire FL job, that is, the network part is completely absent. The design of this execution mode guarantees that a user still benefits from all the remaining features of FADO, namely customizable interfaces and accessible configuration.

One motivation for using the *default mode* instead of the *simulation mode* is the need to evaluate scenarios that require an entity between the clients and the server, which in FADO is represented by the *Interceptor* component. Examples of such scenarios include network attacks where a malicious actor compromises devices supporting the communication between FL nodes, such as routers or switches. The presence of the *Interceptor* naturally demands more resources and results in lower performance due to the layer of abstraction that emulates the FL nodes and the network

[1] Only applicable in scenarios where a target class is being evaluated.

through Docker. On the other hand, use cases that only involve evaluating A&D (Attack and Defense) mechanisms in FL nodes (clients and server), without the need for an intermediary entity or network, could be performed in *simulation mode*.

Simulation mode closely aligns with current literature implementations, featuring a single thread that sequentially executes all the steps of the FL protocol. In contrast to the default execution mode, where the multiple clients and the server operate within distinct execution contexts, in *simulation mode* these are running in the same execution thread and do not rely on the exchange of messages transmitted through a network. Moreover, the tool is still realistic in *simulation mode*, as isolation between entities is guaranteed through encapsulation, since each client is represented as a different object which cannot access properties from other objects (e.g., clients cannot access variables from other clients or the server). Simultaneously, it offers convenient interfaces for researchers aiming to assess node attacks without extensive modifications to existing implementations.

In Figure 3.6, we present the architecture and components of FADO in *simulation mode*:

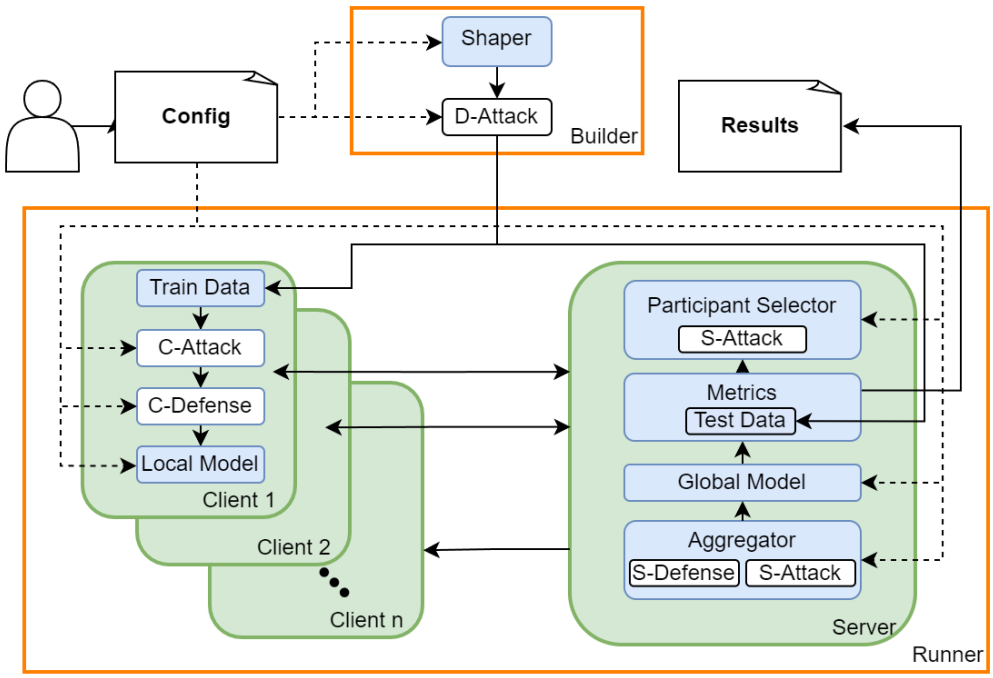

Figure 3.6: FADO Architecture in *simulation mode*.

An additional difference in the *simulation mode* architecture is the absence of the *Downloader* component. We claim that the *Downloader* component adds unnecessary complexity to FADO, as the *Shaper* is able to inherently carry out its task. Thus, we decided to exclude the *Downloader* from the Builder module in the simulation mode.

3.3 Node attacks implementation

In this section, we describe each of FADO's customizable interfaces with the goal of achieving a flexible and accessible experimentation of node attacks, which is one of the main contributions of this work.

3.3.1 FADOModule

This component represents the core foundation of ML within FADO, by providing a level of abstraction that enables users to customize the model used in the FL scenario, which is expected to be contained within the corresponding subclass. Abstract methods for copying and setting model parameters (`get_parameters` and `set_parameters`) ensure a standardized and customizable approach for handling model weights attribution. Moreover, the `train` method enables FL training by receiving a DataLoader that represents the local training dataset at each client, allowing developers to implement training logic specific to their model. Similarly, the `evaluate` method enables model evaluation on local datasets.

This abstraction not only improves the implementation process but also ensures that the core functionalities related to parameter handling, training, and evaluation are adaptable to diverse FL scenarios, promoting flexibility and ease of use.

```
class FADOModule(object):

    @abstractmethod
    def get_parameters(self):
        pass

    @abstractmethod
    def set_parameters(self, new_weights):
        pass

    @abstractmethod
    def train(self, dataset: DataLoader, current_round=None):
        pass

    @abstractmethod
    def evaluate(self, dataset):
        pass
```

Figure 3.7: Implementation of the `FADOModule` abstract class.

3.3.2 ModelAttack

The *ModelAttack* class is the interface that enables users to integrate new model attack strategies into the tool. This integration requires the user to define a subclass of the *ModelAttack* abstract class, the implementation of which is shown in Figure 3.8. In this subclass, the user is expected to override the `attack_training_process` method and specify a special option in the configuration file (`model_attack_name`).

When implementing a new strategy through the `attack_training_process` method, the user has access to the `FADOModule` of the compromised client, its dataset and the current round number. This approach enforces a consistent and straightforward interface for integrating custom model attack strategies into FADO with flexibility and accessibility.

```
class ModelAttack(Attack):

    def attack_training_process(self, local_model: FADOModule, dataset,
    current_round):
        return local_model.train(dataset, current_round)
```

Figure 3.8: Implementation of the `ModelAttack` abstract class.

Figure 3.9 illustrates an example code implementation of a model attack, while Figure 3.10 shows the corresponding configuration options with regard to the attack parameters.

```
# custom_model_attack.py

def get_class():
    return CustomModelAttack

class CustomModelAttack(ModelAttack):

    def attack_training_process(self, local_model: FADOModule, dataset,
    current_round):
        model = local_model.model

        # Singleton which gets all the configuration options
        fado_args = FADOArguments()

        # Initial setup ...

        # Flexibility to customize the optimizer (with custom options)
        optimizer = torch.optim.SGD(
            model.parameters(),
            lr = fado_args.poison_lr,
            momentum=fado_args.poison_momentum,
            weight_decay=fado_args.poison_decay
        )

        # Flexibility to customize the loss function
        criterion = torch.nn.CrossEntropyLoss()

        # The user can modify any part of the training process!
        for inputs, targets in datasets:
            outputs = model(inputs)
            loss = criterion(outputs, targets)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

        # Perform any type of post processing to the model...
```

Figure 3.9: Example implementation of a model attack (*custom_model_attack.py*).

```yaml
# fado_config.yaml

...
attack_args:
    model_attack_name: "custom_model_attack.py"
    poison_lr: 0.03
    poison_momentum: 0.9
    poison_decay: 0.005
...
```

Figure 3.10: Portion of the configuration file (*fado_config.yaml*) containing the required options to enable the injection of the custom model attack presented in Figure 3.9.

FADO expects custom implementations to have the `get_class` method to indicate which class the user wants to inject into the *Runner*. The user might be interested in implementing multiple auxiliary classes inside the file that contains the model attack implementation. As such, the `get_class` method explicitly tells `ClientAttackManager`, the component responsible for parsing the custom file, which class is to be injected.

The example demonstrates several features of a custom model attack implementation. Specifically, it shows the ability to customize the optimizer, the loss function, and the entire training loop. Moreover, the user is free to add new configuration options to the configuration file, which will be available in the custom implementation through the `FADOArguments` singleton, which is a class that contains all the configuration parameters. This singleton is available in the whole domain of implementations inside FADO.

3.3.3 Shaper

The Shaper component is responsible for processing the data examples that will be used in the FL scenario: from the server's test data to clients' training data. As such, it supports the implementation of data poisoning attacks.

These types of attacks, explained in Section 2.3.3, are based on the direct manipulation of the data examples of the compromised client in order to modify the normal behavior (e.g., changing the label of all the examples of a certain class to another value). Moreover, it is worth nothing that the customization of the *Shaper* is not suitable for the evaluation of model poisoning attacks, as this abstraction operates before the execution of model training, which is carried out by the *Runner* module (instead of the *Builder*'s module).

Implementing a custom Shaper inherently enables the user to transmit manipulated data to clients directly from the `shape()` method, making those clients malicious. We do not take into account the possibility of implementing a custom *Downloader* for malicious purposes, as implementing a *malicious Shaper* results in the modification of the entire *Builder* module pipeline.

FADO is very strict when it comes to the return values of the `shape()` method of a *Shaper*. As can be seen from the type hints in *Shaper* abstract class implementation (Figure 3.11 - Line

```python
class Shaper:

    @abstractmethod
    def shape(self) -> [List[DataLoader], DataLoader, DataLoader]:
        pass
```

Figure 3.11: Implementation of the `Shaper` abstract class.

4), FADO requires a tuple of three elements: a list of *PyTorch DataLoaders*[2], each associated to a distinct client; a DataLoader with the server test dataset, which will be used by the server at the end of each round to evaluate the global model; and a DataLoader with the server test target dataset, which is used in scenarios when the user has the goal of evaluating a specific class of a dataset. An example of the application of the server's test target dataset is when the user is evaluating a backdoor attack aiming to modify the behavior of a certain class and wants to evaluate the accuracy of the model on that class. In this situation, this DataLoader would contain only examples of that class.

3.3.4 DataAttack

The *DataAttack* component functions as an interface for users to easily integrate new data attack strategies into the framework. To incorporate a custom data attack, users must create a subclass of the *DataAttack* abstract class and define the *attack_data* method (see implementation in Figure 3.12). Moreover, he or she must define a specific option in the configuration file, *data_attack_name*, which is intended to instruct FADO to load the custom data attack. The *attack_data* method is combined with the *Shaper* component, as the parameters and expected return values are exactly the same. Specifically, if FADO detects that a *DataAttack* was specified by the user, it will retrieve the values from the *Shaper*, obtained from the method `shape()`, and execute the method `attack_data` with those values. The values returned by this latter execution will then compose the data that will be used in the FL scenario.

This approach establishes a consistent and user-friendly interface that enables the integration of diverse data attack strategies within FADO while maintaining adaptability and ease of use.

```python
class DataAttack(Attack):
    # DataLoader -> torch.utils.data.DataLoader
    def attack_data(self, clients_train_data: List[DataLoader],
    server_test_data: DataLoader, target_test_data: DataLoader) -> [List[
    DataLoader], DataLoader, DataLoader]:
        return clients_train_data, server_test_data, target_test_data
```

Figure 3.12: Implementation of the `DataAttack` abstract class.

Figure 3.13 illustrates an example implementation of a data attack, and Figure 3.14 shows the associated configuration file that allows the attack to be injected. Similarly to what was explained in Section 3.3.2, FADO also requires of the `get_class` method within the *DataAttack*

[2] torch.utils.data.Dataloader

custom implementation. This function defines the *DataAttack* subclass, within the implementation file, that the user wants FADO to inject. In this example, the malicious actor is modifying the server dataset, replacing all labels with a single value, obtained from the configuration option `label_fill`. By design, FADO is not aware of this configuration option, however, it is naturally accessible via the `FADOArguments()` singleton, which holds all configuration parameters supplied in the configuration file.

```
# custom_data_attack.py
def get_class():
    return CustomDataAttack

class CustomDataAttack(DataAttack):

    # clients_train_data -> List[DataLoader]
    # server_test_data -> DataLoader
    # target_test_data -> DataLoader
    def attack_data(self, clients_train_data, server_test_data,
    target_test_data):
        fado_args = FADOArguments()
        server_test_data.dataset.targets = [fado_args.label_fill] * len(
    server_test_data.dataset.targets)
        return clients_train_data, server_test_data, target_test_data
```

Figure 3.13: Implementation example of a data attack (*custom_data_attack.py*).

```
# fado_config.yaml
...
attack_args:
    data_attack_name: custom_data_attack.py
    label_fill: 2
...
```

Figure 3.14: Configuration file (*fado_config.yaml*) containing the required options to enable the injection of the data attack.

3.3.5 ClientAttackManager

This component acts as an intermediary between FADO and the user configuration file. Whenever FADO wants to interact with an attack interface, it calls the manager, which will interpret the configuration file and either: (1) load the attack into its database for further use and return the attack class, or (2) simply return the attack class that is already stored into its database. Lastly, it is worth nothing that this component is aware of the fact that the user may either specify an already existing attack in FADO or a custom implementation through a Python file. We show the implementation of this class in Figure 3.15.

