FADO: A Federated Learning Attack and Defense Orchestrator

Filipe Rodrigues, Rodrigo Simões, Nuno Neves
LASIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal
frodrigues@lasige.di.fc.ul.pt, rsimoes@lasige.di.fc.ul.pt, nuno@di.fc.ul.pt

***Abstract*—Federated Learning (FL) is a distributed machine learning approach allowing multiple parties to train a model collaboratively without sharing sensitive data. It has gained widespread popularity recently due to its ability to preserve data privacy. However, FL also poses novel security challenges since training relies on data and computations from many entities that a malicious actor might have compromised, as they are usually geographically dispersed and independently managed.**

Evaluations of current FL security mechanisms in the literature are often based on simplistic testing environments and demand complex programming to integrate new attacks/defenses. Therefore, this work presents an accessible platform that leverages a realistic environment to facilitate the experimentation and evaluation of new solutions in relevant FL scenarios. Comparison with already proposed approaches is also expedited since FADO provides a few out-of-the-box implementations. To demonstrate the platform's utility, we develop a use case based on a recently published network attack.

***Index Terms*—federated learning, security, testing platform, attacks, defenses**

I. Introduction

Federated Learning (FL) [1] [2] 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.

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) [3], autonomous driving [4], and automatic speech recognition (e.g., Siri) [5]. 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.

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) [6].

Even though FL is a technology that brings benefits related to privacy, attacks on these systems are still possible [7]–[11] due to the reliance on many devices that are usually dispersed over a wide area. 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 [10], [11], 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.

Consequently, researchers and industry aiming to evaluate their models in the presence of specific attacks would benefit from a realistic FL platform that facilitates testing without wasting unnecessary effort. However, current benchmarks and platforms to test security mechanisms in FL have several limitations, preventing them from meeting such criteria. For example, often, there is an absence of network communication among the FL entities, a lack of accessibility to implementations, and unrealistic data distributions across clients.

This paper proposes FADO[1], an emulation platform for FL that facilitates the experimentation of attacks and defenses under diverse conditions. The main objectives of FADO are: (i) *Realism:* Experiments carried out in the platform should provide similar results as running them in the real world; (ii) *Comparable Results:* It should be simple to compare different attacks and defenses while executing them in various system settings (e.g., cross-device and cross-silo); (iii) *Scalability:* Physical resources should be managed efficiently and effectively to enable the emulation of large test case scenarios (e.g., high numbers of clients); (iv) *Ease-of-Use and Customizability:* Both novice and expert users should find the platform to be simple to use and flexible to configure by supporting distinct levels of interaction. In this manner, beginner users can run an emulation with mostly default

[1]Available at: https://git.lasige.di.fc.ul.pt/fl/FADO

options, and as they become more proficient, they can tinker with and change the emulation's low-level workings.

The main contributions of this work are: (i) provides the design, architecture, and implementation of FADO, a platform that enables the creation and testing of new attacks and defenses in a consistent, realistic, and scalable way; (ii) the evaluation of a use case, based on a recently published network attack, that demonstrates the utility of the platform.

II. Related Work

This section reviews approaches used to test attacks and defenses (A&D) in FL and then presents some of the most studied techniques to perform these actions.

A. Frameworks

There is a standard limitation with the great majority of state-of-the-art implementations of A&D in FL: lack of realism in the testing environment. For example, even though FL is a distributed solution, many works that have been released as open-source operate as a single process, where communications are just function calls (with no network access) [7]–[15].

We are aware of one FL platform that permits adaptations to enable node attacks, either on the data or during training [14]. The drawback is that it has no support for more complex scenarios, like, for instance, compromising the packets exchanged among the FL nodes over secure channels (e.g., using Transport Layer Security (TLS) [16]). Since most production environments are expected to include such safeguards, it is crucial to consider settings where an attacker is constrained to carry out the same kind of activities as in the real world.

In FADO, we want to provide researchers a platform that, on the one hand, facilitates the experimentation on distinct (but reasonable) deployment scenarios; and, on the other hand, that simplifies comparisons by including implementations of existing approaches from the literature.

B. Attacks

We will briefly overview the three main categories of attacks in FL. As expected, a malicious actor may also orchestrate an attack combining these techniques to maximize the impact.

1) Model poisoning: The attacker modifies the training process conducted by the client(s). This includes tampering with the hyperparameters, the loss function [11], or the model updates directly [13], [17]. To get to a point where it can perform this attack, the adversary has to intrude on the client's device, which is highly dependent on the security configurations of the underlying operating system and applications. This usually leads to situations where the attack is performed by a small percentage of devices.

2) Data poisoning: The attacker is restricted to changing the training data in a subset of the clients. In practice, the dataset is tampered with to accomplish a certain goal [9], [18] – this can be done either by modifying or adding examples to the dataset. Data poisoning may be easier to perform than model poisoning, as in the latter case, the attacker needs to fully compromise the FL procedure in the device (instead of the storage).

