Phase 2 TFF: Simulation Data and Testing Guide for Batch Processing Monitoring

This guide provides details on generating simulation data and testing the Phase 2 Federated Learning (FL) system for Batch Processing Monitoring in your ChainFLIP project. It builds upon the Phase 2 design document and the TFF framework established in Phase 1.

1. Recap of Phase 2 Goals and Features

Goal: Detect collusion among validators/proposers and other anomalous behaviors in the `BatchProcessing.sol` mechanism.

Key Features (Conceptual - to be simulated):

- **For Validators:** `validator_vote_consistency_rate`, `validator_agreement_with_majority_rate`, `validator_reputation_change_per_batch`, `validator_solo_dissent_rate`, `validator_approval_rate_for_low_rep_proposers`.
- **For Proposers:** `proposer_batch_success_rate`, `proposer_batch_flagged_rate`, `proposer_avg_validators_approving`, `proposer_reputation_change_per_proposal`.

Labels: Initially, we might aim for an anomaly detection setup (0 for normal, 1 for anomalous/collusive) or a multi-class setup if distinct malicious patterns are defined.

2. Setting up for Phase 2 Simulation

It's recommended to create a new subdirectory for Phase 2 to keep things organized, or carefully adapt the Phase 1 files.

Option A: New Directory (Recommended for clarity)

1. Inside `ChainFLIP_FL_Dev/`, create `tff_batch_monitoring/`.
2. Copy `model_definition.py`, `federated_training.py`, and `run_simulation.py` from `tff_sybil_detection/` into `tff_batch_monitoring/`.
3. Create a new `data_preparation_phase2.py` in `tff_batch_monitoring/`.

Option B: Adapt Phase 1 Files (More complex to manage) You would add conditional logic or new functions within the existing Phase 1 files.

For this guide, we assume **Option A** and will create `data_preparation_phase2.py`.

3. Generating Controlled Synthetic Data for Phase 2

We need to simulate batch processing events and derive features from them. This is more complex than Phase 1 data generation.