The method `is_attacker` is used by FADO to determine at runtime whether the specified `client_id` is malicious or benign. FADO requires the user to specify the number of malicious clients he or she wants to exist through the `malicious_clients` configuration option.

```python
class ClientAttackManager:

    @classmethod
    def is_attacker(cls, client_id) -> bool:
        if not hasattr(cls, 'adversary_list'):
            cls.adversary_list = [i for i in range(1, config.malicious_clients + 1)]
        return client_id in cls.adversary_list

    @classmethod
    def get_model_attacker(cls, client_id) -> ModelAttack:
        if config.model_attack_name == "blare":
            return BlareAttacker(client_id)
        elif ".py" in config.model_attack_name:
            return load_custom_implementation(config.model_attack_name)(client_id)
        else:
            return ModelAttack()

    @classmethod
    def get_data_attacker(cls, client_id) -> DataAttack:
        if config.data_attack_name == "label-flipping":
            return LabelFlippingAttacker(client_id)
        elif ".py" in config.data_attack_name:
            return load_custom_implementation(config.data_attack_name)
        else:
            return DataAttack()
```

Figure 3.15: Implementation of the `ClientAttackManager` class.

The method `get_model_attacker` injects the model attack the user specified in the configuration. It can either be a custom implementation contained in a Python file (`.py`), or an already existing model attack like BLARE (explained in Chapter 4).

The method `get_data_attacker` is used to inject the data attack the user specified in the configuration. Likewise, it is either a custom implementation contained in a Python file (`.py`), or an already existing data attack such as a traditional label flipping attack.

3.3.6 ShaperManager

The *ShaperManager* is the component responsible for selecting the *Shaper* that will be used in the FL scenario. To do this, FADO has two different approaches: (1) it can load a custom *Shaper* specified by the user through the `shaper` configuration option, or (2) if the user does not specify the `shaper` option, it will get the `dataset` option and select the *Shaper* that provides the specified dataset. This algorithm is depicted in Figure 3.16, which shows the implementation of the `ShaperManager` class.

3.3.7 ParticipantSelector

The *ParticipantSelector* can be used to modify the algorithm that picks the clients that participate in each FL round. This component also receives information from the evaluation mechanism

```python
class ShaperManager:

    @classmethod
    def get_shaper(cls) -> Shaper:

        if '.py' in config.shaper:
            return load_custom_implementation(config.shaper)
        elif config.dataset in LEAF_DATASETS:
            return LEAFShaper()
        elif config.dataset in ('cifar10', 'cifar100'):
            return CIFARShaper()
        elif config.dataset in ('blare_cifar10', 'blare_cifar100'):
            return BlareShaper()
        else:
            raise Exception(...)
```

Figure 3.16: Implementation of the `ShaperManager` class.

(backdoor loss and accuracy), enabling the user to implement sophisticated attacks that exclude clients from the FL round, for example those that produce model updates that are gradually contributing to a decrease in backdoor accuracy. Figure 3.17 shows the implementation of the ParticipantSelector abstract class, whose interface provides two methods, `post_target_test_eval` and `post_server_test_eval`, so the selector instance can process information from the evaluation mechanism.

To select which *ParticipantSelector* to load, FADO executes the `get_selector` method which is contained in the `ParticipantSelectorManager` class, as shown in Figure 3.18.

```python
class ParticipantSelector:

    def __init__(self, initial_round: int) -> None:
        self.initial_round = initial_round

    @abstractmethod
    def get_participants(self, available_clients: List, number_of_clients: int,
     round_number: int):
        pass

    def post_target_test_eval(self, loss, accuracy):
        pass

    def post_server_test_eval(self, loss, accuracy):
        pass
```

Figure 3.17: Implementation of the `ParticipantSelector` abstract class.

By default, the server randomly selects clients from a pool. Figure 3.19 portrays the implementation of the `RandomParticipantSelector`, which is mapped by the `ParticipantSelectorManager` from the configuration option "`random`". When implementing a new strategy for selecting the round participants, the user has access to the list of all available clients, the number of clients per round according to the configuration file and the round number.

```
def get_selector(cls, current_round: int) -> ParticipantSelector:
        if config.participant_selector == 'random':
            return RandomParticipantSelector(current_round)
        elif '.py' in config.participant_selector:
            return load_custom_implementation(config.participant_selector)(
    current_round)
        else:
            raise Exception(...)
```

Figure 3.18: Implementation of `get_selector` method, contained in the `ParticipantSelectorManager` class, which is used by the server to obtain the ParticipantSelector to be used in the FL job.

```
class RandomParticipantSelector(ParticipantSelector):

    def __init__(self, initial_round: int) -> None:
        super().__init__(initial_round)

    def get_participants(self, available_clients: List, num: int, round: int):
            return random.sample(available_clients, num)
```

Figure 3.19: Implementation of the `RandomParticipantSelector`.

3.3.8 Aggregator

An overridden aggregator that changes the normal behavior of the aggregation process can further deviate the global model from the benign objective. This includes excluding certain clients from the aggregation process or further manipulating malicious updates in order to increase the attack effectiveness.

A user is also capable of applying aggregation defenses in the FL process by implementing a custom aggregator that applies those protections. Figure 3.20 shows the implementation of the *Aggregator* abstract class.

```
class Aggregator:

    @abstractmethod
    def aggregate(self, global_model: FADOModule, new_parameters):
        pass
```

Figure 3.20: Implementation of the `Aggregator` abstract class.

3.3.9 AggregatorManager

The component that has the role of selecting which `Aggregator` to load given the configuration option specified by the user is the `AggregatorManager`. The principle of execution is similar to the other managers referenced above (`ClientAttackManager` or `ShaperManager`), as it has the ability to dynamically load a custom aggregator or load an existing one from the built-in FADO database. We show the implementation of the `AggregatorManager` in Figure 3.21.

```python
class AggregatorManager:
    @classmethod
    def get_aggregator(cls) -> Aggregator:
        if config.aggregator == "diff":
            return DiffAggregator() # FedAvg
        elif ".py" in args.aggregator:
            return load_custom_implementation(config.aggregator)()
        else:
            raise Exception(...)
```

Figure 3.21: Implementation of the `AggregatorManager` class.

3.4 Node defenses implementation

FADO also enables users to apply client-side defenses in an accessible and flexible way. To achieve this, the user has to implement a class that inherits from the abstract class `ClientDefense`, as depicted in Figure 3.22. FADO expects the user to define the `defend_model_parameters` method, which receives as parameters the `FADOModule` and the old parameters, i.e., the global model sent by the server. It expects that the user modifies directly the ML model contained in `FADOModule`. Naturally, the default implementation does not apply any type of operation to the model.

```python
class ClientDefense(ABC):

    def defend_model_parameters(self, model: FADOModule, old_parameters: dict)
    -> None:
        return model
```

Figure 3.22: Implementation of the abstract class `ClientDefense`.

Additionally, the user must specify the configuration option `client_defense_name` in the configuration file, as demonstrated in Figure 3.23.

```yaml
# fado_config.yaml
defense_args:
    client:
        client_defense_name: my_defense.py
        param1: 0.2
        param2: 1.0
...
```

Figure 3.23: Configuration file portion that enables the injection of a custom client-side defense contained in the `my_defense.py` file. FADO will make the additional configuration options (*param1* and *param2*) accessible through the `FADOArguments()` singleton.

3.5 Datasets

Regarding the integration of datasets, FADO supports two strategies: the user can either specify a dataset that is already included in the tool, or he can select a custom dataset by implementing a Shaper. In practice, datasets already included in the tool come in the form of Shaper implementations.

In the FL literature, a multitude of datasets are used, encompassing various tasks such as image classification and sentiment analysis. Table 3.5 provides an overview of the most used datasets in the literature, their dimensions, and data distribution.

The task of implementing Shapers to include all those datasets is challenging and time consuming. Therefore, at the moment of this writing, FADO only supports four datasets: FEMNIST, EMNIST (through the LEAF framework[7]), CIFAR10 and CIFAR100.

Dataset	Used in	Total samples	Num. of labels	Non-IID	Task
FEMNIST [7]	[41, 44, 42]	805,263	62	✓	IC
EMNIST [11]	[55, 47, 37]	814,255	62	✗	IC
MNIST [26]	[21, 19, 34, 3, 17, 15]	70,000	10	✗	IC
CIFAR10 [24]	[55, 45, 21, 19, 47, 32, 42, 34, 3, 16, 2]	60,000	10	✗	IC
CIFAR100 [24]	[55, 3, 16]	60,000	100	✗	IC
Twitter [7]	[55, 47]	1,600,498	N/A	✓ / ✗	SA
Shakespeare [7]	[32, 37]	4,226,158	N/A	✓ / ✗	WP
Reddit [7]	[55, 47, 34, 2, 37]	56,587,343	N/A	✓	LM

Table 3.1: Some of the most used datasets in FL's literature: IC - Image Classification; SA - Sentiment Analysis; WP - Word Prediction; LM - Language Modelling.

Chapter 4

BLARE

Backdoor attacks aim to corrupt the model in a stealthy way, changing its behavior for specific inputs. However, traditional backdoors exhibit low durability, which is a metric that quantifies the persistence of the backdoor in the global model. A durable backdoor attack can persist in the global model for longer time (more rounds of training), when there are no longer clients participating maliciously in the FL process.

In this work, we present an entirely novel approach called *Backdoor Less Active REcently* (BLARE) for injecting backdoors into FL models with increased attack durability. To accomplish this goal, BLARE determines the parameters most updated by benign clients in the recent past, and then crafts the malicious model updates to prevent those parameters from being altered.

In this chapter, we will provide a detailed explanation of BLARE, starting by explaining what backdoor types we consider; then, describing why they vanish in a short period of time under traditional backdoor strategies; then, we will show the artifacts that allow BLARE to adjust malicious model updates so that they become significantly more embedded in the global model; finally, we will present the algorithm that BLARE executes to perform the attack and its corresponding implementation in FADO.

4.1 Backdoor types

We consider three types of backdoors: in-distribution (IN), edge-distribution (EDGE), and out-distribution (OUT). Each one of these backdoors targets different aspects of the data distribution to corrupt the model behavior with regard to certain inputs.

In-distribution backdoor (IN): This backdoor has the goal of corrupting the model so that it associates an incorrect label l_b to examples belonging to other label l ($l \rightarrow l_b$) from within the data distribution. For instance, in CIFAR10, an attacker accomplishes this backdoor if it tampers the training dataset so that all examples that contain *dogs* are mislabeled as *trucks* (label $6 \rightarrow 9$).

Edge-distribution backdoor (EDGE): In this type of attack, the backdoor targets examples with features that are rare or unlikely to be included in the data distribution. By maliciously

assigning these examples to a distinct label, the attack will cause these rare examples to be incorrectly classified.

For example, if we consider a dataset like CIFAR10, where we have a specific class for cars (automobiles, label 2), this attack might focus on corrupting the model so that only cars with very rare features within the data distribution would trigger the backdoor, without corrupting the model for the rest of the car examples. In CIFAR10, there are 5,000 examples of cars; however, the attacker could have the goal of causing misclassifications **only** for pink automobiles, which are very rare in the dataset.

Out-distribution backdoor (OUT): This type of backdoor is very stealthy, as it consists of inserting examples into the backdoor training dataset which correspond to a class that is not included in the label space of the data distribution. By associating these examples in the dataset to a label from within same label space of the data distribution, the attacker is making the model aware of new features which naturally associates with a class which does not correspond to the features the example contains.

For example, in CIFAR10, one such backdoor attack would require the attacker to insert examples into the training dataset where the inputs would correspond to images of bears (CIFAR10 does not contain a class of bears) and label those examples as *trucks* (label 9).

4.2 Backdoor persistence

One of the key properties of BLARE is that it successfully injects the backdoor into the FL global model with increased persistence. This persistence allows the backdoor to remain longer in the model, even when the malicious actor no longer injects the backdoor.

One of the reasons why traditional FL backdoors have low persistence is that the weights that the attacker updates to allow the backdoor to be injected into the FL model are overwritten by benign clients during the training process. As such, benign client updates naturally mitigate the effects of the backdoor attack, as in the most common threat scenario, the number of benign clients outnumbers the set of malicious clients.

BLARE crafts malicious updates to avoid the alterations of those regions (active regions), gradually inserting the backdoor into the global model in less active regions. It is expected that future benign model client updates will not apply considerable changes on the parameters of passive regions, thus not overriding malicious updates, guaranteeing a more persistent backdoor injection.

4.3 Active regions

We found that some observations can be made regarding the active regions of an FL model during the training process. These observations are useful in understanding the role of these weights in BLARE's context.

In each round, the most impactful model updates occur in a restricted group of model weights by a small amount. Moreover, the composition of these regions is dynamic between rounds due to the participation of clients and their inherent heterogeneous data distribution (non-i.i.d.), resulting in weights updates with dissimilar directions. Additionally, it was observed that during a certain period of training, a significant portion of the client's highest updates tend to merge on a single active region. Furthermore, it was observed that the parameters contained in active regions determined by BLARE are non-uniformly spread over most of the model layers.

