------------------Cloud SERVER-------------------

from flask import Flask, request, jsonify

import pandas as pd

import numpy as np

import requests

from sklearn.ensemble import VotingClassifier

from sklearn.svm import SVC

from sklearn.tree import DecisionTreeClassifier

from sklearn.neighbors import KNeighborsClassifier

from sklearn.preprocessing import StandardScaler, OneHotEncoder

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score

import time

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Conv1D, Flatten, Dropout, Input

from sklearn.base import BaseEstimator, ClassifierMixin

app = Flask(\_\_name\_\_)

class CNNClassifier(BaseEstimator, ClassifierMixin):

    def \_\_init\_\_(self, input\_shape):

        self.input\_shape = input\_shape

        self.model = self.buildmodl()

    def buildmodl(self):

        model = Sequential()

        model.add(Input(shape=(self.input\_shape, 1)))

        model.add(Conv1D(64, 2, activation='relu'))

        model.add(Flatten())

        model.add(Dense(800, activation='relu'))

        model.add(Dropout(0.5))

        model.add(Dense(1, activation='sigmoid'))

        model.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy'])

        return model

    def fit(self, X, y):

        X = X.reshape(X.shape[0], self.input\_shape, 1)

        self.model.fit(X, y, epochs=10, batch\_size=32, verbose=1)

        return self

    def predict(self, X):

        X = X.reshape(X.shape[0], self.input\_shape, 1)

        return (self.model.predict(X) > 0.5).astype("int32").flatten()

    def predict\_proba(self, X):

        X = X.reshape(X.shape[0], self.input\_shape, 1)

        proba = self.model.predict(X)

        return np.hstack((1 - proba, proba))

class CloudSrvr:

    def \_\_init\_\_(self):

        self.selectdfeats = [

            "protocol\_type", "service", "flag", "land", "duration", "src\_bytes", "dst\_bytes",

            "wrong\_fragment", "urgent", "hot", "srv\_count", "serror\_rate", "srv\_serror\_rate",

            "rerror\_rate", "srv\_rerror\_rate", "same\_srv\_rate", "diff\_srv\_rate",

            "dst\_host\_count", "dst\_host\_srv\_count",

            "dst\_host\_same\_srv\_rate", "dst\_host\_diff\_srv\_rate",

            "dst\_host\_same\_src\_port\_rate", "dst\_host\_serror\_rate",

            "dst\_host\_srv\_serror\_rate", "dst\_host\_rerror\_rate", "dst\_host\_srv\_rerror\_rate"

        ]

        self.standscalr = StandardScaler()

        self.onehotenc = OneHotEncoder(sparse\_output=False, handle\_unknown='ignore')

        self.data = pd.read\_csv("KDD.csv")

        self.tempbuffr = pd.DataFrame(columns=self.data.columns)

        self.trainall()

        self.client\_weights = []

    def prepdata(self, data, trainng=False):

        catfeats = ["protocol\_type", "service", "flag", "land"]

        data[catfeats] = data[catfeats].fillna('NA').astype(str)

        numfeats = self.selectdfeats[4:]

        data[numfeats] = data[numfeats].fillna(0)

        data = data[self.selectdfeats]

        if trainng:

            encat = self.onehotenc.fit\_transform(data[catfeats])

        else:

            encat = self.onehotenc.transform(data[catfeats])

        if trainng:

            scalenum = self.standscalr.fit\_transform(data[numfeats])

        else:

            scalenum = self.standscalr.transform(data[numfeats])

        return np.hstack((encat, scalenum))

    def trainall(self):

        if not self.tempbuffr.empty:

            data\_combined = pd.concat([self.data, self.tempbuffr], ignore\_index=True)

        else:

            data\_combined = self.data

        X = data\_combined[self.selectdfeats]

        y = data\_combined["classnum"]

        Xtrain, Xtest, ytrain, ytest = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

        Xtrainpre = self.prepdata(Xtrain, trainng=True)

        Xtestpre = self.prepdata(Xtest, trainng=False)

        num\_features = Xtrainpre.shape[1]

        base\_models = [

            ('svm', SVC(probability=True)),

            ('knn', KNeighborsClassifier()),

            ('dt', DecisionTreeClassifier()),

            ('cnn', CNNClassifier(input\_shape=num\_features))

        ]

        for name, model in base\_models:

            if name == "cnn":

                Xtrain\_cnn = Xtrainpre.reshape(Xtrainpre.shape[0], num\_features, 1)

                model.fit(Xtrain\_cnn, ytrain)

            else:

                model.fit(Xtrainpre, ytrain)

        self.ensem = VotingClassifier(

            estimators=base\_models,

            voting='soft'

        )

        self.ensem.fit(Xtrainpre, ytrain)

        self.updateweightsbyaccuracy(Xtestpre, ytest)

        ypred = self.ensem.predict(Xtestpre)

        acc = accuracy\_score(ytest, ypred)

        print(f"Voting Classifier Acc: {acc:.4f}")

    def predict\_batch(self, packets):

        packet\_df = pd.DataFrame(packets)

        Xpreprocessed = self.prepdata(packet\_df, trainng=False)

        num\_features = Xpreprocessed.shape[1]

        X\_cnn = Xpreprocessed.reshape(Xpreprocessed.shape[0], num\_features, 1)

        cnn\_predictions = (self.machinemodels["cnn"].predict(X\_cnn) > 0.5).astype("int32").flatten()

        ensemble\_predictions = self.ensem.predict(Xpreprocessed)

        final\_predictions = ["anomaly" if ens == "anomaly" or cnn == 1 else "normal"

                             for ens, cnn in zip(ensemble\_predictions, cnn\_predictions)]

        self.tempbuffr = pd.concat([self.tempbuffr, packet\_df[np.array(final\_predictions) == "anomaly"]], ignore\_index=True)

        return final\_predictions

    def updateweightsbyaccuracy(self, Xtest, ytest):

        accuracies = []

        for name, model in self.ensem.estimators:

            if name == "cnn":

                Xtest\_cnn = Xtest.reshape(Xtest.shape[0], Xtest.shape[1], 1)

                ypred = (model.predict(Xtest\_cnn) > 0.5).astype("int32").flatten()

            else:

                ypred = model.predict(Xtest)

            acc = accuracy\_score(ytest, ypred)

            accuracies.append(acc)

            print(f"{name} Accuracy: {acc:.4f}")

        total\_accuracy = sum(accuracies)

        normalized\_weights = [acc / total\_accuracy for acc in accuracies]

        self.ensem.weights = normalized\_weights

        print("Normalized Weights:")

        for (name, model), weight in zip(self.ensem.estimators, self.ensem.weights):

            print(f"{name}: {weight:.4f}")

    def get\_weights(self):

        return self.ensem.weights

    def update\_weights\_from\_clients(self, client\_weights):

        if not client\_weights:

            return

        mean\_weights = np.mean(client\_weights, axis=0)

        self.ensem.weights = mean\_weights.tolist()

        print("Updated model weights based on client contributions.")

cloudsrv = CloudSrvr()

def periodtrain():

    while True:

        time.sleep(3600)

        cloudsrv.trainall()

threading.Thread(target=periodtrain, daemon=True).start()

ENTITY\_ID = "cloud\_server\_1"

SESSION\_KEY = None

def register\_with\_tra():

    global SESSION\_KEY

    response = requests.post(

        "http://localhost:6000/register",

        json={"entity\_id": ENTITY\_ID, "entity\_type": "cloud\_server"}

    )

    if response.status\_code == 201:

        SESSION\_KEY = response.json()["session\_key"]

        print("Cloud server registered with TRA. SKEY: ", SESSION\_KEY)

def validate\_request(headers):

    entity\_id = headers.get("Entity-ID")

    nonce = headers.get("Nonce")

    received\_hmac = headers.get("HMAC")

    if not all([entity\_id, nonce, received\_hmac]):

        return False

    response = requests.post(

        "http://localhost:6000/authenticate",

        json={"entity\_id": entity\_id, "nonce": nonce, "hmac": received\_hmac}

    )

    return response.status\_code == 200

@app.route('/predict', methods=['POST'])

def predict\_packet():

    if not validate\_request(request.headers):

        return jsonify({"error": "Authentication failed"}), 401

    packets = request.json

    predictions = cloudsrv.predict\_batch(packets)

    predictions\_list = predictions.tolist()

    return jsonify({"predictions": predictions\_list})