File: `ChainFLIP_FL_Dev/tff_batch_monitoring/data_preparation_phase2.py`

```
import tensorflow as tf
import numpy as np
import pandas as pd # For potential CSV handling later
import random

# Define the number of features for Phase 2. This needs to match your feature engineering.
# Let's assume 6 features for validators and 5 for proposers for this example.
# For simplicity in a unified model, we might pad or select a common number, e.g., 6.
NUM_PHASE2_FEATURES = 6

ELEMENT_SPEC_PHASE2 = (
    tf.TensorSpec(shape=(NUM_PHASE2_FEATURES,), dtype=tf.float32),
    tf.TensorSpec(shape=(1,), dtype=tf.int32)  # Label: 0 for normal, 1 for anomalous/collusive
)

# --- Simulation of Batch Processing Environment ---
NUM_SIM_VALIDATORS = 10
NUM_SIM_PROPOSERS = 5
SIM_VALIDATOR_IDS = [f"val_{i}" for i in range(NUM_SIM_VALIDATORS)]
SIM_PROPOSER_IDS = [f"prop_{i}" for i in range(NUM_SIM_PROPOSERS)]

# Define some behavioral profiles for simulation
BEHAVIOR_PROFILES = {
    "normal_validator": {"collusion_tendency": 0.1, "error_rate": 0.05, "label": 0},
    "collusive_validator": {"collusion_tendency": 0.9, "error_rate": 0.05, "label": 1},
    "faulty_validator": {"collusion_tendency": 0.1, "error_rate": 0.5, "label": 1},
    "normal_proposer": {"success_bias": 0.8, "low_quality_rate": 0.1, "label": 0},
    "bad_proposer": {"success_bias": 0.2, "low_quality_rate": 0.7, "label": 1}
}

# Assign profiles to simulated nodes (can be more dynamic)
NODE_PROFILES = {}
for i, vid in enumerate(SIM_VALIDATOR_IDS):
    if i < NUM_SIM_VALIDATORS // 2: # Half normal
        NODE_PROFILES[vid] = BEHAVIOR_PROFILES["normal_validator"]
```

```
    elif i < NUM_SIM_VALIDATORS * 0.8: # Some collusive
        NODE_PROFILES[vid] = BEHAVIOR_PROFILES["collusive_validator"]
    else: # Some faulty
        NODE_PROFILES[vid] = BEHAVIOR_PROFILES["faulty_validator"]

for i, pid in enumerate(SIM_PROPOSER_IDS):
    if i < NUM_SIM_PROPOSERS // 2:
        NODE_PROFILES[pid] = BEHAVIOR_PROFILES["normal_proposer"]
    else:
        NODE_PROFILES[pid] = BEHAVIOR_PROFILES["bad_proposer"]

COLLUSION_GROUP_A = [SIM_VALIDATOR_IDS[i] for i in
range(NUM_SIM_VALIDATORS // 2, int(NUM_SIM_VALIDATORS * 0.8))]

def simulate_batch_events(num_batches=100):
    """Simulates a series of batch proposal and validation events."""
    batch_log = [] # Store (proposer, selected_validators, votes, outcome)
    for batch_id in range(num_batches):
        proposer = random.choice(SIM_PROPOSER_IDS)
        # Simulate proposer quality for this batch
        is_low_quality_batch = random.random() < NODE_PROFILES[proposer]
["low_quality_rate"]

        num_selected_validators = min(NUM_SIM_VALIDATORS, max(3,
NUM_SIM_VALIDATORS // 2))
        selected_validators = random.sample(SIM_VALIDATOR_IDS,
num_selected_validators)

        votes = {} # validator_id: vote (True for approve, False for reject)
        num_approvals = 0
        for validator_id in selected_validators:
            profile = NODE_PROFILES[validator_id]
            # Collusive behavior: if proposer is "bad_proposer" and validator is
"collusive_validator"
            # or if validator is in a specific collusion group and proposer is part of their
scheme (not modeled here explicitly)
            vote_approve = True
            if validator_id in COLLUSION_GROUP_A and proposer ==
SIM_PROPOSER_IDS[-1]: # Collude for last proposer
                 vote_approve = random.random() < profile["collusion_tendency"]
            elif is_low_quality_batch: # Tend to reject low quality unless colluding or faulty
                vote_approve = random.random() < (profile["collusion_tendency"] * 0.5 +
profile["error_rate"])
            else: # Tend to approve high quality unless faulty
                vote_approve = random.random() > profile["error_rate"]

            votes[validator_id] = vote_approve
            if vote_approve: num_approvals +=1

        # Simplified outcome: requires >50% approval (superMajority not fully modeled
here)
        batch_outcome_committed = num_approvals > num_selected_validators / 2
```

```
        batch_log.append({
            "batch_id": batch_id, "proposer": proposer, "is_low_quality":
is_low_quality_batch,
            "selected_validators": selected_validators, "votes": votes, "committed":
batch_outcome_committed
        })
    return batch_log

def extract_features_from_log(batch_log, target_node_id):
    """Extracts features for a specific node from the batch log."""
    # This is a simplified feature extraction. Real one would be more complex.
    node_batches = [b for b in batch_log if target_node_id == b["proposer"] or
target_node_id in b["selected_validators"]]
    if not node_batches: return None # Node didn't participate

    label = NODE_PROFILES[target_node_id]["label"]
    features = np.zeros(NUM_PHASE2_FEATURES, dtype=np.float32)

    if target_node_id.startswith("val_"): # Validator features