4.4 Attack architecture

Figure 4.1 depicts the architecture and functioning of BLARE, at the high level.

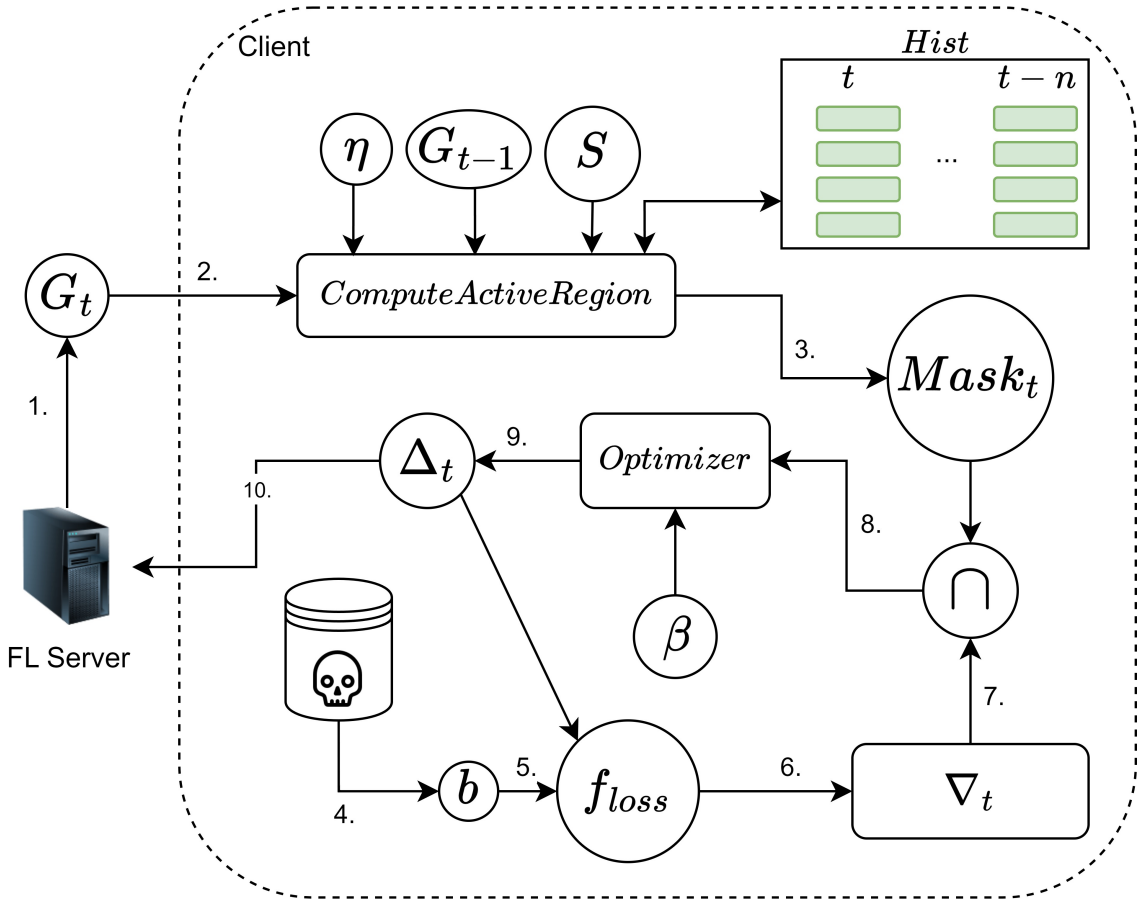

Figure 4.1: BLARE's architecture and functioning.

In Step 1, the server sends the global model to the corrupted client, which proceeds to execute `ComputeActiveRegion` (Step 2) to create $Mask_t$ (Step 3), which contains information about the active regions corresponding to past benign updates.

`ComputeActiveRegion` creates $Mask_t$ from the history of benign active regions ($Hist$), the global learning rate (η), the global model of the past round (G_{t-1}) and the provided percentage of the most active parameters (S), as will be demonstrated in Section 4.5.

From Steps 4 to 6, the traditional SGD steps are executed: the client divides the training dataset into batches, which have been maliciously modified by the attacker to reflect the backdoor's behavior; and then obtains the gradients ∇_t through the loss function.

In Step 7, BLARE uses $Mask_t$, generated in Step 3, to prevent changes in the gradients ∇_t corresponding to the active regions, resulting in a reduced backdoor impact on the model parameters that were most modified by benign clients. This step results in the backdoor update effects being less likely to be overwritten by benign updates, contributing to the increased durability of the backdoor after the attack ceases.

Those gradients, which now contain the (modified) backdoor, are then passed into the optimizer with a backdoor learning rate (β) (Step 9), which outputs the local model. Finally, the model update U_t is sent to the server (Step 10).

4.5 Algorithm

Algorithm 1 and Algorithm 2 depict the execution of an FL scenario under the BLARE attack, on the server and clients, respectively. We now describe each of those algorithms in detail, starting with the server.

Algorithm 1 Server – FL scenario under BLARE

```
    # D_i: Local dataset of client i
    # C: Set of all the participants
    # G_t: Global model on round t
    # N: Total number of participants
    # m: Number of malicious participants
    # T: Total number of rounds
    # η: Global learning rate
 1: function SERVERROUNDTRAINING
 2:     G_0 ← RandomInitialize()
 3:     for each round t = 0, ..., T do
 4:         C_t ← ParticipantSelector(C)          ▷ up to m clients will be malicious
 5:         for each client i ∈ C_t do
 6:             U_t^i ← ClientUpdate(G_t, D_i, N)
 7:         end for
 8:         U_t ← f_agg({U_t^i | i ∈ C_t})        ▷ Aggregate local models
 9:         G_{t+1} ← G_t + ηU_t
10:     end for
11: end function
```

4.5.1 Server

Before the server starts the FL process, it randomly initializes the global model parameters through $RandomInitialize$ (Line 2 of Algorithm 1). In each round t, it will invoke the *ParticipantSelector* to select the clients that will participate in the corresponding round (Line 4). Each selected client i executes $ClientUpdate$, which returns its local model update U_t^i (Line 6).

All local model updates are aggregated into a single model update, through an aggregation function (f_{agg}) (Line 8), which is used to update the global model according to a certain global learning rate η (Line 9). In our case, our aggregation function is *FedAvg* [32] (see Section 2.2.3), as it is one of the most referenced aggregation strategies in current literature. *FedAvg* performs an average update of all the clients model updates, such as $\Delta^t \leftarrow \sum_{i \in C_t} \frac{1}{|C_t|} U_t^i$.

4.5.2 Client

The client algorithm starts at $ClientUpdate$, which recurs to an auxiliary function $isMalicious$ to determine whether it executes a benign or malicious client update (Line 2 of Algorithm 2).

The $BenignClientUpdate$ function contains the expected instructions of a normal FL client, since it performs SGD in batches on its local dataset D_i for a predefined number of local epochs E_b. It returns the difference between its local model (L_t) and the global model (G_t), which is assigned to the local model at the start of the function (Line 25 of Algorithm 2).

If the client is malicious, $MaliciousClientUpdate$ is executed instead, which has a structure similar to $BenignClientUpdate$. However, the malicious dataset (D_i) has been maliciously altered so that it contains examples that are incorrectly classified to reflect the backdoor behavior.

To increase the longevity of the backdoor after it stops attacking, BLARE avoids modifying the parameters that are most modified by the overall benign clients, which we call active regions. To determine what are those active regions, the function $ComputeActiveRegion$ is executed (Line 26), which starts by calculating an approximation of last round's general benign update ($BenignUpdate$) (Line 19): the aggregated global model update of the past round consists of the difference between the global model of the current and past round (G_t and G_{t-1}); however, this update still contains malicious contributions, hence the third clause in the operation on Line 19 ($-m.\eta.L_{t-1}$), which removes those contributions of the update, making it an aggregated benign update.

This calculation is followed by an inference (through $mostActiveParameters$) of what are the top-k% parameters in $BenignUpdate$ ($activeRegion^t$) (Line 20). As such, $activeRegion^t$ is the set of parameters that BLARE considers the most modified parameters of the last round by benign clients. $activeRegion^t$ could be sufficient to accomplish BLARE's goal of avoiding updates on the active regions, however, there is a limitation: relying only on the information of the last round can be insufficient. For this reason, BLARE stores a history of the active regions of the last Q rounds ($Hist$) to increase the amount of information; therefore, after obtaining $activeRegion^t$, BLARE inserts it into $Hist$ (Line 21) and determines the active regions in the entire history instead of only the last calculated active region (Line 22), resulting in a more aware and informed active region.

The generated active region is used in the SGD procedure (Line 33) to prevent updates to the model parameters that form the region – the gradients of these parameters will have a value of zero. After performing the altered SGD procedure for E_m local epochs and with a backdoor learning rate of β, BLARE returns the difference between its local model (L_t) and the global model (G_t).

Algorithm 2 Clients – FL scenario under BLARE

```
    # η: Benign learning rate
    # β: Backdoor learning rate
    # D_i: Local dataset of client i
    # m: Number of malicious participants
    # N: Total number of participants
    # E_b: Number of benign local epochs
    # E_m: Number of malicious local epochs
    # Q: History size
    # S: Percentage of selected most active parameters
 1: function CLIENTUPDATE
 2:     if isMalicious(i) then
 3:         U_t^i ← MaliciousClientUpdate(G_t, D_i, N)
 4:     else
 5:         U_t^i ← BenignClientUpdate(G_t, D_i, N)
 6:     end if
 7:     return U_t^i
 8: end function
 9: function MOSTACTIVEPARAMETERS(D, S)
10:     Mask ← ones(|D|)
11:     for i = 0 to |D| do
12:         if D[i] ∈ {p ∈ D | p is in the top S% of D} then
13:             Mask[i] ← 0
14:         end if
15:     end for
16:     return Mask
17: end function
18: function COMPUTEACTIVEREGION
19:     BenignUpdate ← |G_t − G_{t−1} − (m.η.L_{t−1})/N|        ▷ Calculate benign update
20:     activeRegion^t ← mostActiveParameters(BenignUpdate, S)
21:     Hist ← activeRegion^t + ... + activeRegion^{t−Q}
22:     return mostActiveParameters(Hist, S)
23: end function
24: function MALICIOUSCLIENTUPDATE
25:     L_t ← G_t
26:     Mask^t ← ComputeActiveRegion(L_{t−1}, G_t, G_{t−1}, m, N, η, S)
27:     for e = 1 to E_m do
28:         while b ∈ batches(D_i) do
29:             grad ← ∇f(L_t, b)                            ▷ Compute gradient
30:             grad ← grad × Mask^t                         ▷ Prevent update on active region
31:             L_t ← L_t − β.grad                           ▷ Update local model
32:         end while
33:     end for
34:     return L_t − G_t
35: end function
36: function BENIGNCLIENTUPDATE
37:     L_t ← G_t
38:     for e = 1 to E_b do
39:         while b ∈ batches(D_i) do
40:             grad ← ∇f(L_t, b)                            ▷ Compute gradient
41:             L_t ← L_t − η.grad                           ▷ Update local model
42:         end while
43:     end for
44:     return L_t − G_t
45: end function
```

4.6 Implementation: Shaper

In this section, we describe the implementation of the *shaper* of BLARE in FADO. More specifically, it explains the datasets collection and data distribution, the generation of the benign and malicious clients datasets, and the creation of the server-side datasets.

4.6.1 Datasets origin

At the moment of writing, the implementation of BLARE in FADO supports two datasets: CIFAR10 and CIFAR100.

We obtain these datasets using the *Torchvision* library, which is an extension of *PyTorch*, and aims to simplify the development of computer vision applications by offering a set of tools and datasets for training and testing deep learning models specifically tailored for this sort of tasks.

In the context of `BLAREShaper`, the first stage of execution is the loading of the training and test datasets from the Torchvision library into the *Shaper* memory.

4.6.2 Benign training dataset

We implemented BLARE in FADO taking into account the need for the most realistic scenario in a cross-device FL setting. To accomplish this, we "distribute" the training dataset over the benign clients following the Dirichlet distribution.

The Dirichlet distribution [33] is used to model uncertainty or variability in data, making it an appropriate choice for simulating real-world scenarios [29]. The hyperparameter for the Dirichlet distribution, *alpha* (α), is passed to the *Shaper* through the configuration file.

All the client's datasets are now stored inside the *Shaper* in the form of a list of PyTorch DataLoaders. Here, each element of the list corresponds to the dataset of a client.

4.6.3 Malicious training dataset

The malicious dataset contains a mix of correctly labeled examples and backdoor instances. The proportion of backdoor instances can be customized through the configuration file; however, in our experiments, the proportion rate is 40%.

In BLARE, the backdoor examples are the subset of the training dataset where the label associated with the input is equal to the class that the user wants to attack. This class, which we call `target_class`, is passed through the configuration file to the *Shaper*.

The `BLAREShaper` expects a parameter called `poison_label_swap` in the configuration file, which designates the label to be applied to each attacked training example. In Section 4.7.2 we will review the multiple configuration options required by BLARE.