3) Network-level: Network adversaries compromise FL by observing, altering, or dropping the packets exchanged between the server and the clients. Typically, to conduct this attack, the adversary has to take control of a node that intermediates the FL communications — e.g., a router or a Wi-Fi access point. Furthermore, one aspect that dramatically changes the practicability of network attacks is the applied protections, namely if messages are either secured or transferred in the clear and the geographic spread of the clients.

C. Defenses

Defenses usually aim to detect attacks and mitigate their effects on the final model. Although the literature has tried to categorize defenses accordingly, such a goal is challenging to attain as state-of-art safeguards frequently combine multiple mechanisms, and the same technique can be effective against both untargeted and targeted attacks. From now on, we will talk about the most prevalent approaches.

a) Robust aggregators: The most commonly used algorithm, *FedAvg*, employs the mean as the statistical operator to aggregate the client's gradients – which is susceptible to outliers [19]. To address this limitation, robust aggregation algorithms that consider other statistical operators were proposed: *median*, replaces the mean by choosing the value in the center of the distribution; *trimmed-mean*, filters extreme values below and above the data distribution by k-% and then calculates the mean of the remaining values [20]; and *geometric-median* [19], computes a value that represents the central tendency of the product of all model updates.

b) Norm Clipping: norm clipping was proposed to stop attacks because adversaries may increase the norm of their updates to enhance the impact [12]. Norm clipping ignores client updates above some pre-defined threshold value or restricts the (norm of the) update to a threshold value. This can introduce two limitations: a strong attacker might know the threshold and thus adjust the updates to evade the defense; a fixed threshold value can be escaped by conducting a distributed attack [7]. To address these issues, Guo et al. proposed a dynamic clipping approach that adapts the threshold during training [21]. In any case, in several realistic scenarios, clipping techniques have been shown to be adequate protections, as demonstrated by Shejwalkar et al. [22].

c) Differential Privacy: Sophisticated backdoor attacks are unlikely to be detected by some defenses as malicious model updates can be crafted not to deviate much from benign behavior [11]. For this reason, alternative solutions based on differential privacy (DP) were developed, consisting typically of adding random Gaussian noise $\mathcal{N}(0, \sigma^2)$ to model updates [23], resulting in a reduction of the influence of specific (malicious) data points. Such mechanism has been applied at the server to prevent the injection of backdoors [12], [24], [25], as well as at clients to avoid attacks whose objective is to infer information from the observed model updates [26].

III. Architecture and Components

The architecture and components of FADO are displayed in Figure 1. The architecture is divided into two parts, the Builder

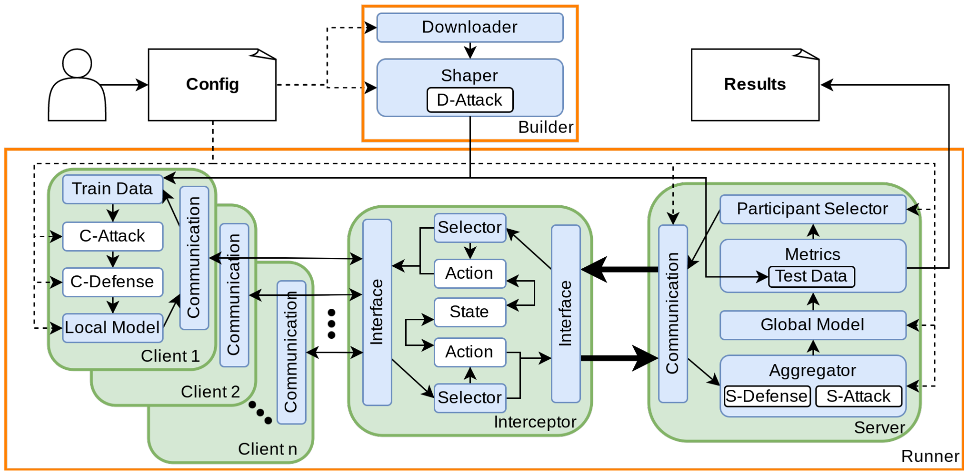

Fig. 1: *FADO Architecture*

and the Runner. The behavior of each of these components is configured by a central file supplied by the user.

A. Builder

The goal of the Builder is to prepare the datasets that FL consumes. Namely, it divides the examples through the various clients to allow the training of their local copy of the model, and it forwards to the server the validation/test data to support the evaluation of the global (aggregated) model.

To accomplish this task, the Builder follows a strict sequence of two steps: 1) downloads the data from a repository on the web or reads it from a file; 2) *shapes* the data to be ready for training/testing. Shaping accounts with a layer of processing that has to consider, for instance: the number of clients being emulated; a specific target class being evaluated; data distribution parameters; or an attack that the user wishes to perform on the examples. For this reason, since these steps can entail some level of complexity, the datasets we include in FADO already have their shaping processes implemented (see Section IV-A).