        participated = 0; consistent_votes = 0; agreed_majority = 0; solo_dissents = 0
        for batch in node_batches:
            if target_node_id not in batch["selected_validators"]: continue
            participated += 1
            my_vote = batch["votes"][target_node_id]
            if my_vote == batch["committed"]: consistent_votes += 1

            num_approvals = sum(batch["votes"].values())
            majority_vote_approves = num_approvals >
len(batch["selected_validators"]) / 2
            if my_vote == majority_vote_approves: agreed_majority +=1
            if len(batch["selected_validators"]) > 1 and my_vote !=
majority_vote_approves and
               ( (my_vote and num_approvals == 1) or (not my_vote and num_approvals
== len(batch["selected_validators"])-1) ):
                solo_dissents +=1

        features[0] = consistent_votes / participated if participated else 0
        features[1] = agreed_majority / participated if participated else 0
        features[2] = solo_dissents / participated if participated else 0
        features[3] = participated / len(batch_log) # Participation rate
        # features 4, 5 can be other validator metrics like approval for low_rep proposers
(needs proposer rep)

    elif target_node_id.startswith("prop_"): # Proposer features
        proposed = 0; succeeded = 0; flagged = 0
        for batch in node_batches:
            if target_node_id != batch["proposer"]: continue
            proposed += 1
            if batch["committed"]: succeeded += 1
            else: flagged +=1
        features[0] = succeeded / proposed if proposed else 0
        features[1] = flagged / proposed if proposed else 0
```

```
        features[2] = proposed / len(batch_log) # Proposal rate
        # features 3, 4 can be other proposer metrics

    return features, np.array([label], dtype=np.int32)


GLOBAL_BATCH_LOG = simulate_batch_events(num_batches=500) # Generate a
global log once

def load_local_data_for_phase2_client(client_id: str, node_ids_for_client: list[str]):
    """Generates client dataset by extracting features for its assigned nodes from the
global log."""
    print(f"Client {client_id}: Extracting features for nodes: {node_ids_for_client}")
    client_features = []
    client_labels = []
    for node_id in node_ids_for_client:
        result = extract_features_from_log(GLOBAL_BATCH_LOG, node_id)
        if result:
            features, label = result
            client_features.append(features)
            client_labels.append(label)

    if not client_features:
        # Create dummy data if no features could be extracted to avoid TFF errors
        # This should ideally not happen with good node assignment
        print(f"Warning: Client {client_id} had no data. Creating dummy data.")
        dummy_features = np.zeros((1, NUM_PHASE2_FEATURES), dtype=np.float32)
        dummy_labels = np.array([[0]], dtype=np.int32)
        return tf.data.Dataset.from_tensor_slices((dummy_features, dummy_labels))

    return tf.data.Dataset.from_tensor_slices((np.array(client_features),
np.array(client_labels)))

def make_federated_data_phase2(num_fl_clients=3):
    """Creates federated datasets. Each FL client gets a subset of all simulated nodes."""
    all_sim_nodes = SIM_VALIDATOR_IDS + SIM_PROPOSER_IDS
    random.shuffle(all_sim_nodes)

    nodes_per_fl_client = len(all_sim_nodes) // num_fl_clients
    client_datasets = []
    for i in range(num_fl_clients):
        start_idx = i * nodes_per_fl_client
        end_idx = (i + 1) * nodes_per_fl_client if i < num_fl_clients - 1 else
len(all_sim_nodes)
        fl_client_id = f"fl_client_{i}"
        assigned_nodes = all_sim_nodes[start_idx:end_idx]
        if not assigned_nodes:
            print(f"Warning: FL Client {fl_client_id} assigned no nodes. Skipping.")
            continue
        client_datasets.append(load_local_data_for_phase2_client(fl_client_id,
assigned_nodes))
    return client_datasets
```

```python
if __name__ == '__main__':
    print("Testing Phase 2 Data Preparation...")
    federated_data = make_federated_data_phase2(num_fl_clients=3)
    print(f"\nCreated {len(federated_data)} FL client datasets for Phase 2.")
    for i, ds in enumerate(federated_data):
        print(f"FL Client {i} dataset element spec: {ds.element_spec}")
        num_elements = 0
        for features, label in ds:
            num_elements += 1
            # print(f"  Features: {features.numpy()}, Label: {label.numpy()}")
        print(f"  Client {i} has {num_elements} data points (nodes).")
        if num_elements > 0:
             assert ds.element_spec[0].shape.as_list() == [NUM_PHASE2_FEATURES],
f"Feature spec mismatch for client {i}"
             assert ds.element_spec[1].shape.as_list() == [1], f"Label spec mismatch for
client {i}"
    print("Phase 2 Data Preparation Test Complete.")
```

