Demo Abstract: Blades: A Unified Benchmark Suite for Byzantine-Resilient in Federated Learning

Shenghui Li*, Edith C.-H. Ngai†, Fanghua Ye‡, Li Ju*, Tianru Zhang*, Thiemo Voigt*§
*Uppsala University, Uppsala, Sweden. *{shenghui.li,li.ju,tianru.zhang}@it.uu.se*
†The University of Hong Kong, Hong Kong, China. *chngai@eee.hku.hk*
‡University College London, London, UK. *fanghua.ye.19@ucl.ac.uk*
§Research Institutes of Sweden, Stockholm, Sweden. *thiemo.voigt@angstrom.uu.se*
https://github.com/lishenghui/blades

***Abstract*—Federated learning (FL) facilitates distributed training across different IoT and edge devices, safeguarding the privacy of their data. The inherently distributed nature of FL introduces vulnerabilities, especially from adversarial devices aiming to skew local updates to their desire. Despite the plethora of research focusing on Byzantine-resilient FL, the academic community has yet to establish a comprehensive benchmark suite, pivotal for the assessment and comparison of different techniques. This demonstration presents Blades, a scalable, extensible, and easily configurable benchmark suite that supports researchers and developers in efficiently implementing and validating strategies against baseline algorithms in Byzantine-resilient FL.**

***Index Terms*—Byzantine attacks, distributed learning, federated learning, IoT, neural networks, robustness**

I. Introduction

Federated learning (FL) [1] allows for collaborative machine learning model construction by leveraging distributed data across a diverse range of client devices, from IoT and edge devices to mobile phones and computers, without disclosing their private data to a central server. Due to the distributed characteristic of optimization, FL is vulnerable to Byzantine failures [2], wherein certain participants may deviate from the prescribed update protocol and upload arbitrary parameters to the central server. In this context, typical FL algorithms like FedAvg [1], which compute the sample mean of client updates for global model aggregation, can be significantly skewed by a single Byzantine client [2]. The server thus requires Byzantine-resilient solutions to defend against malicious clients.

In recent years, the field of FL has seen the emergence of various Byzantine-resilient approaches. They aim to protect distributed optimization from Byzantine clients and assure the performance of the learned models. For instance, robust aggregation rules (AGRs) are widely used to estimate the global update from a collection of local updates while mitigating the impact of malicious behaviors. Typical AGRs include GeoMed [3], and Median [4]. Meanwhile, different attack strategies are emerging, striving to circumvent defense strategies [2]. For instance, the A Little Is Enough (ALIE) attack can bypass various AGRs by taking advantage of the empirical variance between clients' updates if such a variance is high enough, especially when the local datasets are not independent and identically distributed (non-IID) [2].

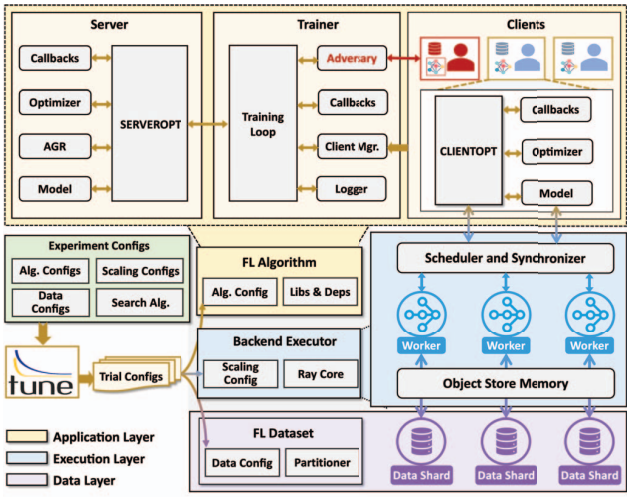

Fig. 1: The three-layer architecture of Blades.

This demonstration presents *Blades* [5], our open-source benchmark suite to address the research gap concerning attack and defense problems in FL. Blades contains built-in implementations of representative attack and defense strategies and offers usability, extensibility, and scalability that facilitate the implementation and expansion of novel attack and defense techniques.

II. Blades Overview

The architecture of Blades consists of three layers: the application layer, the execution layer, and the data layer. Layers are organized as shown in Fig. 1. The application layer serves as the interface for FL tasks, allowing for straightforward configuration and integration of a range of FL-related functionalities and features. The execution layer is built upon a scalable framework Ray[1], dedicated to distributed resource management and local model training. The data layer is responsible for federated dataset partition and preprocessing in the distributed environment.