We provide a simple interface through abstraction so the user of FADO has the flexibility to integrate their datasets and attacks, but in practice, the two steps mentioned before are fully customizable. A user can implement its own downloader to use different datasets, or a shaper, modifying the properties (e.g., distribution) of an existing dataset.

B. Runner

The Runner is responsible for training the model according to the traditional FL protocol.

a) Server: This component is in charge of organizing the FL job. A set of clients is first chosen to participate in the next round of training, and then they are sent the initial global model that was generated. Next, the server waits to receive the local model updates produced by clients. Those updates are then aggregated to create a new version of the global model. Then the model is evaluated against a test dataset, and the results are written into a file. These steps are repeated for a specified number of rounds until the final model is generated.

All the above steps can be customized. For example, every client can be used in each round or just a subset; the aggregation procedure can apply defense (*S-Defense*) or attack (*S-Attack*) operations; the aggregator may wait for all responses or give up after a pre-determined timeout. All of these aspects can be modified through configuration options in the user file — some variants are already built in, while others can be achieved with user-defined implementations of the components. For example, it would be possible to emulate variations of the aggregation process; however, the current implementation does not support aggregators that require client-to-client communication (e.g., secure aggregation [27]).

b) Clients: Clients serve as workers in the FL job who aim to receive a global model from the server and train it with their local data to build a local model update. The update can then be modified to apply an attack (*C-Attack*) or a defense (*C-Defense*). Finally, the local model is sent back to the server. As with the server, the user-written configuration file can alter the specifics of how each client module behaves.

c) Interceptor: This component acts as a middle entity accessing the communications between the server and the clients. Therefore, it is capable of intercepting messages to further develop network attacks. Two *Selectors* decide which traffic is relevant to be processed (while the remaining packets pass through to minimize overheads).

The packets that are pertinent for an attack can be exploited using two types of actions. *Basic actions* may read, drop, delay, or forward a packet. *Complex actions* can modify the packet's content and send it afterward. These actions have access to a common state to track previously observed communications (e.g., current FL round number and list with clients' IDs in a particular round).

C. Configuration and Results

The behavior of all the components is determined by a set of configuration options specified in a user-provided file. This file is the interface where the user selects the parameters of the emulation — such as the number of clients, the dataset to be used, security mechanisms to be applied (attacks or defenses), and the percentage of clients affected by these, etc... As such, the file is the source of all the orchestration behind the Builder and Runner modules.

The emulation outputs the results that the user requests. Some of these are traditional metrics, such as the model's accuracy on the final test set, loss values during the learning process, or the accuracy of a particular class when presented with malicious examples. We include in our architecture a component, Metrics, responsible for aggregating the emulation results and storing them in a file.

IV. Implementation

A. Models and Datasets

To support diverse deep learning models and foster experimentation of novel NN architectures, FADO was integrated with two main frameworks used in this area, namely PyTorch and TensorFlow. Therefore, the tool can readily use existing models with minimal adaptations to the FL environment.

To facilitate the testing of the models, we are gradually porting commonly used datasets to the emulator. This task always involves a little bit of effort as a meaningful distribution of the examples has to be found, and often this requires understanding the particular characteristics of the data (e.g., movie comments from a given person have to go to the same client). Ultimately, the aim is to ensure that clients and the server get a reasonably sized dataset. Recently, we finished the integration of the LEAF benchmark [28], which brings six additional datasets fully tailored to FL.

B. Scale Emulations

The two main FL settings (cross-silo and cross-device) impose distinct requirements [6]. For example, in cross-silo, there are around 2–100 clients, while in cross-device, the numbers can go much higher (up to a few million). Consequently, to emulate different scenarios, FADO must support large numbers of clients while efficiently using the hardware resources (to make experiments cost-effective in practice). Some of the factors that lead to our current approach are:

- *Entity implementation:* The simplest solution would be to use a Docker container for each FADO entity (clients, server, and interceptor). This means an emulation with *n* clients would require *n+2* containers. After trying this first approach, we found that deploying more containers in a single (medium-sized) server rapidly leads to substantial CPU utilization, limiting scalability.
- *Network:* An alternative is hosting numerous clients (as separate processes) in a single container with multiple network interfaces. Although Docker networks provide this sort of support, this implies that to emulate *n* clients, we needed to create *n* docker networks. Through experimentation, we discovered that creating many docker networks also results in a significant CPU overhead.
- *GPU support:* We found out that a process that initializes PyTorch/TensorFlow would occupy in the order of 400 MB of GPU memory. This meant that to run emulations with only 100 clients in a server, we would use a minimum of 40.8 GB (*102*400 MB*) of GPU memory, which again imposes constraints on scalability.

