Detailed Step-by-Step Guide: Phase 1 - Sybil Detection with TFF for ChainFLIP

This guide provides highly detailed instructions for implementing Phase 1 (Sybil Detection using TFF) of the Federated Learning integration for your ChainFLIP project. It specifies the directory structure, file names, and exact code placement.

I. Project Setup and Directory Structure

1. **Create a Project Directory:** Open your terminal or command prompt. Navigate to where you want to create your project and make a new directory. Let's call it `ChainFLIP_FL_Dev`. `bash mkdir ChainFLIP_FL_Dev cd ChainFLIP_FL_Dev`

2. **Set Up Python Virtual Environment (as per previous guide):** Inside `ChainFLIP_FL_Dev`, create and activate your Python virtual environment (e.g., `tff_env`). ```bash # Example for Linux/macOS python3 -m venv tff_env source tff_env/bin/activate

   # Example for Windows Command Prompt

   # python -m venv tff_env

   # tff_env\Scripts\activate.bat

   ```

3. **Install Required Packages (inside the activated environment):** `bash pip install --quiet --upgrade tensorflow-federated pip install --quiet --upgrade nest-asyncio pip install pandas scikit-learn tensorflow`

4. **Create Subdirectory for Phase 1 Code:** Inside `ChainFLIP_FL_Dev`, create a subdirectory for the Sybil detection code. `bash mkdir tff_sybil_detection cd`
   ```

tff_sybil_detection Your current directory in the terminal should now be ChainFLIP_FL_Dev/tff_sybil_detection/ .

II. Creating Python Files and Adding Code

We will create several Python files within the tff_sybil_detection directory.

File 1: data_preparation.py

- **Purpose:** This file will contain functions to simulate loading and preprocessing client data for TFF.
- **Location:** ChainFLIP_FL_Dev/tff_sybil_detection/data_preparation.py
- **Content:**

```
 import tensorflow as tf
import numpy as np
# from sklearn.preprocessing import StandardScaler # Optional: if you use it

# Define an OrderDict for feature specification for TFF
# This describes the structure of a single data point (features, label)
# Features: (num_features,), Label: (1,) for binary classification
ELEMENT_SPEC = (
    tf.TensorSpec(shape=(5,), dtype=tf.float32), # Features (5 in this example)
    tf.TensorSpec(shape=(1,), dtype=tf.int32)   # Labels (single label for binary
classification)
)

NUM_FEATURES = 5 # Define this globally for consistency

def load_local_data_for_client(client_id: str, num_samples=100):
    """Simulates loading and preprocessing data for a single client.

    In a real scenario, this function would:
    1. Connect to the blockchain (e.g., using web3.py).
    2. Query NodeManagement.sol for node registration details (timestamp, role, type).
    3. Query transaction history for initial activity of nodes this client interacts with.
    4. Extract features: e.g., registration_age_days, transaction_frequency,
diversity_of_interactions.
       (Refer to fl_data_sources_explanation.md for more details on feature sources).
    5. Create labels: e.g., 0 for normal, 1 for suspicious (initially, this might be manually
labeled or based on heuristics).
    """
    print(f"Client {client_id}: Simulating local data loading and preprocessing...")
    # Example: num_samples, NUM_FEATURES features
    X_local = np.random.rand(num_samples, NUM_FEATURES).astype(np.float32)
    # Labels should be shape (num_samples, 1) for binary crossentropy with
from_logits=False
    y_local = np.random.randint(0, 2, size=(num_samples, 1)).astype(np.int32)
```

```
    # Simulate some client-specific data variation (non-IID)
    if client_id == "client_1":
        print(f"Client {client_id}: Introducing data variation.")
        X_local[:num_samples//2, 0] += 0.7 # Make some features different for client 1
        y_local[:num_samples//2] = 1      # More suspicious samples for client 1
    elif client_id == "client_2":
        print(f"Client {client_id}: Introducing data variation.")
        X_local[num_samples//4:num_samples//2, 1] -= 0.5
        y_local[num_samples//4:num_samples//2] = 0

    # Optional: Feature Scaling (StandardScaler example)
    # scaler = StandardScaler()
    # X_local = scaler.fit_transform(X_local) # Fit scaler on training data only in real
scenario

    # Create tf.data.Dataset from the client's data
    # TFF expects datasets to yield batches. Here, we make each client's full data a single
batch for simplicity.
    # In practice, you would use .batch(BATCH_SIZE) on the dataset *before* federated
processing.
    # The dataset should yield tuples matching ELEMENT_SPEC (features, labels)
    dataset = tf.data.Dataset.from_tensor_slices((X_local, y_local))
    # For TFF, it's often better to batch within the TFF computation or prepare client
datasets to be pre-batched.
    # For this example, we'll return the unbatched dataset and batch it later if needed by
the TFF process.
    return dataset

def make_federated_data(client_ids: list[str], num_samples_per_client=100):
    """Creates a list of tf.data.Dataset objects for TFF simulation."""
    return [load_local_data_for_client(client_id, num_samples_per_client) for
client_id in client_ids]

if __name__ == '__main__':
    # Test the data preparation
    print("Testing data preparation...")
    CLIENT_IDS_TEST = ["client_0", "client_1", "client_2"]
    federated_train_data_test = make_federated_data(CLIENT_IDS_TEST)

    print(f"\nCreated {len(federated_train_data_test)} client datasets.")
    for i, client_dataset in enumerate(federated_train_data_test):
        print(f"Client {CLIENT_IDS_TEST[i]} dataset element spec:
{client_dataset.element_spec}")
        # Take one element (which is a batch of all samples for this client in this setup)
        for features, labels in client_dataset.take(1):
            print(f"  Features shape: {features.shape}, Labels shape: {labels.shape}")
            print(f"  First feature vector: {features.numpy()[0]}")
            print(f"  First label: {labels.numpy()[0]}")
        # Verify it matches ELEMENT_SPEC
        assert client_dataset.element_spec[0].shape.as_list() == [NUM_FEATURES],
f"Feature spec mismatch for client {i}"
```

```
        assert client_dataset.element_spec[1].shape.as_list() == [1], f"Label spec
mismatch for client {i}"

    print("\nData preparation test complete.")
```