@app.route('/get\_weights', methods=['GET'])

def get\_weights():

    if not validate\_request(request.headers):

        return jsonify({"error": "Authentication failed"}), 401

    weights = cloudsrv.get\_weights()

    return jsonify({"weights": weights})

@app.route('/update\_weights', methods=['POST'])

def update\_weights():

    if not validate\_request(request.headers):

        return jsonify({"error": "Authentication failed"}), 401

    client\_weights = request.json.get("weights")

    if not client\_weights:

        return jsonify({"error": "No weights provided"}), 400

    cloudsrv.client\_weights.append(client\_weights)

    cloudsrv.update\_weights\_from\_clients(cloudsrv.client\_weights)

    return jsonify({"message": "Weights updated successfully"})

register\_with\_tra()

if \_\_name\_\_ == '\_\_main\_\_':

    app.run(host='0.0.0.0', port=5000)

--------------FL Client code------------------

import threading

import time

import requests

from flask import Flask, json, request, jsonify

from scapy.all import \*

import pandas as pd

import numpy as np

import secrets

import hashlib

import hmac

from collections import defaultdict

from sklearn.ensemble import VotingClassifier

from sklearn.svm import SVC

from sklearn.tree import DecisionTreeClassifier

from sklearn.neighbors import KNeighborsClassifier

from sklearn.preprocessing import StandardScaler, OneHotEncoder

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Conv1D, Flatten, Dropout, Input

from sklearn.base import BaseEstimator, ClassifierMixin

app = Flask(\_\_name\_\_)

ENTITY\_ID = "capture\_engine\_1"

SESSION\_KEY = None

TRA\_URL = "http://localhost:6000"

CLOUD\_SERVER\_URL = "http://localhost:5000"

def register\_with\_tra():

    global SESSION\_KEY

    response = requests.post(

        f"{TRA\_URL}/register",

        json={"entity\_id": ENTITY\_ID, "entity\_type": "capture\_engine"}

    )

    if response.status\_code == 201:

        SESSION\_KEY = response.json()["session\_key"]

        print("Successfully registered with TRA. SKEY:", SESSION\_KEY)

    else:

        raise Exception("TRA registration failed")

def generate\_auth\_headers():

    nonce = secrets.token\_hex(16)

    hmac\_val = hmac.new(

        bytes.fromhex(SESSION\_KEY),

        nonce.encode(),

        hashlib.sha256

    ).hexdigest()

    return {

        "Entity-ID": ENTITY\_ID,

        "Nonce": nonce,

        "HMAC": hmac\_val

    }

# Validate TRA authentication

def validate\_request(headers):

    entity\_id = headers.get("Entity-ID")

    nonce = headers.get("Nonce")

    received\_hmac = headers.get("HMAC")

    if not all([entity\_id, nonce, received\_hmac]):

        return False

    response = requests.post(

        f"{TRA\_URL}/authenticate",

        json={"entity\_id": entity\_id, "nonce": nonce, "hmac": received\_hmac}

    )

    return response.status\_code == 200

class CNNClassifier(BaseEstimator, ClassifierMixin):

    def \_\_init\_\_(self, input\_shape):

        self.input\_shape = input\_shape

        self.model = self.\_build\_model()

    def buildmdl(self):

        model = Sequential()

        model.add(Input(shape=(self.input\_shape, 1)))

        model.add(Conv1D(64, 2, activation='relu'))

        model.add(Flatten())

        model.add(Dense(800, activation='relu'))

        model.add(Dropout(0.5))

        model.add(Dense(1, activation='sigmoid'))

        model.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy'])

        return model

    def fit(self, X, y):

        X = X.reshape(X.shape[0], self.input\_shape, 1)

        self.model.fit(X, y, epochs=1, batch\_size=32, verbose=1)

        return self

    def predict(self, X):

        X = X.reshape(X.shape[0], self.input\_shape, 1)

        return (self.model.predict(X) > 0.5).astype("int32").flatten()

    def predict\_proba(self, X):

        X = X.reshape(X.shape[0], self.input\_shape, 1)

        proba = self.model.predict(X)

        return np.hstack((1 - proba, proba))

# Local Predictor

class LocalPredictor:

    def \_\_init\_\_(self):

        self.selectdfeats = [

            "protocol\_type", "service", "flag", "land", "duration", "src\_bytes", "dst\_bytes",

            "wrong\_fragment", "urgent", "hot", "srv\_count", "serror\_rate", "srv\_serror\_rate",

            "rerror\_rate", "srv\_rerror\_rate", "same\_srv\_rate", "diff\_srv\_rate",

            "dst\_host\_count", "dst\_host\_srv\_count",

            "dst\_host\_same\_srv\_rate", "dst\_host\_diff\_srv\_rate",

            "dst\_host\_same\_src\_port\_rate", "dst\_host\_serror\_rate",

            "dst\_host\_srv\_serror\_rate", "dst\_host\_rerror\_rate", "dst\_host\_srv\_rerror\_rate"

        ]

        self.standscalr = StandardScaler()

        self.onehotenc = OneHotEncoder(sparse\_output=False, handle\_unknown='ignore')

        self.data = pd.read\_csv("KDD.csv")

        self.tempbuffr = pd.DataFrame(columns=self.data.columns)

        self.trainall()

    def prepdata(self, data, trainng=False):

        catfeats = ["protocol\_type", "service", "flag", "land"]

        data[catfeats] = data[catfeats].fillna('NA').astype(str)

        numfeats = self.selectdfeats[4:]

        data[numfeats] = data[numfeats].fillna(0)

        data = data[self.selectdfeats]

        if trainng:

            encat = self.onehotenc.fit\_transform(data[catfeats])

        else:

            encat = self.onehotenc.transform(data[catfeats])

        if trainng:

            scalenum = self.standscalr.fit\_transform(data[numfeats])

        else:

            scalenum = self.standscalr.transform(data[numfeats])

        return np.hstack((encat, scalenum))

    def trainall(self):

        if not self.tempbuffr.empty:

            data\_combined = pd.concat([self.data, self.tempbuffr], ignore\_index=True)

        else:

            data\_combined = self.data

        X = data\_combined[self.selectdfeats]

        y = data\_combined["classnum"]

        Xtrain, Xtest, ytrain, ytest = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

        Xtrainpre = self.prepdata(Xtrain, trainng=True)

        Xtestpre = self.prepdata(Xtest, trainng=False)

        num\_features = Xtrainpre.shape[1]

        self.base\_models = [

            ('svm', SVC(probability=True)),

            ('knn', KNeighborsClassifier()),

            ('dt', DecisionTreeClassifier()),

            ('cnn', CNNClassifier(input\_shape=num\_features))

        ]

        for name, model in self.base\_models:

            if name == "cnn":

                Xtrain\_cnn = Xtrainpre.reshape(Xtrainpre.shape[0], num\_features, 1)

                model.fit(Xtrain\_cnn, ytrain)

            else:

                model.fit(Xtrainpre, ytrain)

        self.ensem = VotingClassifier(

            estimators=self.base\_models,

            voting='soft'

        )

        self.ensem.fit(Xtrainpre, ytrain)

        self.update\_weights\_based\_on\_accuracy(Xtestpre, ytest)

        ypred = self.ensem.predict(Xtestpre)

        acc = accuracy\_score(ytest, ypred)

        print(f"Voting Classifier Acc: {acc:.4f}")

    def predict\_batch(self, packets):

        packet\_df = pd.DataFrame(packets)