For example, when using BLARE with the CIFAR10 dataset, if `target_class` $= 5 \wedge$ `poison_label_swap` $= 9$, all images originally labeled as "dog" (class 5) will have their labels changed to "truck" (class 9).

4.6.4 Server-side datasets

The server utilizes two separate datasets for different purposes during the evaluation phase of the global model: the test dataset and the target-test dataset. Both datasets are derived from the CIFAR10/100 interface of the Torchvision library, specifically from the 10,000 examples that compose the test dataset of the library.

The test dataset is directly extracted from the test set included in the CIFAR10/100 interface. It serves as a general benchmark for evaluating the overall performance of the global model.

The target-test dataset is a evaluation dataset that contains only examples of the class that is being attacked, whose purpose is to evaluate BLARE's backdoor accuracy. A very low accuracy in the target-test dataset shows that the backdoor injection is not effective, whereas a very high accuracy signifies successful injection of the backdoor into the global model.

4.7 Implementation: ModelAttack

The goal of this section is to explain some of the most interesting aspects of BLARE's implementation when it comes to the manipulation of the model parameters, namely the operations that BLARE executes in order to prevent backdoor updates on parameters that have been frequently updated by benign clients. This manipulation is crucial for BLARE effectiveness, as only altering the training dataset is not enough to inject a durable backdoor when the attacker ceases to attack the FL model.

4.7.1 BLARE training process

Figure 4.7.1 illustrates the Python code that corresponds to the *ModelAttack* implementation of BLARE in FADO. The main differences between the shown implementation and a traditional training process implementation are the `create_mask` method and the `apply_mask` method, in Lines 4 and 12 of Figure 4.7.1, respectively. It should be noted that, by the time these instructions start executing, the `dataset` variable has already been modified by *BLAREShaper* to contemplate the backdoor's goals. Both of these methods are described in detail in the next sections.

`create_mask`

One of the most important aspects of this implementation part is how BLARE, from a list of model parameter updates, infers the top-k% parameters, which we call active regions, through the `topk` method of the PyTorch library. Figure 4.3 illustrates the use of `topk`, in this context.

As can be seen, the presented implementation of `create_mask` is heavily abstracted, since the calculation of the benign update is omitted. However, in this shown implementation, the `update` variable is already free from malicious contribution, reflecting the expected overall benign update of the past round.

```
fado_args = FADOArguments()

def attack_training_process(fado_module, dataset, client_id, current_round):
    mask = create_mask(model, fado_args.mask)
    optimizer = torch.optim.SGD(..., lr = fado_args.poison_lr)
    criterion = torch.nn.CrossEntropyLoss()
    for epoch in range(fado_args.retrain_poison):
        for inputs, targets in dataset:
            loss = criterion(outputs, targets)
            optimizer.zero_grad()
            loss.backward(retain_graph=True)
            apply_mask(model, mask)
            optimizer.step()
    self.update_state(model)
```

Figure 4.2: Implementation of the BLARE attack.

The complete implementation of this method is extensive and complex. Therefore, we decided to include the complete implementation in Figure A.1.

```
# blare_attacker.py
def create_mask(update, ratio):
    (...)
    _, indices = torch.topk(update, int(len(update_flat)*(1.0-ratio)))
    mask = torch.ones(len(update))
    mask[indices] = 0
    return mask
```

Figure 4.3: Abbreviated implementation of the $create\_mask$ in Python, which determines the top-(`ratio`)% parameters in `update` and sets these parameters to zero.

`apply_mask`

This method is responsible for preventing the model updates from having values inside the calculated active regions. In order to accomplish this, BLARE multiplies each layer contained in the created mask (`mask`) with the corresponding model layer. As expected, the values inside the mask can be one (1) or zero (0). This approach will successfully prevent the most modified parameters by benign clients from being altered by the malicious model update. The implementation of this mechanism is shown in Figure 4.4.

```
# blare_attacker.py
def apply_mask(self, model, mask):
    mask_copy = iter(mask)
    for name, parms in model.named_parameters():
        parms.grad = parms.grad * next(mask_copy)
```

Figure 4.4: Algorithm that BLARE uses to perform the erase active regions from the malicious model update.

4.7.2 BLARE configuration

This section provides an overview of the several configuration options of the BLARE's implementation. Figure 4.5 shows a working configuration file for BLARE with the CIFAR10 dataset.

```
fl_training_process:
    ...
    participant_selector: blare

defense_args:
    client:
        client_defense_name: norm_clip
        clip_norm: 0.2

attack_args:
    model:
        model_attack_name: blare
    rounds_before_attack: 0
    rounds_attacking: 300
    malicious_clients: 1
    target_class: 5
    poison_label_swap: 9
    poison_lr: 0.03
    poison_momentum: 0.9
    poison_decay: 0.005
    retrain_poison: 3
    mask: 0.99
    queue_size: 10
    alpha: 0.9
    size_of_poison_dataset: 512
    fraction: 0.4

dataset_spec:
    dataset: blare_cifar10
```

Figure 4.5: Example configuration file for BLARE that uses the CIFAR10 dataset.

In addition to some of the essential configuration options presented in Section 3.1.3, the `attack_args` section of the configuration file has the relevant configuration parameters that BLARE expects to use to carry out the attack.

Table 4.1 shows the options (`attack_args` to take into account when customizing BLARE functioning.

Configuration option	Description
rounds_attacking	The number of rounds BLARE will attack.
malicious_clients	The number of malicious client in the experiment.
target_class	The class that BLARE will attack (backdoor).
poison_label_swap	The class that will be replace the *target_class* label.
poison_lr	The learning rate of the malicious optimizer
retrain_poison	The number of local epochs of the malicious clients.
mask	BLARE will find the top-(1 - mask)% parameters to build the mask.
queue_size	History ($Hist$) size.
alpha	Dirichlet parameter (α) for the BLARE *Shaper*.
size_of_poison_dataset	Number of examples in the backdoor (malicious) dataset.
fraction	Fraction of the malicious training data that contains backdoor examples.

Table 4.1: List of most relevant parameters for BLARE in the `attack_args` section of the configuration file.

In the next chapter, we will conduct several experiments in which we evaluate the effect of some of these parameters on the effectiveness of BLARE.

Chapter 5

Evaluation

This chapter evaluates the effectiveness of BLARE. The experiments were conducted with FADO in `simulation` mode, covering and exploring a multitude of configurations. In the process, we will discuss how different parameters can impact BLARE's effect on various FL scenarios, especially when it comes to backdoor durability.

5.1 Models & Datasets

In our experiments, we used the CIFAR10 and CIFAR100 datasets along with the ResNet18 model [20], a convolutional neural network that is well suited for image classification tasks.

The CIFAR10 dataset consists of 60,000 color images with 10 different classes, with 6,000 images per class. The dataset is divided into 50,000 training images and 10,000 testing images, with each image being 32x32 pixels in size.

The CIFAR100 dataset has 60,000 images divided into 100 classes, each with 600 images. It includes 50,000 training images and 10,000 testing images. The 100 classes are grouped into 20 superclasses, and each image has both a class and superclass label. This structure increases complexity, requiring models to learn more detailed representations.

Using CIFAR10 and CIFAR100 provides a comprehensive evaluation framework for FL due to their varying levels of complexity and the need for models to generalize across diverse image classes. Furthermore, these datasets are widely recognized benchmarks in the machine learning community, ensuring the relevance and robustness of our evaluation.

Data distribution

The benign training dataset is distributed across clients using the Dirichlet distribution with $\alpha = 0.9$, to provide data heterogeneity that follows the n.i.i.d. property of a realistic FL scenario. In order to demonstrate the data heterogeneity under these conditions, Figure 5.1 shows a graph that depicts the number of examples of a specific class ($label = 5$) that each client has in its training dataset, given a scenario where we distributed CIFAR10 across 100 clients. In this graph, we can observe that some clients do not have any examples with $label = 5$, which can pose a significant

risk if an attacker performs a backdoor attack targeting this class in a round where the majority of selected clients are in this condition.

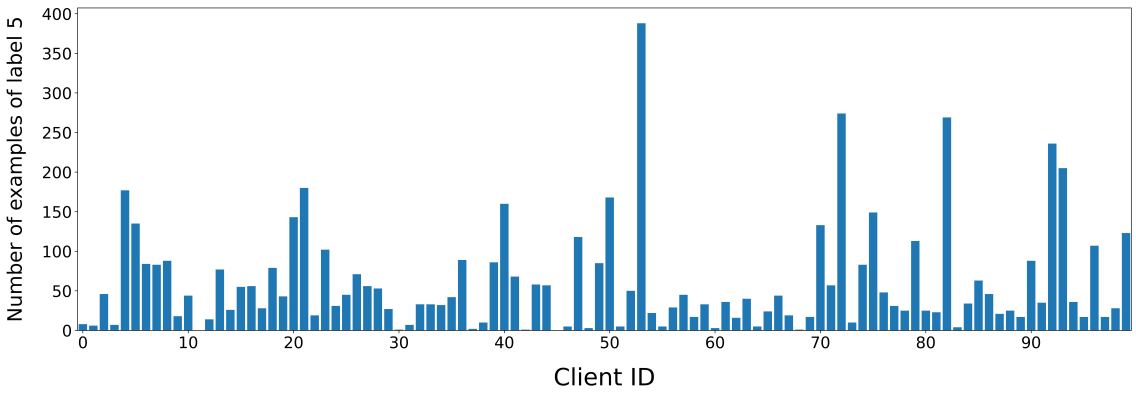

Figure 5.1: Number of examples with $label = 5$ given to each client, in a scenario with 100 clients where data was distributed using the Dirichlet distribution with $\alpha = 0.9$ (CIFAR10 dataset).

Backdoor training dataset

The backdoor training dataset is made up of pds examples (default value for $pds = 512$), in which 40% of these examples are backdoor instances. Backdoor examples which means that these examples were manipulated to represent malicious behavior and are intended to be classified into a targeted class by the model. The remaining 60% of the dataset are correctly labeled examples that do not correspond to the target class under attack. This mix of correctly labeled examples and backdoor instances ensures that the dataset can effectively train the model's ability to recognize and classify both benign and malicious inputs.

Backdoor test dataset

The backdoor test dataset consists only of examples from the class that is being attacked, with the malicious label assigned. This ensures that we have an appropriate test dataset to evaluate the backdoor performance, i.e., if the model classifies the attacked label as the malicious label.

5.2 Configuration

For experiments using CIFAR10, we ran the FL scenario for 1800 rounds with 1000 benign clients, creating a checkpoint with a test dataset accuracy of approximately 92%. For experiments using CIFAR100, we executed the FL scenario for 3200 rounds with 500 benign clients, also creating a checkpoint with a test dataset accuracy of around 66%.

Each attack experiment starts from the respective checkpoint and then runs for 1000 FL rounds where 10 clients participate in each round. This group of clients is randomly selected by the server. For the first 300 rounds, a single attacker is always present in the group of picked clients,

injecting an IN-distribution backdoor[1]. For the remaining 700 rounds, the attacker is no longer present in the scenario, with only benign clients participating. The objective is to maximize the backdoor accuracy after the attacker ceases the attack (Round 300-1000), assessing the durability and longevity of the backdoor attack.

We show, in Table 5.1, the baseline configuration for all our experiments for both of the datasets we will use (CIFAR10 and CIFAR100).

Configuration option	CIFAR10	CIFAR100
Number of clients in the FL experiment	1000	500
Number of rounds of the FL experiment	1000	1000
Number of rounds malicious clients attack	300	100
Clients selected by the server in each round	10	10
Dirichlet distribution parameter (α)	0.9	0.9
Training batch size	64	64
Local training epochs in each benign client	2	2
Learning rate in each benign client	0.001	0.001
Backdoor learning rate (`blr`)	0.03	0.02
Local training epochs in malicious clients (`mle`)	3	3
Queue (history) size	10	2
Mask ratio (S)	0.99	0.99
Size of backdoor training dataset (`pds`)	512	512
Size of backdoor test dataset	512	50
Fraction of backdoor examples in training dataset	0.4	0.4

Table 5.1: Baseline configuration values for the experiments.

Before presenting and discussing the results of our experiments, we must define the metrics that we will consider when comparing different configurations.

5.3 Metrics

In our evaluation, we considered three metrics computed with the global model: benign accuracy, backdoor accuracy, and backdoor longevity.

Backdoor accuracy (`BackAcc`): Measures how the model behaves when it comes to backdoor injection. More specifically, its value represents the percentage of correctly classified backdoor instances, i.e., if one is trying to inject a backdoor with a model update that was obtained from training an altered dataset such as $dog \rightarrow truck$, the backdoor accuracy tells us the percentage of inputs corresponding to $dogs$ that were classified as $trucks$.

Benign accuracy (`BenAcc`): Measures the overall performance of the model on the benign test dataset, indicating the proportion of correctly classified examples. This metric is essential for evaluating how well the model generalizes from the training data to the unseen data and to evaluate the backdoor's impact on the global model behavior.

[1] In CIFAR10: $deer$ becomes $ship$ (label $5 \rightarrow 9$); In CIFAR100: bed becomes $couch$ (label $5 \rightarrow 25$).