[1] www.ray.io

979-8-3503-7025-6/24/$31.00 ©2024 IEEE
DOI 10.1109/IoTDI61053.2024.00030

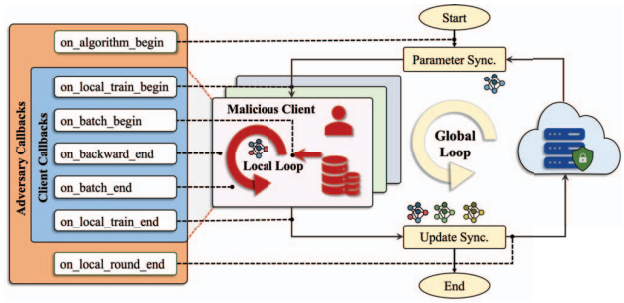

Fig. 2: The pipeline for attack implementation.

III. Demonstration

We will demonstrate the usability of Blades by rapidly implementing attack and defense strategies.

1) Implementing Attacks: Adversarial attackers may perform manipulation before or after specific time points. For instance, LabelFlipping attacks are typically executed at the beginning of batch forward propagation, while SignFlipping attacks are carried out immediately after backpropagation. As depicted in Fig. 2, Blades provides a unified pipeline integrated with a callback mechanism to perform actions at various stages of training. This design offers extensibility to facilitate customization, where the minimal pipeline focuses on repetitive local training and server-side optimization while the malicious behaviors are defined through callbacks. Elementary attacks (*e.g.,* LabelFlipping and SignFlipping), operate solely on local data or model parameters. Therefore, their implementation is streamlined by registering specific behaviors to client objects, enabling concurrent execution across multiple workers. As an illustrative instance, the upper panel of Fig. 3 shows a code snippet that exemplifies the implementation of a LabelFlipping attack on a classification task encompassing 10 classes. Through a straightforward customization of the "`on_batch_begin()`" callback method, users can easily modify the labels from class i to $9 - i$ during local training.

Sophisticated attacks, such as collusion, necessitate clients to exchange information and coordinate their actions. Distinguished from client callbacks, adversary callbacks are executed within the driver program and allow convenient access to various system components and their states. This design simplifies the process of acquiring knowledge for high-level attacks. Fig. 2 also shows two essential adversary callbacks, specifically "`on_algorithm_begin()`" and "`on_local_round_end()`". The former is triggered at the start of the algorithm to initialize the adversary and set up client callbacks, while the latter occurs after a local round and allows for the modification of collected updates from malicious clients before proceeding to server-side optimization and defense. The lower panel of Fig. 3 shows an example of implementing the ALIE attack using adversary callback.

2) Implementing Defenses: Defenses in the context of our study stem from multiple facets, posing challenges to the establishment of a standardized pipeline akin to the one employed for

```python
from blades.clients import ClientCallback
class LabelFlipCallback(ClientCallback):
    def on_batch_begin(self, data, target):
        # Return input data with flipped labels for the current batch (assuming 10 labels).
        return data, 9 - target
```

```python
from blades.adversary import AdversaryCallback
class ALIECallback(AdversaryCallback):
    def on_local_round_end(self, trainer):
        # Compute the dimensional mean and std over client updates, then generate malicious updates by adding std to the mean.
        updates = trainer.get_updates()
        mean = updates.mean(dim=0)
        std = updates.std(dim=0)
        updates = mean + std
        trainer.save_malicious_updates(updates)
```

Fig. 3: Illustration of our callback mechanism used to simulate LabelFlipping (upper) and ALIE (lower) attacks.

attacks. Nevertheless, certain indispensable steps are involved in this process, namely update aggregation and global model optimization, although the specific methodology for each step may vary. The update aggregation step combines the locally collected updates, while the global model optimization step performs an optimization procedure based on the aggregated result.

As such, Blades introduces a foundational abstraction of the server entity, encompassing essential components including global model, AGR, and optimizer. This architecture permits the extension of functionalities through the utilization of sub-classing, thereby facilitating the integration of advanced features. It is noteworthy to mention that even in the minimal server implementation, we offer a configurable SGD optimizer and a variety of pre-defined AGRs. Furthermore, all the components are modularized and inheritable, allowing plug-and-play of different configurations. We believe our designs simplify the process of generating benchmark results with minimal effort.

References

[1] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Artificial intelligence and statistics*. PMLR, 2017.

[2] S. Li, E. C.-H. Ngai, and T. Voigt, "An experimental study of byzantine-robust aggregation schemes in federated learning," *IEEE Transactions on Big Data*, 2023.

[3] Y. Chen, L. Su, and J. Xu, "Distributed statistical machine learning in adversarial settings: Byzantine gradient descent," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2017.

[4] D. Yin, Y. Chen, R. Kannan, and P. Bartlett, "Byzantine-robust distributed learning: Towards optimal statistical rates," in *International Conference on Machine Learning*. PMLR, 2018.

[5] S. Li, E. Ngai, F. Ye, L. Ju, T. Zhang, and T. Voigt, "Blades: A unified benchmark suite for byzantine attacks and defenses in federated learning," in *2024 IEEE/ACM Ninth International Conference on Internet-of-Things Design and Implementation (IoTDI)*, 2024.