        Xpreprocessed = self.prepdata(packet\_df, trainng=False)

        num\_features = Xpreprocessed.shape[1]

        X\_cnn = Xpreprocessed.reshape(Xpreprocessed.shape[0], num\_features, 1)

        cnn\_predictions = (self.base\_models["cnn"].predict(X\_cnn) > 0.5).astype("int32").flatten()

        ensemble\_predictions = self.ensem.predict(Xpreprocessed)

        final\_predictions = ["anomaly" if ens == "anomaly" or cnn == 1 else "normal"

                             for ens, cnn in zip(ensemble\_predictions, cnn\_predictions)]

        self.tempbuffr = pd.concat([self.tempbuffr, packet\_df[np.array(final\_predictions) == "anomaly"]], ignore\_index=True)

        return final\_predictions

    def update\_weights\_based\_on\_accuracy(self, Xtest, ytest):

        accuracies = []

        for name, model in self.ensem.estimators:

            if name == "cnn":

                Xtest\_cnn = Xtest.reshape(Xtest.shape[0], Xtest.shape[1], 1)

                ypred = (model.predict(Xtest\_cnn) > 0.5).astype("int32").flatten()

            else:

                ypred = model.predict(Xtest)

            acc = accuracy\_score(ytest, ypred)

            accuracies.append(acc)

            print(f"{name} Accuracy: {acc:.4f}")

        total\_accuracy = sum(accuracies)

        normalized\_weights = [acc / total\_accuracy for acc in accuracies]

        self.ensem.weights = normalized\_weights

        print("Normalized Weights:")

        for (name, model), weight in zip(self.ensem.estimators, self.ensem.weights):

            print(f"{name}: {weight:.4f}")

    def get\_weights(self):

        return self.ensem.weights

    def set\_weights(self, weights):

        if len(weights) == len(self.ensem.estimators):

            self.ensem.weights = weights

            print("Weights updated successfully.")

        else:

            print("Error: Invalid number of weights provided.")

local\_predictor = LocalPredictor()

@app.route('/get\_weights', methods=['GET'])

def get\_weights():

    if not validate\_request(request.headers):

        return jsonify({"error": "Authentication failed"}), 401

    weights = local\_predictor.get\_weights()

    return jsonify({"weights": weights})

@app.route('/update\_weights', methods=['POST'])

def update\_weights():

    if not validate\_request(request.headers):

        return jsonify({"error": "Authentication failed"}), 401

    weights = request.json.get("weights")

    if not weights:

        return jsonify({"error": "No weights provided"}), 400

    local\_predictor.set\_weights(weights)

    return jsonify({"message": "Weights updated successfully"})

featureslist = []

batch\_data = []

sessions = defaultdict(list)

protocol\_map = {6: "tcp", 17: "udp", 1: "icmp"}

service\_mapping = {

    80: "http", 21: "ftp", 23: "telnet", 25: "smtp", 443: "https", 22: "ssh",

    53: "dns", 110: "pop3", 995: "pop3s", 143: "imap", 993: "imaps", 161: "snmp",

    3306: "mysql", 5432: "postgresql", 8080: "http\_alt"

}

def extractfeatures(packet):

    if IP in packet and TCP in packet:

        sessionkey = (packet[IP].src, packet[IP].dst, packet[TCP].sport, packet[TCP].dport)

        sessions[sessionkey].append(packet)

def computesessionfeatures(sessionpackt):

    features = {}

    features["protocol\_type"] = protocol\_map.get(sessionpackt[0][IP].proto, "other")

    features["service"] = str(sessionpackt[0][TCP].dport)

    features["service"] = pd.Series([features["service"]]).map(service\_mapping).fillna("other").iloc[0]

    features["flag"] = str(sessionpackt[0][TCP].flags)

    features["duration"] = sessionpackt[-1].time - sessionpackt[0].time

    features["land"] = int(sessionpackt[0][IP].src == sessionpackt[0][IP].dst and sessionpackt[0][TCP].sport == sessionpackt[0][TCP].dport)

    srcbytes = sum(len(p) for p in sessionpackt if p[IP].src == sessionpackt[0][IP].src)

    dstbytes = sum(len(p) for p in sessionpackt if p[IP].dst == sessionpackt[0][IP].dst)

    wrong\_fragment = sum(1 for p in sessionpackt if p.haslayer(IP) and p[IP].flags == 1)

    urgent = any(p.haslayer(TCP) and getattr(p[TCP], 'urg', 0) for p in sessionpackt)

    serror\_count = sum(1 for p in sessionpackt if p.haslayer(TCP) and p[TCP].flags & 0x04)

    rerror\_count = sum(1 for p in sessionpackt if p.haslayer(TCP) and p[TCP].flags & 0x01)

    hot = len(set(p[IP].src for p in sessionpackt))

    srv\_count = len(sessionpackt)

    diffsrvcount = set(p[IP].dst for p in sessionpackt)

    features["src\_bytes"] = srcbytes

    features["dst\_bytes"] = dstbytes

    features["wrong\_fragment"] = wrong\_fragment

    features["urgent"] = int(urgent)

    features["hot"] = hot

    features["srv\_count"] = srv\_count

    features["serror\_rate"] = serror\_count / srv\_count if srv\_count > 0 else 0

    features["srv\_serror\_rate"] = features["serror\_rate"]

    features["rerror\_rate"] = rerror\_count / srv\_count if srv\_count > 0 else 0

    features["srv\_rerror\_rate"] = features["rerror\_rate"]

    features["same\_srv\_rate"] = len(set(p[TCP].dport for p in sessionpackt)) / srv\_count if srv\_count > 0 else 0

    features["diff\_srv\_rate"] = len(diffsrvcount) / srv\_count if srv\_count > 0 else 0

    features["srv\_diff\_host\_rate"] = len(set(p[IP].src for p in sessionpackt)) / srv\_count if srv\_count > 0 else 0

    features["dst\_host\_count"] = len(diffsrvcount)

    features["dst\_host\_srv\_count"] = len(set((p[IP].dst, p[TCP].dport) for p in sessionpackt))

    dst\_host\_same\_srv\_rate = sum(1 for p in sessionpackt if p[TCP].dport == sessionpackt[0][TCP].dport) / srv\_count if srv\_count > 0 else 0

    dst\_host\_diff\_srv\_rate = len(set(p[TCP].dport for p in sessionpackt)) / srv\_count if srv\_count > 0 else 0

    dst\_host\_same\_src\_port\_rate = sum(1 for p in sessionpackt if p[TCP].sport == sessionpackt[0][TCP].sport) / srv\_count if srv\_count > 0 else 0

    dst\_host\_serror\_rate = serror\_count / srv\_count if srv\_count > 0 else 0

    dst\_host\_srv\_serror\_rate = dst\_host\_serror\_rate

    dst\_host\_rerror\_rate = rerror\_count / srv\_count if srv\_count > 0 else 0

    dst\_host\_srv\_rerror\_rate = dst\_host\_rerror\_rate

    features["dst\_host\_same\_srv\_rate"] = dst\_host\_same\_srv\_rate

    features["dst\_host\_diff\_srv\_rate"] = dst\_host\_diff\_srv\_rate

    features["dst\_host\_same\_src\_port\_rate"] = dst\_host\_same\_src\_port\_rate

    features["dst\_host\_srv\_diff\_host\_rate"] = len(set(p[IP].src for p in sessionpackt)) / srv\_count if srv\_count > 0 else 0

    features["dst\_host\_serror\_rate"] = dst\_host\_serror\_rate

    features["dst\_host\_srv\_serror\_rate"] = dst\_host\_srv\_serror\_rate

    features["dst\_host\_rerror\_rate"] = dst\_host\_rerror\_rate

    features["dst\_host\_srv\_rerror\_rate"] = dst\_host\_srv\_rerror\_rate

    return features

def packetcallback(packet):

    extractfeatures(packet)

    if len(sessions) > 0:

        for sessionkey in list(sessions.keys()):

            sessionpackt = sessions[sessionkey]

            if len(sessionpackt) > 1:

                features = computesessionfeatures(sessionpackt)

                dataset\_columns = [

                   "duration", "protocol\_type", "service", "flag", "src\_bytes", "dst\_bytes", "land",