The current solution is to create *n virtual* interfaces in a container that hosts the clients. Then, we bind each client socket to a specific interface, allowing the interceptor to distinguish communications belonging to different clients. This way, FADO can launch an emulation in 3 containers (clients/interceptor/server), minimizing the CPU overhead. To optimize the use of GPU memory, we host the clients within a single process, each corresponding to a concurrent thread. This way, FADO requires a minimum of 1.2 GB of GPU memory (400 MB per node), leaving plenty of space for data and model processing.

C. Networking and Traffic Interception

To enhance the realism of the emulations, FADO implements client-server communications, offering unprotected channels based on TCP and secure channels built on top of TLS. With regard to network attackers, one can conceive actors with very different capabilities in terms of access to traffic and actions that they might do on the packets. Therefore, FADO allows for the emulation of diverse attackers by forcing all data to pass through a centralized interceptor. Then, when acting, for example, as a weak attacker, the interceptor would only call the attacker code with the packets sent from the server to a particular client i, while when mimicking a stronger one, the code would observe all packets in both directions for a group of clients.

To implement the interceptor, it was necessary to guarantee network segmentation by splitting the clients and the server into two docker networks. The interceptor is located in a separate docker container, operating as a router connecting both networks and forwarding data between them. To accomplish the role of the Selector (see Fig. 1), we use `iptables` [29] rules to send the relevant FL traffic to two user-level packet queues. Packets can then be processed inside Python thanks to the `NetfilterQueue` [30]. Each packet is delivered to a user-definable Python object corresponding to the attacker. Here, basic actions such as dropping a packet can be carried out. Complex actions are implemented through an interface to `Scapy` [31], a tool for packet manipulation.

D. Attacks on the nodes

Node attacks can be conducted in multiple ways. For example, a compromised client can have its training process modified, its local model weights directly manipulated (model poisoning), or its dataset tampered with before training (data poisoning). Furthermore, an attacker that intrudes on the server can also replace the global model with a malicious one – in which the clients in the FL process trust blindly, as the server is the coordinator of the protocol.

FADO provides specific interfaces to implement data poisoning attacks. The changes to the examples will be applied at the Shaper level after the download of the raw data finishes. We choose to follow this method to decrease the Shaper's complexity and provide a considerable level of accessibility to the user. To implement a custom attack at the data level, the user creates a file with a function that receives a dataset as input and then outputs the modified data. A configuration file indicates to FADO the name of such a function so that it can be called at the appropriate moment (see Figure 2).

Fig. 2: *Example implementation of a data poisoning attack. The function `attack_data()` is applied before the client joins the FL process.*

```
# fado_config.yaml
...
attack_args:
  data:
    data_attack: custom_attack.py

# custom_attack.py
...
def attack_data(dataset):
  """ Poisoned dataset """
  poisoned_dataset = ...
  return poisoned_dataset
```

To support model poisoning attacks, we integrate mechanisms responsible for overriding specific aspects of the FL process. See further explanations in Appendix A.

E. Rapid experimentation

One of the key properties of FADO is its modularity and flexibility to run multiple emulations under different configurations in an accessible way. In FADO, each emulation is driven by a configuration file written in YAML. Appendix B describes this file in more detail through an example.

V. Use Case: Implementing NLAFL in FADO

Network-Level Adversaries in Federated Learning (NLAFL) is a recently proposed FL attack that aims to reduce the accuracy of a chosen target class in a classification task [8]. The attack first identifies the clients that contribute significantly to the learning of the target victim class, and then it drops their packets containing the model updates (thus preventing these clients from contributing to the global model training). This attack was also complemented with a model poison campaign to amplify the decrease in accuracy.

In greater detail, assuming the more realistic scenario where communications are secured, the adversary does the following: (1) it calculates the difference between the loss of the current and the previous rounds using a test dataset, to assess the model improvement on the accuracy of the target class on each FL round; then, it links each client who took part in that round to the improvement; (2) after a pre-determined number of rounds, the attacker computes the average of the improvements due to each client and chooses the clients with the highest values for packet dropping (as they have contributed more to the reduction of the loss value). After this point, and in the remaining rounds, the contribution estimation procedure is repeated to update the roster of clients to be dropped.

TABLE I: *Target class accuracy, in the no attack and packet-dropping scenarios. Accuracy in the original paper (NLAFL), with the code published by the authors but executed on our local environment (Local), and FADO's implementation (FADO). k is the number of clients with examples of the target class.*

k	Target Class	No Attack No Defense			Targeted Drop No Defense		
		NLAFL	Local	FADO	NLAFL	Local	FADO
EMNIST							
9	0	0.66	0.63	0.68	0.52	0.46	0.53
	1	0.92	0.92	0.93	0.76	0.73	0.79
	9	0.56	0.54	0.60	0.46	0.43	0.51
12	0	0.78	0.79	0.77	0.69	0.64	0.73
	1	0.95	0.95	0.95	0.91	0.85	0.92
	9	0.67	0.69	0.70	0.56	0.55	0.61
15	0	0.80	0.85	0.81	0.76	0.82	0.79
	1	0.96	0.96	0.96	0.94	0.95	0.95
	9	0.75	0.73	0.75	0.58	0.57	0.63
FashionMNIST							
15	0	0.47	0.47	0.59	0.39	0.40	0.43
	1	0.95	0.95	0.95	0.94	0.94	0.94
	9	0.87	0.87	0.90	0.82	0.82	0.84
DBPedia							
15	0	0.54	0.54	0.64	0.12	0.10	0.14
	1	0.87	0.87	0.90	0.22	0.31	0.32
	9	0.60	0.60	0.67	0.12	0.15	0.16

