[22bit005@mepcolinux ~]$cat novalty.txt

server:

import os

os.environ['TF\_CPP\_MIN\_LOG\_LEVEL'] = '2'

import socket

import numpy as np

from sklearn.cluster import OPTICS, KMeans, HDBSCAN

from sklearn.decomposition import PCA

import tensorflow as tf

from scipy.stats import dirichlet

import random

# Load datasets

(x\_train\_mnist, y\_train\_mnist), (x\_test\_mnist, y\_test\_mnist) = tf.keras.datasets.mnist.load\_data()

x\_train\_mnist = x\_train\_mnist.reshape(-1, 28\*28).astype(np.float32) / 255.0

x\_test\_mnist = x\_test\_mnist.reshape(-1, 28\*28).astype(np.float32) / 255.0

(x\_train\_fmnist, y\_train\_fmnist), (x\_test\_fmnist, y\_test\_fmnist) = tf.keras.datasets.fashion\_mnist.load\_data()

x\_train\_fmnist = x\_train\_fmnist.reshape(-1, 28\*28).astype(np.float32) / 255.0

x\_test\_fmnist = x\_test\_fmnist.reshape(-1, 28\*28).astype(np.float32) / 255.0

(x\_train\_cifar, y\_train\_cifar), (x\_test\_cifar, y\_test\_cifar) = tf.keras.datasets.cifar10.load\_data()

x\_train\_cifar = x\_train\_cifar.reshape(-1, 32\*32\*3).astype(np.float32) / 255.0

x\_test\_cifar = x\_test\_cifar.reshape(-1, 32\*32\*3).astype(np.float32) / 255.0

y\_train\_cifar = y\_train\_cifar.flatten()

y\_test\_cifar = y\_test\_cifar.flatten()

datasets = {

'MNIST': (x\_train\_mnist, y\_train\_mnist, x\_test\_mnist, y\_test\_mnist, 784),

'FMNIST': (x\_train\_fmnist, y\_train\_fmnist, x\_test\_fmnist, y\_test\_fmnist, 784),

'CIFAR10': (x\_train\_cifar, y\_train\_cifar, x\_test\_cifar, y\_test\_cifar, 3072)

}

def create\_model(input\_shape):

model = tf.keras.Sequential([

tf.keras.layers.Input(shape=(input\_shape,)),

tf.keras.layers.Dense(128, activation='relu'),

tf.keras.layers.Dense(10, activation='softmax')

])

model.compile(optimizer='sgd', loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

return model

def secret\_share(data, n\_parties=2):

shares = [np.random.uniform(-1000, 1000, size=data.shape).astype(np.float32) for \_ in range(n\_parties - 1)]

shares.append(data - sum(shares))

return shares

def reconstruct\_shares(shares):

return sum(shares)

def amsparse(delta\_w, layer\_shapes):

amsparse\_updates = []

start = 0

for shape in layer\_shapes:

size = np.prod(shape)

layer\_delta = delta\_w[start:start + size].reshape(shape)

layer\_max = np.max(np.abs(layer\_delta), axis=tuple(range(len(shape) - 1)))

signs = np.sign(layer\_delta)

amsparse\_layer = signs \* layer\_max

amsparse\_updates.append(amsparse\_layer.flatten())

start += size

return np.concatenate(amsparse\_updates)

def auto\_optics(updates):

if len(updates) < 2:

return np.arange(len(updates))

try:

pca = PCA(n\_components=min(50, updates.shape[1]))

updates\_reduced = pca.fit\_transform(updates)

dists = np.linalg.norm(updates\_reduced - updates\_reduced.mean(axis=0), axis=1)

k = min(max(2, len(updates) // 3), len(updates) - 1)

sorted\_dists = np.sort(dists)[:k]

eps = np.median(sorted\_dists) \* 2.0

clustering = OPTICS(min\_samples=k, eps=eps, metric='euclidean').fit(updates\_reduced)

labels = clustering.labels\_

majority\_group = np.where(labels != -1)[0]

return majority\_group if len(majority\_group) > 0 else np.arange(len(updates))

except:

return np.arange(len(updates))

def adaptive\_clipping(updates, benign\_indices):

if len(updates) == 0 or len(benign\_indices) == 0:

return np.zeros\_like(updates[0]) if len(updates) > 0 else np.array([])

norms = np.linalg.norm(updates, axis=1)

rho = np.median(norms[benign\_indices]) if len(benign\_indices) > 0 else 1.0

clipped\_updates = []

for i in benign\_indices:

update = updates[i]

norm = norms[i]

clipped = update \* min(1, rho / norm) if norm > 0 else update

clipped\_updates.append(clipped)

return np.array(clipped\_updates) if clipped\_updates else np.zeros\_like(updates[0])

def priv\_shuffling(updates\_shares, n\_clients):

pi\_0 = np.random.permutation(n\_clients)

pi\_1 = np.random.permutation(n\_clients)

shuffled\_shares = [[updates\_shares[pi\_0[i]][0], updates\_shares[pi\_1[i]][1]] for i in range(n\_clients)]

return shuffled\_shares

def priv\_selection(shuffled\_shares, m):

norms\_shares = [secret\_share(np.linalg.norm(reconstruct\_shares(shares))) for shares in shuffled\_shares]

norms = [reconstruct\_shares(ns) for ns in norms\_shares]

sorted\_indices = np.argsort(norms)[:m]

return [shuffled\_shares[i] for i in sorted\_indices]

def priv\_optics(updates\_shares):

updates = np.array([reconstruct\_shares(shares) for shares in updates\_shares])

if len(updates) < 2:

return secret\_share(np.arange(len(updates)))

try:

pca = PCA(n\_components=min(50, updates.shape[1]))

updates\_reduced = pca.fit\_transform(updates)

dists = np.linalg.norm(updates\_reduced - updates\_reduced.mean(axis=0), axis=1)

k = min(max(2, len(updates) // 3), len(updates) - 1)

sorted\_dists = np.sort(dists)[:k]

eps = np.median(sorted\_dists) \* 2.0

clustering = OPTICS(min\_samples=k, eps=eps, metric='euclidean').fit(updates\_reduced)

labels = clustering.labels\_

benign\_indices = np.where(labels != -1)[0]

return secret\_share(benign\_indices if len(benign\_indices) > 0 else np.arange(len(updates)))

except:

return secret\_share(np.arange(len(updates)))

def priv\_clipping(updates\_shares, benign\_indices\_shares):

updates = np.array([reconstruct\_shares(shares) for shares in updates\_shares])

benign\_indices = reconstruct\_shares(benign\_indices\_shares).astype(int)

if len(benign\_indices) == 0:

return updates\_shares

norms = np.linalg.norm(updates, axis=1)

rho\_shares = secret\_share(np.median(norms[benign\_indices]) if len(benign\_indices) > 0 else 1.0)

clipped\_shares = []

for i in benign\_indices:

update\_shares = updates\_shares[i]

norm\_shares = secret\_share(norms[i])

rho\_val = reconstruct\_shares(rho\_shares)

norm\_val = reconstruct\_shares(norm\_shares)

factor = min(1, rho\_val / norm\_val) if norm\_val > 0 else 1.0

clipped\_shares.append([share \* factor for share in update\_shares])

return clipped\_shares if clipped\_shares else updates\_shares

def priv\_aggregation(clipped\_shares):

if not clipped\_shares:

return secret\_share(np.zeros\_like(reconstruct\_shares(clipped\_shares[0])))

clipped\_updates = np.array([reconstruct\_shares(shares) for shares in clipped\_shares])

return secret\_share(np.mean(clipped\_updates, axis=0))

def fedavg(updates):

return np.mean(updates, axis=0) if len(updates) > 0 else np.zeros\_like(updates[0])

def median(updates):

return np.median(updates, axis=0) if len(updates) > 0 else np.zeros\_like(updates[0])

def trimmed\_mean(updates, trim\_ratio=0.1):

if len(updates) < 3:

return np.mean(updates, axis=0) if len(updates) > 0 else np.zeros\_like(updates[0])

n\_trim = min(int(len(updates) \* trim\_ratio), len(updates) // 2 - 1)

if n\_trim == 0:

return np.mean(updates, axis=0)

sorted\_updates = np.sort(updates, axis=0)

trimmed = sorted\_updates[n\_trim:-n\_trim]

return np.mean(trimmed, axis=0)

def fedcc(updates):

if len(updates) < 2:

return np.mean(updates, axis=0) if len(updates) > 0 else np.zeros\_like(updates[0])

try:

pca = PCA(n\_components=min(50, updates.shape[1]))

updates\_reduced = pca.fit\_transform(updates)

clustering = KMeans(n\_clusters=2, n\_init=10, random\_state=42).fit(updates\_reduced)

labels = clustering.labels\_

benign\_cluster = np.argmax(np.bincount(labels))

benign\_indices = np.where(labels == benign\_cluster)[0]

if len(benign\_indices) == 0:

return np.mean(updates, axis=0)

return np.mean(updates[benign\_indices], axis=0)

except:

return np.mean(updates, axis=0)

def rdfl(updates):

if len(updates) < 2:

return np.mean(updates, axis=0) if len(updates) > 0 else np.zeros\_like(updates[0])

try:

pca = PCA(n\_components=min(50, updates.shape[1]))

updates\_reduced = pca.fit\_transform(updates)

clustering = HDBSCAN(min\_cluster\_size=2, min\_samples=2).fit(updates\_reduced)

labels = clustering.labels\_

benign\_indices = np.where(labels != -1)[0]

return np.mean(updates[benign\_indices], axis=0) if len(benign\_indices) > 0 else np.mean(updates, axis=0)

except:

return np.mean(updates, axis=0)

def flame(updates):

if len(updates) < 2:

return np.mean(updates, axis=0) if len(updates) > 0 else np.zeros\_like(updates[0])

try:

norms = np.linalg.norm(updates, axis=1)

rho = np.median(norms)

clipped = [u \* min(1, rho / np.linalg.norm(u)) if np.linalg.norm(u) > 0 else u for u in updates]

pca = PCA(n\_components=min(50, updates.shape[1]))

clipped\_reduced = pca.fit\_transform(clipped)

clustering = HDBSCAN(min\_cluster\_size=2, min\_samples=2).fit(clipped\_reduced)

labels = clustering.labels\_

benign\_indices = np.where(labels != -1)[0]

return np.mean(np.array(clipped)[benign\_indices], axis=0) if len(benign\_indices) > 0 else np.mean(clipped, axis=0)

except:

return np.mean(updates, axis=0)

def generate\_trigger\_pattern(input\_shape, trigger\_type, trigger\_size):

trigger = np.zeros(input\_shape, dtype=np.float32)

if trigger\_type == 'pixel':

trigger[:trigger\_size:2] = 1.0

elif trigger\_type == 'square':

trigger[:trigger\_size] = 1.0

elif trigger\_type == 'cross':

trigger[:trigger\_size:trigger\_size//2] = 1.0

trigger[trigger\_size//2:trigger\_size] = 1.0

else: # random

trigger[:trigger\_size] = np.random.choice([0, 1], size=trigger\_size, p=[0.7, 0.3]).astype(np.float32)

return trigger

def apply\_trigger(x, trigger\_pattern, trigger\_size):

triggered\_x = x.copy()

triggered\_x[:trigger\_size] = np.maximum(triggered\_x[:trigger\_size], trigger\_pattern[:trigger\_size])

return triggered\_x

def send\_data(client\_socket, data):

try:

data\_bytes = data.tobytes()

size = np.array([len(data\_bytes)], dtype=np.int64).tobytes()

client\_socket.sendall(size + data\_bytes)

except socket.error:

raise

def receive\_data(client\_socket, expected\_size=None):

try:

client\_socket.settimeout(60.0)

size\_data = client\_socket.recv(8)

if not size\_data:

raise socket.error("Client disconnected")

data\_size = np.frombuffer(size\_data, dtype=np.int64)[0]

received\_data = bytearray()

while len(received\_data) < data\_size:

packet = client\_socket.recv(min(4096, data\_size - len(received\_data)))

if not packet:

raise socket.error("Client disconnected during data transfer")

received\_data.extend(packet)

data = np.frombuffer(received\_data, dtype=np.float32)

if expected\_size and len(data) != expected\_size:

raise ValueError(f"Received data size {len(data)} does not match expected {expected\_size}")

return data

except socket.error:

raise

# Server setup

server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

server\_socket.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)

server\_address = ('10.100.14.37', 9999) # Use localhost for testing

server\_socket.bind(server\_address)

server\_socket.listen(5)

print("Server listening at", server\_address)

T = 10

n\_clients = 5

m\_selected = 5

AR = 0.4

PDR = 0.15625

NIR\_values = [0.0, 0.25, 0.50, 0.75]

methods = {

'RFBDS': lambda updates: np.mean(adaptive\_clipping(updates, auto\_optics(np.array([amsparse(u, layer\_shapes) for u in updates]))), axis=0),

'PrivRFBDS': lambda updates\_shares: reconstruct\_shares(priv\_aggregation(

priv\_clipping(

priv\_selection(

priv\_shuffling(updates\_shares, len(updates\_shares)), m\_selected),

priv\_optics(

priv\_selection(

priv\_shuffling(updates\_shares, len(updates\_shares)), m\_selected))))),

'FedAvg': fedavg,

'Median': median,

'Trimmed-Mean': trimmed\_mean,

'FedCC': fedcc,

'RDFL': rdfl,

'FLAME': flame

}

for dataset\_name in ['MNIST', 'FMNIST', 'CIFAR10']:

x\_train, y\_train, x\_test, y\_test, input\_shape = datasets[dataset\_name]

global\_model = create\_model(input\_shape)

global\_weights = global\_model.get\_weights()

layer\_shapes = [w.shape for w in global\_weights]

global\_weights\_flat = np.concatenate([w.flatten() for w in global\_weights]).astype(np.float32)

samples\_per\_client = 6000 if dataset\_name != 'CIFAR10' else 5000

expected\_update\_size = len(global\_weights\_flat) \* 2 # delta\_w + delta\_w\_am

for NIR in NIR\_values:

print(f"\nDataset: {dataset\_name}, NIR: {NIR}")

client\_data = []

indices = np.arange(len(x\_train))

if NIR == 0.0: # IID

np.random.shuffle(indices)

for i in range(n\_clients):

client\_indices = indices[i \* samples\_per\_client:(i + 1) \* samples\_per\_client]

client\_data.append((x\_train[client\_indices], y\_train[client\_indices], i >= int(n\_clients \* (1 - AR))))

else: # Non-IID

label\_dist = dirichlet.rvs([1.0 - NIR] \* 10, size=n\_clients)

for i in range(n\_clients):

client\_indices = []

probs = label\_dist[i]

for label in range(10):

label\_indices = np.where(y\_train == label)[0]

n\_samples = int(samples\_per\_client \* probs[label])

if len(label\_indices) > 0:

selected = np.random.choice(label\_indices, min(n\_samples, len(label\_indices)), replace=False)

client\_indices.extend(selected)

client\_indices = np.array(client\_indices)

if len(client\_indices) < samples\_per\_client:

extra\_indices = np.random.choice(indices, samples\_per\_client - len(client\_indices), replace=False)

client\_indices = np.concatenate([client\_indices, extra\_indices])

elif len(client\_indices) > samples\_per\_client:

client\_indices = client\_indices[:samples\_per\_client]

np.random.shuffle(client\_indices)

client\_data.append((x\_train[client\_indices], y\_train[client\_indices], i >= int(n\_clients \* (1 - AR))))

results = {}

for method\_name, method in methods.items():

results[method\_name] = {'MA': None, 'AASR': None}

print(f"\nMethod: {method\_name}")

client\_sockets = []

try:

for i in range(n\_clients):

client\_socket, addr = server\_socket.accept()

client\_socket.settimeout(60.0)

print(f"Client {i} connected from {addr}")

client\_sockets.append(client\_socket)

x\_client, y\_client, is\_malicious = client\_data[i]

client\_data\_flat = np.concatenate([

x\_client.flatten(),

y\_client.astype(np.float32),

np.array([float(is\_malicious)], dtype=np.float32)

]).astype(np.float32)

send\_data(client\_socket, client\_data\_flat)

send\_data(client\_socket, global\_weights\_flat)

current\_weights\_flat = global\_weights\_flat.copy()

current\_model = create\_model(input\_shape)

current\_weights = [

current\_weights\_flat[i:i + w.size].reshape(w.shape)

for i, w in zip(np.cumsum([0] + [w.size for w in global\_weights[:-1]]), global\_weights)

]

current\_model.set\_weights(current\_weights)

for t in range(T):

client\_updates = []

client\_updates\_shares = []

active\_sockets = []

for i, client\_socket in enumerate(client\_sockets):

if client\_socket is None:

continue

try:

updates = receive\_data(client\_socket, expected\_update\_size)

split\_idx = len(updates) // 2

delta\_w = updates[:split\_idx]

delta\_w\_am = updates[split\_idx:]

client\_updates.append(delta\_w)

client\_updates\_shares.append(secret\_share(delta\_w\_am))

active\_sockets.append(client\_socket)

except socket.error:

client\_sockets[i] = None

client\_sockets = active\_sockets

if not client\_updates:

print(f"Round {t+1}/{T} - No active clients")

continue

try:

if method\_name == 'PrivRFBDS':

aggregated\_update = method(client\_updates\_shares)

else:

aggregated\_update = method(np.array(client\_updates))

current\_weights\_flat += aggregated\_update

current\_weights = [

current\_weights\_flat[i:i + w.size].reshape(w.shape)

for i, w in zip(np.cumsum([0] + [w.size for w in global\_weights[:-1]]), global\_weights)

]

current\_model.set\_weights(current\_weights)

loss, ma = current\_model.evaluate(x\_test, y\_test, verbose=0)

# Dynamic trigger for evaluation

trigger\_type = random.choice(['pixel', 'square', 'cross', 'random'])

trigger\_size = 5\*28 if dataset\_name != 'CIFAR10' else 8\*32

trigger\_pattern = generate\_trigger\_pattern(input\_shape, trigger\_type, trigger\_size)

x\_test\_backdoor = np.array([apply\_trigger(x, trigger\_pattern, trigger\_size) for x in x\_test])

y\_test\_backdoor = np.ones\_like(y\_test) \* 7

\_, aasr = current\_model.evaluate(x\_test\_backdoor, y\_test\_backdoor, verbose=0)

if t == T - 1:

results[method\_name]['MA'] = ma \* 100

results[method\_name]['AASR'] = aasr \* 100

print(f"Round {t+1}/{T} - MA: {ma\*100:.2f}%, AASR: {aasr\*100:.2f}%")

except Exception as e:

print(f"Round {t+1}/{T} - Aggregation error: {e}")

continue

for client\_socket in client\_sockets:

if client\_socket:

try:

send\_data(client\_socket, current\_weights\_flat)

except socket.error:

client\_sockets[client\_sockets.index(client\_socket)] = None

except Exception as e:

print(f"Error during method {method\_name}: {e}")

finally:

for client\_socket in client\_sockets:

if client\_socket:

try:

client\_socket.close()

except:

pass

print(f"\nResults for Dataset: {dataset\_name}, NIR: {NIR}, AR: {AR\*100}%, PDR: {PDR\*100}%")

print("Method\t\tAASR (%)\tMA (%)")

print("-" \* 40)

for method\_name in methods:

ma = results[method\_name]['MA']

aasr = results[method\_name]['AASR']

if ma is not None and aasr is not None:

print(f"{method\_name:<15}\t{aasr:.2f}\t\t{ma:.2f}")

server\_socket.close()

client :

import socket

import numpy as np

import tensorflow as tf

import threading

import time

import random

def create\_model(input\_shape):

model = tf.keras.Sequential([

tf.keras.layers.Input(shape=(input\_shape,)),

tf.keras.layers.Dense(128, activation='relu'),

tf.keras.layers.Dense(10, activation='softmax')

])

model.compile(optimizer='sgd', loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

return model

def amsparse(delta\_w, layer\_shapes):

amsparse\_updates = []

start = 0

for shape in layer\_shapes:

size = np.prod(shape)

layer\_delta = delta\_w[start:start + size].reshape(shape)

layer\_max = np.max(np.abs(layer\_delta), axis=tuple(range(len(shape) - 1)))

signs = np.sign(layer\_delta)

amsparse\_layer = signs \* layer\_max

amsparse\_updates.append(amsparse\_layer.flatten())

start += size

return np.concatenate(amsparse\_updates)

def generate\_trigger\_pattern(input\_shape, trigger\_type, trigger\_size):

trigger = np.zeros(input\_shape, dtype=np.float32)

if trigger\_type == 'pixel':

trigger[:trigger\_size:2] = 1.0

elif trigger\_type == 'square':

trigger[:trigger\_size] = 1.0

elif trigger\_type == 'cross':

trigger[:trigger\_size:trigger\_size//2] = 1.0

trigger[trigger\_size//2:trigger\_size] = 1.0

else: # random

trigger[:trigger\_size] = np.random.choice([0, 1], size=trigger\_size, p=[0.7, 0.3]).astype(np.float32)

return trigger

def apply\_trigger(x, trigger\_pattern, trigger\_size):

triggered\_x = x.copy()

triggered\_x[:trigger\_size] = np.maximum(triggered\_x[:trigger\_size], trigger\_pattern[:trigger\_size])

return triggered\_x

def send\_data(client\_socket, data):

try:

data\_bytes = data.tobytes()

size = np.array([len(data\_bytes)], dtype=np.int64).tobytes()

client\_socket.sendall(size + data\_bytes)

except socket.error:

raise

def receive\_data(client\_socket, expected\_size=None):

try:

client\_socket.settimeout(60.0)

size\_data = client\_socket.recv(8)

if not size\_data:

raise socket.error("Server disconnected")

data\_size = np.frombuffer(size\_data, dtype=np.int64)[0]

received\_data = bytearray()

while len(received\_data) < data\_size:

packet = client\_socket.recv(min(4096, data\_size - len(received\_data)))

if not packet:

raise socket.error("Server disconnected during data transfer")

received\_data.extend(packet)

data = np.frombuffer(received\_data, dtype=np.float32)

if expected\_size and len(data) != expected\_size:

raise ValueError(f"Received data size {len(data)} does not match expected {expected\_size}")

return data

except socket.error:

raise

def client\_thread(client\_id, server\_address, T, batch\_size, input\_shape, dataset\_name):

samples\_per\_client = 6000 if dataset\_name != 'CIFAR10' else 5000

for method\_idx in range(8): # One for each method

client\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

retry\_count = 0

max\_retries = 5

while retry\_count < max\_retries:

try:

client\_socket.connect(server\_address)

print(f"Client {client\_id} connected to {server\_address} for method {method\_idx}")

break

except ConnectionRefusedError:

retry\_count += 1

time.sleep(1)

if retry\_count == max\_retries:

print(f"Client {client\_id} failed to connect after {max\_retries} retries")

return

try:

client\_data\_flat = receive\_data(client\_socket)

data\_size = samples\_per\_client \* input\_shape

x\_client = client\_data\_flat[:data\_size].reshape(samples\_per\_client, input\_shape)

y\_client = client\_data\_flat[data\_size:data\_size + samples\_per\_client].astype(np.int32)

is\_malicious = bool(client\_data\_flat[-1])

global\_weights\_flat = receive\_data(client\_socket)

model = create\_model(input\_shape)

global\_weights = [

global\_weights\_flat[i:i + w.size].reshape(w.shape)

for i, w in zip(np.cumsum([0] + [w.size for w in model.get\_weights()[:-1]]), model.get\_weights())