Backdoor durability (`BackDur`): Measures how long the backdoor remains effective after the malicious clients no longer perform the attack, providing a clear metric for the resilience of the backdoor against future benign model updates. In practice, high durability indicates that the backdoor persists in the model for a longer period, meaning the model remains vulnerable to incorrect inferences on the attacked class.

5.4 Results

In this section, we assess the behavior of BLARE in various adversarial scenarios. In each experiment with CIFAR10, we will evaluate the effect of BLARE and discuss those results as we vary a certain parameter (Section 5.4.1). Furthermore, we carried out an experiment for CIFAR100 (Section 5.4.3) where we evaluated the effect of norm clipping defense on BLARE's effectiveness.

Each of the figures that will be shown in the following sections is composed of two parts: a graph of the global model benign accuracy and a graph of the backdoor accuracy. Each graph result is composed of the average value of three (3) distinct runs, each using a distinct random number seed.

Furthermore, each section is accompanied by a table that will provide statistics related to the backdoor accuracy (`BackAcc`), such as the maximum value it achieved (`MaxAcc`) and its value t rounds after the malicious client ceased the attack (e.g., $t = 100$ corresponds to the backdoor accuracy at round 400). By showing the `BackAcc` with $t = 100, 300, 600$, the table will provide an intuition for the durability of the backdoor (`BackDur`).

5.4.1 CIFAR10

Backdoor learning rate (`blr`)

In this experiment, we test four different configurations for the backdoor learning rate: 0.03, 0.05, 0.07 and 0.09 (see Figure 5.2).

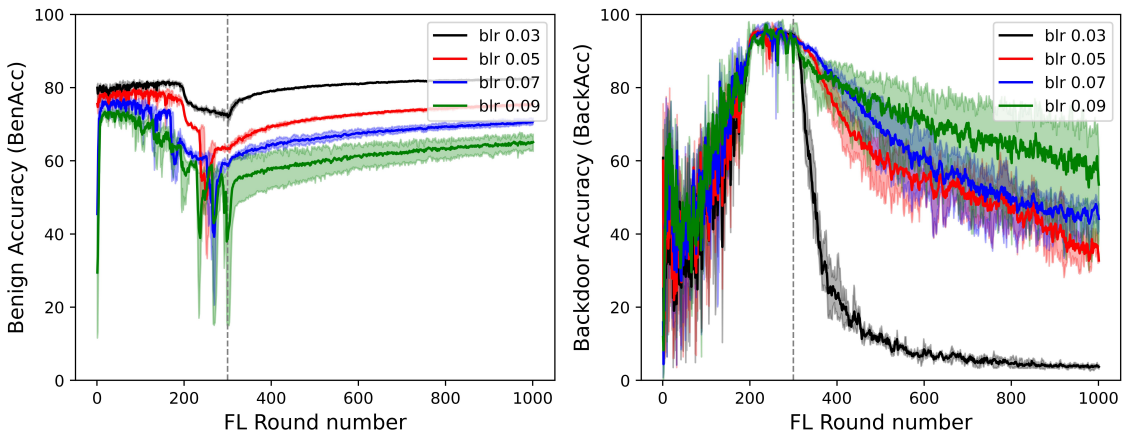

Figure 5.2: Impact of varying the backdoor learning rate (CIFAR10).

In all values considered for the backdoor learning rate (`blr`), the backdoor is successfully injected in the initial phase of the attack (first 300 rounds). We observe that a higher `blr` results in an increased backdoor durability, i.e., the backdoor accuracy remains high for a longer time after the attacker ceases the attack.

We argue that a higher `blr` causes the model to move more aggressively toward the optimal point determined by the backdoor data, resulting in faster convergence in the backdoor examples. The large gap observed between the accuracy and durability of the backdoor indicates that when $\texttt{blr} > 0.03$, the model is able to reach an optimal malicious point which makes it very effective.

Throughout the backdoor injection, the global model is being influenced by the attacker, resulting in updates that have contrary directions to fully benign ones. As such, it is expected that since we are attacking a class from the data distribution, the global model accuracy decreases more as we increase `blr`. This can be observed in the graph on the left of Figure 5.2.

Moreover, we notice that when $\texttt{blr} = 0.09$, the model suffers updates that are so aggressive that in some rounds the global model behaves badly with a low benign accuracy ($< 30\%$).

Furthermore, this experiment shows that the value of the backdoor learning rate has a strong impact on backdoor accuracy, its longevity, and the benign accuracy. Table 5.2 shows the statistics of this experiment relative to the backdoor accuracy (`BackAcc`).

`blr`	`MaxAcc`	`BackAcc` ($t = 100$)	`BackAcc` ($t = 300$)	`BackAcc` ($t = 600$)
0.03	95.18	23.31	5.86	4.10
0.05	96.09	74.48	55.92	42.84
0.07	96.22	**82.36**	61.91	47.40
0.09	**97.20**	82.29	**71.16**	**61.59**

Table 5.2: Statistics on `BackAcc` when varying the backdoor learning rate (CIFAR10).

Backdoor learning rate (with norm clipping defense)

This experiment is very similar to the previous one; however, the scenario is now under the norm (L2) clipping defense [44]. Norm clipping constrains the magnitude of model updates by bounding the norm of each update to a predefined threshold, which in our case is 0.2. This means that any update with a norm exceeding this threshold will be scaled down to fit within the limit. By doing so, norm clipping ensures that no single update can have an excessively large influence on the aggregated model parameters.

Figure 5.3 depicts the graph with the results of this experiment. The baseline `blr` (0.03) has better results in the global model benign accuracy and the backdoor accuracy. A `blr` higher than 0.03 will cause the norm of malicious updates to be very high. The defense limits the influence of malicious model updates, making the backdoor more difficult to insert in the initial phase of the experiment. Such behavior can be observed as the backdoor accuracy, in all cases, only increases significantly after 200 rounds of the attacker participant in the FL scenario. Furthermore, in a scenario with a norm clipping defense, we observe that a lower `blr` will result in a more effective and durable backdoor.

Moreover, there is a significant deterioration of the global model when the attacker is present in the scenario, since the global model accuracy decreases when we increase the `blr`. This phenomenon can be explained as follows: When the malicious client joins the FL process, the model has mostly converged, which means that the updates of the benign client have a low l2-norm value; the l2-norm of the malicious model update is expected to be very high ($>> 0.2$), as it reflects backdoor data; given that the server performs *FedAvg*, treating every client update evenly, the malicious update has a very high impact on the global model, making it unstable and deviated from the benign optimal point. This is not the case when $blr = 0.03$, which is probably related to the fact that such `blr` does result in a malicious update with a lower l2-norm, decreasing the overall impact on the global model.

Table 5.3 shows the statistics of this experiment relative to the backdoor accuracy (`BackAcc`).

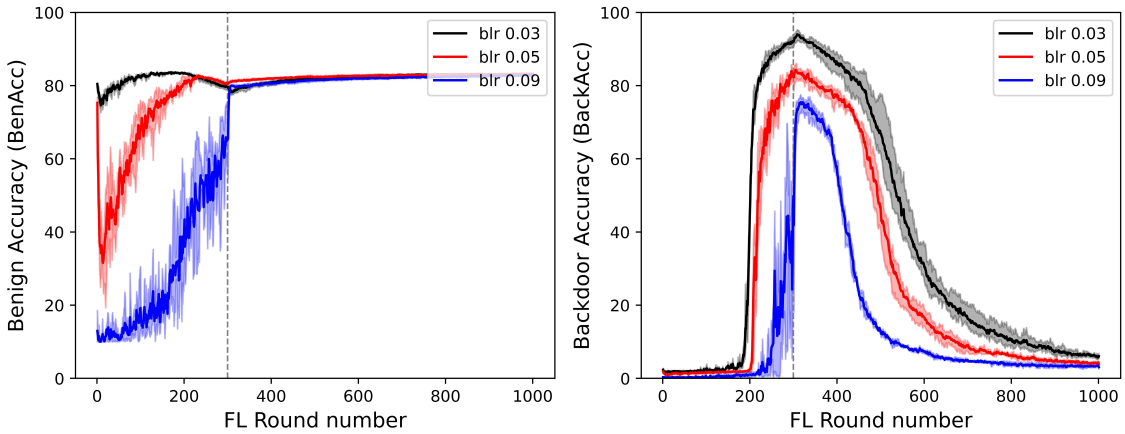

Figure 5.3: Impact of varying the backdoor learning rate against norm clipping defense (CIFAR10).

`blr`	`MaxAcc`	`BackAcc` ($t = 100$)	`BackAcc` ($t = 300$)	`BackAcc` ($t = 600$)
0.03	**94.01**	**85.87**	**30.34**	**7.42**
0.05	84.31	76.76	16.60	4.82
0.09	75.46	55.79	7.22	3.39

Table 5.3: Statistics on `BackAcc` when varying the backdoor learning rate against norm clipping defense (CIFAR10).

Mask ratio (S)

In this experiment (see Figure 5.4), our goal is to evaluate the effect of varying the ratio (S) of the model parameters that BLARE uses when calculating the active regions. In practice, by reducing the value in S, the percentage of benign model parameters that BLARE considers to calculate the active region increases. For this reason, the graphs in Figure 5.4 have the label arr, such that $arr = 1 - S$.

We note that the variation of this parameter does not result in a significant reduction of the

global model benign accuracy, which guarantees stealth. Furthermore, the backdoor accuracy graph indicates that a decrease in S results in a more durable backdoor, which is explained by the fact that BLARE has more information on the model parameters that benign participants modify to train their local model. We notice that there is a significant improvement between 10% to 20% of *active region ratio*, which indicates that there might be an optimal value for this parameter.

Table 5.4 shows the statistics of this experiment relative to the backdoor accuracy (`BackAcc`).

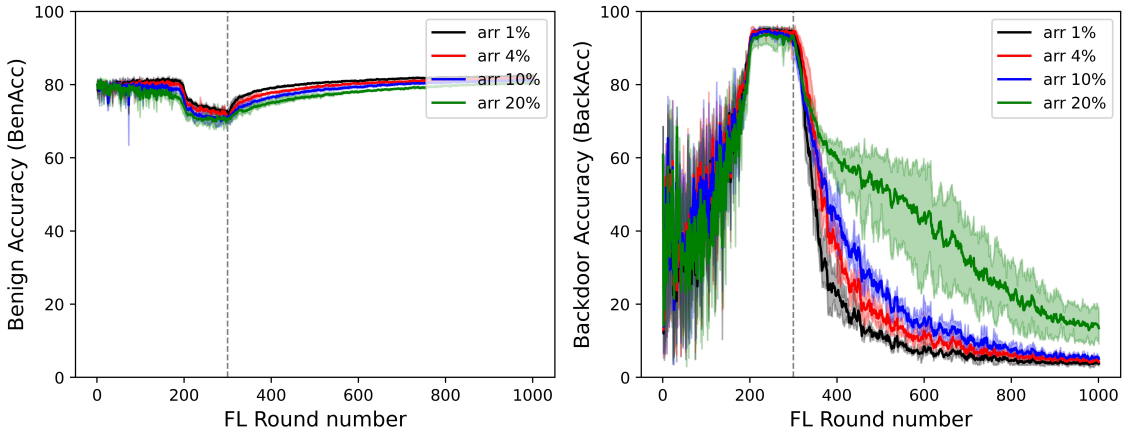

Figure 5.4: Impact of varying the percentage of top-$arr\%$ benign model parameters BLARE uses to infer the active regions and build the mask (CIFAR10).

S	`MaxAcc`	`BackAcc` ($t = 100$)	`BackAcc` ($t = 300$)	`BackAcc` ($t = 600$)
0.99	**95.18**	23.31	8.14	4.10
0.96	94.92	36.46	12.43	5.01
0.90	94.86	44.21	16.34	6.32
0.80	93.82	**59.38**	**45.12**	**16.67**

Table 5.4: Statistics on `BackAcc` when varying the percentage of top-$arr\%$ benign model parameters BLARE uses to infer the active regions and build the mask (CIFAR10).

Size of malicious dataset (`pds`)

The size of the malicious dataset refers to the number of examples included in the backdoor's training dataset. A larger malicious dataset provides the attacker with more training data, increasing the likelihood that the model will generalize well under the malicious conditions introduced by the backdoor.

Figure 5.5 suggests that when $pds = 512$, the backdoor quickly vanishes after the attacker ceases the attack. On the other hand, $pds >= 1024$ results in an extremely durable backdoor, as it does not vanish from the model after (at least) 700 rounds without adversary contributions. Moreover, the global model benign accuracy does not suffer considerable perturbation in all the cases. Despite vanishing much earlier than in the other two cases, the $pds = 512$ configuration achieves better backdoor accuracy (nearly 100%), which can be explained by the fact that the model is likely to have reached an optimal point.

Table 5.5 shows the statistics of this experiment relative to the backdoor accuracy (BackAcc).