Key aspects of this `data_preparation_phase2.py`: * Simulates Batch Events: `simulate_batch_events` creates a log of proposals, validator selections, votes, and outcomes. * **Behavioral Profiles:** `BEHAVIOR_PROFILES` and `NODE_PROFILES` define how different simulated validators/proposers behave (e.g., collusive, faulty, bad proposer). * **Feature Extraction:** `extract_features_from_log` calculates features for a given node based on its activity in the simulated log. **This is a crucial part you'll need to expand and refine with more sophisticated features. * FL Client Data:** `load_local_data_for_phase2_client` creates a `tf.data.Dataset` for an FL client, containing features for a subset of all simulated nodes (validators/proposers). * **Global Log:** `GLOBAL_BATCH_LOG` is generated once; all clients derive their local views from this shared history for simulation consistency.

4. Adapting TFF Scripts for Phase 2

1. **`model_definition.py` (in `tff_batch_monitoring/`)**
    - Update `NUM_FEATURES` to `NUM_PHASE2_FEATURES` (e.g., 6).
    - Ensure `ELEMENT_SPEC` imported or defined matches `ELEMENT_SPEC_PHASE2`.
    - The Keras model's input layer `input_shape=(NUM_PHASE2_FEATURES,)` must be updated.
    - The rest of the model architecture can be similar to Phase 1 initially, or you can make it more complex.

2. **`federated_training.py` (in `tff_batch_monitoring/`)**

   - This script likely needs minimal changes if it correctly imports `tff_model_fn` from your adapted `model_definition.py`.

3. **`run_simulation.py` (in `tff_batch_monitoring/`)**

   - Change imports: `from data_preparation_phase2 import make_federated_data_phase2, ELEMENT_SPEC_PHASE2, NUM_PHASE2_FEATURES`
   - Use `make_federated_data_phase2` to generate client datasets.
   - Update `NUM_FEATURES` variable if used directly.
   - The `preprocess_client_dataset` function (shuffling, batching) should still be applied to the datasets returned by `make_federated_data_phase2`.
   - The interpretation of metrics (loss, accuracy, AUC) will now relate to how well the model identifies anomalous/collusive behavior in batch processing.

5. Running and Testing Phase 2 Simulation

1. Ensure all files are in `ChainFLIP_FL_Dev/tff_batch_monitoring/`.
2. Activate your Python virtual environment.
3. Run the main simulation: `python tff_batch_monitoring/run_simulation.py`

Interpreting Results: * Look for the model to learn to distinguish between the `label=0` (normal) and `label=1` (anomalous/collusive) nodes based on the simulated behavioral features. * High accuracy/AUC would indicate the model is successfully identifying these patterns. * Experiment with: * The number of simulated batches in `simulate_batch_events`. * The complexity of `BEHAVIOR_PROFILES` and collusion logic. * The feature extraction logic in `extract_features_from_log` (this is key!). * The TFF model architecture and hyperparameters.

6. Next Steps for Phase 2

- **Refine Feature Engineering:** The `extract_features_from_log` is very basic. You need to implement more robust features as outlined in the Phase 2 design document.
- **Real Data Hooks:** Plan how each FL client (organization) would actually query the ChainFLIP blockchain for the real event logs and state needed for feature extraction.
- **Admin Dashboard Integration:** Design how the risk scores for validators/proposers from this Phase 2 model will be displayed and used by administrators.

This guide provides a starting point for simulating and testing Phase 2. The core challenge lies in creating realistic simulated behaviors and robust feature engineering that can capture signals of collusion or anomaly from batch processing data.