File 2: `model_definition.py`

- **Purpose:** Defines the Keras model and wraps it for TFF.
- **Location:** `ChainFLIP_FL_Dev/tff_sybil_detection/model_definition.py`
- **Content:**

```
 import tensorflow as tf
import tensorflow_federated as tff
from data_preparation import NUM_FEATURES, ELEMENT_SPEC
# Import from our data_preparation module

def create_keras_model():
    """Creates a simple Keras model for binary classification."""
    model = tf.keras.models.Sequential([
        tf.keras.layers.Dense(
            units=16, # Increased units
            activation= f.nn.relu,
            kernel_initializer= f.keras.initializers.GlorotUniform(), # Good default
            input_shape=(NUM_FEATURES,)
        ),
        tf.keras.layers.Dropout(0.3), # Added dropout for regularization
        tf.keras.layers.Dense(
            units=8,
            activation=tf.nn.relu,
            kernel_initializer=tf.keras.initializers.GlorotUniform()
        ),
        tf.keras.layers.Dense(
            units=1,
            activation=tf.nn.sigmoid # Sigmoid for binary classification probability
        )
    ])
    return model

def tff_model_fn():
    """Wraps the Keras model for use with TFF.
    Returns a tff.learning.models.VariableModel.
    """
    keras_model_instance = create_keras_model()
    # The input_spec for from_keras_model should be the spec of a single data point (not
a batch)
    # It should match the feature part of ELEMENT_SPEC
    return tff.learning.models.from_keras_model(
        keras_model_instance,
        input_spec=ELEMENT_SPEC[0], # Feature spec from data_preparation
        loss=tf.keras.losses.BinaryCrossentropy(), # Standard loss for binary
```

```
classification
        metrics=[tf.keras.metrics.BinaryAccuracy(name=\"accuracy\"),
tf.keras.metrics.AUC(name=\"auc\")]
    )

if __name__ == '__main__':
    # Test model creation and TFF wrapping
    print("Testing Keras model creation...")
    keras_m = create_keras_model()
    keras_m.summary()

    print("\nTesting TFF model function...")
    tff_m_fn = tff_model_fn()

# You can inspect properties of the TFF model if needed, but it's a callable, not a direct
model instance.
    print("TFF model function created successfully.")
    # Example: Create a concrete TFF model (not usually done directly like this for
training)
    # state_manager = tff.learning.models.ModelWeights.get_model_weights(tff_m_fn())
    # print(f"Model weights structure: {state_manager}")
```