]

model.set\_weights(global\_weights)

layer\_shapes = [w.shape for w in model.get\_weights()]

expected\_update\_size = len(global\_weights\_flat) \* 2

# Initialize trigger pattern for this client

trigger\_type = random.choice(['pixel', 'square', 'cross', 'random'])

trigger\_size = 5\*28 if dataset\_name != 'CIFAR10' else 8\*32

trigger\_pattern = generate\_trigger\_pattern(input\_shape, trigger\_type, trigger\_size)

for t in range(T):

try:

if is\_malicious:

n\_poison = int(len(x\_client) \* 0.15625)

poisoned\_data = x\_client.copy()

# Evolve trigger every 5 rounds

if t % 5 == 0:

trigger\_type = random.choice(['pixel', 'square', 'cross', 'random'])

trigger\_pattern = generate\_trigger\_pattern(input\_shape, trigger\_type, trigger\_size)

poisoned\_data[:n\_poison] = np.array([

apply\_trigger(x, trigger\_pattern, trigger\_size)

for x in poisoned\_data[:n\_poison]

])

poisoned\_labels = y\_client.copy()

poisoned\_labels[:n\_poison] = 7

model.fit(poisoned\_data, poisoned\_labels, epochs=1, batch\_size=batch\_size, verbose=0)

else:

model.fit(x\_client, y\_client, epochs=1, batch\_size=batch\_size, verbose=0)

new\_weights\_flat = np.concatenate([w.flatten() for w in model.get\_weights()]).astype(np.float32)

delta\_w = new\_weights\_flat - global\_weights\_flat

delta\_w\_am = amsparse(delta\_w, layer\_shapes)

updates = np.concatenate([delta\_w, delta\_w\_am]).astype(np.float32)

send\_data(client\_socket, updates)

global\_weights\_flat = receive\_data(client\_socket, len(global\_weights\_flat))

global\_weights = [

global\_weights\_flat[i:i + w.size].reshape(w.shape)

for i, w in zip(np.cumsum([0] + [w.size for w in model.get\_weights()[:-1]]), model.get\_weights())

]

model.set\_weights(global\_weights)

except socket.error:

print(f"Client {client\_id} disconnected during round {t+1}")

break

except Exception as e:

print(f"Client {client\_id} error: {e}")

finally:

try:

client\_socket.close()

except:

pass

# Configuration

server\_address = ('localhost', 9999) # Use localhost for testing

T = 10

batch\_size = 64

n\_clients = 5

dataset\_configs = {

'MNIST': 784,

'FMNIST': 784,

'CIFAR10': 3072

}

for dataset\_name, input\_shape in dataset\_configs.items():

threads = []

for i in range(n\_clients):

thread = threading.Thread(target=client\_thread, args=(i, server\_address, T, batch\_size, input\_shape, dataset\_name))

threads.append(thread)

thread.start()

for thread in threads:

thread.join()

output:

C:\Users\P . Bharathi\Desktop>conda activate tf\_env

(tf\_env) C:\Users\P . Bharathi\Desktop>python noval.py

Server listening at ('10.100.14.37', 9999)

Dataset: MNIST, NIR: 0.0

Method: RFBDS

Client 0 connected from ('10.100.14.180', 58694)

Client 1 connected from ('10.100.14.180', 58697)

Client 2 connected from ('10.100.14.180', 58695)

Client 3 connected from ('10.100.14.180', 58696)

Client 4 connected from ('10.100.14.180', 58698)

Round 1/10 - MA: 61.40%, AASR: 37.70%

Round 2/10 - MA: 76.73%, AASR: 14.61%

Round 3/10 - MA: 81.54%, AASR: 66.02%

Round 4/10 - MA: 84.00%, AASR: 74.23%

Round 5/10 - MA: 85.26%, AASR: 84.92%

Round 6/10 - MA: 86.27%, AASR: 81.38%

Round 7/10 - MA: 86.90%, AASR: 45.84%

Round 8/10 - MA: 87.36%, AASR: 84.82%

Round 9/10 - MA: 87.83%, AASR: 96.80%

Round 10/10 - MA: 88.23%, AASR: 53.58%

Method: PrivRFBDS

Client 0 connected from ('10.100.14.180', 58752)

Client 1 connected from ('10.100.14.180', 58753)

Client 2 connected from ('10.100.14.180', 58754)

Client 3 connected from ('10.100.14.180', 58755)

Client 4 connected from ('10.100.14.180', 58756)

Round 1/10 - MA: 18.07%, AASR: 0.00%

Round 2/10 - MA: 9.49%, AASR: 0.00%

Round 3/10 - MA: 16.80%, AASR: 39.76%

Round 4/10 - MA: 23.57%, AASR: 0.28%

Round 5/10 - MA: 20.16%, AASR: 34.58%

Round 6/10 - MA: 32.89%, AASR: 0.84%

Round 7/10 - MA: 14.24%, AASR: 1.61%

Round 8/10 - MA: 46.38%, AASR: 43.54%

Round 9/10 - MA: 45.78%, AASR: 42.54%

Round 10/10 - MA: 59.55%, AASR: 85.02%

Method: FedAvg

Client 0 connected from ('10.100.14.180', 58782)

Client 1 connected from ('10.100.14.180', 58783)

Client 2 connected from ('10.100.14.180', 58784)

Client 3 connected from ('10.100.14.180', 58785)

Client 4 connected from ('10.100.14.180', 58786)

Round 1/10 - MA: 60.67%, AASR: 10.94%

Round 2/10 - MA: 76.14%, AASR: 17.84%

Round 3/10 - MA: 81.67%, AASR: 28.99%

Round 4/10 - MA: 83.93%, AASR: 27.93%

Round 5/10 - MA: 85.34%, AASR: 34.72%

Round 6/10 - MA: 86.17%, AASR: 67.42%

Round 7/10 - MA: 87.10%, AASR: 28.12%

Round 8/10 - MA: 87.43%, AASR: 65.60%

Round 9/10 - MA: 87.82%, AASR: 67.54%

Round 10/10 - MA: 88.31%, AASR: 99.01%

Method: Median

Client 0 connected from ('10.100.14.180', 58802)

Client 1 connected from ('10.100.14.180', 58803)

Client 2 connected from ('10.100.14.180', 58804)

Client 3 connected from ('10.100.14.180', 58805)

Client 4 connected from ('10.100.14.180', 58806)

Round 1/10 - MA: 63.24%, AASR: 1.93%

Round 2/10 - MA: 77.08%, AASR: 4.21%

Round 3/10 - MA: 81.74%, AASR: 1.45%

Round 4/10 - MA: 84.03%, AASR: 2.95%

Round 5/10 - MA: 85.54%, AASR: 6.68%

Round 6/10 - MA: 86.45%, AASR: 7.29%

Round 7/10 - MA: 87.04%, AASR: 8.84%

Round 8/10 - MA: 87.50%, AASR: 7.29%

Round 9/10 - MA: 88.03%, AASR: 7.50%

Round 10/10 - MA: 88.41%, AASR: 8.63%

Method: Trimmed-Mean

Client 0 connected from ('10.100.14.180', 58826)

Client 1 connected from ('10.100.14.180', 58827)

Client 2 connected from ('10.100.14.180', 58829)

Client 3 connected from ('10.100.14.180', 58830)

Client 4 connected from ('10.100.14.180', 58831)

Round 1/10 - MA: 61.30%, AASR: 11.18%

Round 2/10 - MA: 76.66%, AASR: 39.87%

Round 3/10 - MA: 81.47%, AASR: 89.11%

Round 4/10 - MA: 83.84%, AASR: 76.48%

Round 5/10 - MA: 85.45%, AASR: 94.91%

Round 6/10 - MA: 86.30%, AASR: 98.50%

Round 7/10 - MA: 87.04%, AASR: 86.25%

Round 8/10 - MA: 87.54%, AASR: 87.61%

Round 9/10 - MA: 87.89%, AASR: 89.93%

Round 10/10 - MA: 88.37%, AASR: 90.27%

Method: FedCC

Client 0 connected from ('10.100.14.180', 58865)

Client 1 connected from ('10.100.14.180', 58866)

Client 2 connected from ('10.100.14.180', 58867)

Client 3 connected from ('10.100.14.180', 58868)

Client 4 connected from ('10.100.14.180', 58870)

Round 1/10 - MA: 61.17%, AASR: 54.94%

Round 2/10 - MA: 76.46%, AASR: 13.18%

Round 3/10 - MA: 81.52%, AASR: 83.08%

Round 4/10 - MA: 83.98%, AASR: 74.92%

Round 5/10 - MA: 85.42%, AASR: 21.85%

Round 6/10 - MA: 86.36%, AASR: 94.32%

Round 7/10 - MA: 87.11%, AASR: 96.00%

Round 8/10 - MA: 87.57%, AASR: 96.95%

Round 9/10 - MA: 88.00%, AASR: 32.82%

Round 10/10 - MA: 88.40%, AASR: 30.19%

Method: RDFL

Client 0 connected from ('10.100.14.180', 58889)

Client 1 connected from ('10.100.14.180', 58890)

Client 2 connected from ('10.100.14.180', 58892)

Client 3 connected from ('10.100.14.180', 58893)

Client 4 connected from ('10.100.14.180', 58898)

Round 1/10 - MA: 61.42%, AASR: 44.88%

Round 2/10 - MA: 76.42%, AASR: 84.47%

Round 3/10 - MA: 81.49%, AASR: 89.07%

Round 4/10 - MA: 84.02%, AASR: 75.59%

Round 5/10 - MA: 85.42%, AASR: 50.89%

Round 6/10 - MA: 86.16%, AASR: 84.09%

Round 7/10 - MA: 87.03%, AASR: 99.26%

Round 8/10 - MA: 87.46%, AASR: 47.14%

Round 9/10 - MA: 87.86%, AASR: 99.83%

Round 10/10 - MA: 88.32%, AASR: 44.75%

Method: FLAME

Client 0 connected from ('10.100.14.180', 58916)

Client 1 connected from ('10.100.14.180', 58917)

Client 2 connected from ('10.100.14.180', 58918)

Client 3 connected from ('10.100.14.180', 58919)

Client 4 connected from ('10.100.14.180', 58920)

Round 1/10 - MA: 60.86%, AASR: 15.12%

Round 2/10 - MA: 76.61%, AASR: 29.97%

Round 3/10 - MA: 81.64%, AASR: 31.24%

Round 4/10 - MA: 83.91%, AASR: 81.05%

Round 5/10 - MA: 85.55%, AASR: 35.14%

Round 6/10 - MA: 86.43%, AASR: 43.79%

Round 7/10 - MA: 86.99%, AASR: 32.35%

Round 8/10 - MA: 87.52%, AASR: 42.11%

Round 9/10 - MA: 88.00%, AASR: 56.83%

Round 10/10 - MA: 88.40%, AASR: 97.71%

Results for Dataset: MNIST, NIR: 0.0, AR: 40.0%, PDR: 15.625%

Method AASR (%) MA (%)

----------------------------------------

RFBDS 53.58 88.23

PrivRFBDS 85.02 59.55

FedAvg 99.01 88.31

Median 8.63 88.41

Trimmed-Mean 90.27 88.37

FedCC 30.19 88.40

RDFL 44.75 88.32

FLAME 97.71 88.40

Dataset: MNIST, NIR: 0.25

Method: RFBDS

Client 0 connected from ('10.100.14.180', 58956)

Client 1 connected from ('10.100.14.180', 58957)

Client 2 connected from ('10.100.14.180', 58958)

Client 3 connected from ('10.100.14.180', 58959)

Client 4 connected from ('10.100.14.180', 58960)

Round 1/10 - MA: 40.59%, AASR: 2.74%

Round 2/10 - MA: 61.45%, AASR: 91.35%

Round 3/10 - MA: 70.93%, AASR: 92.57%

Round 4/10 - MA: 76.07%, AASR: 92.37%

Round 5/10 - MA: 79.39%, AASR: 60.22%

Round 6/10 - MA: 81.46%, AASR: 97.97%

Round 7/10 - MA: 82.73%, AASR: 77.06%

Round 8/10 - MA: 83.74%, AASR: 80.43%

Round 9/10 - MA: 84.20%, AASR: 97.82%

Round 10/10 - MA: 85.08%, AASR: 99.83%

Method: PrivRFBDS

Client 0 connected from ('10.100.14.180', 59000)

Client 1 connected from ('10.100.14.180', 59001)

Client 2 connected from ('10.100.14.180', 59002)

Client 3 connected from ('10.100.14.180', 59003)

Client 4 connected from ('10.100.14.180', 59004)

Round 1/10 - MA: 40.23%, AASR: 16.64%

Round 2/10 - MA: 20.09%, AASR: 52.76%

Round 3/10 - MA: 16.07%, AASR: 0.72%

Round 4/10 - MA: 31.41%, AASR: 12.44%

Round 5/10 - MA: 34.67%, AASR: 98.95%

Round 6/10 - MA: 26.16%, AASR: 0.00%

Round 7/10 - MA: 26.90%, AASR: 99.53%

Round 8/10 - MA: 42.20%, AASR: 29.57%

Round 9/10 - MA: 36.27%, AASR: 99.91%

Round 10/10 - MA: 25.22%, AASR: 0.00%

Method: FedAvg

Client 0 connected from ('10.100.14.180', 59022)

Client 1 connected from ('10.100.14.180', 59023)

Client 2 connected from ('10.100.14.180', 59024)

Client 3 connected from ('10.100.14.180', 59026)

Client 4 connected from ('10.100.14.180', 59027)

Round 1/10 - MA: 41.93%, AASR: 21.54%

Round 2/10 - MA: 61.94%, AASR: 42.03%

Round 3/10 - MA: 71.06%, AASR: 44.38%

Round 4/10 - MA: 76.17%, AASR: 24.68%

Round 5/10 - MA: 79.39%, AASR: 47.77%

Round 6/10 - MA: 81.36%, AASR: 35.49%

Round 7/10 - MA: 82.79%, AASR: 78.59%

Round 8/10 - MA: 83.79%, AASR: 99.91%

Round 9/10 - MA: 84.60%, AASR: 98.08%

Round 10/10 - MA: 85.33%, AASR: 49.90%

Method: Median

Client 0 connected from ('10.100.14.180', 59047)

Client 1 connected from ('10.100.14.180', 59048)

Client 2 connected from ('10.100.14.180', 59050)

Client 3 connected from ('10.100.14.180', 59051)

Client 4 connected from ('10.100.14.180', 59052)

Round 1/10 - MA: 38.38%, AASR: 20.25%

Round 2/10 - MA: 54.45%, AASR: 44.00%

Round 3/10 - MA: 66.19%, AASR: 20.66%

Round 4/10 - MA: 74.49%, AASR: 44.24%

Round 5/10 - MA: 78.27%, AASR: 16.55%

Round 6/10 - MA: 80.51%, AASR: 9.25%

Round 7/10 - MA: 82.20%, AASR: 14.53%

Round 8/10 - MA: 83.03%, AASR: 15.51%

Round 9/10 - MA: 84.36%, AASR: 12.75%

Round 10/10 - MA: 84.98%, AASR: 44.06%

Method: Trimmed-Mean

Client 0 connected from ('10.100.14.180', 59081)

Client 1 connected from ('10.100.14.180', 59082)

Client 2 connected from ('10.100.14.180', 59083)

Client 3 connected from ('10.100.14.180', 59084)

Client 4 connected from ('10.100.14.180', 59085)

Round 1/10 - MA: 43.57%, AASR: 28.64%

Round 2/10 - MA: 61.24%, AASR: 94.82%

Round 3/10 - MA: 71.31%, AASR: 54.39%

Round 4/10 - MA: 76.23%, AASR: 55.95%

Round 5/10 - MA: 79.41%, AASR: 56.29%

Round 6/10 - MA: 81.68%, AASR: 23.80%

Round 7/10 - MA: 82.99%, AASR: 56.23%

Round 8/10 - MA: 83.89%, AASR: 34.61%

Round 9/10 - MA: 84.79%, AASR: 29.18%

Round 10/10 - MA: 85.54%, AASR: 98.59%

Method: FedCC

Client 0 connected from ('10.100.14.180', 59110)

Client 1 connected from ('10.100.14.180', 59111)

Client 2 connected from ('10.100.14.180', 59112)

Client 3 connected from ('10.100.14.180', 59113)

Client 4 connected from ('10.100.14.180', 59115)

Round 1/10 - MA: 43.25%, AASR: 60.55%

Round 2/10 - MA: 62.09%, AASR: 24.22%

Round 3/10 - MA: 70.74%, AASR: 95.14%

Round 4/10 - MA: 76.47%, AASR: 43.64%

Round 5/10 - MA: 79.26%, AASR: 61.47%

Round 6/10 - MA: 81.50%, AASR: 78.73%

Round 7/10 - MA: 82.78%, AASR: 51.61%

Round 8/10 - MA: 83.78%, AASR: 72.68%

Round 9/10 - MA: 84.60%, AASR: 71.19%

Round 10/10 - MA: 85.28%, AASR: 60.88%

Method: RDFL

Client 0 connected from ('10.100.14.180', 59126)

Client 1 connected from ('10.100.14.180', 59127)

Client 2 connected from ('10.100.14.180', 59128)

Client 3 connected from ('10.100.14.180', 59129)

Client 4 connected from ('10.100.14.180', 59130)

Round 1/10 - MA: 43.17%, AASR: 76.59%

Round 2/10 - MA: 61.59%, AASR: 64.00%

Round 3/10 - MA: 71.35%, AASR: 97.73%

Round 4/10 - MA: 75.92%, AASR: 37.57%

Round 5/10 - MA: 79.30%, AASR: 32.29%

Round 6/10 - MA: 81.39%, AASR: 71.32%

Round 7/10 - MA: 82.80%, AASR: 31.62%

Round 8/10 - MA: 83.73%, AASR: 99.47%

Round 9/10 - MA: 84.66%, AASR: 99.54%

Round 10/10 - MA: 85.14%, AASR: 88.75%

Method: FLAME

Client 0 connected from ('10.100.14.180', 59177)

Client 1 connected from ('10.100.14.180', 59178)

Client 2 connected from ('10.100.14.180', 59179)

Client 3 connected from ('10.100.14.180', 59180)

Client 4 connected from ('10.100.14.180', 59181)

Round 1/10 - MA: 42.89%, AASR: 31.14%

Round 2/10 - MA: 61.25%, AASR: 41.09%

Round 3/10 - MA: 71.02%, AASR: 83.55%

Round 4/10 - MA: 76.47%, AASR: 93.85%

Round 5/10 - MA: 79.41%, AASR: 57.96%

Round 6/10 - MA: 81.37%, AASR: 37.65%

Round 7/10 - MA: 82.89%, AASR: 63.64%

Round 8/10 - MA: 83.80%, AASR: 89.88%

Round 9/10 - MA: 84.67%, AASR: 91.33%

Round 10/10 - MA: 85.42%, AASR: 92.41%

Results for Dataset: MNIST, NIR: 0.25, AR: 40.0%, PDR: 15.625%

Method AASR (%) MA (%)

----------------------------------------

RFBDS 99.83 85.08

PrivRFBDS 0.00 25.22

FedAvg 49.90 85.33

Median 44.06 84.98

Trimmed-Mean 98.59 85.54

FedCC 60.88 85.28

RDFL 88.75 85.14

FLAME 92.41 85.42

[22bit005@mepcolinux ~]$ls

0.25.txt client.py diamond.c finish grokall.txt mca.txt os sc.py sixthrun.txt Vsem

0.75.txt core.228416 environ.txt five.txt grokk.txt merror.txt output.txt se1.txt socket.txt

15.25.txt datastructure env.txt got.txt Isem ml.txt pclients.txt sem2 sparkdownload.txt

15.75.txt dbms error.txt graph0.25pdr2.txt itt new.txt pdr.txt server.c spdiamond.c

angular.txt deep.txt e.txt graph0.75pdr.txt java no.txt robust.txt serverclient.txt star

c1.c description.txt examplelab.txt graph5.txt jdbc.java novalty.txt rpp.txt server.py upto0\_5.txt

client.c diamond fifthrunserver.txt graph.txt job.txt n.txt s1.c set1.txt VIsem

[22bit005@mepcolinux ~]$cat 0.25.txt

import os

os.environ['TF\_CPP\_MIN\_LOG\_LEVEL'] = '2'

import socket

import numpy as np

from sklearn.cluster import OPTICS, KMeans, HDBSCAN

from sklearn.decomposition import PCA

import tensorflow as tf

from scipy.stats import dirichlet

import random

import time

# Load datasets

(x\_train\_mnist, y\_train\_mnist), (x\_test\_mnist, y\_test\_mnist) = tf.keras.datasets.mnist.load\_data()

x\_train\_mnist = x\_train\_mnist.reshape(-1, 28\*28).astype(np.float32) / 255.0

x\_test\_mnist = x\_test\_mnist.reshape(-1, 28\*28).astype(np.float32) / 255.0

(x\_train\_fmnist, y\_train\_fmnist), (x\_test\_fmnist, y\_test\_fmnist) = tf.keras.datasets.fashion\_mnist.load\_data()

x\_train\_fmnist = x\_train\_fmnist.reshape(-1, 28\*28).astype(np.float32) / 255.0

x\_test\_fmnist = x\_test\_fmnist.reshape(-1, 28\*28).astype(np.float32) / 255.0

(x\_train\_cifar, y\_train\_cifar), (x\_test\_cifar, y\_test\_cifar) = tf.keras.datasets.cifar10.load\_data()

x\_train\_cifar = x\_train\_cifar.reshape(-1, 32\*32\*3).astype(np.float32) / 255.0

x\_test\_cifar = x\_test\_cifar.reshape(-1, 32\*32\*3).astype(np.float32) / 255.0

y\_train\_cifar = y\_train\_cifar.flatten()

y\_test\_cifar = y\_test\_cifar.flatten()

datasets = {

'MNIST': (x\_train\_mnist, y\_train\_mnist, x\_test\_mnist, y\_test\_mnist, 784),

'FMNIST': (x\_train\_fmnist, y\_train\_fmnist, x\_test\_fmnist, y\_test\_fmnist, 784),

'CIFAR10': (x\_train\_cifar, y\_train\_cifar, x\_test\_cifar, y\_test\_cifar, 3072)

}

def create\_model(input\_shape):

model = tf.keras.Sequential([

tf.keras.layers.Input(shape=(input\_shape,)),

tf.keras.layers.Dense(128, activation='relu'),

tf.keras.layers.Dense(10, activation='softmax')