                    "wrong\_fragment", "urgent", "hot", "srv\_count", "serror\_rate", "srv\_serror\_rate",

                    "rerror\_rate", "srv\_rerror\_rate", "same\_srv\_rate", "diff\_srv\_rate", "srv\_diff\_host\_rate",

                    "dst\_host\_count", "dst\_host\_srv\_count", "dst\_host\_same\_srv\_rate", "dst\_host\_diff\_srv\_rate",

                    "dst\_host\_same\_src\_port\_rate", "dst\_host\_srv\_diff\_host\_rate", "dst\_host\_serror\_rate",

                    "dst\_host\_srv\_serror\_rate", "dst\_host\_rerror\_rate", "dst\_host\_srv\_rerror\_rate"

                ]

                ordered\_features = {col: features.get(col, 0) for col in dataset\_columns}

                featureslist.append(ordered\_features)

                batch\_data.append(features)

                if len(batch\_data) >= 10:

                    process\_batch()

def process\_batch():

    global batch\_data

    try:

        batch\_df = pd.DataFrame(batch\_data)

        required\_columns = local\_predictor.selectdfeats

        for col in required\_columns:

            if col not in batch\_df.columns:

                batch\_df[col] = 0

        predictions = local\_predictor.predict\_batch(batch\_df)

        print("Batch processed. Predictions:", predictions)

        if all(pred == "anomaly" for pred in predictions):

            requests.post("http://localhost:5550/error")

        batch\_data = []

    except Exception as e:

        print(f"Error processing batch: {e}")

def fetch\_weights\_from\_cloud():

    try:

        headers = generate\_auth\_headers()

        response = requests.get(

            f"{CLOUD\_SERVER\_URL}/get\_weights",

            headers=headers

        )

        if response.status\_code == 200:

            weights = response.json().get("weights")

            return weights

        else:

            print(f"Failed to fetch weights: {response.status\_code}")

            return None

    except Exception as e:

        print(f"Error fetching weights from cloud: {e}")

        return None

def send\_weights\_to\_cloud(weights):

    try:

        headers = generate\_auth\_headers()

        response = requests.post(

            f"{CLOUD\_SERVER\_URL}/update\_weights",

            json={"weights": weights},

            headers=headers

        )

        if response.status\_code == 200:

            print("Weights sent to cloud server successfully.")

            return True

        else:

            print(f"Failed to send weights: {response.status\_code}")

            return False

    except Exception as e:

        print(f"Error sending weights to cloud: {e}")

        return False

def periodtrain():

    while True:

        time.sleep(3600)

        local\_weights = local\_predictor.get\_weights()

        if not local\_weights:

            print("No local weights available.")

            continue

        if not send\_weights\_to\_cloud(local\_weights):

            print("Failed to send weights to the cloud server.")

            continue

        updated\_weights = fetch\_weights\_from\_cloud()

        if not updated\_weights:

            print("Failed to fetch updated weights from the cloud server.")

            continue

        local\_predictor.set\_weights(updated\_weights)

        print("Local model weights updated successfully.")

threading.Thread(target=periodtrain, daemon=True).start()

def start\_flask\_server():

    app.run(host='0.0.0.0', port=5000)

register\_with\_tra()

threading.Thread(target=start\_flask\_server, daemon=True).start()

print("Starting packet capture...")

sniff(prn=packetcallback ,store=0, filter="tcp")

print("Saving Captured Packet Features...")

pd.DataFrame(featureslist).to\_csv('packet\_features.csv', index=False)

print("✅ Packet Features Saved to packet\_features.csv")

------------TrustedRegistrationAuthority.py----------

from flask import Flask, request, jsonify

import secrets

import hashlib

import hmac

app = Flask(\_\_name\_\_)

entities = {}

secret\_key = secrets.token\_bytes(32)

@app.route('/register', methods=['POST'])

def register():

data = request.json

entity\_id = data.get('entity\_id')

entity\_type = data.get('entity\_type')

if not entity\_id or not entity\_type:

return jsonify({"error": "missing entity\_id/entitytype"}), 400

if entity\_id in entities:

return jsonify({"error": "entity already registered"}), 409

session\_key = secrets.token\_bytes(32)

entities[entity\_id] = {

"type": entity\_type,

"session\_key": session\_key.hex(),

"status": "active"

}

return jsonify({

"session\_key": session\_key.hex(),

"message": "registration successful"

}), 201

@app.route('/authenticate', methods=['POST'])

def authenticate():

data = request.json

entity\_id = data.get('entity\_id')

nonce = data.get('nonce')

received\_hmac = data.get('hmac')

if not all([entity\_id, nonce, received\_hmac]):

return jsonify({"error": "missing authentication parameters"}), 400

entity = entities.get(entity\_id)

if not entity:

return jsonify({"error": "unknown entity"}), 404

session\_key = bytes.fromhex(entity['session\_key'])

valid\_hmac = hmac.new(session\_key, nonce.encode(), hashlib.sha256).hexdigest()

if hmac.compare\_digest(received\_hmac, valid\_hmac):

return jsonify({"status": "authenticated"}), 200

else:

return jsonify({"error": "authentication failed"}), 401

if \_\_name\_\_ == '\_\_main\_\_':

app.run(host='0.0.0.0', port=6000)

----------CloudServer.py (Base paper)-------------

import threading

from flask import Flask, app, render\_template, request, jsonify

import pandas as pd

import hmac

import hashlib

import numpy as np

import requests

from sklearn.ensemble import VotingClassifier

from sklearn.svm import SVC

from sklearn.tree import DecisionTreeClassifier

from sklearn.neighbors import KNeighborsClassifier

from sklearn.preprocessing import StandardScaler, OneHotEncoder

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score

import threading

import time

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Conv1D, Flatten, Dropout, Input

app = Flask(\_\_name\_\_)

class cloudserver:

def \_\_init\_\_(self):

self.selectedfeatures = [

"protocol\_type", "service", "flag", "land", "duration", "src\_bytes", "dst\_bytes",

"wrong\_fragment", "urgent", "hot", "srv\_count", "serror\_rate", "srv\_serror\_rate",

"rerror\_rate", "srv\_rerror\_rate", "same\_srv\_rate", "diff\_srv\_rate",

"dst\_host\_count", "dst\_host\_srv\_count",

"dst\_host\_same\_srv\_rate", "dst\_host\_diff\_srv\_rate",

"dst\_host\_same\_src\_port\_rate", "dst\_host\_serror\_rate",

"dst\_host\_srv\_serror\_rate", "dst\_host\_rerror\_rate", "dst\_host\_srv\_rerror\_rate"

]

self.machinelearningmodels = {

"svm": SVC(probability=True),

"knn": KNeighborsClassifier(),

"dt": DecisionTreeClassifier()

}

self.ensemble = None

self.standardscaler = StandardScaler()

self.onehotencoder = OneHotEncoder(sparse\_output=False, handle\_unknown='ignore')

self.data = pd.read\_csv("KDD.csv")

self.temporarybuffer = pd.DataFrame(columns=self.data.columns)

self.trainallmodels()

def buildcnn(self, inputshape):

model = Sequential()

model.add(Input(shape=(inputshape, 1)))

model.add(Conv1D(64, 2, activation='relu'))

model.add(Flatten())

model.add(Dense(800, activation='relu'))

model.add(Dropout(0.5))

model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy'])

return model

def preprocessdata(self, data, training=False):

categoricalfeatures = ["protocol\_type", "service", "flag", "land"]

data[categoricalfeatures] = data[categoricalfeatures].fillna('NA').astype(str)

numericalfeatures = self.selectedfeatures[4:]

data[numericalfeatures] = data[numericalfeatures].fillna(0)

data = data[self.selectedfeatures]

if training:

encodedcategories = self.onehotencoder.fit\_transform(data[categoricalfeatures])

else:

encodedcategories = self.onehotencoder.transform(data[categoricalfeatures])

if training:

scalednumbers = self.standardscaler.fit\_transform(data[numericalfeatures])

else:

scalednumbers = self.standardscaler.transform(data[numericalfeatures])

return np.hstack((encodedcategories, scalednumbers))

def trainallmodels(self):

if not self.temporarybuffer.empty:

combineddata = pd.concat([self.data, self.temporarybuffer], ignore\_index=True)

else:

combineddata = self.data

X = combineddata[self.selectedfeatures]

y = combineddata["classnum"]