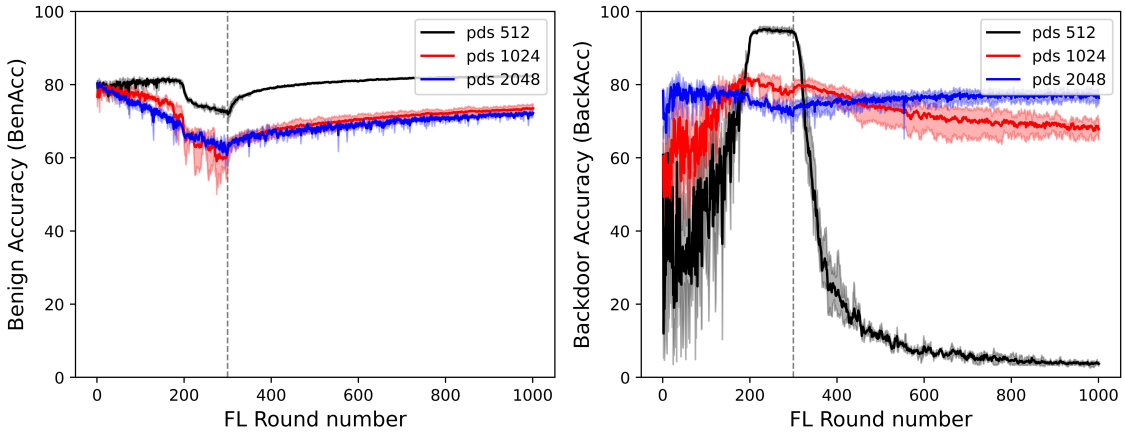

Figure 5.5: Impact of varying the size of the backdoor (poison) training dataset (CIFAR10).

pds	MaxAcc	BackAcc ($t = 100$)	BackAcc ($t = 300$)	BackAcc ($t = 600$)
512	**95.18**	23.31	8.14	4.10
1024	81.80	**77.44**	72.23	69.40
2048	80.34	75.05	**76.76**	**77.64**

Table 5.5: Statistics on BackAcc when varying the size of the backdoor (poison) training dataset (CIFAR10).

Number of clients

This experiment, depicted in Figure 5.6, aims to provide an evaluation on the effect of the number of clients in the FL scenario to evaluate our backdoor attack. In all scenarios, there is a single attacker in the first 300 rounds.

In the initial phase of the attack (Round 1 to 300), the global model benign accuracy is very similar, with slight variations. When it comes to backdoor accuracy, we observe that more clients will result in a backdoor that is injected earlier, with higher top accuracy on the backdoor test dataset. However, when the attacker stops injecting the backdoor, we notice that fewer clients in the scenario will result in a very low durability of the backdoor attack.

The data distribution across clients is n.i.i.d through the Dirichlet distribution. Moreover, with fewer clients, each client has more examples in its training dataset, so it is more likely to have more examples of the attacked class by the backdoor, resulting in model updates that greatly override the ones produced by BLARE.

Table 5.6 shows the statistics of this experiment relative to the backdoor accuracy (BackAcc).

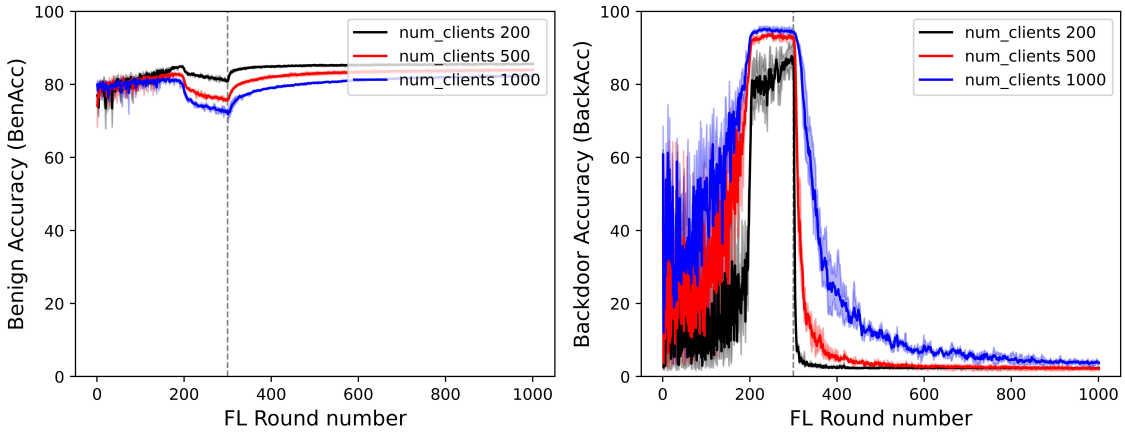

Figure 5.6: Impact of the backdoor with different number of clients in the FL scenario (CIFAR10).

NumClients	MaxAcc	BackAcc ($t = 100$)	BackAcc ($t = 300$)	BackAcc ($t = 600$)
200	87.76	2.41	2.34	2.28
500	93.82	5.14	2.73	2.28
1000	**95.18**	**23.31**	**8.14**	**4.10**

Table 5.6: Statistics on BackAcc with different number of clients in the FL scenario (CIFAR10).

Malicious local epochs (mle)

The epochs executed by an SGD algorithm are crucial in any scenario that implies deep learning. More epochs will result in the model being updated more times; however, too many epochs could result in overfitting, deteriorating the global model.

In Figure 5.7, we observe that more malicious epochs will significantly increase backdoor durability. However, the observed reduction in the global model accuracy as we increase the epochs might be explained by the fact that the model is moving more aggressively toward the optimal point related to the backdoor effect. As we are performing a IN-distribution backdoor attack, a reduction in the global model accuracy is expected, as one of the classes from the data distribution is being attacked. Furthermore, a reduction in the global model accuracy can also be explained by a possibility of the model being overfitted because of the excessive local epochs.

Table 5.7 shows the statistics of this experiment relative to the backdoor accuracy (BackAcc).

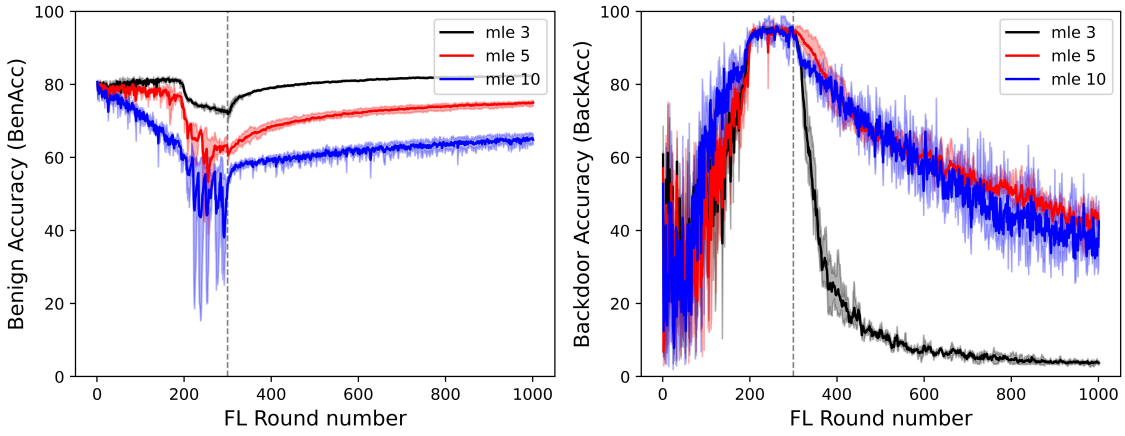

Figure 5.7: Impact of varying the number of malicious SGD epochs the attacker performs (CIFAR10).

mle	MaxAcc	BackAcc ($t = 100$)	BackAcc ($t = 300$)	BackAcc ($t = 600$)
3	95.18	23.31	8.14	4.10
5	95.70	**76.83**	**61.59**	**46.61**
10	**95.96**	75.91	60.42	36.52

Table 5.7: Statistics on BackAcc when varying the number of malicious SGD epochs the attacker performs.

Dirichlet distribution alpha (α)

In this experiment, we aim to evaluate how different *alpha* α values affect the backdoor metrics. The *alpha* parameter controls the level of concentration or sparsity of the data distribution across clients. Higher values of α will result in more balanced data distributions and a lower concentration of examples across clients.

According to the results presented in Figure 5.8, a very low α parameter (such as $\alpha = 0.1$) causes the backdoor to vanish slightly faster than the other options, whereas the global model benign accuracy is not significantly affected. Moreover, we can observe some downspikes in the initial phase of the attack for an $\alpha = 0.1$. Since the data distribution is very unbalanced, there are benign clients that only have examples from one class. Therefore, when these benign clients are selected to train, they strongly deviate the global model from an optimal point, resulting in a increased reduction on the global model benign accuracy. We consider that, under such conditions, this parameter has a reduced influence on BLARE's effectiveness.

Table 5.8 shows the statistics of this experiment relative to the backdoor accuracy (BackAcc).

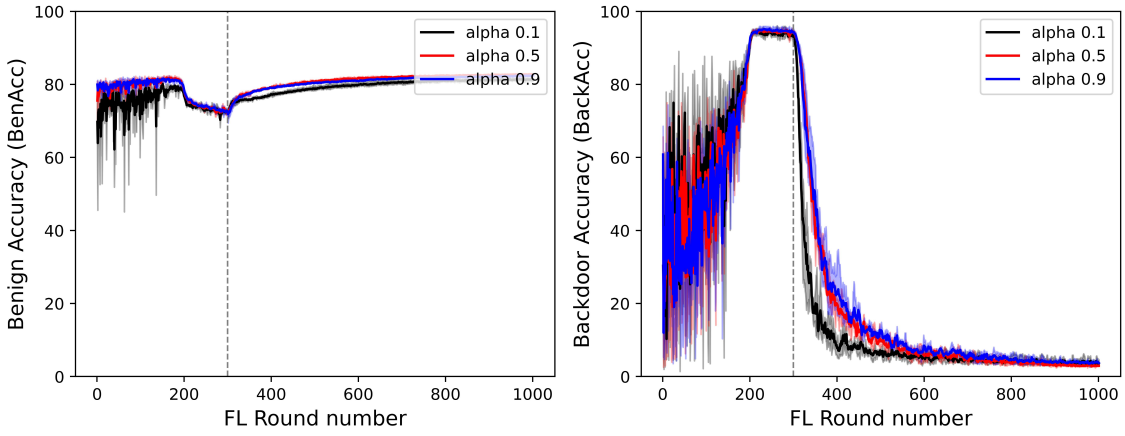

Figure 5.8: Impact of the data heterogeneity level (through Dirichlet distribution) data on the backdoor's performance (CIFAR10).

Dirichlet α	MaxAcc	BackAcc ($t = 100$)	BackAcc ($t = 300$)	BackAcc ($t = 600$)
0.1	94.86	7.68	5.21	3.97
0.5	94.86	19.66	5.60	3.39
0.9	**95.18**	**23.31**	**8.14**	**4.10**

Table 5.8: Statistics on BackAcc with different levels of data heterogeneity through Dirichlet distribution (CIFAR10).

5.4.2 Norm clipping defense

In this experiment, our objective is to evaluate the norm clipping defense against BLARE, while using CIFAR10. As can be observed in Figure 5.9, an higher norm clipping value makes the backdoor less durable without affecting the global model accuracy significantly. However, higher norm clipping values cause the backdoor to be injected earlier and with a small increase of the maximum backdoor accuracy. This happens because the malicious model updates are not being so restricted/constrained as in cases where the norm clipping threshold is low (e.g., $nc = 0.2$).

After the attacker ceases the attack, benign contributions prevail in the FL scenario. These benign contributions are being less constrained by the defense, overriding the backdoor effect, which results in the backdoor vanishing faster. If we decrease the norm clipping threshold ($nc = 0.2$), the backdoor is able to achieve higher durability, since the benign contributions that would override the backdoor effect through high norm model updates are constrained by the defense.

Table 5.9 shows the statistics of this experiment relative to the backdoor accuracy (BackAcc).

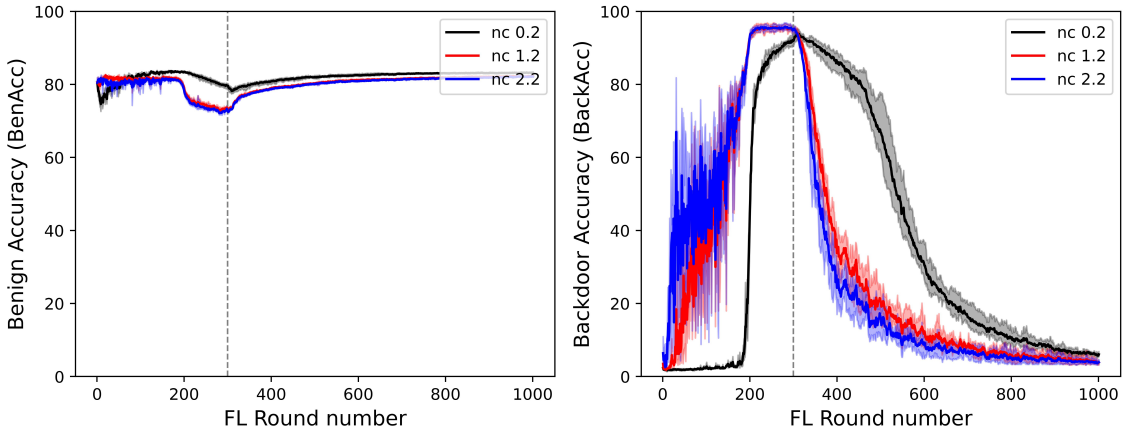

Figure 5.9: Impact of varying the norm clipping (nc) threshold (CIFAR10).

nc threshold	MaxAcc	BackAcc ($t = 100$)	BackAcc ($t = 300$)	BackAcc ($t = 600$)
0.2	94.01	**85.87**	**30.34**	**7.42**
1.2	95.77	34.77	12.37	4.49
2.2	**95.83**	25.26	9.31	4.23

Table 5.9: Statistics on BackAcc when varying the norm clipping (nc) threshold.