])

model.compile(optimizer='sgd', loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

return model

def secret\_share(data, n\_parties=2):

shares = [np.random.uniform(-1000, 1000, size=data.shape).astype(np.float32) for \_ in range(n\_parties - 1)]

shares.append(data - sum(shares))

return shares

def reconstruct\_shares(shares):

return sum(shares)

def amsparse(delta\_w, layer\_shapes):

amsparse\_updates = []

start = 0

for shape in layer\_shapes:

size = np.prod(shape)

layer\_delta = delta\_w[start:start + size].reshape(shape)

layer\_max = np.max(np.abs(layer\_delta), axis=tuple(range(len(shape) - 1)))

signs = np.sign(layer\_delta)

amsparse\_layer = signs \* layer\_max

amsparse\_updates.append(amsparse\_layer.flatten())

start += size

return np.concatenate(amsparse\_updates)

def auto\_optics(updates):

if len(updates) < 2:

return np.arange(len(updates))

try:

pca = PCA(n\_components=min(50, updates.shape[1]))

updates\_reduced = pca.fit\_transform(updates)

dists = np.linalg.norm(updates\_reduced - updates\_reduced.mean(axis=0), axis=1)

k = min(max(2, len(updates) // 3), len(updates) - 1)

sorted\_dists = np.sort(dists)[:k]

eps = np.median(sorted\_dists) \* 1.5 # Adjusted for tighter clustering

clustering = OPTICS(min\_samples=k, eps=eps, metric='euclidean').fit(updates\_reduced)

labels = clustering.labels\_

majority\_group = np.where(labels != -1)[0]

return majority\_group if len(majority\_group) > 0 else np.arange(len(updates))

except:

return np.arange(len(updates))

def adaptive\_clipping(updates, benign\_indices):

if len(updates) == 0:

return np.zeros\_like(updates)

norms = np.linalg.norm(updates, axis=1)

rho = np.median(norms[benign\_indices]) if len(benign\_indices) > 0 else 1.0

clipped\_updates = []

for i in benign\_indices:

update = updates[i]

norm = norms[i]

clipped = update \* min(1, rho / norm) if norm > 0 else update

clipped\_updates.append(clipped)

return np.array(clipped\_updates) if clipped\_updates else updates

def priv\_shuffling(updates\_shares, n\_clients):

pi\_0 = np.random.permutation(n\_clients)

pi\_1 = np.random.permutation(n\_clients)

shuffled\_shares = [[updates\_shares[pi\_0[i]][0], updates\_shares[pi\_1[i]][1]] for i in range(n\_clients)]

return shuffled\_shares

def priv\_selection(shuffled\_shares, m):

norms\_shares = [secret\_share(np.linalg.norm(reconstruct\_shares(shares))) for shares in shuffled\_shares]

norms = [reconstruct\_shares(ns) for ns in norms\_shares]

sorted\_indices = np.argsort(norms)[:m]

return [shuffled\_shares[i] for i in sorted\_indices]

def priv\_optics(updates\_shares):

updates = np.array([reconstruct\_shares(shares) for shares in updates\_shares])

if len(updates) < 2:

return secret\_share(np.arange(len(updates)))

try:

pca = PCA(n\_components=min(50, updates.shape[1]))

updates\_reduced = pca.fit\_transform(updates)

dists = np.linalg.norm(updates\_reduced - updates\_reduced.mean(axis=0), axis=1)

k = min(max(2, len(updates) // 3), len(updates) - 1)

sorted\_dists = np.sort(dists)[:k]

eps = np.median(sorted\_dists) \* 1.5

clustering = OPTICS(min\_samples=k, eps=eps, metric='euclidean').fit(updates\_reduced)

labels = clustering.labels\_

benign\_indices = np.where(labels != -1)[0]

return secret\_share(benign\_indices if len(benign\_indices) > 0 else np.arange(len(updates)))

except:

return secret\_share(np.arange(len(updates)))

def priv\_clipping(updates\_shares, benign\_indices\_shares):

updates = np.array([reconstruct\_shares(shares) for shares in updates\_shares])

benign\_indices = reconstruct\_shares(benign\_indices\_shares).astype(int)

norms = np.linalg.norm(updates, axis=1)

rho\_shares = secret\_share(np.median(norms[benign\_indices]) if len(benign\_indices) > 0 else 1.0)

clipped\_shares = []

for i in benign\_indices:

update\_shares = updates\_shares[i]

norm\_shares = secret\_share(norms[i])

rho\_val = reconstruct\_shares(rho\_shares)

norm\_val = reconstruct\_shares(norm\_shares)

factor = min(1, rho\_val / norm\_val) if norm\_val > 0 else 1.0

clipped\_shares.append([share \* factor for share in update\_shares])

return clipped\_shares if clipped\_shares else updates\_shares

def priv\_aggregation(clipped\_shares):

if not clipped\_shares:

return secret\_share(np.zeros\_like(reconstruct\_shares(clipped\_shares[0])))

clipped\_updates = np.array([reconstruct\_shares(shares) for shares in clipped\_shares])

return secret\_share(np.mean(clipped\_updates, axis=0))

def fedavg(updates):

return np.mean(updates, axis=0) if len(updates) > 0 else np.zeros\_like(updates[0])

def median(updates):

return np.median(updates, axis=0) if len(updates) > 0 else np.zeros\_like(updates[0])

def trimmed\_mean(updates, trim\_ratio=0.1):

if len(updates) < 3:

return np.mean(updates, axis=0) if len(updates) > 0 else np.zeros\_like(updates[0])

n\_trim = min(int(len(updates) \* trim\_ratio), len(updates) // 2 - 1)

if n\_trim == 0:

return np.mean(updates, axis=0)

sorted\_updates = np.sort(updates, axis=0)

trimmed = sorted\_updates[n\_trim:-n\_trim]

return np.mean(trimmed, axis=0)

def fedcc(updates):

if len(updates) < 2:

return np.mean(updates, axis=0) if len(updates) > 0 else np.zeros\_like(updates[0])

try:

pca = PCA(n\_components=min(50, updates.shape[1]))

updates\_reduced = pca.fit\_transform(updates)

clustering = KMeans(n\_clusters=2, n\_init=10, random\_state=42).fit(updates\_reduced)

labels = clustering.labels\_

benign\_cluster = np.argmax(np.bincount(labels))

benign\_indices = np.where(labels == benign\_cluster)[0]

if len(benign\_indices) == 0:

return np.mean(updates, axis=0)

return np.mean(updates[benign\_indices], axis=0)

except:

return np.mean(updates, axis=0)

def rdfl(updates):

if len(updates) < 2:

return np.mean(updates, axis=0) if len(updates) > 0 else np.zeros\_like(updates[0])

try:

pca = PCA(n\_components=min(50, updates.shape[1]))

updates\_reduced = pca.fit\_transform(updates)

clustering = HDBSCAN(min\_cluster\_size=2, min\_samples=2).fit(updates\_reduced)

labels = clustering.labels\_

benign\_indices = np.where(labels != -1)[0]

return np.mean(updates[benign\_indices], axis=0) if len(benign\_indices) > 0 else np.mean(updates, axis=0)

except:

return np.mean(updates, axis=0)

def flame(updates):

if len(updates) < 2:

return np.mean(updates, axis=0) if len(updates) > 0 else np.zeros\_like(updates[0])

try:

norms = np.linalg.norm(updates, axis=1)

rho = np.median(norms)

clipped = [u \* min(1, rho / np.linalg.norm(u)) if np.linalg.norm(u) > 0 else u for u in updates]

pca = PCA(n\_components=min(50, updates.shape[1]))

clipped\_reduced = pca.fit\_transform(clipped)

clustering = HDBSCAN(min\_cluster\_size=2, min\_samples=2).fit(clipped\_reduced)

labels = clustering.labels\_

benign\_indices = np.where(labels != -1)[0]

return np.mean(np.array(clipped)[benign\_indices], axis=0) if len(benign\_indices) > 0 else np.mean(clipped, axis=0)

except:

return np.mean(updates, axis=0)

def generate\_trigger\_pattern(shape, trigger\_type, trigger\_size):

if trigger\_type == 'pixel':

trigger = np.zeros(shape)

trigger[:trigger\_size:2] = 1.0

elif trigger\_type == 'square':

trigger = np.zeros(shape)

trigger[:trigger\_size] = 1.0

elif trigger\_type == 'cross':

trigger = np.zeros(shape)

trigger[trigger\_size//2] = 1.0

else:

trigger = np.random.choice([0, 1], size=shape, p=[0.7, 0.3]).astype(np.float32)

return trigger

def apply\_trigger(x, trigger\_pattern, trigger\_size):

triggered\_x = x.copy()

triggered\_x[:trigger\_size] = np.maximum(triggered\_x[:trigger\_size], trigger\_pattern[:trigger\_size])

return triggered\_x

def send\_data(client\_socket, data):

try:

data\_bytes = data.tobytes()

size = np.array([len(data\_bytes)], dtype=np.int64).tobytes()

client\_socket.sendall(size + data\_bytes)

except socket.error:

raise

def receive\_data(client\_socket):

try:

client\_socket.settimeout(120.0) # Increased timeout

size\_data = client\_socket.recv(8)

if not size\_data:

raise socket.error("Client disconnected")

data\_size = np.frombuffer(size\_data, dtype=np.int64)[0]

received\_data = bytearray()

while len(received\_data) < data\_size:

packet = client\_socket.recv(min(4096, data\_size - len(received\_data)))

if not packet:

raise socket.error("Client disconnected during data transfer")

received\_data.extend(packet)

return np.frombuffer(received\_data, dtype=np.float32)

except socket.error:

raise

# Server setup

server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

server\_socket.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)

server\_address = ('172.16.24.61', 9999)

server\_socket.bind(server\_address)

server\_socket.listen(5)

print("Server listening at", server\_address)

T = 10

n\_clients = 5

m\_selected = 5

AR = 0.4

PDR = 0.46875

NIR = 0.25

methods = {

'RFBDS': lambda updates: np.mean(adaptive\_clipping(updates, auto\_optics(np.array([amsparse(u, layer\_shapes) for u in updates]))), axis=0),

'PrivRFBDS': lambda updates\_shares: reconstruct\_shares(priv\_aggregation(

priv\_clipping(

priv\_selection(

priv\_shuffling(updates\_shares, len(updates\_shares)), m\_selected),

priv\_optics(

priv\_selection(

priv\_shuffling(updates\_shares, len(updates\_shares)), m\_selected))))),

'FedAvg': fedavg,

'Median': median,

'Trimmed-Mean': trimmed\_mean,

'FedCC': fedcc,

'RDFL': rdfl,

'FLAME': flame

}

for dataset\_name in ['MNIST', 'FMNIST', 'CIFAR10']:

x\_train, y\_train, x\_test, y\_test, input\_shape = datasets[dataset\_name]

global\_model = create\_model(input\_shape)

global\_weights = global\_model.get\_weights()

layer\_shapes = [w.shape for w in global\_weights]

global\_weights\_flat = np.concatenate([w.flatten() for w in global\_weights]).astype(np.float32)

samples\_per\_client = 6000 if dataset\_name != 'CIFAR10' else 5000

print(f"\nDataset: {dataset\_name}, NIR: {NIR}")

client\_data = []

indices = np.arange(len(x\_train))

label\_dist = dirichlet.rvs([1.0 - NIR] \* 10, size=n\_clients)

for i in range(n\_clients):

client\_indices = []

probs = label\_dist[i]

for label in range(10):

label\_indices = np.where(y\_train == label)[0]

n\_samples = int(samples\_per\_client \* probs[label])

if len(label\_indices) > 0:

selected = np.random.choice(label\_indices, min(n\_samples, len(label\_indices)), replace=False)

client\_indices.extend(selected)

client\_indices = np.array(client\_indices)

if len(client\_indices) < samples\_per\_client:

extra\_indices = np.random.choice(indices, samples\_per\_client - len(client\_indices), replace=False)

client\_indices = np.concatenate([client\_indices, extra\_indices])

elif len(client\_indices) > samples\_per\_client:

client\_indices = client\_indices[:samples\_per\_client]

np.random.shuffle(client\_indices)

client\_data.append((x\_train[client\_indices], y\_train[client\_indices], i >= int(n\_clients \* (1 - AR))))

results = {method\_name: {'MA': None, 'AASR': None} for method\_name in methods}

for method\_name, method in methods.items():

print(f"\nMethod: {method\_name}")

client\_sockets = []

try:

for i in range(n\_clients):

client\_socket, addr = server\_socket.accept()

client\_socket.settimeout(120.0)

print(f"Client {i} connected from {addr}")

client\_sockets.append(client\_socket)

x\_client, y\_client, is\_malicious = client\_data[i]

client\_data\_flat = np.concatenate([

x\_client.flatten(),

y\_client.astype(np.float32),

np.array([float(is\_malicious)], dtype=np.float32)

]).astype(np.float32)

send\_data(client\_socket, client\_data\_flat)

send\_data(client\_socket, global\_weights\_flat)

current\_weights\_flat = global\_weights\_flat.copy()

current\_model = create\_model(input\_shape)

current\_weights = [

current\_weights\_flat[i:i + w.size].reshape(w.shape)

for i, w in zip(np.cumsum([0] + [w.size for w in global\_weights[:-1]]), global\_weights)

]

current\_model.set\_weights(current\_weights)

for t in range(T):

client\_updates = []

client\_updates\_shares = []

active\_sockets = []

for i, client\_socket in enumerate(client\_sockets):

if client\_socket is None:

continue

try:

updates = receive\_data(client\_socket)

split\_idx = len(updates) // 2

delta\_w = updates[:split\_idx]

delta\_w\_am = updates[split\_idx:]

client\_updates.append(delta\_w)

client\_updates\_shares.append(secret\_share(delta\_w\_am))

active\_sockets.append(client\_socket)

except socket.error:

client\_sockets[i] = None

client\_sockets = active\_sockets

if not client\_updates:

print(f"Round {t+1}/{T} - No active clients")

continue

try:

if method\_name == 'PrivRFBDS':

aggregated\_update = method(client\_updates\_shares)

else:

aggregated\_update = method(np.array(client\_updates))

current\_weights\_flat += aggregated\_update

current\_weights = [

current\_weights\_flat[i:i + w.size].reshape(w.shape)

for i, w in zip(np.cumsum([0] + [w.size for w in global\_weights[:-1]]), global\_weights)

]

current\_model.set\_weights(current\_weights)

loss, ma = current\_model.evaluate(x\_test, y\_test, verbose=0)

trigger\_type = random.choice(['pixel', 'square', 'cross', 'random'])

trigger\_size = 5\*28 if dataset\_name != 'CIFAR10' else 8\*32

trigger\_pattern = generate\_trigger\_pattern(input\_shape, trigger\_type, trigger\_size)

x\_test\_backdoor = np.array([apply\_trigger(x, trigger\_pattern, trigger\_size) for x in x\_test])

y\_test\_backdoor = np.ones\_like(y\_test) \* 7

\_, aasr = current\_model.evaluate(x\_test\_backdoor, y\_test\_backdoor, verbose=0)

if t == T - 1:

results[method\_name]['MA'] = ma \* 100

results[method\_name]['AASR'] = aasr \* 100

print(f"Round {t+1}/{T} - MA: {ma\*100:.2f}%, AASR: {aasr\*100:.2f}%")

except Exception as e:

print(f"Round {t+1}/{T} - Aggregation error: {e}")

continue

for client\_socket in client\_sockets:

if client\_socket:

try:

send\_data(client\_socket, current\_weights\_flat)

except socket.error:

client\_sockets[client\_sockets.index(client\_socket)] = None

except Exception as e:

print(f"Error during method {method\_name}: {e}")

finally:

for client\_socket in client\_sockets:

if client\_socket:

try:

client\_socket.close()

except:

pass

print(f"\nResults for Dataset: {dataset\_name}, NIR: {NIR}, AR: {AR\*100}%, PDR: {PDR\*100}%")

print("Method\t\tAASR (%)\tMA (%)")

print("-" \* 40)

for method\_name in methods:

ma = results[method\_name]['MA']

aasr = results[method\_name]['AASR']

if ma is not None and aasr is not None:

print(f"{method\_name:<15}\t{aasr:.2f}\t\t{ma:.2f}")

server\_socket.close()

import socket

import numpy as np

import tensorflow as tf

import threading

import time

import random

def create\_model(input\_shape):

model = tf.keras.Sequential([

tf.keras.layers.Input(shape=(input\_shape,)),

tf.keras.layers.Dense(128, activation='relu'),

tf.keras.layers.Dense(10, activation='softmax')

])

model.compile(optimizer='sgd', loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

return model

def amsparse(delta\_w, layer\_shapes):

amsparse\_updates = []

start = 0

for shape in layer\_shapes:

size = np.prod(shape)

layer\_delta = delta\_w[start:start + size].reshape(shape)

layer\_max = np.max(np.abs(layer\_delta), axis=tuple(range(len(shape) - 1)))

signs = np.sign(layer\_delta)

amsparse\_layer = signs \* layer\_max

amsparse\_updates.append(amsparse\_layer.flatten())

start += size

return np.concatenate(amsparse\_updates)

def generate\_trigger\_pattern(shape, trigger\_type, trigger\_size):

if trigger\_type == 'pixel':

trigger = np.zeros(shape)

trigger[:trigger\_size:2] = 1.0

elif trigger\_type == 'square':

trigger = np.zeros(shape)

trigger[:trigger\_size] = 1.0

elif trigger\_type == 'cross':

trigger = np.zeros(shape)

trigger[trigger\_size//2] = 1.0

else:

trigger = np.random.choice([0, 1], size=shape, p=[0.7, 0.3]).astype(np.float32)

return trigger

def apply\_trigger(x, trigger\_pattern, trigger\_size):

triggered\_x = x.copy()

triggered\_x[:trigger\_size] = np.maximum(triggered\_x[:trigger\_size], trigger\_pattern[:trigger\_size])

return triggered\_x

def send\_data(client\_socket, data):

try:

data\_bytes = data.tobytes()

size = np.array([len(data\_bytes)], dtype=np.int64).tobytes()

client\_socket.sendall(size + data\_bytes)

except socket.error:

raise

def receive\_data(client\_socket):

try:

client\_socket.settimeout(120.0)

size\_data = client\_socket.recv(8)

if not size\_data:

raise socket.error("Server disconnected")

data\_size = np.frombuffer(size\_data, dtype=np.int64)[0]

received\_data = bytearray()

while len(received\_data) < data\_size:

packet = client\_socket.recv(min(4096, data\_size - len(received\_data)))

if not packet:

raise socket.error("Server disconnected during data transfer")

received\_data.extend(packet)

return np.frombuffer(received\_data, dtype=np.float32)

except socket.error:

raise

def client\_thread(client\_id, server\_address, T, batch\_size, input\_shape, dataset\_name):

samples\_per\_client = 6000 if dataset\_name != 'CIFAR10' else 5000

for method\_idx in range(8):

client\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

retry\_count = 0

max\_retries = 10

while retry\_count < max\_retries:

try:

client\_socket.connect(server\_address)

print(f"Client {client\_id} connected to {server\_address} for method {method\_idx}")

break

except ConnectionRefusedError:

retry\_count += 1

time.sleep(2)

if retry\_count == max\_retries:

print(f"Client {client\_id} failed to connect after {max\_retries} retries")

return

try:

client\_data\_flat = receive\_data(client\_socket)

data\_size = samples\_per\_client \* input\_shape

expected\_size = data\_size + samples\_per\_client + 1

if client\_data\_flat.size != expected\_size:

print(f"Client {client\_id} data size mismatch: expected {expected\_size}, got {client\_data\_flat.size}")

client\_socket.close()

return

x\_client = client\_data\_flat[:data\_size].reshape(samples\_per\_client, input\_shape)

y\_client = client\_data\_flat[data\_size:data\_size + samples\_per\_client].astype(np.int32)

is\_malicious = bool(client\_data\_flat[-1])

global\_weights\_flat = receive\_data(client\_socket)

model = create\_model(input\_shape)

global\_weights = [

global\_weights\_flat[i:i + w.size].reshape(w.shape)

for i, w in zip(np.cumsum([0] + [w.size for w in model.get\_weights()[:-1]]), model.get\_weights())

]

model.set\_weights(global\_weights)

layer\_shapes = [w.shape for w in model.get\_weights()]

trigger\_type = random.choice(['pixel', 'square', 'cross', 'random'])

trigger\_size = 5\*28 if dataset\_name != 'CIFAR10' else 8\*32

trigger\_pattern = generate\_trigger\_pattern(input\_shape, trigger\_type, trigger\_size)

for t in range(T):

try:

if is\_malicious:

n\_poison = int(len(x\_client) \* 0.46875)

poisoned\_data = x\_client.copy()

if t % 5 == 0:

trigger\_type = random.choice(['pixel', 'square', 'cross', 'random'])

trigger\_pattern = generate\_trigger\_pattern(input\_shape, trigger\_type, trigger\_size)

poisoned\_data[:n\_poison] = np.array([

apply\_trigger(x, trigger\_pattern, trigger\_size)

for x in poisoned\_data[:n\_poison]

])

poisoned\_labels = y\_client.copy()

poisoned\_labels[:n\_poison] = 7

model.fit(poisoned\_data, poisoned\_labels, epochs=1, batch\_size=batch\_size, verbose=0)

else:

model.fit(x\_client, y\_client, epochs=1, batch\_size=batch\_size, verbose=0)

new\_weights\_flat = np.concatenate([w.flatten() for w in model.get\_weights()]).astype(np.float32)

delta\_w = new\_weights\_flat - global\_weights\_flat

delta\_w\_am = amsparse(delta\_w, layer\_shapes)

updates = np.concatenate([delta\_w, delta\_w\_am]).astype(np.float32)

send\_data(client\_socket, updates)

global\_weights\_flat = receive\_data(client\_socket)

global\_weights = [

global\_weights\_flat[i:i + w.size].reshape(w.shape)

for i, w in zip(np.cumsum([0] + [w.size for w in model.get\_weights()[:-1]]), model.get\_weights())

]

model.set\_weights(global\_weights)

except socket.error:

print(f"Client {client\_id} disconnected during round {t+1}")

break

except Exception as e:

print(f"Client {client\_id} error: {e}")

finally:

try:

client\_socket.close()

except:

pass

# Configuration

server\_address = ('172.16.24.61', 9999)

T = 10

batch\_size = 64

n\_clients = 5

dataset\_configs = {

'MNIST': 784,

'FMNIST': 784,

'CIFAR10': 3072

}

for dataset\_name, input\_shape in dataset\_configs.items():

threads = []

for i in range(n\_clients):

thread = threading.Thread(target=client\_thread, args=(i, server\_address, T, batch\_size, input\_shape, dataset\_name))

threads.append(thread)

thread.start()

for thread in threads:

thread.join()

(tf\_env) C:\Users\login01\Documents>python server3.py

Server listening at ('172.16.24.61', 9999)

Dataset: MNIST, NIR: 0.25

Method: RFBDS

Client 0 connected from ('172.16.24.60', 64846)

Client 1 connected from ('172.16.24.60', 64847)

Client 2 connected from ('172.16.24.60', 64850)

Client 3 connected from ('172.16.24.60', 64848)

Client 4 connected from ('172.16.24.60', 64849)

Round 10/10 - MA: 74.48%, AASR: 10.75%

Method: PrivRFBDS

Client 0 connected from ('172.16.24.60', 64851)

Client 1 connected from ('172.16.24.60', 64852)