File 3: `federated_training.py`

- **Purpose:** Defines the federated training process (e.g., FedAvg).
- **Location:** `ChainFLIP_FL_Dev/tff_sybil_detection/federated_training.py`
- **Content:**

```
 import tensorflow_federated as tff
import tensorflow as tf
from model_definition import tff_model_fn # Import from our model_definition
module

def build_fed_avg_process():
    """Builds the Federated Averaging iterative process."""
    # Client optimizer: Adam is often a good choice
    client_optimizer = lambda: tf.keras.optimizers.Adam(learning_rate=0.001)
    # Server optimizer: SGD is common for FedAvg, LR of 1.0 is typical if server model is
directly updated with avg weights
    server_optimizer = lambda: tf.keras.optimizers.SGD(learning_rate=1.0)

    fed_avg_process = tff.learning.algorithms.build_weighted_fed_avg(
        tff_model_fn, # The TFF model function
        client_optimizer_fn=client_optimizer,
        server_optimizer_fn=server_optimizer,
        # model_aggregator=tff.learning.robust_aggregator(), # Example: for more robust
aggregation
        # metrics_aggregator=tff.learning.metrics.sum_then_finalize, # Default metrics
aggregation
    )
```

```python
    return fed_avg_process

if __name__ == '__main__':
    print("Building Federated Averaging process...")
    iterative_process = build_fed_avg_process()
    print("Federated Averaging process built successfully.")
    print("Initialize signature:", iterative_process.initialize.type_signature)
    print("Next signature:", iterative_process.next.type_signature)
```