5.4.3 CIFAR100

In this section, we demonstrate the results using CIFAR100 in two distinct configurations: one with norm clipping defense and another without it (see Figure 5.10). In these experiments, the malicious client only attacks for 100 rounds, and there are 500 clients participating in the FL scenario.

We demonstrate that BLARE is capable of injecting the backdoor into the FL scenario successfully, reaching a very high level of durability in both configurations. We argue that CIFAR100 is more vulnerable to backdoor attacks, since there are 100 classes, each with 600 images. It is more likely that the clients participating in the FL round do not have examples of the attacked class, as there are few examples of it. The norm clipping defense mitigates the backdoor durability in higher rounds by a small amount.

Furthermore, we also note that in both cases, the global model benign accuracy is not heavily impacted by the backdoor attack, which can be explained by the fact that the attack is targeting only one of a hundred classes (1/100), thus not resulting in a significant impact.

Table 5.10 shows the statistics of this experiment relative to the backdoor accuracy (BackAcc).

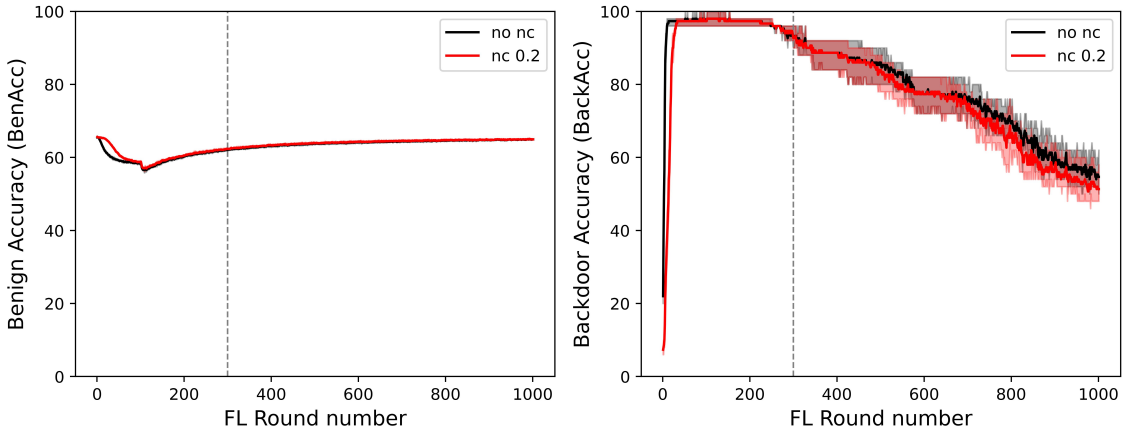

Figure 5.10: BLARE's effect with CIFAR100 dataset, with and without norm clipping defense.

Defense	MaxAcc	BackAcc ($t = 100$)	BackAcc ($t = 300$)	BackAcc ($t = 600$)
✓	**98.0**	**88.67**	74.67	57.33
✗	**98.0**	**88.67**	**76.0**	**61.33**

Table 5.10: Statistics on BackAcc with and without norm clipping defense (CIFAR100).

5.5 Discussion

In Section 1.2.1, we outlined the properties we aimed to incorporate into FADO's design to address the limitations of existing implementations in the literature: realism, comparable results, scalability, and ease of use and customizability. In this section, we cover these properties, discussing whether current implementations align with the desired goals.

5.5.1 FADO Properties

Realism

We find that FADO ensures a relatively high degree of realism, as the tool is able to simulate FL scenarios following real-world conditions, such as the fact that it enables users to distribute data in a n.i.i.d. fashion using Dirichlet distribution, for instance. In addition, FADO is prepared to handle cross-device and cross-silo scenarios, with certain limitations of cross-device scenarios that imply a very large number of clients.

Moreover, the tool guarantees, by design, isolation between entities through encapsulation, since each client is represented as a different object which cannot access properties from other objects (e.g., clients cannot access variables from other clients or the server). This prevents evaluation scenarios where users might try to directly manipulate information of a certain client from a distinct client.

Comparable Results

We found that FADO has a similar behavior across different system setups, as it does not require any type of special configuration or hardware. Moreover, configuration files can be shared and the experiments will have the same behavior. However, it might be worth mentioning that results can differ slightly, especially when we consider that different users might have different GPU hardware, or that the RNGs (Random number generators) might produce different random values, which naturally can affect the experiment results.

Scalability

FADO has a moderate level of scalability, as it can execute without any effort an experiment with 1000 clients. FADO is equipped with mechanisms to avoid OOM (out-of-memory) issues in GPUs and to accelerate the training of clients through threading. However, it is not able to run experiments with millions of clients, as should be expected in a more extreme cross-device scenario.

One strategy to make this possible would be to implement mechanisms into FADO that would enable the tool to be executed on different physical devices (such as Docker Swarm architecture). It should be noted that implementing these mechanisms requires a significant level of effort and resources.

Ease-of-use and Customizability

FADO provides accessible interfaces to implement attacks and defenses in multiple parts of the FL scenario: from the clients' data and training process to the server mechanisms such as the aggregator or the participant selector. The tool works around a configuration file that is simple and straightforward when it comes to the different FL parameters, providing the user the possibility of creating new configuration options that are automatically accessible in their custom implementations.

Moreover, the environment in which FADO executes is easily configurable, only requiring Python. This implies that a newcomer to FL can quickly create and execute an FL scenario.

Chapter 6

Conclusion

Federated learning (FL) is a promising technology within distributed machine learning, offering benefits such as reduced communication costs and enhanced data privacy. It allows for training machine learning models on decentralized data, significantly improving applications in sectors such as finance, healthcare, and autonomous driving.

This work's objective was to address certain limitations related to FL literature, namely when it comes to the evaluation of node attacks, which can corrupt the model's behavior to align with the attacker's motivation. We introduced FADO, a versatile tool that enables researchers and industry professionals to evaluate FL scenarios under various attack conditions. FADO is scalable, capable of executing scenarios with thousands of clients while maintaining realism and high customizability. Its user-friendly interfaces make it easy to integrate different FL components and add external modules, especially those related to attacks and defenses.

Furthermore, this work also focused on presenting a novel backdoor attack called BLARE, which was designed to improve the persistence of malicious backdoors in the global model even after the attacker ceases to inject malicious updates. BLARE manipulates the attacker's model parameters during training to ensure the backdoor remains effective longer than those in existing literature. We evaluated BLARE using FADO with CIFAR10 and CIFAR100 datasets, conducting numerous experiments to assess the impact of various parameters on the attack's effectiveness. The results indicate that BLARE successfully injects a durable malicious backdoor into the FL process. By implementing BLARE in FADO, we conducted extensive experiments to validate BLARE's effectiveness under various conditions, while also demonstrating the utility of FADO in the evaluation of FL scenarios regarding security mechanisms.

Finally, this work has successfully achieved its goals by developing FADO, a tool that addresses the limitations of existing FL implementations, and introducing a novel and more durable backdoor attack (BLARE). These contributions provide valuable insights and knowledge to further increase the awareness of security in FL systems.

6.1 Future Work

Although this work has made significant progress in addressing FL limitations alongside introducing a new backdoor attack, there are many potential areas of future research to advance these contributions further.

6.1.1 FADO

Extending FADO's ability to carry out simulations across multiple computation nodes is an essential step toward increasing its scalability. By distributing computational load across multiple nodes, FADO will be able to run more complex FL scenarios, resulting in more realistic and scalable tool and allowing for a broader range of use cases and attacks.

Another important area for future development is the incorporation of more complex datasets and models into FADO, particularly those which employ Generative Pre-trained Transformers (GPTs). This improvements will increase FADO's utility by enabling researchers to explore the unique challenges and vulnerabilities of these advanced models, which represent a significant breakthrough in AI applications.

6.1.2 BLARE

For BLARE, future work could focus on testing the attack strategy across a wider variety of datasets. While this work has validated BLARE's effectiveness using CIFAR10 and CIFAR100, evaluating it on different datasets will provide a more comprehensive understanding of its behavior and impact.

Furthermore, exploring the interaction between BLARE and different defense mechanisms remains an essential area of future work. By understanding how BLARE can be detected and mitigated by various defenses, researchers can develop more effective countermeasures to improve the security of FL systems.

Appendix A

Mask creation implementation

This appendix shows the complete Python implementation of `create_mask` (see Figure A.1), which creates the mask to be used by BLARE to prevent updates on benign active regions. Moreover, it also includes an exhaustive explanation of how the history ($Hist$) artifact is implemented in Python.

```
update_list_benign = list()

for name, data in model.named_parameters():

    past_update_poison = self.past_delta_weights_poison[name] * (1/args.
    clients_per_round) * lr

    past_update_benign = data - self.past_weights[name] - past_update_poison
    update_list_benign.append(past_update_benign.abs().flatten())

update_flat = torch.cat(update_list_benign)

_, indices = torch.topk(update_flat, int(len(update_flat)*(1.0-ratio)))

top_flat = torch.zeros(len(update_flat))
top_flat[indices] = 1.0

# Global mask

top_global = self.mask_grad_list

if len(self.past_tops) >= args.queue_size:
    top_old = self.past_tops.pop(0)
    top_global = top_global - top_old + top_flat
else:
    if top_global is None:
        top_global = top_flat
    else:
        top_global = top_global + top_flat

self.mask_grad_list = top_global
self.past_tops.append(top_flat.cpu())
```

Figure A.1: Complete implementation of `create_mask`.

The generation of $Mask_t$ (or, in the code, `mask_grad_list`), has two parts: the first being the inference of the most active parameters in the current round t ($activeRegion^t$) and after the insertion of this artifact into the queue $Hist$ (history of *active regions*), the inference of the actual mask that will be returned to the local training process to be used throughout the training to prevent updates on the active region.

Line 3-8 of Figure A.1 builds an estimate of the overall benign update of the previous round using the information that was stored by BLARE in previous rounds. The overall benign update information is used to obtain the top-k% most modified parameters, through the `topk` method of the PyTorch library (Line 12 of Figure A.1). In Line 15, `top_flat` is the representation of $activeRegion_t$.

The following lines (Line 19-31) describe the insertion of `top_flat` into the entire history that BLARE stores throughout the rounds, namely `top_global`. It is expected that, for example, after ten rounds, `self.past_tops` contains ten entries and `top_global` might have some list elements (model parameters) that equal to the value 10.0, particularly if these parameters are always selected (as being the most active) in the last ten rounds.

To successfully prevent updates on certain parameters, those parameters need to have a value of zero. As such, BLARE's implementation has the final goal of transforming the `top_global` list, where the most active parameters have high values (such as 10.0), into a list with zero on those indices. Table A.1 shows the evolution of a possible state of `top_global` variable throughout ten rounds. In this small example, we are assuming that the model is composed of only ten parameters, which is unrealistic – however, the purpose is only to provide a small and simple example. We present the strategy that replaces the parameters that have a value higher than `max_v` with zero (0.0), and the others with a value of one (1.0).

round t	`top_global`
t = 1	[0.0, 0.0, 1.0, 0.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0]
t = 2	[0.0, 0.0, 1.0, 1.0, 2.0, 2.0, 0.0, 0.0, 1.0, 0.0]
t = 3	[1.0, 0.0, 1.0, 2.0, 3.0, 3.0, 0.0, 0.0, 1.0, 0.0]
t = ...	[...]
t = 10	[1.0, 0.0, 2.0, 6.0, 10.0, 10.0, 1.0, 0.0, 2.0, 0.0]

Table A.1: Example of the evolution of `top_global` state over ten rounds on a model with only ten parameters.

Based on the example we are focusing on, the value of `max_v` is half of the length of the `self.past_tops` list (5.0). After ten rounds, BLARE will only set to zero parameters that have an accumulative value higher than 5.0, which ignores parameters that are less updated, such as those with values like 2.0 or 3.0. As expected, in round $t = 10$, the fourth parameter, which has a value of 6.0, will also be selected. The corresponding implementation of this method is presented in Figure A.2.

```
max_v = math.ceil(len(self.past_tops)/2.0)
mask_flat = torch.where(top_global > max_v, 0.0, 1.0)
```

Figure A.2: Implementation part of BLARE that selects the parameters to be erased from `top_global`.

The final part of mask creation is to turn `mask_flat` from a flat list of parameters to a structured list that comprehends the existence of layers, so that erasing active regions becomes more straightforward, as demonstrated in Section 4.7.1.