Client 2 connected from ('172.16.24.60', 64853)

Client 3 connected from ('172.16.24.60', 64854)

Client 4 connected from ('172.16.24.60', 64855)

Round 10/10 - MA: 11.62%, AASR: 0.00%

Method: FedAvg

Client 0 connected from ('172.16.24.60', 64856)

Client 1 connected from ('172.16.24.60', 64857)

Client 2 connected from ('172.16.24.60', 64858)

Client 3 connected from ('172.16.24.60', 64859)

Client 4 connected from ('172.16.24.60', 64860)

Round 10/10 - MA: 76.54%, AASR: 84.03%

Method: Median

Client 0 connected from ('172.16.24.60', 64863)

Client 1 connected from ('172.16.24.60', 64864)

Client 2 connected from ('172.16.24.60', 64865)

Client 3 connected from ('172.16.24.60', 64866)

Client 4 connected from ('172.16.24.60', 64867)

Round 10/10 - MA: 72.57%, AASR: 2.37%

Method: Trimmed-Mean

Client 0 connected from ('172.16.24.60', 64869)

Client 1 connected from ('172.16.24.60', 64870)

Client 2 connected from ('172.16.24.60', 64871)

Client 3 connected from ('172.16.24.60', 64872)

Client 4 connected from ('172.16.24.60', 64873)

Round 10/10 - MA: 76.91%, AASR: 100.00%

Method: FedCC

Client 0 connected from ('172.16.24.60', 64875)

Client 1 connected from ('172.16.24.60', 64876)

Client 2 connected from ('172.16.24.60', 64877)

Client 3 connected from ('172.16.24.60', 64878)

Client 4 connected from ('172.16.24.60', 64879)

Round 10/10 - MA: 76.46%, AASR: 91.98%

Method: RDFL

Client 0 connected from ('172.16.24.60', 64880)

Client 1 connected from ('172.16.24.60', 64881)

Client 2 connected from ('172.16.24.60', 64882)

Client 3 connected from ('172.16.24.60', 64883)

Client 4 connected from ('172.16.24.60', 64884)

Round 10/10 - MA: 76.21%, AASR: 98.56%

Method: FLAME

Client 0 connected from ('172.16.24.60', 64886)

Client 1 connected from ('172.16.24.60', 64887)

Client 2 connected from ('172.16.24.60', 64888)

Client 3 connected from ('172.16.24.60', 64889)

Client 4 connected from ('172.16.24.60', 64890)

Round 10/10 - MA: 76.55%, AASR: 5.07%

Results for Dataset: MNIST, NIR: 0.25, AR: 40.0%, PDR: 46.875%

Method AASR (%) MA (%)

----------------------------------------

RFBDS 10.75 74.48

PrivRFBDS 0.00 11.62

FedAvg 84.03 76.54

Median 2.37 72.57

Trimmed-Mean 100.00 76.91

FedCC 91.98 76.46

RDFL 98.56 76.21

FLAME 5.07 76.55

Dataset: FMNIST, NIR: 0.25

Method: RFBDS

Client 0 connected from ('172.16.24.60', 64891)

Client 1 connected from ('172.16.24.60', 64892)

Client 2 connected from ('172.16.24.60', 64893)

Client 3 connected from ('172.16.24.60', 64894)

Client 4 connected from ('172.16.24.60', 64895)

Round 10/10 - MA: 70.31%, AASR: 95.48%

Method: PrivRFBDS

Client 0 connected from ('172.16.24.60', 64897)

Client 1 connected from ('172.16.24.60', 64898)

Client 2 connected from ('172.16.24.60', 64899)

Client 3 connected from ('172.16.24.60', 64900)

Client 4 connected from ('172.16.24.60', 64901)

Round 10/10 - MA: 57.23%, AASR: 23.68%

Method: FedAvg

Client 0 connected from ('172.16.24.60', 64902)

Client 1 connected from ('172.16.24.60', 64903)

Client 2 connected from ('172.16.24.60', 64904)

Client 3 connected from ('172.16.24.60', 64905)

Client 4 connected from ('172.16.24.60', 64906)

Round 10/10 - MA: 71.24%, AASR: 89.40%

Method: Median

Client 0 connected from ('172.16.24.60', 64907)

Client 1 connected from ('172.16.24.60', 64908)

Client 2 connected from ('172.16.24.60', 64909)

Client 3 connected from ('172.16.24.60', 64910)

Client 4 connected from ('172.16.24.60', 64911)

Round 10/10 - MA: 72.08%, AASR: 11.71%

Method: Trimmed-Mean

Client 0 connected from ('172.16.24.60', 64913)

Client 1 connected from ('172.16.24.60', 64914)

Client 2 connected from ('172.16.24.60', 64915)

Client 3 connected from ('172.16.24.60', 64916)

Client 4 connected from ('172.16.24.60', 64917)

Round 10/10 - MA: 71.91%, AASR: 36.29%

Method: FedCC

Client 0 connected from ('172.16.24.60', 64919)

Client 1 connected from ('172.16.24.60', 64920)

Client 2 connected from ('172.16.24.60', 64921)

Client 3 connected from ('172.16.24.60', 64922)

Client 4 connected from ('172.16.24.60', 64923)

Round 10/10 - MA: 71.05%, AASR: 34.96%

Method: RDFL

Client 0 connected from ('172.16.24.60', 64924)

Client 1 connected from ('172.16.24.60', 64925)

Client 2 connected from ('172.16.24.60', 64926)

Client 3 connected from ('172.16.24.60', 64927)

Client 4 connected from ('172.16.24.60', 64928)

Round 10/10 - MA: 71.82%, AASR: 11.93%

Method: FLAME

Client 0 connected from ('172.16.24.60', 64929)

Client 1 connected from ('172.16.24.60', 64930)

Client 2 connected from ('172.16.24.60', 64931)

Client 3 connected from ('172.16.24.60', 64932)

Client 4 connected from ('172.16.24.60', 64933)

Round 10/10 - MA: 70.75%, AASR: 16.44%

Results for Dataset: FMNIST, NIR: 0.25, AR: 40.0%, PDR: 46.875%

Method AASR (%) MA (%)

----------------------------------------

RFBDS 95.48 70.31

PrivRFBDS 23.68 57.23

FedAvg 89.40 71.24

Median 11.71 72.08

Trimmed-Mean 36.29 71.91

FedCC 34.96 71.05

RDFL 11.93 71.82

FLAME 16.44 70.75

Dataset: CIFAR10, NIR: 0.25

Method: RFBDS

Client 0 connected from ('172.16.24.60', 64934)

Client 1 connected from ('172.16.24.60', 64935)

Client 2 connected from ('172.16.24.60', 64936)

Client 3 connected from ('172.16.24.60', 64937)

Client 4 connected from ('172.16.24.60', 64938)

Round 10/10 - MA: 26.85%, AASR: 89.22%

Method: PrivRFBDS

Client 0 connected from ('172.16.24.60', 64940)

Client 1 connected from ('172.16.24.60', 64941)

Client 2 connected from ('172.16.24.60', 64942)

Client 3 connected from ('172.16.24.60', 64943)

Client 4 connected from ('172.16.24.60', 64944)

Round 10/10 - MA: 13.21%, AASR: 17.48%

Method: FedAvg

Client 0 connected from ('172.16.24.60', 64945)

Client 1 connected from ('172.16.24.60', 64946)

Client 2 connected from ('172.16.24.60', 64947)

Client 3 connected from ('172.16.24.60', 64948)

Client 4 connected from ('172.16.24.60', 64949)

Round 10/10 - MA: 31.79%, AASR: 62.43%

Method: Median

Client 0 connected from ('172.16.24.60', 64955)

Client 1 connected from ('172.16.24.60', 64956)

Client 2 connected from ('172.16.24.60', 64957)

Client 3 connected from ('172.16.24.60', 64958)

Client 4 connected from ('172.16.24.60', 64959)

Round 10/10 - MA: 29.81%, AASR: 71.57%

Method: Trimmed-Mean

Client 0 connected from ('172.16.24.60', 64960)

Client 1 connected from ('172.16.24.60', 64961)

Client 2 connected from ('172.16.24.60', 64962)

Client 3 connected from ('172.16.24.60', 64963)

Client 4 connected from ('172.16.24.60', 64964)

Round 10/10 - MA: 27.66%, AASR: 87.04%

Method: FedCC

Client 0 connected from ('172.16.24.60', 64966)

Client 1 connected from ('172.16.24.60', 64967)

Client 2 connected from ('172.16.24.60', 64968)

Client 3 connected from ('172.16.24.60', 64969)

Client 4 connected from ('172.16.24.60', 64970)

Round 10/10 - MA: 28.62%, AASR: 93.36%

Method: RDFL

Client 0 connected from ('172.16.24.60', 64973)

Client 1 connected from ('172.16.24.60', 64974)

Client 2 connected from ('172.16.24.60', 64975)

Client 3 connected from ('172.16.24.60', 64976)

Client 4 connected from ('172.16.24.60', 64977)

Round 10/10 - MA: 29.97%, AASR: 92.47%

Method: FLAME

Client 0 connected from ('172.16.24.60', 64978)

Client 1 connected from ('172.16.24.60', 64979)

Client 2 connected from ('172.16.24.60', 64980)

Client 3 connected from ('172.16.24.60', 64981)

Client 4 connected from ('172.16.24.60', 64982)

Round 10/10 - MA: 26.09%, AASR: 93.81%

Results for Dataset: CIFAR10, NIR: 0.25, AR: 40.0%, PDR: 46.875%

Method AASR (%) MA (%)

----------------------------------------

RFBDS 89.22 26.85

PrivRFBDS 17.48 13.21

FedAvg 62.43 31.79

Median 71.57 29.81

Trimmed-Mean 87.04 27.66

FedCC 93.36 28.62

RDFL 92.47 29.97

FLAME 93.81 26.09

[22bit005@mepcolinux ~]$cat 0.75.txt

import os

os.environ['TF\_CPP\_MIN\_LOG\_LEVEL'] = '2'

import socket

import numpy as np

from sklearn.cluster import OPTICS, KMeans, HDBSCAN

from sklearn.decomposition import PCA

import tensorflow as tf

from scipy.stats import dirichlet

import random

import time

# Load datasets

(x\_train\_mnist, y\_train\_mnist), (x\_test\_mnist, y\_test\_mnist) = tf.keras.datasets.mnist.load\_data()

x\_train\_mnist = x\_train\_mnist.reshape(-1, 28\*28).astype(np.float32) / 255.0

x\_test\_mnist = x\_test\_mnist.reshape(-1, 28\*28).astype(np.float32) / 255.0

(x\_train\_fmnist, y\_train\_fmnist), (x\_test\_fmnist, y\_test\_fmnist) = tf.keras.datasets.fashion\_mnist.load\_data()

x\_train\_fmnist = x\_train\_fmnist.reshape(-1, 28\*28).astype(np.float32) / 255.0

x\_test\_fmnist = x\_test\_fmnist.reshape(-1, 28\*28).astype(np.float32) / 255.0

(x\_train\_cifar, y\_train\_cifar), (x\_test\_cifar, y\_test\_cifar) = tf.keras.datasets.cifar10.load\_data()

x\_train\_cifar = x\_train\_cifar.reshape(-1, 32\*32\*3).astype(np.float32) / 255.0

x\_test\_cifar = x\_test\_cifar.reshape(-1, 32\*32\*3).astype(np.float32) / 255.0

y\_train\_cifar = y\_train\_cifar.flatten()

y\_test\_cifar = y\_test\_cifar.flatten()

datasets = {

'MNIST': (x\_train\_mnist, y\_train\_mnist, x\_test\_mnist, y\_test\_mnist, 784),

'FMNIST': (x\_train\_fmnist, y\_train\_fmnist, x\_test\_fmnist, y\_test\_fmnist, 784),

'CIFAR10': (x\_train\_cifar, y\_train\_cifar, x\_test\_cifar, y\_test\_cifar, 3072)

}

def create\_model(input\_shape):

model = tf.keras.Sequential([

tf.keras.layers.Input(shape=(input\_shape,)),

tf.keras.layers.Dense(128, activation='relu'),

tf.keras.layers.Dense(10, activation='softmax')

])

model.compile(optimizer='sgd', loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

return model

def secret\_share(data, n\_parties=2):

shares = [np.random.uniform(-1000, 1000, size=data.shape).astype(np.float32) for \_ in range(n\_parties - 1)]

shares.append(data - sum(shares))

return shares

def reconstruct\_shares(shares):

return sum(shares)

def amsparse(delta\_w, layer\_shapes):

amsparse\_updates = []

start = 0

for shape in layer\_shapes:

size = np.prod(shape)

layer\_delta = delta\_w[start:start + size].reshape(shape)

layer\_max = np.max(np.abs(layer\_delta), axis=tuple(range(len(shape) - 1)))

signs = np.sign(layer\_delta)

amsparse\_layer = signs \* layer\_max

amsparse\_updates.append(amsparse\_layer.flatten())

start += size

return np.concatenate(amsparse\_updates)

def auto\_optics(updates):

if len(updates) < 2:

return np.arange(len(updates))

try:

pca = PCA(n\_components=min(50, updates.shape[1]))

updates\_reduced = pca.fit\_transform(updates)

dists = np.linalg.norm(updates\_reduced - updates\_reduced.mean(axis=0), axis=1)

k = min(max(2, len(updates) // 3), len(updates) - 1)

sorted\_dists = np.sort(dists)[:k]

eps = np.median(sorted\_dists) \* 1.5 # Adjusted for tighter clustering

clustering = OPTICS(min\_samples=k, eps=eps, metric='euclidean').fit(updates\_reduced)

labels = clustering.labels\_

majority\_group = np.where(labels != -1)[0]

return majority\_group if len(majority\_group) > 0 else np.arange(len(updates))

except:

return np.arange(len(updates))

def adaptive\_clipping(updates, benign\_indices):

if len(updates) == 0:

return np.zeros\_like(updates)

norms = np.linalg.norm(updates, axis=1)

rho = np.median(norms[benign\_indices]) if len(benign\_indices) > 0 else 1.0

clipped\_updates = []

for i in benign\_indices:

update = updates[i]

norm = norms[i]

clipped = update \* min(1, rho / norm) if norm > 0 else update

clipped\_updates.append(clipped)

return np.array(clipped\_updates) if clipped\_updates else updates

def priv\_shuffling(updates\_shares, n\_clients):

pi\_0 = np.random.permutation(n\_clients)

pi\_1 = np.random.permutation(n\_clients)

shuffled\_shares = [[updates\_shares[pi\_0[i]][0], updates\_shares[pi\_1[i]][1]] for i in range(n\_clients)]

return shuffled\_shares

def priv\_selection(shuffled\_shares, m):

norms\_shares = [secret\_share(np.linalg.norm(reconstruct\_shares(shares))) for shares in shuffled\_shares]

norms = [reconstruct\_shares(ns) for ns in norms\_shares]

sorted\_indices = np.argsort(norms)[:m]

return [shuffled\_shares[i] for i in sorted\_indices]

def priv\_optics(updates\_shares):

updates = np.array([reconstruct\_shares(shares) for shares in updates\_shares])

if len(updates) < 2:

return secret\_share(np.arange(len(updates)))

try:

pca = PCA(n\_components=min(50, updates.shape[1]))

updates\_reduced = pca.fit\_transform(updates)

dists = np.linalg.norm(updates\_reduced - updates\_reduced.mean(axis=0), axis=1)

k = min(max(2, len(updates) // 3), len(updates) - 1)

sorted\_dists = np.sort(dists)[:k]

eps = np.median(sorted\_dists) \* 1.5

clustering = OPTICS(min\_samples=k, eps=eps, metric='euclidean').fit(updates\_reduced)

labels = clustering.labels\_

benign\_indices = np.where(labels != -1)[0]

return secret\_share(benign\_indices if len(benign\_indices) > 0 else np.arange(len(updates)))

except:

return secret\_share(np.arange(len(updates)))

def priv\_clipping(updates\_shares, benign\_indices\_shares):

updates = np.array([reconstruct\_shares(shares) for shares in updates\_shares])

benign\_indices = reconstruct\_shares(benign\_indices\_shares).astype(int)

norms = np.linalg.norm(updates, axis=1)

rho\_shares = secret\_share(np.median(norms[benign\_indices]) if len(benign\_indices) > 0 else 1.0)

clipped\_shares = []

for i in benign\_indices:

update\_shares = updates\_shares[i]

norm\_shares = secret\_share(norms[i])

rho\_val = reconstruct\_shares(rho\_shares)

norm\_val = reconstruct\_shares(norm\_shares)

factor = min(1, rho\_val / norm\_val) if norm\_val > 0 else 1.0

clipped\_shares.append([share \* factor for share in update\_shares])

return clipped\_shares if clipped\_shares else updates\_shares

def priv\_aggregation(clipped\_shares):

if not clipped\_shares:

return secret\_share(np.zeros\_like(reconstruct\_shares(clipped\_shares[0])))

clipped\_updates = np.array([reconstruct\_shares(shares) for shares in clipped\_shares])

return secret\_share(np.mean(clipped\_updates, axis=0))

def fedavg(updates):

return np.mean(updates, axis=0) if len(updates) > 0 else np.zeros\_like(updates[0])

def median(updates):

return np.median(updates, axis=0) if len(updates) > 0 else np.zeros\_like(updates[0])

def trimmed\_mean(updates, trim\_ratio=0.1):

if len(updates) < 3:

return np.mean(updates, axis=0) if len(updates) > 0 else np.zeros\_like(updates[0])

n\_trim = min(int(len(updates) \* trim\_ratio), len(updates) // 2 - 1)

if n\_trim == 0:

return np.mean(updates, axis=0)

sorted\_updates = np.sort(updates, axis=0)

trimmed = sorted\_updates[n\_trim:-n\_trim]

return np.mean(trimmed, axis=0)

def fedcc(updates):

if len(updates) < 2:

return np.mean(updates, axis=0) if len(updates) > 0 else np.zeros\_like(updates[0])

try:

pca = PCA(n\_components=min(50, updates.shape[1]))

updates\_reduced = pca.fit\_transform(updates)

clustering = KMeans(n\_clusters=2, n\_init=10, random\_state=42).fit(updates\_reduced)

labels = clustering.labels\_

benign\_cluster = np.argmax(np.bincount(labels))

benign\_indices = np.where(labels == benign\_cluster)[0]

if len(benign\_indices) == 0:

return np.mean(updates, axis=0)

return np.mean(updates[benign\_indices], axis=0)

except:

return np.mean(updates, axis=0)

def rdfl(updates):

if len(updates) < 2:

return np.mean(updates, axis=0) if len(updates) > 0 else np.zeros\_like(updates[0])

try:

pca = PCA(n\_components=min(50, updates.shape[1]))

updates\_reduced = pca.fit\_transform(updates)

clustering = HDBSCAN(min\_cluster\_size=2, min\_samples=2).fit(updates\_reduced)

labels = clustering.labels\_

benign\_indices = np.where(labels != -1)[0]

return np.mean(updates[benign\_indices], axis=0) if len(benign\_indices) > 0 else np.mean(updates, axis=0)

except:

return np.mean(updates, axis=0)

def flame(updates):

if len(updates) < 2:

return np.mean(updates, axis=0) if len(updates) > 0 else np.zeros\_like(updates[0])

try:

norms = np.linalg.norm(updates, axis=1)

rho = np.median(norms)

clipped = [u \* min(1, rho / np.linalg.norm(u)) if np.linalg.norm(u) > 0 else u for u in updates]

pca = PCA(n\_components=min(50, updates.shape[1]))

clipped\_reduced = pca.fit\_transform(clipped)

clustering = HDBSCAN(min\_cluster\_size=2, min\_samples=2).fit(clipped\_reduced)

labels = clustering.labels\_

benign\_indices = np.where(labels != -1)[0]

return np.mean(np.array(clipped)[benign\_indices], axis=0) if len(benign\_indices) > 0 else np.mean(clipped, axis=0)

except:

return np.mean(updates, axis=0)

def generate\_trigger\_pattern(shape, trigger\_type, trigger\_size):

if trigger\_type == 'pixel':

trigger = np.zeros(shape)

trigger[:trigger\_size:2] = 1.0

elif trigger\_type == 'square':

trigger = np.zeros(shape)

trigger[:trigger\_size] = 1.0

elif trigger\_type == 'cross':

trigger = np.zeros(shape)

trigger[trigger\_size//2] = 1.0

else:

trigger = np.random.choice([0, 1], size=shape, p=[0.7, 0.3]).astype(np.float32)

return trigger

def apply\_trigger(x, trigger\_pattern, trigger\_size):

triggered\_x = x.copy()

triggered\_x[:trigger\_size] = np.maximum(triggered\_x[:trigger\_size], trigger\_pattern[:trigger\_size])

return triggered\_x

def send\_data(client\_socket, data):

try:

data\_bytes = data.tobytes()

size = np.array([len(data\_bytes)], dtype=np.int64).tobytes()

client\_socket.sendall(size + data\_bytes)

except socket.error:

raise

def receive\_data(client\_socket):

try:

client\_socket.settimeout(120.0) # Increased timeout

size\_data = client\_socket.recv(8)

if not size\_data:

raise socket.error("Client disconnected")

data\_size = np.frombuffer(size\_data, dtype=np.int64)[0]

received\_data = bytearray()

while len(received\_data) < data\_size:

packet = client\_socket.recv(min(4096, data\_size - len(received\_data)))

if not packet:

raise socket.error("Client disconnected during data transfer")

received\_data.extend(packet)

return np.frombuffer(received\_data, dtype=np.float32)

except socket.error:

raise

# Server setup

server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

server\_socket.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)

server\_address = ('172.16.24.61', 9999)

server\_socket.bind(server\_address)

server\_socket.listen(5)

print("Server listening at", server\_address)

T = 10

n\_clients = 5

m\_selected = 5

AR = 0.4

PDR = 0.46875

NIR = 0.75

methods = {

'RFBDS': lambda updates: np.mean(adaptive\_clipping(updates, auto\_optics(np.array([amsparse(u, layer\_shapes) for u in updates]))), axis=0),