File 4: `run_simulation.py`

- **Purpose:** Main script to run the FL simulation (initialize, train, evaluate).
- **Location:** `ChainFLIP_FL_Dev/tff_sybil_detection/run_simulation.py`
- **Content:**

```python
 import tensorflow_federated as tff
import tensorflow as tf
import nest_asyncio

from data_preparation import make_federated_data, ELEMENT_SPEC,
NUM_FEATURES
from federated_training import build_fed_avg_process
from model_definition import create_keras_model # For loading final weights

# Apply nest_asyncio to allow TFF to run in environments like Jupyter or scripts easily.
nest_asyncio.apply()

def main():
    print("Starting Federated Learning Simulation for Sybil Detection...")

    # 1. Data Preparation
    NUM_CLIENTS_SIMULATION = 3
    CLIENT_IDS_SIMULATION = [f"sim_client_{i}" for i in
range(NUM_CLIENTS_SIMULATION)]
    print(f"Preparing data for {NUM_CLIENTS_SIMULATION} clients...")
    # Each element in federated_train_data is a tf.data.Dataset for one client
    # These datasets should be preprocessed and batched appropriately for the model_fn
    # For TFF's from_keras_model, the dataset should yield (features, labels) tuples
    # where features and labels are for ONE batch.
    # Our load_local_data_for_client currently returns a dataset of individual examples.
    # We need to batch them before passing to iterative_process.next

    BATCH_SIZE = 32 # Define a batch size for client datasets

    raw_client_datasets = make_federated_data(CLIENT_IDS_SIMULATION,
num_samples_per_client=200)

    def preprocess_client_dataset(dataset):
        # Shuffle and batch the client's dataset
        return dataset.shuffle(buffer_size=100).batch(BATCH_SIZE)
```

```python
    federated_train_datasets = [preprocess_client_dataset(ds) for ds in raw_client_datasets]
    print("Client datasets prepared and batched.")

    # 2. Build Federated Training Process
    print("Building the federated training process (FedAvg)...")
    iterative_process = build_fed_avg_process()
    print("Federated training process built.")

    # 3. Initialize the Process
    print("Initializing the iterative process...")
    server_state = iterative_process.initialize()
    print("Initialization complete.")

    # 4. Run Federated Training Rounds
    NUM_ROUNDS = 10 # More rounds for better convergence
    print(f"Starting {NUM_ROUNDS} rounds of federated training...")

    for round_num in range(1, NUM_ROUNDS + 1):
        # Select a subset of clients for the round (here, using all for simplicity)
        # In a real system, client_data would be sampled from available clients.
        # The structure of client_data for iterative_process.next should be a list of client datasets.
        result = iterative_process.next(server_state, federated_train_datasets) # Pass the list of datasets
        server_state = result.state
        metrics = result.metrics
        # The metrics structure depends on what's aggregated.
        # For FedAvg with Keras model, it's usually under 'client_work' then 'train'.
        round_loss = metrics['client_work']['train']['loss']
        round_accuracy = metrics['client_work']['train']['accuracy'] # if accuracy is a metric
        print(f"Round {round_num:2d}: loss={round_loss:.4f}, accuracy={round_accuracy:.4f}")

    print("Federated training completed.")

    # 5. Extract and Use the Global Model (Example)
    print("Extracting final global model weights...")
    model_weights = iterative_process.get_model_weights(server_state)

    # Create a new Keras model instance and assign the learned weights
    final_keras_model = create_keras_model() # from model_definition.py
    final_keras_model.compile(
        optimizer=tf.keras.optimizers.Adam(), # Not strictly needed for inference but good practice
        loss=tf.keras.losses.BinaryCrossentropy(),
        metrics=[tf.keras.metrics.BinaryAccuracy(name="accuracy"), tf.keras.metrics.AUC(name="auc")]
    )
```

```python
    # Assign weights to the Keras model
    model_weights.assign_weights_to(final_keras_model)
    print("Final global model weights assigned to a new Keras model instance.")

    # Example: Evaluate the global model on some test data (simulated here)
    # In a real scenario, you'd have a separate federated_eval_data or a centralized test set.
    print("Simulating evaluation of the global model...")
    eval_data_X = np.random.rand(50, NUM_FEATURES).astype(np.float32)
    eval_data_y = np.random.randint(0, 2, size=(50, 1)).astype(np.int32)
    eval_results = final_keras_model.evaluate(eval_data_X, eval_data_y, verbose=0)
    print(f"Global model evaluation on simulated test data - Loss: {eval_results[0]:.4f}, Accuracy: {eval_results[1]:.4f}, AUC: {eval_results[2]:.4f}")

    # Here, you would save final_keras_model or its weights for the Admin Dashboard Prediction Service
    # final_keras_model.save("global_sybil_detection_model.h5")
    # print("Global Keras model saved to global_sybil_detection_model.h5")

    print("\nSimulation finished.")

if __name__ == '__main__':
    main()
```

III. Running the Phase 1 Simulation

1. **Navigate to the Main Project Directory:** Open your terminal (with the `tff_env` virtual environment activated) and make sure you are in the `ChainFLIP_FL_Dev` directory (the one above `tff_sybil_detection`).

2. **Run the Simulation Script:** Execute the `run_simulation.py` script using Python. `bash python tff_sybil_detection/run_simulation.py`

Expected Output:

You should see output indicating: * Data preparation for each client. * Building of the federated training process. * Initialization of the process. * Metrics (loss, accuracy) for each training round. * Extraction of final model weights and simulated evaluation.

IV. Next Steps (Conceptual - Same as Previous Guide)

- **Real Data Integration:** Replace the simulated `load_local_data_for_client` in `data_preparation.py` with actual logic to fetch and process data from the ChainFLIP blockchain and local client sources.

- **Feature Engineering:** Implement robust feature engineering for Sybil detection based on `fl_data_sources_explanation.md`.
- **Admin Dashboard Integration:** Develop the prediction service and admin dashboard to use the trained global model for risk scoring.
- **Smart Contract Adjustments:** Implement any necessary additions to `NodeManagement.sol` (e.g., `getRegistrationTimestamp`, `isFlaggedByFL` status).

This detailed guide should help you set up and run the Phase 1 Sybil detection using TFF. Remember that the data simulation is a placeholder; the core of a successful FL system lies in meaningful feature engineering from real data sources.