In the original NLAFL paper, all experiments were carried out on a single process without network support. Therefore, we decided to take NLAFL as a use case to explain the steps needed to move the attack logic to FADO and observe how the evaluation in FADO behaves.

a) Datasets: In the original implementation, datasets are downloaded and then divided among the clients. Three datasets were tested, namely EMNIST [32], FashionMNIST [33], and DBPedia [34]. Dataset division is accomplished by separating clients into two groups, depending on the examples they will use during training. The first group gets many examples from the target victim class, and the remaining from the other classes. The second group only receives examples from the other classes. In both cases, the sampling is produced using a Dirichlet distribution.

To implement this logic in FADO, we copied the open-source code supplied by the authors to create a new *Downloader* and *Shaper* extensions, which will perform the download and distribution steps, respectively. Additional code was also added to generate files to be given to clients and server.

b) Models: Two similar convolutional models were used with the EMNIST and FashionMNIST datasets — containing two 2D convolution layers plus two linear layers. For DBPedia, the model comprised a pre-trained GloVe embedding followed by two 1D convolution layers and two linear layers. The EMNIST model was trained for 105 rounds, while for FashionMNIST and DBPedia training lasted for 305 rounds.

The migration of the models to FADO was accomplished by programming five python methods[2]. We developed two versions of the code to try out the platform, one based on TensorFlow and the other with PyTorch.

c) Network attack: NLAFL code was constructed as a simulation, and consequently, specific details could be skipped

[2]Methods: *initialize*, *get_parameters*, *set_parameters*, *train*, and *evaluate*

(e.g., inferring which clients participate in each round). In FADO, it was necessary to address these issues, but the actual development required little effort due to the built-in services provided. Our implementation employs secure channels, which makes the attack considerably more challenging.

Like in the original paper, we ensured the attacker had access to the global model in all rounds. This was achieved by providing an endpoint on the server that the adversary can query to get a copy of the model. This allows the detection when a new round is starting as model parameters change.

The adversary code can inspect all traffic by setting appropriate Selectors in the interceptor. By looking at the IP addresses, the adversary can infer which clients are involved in the training of the round. This information can be stored in the state module, and through it, a client contribution can be estimated as in the original proposal. After the attacker discovers which clients should have their model updates dropped, the Actions module can be instructed to execute such a task.

d) Poisoning attack and Clipping defense: The attack is made in two steps: (i) a set of attacker clients gets a dataset similar to the clients with access to the target class; in those clients, the examples of the target class have their label changed to a distinct pre-selected class (corresponds to a *data-poisoning attack*); (ii) during model training, parameter updates of the poisoned clients are upsampled to increase their influence in the next version of the global model (corresponds to a *model replacement attack*). The defense against the poisoning attack is attained during the aggregation process by clipping the norm of the received updates from the clients.

In FADO, the *D-Attack* and *C-Attack* modules were used to do the data poisoning and model poisoning, while the *S-Defense* module was used to create the clipping defense.

A. Evaluation

We implemented four test configurations to evaluate the performance of the attack in FADO and compare it with the results of the original paper: (i) training with no attacks or defenses; (ii) training solely with the packet drop attack; (iii) training with the drop and poison attacks; and, (iv) training with the attacks and the clipping defense. We compare the original results published in the NLAFL paper for each of these settings with the ones collected while running the author's code in our environment. In addition, we got the performance results with our FADO implementation. The table values are the average of 5 runs with different seed values.

Tables I and II present the results for the attack-free and network attack scenarios ((i) and (ii)). The accuracy values for the target class were obtained by averaging the observed accuracies in the last five rounds of training (between rounds *n* and *n+5*). The EMNIST model was trained for $n = 100$ rounds, while the other two models were trained for $n = 300$ rounds, given their increased complexity. In the EMNIST case, we experimented with different numbers of clients with the target class ($k = 9, 12$, and 15) while the other two kept the same number ($k = 15$). The main observations are:

TABLE II: *Target class accuracy, when considering a targeted dropping + node poison without and with a clipping defense. Remaining parameters are similar to Table I.*

k	Target Class	Targeted Drop + Poison					
		No Defense			Clip		
		NLAFL	Local	FADO	NLAFL	Local	FADO
EMNIST							
9	0	0.00	0.00	0.00	0.40	0.36	0.49
	1	0.00	0.00	0.00	0.52	0.41	0.60
	9	0.00	0.01	0.00	0.25	0.21	0.37
12	0	0.00	0.00	0.00	0.36	0.32	0.51
	1	0.00	0.00	0.00	0.43	0.37	0.59
	9	0.01	0.01	0.00	0.27	0.26	0.43
15	0	0.00	0.00	0.00	0.44	0.45	0.61
	1	0.00	0.00	0.00	0.48	0.42	0.56
	9	0.01	0.00	0.00	0.35	0.30	0.34
FashionMNIST							
15	0	0.02	0.02	0.00	0.36	0.31	0.35
	1	0.03	0.05	0.03	0.81	0.77	0.84
	9	0.05	0.04	0.02	0.59	0.60	0.62
DBPedia							
15	0	0.00	0.00	0.00	0.00	0.00	0.00
	1	0.00	0.00	0.00	0.00	0.00	0.00
	9	0.00	0.00	0.00	0.00	0.00	0.00

- The results collected with FADO lead to similar conclusions as the original implementation, indicating that the platform can reproduce the impact of attack and defense techniques in FL.
- The network attack is less effective when the target class examples are distributed through more clients.
- The effectiveness of the network attack depends on the target class and dataset as there are scenarios where the accuracy reduction is strong (e.g., class 9 in DBPedia) while in others is less visible (e.g., class 0 in EMNIST with $k = 15$).
- The accuracy decrease is highly significant when the network and node attacks are performed simultaneously.
- The clipping defense can reduce the attack impact in some datasets (e.g., EMNIST and FashionMNIST) but is unsuccessful in others (e.g., DBPedia).

VI. Conclusion

The paper presents a platform that enables the experimentation of security mechanisms in federated learning under realistic scenarios. Users can develop novel attack and defense techniques in FADO and then observe their impact in the training of deep-learning models. To demonstrate the platform's utility, we created a use case based on a recently published work, which includes network and node attacks and defenses. Our results show that FADO can accurately reproduce the original paper's conclusions.

VII. Acknowledgements

This work was partially supported by the European Union's Horizon 2020 research and innovation programme under grant agreement No 871259 (ADMORPH project), the ITEA3 European through the XIVT project (I3C4-17039/FEDER-039238), and by FCT through the LASIGE research unit (UIDB/00408/2020 & UIDP/00408/2020).

References

[1] H. B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. Y. Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Proc. International Conference on Artificial Intelligence and Statistics*, vol. 54, pp. 1273–1282, 2017.

[2] J. Konečný, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, and D. Bacon, "Federated learning: Strategies for improving communication efficiency," 2016, arXiv:1610.05492.

[3] T. Yang, G. Andrew, H. Eichner, H. Sun, W. Li, N. Kong, D. Ramage, and F. Beaufays, "Applied federated learning: Improving google keyboard query suggestions," *arXiv:1812.02903*, 2018.

[4] A. M. Elbir, B. Soner, S. Çöleri, D. Gündüz, and M. Bennis, "Federated learning in vehicular networks," in *Proceedings of the IEEE International Mediterranean Conference on Communications and Networking*, pp. 72–77, 2022.

[5] M. Paulik, M. Seigel, H. Mason, D. Telaar, J. Kluivers, R. van Dalen, C. W. Lau, L. Carlson, F. Granqvist, C. Vandevelde, S. Agarwal, J. Freudiger, A. Byde, A. Bhowmick, G. Kapoor, S. Beaumont, A. Cahill, D. Hughes, O. Javidbakht, F. Dong, R. Rishi, and S. Hung, "Federated evaluation and tuning for on-device personalization: System design and applications," 2021, arXiv:2102.08503.

[6] P. Kairouz, H. B. McMahan, B. Avent, A. Bellet, M. Bennis, A. N. Bhagoji, K. Bonawit, Z. Charles, G. Cormode, R. Cummings, R. G. L. D'Oliveira, H. Eichner, S. El Rouayheb, D. Evans, J. Gardner, Z. Garrett, A. Gascón, B. Ghazi, P. B. Gibbons, M. Gruteser, Z. Harchaoui, C. He, L. He, Z. Huo, B. Hutchinson, J. Hsu, M. Jaggi, T. Javidi, G. Joshi, M. Khodak, J. Konecný, A. Korolova, F. Koushanfar, S. Koyejo, T. Lepoint, Y. Liu, P. Mittal, M. Mohri, R. Nock, A. Özgür, R. Pagh, H. Qi, D. Ramage, R. Raskar, M. Raykova, D. Song, W. Song, S. U. Stich, Z. Sun, A. Theertha Suresh, F. Tramèr, P. Vepakomma, J. Wang, L. Xiong, Z. Xu, Q. Yang, F. X. Yu, H. Yu, and S. Zhao, "Advances and open problems in federated learning," *Foundations and Trends in Machine Learning*, vol. 14, no. 1–2, pp. 1–210, 2021.