'PrivRFBDS': lambda updates\_shares: reconstruct\_shares(priv\_aggregation(

priv\_clipping(

priv\_selection(

priv\_shuffling(updates\_shares, len(updates\_shares)), m\_selected),

priv\_optics(

priv\_selection(

priv\_shuffling(updates\_shares, len(updates\_shares)), m\_selected))))),

'FedAvg': fedavg,

'Median': median,

'Trimmed-Mean': trimmed\_mean,

'FedCC': fedcc,

'RDFL': rdfl,

'FLAME': flame

}

for dataset\_name in ['MNIST', 'FMNIST', 'CIFAR10']:

x\_train, y\_train, x\_test, y\_test, input\_shape = datasets[dataset\_name]

global\_model = create\_model(input\_shape)

global\_weights = global\_model.get\_weights()

layer\_shapes = [w.shape for w in global\_weights]

global\_weights\_flat = np.concatenate([w.flatten() for w in global\_weights]).astype(np.float32)

samples\_per\_client = 6000 if dataset\_name != 'CIFAR10' else 5000

print(f"\nDataset: {dataset\_name}, NIR: {NIR}")

client\_data = []

indices = np.arange(len(x\_train))

label\_dist = dirichlet.rvs([1.0 - NIR] \* 10, size=n\_clients)

for i in range(n\_clients):

client\_indices = []

probs = label\_dist[i]

for label in range(10):

label\_indices = np.where(y\_train == label)[0]

n\_samples = int(samples\_per\_client \* probs[label])

if len(label\_indices) > 0:

selected = np.random.choice(label\_indices, min(n\_samples, len(label\_indices)), replace=False)

client\_indices.extend(selected)

client\_indices = np.array(client\_indices)

if len(client\_indices) < samples\_per\_client:

extra\_indices = np.random.choice(indices, samples\_per\_client - len(client\_indices), replace=False)

client\_indices = np.concatenate([client\_indices, extra\_indices])

elif len(client\_indices) > samples\_per\_client:

client\_indices = client\_indices[:samples\_per\_client]

np.random.shuffle(client\_indices)

client\_data.append((x\_train[client\_indices], y\_train[client\_indices], i >= int(n\_clients \* (1 - AR))))

results = {method\_name: {'MA': None, 'AASR': None} for method\_name in methods}

for method\_name, method in methods.items():

print(f"\nMethod: {method\_name}")

client\_sockets = []

try:

for i in range(n\_clients):

client\_socket, addr = server\_socket.accept()

client\_socket.settimeout(120.0)

print(f"Client {i} connected from {addr}")

client\_sockets.append(client\_socket)

x\_client, y\_client, is\_malicious = client\_data[i]

client\_data\_flat = np.concatenate([

x\_client.flatten(),

y\_client.astype(np.float32),

np.array([float(is\_malicious)], dtype=np.float32)

]).astype(np.float32)

send\_data(client\_socket, client\_data\_flat)

send\_data(client\_socket, global\_weights\_flat)

current\_weights\_flat = global\_weights\_flat.copy()

current\_model = create\_model(input\_shape)

current\_weights = [

current\_weights\_flat[i:i + w.size].reshape(w.shape)

for i, w in zip(np.cumsum([0] + [w.size for w in global\_weights[:-1]]), global\_weights)

]

current\_model.set\_weights(current\_weights)

for t in range(T):

client\_updates = []

client\_updates\_shares = []

active\_sockets = []

for i, client\_socket in enumerate(client\_sockets):

if client\_socket is None:

continue

try:

updates = receive\_data(client\_socket)

split\_idx = len(updates) // 2

delta\_w = updates[:split\_idx]

delta\_w\_am = updates[split\_idx:]

client\_updates.append(delta\_w)

client\_updates\_shares.append(secret\_share(delta\_w\_am))

active\_sockets.append(client\_socket)

except socket.error:

client\_sockets[i] = None

client\_sockets = active\_sockets

if not client\_updates:

print(f"Round {t+1}/{T} - No active clients")

continue

try:

if method\_name == 'PrivRFBDS':

aggregated\_update = method(client\_updates\_shares)

else:

aggregated\_update = method(np.array(client\_updates))

current\_weights\_flat += aggregated\_update

current\_weights = [

current\_weights\_flat[i:i + w.size].reshape(w.shape)

for i, w in zip(np.cumsum([0] + [w.size for w in global\_weights[:-1]]), global\_weights)

]

current\_model.set\_weights(current\_weights)

loss, ma = current\_model.evaluate(x\_test, y\_test, verbose=0)

trigger\_type = random.choice(['pixel', 'square', 'cross', 'random'])

trigger\_size = 5\*28 if dataset\_name != 'CIFAR10' else 8\*32

trigger\_pattern = generate\_trigger\_pattern(input\_shape, trigger\_type, trigger\_size)

x\_test\_backdoor = np.array([apply\_trigger(x, trigger\_pattern, trigger\_size) for x in x\_test])

y\_test\_backdoor = np.ones\_like(y\_test) \* 7

\_, aasr = current\_model.evaluate(x\_test\_backdoor, y\_test\_backdoor, verbose=0)

if t == T - 1:

results[method\_name]['MA'] = ma \* 100

results[method\_name]['AASR'] = aasr \* 100

print(f"Round {t+1}/{T} - MA: {ma\*100:.2f}%, AASR: {aasr\*100:.2f}%")

except Exception as e:

print(f"Round {t+1}/{T} - Aggregation error: {e}")

continue

for client\_socket in client\_sockets:

if client\_socket:

try:

send\_data(client\_socket, current\_weights\_flat)

except socket.error:

client\_sockets[client\_sockets.index(client\_socket)] = None

except Exception as e:

print(f"Error during method {method\_name}: {e}")

finally:

for client\_socket in client\_sockets:

if client\_socket:

try:

client\_socket.close()

except:

pass

print(f"\nResults for Dataset: {dataset\_name}, NIR: {NIR}, AR: {AR\*100}%, PDR: {PDR\*100}%")

print("Method\t\tAASR (%)\tMA (%)")

print("-" \* 40)

for method\_name in methods:

ma = results[method\_name]['MA']

aasr = results[method\_name]['AASR']

if ma is not None and aasr is not None:

print(f"{method\_name:<15}\t{aasr:.2f}\t\t{ma:.2f}")

server\_socket.close()

import socket

import numpy as np

import tensorflow as tf

import threading

import time

import random

def create\_model(input\_shape):

model = tf.keras.Sequential([

tf.keras.layers.Input(shape=(input\_shape,)),

tf.keras.layers.Dense(128, activation='relu'),

tf.keras.layers.Dense(10, activation='softmax')

])

model.compile(optimizer='sgd', loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

return model

def amsparse(delta\_w, layer\_shapes):

amsparse\_updates = []

start = 0

for shape in layer\_shapes:

size = np.prod(shape)

layer\_delta = delta\_w[start:start + size].reshape(shape)

layer\_max = np.max(np.abs(layer\_delta), axis=tuple(range(len(shape) - 1)))

signs = np.sign(layer\_delta)

amsparse\_layer = signs \* layer\_max

amsparse\_updates.append(amsparse\_layer.flatten())

start += size

return np.concatenate(amsparse\_updates)

def generate\_trigger\_pattern(shape, trigger\_type, trigger\_size):

if trigger\_type == 'pixel':

trigger = np.zeros(shape)

trigger[:trigger\_size:2] = 1.0

elif trigger\_type == 'square':

trigger = np.zeros(shape)

trigger[:trigger\_size] = 1.0

elif trigger\_type == 'cross':

trigger = np.zeros(shape)

trigger[trigger\_size//2] = 1.0

else:

trigger = np.random.choice([0, 1], size=shape, p=[0.7, 0.3]).astype(np.float32)

return trigger

def apply\_trigger(x, trigger\_pattern, trigger\_size):

triggered\_x = x.copy()

triggered\_x[:trigger\_size] = np.maximum(triggered\_x[:trigger\_size], trigger\_pattern[:trigger\_size])

return triggered\_x

def send\_data(client\_socket, data):

try:

data\_bytes = data.tobytes()

size = np.array([len(data\_bytes)], dtype=np.int64).tobytes()

client\_socket.sendall(size + data\_bytes)

except socket.error:

raise

def receive\_data(client\_socket):

try:

client\_socket.settimeout(120.0)

size\_data = client\_socket.recv(8)

if not size\_data:

raise socket.error("Server disconnected")

data\_size = np.frombuffer(size\_data, dtype=np.int64)[0]

received\_data = bytearray()

while len(received\_data) < data\_size:

packet = client\_socket.recv(min(4096, data\_size - len(received\_data)))

if not packet:

raise socket.error("Server disconnected during data transfer")

received\_data.extend(packet)

return np.frombuffer(received\_data, dtype=np.float32)

except socket.error:

raise

def client\_thread(client\_id, server\_address, T, batch\_size, input\_shape, dataset\_name):

samples\_per\_client = 6000 if dataset\_name != 'CIFAR10' else 5000

for method\_idx in range(8):

client\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

retry\_count = 0

max\_retries = 10

while retry\_count < max\_retries:

try:

client\_socket.connect(server\_address)

print(f"Client {client\_id} connected to {server\_address} for method {method\_idx}")

break

except ConnectionRefusedError:

retry\_count += 1

time.sleep(2)

if retry\_count == max\_retries:

print(f"Client {client\_id} failed to connect after {max\_retries} retries")

return

try:

client\_data\_flat = receive\_data(client\_socket)

data\_size = samples\_per\_client \* input\_shape

expected\_size = data\_size + samples\_per\_client + 1

if client\_data\_flat.size != expected\_size:

print(f"Client {client\_id} data size mismatch: expected {expected\_size}, got {client\_data\_flat.size}")

client\_socket.close()

return

x\_client = client\_data\_flat[:data\_size].reshape(samples\_per\_client, input\_shape)

y\_client = client\_data\_flat[data\_size:data\_size + samples\_per\_client].astype(np.int32)

is\_malicious = bool(client\_data\_flat[-1])

global\_weights\_flat = receive\_data(client\_socket)

model = create\_model(input\_shape)

global\_weights = [

global\_weights\_flat[i:i + w.size].reshape(w.shape)

for i, w in zip(np.cumsum([0] + [w.size for w in model.get\_weights()[:-1]]), model.get\_weights())

]

model.set\_weights(global\_weights)

layer\_shapes = [w.shape for w in model.get\_weights()]

trigger\_type = random.choice(['pixel', 'square', 'cross', 'random'])

trigger\_size = 5\*28 if dataset\_name != 'CIFAR10' else 8\*32

trigger\_pattern = generate\_trigger\_pattern(input\_shape, trigger\_type, trigger\_size)

for t in range(T):

try:

if is\_malicious:

n\_poison = int(len(x\_client) \* 0.46875)

poisoned\_data = x\_client.copy()

if t % 5 == 0:

trigger\_type = random.choice(['pixel', 'square', 'cross', 'random'])

trigger\_pattern = generate\_trigger\_pattern(input\_shape, trigger\_type, trigger\_size)

poisoned\_data[:n\_poison] = np.array([

apply\_trigger(x, trigger\_pattern, trigger\_size)

for x in poisoned\_data[:n\_poison]

])

poisoned\_labels = y\_client.copy()

poisoned\_labels[:n\_poison] = 7

model.fit(poisoned\_data, poisoned\_labels, epochs=1, batch\_size=batch\_size, verbose=0)

else:

model.fit(x\_client, y\_client, epochs=1, batch\_size=batch\_size, verbose=0)

new\_weights\_flat = np.concatenate([w.flatten() for w in model.get\_weights()]).astype(np.float32)

delta\_w = new\_weights\_flat - global\_weights\_flat

delta\_w\_am = amsparse(delta\_w, layer\_shapes)

updates = np.concatenate([delta\_w, delta\_w\_am]).astype(np.float32)

send\_data(client\_socket, updates)

global\_weights\_flat = receive\_data(client\_socket)

global\_weights = [

global\_weights\_flat[i:i + w.size].reshape(w.shape)

for i, w in zip(np.cumsum([0] + [w.size for w in model.get\_weights()[:-1]]), model.get\_weights())

]

model.set\_weights(global\_weights)

except socket.error:

print(f"Client {client\_id} disconnected during round {t+1}")

break

except Exception as e:

print(f"Client {client\_id} error: {e}")

finally:

try:

client\_socket.close()

except:

pass

# Configuration

server\_address = ('172.16.24.61', 9999)

T = 10

batch\_size = 64

n\_clients = 5

dataset\_configs = {

'MNIST': 784,

'FMNIST': 784,

'CIFAR10': 3072

}

for dataset\_name, input\_shape in dataset\_configs.items():

threads = []

for i in range(n\_clients):

thread = threading.Thread(target=client\_thread, args=(i, server\_address, T, batch\_size, input\_shape, dataset\_name))

threads.append(thread)

thread.start()

for thread in threads:

thread.join()

(tf\_env) C:\Users\login01\Documents>python server4.py

Server listening at ('172.16.24.61', 9999)

Dataset: MNIST, NIR: 0.75

Method: RFBDS

Client 0 connected from ('172.16.24.60', 65005)

Client 1 connected from ('172.16.24.60', 65006)

Client 2 connected from ('172.16.24.60', 65007)

Client 3 connected from ('172.16.24.60', 65008)

Client 4 connected from ('172.16.24.60', 65009)

Round 10/10 - MA: 67.82%, AASR: 1.43%

Method: PrivRFBDS

Client 0 connected from ('172.16.24.60', 65010)

Client 1 connected from ('172.16.24.60', 65011)

Client 2 connected from ('172.16.24.60', 65012)

Client 3 connected from ('172.16.24.60', 65013)

Client 4 connected from ('172.16.24.60', 65014)

Round 10/10 - MA: 12.24%, AASR: 100.00%

Method: FedAvg

Client 0 connected from ('172.16.24.60', 65019)

Client 1 connected from ('172.16.24.60', 65020)

Client 2 connected from ('172.16.24.60', 65021)

Client 3 connected from ('172.16.24.60', 65022)

Client 4 connected from ('172.16.24.60', 65023)

Round 10/10 - MA: 68.31%, AASR: 1.46%

Method: Median

Client 0 connected from ('172.16.24.60', 65024)

Client 1 connected from ('172.16.24.60', 65025)

Client 2 connected from ('172.16.24.60', 65026)

Client 3 connected from ('172.16.24.60', 65027)

Client 4 connected from ('172.16.24.60', 65028)

Round 10/10 - MA: 68.79%, AASR: 13.35%

Method: Trimmed-Mean

Client 0 connected from ('172.16.24.60', 65030)

Client 1 connected from ('172.16.24.60', 65031)

Client 2 connected from ('172.16.24.60', 65032)

Client 3 connected from ('172.16.24.60', 65033)

Client 4 connected from ('172.16.24.60', 65034)

Round 10/10 - MA: 68.67%, AASR: 48.03%

Method: FedCC

Client 0 connected from ('172.16.24.60', 65035)

Client 1 connected from ('172.16.24.60', 65036)

Client 2 connected from ('172.16.24.60', 65037)

Client 3 connected from ('172.16.24.60', 65038)

Client 4 connected from ('172.16.24.60', 65039)

Round 10/10 - MA: 69.57%, AASR: 79.92%

Method: RDFL

Client 0 connected from ('172.16.24.60', 65040)

Client 1 connected from ('172.16.24.60', 65041)

Client 2 connected from ('172.16.24.60', 65042)

Client 3 connected from ('172.16.24.60', 65043)

Client 4 connected from ('172.16.24.60', 65044)

Round 10/10 - MA: 67.70%, AASR: 99.99%

Method: FLAME

Client 0 connected from ('172.16.24.60', 65045)

Client 1 connected from ('172.16.24.60', 65046)

Client 2 connected from ('172.16.24.60', 65047)

Client 3 connected from ('172.16.24.60', 65048)

Client 4 connected from ('172.16.24.60', 65049)

Round 10/10 - MA: 69.70%, AASR: 100.00%

Results for Dataset: MNIST, NIR: 0.75, AR: 40.0%, PDR: 46.875%

Method AASR (%) MA (%)

----------------------------------------

RFBDS 1.43 67.82

PrivRFBDS 100.00 12.24

FedAvg 1.46 68.31

Median 13.35 68.79

Trimmed-Mean 48.03 68.67

FedCC 79.92 69.57

RDFL 99.99 67.70

FLAME 100.00 69.70

Dataset: FMNIST, NIR: 0.75

Method: RFBDS

Client 0 connected from ('172.16.24.60', 65051)

Client 1 connected from ('172.16.24.60', 65052)

Client 2 connected from ('172.16.24.60', 65053)

Client 3 connected from ('172.16.24.60', 65054)

Client 4 connected from ('172.16.24.60', 65055)

Round 10/10 - MA: 66.17%, AASR: 50.77%

Method: PrivRFBDS

Client 0 connected from ('172.16.24.60', 65056)

Client 1 connected from ('172.16.24.60', 65057)

Client 2 connected from ('172.16.24.60', 65058)

Client 3 connected from ('172.16.24.60', 65059)

Client 4 connected from ('172.16.24.60', 65060)

Round 10/10 - MA: 45.19%, AASR: 4.16%

Method: FedAvg

Client 0 connected from ('172.16.24.60', 65061)

Client 1 connected from ('172.16.24.60', 65062)

Client 2 connected from ('172.16.24.60', 65063)

Client 3 connected from ('172.16.24.60', 65064)

Client 4 connected from ('172.16.24.60', 65065)

Round 10/10 - MA: 66.13%, AASR: 74.54%

Method: Median

Client 0 connected from ('172.16.24.60', 65066)

Client 1 connected from ('172.16.24.60', 65067)

Client 2 connected from ('172.16.24.60', 65068)

Client 3 connected from ('172.16.24.60', 65069)

Client 4 connected from ('172.16.24.60', 65070)

Round 10/10 - MA: 55.69%, AASR: 18.66%

Method: Trimmed-Mean

Client 0 connected from ('172.16.24.60', 65086)

Client 1 connected from ('172.16.24.60', 65087)

Client 2 connected from ('172.16.24.60', 65088)

Client 3 connected from ('172.16.24.60', 65089)

Client 4 connected from ('172.16.24.60', 65090)

Round 10/10 - MA: 66.62%, AASR: 86.47%

Method: FedCC

Client 0 connected from ('172.16.24.60', 65092)

Client 1 connected from ('172.16.24.60', 65093)

Client 2 connected from ('172.16.24.60', 65094)

Client 3 connected from ('172.16.24.60', 65095)

Client 4 connected from ('172.16.24.60', 65096)

Round 10/10 - MA: 66.70%, AASR: 96.56%

Method: RDFL

Client 0 connected from ('172.16.24.60', 65097)

Client 1 connected from ('172.16.24.60', 65098)

Client 2 connected from ('172.16.24.60', 65099)

Client 3 connected from ('172.16.24.60', 65100)

Client 4 connected from ('172.16.24.60', 65101)

Round 10/10 - MA: 66.77%, AASR: 80.29%

Method: FLAME

Client 0 connected from ('172.16.24.60', 65102)

Client 1 connected from ('172.16.24.60', 65103)

Client 2 connected from ('172.16.24.60', 65104)

Client 3 connected from ('172.16.24.60', 65105)

Client 4 connected from ('172.16.24.60', 65106)

Round 10/10 - MA: 67.08%, AASR: 42.90%

Results for Dataset: FMNIST, NIR: 0.75, AR: 40.0%, PDR: 46.875%

Method AASR (%) MA (%)

----------------------------------------

RFBDS 50.77 66.17

PrivRFBDS 4.16 45.19

FedAvg 74.54 66.13

Median 18.66 55.69

Trimmed-Mean 86.47 66.62

FedCC 96.56 66.70

RDFL 80.29 66.77

FLAME 42.90 67.08

Dataset: CIFAR10, NIR: 0.75

Method: RFBDS

Client 0 connected from ('172.16.24.60', 65108)

Client 1 connected from ('172.16.24.60', 65109)

Client 2 connected from ('172.16.24.60', 65110)

Client 3 connected from ('172.16.24.60', 65111)

Client 4 connected from ('172.16.24.60', 65112)

Round 10/10 - MA: 25.65%, AASR: 21.55%

Method: PrivRFBDS

Client 0 connected from ('172.16.24.60', 65114)

Client 1 connected from ('172.16.24.60', 65115)

Client 2 connected from ('172.16.24.60', 65116)

Client 3 connected from ('172.16.24.60', 65117)

Client 4 connected from ('172.16.24.60', 65118)

Round 10/10 - MA: 12.86%, AASR: 19.75%

Method: FedAvg

Client 0 connected from ('172.16.24.60', 65120)

Client 1 connected from ('172.16.24.60', 65121)

Client 2 connected from ('172.16.24.60', 65122)

Client 3 connected from ('172.16.24.60', 65123)

Client 4 connected from ('172.16.24.60', 65124)

Round 10/10 - MA: 27.87%, AASR: 17.67%

Method: Median

Client 0 connected from ('172.16.24.60', 65125)

Client 1 connected from ('172.16.24.60', 65126)

Client 2 connected from ('172.16.24.60', 65127)

Client 3 connected from ('172.16.24.60', 65128)

Client 4 connected from ('172.16.24.60', 65129)

Round 10/10 - MA: 23.67%, AASR: 86.00%

Method: Trimmed-Mean

Client 0 connected from ('172.16.24.60', 65132)

Client 1 connected from ('172.16.24.60', 65133)

Client 2 connected from ('172.16.24.60', 65134)

Client 3 connected from ('172.16.24.60', 65135)

Client 4 connected from ('172.16.24.60', 65136)

Round 10/10 - MA: 18.08%, AASR: 92.39%

Method: FedCC

Client 0 connected from ('172.16.24.60', 65137)

Client 1 connected from ('172.16.24.60', 65138)

Client 2 connected from ('172.16.24.60', 65139)

Client 3 connected from ('172.16.24.60', 65140)

Client 4 connected from ('172.16.24.60', 65141)

Round 10/10 - MA: 27.43%, AASR: 23.38%

Method: RDFL

Client 0 connected from ('172.16.24.60', 65142)

Client 1 connected from ('172.16.24.60', 65143)

Client 2 connected from ('172.16.24.60', 65144)

Client 3 connected from ('172.16.24.60', 65145)

Client 4 connected from ('172.16.24.60', 65146)

Round 10/10 - MA: 27.75%, AASR: 57.65%

Method: FLAME

Client 0 connected from ('172.16.24.60', 65148)

Client 1 connected from ('172.16.24.60', 65149)

Client 2 connected from ('172.16.24.60', 65150)

Client 3 connected from ('172.16.24.60', 65151)

Client 4 connected from ('172.16.24.60', 65152)

Round 10/10 - MA: 27.44%, AASR: 62.63%

Results for Dataset: CIFAR10, NIR: 0.75, AR: 40.0%, PDR: 46.875%

Method AASR (%) MA (%)

----------------------------------------

RFBDS 21.55 25.65

PrivRFBDS 19.75 12.86

FedAvg 17.67 27.87

Median 86.00 23.67

Trimmed-Mean 92.39 18.08

FedCC 23.38 27.43

RDFL 57.65 27.75

FLAME 62.63 27.44

[22bit005@mepcolinux ~]$ls

0.25.txt client.py diamond.c finish grokall.txt mca.txt os sc.py sixthrun.txt Vsem

0.75.txt core.228416 environ.txt five.txt grokk.txt merror.txt output.txt se1.txt socket.txt

15.25.txt datastructure env.txt got.txt Isem ml.txt pclients.txt sem2 sparkdownload.txt