Bibliography

[1] Martin Abadi, Andy Chu, Ian Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. pages 308–318, 2016.

[2] Eugene Bagdasaryan, Andreas Veit, Yiqing Hua, Deborah Estrin, and Vitaly Shmatikov. How to backdoor federated learning. In *Proceedings of the International Conference on Artificial Intelligence and Statistics*, pages 2938–2948, 2020.

[3] Moran Baruch, Gilad Baruch, and Yoav Goldberg. A little is enough: Circumventing defenses for distributed learning. In *Proceedings of the International Conference on Neural Information Processing Systems*, pages 8635–8645, 2019.

[4] Peva Blanchard, El Mahdi El Mhamdi, Rachid Guerraoui, and Julien Stainer. Machine learning with adversaries: Byzantine tolerant gradient descent. In *Proceedings of the Advances in Neural Information Processing Systems*, pages 118–128, 2017.

[5] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloe Kiddon, Jakub Konečný, Stefano Mazzocchi, H. Brendan McMahan, Timon Van Overveldt, David Petrou, Daniel Ramage, and Jason Roselander. Towards federated learning at scale: System design. In *Proceedings of the Machine Learning and Systems*, pages 374–388, 2019.

[6] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of the International Conference on Computational Statistics*, pages 177–186, 2010.

[7] Sebastian Caldas, Sai Meher Karthik Duddu, Peter Wu, Tian Li, Jakub Konečný, H. Brendan McMahan, Virginia Smith, and Ameet Talwalkar. LEAF: A benchmark for federated settings. 2018.

[8] Xiaoyu Cao, Minghong Fang, Jia Liu, and Neil Zhenqiang Gong. Fltrust: Byzantine-robust federated learning via trust bootstrapping. 2020.

[9] Xinyun Chen, Chang Liu, Bo Li, Kimberly Lu, and Dawn Song. Targeted backdoor attacks on deep learning systems using data poisoning. 2017.

[10] Francois Chollet. *Deep learning with Python*. Simon and Schuster, 2021.

[11] Gregory Cohen, Saeed Afshar, Jonathan Tapson, and André van Schaik. Emnist: Extending mnist to handwritten letters. In *Proceedings of the International Joint Conference on Neural Networks*, pages 2921–2926, 2017.

[12] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. Large scale distributed deep networks. In *Proceedings of the Advances in neural information processing systems*, pages 1223–1231, 2012.

[13] Cynthia Dwork. Differential privacy. In *Proceedings of the Automata, Languages and Programming*, pages 1–12, 2006.

[14] Ahmet M Elbir, Burak Soner, Sinem Çöleri, Deniz Gündüz, and Mehdi Bennis. Federated learning in vehicular networks. In *Proceedings of the IEEE International Mediterranean Conference on Communications and Networking*, pages 72–77, 2022.

[15] Minghong Fang, Xiaoyu Cao, Jinyuan Jia, and Neil Zhenqiang Gong. Local model poisoning attacks to Byzantine-Robust federated learning. In *Proceedings of the USENIX Security Symposium*, pages 1623–1640, 2020.

[16] Chong Fu, Xuhong Zhang, Shouling Ji, Jinyin Chen, Jingzheng Wu, Shanqing Guo, Jun Zhou, Alex X Liu, and Ting Wang. Label inference attacks against vertical federated learning. In *Proceedings of the USENIX Security Symposium*, pages 1397–1414, 2022.

[17] Clement Fung, Chris JM Yoon, and Ivan Beschastnikh. Mitigating sybils in federated learning poisoning. 2018.

[18] Simson Garfinkel. Differential Privacy and the 2020 US Census. *MIT Case Studies in Social and Ethical Responsibilities of Computing*. https://mit-serc.pubpub.org/pub/differential-privacy-2020-us-census.

[19] Yifan Guo, Qianlong Wang, Tianxi Ji, Xufei Wang, and Pan Li. Resisting distributed backdoor attacks in federated learning: A dynamic norm clipping approach. In *Proceedings of the IEEE International Conference on Big Data*, pages 1172–1182, 2021.

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[21] Najeeb Moharram Jebreel, Josep Domingo-Ferrer, David Sánchez, and Alberto Blanco-Justicia. Defending against the label-flipping attack in federated learning. 2022.

[22] Peter Kairouz, H. Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Kallista Bonawit, Zachary Charles, Graham Cormode, Rachel Cummings, Rafael G. L. D'Oliveira, Hubert Eichner, Salim El Rouayheb, David Evans, Josh Gardner,

Zachary Garrett, Adrià Gascón, Badih Ghazi, Phillip B. Gibbons, Marco Gruteser, Zaid Harchaoui, Chaoyang He, Lie He, Zhouyuan Huo, Ben Hutchinson, Justin Hsu, Martin Jaggi, Tara Javidi, Gauri Joshi, Mikhail Khodak, Jakub Konecný, Aleksandra Korolova, Farinaz Koushanfar, Sanmi Koyejo, Tancrède Lepoint, Yang Liu, Prateek Mittal, Mehryar Mohri, Richard Nock, Ayfer Özgür, Rasmus Pagh, Hang Qi, Daniel Ramage, Ramesh Raskar, Mariana Raykova, Dawn Song, Weikang Song, Sebastian U. Stich, Ziteng Sun, Ananda Theertha Suresh, Florian Tramèr, Praneeth Vepakomma, Jianyu Wang, Li Xiong, Zheng Xu, Qiang Yang, Felix X. Yu, Han Yu, and Sen Zhao. Advances and open problems in federated learning. *Foundations and Trends in Machine Learning*, 14(1–2):1–210, 2021.

[23] Jakub Konečný, H. Brendan McMahan, Felix X. Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. Federated learning: Strategies for improving communication efficiency. 2016.

[24] A. Krizhevsky and G. Hinton. Learning multiple layers of features from tiny images. *Master's thesis, Department of Computer Science, University of Toronto*, 2009.

[25] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the Advances in Neural Information Processing Systems*, pages 84–90, 2012.

[26] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. pages 2278–2324, 1998.

[27] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. pages 2278–2324, 1998.

[28] Minghui Li, Wei Wan, Jianrong Lu, Shengshan Hu, Junyu Shi, Leo Yu Zhang, Man Zhou, and Yifeng Zheng. Shielding federated learning: Mitigating byzantine attacks with less constraints. 2022.

[29] Qinbin Li, Yiqun Diao, Quan Chen, and Bingsheng He. Federated learning on non-iid data silos: An experimental study. 2021.

[30] Qinbin Li, Zeyi Wen, Zhaomin Wu, Sixu Hu, Naibo Wang, Yuan Li, Xu Liu, and Bingsheng He. A survey on federated learning systems: Vision, hype and reality for data privacy and protection. In *Proceedings of the IEEE Transactions on Knowledge and Data Engineering*, 2021.

[31] Yang Liu, Yan Kang, Xinwei Zhang, Liping Li, Yong Cheng, Tianjian Chen, Mingyi Hong, and Qiang Yang. A communication efficient collaborative learning framework for distributed features. 2020.

[32] H. Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Proceedings of the the International Conference on Artificial Intelligence and Statistics*, pages 1273–1282, 2017.

[33] Tom Minka. Estimating a dirichlet distribution. 2000.

[34] Thien Duc Nguyen, Phillip Rieger, Huili Chen, Hossein Yalame, Helen Möllering, Hossein Fereidooni, Samuel Marchal, Markus Miettinen, Azalia Mirhoseini, Shaza Zeitouni, et al. FLAME: Taming backdoors in federated learning. In *Proceedings of the USENIX Security Symposium*, pages 1415–1432, 2022.

[35] Mustafa Safa Ozdayi, Murat Kantarcioglu, and Yulia R Gel. Defending against backdoors in federated learning with robust learning rate. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 9268–9276, 2021.

[36] Matthias Paulik, Matt Seigel, Henry Mason, Dominic Telaar, Joris Kluivers, Rogier van Dalen, Chi Wai Lau, Luke Carlson, Filip Granqvist, Chris Vandevelde, Sudeep Agarwal, Julien Freudiger, Andrew Byde, Abhishek Bhowmick, Gaurav Kapoor, Si Beaumont, Áine Cahill, Dominic Hughes, Omid Javidbakht, Fei Dong, Rehan Rishi, and Stanley Hung. Federated evaluation and tuning for on-device personalization: System design applications. 2021.

[37] Krishna Pillutla, Sham M. Kakade, and Zaid Harchaoui. Robust aggregation for federated learning. *IEEE Transactions on Signal Processing*, 70:1142–1154, 2022.

[38] Nicola Rieke, Jonny Hancox, Wenqi Li, Fausto Milletarì, Holger R. Roth, Shadi Albarqouni, Spyridon Bakas, Mathieu N. Galtier, Bennett A. Landman, Klaus Maier-Hein, Sébastien Ourselin, Micah Sheller, Ronald M. Summers, Andrew Trask, Daguang Xu, Maximilian Baust, and M. Jorge Cardoso. The future of digital health with federated learning. *npj Digital Medicine*, September 2020.

[39] Filipe Rodrigues, Rodrigo Simões, and Nuno Neves. FADO: A federated learning attack and defense orchestrator. In *Proceedings of the 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 141–148, 2023.

[40] Giorgio Severi, Matthew Jagielski, Gökberk Yar, Yuxuan Wang, Alina Oprea, and Cristina Nita-Rotaru. Network-level adversaries in federated learning. In *Proceedings of the IEEE Conference on Communications and Network Security*, pages 19–27, 2022.

[41] Virat Shejwalkar and Amir Houmansadr. Manipulating the byzantine: Optimizing model poisoning attacks and defenses for federated learning. In *Proceedings of the Network and Distributed System Security Symposium*, pages 18–37, 2021.

[42] Virat Shejwalkar, Amir Houmansadr, Peter Kairouz, and Daniel Ramage. Back to the drawing board: A critical evaluation of poisoning attacks on production federated learning. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 1354–1371, 2022.

[43] Jacob Steinhardt, Pang Wei Koh, and Percy Liang. Certified defenses for data poisoning attacks. In *Proceedings of the International Conference on Neural Information Processing Systems*, pages 3520–3532, 2017.

[44] Ziteng Sun, Peter Kairouz, Ananda Theertha Suresh, and H. Brendan McMahan. Can you really backdoor federated learning? 2019.

[45] Vale Tolpegin, Stacey Truex, Mehmet Emre Gursoy, and Ling Liu. Data poisoning attacks against federated learning systems. In *Proc. European Symposium On Research In Computer Security*, pages 480–501, 2020.

[46] Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao. Improving the speed of neural networks on CPUs. In *Proceedings of the Deep Learning and Unsupervised Feature Learning Workshop*, 2011.

[47] Hongyi Wang, Kartik Sreenivasan, Shashank Rajput, Harit Vishwakarma, Saurabh Agarwal, Jy-yong Sohn, Kangwook Lee, and Dimitris Papailiopoulos. Attack of the tails: Yes, you really can backdoor federated learning. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, pages 16070–16084, 2020.

[48] Kang Wei, Jun Li, Ming Ding, Chuan Ma, Howard H. Yang, Farhad Farokhi, Shi Jin, Tony Q. S. Quek, and H. Vincent Poor. In *Federated Learning With Differential Privacy: Algorithms and Performance Analysis*, pages 3454–3469, 2020.

[49] Chulin Xie, Minghao Chen, Pin-Yu Chen, and Bo Li. Crfl: Certifiably robust federated learning against backdoor attacks. In *Proceedings of the International Conference on Machine Learning*, pages 11372–11382, 2021.

[50] Chulin Xie, Keli Huang, Pin-Yu Chen, and Bo Li. Dba: Distributed backdoor attacks against federated learning. In *Proceedings of the International Conference on Learning Representations*, 2019.

[51] Liu Yang, Di Chai, Junxue Zhang, Yilun Jin, Leye Wang, Hao Liu, Han Tian, Qian Xu, and Kai Chen. A survey on vertical federated learning: From a layered perspective. 2023.

[52] Timothy Yang, Galen Andrew, Hubert Eichner, Haicheng Sun, Wei Li, Nicholas Kong, Daniel Ramage, and Françoise Beaufays. Applied federated learning: Improving google keyboard query suggestions. *arXiv:1812.02903*, 2018.

[53] Dong Yin, Yudong Chen, Ramchandran Kannan, and Peter Bartlett. Byzantine-robust distributed learning: Towards optimal statistical rates. In *Proceedings of the International Conference on Machine Learning*, pages 5650–5659, 2018.

[54] Syed Zawad, Ahsan Ali, Pin-Yu Chen, Ali Anwar, Yi Zhou, Nathalie Baracaldo, Yuan Tian, and Feng Yan. Curse or redemption? how data heterogeneity affects the robustness of federated learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 10807–10814, 2021.

[55] Zhengming Zhang, Ashwinee Panda, Linyue Song, Yaoqing Yang, Michael Mahoney, Prateek Mittal, Ramchandran Kannan, and Joseph Gonzalez. Neurotoxin: Durable backdoors in federated learning. In *Proceedings of the International Conference on Machine Learning*, pages 26429–26446, 2022.