Xtrain, Xtest, ytrain, ytest = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

Xtrainprocessed = self.preprocessdata(Xtrain, training=True)

Xtestprocessed = self.preprocessdata(Xtest, training=False)

pca=PCA(n\_components=0.95)

Xtrainpreprocessed=pca.fit\_transform(Xtrainpreprocessed)

Xtestpreprocessed=pca.transform(Xtestpreprocessed)

numfeatures = Xtrainprocessed.shape[1]

Xtraincnn = Xtrainprocessed.reshape(Xtrainprocessed.shape[0], numfeatures, 1)

Xtestcnn = Xtestprocessed.reshape(Xtestprocessed.shape[0], numfeatures, 1)

self.machinelearningmodels["cnn"] = self.buildcnn(numfeatures)

for name, model in self.machinelearningmodels.items():

if name == "cnn":

model.fit(Xtraincnn, ytrain, epochs=10, batch\_size=32, verbose=1)

ypred = (model.predict(Xtestcnn) > 0.5).astype("int32")

else:

model.fit(Xtrainprocessed, ytrain)

ypred = model.predict(Xtestprocessed)

acc = accuracy\_score(ytest, ypred)

print(f"{name} Acc: {acc:.4f}")

self.ensemble = VotingClassifier(

estimators=[(name, model) for name, model in self.machinelearningmodels.items() if name != "cnn"],

voting='soft'

)

self.ensemble.fit(Xtrainprocessed, ytrain)

enspre = self.ensemble.predict(Xtestprocessed)

ensacc = accuracy\_score(ytest, enspre)

cnnpredictions = (self.machinelearningmodels["cnn"].predict(Xtestcnn) > 0.5).astype("int32")

combinedpredictions = (enspre + cnnpredictions.flatten()) / 2

combinedpredictions = (combinedpredictions > 0.5).astype("int32")

combinedacc = accuracy\_score(ytest, combinedpredictions)

print("Combined CNN and Ensemble Accuracy is:", combinedacc)

def predict(self, packet):

packetdf = pd.DataFrame([packet])

Xprocessed = self.preprocessdata(packetdf, training=False)

numfeatures = Xprocessed.shape[1]

Xcnn = Xprocessed.reshape(1, numfeatures, 1)

cnnpredict = (self.machinelearningmodels["cnn"].predict(Xcnn) > 0.5).astype("int32")[0][0]

ensemblepredict = self.ensemble.predict(Xprocessed)

finalprediction = "anomaly" if ensemblepredict[0] == "anomaly" or cnnpredict==1 else "normal"

if finalprediction == "anomaly":

self.temporarybuffer = pd.concat([self.temporarybuffer, packetdf], ignore\_index=True)

return finalprediction

cloudserver = cloudserver()

def periodictraining():

while True:

time.sleep(3600)

cloudserver.trainallmodels()

threading.Thread(target=periodictraining, daemon=True).start()

entityid = "cloud\_server\_1"

sessionkey = None

def registerwithtra():

global sessionkey

response = requests.post(

"http://localhost:6000/register",

json={"entity\_id": entityid, "entity\_type": "cloud\_server"}

)

if response.status\_code == 201:

sessionkey = response.json()["session\_key"]

print("Cloud server registered with TRA. SKEY: ",sessionkey)

else:

print("registration error")

def validaterequest(headers):

entityid = headers.get("Entity-ID")

nonce = headers.get("Nonce")

receivedhmac = headers.get("HMAC")

if not all([entityid, nonce, receivedhmac]):

return False

response = requests.post(

"http://localhost:6000/authenticate",

json={"entity\_id": entityid, "nonce": nonce, "hmac": receivedhmac}

)

return response.status\_code == 200

@app.route('/predict', methods=['POST'])

def predictpacket():

if not validaterequest(request.headers):

return jsonify({"error": "Authentication failed"}), 401

packet = request.json

prediction = cloudserver.predict(packet)

return jsonify({"prediction": prediction})

@app.route('/status', methods=['GET'])

def showstatus():

stat = "Ready."

return render\_template('status.html', status=stat)

registerwithtra()

if \_\_name\_\_ == '\_\_main\_\_':

app.run(host='0.0.0.0', port=5000)

----------CaptureEngine.py (Base paper)-------------

from scapy.all import \*

import pandas as pd

import requests

import secrets

import hashlib

import hmac

from collections import defaultdict

tra\_url = "http://localhost:6000"

ml\_server\_url = "http://ensemble-learning-based-intrusion.onrender.com:5000/predict"

entity\_id = "capture\_engine\_1"

session\_key = None

def registerwithtra():

global session\_key

response = requests.post(

f"{tra\_url}/register",

json={"entity\_id": entity\_id, "entity\_type": "capture\_engine"}

)

if response.status\_code == 201:

session\_key = response.json()["session\_key"]

print("Successfully registered with TRA. SKEY:", session\_key)

else:

raise Exception("TRA registration failed")

def generateauthheaders():

nonce = secrets.token\_hex(16)

hmacval = hmac.new(

bytes.fromhex(session\_key),

nonce.encode(),

hashlib.sha256

).hexdigest()

return {

"Entity-ID": entity\_id,

"Nonce": nonce,

"HMAC": hmacval

}

featureslist = []

sessions = defaultdict(list)

protocolmap = {6: "tcp", 17: "udp", 1: "icmp"}

servicemapping = {

80: "http", 21: "ftp", 23: "telnet", 25: "smtp", 443: "https", 22: "ssh",

53: "dns", 110: "pop3", 995: "pop3s", 143: "imap", 993: "imaps", 161: "snmp",

3306: "mysql", 5432: "postgresql", 8080: "http\_alt"

}

def extractfeatures(packet):

if IP in packet and TCP in packet:

sessionkey = (packet[IP].src, packet[IP].dst, packet[TCP].sport, packet[TCP].dport)

sessions[sessionkey].append(packet)

def computesessionfeatures(sessionpackt):

features = {}

features["protocol\_type"] = protocolmap.get(sessionpackt[0][IP].proto, "other")

features["service"] = str(sessionpackt[0][TCP].dport)

features["service"] = pd.Series([features["service"]]).map(servicemapping).fillna("other").iloc[0]

features["flag"] = str(sessionpackt[0][TCP].flags)

features["duration"] = sessionpackt[-1].time - sessionpackt[0].time

features["land"] = int(sessionpackt[0][IP].src == sessionpackt[0][IP].dst and sessionpackt[0][TCP].sport == sessionpackt[0][TCP].dport)

srcbytes = sum(len(p) for p in sessionpackt if p[IP].src == sessionpackt[0][IP].src)

dstbytes = sum(len(p) for p in sessionpackt if p[IP].dst == sessionpackt[0][IP].dst)

wrongfragment = sum(1 for p in sessionpackt if p.haslayer(IP) and p[IP].flags == 1)

urgent = any(p.haslayer(TCP) and getattr(p[TCP], 'urg', 0) for p in sessionpackt)

serrorcount = sum(1 for p in sessionpackt if p.haslayer(TCP) and p[TCP].flags & 0x04)

rerrorcount = sum(1 for p in sessionpackt if p.haslayer(TCP) and p[TCP].flags & 0x01)

srvcount = len(sessionpackt)

diffsrvcount = len(set(p[IP].dst for p in sessionpackt))