15.75.txt dbms error.txt graph0.25pdr2.txt itt new.txt pdr.txt server.c spdiamond.c

angular.txt deep.txt e.txt graph0.75pdr.txt java no.txt robust.txt serverclient.txt star

c1.c description.txt examplelab.txt graph5.txt jdbc.java novalty.txt rpp.txt server.py upto0\_5.txt

client.c diamond fifthrunserver.txt graph.txt job.txt n.txt s1.c set1.txt VIsem

[22bit005@mepcolinux ~]$cat 15.25.txt

server:

import os

os.environ['TF\_CPP\_MIN\_LOG\_LEVEL'] = '2'

import socket

import numpy as np

from sklearn.cluster import OPTICS, KMeans, HDBSCAN

from sklearn.decomposition import PCA

import tensorflow as tf

from scipy.stats import dirichlet

import random

import time

# Load datasets

(x\_train\_mnist, y\_train\_mnist), (x\_test\_mnist, y\_test\_mnist) = tf.keras.datasets.mnist.load\_data()

x\_train\_mnist = x\_train\_mnist.reshape(-1, 28\*28).astype(np.float32) / 255.0

x\_test\_mnist = x\_test\_mnist.reshape(-1, 28\*28).astype(np.float32) / 255.0

(x\_train\_fmnist, y\_train\_fmnist), (x\_test\_fmnist, y\_test\_fmnist) = tf.keras.datasets.fashion\_mnist.load\_data()

x\_train\_fmnist = x\_train\_fmnist.reshape(-1, 28\*28).astype(np.float32) / 255.0

x\_test\_fmnist = x\_test\_fmnist.reshape(-1, 28\*28).astype(np.float32) / 255.0

(x\_train\_cifar, y\_train\_cifar), (x\_test\_cifar, y\_test\_cifar) = tf.keras.datasets.cifar10.load\_data()

x\_train\_cifar = x\_train\_cifar.reshape(-1, 32\*32\*3).astype(np.float32) / 255.0

x\_test\_cifar = x\_test\_cifar.reshape(-1, 32\*32\*3).astype(np.float32) / 255.0

y\_train\_cifar = y\_train\_cifar.flatten()

y\_test\_cifar = y\_test\_cifar.flatten()

datasets = {

'MNIST': (x\_train\_mnist, y\_train\_mnist, x\_test\_mnist, y\_test\_mnist, 784),

'FMNIST': (x\_train\_fmnist, y\_train\_fmnist, x\_test\_fmnist, y\_test\_fmnist, 784),

'CIFAR10': (x\_train\_cifar, y\_train\_cifar, x\_test\_cifar, y\_test\_cifar, 3072)

}

def create\_model(input\_shape):

model = tf.keras.Sequential([

tf.keras.layers.Input(shape=(input\_shape,)),

tf.keras.layers.Dense(128, activation='relu'),

tf.keras.layers.Dense(10, activation='softmax')

])

model.compile(optimizer='sgd', loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

return model

def secret\_share(data, n\_parties=2):

shares = [np.random.uniform(-1000, 1000, size=data.shape).astype(np.float32) for \_ in range(n\_parties - 1)]

shares.append(data - sum(shares))

return shares

def reconstruct\_shares(shares):

return sum(shares)

def amsparse(delta\_w, layer\_shapes):

amsparse\_updates = []

start = 0

for shape in layer\_shapes:

size = np.prod(shape)

layer\_delta = delta\_w[start:start + size].reshape(shape)

layer\_max = np.max(np.abs(layer\_delta), axis=tuple(range(len(shape) - 1)))

signs = np.sign(layer\_delta)

amsparse\_layer = signs \* layer\_max

amsparse\_updates.append(amsparse\_layer.flatten())

start += size

return np.concatenate(amsparse\_updates)

def auto\_optics(updates):

if len(updates) < 2:

return np.arange(len(updates))

try:

pca = PCA(n\_components=min(50, updates.shape[1]))

updates\_reduced = pca.fit\_transform(updates)

dists = np.linalg.norm(updates\_reduced - updates\_reduced.mean(axis=0), axis=1)

k = min(max(2, len(updates) // 3), len(updates) - 1)

sorted\_dists = np.sort(dists)[:k]

eps = np.median(sorted\_dists) \* 2.0

clustering = OPTICS(min\_samples=k, eps=eps, metric='euclidean').fit(updates\_reduced)

labels = clustering.labels\_

majority\_group = np.where(labels != -1)[0]

return majority\_group if len(majority\_group) > 0 else np.arange(len(updates))

except:

return np.arange(len(updates))

def adaptive\_clipping(updates, benign\_indices):

if len(updates) == 0:

return np.zeros\_like(updates)

norms = np.linalg.norm(updates, axis=1)

rho = np.median(norms[benign\_indices]) if len(benign\_indices) > 0 else 1.0

clipped\_updates = []

for i in benign\_indices:

update = updates[i]

norm = norms[i]

clipped = update \* min(1, rho / norm) if norm > 0 else update

clipped\_updates.append(clipped)

return np.array(clipped\_updates) if clipped\_updates else updates

def priv\_shuffling(updates\_shares, n\_clients):

pi\_0 = np.random.permutation(n\_clients)

pi\_1 = np.random.permutation(n\_clients)

shuffled\_shares = [[updates\_shares[pi\_0[i]][0], updates\_shares[pi\_1[i]][1]] for i in range(n\_clients)]

return shuffled\_shares

def priv\_selection(shuffled\_shares, m):

norms\_shares = [secret\_share(np.linalg.norm(reconstruct\_shares(shares))) for shares in shuffled\_shares]

norms = [reconstruct\_shares(ns) for ns in norms\_shares]

sorted\_indices = np.argsort(norms)[:m]

return [shuffled\_shares[i] for i in sorted\_indices]

def priv\_optics(updates\_shares):

updates = np.array([reconstruct\_shares(shares) for shares in updates\_shares])

if len(updates) < 2:

return secret\_share(np.arange(len(updates)))

try:

pca = PCA(n\_components=min(50, updates.shape[1]))

updates\_reduced = pca.fit\_transform(updates)

dists = np.linalg.norm(updates\_reduced - updates\_reduced.mean(axis=0), axis=1)

k = min(max(2, len(updates) // 3), len(updates) - 1)

sorted\_dists = np.sort(dists)[:k]

eps = np.median(sorted\_dists) \* 2.0

clustering = OPTICS(min\_samples=k, eps=eps, metric='euclidean').fit(updates\_reduced)

labels = clustering.labels\_

benign\_indices = np.where(labels != -1)[0]

return secret\_share(benign\_indices if len(benign\_indices) > 0 else np.arange(len(updates)))

except:

return secret\_share(np.arange(len(updates)))

def priv\_clipping(updates\_shares, benign\_indices\_shares):

updates = np.array([reconstruct\_shares(shares) for shares in updates\_shares])

benign\_indices = reconstruct\_shares(benign\_indices\_shares).astype(int)

norms = np.linalg.norm(updates, axis=1)

rho\_shares = secret\_share(np.median(norms[benign\_indices]) if len(benign\_indices) > 0 else 1.0)

clipped\_shares = []

for i in benign\_indices:

update\_shares = updates\_shares[i]

norm\_shares = secret\_share(norms[i])

rho\_val = reconstruct\_shares(rho\_shares)

norm\_val = reconstruct\_shares(norm\_shares)

factor = min(1, rho\_val / norm\_val) if norm\_val > 0 else 1.0

clipped\_shares.append([share \* factor for share in update\_shares])

return clipped\_shares if clipped\_shares else updates\_shares

def priv\_aggregation(clipped\_shares):

if not clipped\_shares:

return secret\_share(np.zeros\_like(reconstruct\_shares(clipped\_shares[0])))

clipped\_updates = np.array([reconstruct\_shares(shares) for shares in clipped\_shares])

return secret\_share(np.mean(clipped\_updates, axis=0))

def fedavg(updates):

return np.mean(updates, axis=0) if len(updates) > 0 else np.zeros\_like(updates[0])

def median(updates):

return np.median(updates, axis=0) if len(updates) > 0 else np.zeros\_like(updates[0])

def trimmed\_mean(updates, trim\_ratio=0.1):

if len(updates) < 3:

return np.mean(updates, axis=0) if len(updates) > 0 else np.zeros\_like(updates[0])

n\_trim = min(int(len(updates) \* trim\_ratio), len(updates) // 2 - 1)

if n\_trim == 0:

return np.mean(updates, axis=0)

sorted\_updates = np.sort(updates, axis=0)

trimmed = sorted\_updates[n\_trim:-n\_trim]

return np.mean(trimmed, axis=0)

def fedcc(updates):

if len(updates) < 2:

return np.mean(updates, axis=0) if len(updates) > 0 else np.zeros\_like(updates[0])

try:

pca = PCA(n\_components=min(50, updates.shape[1]))

updates\_reduced = pca.fit\_transform(updates)

clustering = KMeans(n\_clusters=2, n\_init=10, random\_state=42).fit(updates\_reduced)

labels = clustering.labels\_

benign\_cluster = np.argmax(np.bincount(labels))

benign\_indices = np.where(labels == benign\_cluster)[0]

if len(benign\_indices) == 0:

return np.mean(updates, axis=0)

return np.mean(updates[benign\_indices], axis=0)

except:

return np.mean(updates, axis=0)

def rdfl(updates):

if len(updates) < 2:

return np.mean(updates, axis=0) if len(updates) > 0 else np.zeros\_like(updates[0])

try:

pca = PCA(n\_components=min(50, updates.shape[1]))

updates\_reduced = pca.fit\_transform(updates)

clustering = HDBSCAN(min\_cluster\_size=2, min\_samples=2).fit(updates\_reduced)

labels = clustering.labels\_

benign\_indices = np.where(labels != -1)[0]

return np.mean(updates[benign\_indices], axis=0) if len(benign\_indices) > 0 else np.mean(updates, axis=0)

except:

return np.mean(updates, axis=0)

def flame(updates):

if len(updates) < 2:

return np.mean(updates, axis=0) if len(updates) > 0 else np.zeros\_like(updates[0])

try:

norms = np.linalg.norm(updates, axis=1)

rho = np.median(norms)

clipped = [u \* min(1, rho / np.linalg.norm(u)) if np.linalg.norm(u) > 0 else u for u in updates]

pca = PCA(n\_components=min(50, updates.shape[1]))

clipped\_reduced = pca.fit\_transform(clipped)

clustering = HDBSCAN(min\_cluster\_size=2, min\_samples=2).fit(clipped\_reduced)

labels = clustering.labels\_

benign\_indices = np.where(labels != -1)[0]

return np.mean(np.array(clipped)[benign\_indices], axis=0) if len(benign\_indices) > 0 else np.mean(clipped, axis=0)

except:

return np.mean(updates, axis=0)

def generate\_trigger\_pattern(shape, trigger\_type, trigger\_size):

if trigger\_type == 'pixel':

trigger = np.zeros(shape)

trigger[:trigger\_size:2] = 1.0

elif trigger\_type == 'square':

trigger = np.zeros(shape)

trigger[:trigger\_size] = 1.0

elif trigger\_type == 'cross':

trigger = np.zeros(shape)

trigger[trigger\_size//2] = 1.0

else:

trigger = np.random.choice([0, 1], size=shape, p=[0.7, 0.3]).astype(np.float32)

return trigger

def apply\_trigger(x, trigger\_pattern, trigger\_size):

triggered\_x = x.copy()

triggered\_x[:trigger\_size] = np.maximum(triggered\_x[:trigger\_size], trigger\_pattern[:trigger\_size])

return triggered\_x

def send\_data(client\_socket, data):

try:

data\_bytes = data.tobytes()

size = np.array([len(data\_bytes)], dtype=np.int64).tobytes()

client\_socket.sendall(size + data\_bytes)

except socket.error:

raise

def receive\_data(client\_socket):

try:

client\_socket.settimeout(60.0)

size\_data = client\_socket.recv(8)

if not size\_data:

raise socket.error("Client disconnected")

data\_size = np.frombuffer(size\_data, dtype=np.int64)[0]

received\_data = bytearray()

while len(received\_data) < data\_size:

packet = client\_socket.recv(min(4096, data\_size - len(received\_data)))

if not packet:

raise socket.error("Client disconnected during data transfer")

received\_data.extend(packet)

return np.frombuffer(received\_data, dtype=np.float32)

except socket.error:

raise

# Server setup

server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

server\_socket.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)

server\_address = ('172.16.24.61', 9999)

server\_socket.bind(server\_address)

server\_socket.listen(5)

print("Server listening at", server\_address)

T = 10

n\_clients = 5

m\_selected = 5

AR = 0.4

PDR = 0.15625

NIR = 0.25

methods = {

'RFBDS': lambda updates: np.mean(adaptive\_clipping(updates, auto\_optics(np.array([amsparse(u, layer\_shapes) for u in updates]))), axis=0),

'PrivRFBDS': lambda updates\_shares: reconstruct\_shares(priv\_aggregation(

priv\_clipping(

priv\_selection(

priv\_shuffling(updates\_shares, len(updates\_shares)), m\_selected),

priv\_optics(

priv\_selection(

priv\_shuffling(updates\_shares, len(updates\_shares)), m\_selected))))),

'FedAvg': fedavg,

'Median': median,

'Trimmed-Mean': trimmed\_mean,

'FedCC': fedcc,

'RDFL': rdfl,

'FLAME': flame

}

for dataset\_name in ['MNIST', 'FMNIST', 'CIFAR10']:

x\_train, y\_train, x\_test, y\_test, input\_shape = datasets[dataset\_name]

global\_model = create\_model(input\_shape)

global\_weights = global\_model.get\_weights()

layer\_shapes = [w.shape for w in global\_weights]

global\_weights\_flat = np.concatenate([w.flatten() for w in global\_weights]).astype(np.float32)

samples\_per\_client = 6000 if dataset\_name != 'CIFAR10' else 5000

print(f"\nDataset: {dataset\_name}, NIR: {NIR}")

client\_data = []

indices = np.arange(len(x\_train))

label\_dist = dirichlet.rvs([1.0 - NIR] \* 10, size=n\_clients)

for i in range(n\_clients):

client\_indices = []

probs = label\_dist[i]

for label in range(10):

label\_indices = np.where(y\_train == label)[0]

n\_samples = int(samples\_per\_client \* probs[label])

if len(label\_indices) > 0:

selected = np.random.choice(label\_indices, min(n\_samples, len(label\_indices)), replace=False)

client\_indices.extend(selected)

client\_indices = np.array(client\_indices)

if len(client\_indices) < samples\_per\_client:

extra\_indices = np.random.choice(indices, samples\_per\_client - len(client\_indices), replace=False)

client\_indices = np.concatenate([client\_indices, extra\_indices])

elif len(client\_indices) > samples\_per\_client:

client\_indices = client\_indices[:samples\_per\_client]

np.random.shuffle(client\_indices)

client\_data.append((x\_train[client\_indices], y\_train[client\_indices], i >= int(n\_clients \* (1 - AR))))

results = {method\_name: {'MA': None, 'AASR': None} for method\_name in methods}

for method\_name, method in methods.items():

print(f"\nMethod: {method\_name}")

client\_sockets = []

try:

for i in range(n\_clients):

client\_socket, addr = server\_socket.accept()

client\_socket.settimeout(60.0)

print(f"Client {i} connected from {addr}")

client\_sockets.append(client\_socket)

x\_client, y\_client, is\_malicious = client\_data[i]

client\_data\_flat = np.concatenate([

x\_client.flatten(),

y\_client.astype(np.float32),

np.array([float(is\_malicious)], dtype=np.float32)

]).astype(np.float32)

send\_data(client\_socket, client\_data\_flat)

send\_data(client\_socket, global\_weights\_flat)

current\_weights\_flat = global\_weights\_flat.copy()

current\_model = create\_model(input\_shape)

current\_weights = [

current\_weights\_flat[i:i + w.size].reshape(w.shape)

for i, w in zip(np.cumsum([0] + [w.size for w in global\_weights[:-1]]), global\_weights)

]

current\_model.set\_weights(current\_weights)

for t in range(T):

client\_updates = []

client\_updates\_shares = []

active\_sockets = []

for i, client\_socket in enumerate(client\_sockets):

if client\_socket is None:

continue

try:

updates = receive\_data(client\_socket)

split\_idx = len(updates) // 2

delta\_w = updates[:split\_idx]

delta\_w\_am = updates[split\_idx:]

client\_updates.append(delta\_w)

client\_updates\_shares.append(secret\_share(delta\_w\_am))

active\_sockets.append(client\_socket)

except socket.error:

client\_sockets[i] = None

client\_sockets = active\_sockets

if not client\_updates:

print(f"Round {t+1}/{T} - No active clients")

continue

try:

if method\_name == 'PrivRFBDS':

aggregated\_update = method(client\_updates\_shares)

else:

aggregated\_update = method(np.array(client\_updates))

current\_weights\_flat += aggregated\_update

current\_weights = [

current\_weights\_flat[i:i + w.size].reshape(w.shape)

for i, w in zip(np.cumsum([0] + [w.size for w in global\_weights[:-1]]), global\_weights)

]

current\_model.set\_weights(current\_weights)

loss, ma = current\_model.evaluate(x\_test, y\_test, verbose=0)

trigger\_type = random.choice(['pixel', 'square', 'cross', 'random'])

trigger\_size = 5\*28 if dataset\_name != 'CIFAR10' else 8\*32

trigger\_pattern = generate\_trigger\_pattern(input\_shape, trigger\_type, trigger\_size)

x\_test\_backdoor = np.array([apply\_trigger(x, trigger\_pattern, trigger\_size) for x in x\_test])

y\_test\_backdoor = np.ones\_like(y\_test) \* 7

\_, aasr = current\_model.evaluate(x\_test\_backdoor, y\_test\_backdoor, verbose=0)

if t == T - 1:

results[method\_name]['MA'] = ma \* 100

results[method\_name]['AASR'] = aasr \* 100

print(f"Round {t+1}/{T} - MA: {ma\*100:.2f}%, AASR: {aasr\*100:.2f}%")

except Exception as e:

print(f"Round {t+1}/{T} - Aggregation error: {e}")

continue

for client\_socket in client\_sockets:

if client\_socket:

try:

send\_data(client\_socket, current\_weights\_flat)

except socket.error:

client\_sockets[client\_sockets.index(client\_socket)] = None

except Exception as e:

print(f"Error during method {method\_name}: {e}")

finally:

for client\_socket in client\_sockets:

if client\_socket:

try:

client\_socket.close()

except:

pass

print(f"\nResults for Dataset: {dataset\_name}, NIR: {NIR}, AR: {AR\*100}%, PDR: {PDR\*100}%")

print("Method\t\tAASR (%)\tMA (%)")

print("-" \* 40)

for method\_name in methods:

ma = results[method\_name]['MA']

aasr = results[method\_name]['AASR']

if ma is not None and aasr is not None:

print(f"{method\_name:<15}\t{aasr:.2f}\t\t{ma:.2f}")

server\_socket.close()

client :

import socket

import numpy as np

import tensorflow as tf

import threading

import time

import random

def create\_model(input\_shape):

model = tf.keras.Sequential([

tf.keras.layers.Input(shape=(input\_shape,)),

tf.keras.layers.Dense(128, activation='relu'),

tf.keras.layers.Dense(10, activation='softmax')

])

model.compile(optimizer='sgd', loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

return model

def amsparse(delta\_w, layer\_shapes):

amsparse\_updates = []

start = 0

for shape in layer\_shapes:

size = np.prod(shape)

layer\_delta = delta\_w[start:start + size].reshape(shape)

layer\_max = np.max(np.abs(layer\_delta), axis=tuple(range(len(shape) - 1)))

signs = np.sign(layer\_delta)

amsparse\_layer = signs \* layer\_max

amsparse\_updates.append(amsparse\_layer.flatten())

start += size

return np.concatenate(amsparse\_updates)

def generate\_trigger\_pattern(shape, trigger\_type, trigger\_size):

if trigger\_type == 'pixel':

trigger = np.zeros(shape)

trigger[:trigger\_size:2] = 1.0

elif trigger\_type == 'square':

trigger = np.zeros(shape)

trigger[:trigger\_size] = 1.0

elif trigger\_type == 'cross':

trigger = np.zeros(shape)

trigger[trigger\_size//2] = 1.0

else:

trigger = np.random.choice([0, 1], size=shape, p=[0.7, 0.3]).astype(np.float32)

return trigger

def apply\_trigger(x, trigger\_pattern, trigger\_size):

triggered\_x = x.copy()

triggered\_x[:trigger\_size] = np.maximum(triggered\_x[:trigger\_size], trigger\_pattern[:trigger\_size])

return triggered\_x

def send\_data(client\_socket, data):

try:

data\_bytes = data.tobytes()

size = np.array([len(data\_bytes)], dtype=np.int64).tobytes()

client\_socket.sendall(size + data\_bytes)

except socket.error:

raise

def receive\_data(client\_socket):

try:

client\_socket.settimeout(60.0)

size\_data = client\_socket.recv(8)

if not size\_data:

raise socket.error("Server disconnected")

data\_size = np.frombuffer(size\_data, dtype=np.int64)[0]

received\_data = bytearray()

while len(received\_data) < data\_size:

packet = client\_socket.recv(min(4096, data\_size - len(received\_data)))

if not packet:

raise socket.error("Server disconnected during data transfer")

received\_data.extend(packet)

return np.frombuffer(received\_data, dtype=np.float32)

except socket.error:

raise

def client\_thread(client\_id, server\_address, T, batch\_size, input\_shape, dataset\_name):

samples\_per\_client = 6000 if dataset\_name != 'CIFAR10' else 5000

for method\_idx in range(8): # One for each method

client\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

retry\_count = 0

max\_retries = 5

while retry\_count < max\_retries:

try:

client\_socket.connect(server\_address)

print(f"Client {client\_id} connected to {server\_address} for method {method\_idx}")

break

except ConnectionRefusedError:

retry\_count += 1

time.sleep(1)

if retry\_count == max\_retries:

print(f"Client {client\_id} failed to connect after {max\_retries} retries")

return

try:

client\_data\_flat = receive\_data(client\_socket)

data\_size = samples\_per\_client \* input\_shape

x\_client = client\_data\_flat[:data\_size].reshape(samples\_per\_client, input\_shape)

y\_client = client\_data\_flat[data\_size:data\_size + samples\_per\_client].astype(np.int32)

is\_malicious = bool(client\_data\_flat[-1])

global\_weights\_flat = receive\_data(client\_socket)

model = create\_model(input\_shape)

global\_weights = [

global\_weights\_flat[i:i + w.size].reshape(w.shape)

for i, w in zip(np.cumsum([0] + [w.size for w in model.get\_weights()[:-1]]), model.get\_weights())

]

model.set\_weights(global\_weights)

layer\_shapes = [w.shape for w in model.get\_weights()]

trigger\_type = random.choice(['pixel', 'square', 'cross', 'random'])

trigger\_size = 5\*28 if dataset\_name != 'CIFAR10' else 8\*32

trigger\_pattern = generate\_trigger\_pattern(input\_shape, trigger\_type, trigger\_size)

for t in range(T):

try:

if is\_malicious:

n\_poison = int(len(x\_client) \* 0.15625)

poisoned\_data = x\_client.copy()

if t % 5 == 0:

trigger\_type = random.choice(['pixel', 'square', 'cross', 'random'])

trigger\_pattern = generate\_trigger\_pattern(input\_shape, trigger\_type, trigger\_size)

poisoned\_data[:n\_poison] = np.array([

apply\_trigger(x, trigger\_pattern, trigger\_size)

for x in poisoned\_data[:n\_poison]

