# Assemble datasets

def assemble\_datasets(file\_paths, label\_extractor):

tables = []

for path in file\_paths:

data = pd.read\_csv(path, sep='|', low\_memory=False)

data.columns = data.columns.str.strip()

if 'label' not in data.columns:

data['label'] = label\_extractor(path)

tables.append(data)

return pd.concat(tables, ignore\_index=True)

# Clean and split data

def clean\_and\_split\_data(data, columns\_to\_remove, metadata\_fields):

data['label'] = (

data['label'].astype(str)

.str.strip().str.lower()

.apply(lambda x: 'other' if 'attack' in x or 'c&c' in x else x)

)

data = data[data['label'] != 'other']

metadata = data[metadata\_fields].copy()

data = data.drop(columns=columns\_to\_remove, errors='ignore')

data = data.rename(columns={'label': 'Label'})

encoder = LabelEncoder()

data['Label'] = encoder.fit\_transform(data['Label'])

data = data.select\_dtypes(include=[np.number]).fillna(0)

features = data.drop('Label', axis=1)

labels = data['Label']

return features, labels, metadata, encoder.classes\_

# Sample and oversample rare classes

def sample\_and\_oversample(features, labels, max\_samples, rare\_cutoff, smote\_neighbors):

subset = pd.concat([features, labels], axis=1).sample(min(len(labels), max\_samples), random\_state=42)

X\_sub = subset.drop('Label', axis=1)

y\_sub = subset['Label']

counts = y\_sub.value\_counts()

rare\_classes = counts[counts < rare\_cutoff].index.tolist()

X\_bal, y\_bal = X\_sub.copy(), y\_sub.copy()

for cls in rare\_classes:

idx = y\_sub[y\_sub == cls].index

if len(idx) >= smote\_neighbors + 1:

smoter = SMOTE(random\_state=42, k\_neighbors=smote\_neighbors)

X\_extra, y\_extra = smoter.fit\_resample(X\_sub.loc[idx], y\_sub.loc[idx])

X\_bal = pd.concat([X\_bal, pd.DataFrame(X\_extra, columns=X\_sub.columns)], ignore\_index=True)

y\_bal = pd.concat([y\_bal, pd.Series(y\_extra)], ignore\_index=True)

return X\_bal, y\_bal

# Fit hybrid model

def fit\_hybrid\_model(train\_features, train\_labels, meta\_features, meta\_labels, weights):

base\_model = CatBoostClassifier(verbose=0, class\_weights=weights, random\_seed=42)

base\_model.fit(train\_features, train\_labels)

prob\_inputs = base\_model.predict\_proba(meta\_features)

resolver\_model = LogisticRegression(max\_iter=200)

resolver\_model.fit(prob\_inputs, meta\_labels)

return base\_model, resolver\_model

# Hybrid predict

def hybrid\_predict(base\_model, resolver\_model, test\_data, conf\_limit=0.6):

base\_probs = base\_model.predict\_proba(test\_data)

hybrid\_probs = resolver\_model.predict\_proba(base\_probs)

preds = resolver\_model.predict(base\_probs)

scores = np.max(hybrid\_probs, axis=1)

preds[scores < conf\_limit] = -1

valid = preds != -1

return preds, hybrid\_probs, valid

# Generate metrics table

def generate\_metrics\_table(actual\_labels, predicted\_labels, class\_labels):

report = classification\_report(actual\_labels, predicted\_labels, output\_dict=True, target\_names=class\_labels)

df\_rep = pd.DataFrame(report).T

for metric in ['precision', 'recall', 'f1-score']:

df\_rep[metric] = (df\_rep[metric] \* 100).round(2)

df\_rep['support'] = df\_rep['support'].astype(int)

return df\_rep

# Get confusion matrix

def get\_confusion\_matrix(actual\_labels, predicted\_labels):

return confusion\_matrix(actual\_labels, predicted\_labels)

# Calculate ROC metrics

def calculate\_roc\_metrics(actual\_labels, probabilities, class\_labels):

roc\_map = {}

for i, name in enumerate(class\_labels):

fpr, tpr, \_ = roc\_curve((actual\_labels == i).astype(int), probabilities[:, i])

roc\_map[name] = (fpr, tpr, auc(fpr, tpr))

return roc\_map

# Submit to blockchain

def submit\_to\_blockchain(web3\_instance, contract\_instance, account, metadata\_df, indices, scores, record\_limit=10):

transactions = []

for idx, rec\_idx in enumerate(indices[:record\_limit], start=1):

entry = (

f"UID:{metadata\_df.loc[rec\_idx,'uid']}|"

f"Service:{metadata\_df.loc[rec\_idx,'service']}|"

f"Score:{scores[idx-1]:.4f}"

)

tx\_hash = web3\_instance.keccak(text=entry)

tx = contract\_instance.functions.addRecords([tx\_hash]).transact({'from': account})

receipt = web3\_instance.eth.wait\_for\_transaction\_receipt(tx)

transactions.append({

'position': idx,

'hash': receipt.transactionHash.hex(),

'block': receipt.blockNumber,

'gas': receipt.gasUsed

})

return transactions

# Create chain graph

def create\_chain\_graph(transactions):

G = nx.DiGraph()

for tx in transactions:

G.add\_node(tx['position'])

for tx in transactions[1:]:

G.add\_edge(tx['position'] - 1, tx['position'])

return G