[7] C. Xie, K. Huang, P.-Y. Chen, and B. Li, "DBA: Distributed Backdoor Attacks against Federated Learning," in *Proc. International Conference on Learning Representations*, 2020.

[8] G. Severi, M. Jagielski, G. Yar, Y. Wang, A. Oprea, and C. Nita-Rotaru, "Network-level adversaries in federated learning," in *Proc. IEEE Conference on Communications and Network Security*, pp. 19–27, 2022.

[9] V. Tolpegin, S. Truex, M. E. Gursoy, and L. Liu, "Data poisoning attacks against federated learning systems," in *Proc. European Symposium On Research In Computer Security*, pp. 480–501, 2020.

[10] H. Wang, K. Sreenivasan, S. Rajput, H. Vishwakarma, S. Agarwal, J. Sohn, K. Lee, and D. S. Papailiopoulos, "Attack of the tails: Yes, you really can backdoor federated learning," in *Proc. International Conference on Neural Information Processing Systems*, 2020.

[11] E. Bagdasaryan, A. Veit, Y. Hua, D. Estrin, and V. Shmatikov, "How to backdoor federated learning," in *Proc. International Conference on Artificial Intelligence and Statistics*, vol. 108, pp. 2938–2948, 2020.

[12] Z. Sun, P. Kairouz, A. T. Suresh, and H. B. McMahan, "Can you really backdoor federated learning?," 2019, arXiv:1911.07963.

[13] G. Baruch, M. Baruch, and Y. Goldberg, "A little is enough: Circumventing defenses for distributed learning," in *Proc. Advances in Neural Information Processing Systems*, vol. 32, pp. 8632–8642, 2019.

[14] C. He, S. Li, J. So, X. Zeng, M. Zhang, H. Wang, X. Wang, P. Vepakomma, A. Singh, H. Qiu, X. Zhu, J. Wang, L. Shen, P. Zhao, Y. Kang, Y. Liu, R. Raskar, Q. Yang, M. Annavaram, and S. Avestimehr, "FedML: A research library and benchmark for federated machine learning," 2020, arXiv:2007.13518.

[15] Z. Zhang, A. Panda, L. Song, Y. Yang, M. Mahoney, P. Mittal, R. Kannan, and J. Gonzalez, "Neurotoxin: Durable backdoors in federated learning," in *Proc. International Conference on Machine Learning*, pp. 26429–26446, 2022.

[16] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3." https://www.rfc-editor.org/info/rfc8446, 2018.

[17] V. Shejwalkar and A. Houmansadr, "Manipulating the byzantine: Optimizing model poisoning attacks and defenses for federated learning," in *Proceedings of the Network and Distributed System Security Symposium*, pp. 18–37, 2021.

[18] X. Chen, C. Liu, B. Li, K. Lu, and D. Song, "Targeted backdoor attacks on deep learning systems using data poisoning," 2017, arXiv:1712.05526.

[19] K. Pillutla, S. M. Kakade, and Z. Harchaoui, "Robust aggregation for federated learning," *IEEE Transactions on Signal Processing*, vol. 70, pp. 1142–1154, 2022.

[20] D. Yin, Y. Chen, R. Kannan, and P. Bartlett, "Byzantine-robust distributed learning: Towards optimal statistical rates," in *Proceedings of the International Conference on Machine Learning*, pp. 5650–5659, 2018.

[21] Y. Guo, Q. Wang, T. Ji, X. Wang, and P. Li, "Resisting distributed backdoor attacks in federated learning: A dynamic norm clipping approach," in *Proceedings of the IEEE International Conference on Big Data*, pp. 1172–1182, 2021.

[22] V. Shejwalkar, A. Houmansadr, P. Kairouz, and D. Ramage, "Back to the drawing board: A critical evaluation of poisoning attacks on production federated learning," in *Proc. Symposium on Security and Privacy*, pp. 1354–1371, 2022.

[23] M. Abadi, A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang, "Deep learning with differential privacy," pp. 308–318, 2016.

[24] C. Xie, M. Chen, P.-Y. Chen, and B. Li, "Crfl: Certifiably robust federated learning against backdoor attacks," in *Proceedings of the International Conference on Machine Learning*, pp. 11372–11382, 2021.

[25] T. D. Nguyen, P. Rieger, H. Chen, H. Yalame, H. Möllering, H. Fereidooni, S. Marchal, M. Miettinen, A. Mirhoseini, S. Zeitouni, *et al.*, "FLAME: Taming backdoors in federated learning," in *Proceedings of the USENIX Security Symposium*, pp. 1415–1432, 2022.

[26] K. Wei, J. Li, M. Ding, C. Ma, H. H. Yang, F. Farokhi, S. Jin, T. Q. S. Quek, and H. V. Poor, "Federated learning with differential privacy: Algorithms and performance analysis," *Proc. IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 3454–3469, 2020.