])

poisoned\_labels = y\_client.copy()

poisoned\_labels[:n\_poison] = 7

model.fit(poisoned\_data, poisoned\_labels, epochs=1, batch\_size=batch\_size, verbose=0)

else:

model.fit(x\_client, y\_client, epochs=1, batch\_size=batch\_size, verbose=0)

new\_weights\_flat = np.concatenate([w.flatten() for w in model.get\_weights()]).astype(np.float32)

delta\_w = new\_weights\_flat - global\_weights\_flat

delta\_w\_am = amsparse(delta\_w, layer\_shapes)

updates = np.concatenate([delta\_w, delta\_w\_am]).astype(np.float32)

send\_data(client\_socket, updates)

global\_weights\_flat = receive\_data(client\_socket)

global\_weights = [

global\_weights\_flat[i:i + w.size].reshape(w.shape)

for i, w in zip(np.cumsum([0] + [w.size for w in model.get\_weights()[:-1]]), model.get\_weights())

]

model.set\_weights(global\_weights)

except socket.error:

print(f"Client {client\_id} disconnected during round {t+1}")

break

except Exception as e:

print(f"Client {client\_id} error: {e}")

finally:

try:

client\_socket.close()

except:

pass

# Configuration

server\_address = ('localhost', 9999)

T = 10

batch\_size = 64

n\_clients = 5

dataset\_configs = {

'MNIST': 784,

'FMNIST': 784,

'CIFAR10': 3072

}

for dataset\_name, input\_shape in dataset\_configs.items():

threads = []

for i in range(n\_clients):

thread = threading.Thread(target=client\_thread, args=(i, server\_address, T, batch\_size, input\_shape, dataset\_name))

threads.append(thread)

thread.start()

for thread in threads:

thread.join()

output:

C:\Users\login01\Documents>conda activate tf\_env

(tf\_env) C:\Users\login01\Documents>python server.py

Server listening at ('172.16.24.61', 9999)

Dataset: MNIST, NIR: 0.25

Method: RFBDS

Client 0 connected from ('172.16.24.60', 60293)

Client 1 connected from ('172.16.24.60', 60294)

Client 2 connected from ('172.16.24.60', 60296)

Client 3 connected from ('172.16.24.60', 60295)

Client 4 connected from ('172.16.24.60', 60297)

Round 10/10 - MA: 84.27%, AASR: 99.94%

Method: PrivRFBDS

Client 0 connected from ('172.16.24.60', 60299)

Client 1 connected from ('172.16.24.60', 60300)

Client 2 connected from ('172.16.24.60', 60301)

Client 3 connected from ('172.16.24.60', 60302)

Client 4 connected from ('172.16.24.60', 60303)

Round 10/10 - MA: 48.18%, AASR: 1.34%

Method: FedAvg

Client 0 connected from ('172.16.24.60', 60318)

Client 1 connected from ('172.16.24.60', 60319)

Client 2 connected from ('172.16.24.60', 60320)

Client 3 connected from ('172.16.24.60', 60321)

Client 4 connected from ('172.16.24.60', 60322)

Round 10/10 - MA: 84.19%, AASR: 25.16%

Method: Median

Client 0 connected from ('172.16.24.60', 60325)

Client 1 connected from ('172.16.24.60', 60326)

Client 2 connected from ('172.16.24.60', 60327)

Client 3 connected from ('172.16.24.60', 60328)

Client 4 connected from ('172.16.24.60', 60329)

Round 10/10 - MA: 81.78%, AASR: 10.40%

Method: Trimmed-Mean

Client 0 connected from ('172.16.24.60', 60333)

Client 1 connected from ('172.16.24.60', 60334)

Client 2 connected from ('172.16.24.60', 60335)

Client 3 connected from ('172.16.24.60', 60336)

Client 4 connected from ('172.16.24.60', 60337)

Round 10/10 - MA: 83.81%, AASR: 99.31%

Method: FedCC

Client 0 connected from ('172.16.24.60', 60344)

Client 1 connected from ('172.16.24.60', 60345)

Client 2 connected from ('172.16.24.60', 60346)

Client 3 connected from ('172.16.24.60', 60347)

Client 4 connected from ('172.16.24.60', 60348)

Round 10/10 - MA: 83.87%, AASR: 32.92%

Method: RDFL

Client 0 connected from ('172.16.24.60', 60350)

Client 1 connected from ('172.16.24.60', 60351)

Client 2 connected from ('172.16.24.60', 60352)

Client 3 connected from ('172.16.24.60', 60353)

Client 4 connected from ('172.16.24.60', 60354)

Round 10/10 - MA: 84.18%, AASR: 98.23%

Method: FLAME

Client 0 connected from ('172.16.24.60', 60356)

Client 1 connected from ('172.16.24.60', 60357)

Client 2 connected from ('172.16.24.60', 60358)

Client 3 connected from ('172.16.24.60', 60359)

Client 4 connected from ('172.16.24.60', 60360)

Round 10/10 - MA: 83.89%, AASR: 76.98%

Results for Dataset: MNIST, NIR: 0.25, AR: 40.0%, PDR: 15.625%

Method AASR (%) MA (%)

----------------------------------------

RFBDS 99.94 84.27

PrivRFBDS 1.34 48.18

FedAvg 25.16 84.19

Median 10.40 81.78

Trimmed-Mean 99.31 83.81

FedCC 32.92 83.87

RDFL 98.23 84.18

FLAME 76.98 83.89

Dataset: FMNIST, NIR: 0.25

Method: RFBDS

Client 0 connected from ('172.16.24.60', 60370)

Client 1 connected from ('172.16.24.60', 60371)

Client 2 connected from ('172.16.24.60', 60372)

Client 3 connected from ('172.16.24.60', 60373)

Client 4 connected from ('172.16.24.60', 60374)

Round 10/10 - MA: 69.91%, AASR: 12.00%

Method: PrivRFBDS

Client 0 connected from ('172.16.24.60', 60375)

Client 1 connected from ('172.16.24.60', 60376)

Client 2 connected from ('172.16.24.60', 60377)

Client 3 connected from ('172.16.24.60', 60378)

Client 4 connected from ('172.16.24.60', 60379)

Round 10/10 - MA: 36.27%, AASR: 0.24%

Method: FedAvg

Client 0 connected from ('172.16.24.60', 60381)

Client 1 connected from ('172.16.24.60', 60382)

Client 2 connected from ('172.16.24.60', 60383)

Client 3 connected from ('172.16.24.60', 60384)

Client 4 connected from ('172.16.24.60', 60385)

Round 10/10 - MA: 70.50%, AASR: 24.98%

Method: Median

Client 0 connected from ('172.16.24.60', 60388)

Client 1 connected from ('172.16.24.60', 60389)

Client 2 connected from ('172.16.24.60', 60390)

Client 3 connected from ('172.16.24.60', 60391)

Client 4 connected from ('172.16.24.60', 60392)

Round 10/10 - MA: 69.65%, AASR: 3.83%

Method: Trimmed-Mean

Client 0 connected from ('172.16.24.60', 60394)

Client 1 connected from ('172.16.24.60', 60395)

Client 2 connected from ('172.16.24.60', 60396)

Client 3 connected from ('172.16.24.60', 60397)

Client 4 connected from ('172.16.24.60', 60398)

Round 10/10 - MA: 70.62%, AASR: 5.29%

Method: FedCC

Client 0 connected from ('172.16.24.60', 60403)

Client 1 connected from ('172.16.24.60', 60404)

Client 2 connected from ('172.16.24.60', 60405)

Client 3 connected from ('172.16.24.60', 60406)

Client 4 connected from ('172.16.24.60', 60407)

Round 10/10 - MA: 70.78%, AASR: 70.05%

Method: RDFL

Client 0 connected from ('172.16.24.60', 60417)

Client 1 connected from ('172.16.24.60', 60418)

Client 2 connected from ('172.16.24.60', 60419)

Client 3 connected from ('172.16.24.60', 60420)

Client 4 connected from ('172.16.24.60', 60421)

Round 10/10 - MA: 71.28%, AASR: 25.71%

Method: FLAME

Client 0 connected from ('172.16.24.60', 60428)

Client 1 connected from ('172.16.24.60', 60429)

Client 2 connected from ('172.16.24.60', 60430)

Client 3 connected from ('172.16.24.60', 60431)

Client 4 connected from ('172.16.24.60', 60432)

Round 10/10 - MA: 71.31%, AASR: 4.25%

Results for Dataset: FMNIST, NIR: 0.25, AR: 40.0%, PDR: 15.625%

Method AASR (%) MA (%)

----------------------------------------

RFBDS 12.00 69.91

PrivRFBDS 0.24 36.27

FedAvg 24.98 70.50

Median 3.83 69.65

Trimmed-Mean 5.29 70.62

FedCC 70.05 70.78

RDFL 25.71 71.28

FLAME 4.25 71.31

Dataset: CIFAR10, NIR: 0.25

Method: RFBDS

Client 0 connected from ('172.16.24.60', 60443)

Client 1 connected from ('172.16.24.60', 60444)

Client 2 connected from ('172.16.24.60', 60445)

Client 3 connected from ('172.16.24.60', 60446)

Client 4 connected from ('172.16.24.60', 60447)

Round 10/10 - MA: 27.95%, AASR: 67.03%

Method: PrivRFBDS

Client 0 connected from ('172.16.24.60', 60451)

Client 1 connected from ('172.16.24.60', 60452)

Client 2 connected from ('172.16.24.60', 60453)

Client 3 connected from ('172.16.24.60', 60454)

Client 4 connected from ('172.16.24.60', 60455)

Round 10/10 - MA: 15.29%, AASR: 7.96%

Method: FedAvg

Client 0 connected from ('172.16.24.60', 60461)

Client 1 connected from ('172.16.24.60', 60462)

Client 2 connected from ('172.16.24.60', 60463)

Client 3 connected from ('172.16.24.60', 60464)

Client 4 connected from ('172.16.24.60', 60465)

Round 10/10 - MA: 26.75%, AASR: 54.51%

Method: Median

Client 0 connected from ('172.16.24.60', 60641)

Error during method Median: [WinError 10053] An established connection was aborted by the software in your host machine

Method: Trimmed-Mean

Client 0 connected from ('172.16.24.60', 60642)

Error during method Trimmed-Mean: [WinError 10053] An established connection was aborted by the software in your host machine

Method: FedCC

Client 0 connected from ('172.16.24.60', 60643)

Error during method FedCC: [WinError 10053] An established connection was aborted by the software in your host machine

Method: RDFL

Client 0 connected from ('172.16.24.60', 60644)

Error during method RDFL: [WinError 10053] An established connection was aborted by the software in your host machine

Method: FLAME

Client 0 connected from ('172.16.24.60', 60645)

Error during method FLAME: [WinError 10053] An established connection was aborted by the software in your host machine

Results for Dataset: CIFAR10, NIR: 0.25, AR: 40.0%, PDR: 15.625%

Method AASR (%) MA (%)

----------------------------------------

RFBDS 67.03 27.95

PrivRFBDS 7.96 15.29

FedAvg 54.51 26.75

[22bit005@mepcolinux ~]$cat 15.75.txt

import os

os.environ['TF\_CPP\_MIN\_LOG\_LEVEL'] = '2'

import socket

import numpy as np

from sklearn.cluster import OPTICS, KMeans, HDBSCAN

from sklearn.decomposition import PCA

import tensorflow as tf

from scipy.stats import dirichlet

import random

import time

# Load datasets

(x\_train\_mnist, y\_train\_mnist), (x\_test\_mnist, y\_test\_mnist) = tf.keras.datasets.mnist.load\_data()

x\_train\_mnist = x\_train\_mnist.reshape(-1, 28\*28).astype(np.float32) / 255.0

x\_test\_mnist = x\_test\_mnist.reshape(-1, 28\*28).astype(np.float32) / 255.0

(x\_train\_fmnist, y\_train\_fmnist), (x\_test\_fmnist, y\_test\_fmnist) = tf.keras.datasets.fashion\_mnist.load\_data()

x\_train\_fmnist = x\_train\_fmnist.reshape(-1, 28\*28).astype(np.float32) / 255.0

x\_test\_fmnist = x\_test\_fmnist.reshape(-1, 28\*28).astype(np.float32) / 255.0

(x\_train\_cifar, y\_train\_cifar), (x\_test\_cifar, y\_test\_cifar) = tf.keras.datasets.cifar10.load\_data()

x\_train\_cifar = x\_train\_cifar.reshape(-1, 32\*32\*3).astype(np.float32) / 255.0

x\_test\_cifar = x\_test\_cifar.reshape(-1, 32\*32\*3).astype(np.float32) / 255.0

y\_train\_cifar = y\_train\_cifar.flatten()

y\_test\_cifar = y\_test\_cifar.flatten()

datasets = {

'MNIST': (x\_train\_mnist, y\_train\_mnist, x\_test\_mnist, y\_test\_mnist, 784),

'FMNIST': (x\_train\_fmnist, y\_train\_fmnist, x\_test\_fmnist, y\_test\_fmnist, 784),

'CIFAR10': (x\_train\_cifar, y\_train\_cifar, x\_test\_cifar, y\_test\_cifar, 3072)

}

def create\_model(input\_shape):

model = tf.keras.Sequential([

tf.keras.layers.Input(shape=(input\_shape,)),

tf.keras.layers.Dense(128, activation='relu'),

tf.keras.layers.Dense(10, activation='softmax')

])

model.compile(optimizer='sgd', loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

return model

def secret\_share(data, n\_parties=2):

shares = [np.random.uniform(-1000, 1000, size=data.shape).astype(np.float32) for \_ in range(n\_parties - 1)]

shares.append(data - sum(shares))

return shares

def reconstruct\_shares(shares):

return sum(shares)

def amsparse(delta\_w, layer\_shapes):

amsparse\_updates = []

start = 0

for shape in layer\_shapes:

size = np.prod(shape)

layer\_delta = delta\_w[start:start + size].reshape(shape)

layer\_max = np.max(np.abs(layer\_delta), axis=tuple(range(len(shape) - 1)))

signs = np.sign(layer\_delta)

amsparse\_layer = signs \* layer\_max

amsparse\_updates.append(amsparse\_layer.flatten())

start += size

return np.concatenate(amsparse\_updates)

def auto\_optics(updates):

if len(updates) < 2:

return np.arange(len(updates))

try:

pca = PCA(n\_components=min(50, updates.shape[1]))

updates\_reduced = pca.fit\_transform(updates)

dists = np.linalg.norm(updates\_reduced - updates\_reduced.mean(axis=0), axis=1)

k = min(max(2, len(updates) // 3), len(updates) - 1)

sorted\_dists = np.sort(dists)[:k]

eps = np.median(sorted\_dists) \* 1.5 # Adjusted for tighter clustering

clustering = OPTICS(min\_samples=k, eps=eps, metric='euclidean').fit(updates\_reduced)

labels = clustering.labels\_

majority\_group = np.where(labels != -1)[0]

return majority\_group if len(majority\_group) > 0 else np.arange(len(updates))

except:

return np.arange(len(updates))

def adaptive\_clipping(updates, benign\_indices):

if len(updates) == 0:

return np.zeros\_like(updates)

norms = np.linalg.norm(updates, axis=1)

rho = np.median(norms[benign\_indices]) if len(benign\_indices) > 0 else 1.0

clipped\_updates = []

for i in benign\_indices:

update = updates[i]

norm = norms[i]

clipped = update \* min(1, rho / norm) if norm > 0 else update

clipped\_updates.append(clipped)

return np.array(clipped\_updates) if clipped\_updates else updates

def priv\_shuffling(updates\_shares, n\_clients):

pi\_0 = np.random.permutation(n\_clients)

pi\_1 = np.random.permutation(n\_clients)

shuffled\_shares = [[updates\_shares[pi\_0[i]][0], updates\_shares[pi\_1[i]][1]] for i in range(n\_clients)]

return shuffled\_shares

def priv\_selection(shuffled\_shares, m):

norms\_shares = [secret\_share(np.linalg.norm(reconstruct\_shares(shares))) for shares in shuffled\_shares]

norms = [reconstruct\_shares(ns) for ns in norms\_shares]

sorted\_indices = np.argsort(norms)[:m]

return [shuffled\_shares[i] for i in sorted\_indices]

def priv\_optics(updates\_shares):

updates = np.array([reconstruct\_shares(shares) for shares in updates\_shares])

if len(updates) < 2:

return secret\_share(np.arange(len(updates)))

try:

pca = PCA(n\_components=min(50, updates.shape[1]))

updates\_reduced = pca.fit\_transform(updates)

dists = np.linalg.norm(updates\_reduced - updates\_reduced.mean(axis=0), axis=1)

k = min(max(2, len(updates) // 3), len(updates) - 1)

sorted\_dists = np.sort(dists)[:k]

eps = np.median(sorted\_dists) \* 1.5

clustering = OPTICS(min\_samples=k, eps=eps, metric='euclidean').fit(updates\_reduced)

labels = clustering.labels\_

benign\_indices = np.where(labels != -1)[0]

return secret\_share(benign\_indices if len(benign\_indices) > 0 else np.arange(len(updates)))

except:

return secret\_share(np.arange(len(updates)))

def priv\_clipping(updates\_shares, benign\_indices\_shares):

updates = np.array([reconstruct\_shares(shares) for shares in updates\_shares])

benign\_indices = reconstruct\_shares(benign\_indices\_shares).astype(int)

norms = np.linalg.norm(updates, axis=1)

rho\_shares = secret\_share(np.median(norms[benign\_indices]) if len(benign\_indices) > 0 else 1.0)

clipped\_shares = []

for i in benign\_indices:

update\_shares = updates\_shares[i]

norm\_shares = secret\_share(norms[i])

rho\_val = reconstruct\_shares(rho\_shares)

norm\_val = reconstruct\_shares(norm\_shares)

factor = min(1, rho\_val / norm\_val) if norm\_val > 0 else 1.0

clipped\_shares.append([share \* factor for share in update\_shares])

return clipped\_shares if clipped\_shares else updates\_shares

def priv\_aggregation(clipped\_shares):

if not clipped\_shares:

return secret\_share(np.zeros\_like(reconstruct\_shares(clipped\_shares[0])))

clipped\_updates = np.array([reconstruct\_shares(shares) for shares in clipped\_shares])

return secret\_share(np.mean(clipped\_updates, axis=0))

def fedavg(updates):

return np.mean(updates, axis=0) if len(updates) > 0 else np.zeros\_like(updates[0])

def median(updates):

return np.median(updates, axis=0) if len(updates) > 0 else np.zeros\_like(updates[0])

def trimmed\_mean(updates, trim\_ratio=0.1):

if len(updates) < 3:

return np.mean(updates, axis=0) if len(updates) > 0 else np.zeros\_like(updates[0])

n\_trim = min(int(len(updates) \* trim\_ratio), len(updates) // 2 - 1)

if n\_trim == 0:

return np.mean(updates, axis=0)

sorted\_updates = np.sort(updates, axis=0)

trimmed = sorted\_updates[n\_trim:-n\_trim]

return np.mean(trimmed, axis=0)

def fedcc(updates):

if len(updates) < 2:

return np.mean(updates, axis=0) if len(updates) > 0 else np.zeros\_like(updates[0])

try:

pca = PCA(n\_components=min(50, updates.shape[1]))

updates\_reduced = pca.fit\_transform(updates)

clustering = KMeans(n\_clusters=2, n\_init=10, random\_state=42).fit(updates\_reduced)

labels = clustering.labels\_

benign\_cluster = np.argmax(np.bincount(labels))

benign\_indices = np.where(labels == benign\_cluster)[0]

if len(benign\_indices) == 0:

return np.mean(updates, axis=0)

return np.mean(updates[benign\_indices], axis=0)

except:

return np.mean(updates, axis=0)

def rdfl(updates):

if len(updates) < 2:

return np.mean(updates, axis=0) if len(updates) > 0 else np.zeros\_like(updates[0])

try:

pca = PCA(n\_components=min(50, updates.shape[1]))

updates\_reduced = pca.fit\_transform(updates)

clustering = HDBSCAN(min\_cluster\_size=2, min\_samples=2).fit(updates\_reduced)

labels = clustering.labels\_

benign\_indices = np.where(labels != -1)[0]

return np.mean(updates[benign\_indices], axis=0) if len(benign\_indices) > 0 else np.mean(updates, axis=0)

except:

return np.mean(updates, axis=0)

def flame(updates):

if len(updates) < 2:

return np.mean(updates, axis=0) if len(updates) > 0 else np.zeros\_like(updates[0])

try:

norms = np.linalg.norm(updates, axis=1)

rho = np.median(norms)

clipped = [u \* min(1, rho / np.linalg.norm(u)) if np.linalg.norm(u) > 0 else u for u in updates]

pca = PCA(n\_components=min(50, updates.shape[1]))

clipped\_reduced = pca.fit\_transform(clipped)

clustering = HDBSCAN(min\_cluster\_size=2, min\_samples=2).fit(clipped\_reduced)

labels = clustering.labels\_

benign\_indices = np.where(labels != -1)[0]

return np.mean(np.array(clipped)[benign\_indices], axis=0) if len(benign\_indices) > 0 else np.mean(clipped, axis=0)

except:

return np.mean(updates, axis=0)

def generate\_trigger\_pattern(shape, trigger\_type, trigger\_size):

if trigger\_type == 'pixel':

trigger = np.zeros(shape)

trigger[:trigger\_size:2] = 1.0

elif trigger\_type == 'square':

trigger = np.zeros(shape)

trigger[:trigger\_size] = 1.0

elif trigger\_type == 'cross':

trigger = np.zeros(shape)

trigger[trigger\_size//2] = 1.0

else:

trigger = np.random.choice([0, 1], size=shape, p=[0.7, 0.3]).astype(np.float32)

return trigger

def apply\_trigger(x, trigger\_pattern, trigger\_size):

triggered\_x = x.copy()

triggered\_x[:trigger\_size] = np.maximum(triggered\_x[:trigger\_size], trigger\_pattern[:trigger\_size])

return triggered\_x

def send\_data(client\_socket, data):

try:

data\_bytes = data.tobytes()

size = np.array([len(data\_bytes)], dtype=np.int64).tobytes()

client\_socket.sendall(size + data\_bytes)

except socket.error:

raise

def receive\_data(client\_socket):

try:

client\_socket.settimeout(120.0) # Increased timeout

size\_data = client\_socket.recv(8)

if not size\_data:

raise socket.error("Client disconnected")

data\_size = np.frombuffer(size\_data, dtype=np.int64)[0]

received\_data = bytearray()

while len(received\_data) < data\_size:

packet = client\_socket.recv(min(4096, data\_size - len(received\_data)))

if not packet:

raise socket.error("Client disconnected during data transfer")

received\_data.extend(packet)

return np.frombuffer(received\_data, dtype=np.float32)

except socket.error:

raise

# Server setup

server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

server\_socket.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)

server\_address = ('172.16.24.61', 9999)

server\_socket.bind(server\_address)

server\_socket.listen(5)

print("Server listening at", server\_address)

T = 10

n\_clients = 5

m\_selected = 5

AR = 0.4

PDR = 0.15625

NIR = 0.75

methods = {

'RFBDS': lambda updates: np.mean(adaptive\_clipping(updates, auto\_optics(np.array([amsparse(u, layer\_shapes) for u in updates]))), axis=0),