features["src\_bytes"] = srcbytes

features["dst\_bytes"] = dstbytes

features["wrong\_fragment"] = wrongfragment

features["urgent"] = int(urgent)

features["hot"] = hot

features["srv\_count"] = srvcount

features["serror\_rate"] = serrorcount / srvcount if srvcount > 0 else 0

features["srv\_serror\_rate"] = features["serror\_rate"]

features["rerror\_rate"] = rerrorcount / srvcount if srvcount > 0 else 0

features["srv\_rerror\_rate"] = features["rerror\_rate"]

features["same\_srv\_rate"] = len(set(p[TCP].dport for p in sessionpackt)) / srvcount if srvcount > 0 else 0

features["diff\_srv\_rate"] = len(diffsrvcount) / srvcount if srvcount > 0 else 0

features["srv\_diff\_host\_rate"] = len(set(p[IP].src for p in sessionpackt)) / srvcount if srvcount > 0 else 0

features["dst\_host\_count"] = len(diffsrvcount)

features["dst\_host\_srv\_count"] = len(set((p[IP].dst, p[TCP].dport) for p in sessionpackt))

dsthostsame\_srvrate = sum(1 for p in sessionpackt if p[TCP].dport == sessionpackt[0][TCP].dport) / srvcount if srvcount > 0 else 0

dsthostdiff\_srvrate = len(set(p[TCP].dport for p in sessionpackt)) / srvcount if srvcount > 0 else 0

dsthostsame\_srcportrate = sum(1 for p in sessionpackt if p[TCP].sport == sessionpackt[0][TCP].sport) / srvcount if srvcount > 0 else 0

dsthostserrorrate = serrorcount / srvcount if srvcount > 0 else 0

dsthostsrvserrorrate = dsthostserrorrate

dsthostrerrorrate = rerrorcount / srvcount if srvcount > 0 else 0

dsthostsrvrerrorrate = dsthostrerrorrate

features["dst\_host\_same\_srv\_rate"] = dsthostsame\_srvrate

features["dst\_host\_diff\_srv\_rate"] = dsthostdiff\_srvrate

features["dst\_host\_same\_src\_port\_rate"] = dsthostsame\_srcportrate

features["dst\_host\_srv\_diff\_host\_rate"] = len(set(p[IP].src for p in sessionpackt)) / srvcount if srvcount > 0 else 0

features["dst\_host\_serror\_rate"] = dsthostserrorrate

features["dst\_host\_srv\_serror\_rate"] = dsthostsrvserrorrate

features["dst\_host\_rerror\_rate"] = dsthostrerrorrate

features["dst\_host\_srv\_rerror\_rate"] = dsthostsrvrerrorrate

return features

def packetcallback(packet):

extractfeatures(packet)

if len(sessions) > 0:

for sessionkey in list(sessions.keys()):

sessionpackt = sessions[sessionkey]

if len(sessionpackt) > 1:

features = computesessionfeatures(sessionpackt)

featureslist.append(features)

sendtoserver(features)

def sendtoserver(features):

try:

headers = generateauthheaders()

response = requests.post(

ml\_server\_url,

json=features,

headers=headers

)

if response.status\_code == 200:

result = response.json()

print("Packet received. Prediction from Cloud server: ",result)

if result.get("prediction") == "anomaly":

requests.post("http://localhost:5550/error")

except Exception as e:

print("Communication error:”, e)

registerwithtra()

print("Starting packet capture...")

sniff(prn=packetcallback, store=0, filter="tcp and port 5550")

pd.DataFrame(featureslist).to\_csv('packet\_features.csv', index=False)

print("Packet Features Saved to packet\_features.csv")

-----------------HospitalServer.js--------------

const express=require('express')

const axios=require('axios')

const crypto=require('crypto')

const app=express()

const port=5550

const traurl="http://localhost:6000"

const entityid="hospital\_server\_1"

let sessionkey=null

async function registerwithtra() {

try {

const response=await axios.post(`${traurl}/register`, {

entity\_id:entityid,

entity\_type:"hospital\_server"

})

sessionkey=response.data.session\_key

console.log("hospital server registered with tra")

} catch (error) {

console.error("registration failed:", error.message)

}

}

function generateauthheaders() {

const nonce=crypto.randomBytes(16).toString('hex')

const hmac=crypto.createHmac('sha256', sessionkey)

.update(nonce)

.digest('hex')

return {

"entity-id":entityid,

"nonce":nonce,

"hmac":hmac

}

}

let sensorData={}

let intrusion=false

app.post('/data', (req, res) => {

const {id,bpm}=req.body

if(!id||bpm===undefined) {

return res.status(400).json({message:'invalid input. please provide id and bpm.'})

}

const timestamp=moment().tz("Asia/Kolkata").format('DD-MM-YYYY HH:mm:ss')

if(!sensorData[id]) {

sensorData[id]=[]

}

sensorData[id].push({id,timestamp,bpm})

return res.status(201).json({message:'data added successfully.',d:{id,timestamp,bpm}})

})

app.post('/error', (req, res) => {

console.log(" intrusion detected! alert sent to hospital authorities.")

intrusion=true

res.status(201).json({message:'system intrusion detected!'})

})

app.get('/data', (req, res) => {

return res.status(200).json(sensorData)

})

app.get('/data/:id', (req, res) => {

const {id}=req.params

if(!sensorData[id]) {

return res.status(404).json({message:'data not found.'})

}

return res.status(200).json(sensorData[id])

})

app.get('/', (req, res) => {

const htmlcontent=`

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Sensor Data</title>

<script src="https://cdn.jsdelivr.net/npm/chart.js"></script>

<style>

body { font-family: Arial, sans-serif; background-color: #f4f7f6; text-align: center; padding: 20px; }

table { width: 80%; margin: 20px auto; border-collapse: collapse; background-color: white; }

th, td { border: 1px solid #ddd; padding: 10px; text-align: center; }

th { background-color: #4CAF50; color: white; }

canvas { max-width: 800px; margin: 20px auto; display: block; }

</style>

</head>

<body>

<h1>Sensor Data Dashboard</h1>

<script>

fetch('/error-status')

.then(response => response.json())

.then(status => {

if (status.intrusion) {

alert("⚠️ Intrusion Detected!");

}

});

fetch('/data')

.then(response => response.json())

.then(data => {

const tableBody = document.getElementById('table-body');

Object.keys(data).forEach(sensor => {

data[sensor].forEach(item => {

const row = document.createElement('tr');

row.innerHTML = \`

<td>\${sensor}</td>

<td>\${item.timestamp.split(' ')[0]}</td>

<td>\${item.timestamp.split(' ')[1]}</td>

<td>\${item.bpm}</td>

\`;

tableBody.appendChild(row);

});

});

});

const sensorDropdown = document.getElementById('sensor-dropdown');

const sensorGraph = document.getElementById('sensorGraph');

const chartCtx = sensorGraph.getContext('2d');

let chart;

sensorDropdown.addEventListener('change', (event) => {

const sensorId = event.target.value;

if (chart) {

chart.destroy();

}

if (sensorId === 'none') {

sensorGraph.style.display = 'none';

} else {

fetch(\`/data/\${sensorId}\`)

.then(response => response.json())

.then(sensorData => {

const labels = sensorData.map(item => item.timestamp);

const bpmData = sensorData.map(item => item.bpm);

sensorGraph.style.display = 'block';

chart = new Chart(chartCtx, {

type: 'line',

data: {

labels: labels,

datasets: [{

label: 'BPM over Time',

data: bpmData,

borderColor: 'rgba(75, 192, 192, 1)',

backgroundColor: 'rgba(75, 192, 192, 0.2)',

fill: true

}]

},

options: {

scales: {

x: { title: { display: true, text: 'Date & Time' } },

y: { title: { display: true, text: 'BPM' } }

}

}

});

});

}

});