[27] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth, "Practical secure aggregation for privacy-preserving machine learning," in *Proc. ACM SIGSAC Conference on Computer and Communications Security*, pp. 1175—-1191, 2017.

[28] S. Caldas, S. M. K. Duddu, P. Wu, T. Li, J. Konečný, H. B. McMahan, V. Smith, and A. Talwalkar, "LEAF: A benchmark for federated settings," in *Workshop on Federated Learning for Data Privacy and Confidentiality*, 2019.

[29] R. Russell, "iptables." https://linux.die.net/man/8/iptables. Accessed: 2023-03-04.

[30] M. Fox, "Netfilterqueue." https://pypi.org/project/NetfilterQueue. Accessed: 2023-03-04.

[31] P. Biondi, "Scapy." https://scapy.net. Accessed: 2023-03-04.

[32] G. Cohen, S. Afshar, J. Tapson, and A. van Schaik, "Emnist: Extending mnist to handwritten letters," in *Proc. International Joint Conference on Neural Networks*, pp. 2921–2926, 2017.

[33] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms," 2017, arXiv:1708.07747.

[34] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives, "Dbpedia: A nucleus for a web of open data," in *The Semantic Web* (K. Aberer, K.-S. Choi, N. Noy, D. Allemang, K.-I. Lee, L. Nixon, J. Golbeck, P. Mika, D. Maynard, R. Mizoguchi, G. Schreiber, and P. Cudré-Mauroux, eds.), pp. 722–735, 2007.

Appendix

A. Performing Model Poisoning Attacks

When a model poisoning attack is specified in the configuration file, the implementation of that attack is injected into the compromised clients, and consequently, the behavior of these clients is changed. More specifically, the training process can be overwritten, or the model updates directly manipulated [13], [17] to perform untargeted attacks. When it comes to the server, FADO also has mechanisms to change the global model in any way the user desires, based on the specified implementation.

Fig. 3: *Example model poisoning attack. The implementation of each method is not mutually exclusive.*

```
# fado_config.yaml
...
attack_args:
  model:
    model_attack: custom_attack.py

# custom_attack.py
...
def attack_training(dataset):
    """ Attack training process ... """
    model_updates = ...
    return model_updates
def attack_after_training(model_updates):
    """ Attack model updates """
    poisoned_model_updates = ...
    return poisoned_model_updates
```

When implementing a model poisoning attack, the user has two interfaces at its disposal: one to override the training process completely [11], [12], and the other to directly manipulate the model updates that were the result of the training process [13]. In Figure 3, we show a high-level view of the implementation of a model poisoning attack: 1) specify in the $fado\_config.yaml$ file the name of the function (and the name of the file) making the attack; 2) the file $custom\_attack.py$ with the code of the adversarial functions.

B. Configuration File

The configuration file that drives the emulation behavior (as explained in Section III) is written in YAML, in which each key-value pair defines a particular setting. Besides a value, a user can also indicate a list of values in order to instruct FADO to run multiple emulations. For example, if a user sets three of the keys with three lists of size 4 (four different settings), then FADO will perform 4^3 different emulations using the product of all varying settings. By relying on this mechanism, the user can rapidly evaluate the effectiveness of an attack under certain conditions (e.g., reduced vs. a high number of clients).

Fig. 4 shows that the user is able to customize a great number of aspects of the emulation: in the FL training procedure (number of rounds, aggregator properties, number of clients, clients per round, ...); in the client tasks (batch size, learning rate, ...); in the implementation of security mechanisms (attacks and defenses to apply and their properties); in the training dataset specifications (e.g., which and how to distribute among clients); and finally, in the network communication properties (encrypted or not).

In addition, when specifying custom implementations as described in section A, a user can add new key-value pairs and then use them inside the code — FADO makes this information visible through a dictionary.

Fig. 4: *Example of a configuration file following the structure and key-value pairs accepted by FADO.*

```
general:
    random_seed: [0,2,4,6,8]
    use_gpu: false

fl_training_process:
    rounds: 105
    aggregator: mean
    agg_learning_rate: 0.25
    client_optimizer: sgd
    number_clients: 100
    clients_per_round: 10
    participant_selector: random

client_training_process:
    engine: tensorflow
    model: nlafl_emnist_tf
    batch_size: 32
    epochs: 2
    learning_rate: 0.1

attack_args:
    malicious_clients: 10
    target_class: [0,1,9]
    network:
        network_attack: nlafl
        drop_count_multiplier: 2
        drop_start: 30
    model:
        model_attack_name: 'nlafl_poison'
        poison_count_multiplier: 2
        boost_factor: 10.0

defense_args:
    server:
        server_defense_name: clip
        clip_norm: 1

dataset_spec:
    dataset: nlafl_emnist
    data_distribution: niid
    num_classes: 10

communication:
    encrypt_comm: true
```