'PrivRFBDS': lambda updates\_shares: reconstruct\_shares(priv\_aggregation(

priv\_clipping(

priv\_selection(

priv\_shuffling(updates\_shares, len(updates\_shares)), m\_selected),

priv\_optics(

priv\_selection(

priv\_shuffling(updates\_shares, len(updates\_shares)), m\_selected))))),

'FedAvg': fedavg,

'Median': median,

'Trimmed-Mean': trimmed\_mean,

'FedCC': fedcc,

'RDFL': rdfl,

'FLAME': flame

}

for dataset\_name in ['MNIST', 'FMNIST', 'CIFAR10']:

x\_train, y\_train, x\_test, y\_test, input\_shape = datasets[dataset\_name]

global\_model = create\_model(input\_shape)

global\_weights = global\_model.get\_weights()

layer\_shapes = [w.shape for w in global\_weights]

global\_weights\_flat = np.concatenate([w.flatten() for w in global\_weights]).astype(np.float32)

samples\_per\_client = 6000 if dataset\_name != 'CIFAR10' else 5000

print(f"\nDataset: {dataset\_name}, NIR: {NIR}")

client\_data = []

indices = np.arange(len(x\_train))

label\_dist = dirichlet.rvs([1.0 - NIR] \* 10, size=n\_clients)

for i in range(n\_clients):

client\_indices = []

probs = label\_dist[i]

for label in range(10):

label\_indices = np.where(y\_train == label)[0]

n\_samples = int(samples\_per\_client \* probs[label])

if len(label\_indices) > 0:

selected = np.random.choice(label\_indices, min(n\_samples, len(label\_indices)), replace=False)

client\_indices.extend(selected)

client\_indices = np.array(client\_indices)

if len(client\_indices) < samples\_per\_client:

extra\_indices = np.random.choice(indices, samples\_per\_client - len(client\_indices), replace=False)

client\_indices = np.concatenate([client\_indices, extra\_indices])

elif len(client\_indices) > samples\_per\_client:

client\_indices = client\_indices[:samples\_per\_client]

np.random.shuffle(client\_indices)

client\_data.append((x\_train[client\_indices], y\_train[client\_indices], i >= int(n\_clients \* (1 - AR))))

results = {method\_name: {'MA': None, 'AASR': None} for method\_name in methods}

for method\_name, method in methods.items():

print(f"\nMethod: {method\_name}")

client\_sockets = []

try:

for i in range(n\_clients):

client\_socket, addr = server\_socket.accept()

client\_socket.settimeout(120.0)

print(f"Client {i} connected from {addr}")

client\_sockets.append(client\_socket)

x\_client, y\_client, is\_malicious = client\_data[i]

client\_data\_flat = np.concatenate([

x\_client.flatten(),

y\_client.astype(np.float32),

np.array([float(is\_malicious)], dtype=np.float32)

]).astype(np.float32)

send\_data(client\_socket, client\_data\_flat)

send\_data(client\_socket, global\_weights\_flat)

current\_weights\_flat = global\_weights\_flat.copy()

current\_model = create\_model(input\_shape)

current\_weights = [

current\_weights\_flat[i:i + w.size].reshape(w.shape)

for i, w in zip(np.cumsum([0] + [w.size for w in global\_weights[:-1]]), global\_weights)

]

current\_model.set\_weights(current\_weights)

for t in range(T):

client\_updates = []

client\_updates\_shares = []

active\_sockets = []

for i, client\_socket in enumerate(client\_sockets):

if client\_socket is None:

continue

try:

updates = receive\_data(client\_socket)

split\_idx = len(updates) // 2

delta\_w = updates[:split\_idx]

delta\_w\_am = updates[split\_idx:]

client\_updates.append(delta\_w)

client\_updates\_shares.append(secret\_share(delta\_w\_am))

active\_sockets.append(client\_socket)

except socket.error:

client\_sockets[i] = None

client\_sockets = active\_sockets

if not client\_updates:

print(f"Round {t+1}/{T} - No active clients")

continue

try:

if method\_name == 'PrivRFBDS':

aggregated\_update = method(client\_updates\_shares)

else:

aggregated\_update = method(np.array(client\_updates))

current\_weights\_flat += aggregated\_update

current\_weights = [

current\_weights\_flat[i:i + w.size].reshape(w.shape)

for i, w in zip(np.cumsum([0] + [w.size for w in global\_weights[:-1]]), global\_weights)

]

current\_model.set\_weights(current\_weights)

loss, ma = current\_model.evaluate(x\_test, y\_test, verbose=0)

trigger\_type = random.choice(['pixel', 'square', 'cross', 'random'])

trigger\_size = 5\*28 if dataset\_name != 'CIFAR10' else 8\*32

trigger\_pattern = generate\_trigger\_pattern(input\_shape, trigger\_type, trigger\_size)

x\_test\_backdoor = np.array([apply\_trigger(x, trigger\_pattern, trigger\_size) for x in x\_test])

y\_test\_backdoor = np.ones\_like(y\_test) \* 7

\_, aasr = current\_model.evaluate(x\_test\_backdoor, y\_test\_backdoor, verbose=0)

if t == T - 1:

results[method\_name]['MA'] = ma \* 100

results[method\_name]['AASR'] = aasr \* 100

print(f"Round {t+1}/{T} - MA: {ma\*100:.2f}%, AASR: {aasr\*100:.2f}%")

except Exception as e:

print(f"Round {t+1}/{T} - Aggregation error: {e}")

continue

for client\_socket in client\_sockets:

if client\_socket:

try:

send\_data(client\_socket, current\_weights\_flat)

except socket.error:

client\_sockets[client\_sockets.index(client\_socket)] = None

except Exception as e:

print(f"Error during method {method\_name}: {e}")

finally:

for client\_socket in client\_sockets:

if client\_socket:

try:

client\_socket.close()

except:

pass

print(f"\nResults for Dataset: {dataset\_name}, NIR: {NIR}, AR: {AR\*100}%, PDR: {PDR\*100}%")

print("Method\t\tAASR (%)\tMA (%)")

print("-" \* 40)

for method\_name in methods:

ma = results[method\_name]['MA']

aasr = results[method\_name]['AASR']

if ma is not None and aasr is not None:

print(f"{method\_name:<15}\t{aasr:.2f}\t\t{ma:.2f}")

server\_socket.close()

import socket

import numpy as np

import tensorflow as tf

import threading

import time

import random

def create\_model(input\_shape):

model = tf.keras.Sequential([

tf.keras.layers.Input(shape=(input\_shape,)),

tf.keras.layers.Dense(128, activation='relu'),

tf.keras.layers.Dense(10, activation='softmax')

])

model.compile(optimizer='sgd', loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

return model

def amsparse(delta\_w, layer\_shapes):

amsparse\_updates = []

start = 0

for shape in layer\_shapes:

size = np.prod(shape)

layer\_delta = delta\_w[start:start + size].reshape(shape)

layer\_max = np.max(np.abs(layer\_delta), axis=tuple(range(len(shape) - 1)))

signs = np.sign(layer\_delta)

amsparse\_layer = signs \* layer\_max

amsparse\_updates.append(amsparse\_layer.flatten())

start += size

return np.concatenate(amsparse\_updates)

def generate\_trigger\_pattern(shape, trigger\_type, trigger\_size):

if trigger\_type == 'pixel':

trigger = np.zeros(shape)

trigger[:trigger\_size:2] = 1.0

elif trigger\_type == 'square':

trigger = np.zeros(shape)

trigger[:trigger\_size] = 1.0

elif trigger\_type == 'cross':

trigger = np.zeros(shape)

trigger[trigger\_size//2] = 1.0

else:

trigger = np.random.choice([0, 1], size=shape, p=[0.7, 0.3]).astype(np.float32)

return trigger

def apply\_trigger(x, trigger\_pattern, trigger\_size):

triggered\_x = x.copy()

triggered\_x[:trigger\_size] = np.maximum(triggered\_x[:trigger\_size], trigger\_pattern[:trigger\_size])

return triggered\_x

def send\_data(client\_socket, data):

try:

data\_bytes = data.tobytes()

size = np.array([len(data\_bytes)], dtype=np.int64).tobytes()

client\_socket.sendall(size + data\_bytes)

except socket.error:

raise

def receive\_data(client\_socket):

try:

client\_socket.settimeout(120.0)

size\_data = client\_socket.recv(8)

if not size\_data:

raise socket.error("Server disconnected")

data\_size = np.frombuffer(size\_data, dtype=np.int64)[0]

received\_data = bytearray()

while len(received\_data) < data\_size:

packet = client\_socket.recv(min(4096, data\_size - len(received\_data)))

if not packet:

raise socket.error("Server disconnected during data transfer")

received\_data.extend(packet)

return np.frombuffer(received\_data, dtype=np.float32)

except socket.error:

raise

def client\_thread(client\_id, server\_address, T, batch\_size, input\_shape, dataset\_name):

samples\_per\_client = 6000 if dataset\_name != 'CIFAR10' else 5000

for method\_idx in range(8):

client\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

retry\_count = 0

max\_retries = 10

while retry\_count < max\_retries:

try:

client\_socket.connect(server\_address)

print(f"Client {client\_id} connected to {server\_address} for method {method\_idx}")

break

except ConnectionRefusedError:

retry\_count += 1

time.sleep(2)

if retry\_count == max\_retries:

print(f"Client {client\_id} failed to connect after {max\_retries} retries")

return

try:

client\_data\_flat = receive\_data(client\_socket)

data\_size = samples\_per\_client \* input\_shape

expected\_size = data\_size + samples\_per\_client + 1

if client\_data\_flat.size != expected\_size:

print(f"Client {client\_id} data size mismatch: expected {expected\_size}, got {client\_data\_flat.size}")

client\_socket.close()

return

x\_client = client\_data\_flat[:data\_size].reshape(samples\_per\_client, input\_shape)

y\_client = client\_data\_flat[data\_size:data\_size + samples\_per\_client].astype(np.int32)

is\_malicious = bool(client\_data\_flat[-1])

global\_weights\_flat = receive\_data(client\_socket)

model = create\_model(input\_shape)

global\_weights = [

global\_weights\_flat[i:i + w.size].reshape(w.shape)

for i, w in zip(np.cumsum([0] + [w.size for w in model.get\_weights()[:-1]]), model.get\_weights())

]

model.set\_weights(global\_weights)

layer\_shapes = [w.shape for w in model.get\_weights()]

trigger\_type = random.choice(['pixel', 'square', 'cross', 'random'])

trigger\_size = 5\*28 if dataset\_name != 'CIFAR10' else 8\*32

trigger\_pattern = generate\_trigger\_pattern(input\_shape, trigger\_type, trigger\_size)

for t in range(T):

try:

if is\_malicious:

n\_poison = int(len(x\_client) \* 0.15625)

poisoned\_data = x\_client.copy()

if t % 5 == 0:

trigger\_type = random.choice(['pixel', 'square', 'cross', 'random'])

trigger\_pattern = generate\_trigger\_pattern(input\_shape, trigger\_type, trigger\_size)

poisoned\_data[:n\_poison] = np.array([

apply\_trigger(x, trigger\_pattern, trigger\_size)

for x in poisoned\_data[:n\_poison]

])

poisoned\_labels = y\_client.copy()

poisoned\_labels[:n\_poison] = 7

model.fit(poisoned\_data, poisoned\_labels, epochs=1, batch\_size=batch\_size, verbose=0)

else:

model.fit(x\_client, y\_client, epochs=1, batch\_size=batch\_size, verbose=0)

new\_weights\_flat = np.concatenate([w.flatten() for w in model.get\_weights()]).astype(np.float32)

delta\_w = new\_weights\_flat - global\_weights\_flat

delta\_w\_am = amsparse(delta\_w, layer\_shapes)

updates = np.concatenate([delta\_w, delta\_w\_am]).astype(np.float32)

send\_data(client\_socket, updates)

global\_weights\_flat = receive\_data(client\_socket)

global\_weights = [

global\_weights\_flat[i:i + w.size].reshape(w.shape)

for i, w in zip(np.cumsum([0] + [w.size for w in model.get\_weights()[:-1]]), model.get\_weights())

]

model.set\_weights(global\_weights)

except socket.error:

print(f"Client {client\_id} disconnected during round {t+1}")

break

except Exception as e:

print(f"Client {client\_id} error: {e}")

finally:

try:

client\_socket.close()

except:

pass

# Configuration

server\_address = ('172.16.24.61', 9999)

T = 10

batch\_size = 64

n\_clients = 5

dataset\_configs = {

'MNIST': 784,

'FMNIST': 784,

'CIFAR10': 3072

}

for dataset\_name, input\_shape in dataset\_configs.items():

threads = []

for i in range(n\_clients):

thread = threading.Thread(target=client\_thread, args=(i, server\_address, T, batch\_size, input\_shape, dataset\_name))

threads.append(thread)

thread.start()

for thread in threads:

thread.join()

(tf\_env) C:\Users\login01\Documents>python server2.py

Server listening at ('172.16.24.61', 9999)

Dataset: MNIST, NIR: 0.75

Method: RFBDS

Client 0 connected from ('172.16.24.60', 52913)

Client 1 connected from ('172.16.24.60', 52914)

Client 2 connected from ('172.16.24.60', 52915)

Client 3 connected from ('172.16.24.60', 52917)

Client 4 connected from ('172.16.24.60', 52916)

Round 10/10 - MA: 75.21%, AASR: 97.21%

Method: PrivRFBDS

Client 0 connected from ('172.16.24.60', 52919)

Client 1 connected from ('172.16.24.60', 52920)

Client 2 connected from ('172.16.24.60', 52921)

Client 3 connected from ('172.16.24.60', 52922)

Client 4 connected from ('172.16.24.60', 52923)

Round 10/10 - MA: 31.59%, AASR: 99.60%

Method: FedAvg

Client 0 connected from ('172.16.24.60', 52926)

Client 1 connected from ('172.16.24.60', 52927)

Client 2 connected from ('172.16.24.60', 52928)

Client 3 connected from ('172.16.24.60', 52929)

Client 4 connected from ('172.16.24.60', 52930)

Round 10/10 - MA: 74.72%, AASR: 45.77%

Method: Median

Client 0 connected from ('172.16.24.60', 52931)

Client 1 connected from ('172.16.24.60', 52932)

Client 2 connected from ('172.16.24.60', 52933)

Client 3 connected from ('172.16.24.60', 52934)

Client 4 connected from ('172.16.24.60', 52935)

Round 10/10 - MA: 70.98%, AASR: 33.53%

Method: Trimmed-Mean

Client 0 connected from ('172.16.24.60', 52937)

Client 1 connected from ('172.16.24.60', 52938)

Client 2 connected from ('172.16.24.60', 52939)

Client 3 connected from ('172.16.24.60', 52940)

Client 4 connected from ('172.16.24.60', 52941)

Round 10/10 - MA: 73.37%, AASR: 89.20%

Method: FedCC

Client 0 connected from ('172.16.24.60', 52943)

Client 1 connected from ('172.16.24.60', 52944)

Client 2 connected from ('172.16.24.60', 52945)

Client 3 connected from ('172.16.24.60', 52946)

Client 4 connected from ('172.16.24.60', 52947)

Round 10/10 - MA: 72.95%, AASR: 30.01%

Method: RDFL

Client 0 connected from ('172.16.24.60', 52948)

Client 1 connected from ('172.16.24.60', 52949)

Client 2 connected from ('172.16.24.60', 52950)

Client 3 connected from ('172.16.24.60', 52951)

Client 4 connected from ('172.16.24.60', 52952)

Round 10/10 - MA: 75.28%, AASR: 16.14%

Method: FLAME

Client 0 connected from ('172.16.24.60', 52954)

Client 1 connected from ('172.16.24.60', 52955)

Client 2 connected from ('172.16.24.60', 52956)

Client 3 connected from ('172.16.24.60', 52957)

Client 4 connected from ('172.16.24.60', 52958)

Round 10/10 - MA: 75.10%, AASR: 96.64%

Results for Dataset: MNIST, NIR: 0.75, AR: 40.0%, PDR: 15.625%

Method AASR (%) MA (%)

----------------------------------------

RFBDS 97.21 75.21

PrivRFBDS 99.60 31.59

FedAvg 45.77 74.72

Median 33.53 70.98

Trimmed-Mean 89.20 73.37

FedCC 30.01 72.95

RDFL 16.14 75.28

FLAME 96.64 75.10

Dataset: FMNIST, NIR: 0.75

Method: RFBDS

Client 0 connected from ('172.16.24.60', 52961)

Client 1 connected from ('172.16.24.60', 52962)

Client 2 connected from ('172.16.24.60', 52963)

Client 3 connected from ('172.16.24.60', 52964)

Client 4 connected from ('172.16.24.60', 52965)

Round 10/10 - MA: 65.03%, AASR: 51.09%

Method: PrivRFBDS

Client 0 connected from ('172.16.24.60', 52967)

Client 1 connected from ('172.16.24.60', 52968)

Client 2 connected from ('172.16.24.60', 52969)

Client 3 connected from ('172.16.24.60', 52970)

Client 4 connected from ('172.16.24.60', 52971)

Round 10/10 - MA: 23.57%, AASR: 93.53%

Method: FedAvg

Client 0 connected from ('172.16.24.60', 52972)

Client 1 connected from ('172.16.24.60', 52973)

Client 2 connected from ('172.16.24.60', 52974)

Client 3 connected from ('172.16.24.60', 52975)

Client 4 connected from ('172.16.24.60', 52976)

Round 10/10 - MA: 66.82%, AASR: 8.15%

Method: Median

Client 0 connected from ('172.16.24.60', 52978)

Client 1 connected from ('172.16.24.60', 52979)

Client 2 connected from ('172.16.24.60', 52980)

Client 3 connected from ('172.16.24.60', 52981)

Client 4 connected from ('172.16.24.60', 52982)

Round 10/10 - MA: 60.72%, AASR: 14.13%

Method: Trimmed-Mean

Client 0 connected from ('172.16.24.60', 52983)

Client 1 connected from ('172.16.24.60', 52984)

Client 2 connected from ('172.16.24.60', 52986)

Client 3 connected from ('172.16.24.60', 52985)

Client 4 connected from ('172.16.24.60', 52987)

Round 10/10 - MA: 66.41%, AASR: 32.75%

Method: FedCC

Client 0 connected from ('172.16.24.60', 52991)

Client 1 connected from ('172.16.24.60', 52992)

Client 2 connected from ('172.16.24.60', 52993)

Client 3 connected from ('172.16.24.60', 52994)

Client 4 connected from ('172.16.24.60', 52995)

Round 10/10 - MA: 67.03%, AASR: 24.00%

Method: RDFL

Client 0 connected from ('172.16.24.60', 52996)

Client 1 connected from ('172.16.24.60', 52997)

Client 2 connected from ('172.16.24.60', 52998)

Client 3 connected from ('172.16.24.60', 52999)

Client 4 connected from ('172.16.24.60', 53000)

Round 10/10 - MA: 66.38%, AASR: 39.62%

Method: FLAME

Client 0 connected from ('172.16.24.60', 53003)

Client 1 connected from ('172.16.24.60', 53004)

Client 2 connected from ('172.16.24.60', 53005)

Client 3 connected from ('172.16.24.60', 53006)

Client 4 connected from ('172.16.24.60', 53007)

Round 10/10 - MA: 66.75%, AASR: 8.02%

Results for Dataset: FMNIST, NIR: 0.75, AR: 40.0%, PDR: 15.625%

Method AASR (%) MA (%)

----------------------------------------

RFBDS 51.09 65.03

PrivRFBDS 93.53 23.57

FedAvg 8.15 66.82

Median 14.13 60.72

Trimmed-Mean 32.75 66.41

FedCC 24.00 67.03

RDFL 39.62 66.38

FLAME 8.02 66.75

Dataset: CIFAR10, NIR: 0.75

Method: RFBDS

Client 0 connected from ('172.16.24.60', 53008)

Client 1 connected from ('172.16.24.60', 53009)

Client 2 connected from ('172.16.24.60', 53010)

Client 3 connected from ('172.16.24.60', 53011)

Client 4 connected from ('172.16.24.60', 53012)

Round 10/10 - MA: 17.78%, AASR: 97.84%

Method: PrivRFBDS

Client 0 connected from ('172.16.24.60', 53014)

Client 1 connected from ('172.16.24.60', 53015)

Client 2 connected from ('172.16.24.60', 53016)

Client 3 connected from ('172.16.24.60', 53017)

Client 4 connected from ('172.16.24.60', 53018)

Round 10/10 - MA: 12.23%, AASR: 75.93%

Method: FedAvg

Client 0 connected from ('172.16.24.60', 53024)

Client 1 connected from ('172.16.24.60', 53025)

Client 2 connected from ('172.16.24.60', 53026)

Client 3 connected from ('172.16.24.60', 53027)

Client 4 connected from ('172.16.24.60', 53028)

Round 10/10 - MA: 21.30%, AASR: 68.76%

Method: Median

Client 0 connected from ('172.16.24.60', 53029)

Client 1 connected from ('172.16.24.60', 53030)

Client 2 connected from ('172.16.24.60', 53031)

Client 3 connected from ('172.16.24.60', 53032)

Client 4 connected from ('172.16.24.60', 53033)

Round 10/10 - MA: 26.21%, AASR: 71.66%

Method: Trimmed-Mean

Client 0 connected from ('172.16.24.60', 53035)

Client 1 connected from ('172.16.24.60', 53036)

Client 2 connected from ('172.16.24.60', 53037)

Client 3 connected from ('172.16.24.60', 53038)

Client 4 connected from ('172.16.24.60', 53039)

Round 10/10 - MA: 19.78%, AASR: 70.75%

Method: FedCC

Client 0 connected from ('172.16.24.60', 53041)

Client 1 connected from ('172.16.24.60', 53042)

Client 2 connected from ('172.16.24.60', 53043)

Client 3 connected from ('172.16.24.60', 53044)

Client 4 connected from ('172.16.24.60', 53045)

Round 10/10 - MA: 16.54%, AASR: 81.16%

Method: RDFL

Client 0 connected from ('172.16.24.60', 53047)

Client 1 connected from ('172.16.24.60', 53049)

Client 2 connected from ('172.16.24.60', 53048)

Client 3 connected from ('172.16.24.60', 53050)

Client 4 connected from ('172.16.24.60', 53051)

Round 10/10 - MA: 22.93%, AASR: 86.64%

Method: FLAME

Client 0 connected from ('172.16.24.60', 53053)

Client 1 connected from ('172.16.24.60', 53054)

Client 2 connected from ('172.16.24.60', 53055)

Client 3 connected from ('172.16.24.60', 53056)

Client 4 connected from ('172.16.24.60', 53057)

Round 10/10 - MA: 21.72%, AASR: 66.34%

Results for Dataset: CIFAR10, NIR: 0.75, AR: 40.0%, PDR: 15.625%

Method AASR (%) MA (%)

----------------------------------------

RFBDS 97.84 17.78

PrivRFBDS 75.93 12.23

FedAvg 68.76 21.30

Median 71.66 26.21

Trimmed-Mean 70.75 19.78

FedCC 81.16 16.54

RDFL 86.64 22.93

FLAME 66.34 21.72