</script>

<table>

<thead>

<tr>

<th>Sensor ID</th>

<th>Date</th>

<th>Time</th>

<th>BPM</th>

</tr>

</thead>

<tbody id="table-body"></tbody>

</table>

<select id="sensor-dropdown">

<option value="none">None</option>

<option value="sensor1">Sensor 1</option>

<option value="sensor2">Sensor 2</option>

</select>

<canvas id="sensorGraph"></canvas>

</body>

</html>

`;

res.send(htmlcontent);

})

app.get('/error-status', (req, res) => {

return res.status(200).json({ intrusion });

})

registerwithtra();

app.listen(port, () => console.log(`hospital server running on port ${port}`));

-----------ProxyServer.js-------------

const express=require('express')

const axios=require('axios')

const crypto=require('crypto')

const moment=require('moment-timezone')

const app=express()

const port=5551

const hospitalUrl="http://localhost:5550"

const traUrl="http://localhost:6000"

const entityId="proxyserver1"

let sessionKey=null

app.use(express.json())

async function registerWithTra() {

try {

const response=await axios.post(`${traUrl}/register`, {

entity\_id:entityId,

entity\_type:"proxy\_server"

})

sessionKey=response.data.session\_key

console.log("proxy server registered with tra. skey:", sessionKey)

} catch (error) {

console.error("tra registration failed. retrying...", error.message)

setTimeout(registerWithTra, 5000)

}

}

function generateHmac(nonce) {

return crypto.createHmac('sha256', sessionKey)

.update(nonce)

.digest('hex')

}

function validateRequest(headers) {

const entityid=headers["entity-id"]

const nonce=headers["nonce"]

const receivedHmac=headers["hmac"]

if(!entityid||!nonce||!receivedHmac) {

return false

}

const computedHmac=generateHmac(nonce)

return crypto.timingSafeEqual(Buffer.from(receivedHmac), Buffer.from(computedHmac))

}

async function forwardRequest(req,res,endpoint) {

try {

if(!validateRequest(req.headers)) {

return res.status(401).json({message:'authentication failed'})

}

print("forwarding a request to the original hospital server.")

const response=await axios.post(`${hospitalUrl}${endpoint}`, req.body, {

headers: {

"entity-id":entityId,

"nonce":crypto.randomBytes(16).toString('hex'),

"hmac":generateHmac(crypto.randomBytes(16).toString('hex'))

}

})

res.status(response.status).json(response.data)

} catch (error) {

console.error("error forwarding request:", error.message)

res.status(500).json({message:'internal server error'})

}

}

app.post('/data', (req,res) => {

forwardRequest(req,res,'/data')

})

app.post('/error', (req,res) => {

forwardRequest(req,res,'/error')

})

app.get('/data', async (req,res) => {

try {

const response=await axios.get(`${hospitalUrl}/data`)

res.status(response.status).json(response.data)

} catch (error) {

console.error("error retrieving sensor data:", error.message)

res.status(500).json({message:'internal server error'})

}

})

app.get('/data/:id', async (req,res) => {

try {

const response=await axios.get(`${hospitalUrl}/data/${req.params.id}`)

res.status(response.status).json(response.data)

} catch (error) {

console.error("error retrieving sensor data:", error.message)

res.status(500).json({message:'internal server error'})

}

})

app.get('/', async (req,res) => {

try {

const response=await axios.get(`${hospitalUrl}/`)

res.send(response.data)

} catch (error) {

console.error("error serving web ui:", error.message)

res.status(500).json({message:'internal server error'})

}

})

app.get('/error-status', async (req,res) => {

try {

const response=await axios.get(`${hospitalUrl}/error-status`)

res.status(response.status).json(response.data)

} catch (error) {

console.error("error retrieving intrusion status:", error.message)

res.status(500).json({message:'internal server error'})

}

})

registerWithTra()

app.listen(port, () => console.log(`proxy server running on port ${port}`))

----------------pulsesensor.ino--------------

#include <PulseSensorPlayground.h>

#include <WiFi.h>

#include <HTTPClient.h>

#include <mbedtls/md.h>

const int pulsePin=34;

const int threshold=550;

const char\* wifiSsid="moto g32";

const char\* wifiPassword="sriganesh";

const char\* serverUrl="https://eids-hs.onrender.com/data";

const char\* traUrl="http://localhost:6000";

const char\* entityId="pulse\_sensor\_1";

String sessionKey;

PulseSensorPlayground pulseSensor;

void setup() {

Serial.begin(115200);

Serial.println("Connecting to Wi-Fi...");

WiFi.begin(wifiSsid, wifiPassword);

while(WiFi.status()!=WL\_CONNECTED) {

delay(500);

Serial.print(".");

}

Serial.println("\nWi-Fi connected!");

pulseSensor.analogInput(pulsePin);

pulseSensor.setThreshold(threshold);

if(!pulseSensor.begin()) {

Serial.println("Pulse Sensor initialization failed!");

while(1);

}

Serial.println("Pulse Sensor initialized.");

registerWithTra();

}

void loop() {

if(pulseSensor.sawStartOfBeat()) {

int bpm=pulseSensor.getBeatsPerMinute();

Serial.print("Heartbeat detected! BPM: ");

Serial.println(bpm);

sendDataToServer(bpm);

}

delay(20);

}

void registerWithTra() {

if(WiFi.status()==WL\_CONNECTED) {

HTTPClient http;

http.begin(String(traUrl)+"/register");

http.addHeader("Content-Type", "application/json");

String payload="{";

payload+="\"entity\_id\": \""+String(entityId)+"\", ";

payload+="\"entity\_type\": \"pulse\_sensor\"";

payload+="}";

int httpResponseCode=http.POST(payload);

if(httpResponseCode==201) {

String response=http.getString();

DynamicJsonDocument doc(1024);

deserializeJson(doc, response);

sessionKey=doc["session\_key"].as<String>();

Serial.println("Successfully registered with TRA. SKEY: "+sessionKey);

} else {

Serial.println("TRA registration failed: "+String(httpResponseCode));

}

http.end();

} else {

Serial.println("Wi-Fi not connected!");

}

}

String generateHmac(String message) {

byte hmacResult[32];

mbedtls\_md\_context\_t ctx;

mbedtls\_md\_type\_t mdType=MBEDTLS\_MD\_SHA256;

const size\_t keyLength=sessionKey.length();

mbedtls\_md\_init(&ctx);

mbedtls\_md\_setup(&ctx, mbedtls\_md\_info\_from\_type(mdType), 1);

mbedtls\_md\_hmac\_starts(&ctx, (unsigned char\*)sessionKey.c\_str(), keyLength);

mbedtls\_md\_hmac\_update(&ctx, (unsigned char\*)message.c\_str(), message.length());

mbedtls\_md\_hmac\_finish(&ctx, hmacResult);

mbedtls\_md\_free(&ctx);

String hmacStr=(char \*)hmacResult;

return hmacStr;

}

void sendDataToServer(int bpm) {

if(WiFi.status()==WL\_CONNECTED) {

HTTPClient http;

http.begin(serverUrl);

http.addHeader("Content-Type", "application/json");

String nonce=String(random(0, 999999));

String hmac=generateHmac(nonce);

http.addHeader("Entity-ID", entityId);

http.addHeader("Nonce", nonce);

http.addHeader("HMAC", hmac);

String payload="{";

payload+="\"id\": \"sensor1\", ";

payload+="\"bpm\": "+String(bpm);

payload+="}";

int httpResponseCode=http.POST(payload);

if(httpResponseCode>0) {

String response=http.getString();

Serial.println("Server response: "+response);

} else {

Serial.println("Error in sending POST request: "+String(httpResponseCode));

}

http.end();

} else {

Serial.println("Wi-Fi not connected!");

}

}

-----------------------Algorithm1 Code (Model Training)----------------------

import numpy as np

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score, f1\_score

from sklearn.svm import SVC

from sklearn.tree import DecisionTreeClassifier

from sklearn.neighbors import KNeighborsClassifier

from sklearn.preprocessing import StandardScaler, OneHotEncoder

def normalizeaccs(accs):

totalacc = sum(accs)

return [a / totalacc for a in accs]

def trainevalmodels(T, V, k, catfeats, numfeats):

models = {

"svm": SVC(probability=True),

"knn": KNeighborsClassifier(n\_neighbors=k),

"dt": DecisionTreeClassifier()

}

accs = []

f1s = []

trainedmodels = []

T = T.drop\_duplicates()

V = V.drop\_duplicates()

onehotenc = OneHotEncoder(sparse\_output=False, handle\_unknown='ignore')

Tenc = onehotenc.fit\_transform(T[catfeats])

Venc = onehotenc.transform(V[catfeats])

scaler = StandardScaler()

Tscaled = scaler.fit\_transform(T[numfeats])

Vscaled = scaler.transform(V[numfeats])

Tpre = np.hstack((Tenc, Tscaled))

Vpre = np.hstack((Venc, Vscaled))

for name, model in models.items():

model.fit(Tpre, T["classnum"])

preds = model.predict(Vpre)

acc = accuracy\_score(V["classnum"], preds)

accs.append(acc)

f1 = f1\_score(V["classnum"], preds, average='weighted')

f1s.append(f1)

trainedmodels.append(model)

return trainedmodels, accs, f1s, onehotenc, scaler

def ensemblepred(models, weights, instance):

preds = [model.predict\_proba(instance) for model in models]

weightedpreds = sum(w \* p for w, p in zip(weights, preds))

return np.argmax(weightedpreds)

def genensemblemodel(D, k, thresh, catfeats, numfeats):

bestmodel = None

bestacc = 0

while True:

T, E = train\_test\_split(D, test\_size=0.3)

T, V = train\_test\_split(T, test\_size=0.2)

models, accs, f1s, onehotenc, scaler = trainevalmodels(T, V, k, catfeats, numfeats)

weights = normalizeaccs(accs)

Venc = onehotenc.transform(V[catfeats])

Vscaled = scaler.transform(V[numfeats])

Vpre = np.hstack((Venc, Vscaled))

ensemblepreds = V.apply(

lambda x: ensemblepred(models, weights, Vpre[V.index.get\_loc(x.name)].reshape(1, -1)), axis=1

)

ensembleacc = accuracy\_score(V["classnum"], ensemblepreds)

if ensembleacc > thresh:

bestmodel = (models, weights, onehotenc, scaler)

bestacc = ensembleacc

break

return bestmodel, bestacc, accs, f1s

if \_\_name\_\_ == "\_\_main\_\_":

D = pd.read\_csv("KDD.csv")

k = 5

thresh = 0.85

catfeats = ["protocol\_type", "service", "flag"]

numfeats = ["duration", "src\_bytes", "dst\_bytes", "land", "wrong\_fragment", "urgent", "hot",

"srv\_count", "serror\_rate", "srv\_serror\_rate", "rerror\_rate", "srv\_rerror\_rate", "same\_srv\_rate", "diff\_srv\_rate",

"srv\_diff\_host\_rate", "dst\_host\_count", "dst\_host\_srv\_count", "dst\_host\_same\_srv\_rate", "dst\_host\_diff\_srv\_rate",

"dst\_host\_same\_src\_port\_rate", "dst\_host\_serror\_rate", "dst\_host\_srv\_serror\_rate","dst\_host\_rerror\_rate", "dst\_host\_srv\_rerror\_rate"]

ensemblemodel, acc, indvaccu, indvf1s = genensemblemodel(D, k, thresh, catfeats, numfeats)

if ensemblemodel:

print("Ensemble model generated with accuracy: ",acc)

print("Individual model accuracies:")

print("SVM accuracy: ",indvaccu[0])

print("KNN accuracy: ",indvaccu[1])

print("DT accuracy: ",indvaccu[2])

print("Ensemble weights: ",ensemblemodel[1])

else:

print("Failed to generate a suitable ensemble model.")

------------------------Algorithm 2 (Intrusion Detection)--------------------------------

import numpy as np

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler, OneHotEncoder, LabelEncoder

from sklearn.decomposition import PCA

from sklearn.svm import SVC

from sklearn.tree import DecisionTreeClassifier

from sklearn.neighbors import KNeighborsClassifier

from sklearn.metrics import accuracy\_score, f1\_score

import matplotlib.pyplot as plt

def normalizeweights(scores):

total=sum(scores)

return [score/total for score in scores]

def trainandevaluatemodels(xtrain,xtest,ytrain,ytest,k):

models={

"svm":SVC(probability=True),

"knn":KNeighborsClassifier(n\_neighbors=k),

"dt":DecisionTreeClassifier()

}

accuracies=[]

f1scores=[]

trainedmodels=[]

for name,model in models.items():

model.fit(xtrain,ytrain)

predictions=model.predict(xtest)

acc=accuracy\_score(ytest,predictions)

f1=f1\_score(ytest,predictions,average='weighted')

accuracies.append(acc)

f1scores.append(f1)

trainedmodels.append(model)

return trainedmodels,accuracies,f1scores

def ensembleprediction(models,weights,instance):

predictions=[model.predict\_proba(instance) for model in models]

weightedpredictions=sum(weight\*prediction for weight,prediction in zip(weights,predictions))

return np.argmax(weightedpredictions)

def detectintrusions(datasetpath,k=5,threshold=0.85):

data=pd.read\_csv(datasetpath)

data=data.drop\_duplicates()

x=data.drop(columns=['classnum'])

y=data['classnum']

xtrain,xtest,ytrain,ytest=train\_test\_split(x,y,test\_size=0.3,random\_state=42)

scaler=StandardScaler()

xtrainscaled=scaler.fit\_transform(xtrain.select\_dtypes(include=[np.number]))

xtestscaled=scaler.transform(xtest.select\_dtypes(include=[np.number]))

onehotencoder=OneHotEncoder(sparse\_output=False,handle\_unknown='ignore')

xtrainencoded=onehotencoder.fit\_transform(xtrain.select\_dtypes(include=[object]))

xtestencoded=onehotencoder.transform(xtest.select\_dtypes(include=[object]))

xtrainpreprocessed=np.hstack((xtrainscaled,xtrainencoded))

xtestpreprocessed=np.hstack((xtestscaled,xtestencoded))

pca=PCA(n\_components=0.95)

xtrainpca=pca.fit\_transform(xtrainpreprocessed)

xtestpca=pca.transform(xtestpreprocessed)

models,accuracies,f1scores=trainandevaluatemodels(xtrainpca,xtestpca,ytrain,ytest,k)

weights=normalizeweights(accuracies)

ypredensemble=np.array([ensembleprediction(models,weights,instance.reshape(1,-1)) for instance in xtestpca])

ensembleaccuracy=accuracy\_score(ytest,ypredensemble)

ensemblef1score=f1\_score(ytest,ypredensemble,average='weighted')

detectedintrusions=ytest[ytest!=ypredensemble]

print("Individual model accuracies:")

for modelname,acc in zip(models,accuracies):

print(f"{modelname}: {acc}")

print("Ensemble weights:",weights)

print("Ensemble model accuracy:", ensembleaccuracy)

print("Ensemble model f1 score:",ensemblef1score)

plt.figure(figsize=(10, 5))

plt.bar(["SVM", "KNN", "DT"], [acc \* 100 for acc in accuracies], color=['blue', 'green', 'red'])

plt.xlabel("Models")

plt.ylabel("Accuracy (%)")

plt.title("Accuracy of Different Models")

plt.show()

plt.figure(figsize=(10, 5))

plt.bar(["SVM", "KNN", "DT"], [f1 \* 100 for f1 in f1scores], color=['blue', 'green', 'red'])

plt.xlabel("Models")

plt.ylabel("F1 Score (%)")

plt.title("F1 Score of Different Models")

plt.show()

return detectedintrusions

if \_\_name\_\_ == "\_\_main\_\_":

detected\_intrusions = detectintrusions("KDD.csv")

print("Detected intrusions:")

print(detected\_intrusions)