Phase 2 Design: FL for Batch Processing Monitoring in ChainFLIP (TFF)

This document outlines the design for Phase 2 of the Federated Learning (FL) integration into your ChainFLIP project. Phase 2 focuses on using TFF to monitor the batch processing mechanism (as defined in `BatchProcessing.sol`) to detect potential collusion among validators/proposers or other anomalous behaviors.

This builds upon the Phase 1 setup (Sybil Detection) and assumes the TFF environment and basic project structure (`ChainFLIP_FL_Dev/tff_sybil_detection/`) are in place.

1. Goal of Phase 2

The primary goal is to enhance the integrity of the batch processing system by identifying:

- **Collusive Behavior:** Groups of validators consistently voting together, especially if it benefits specific proposers or leads to the approval of low-quality/malicious batches.
- **Anomalous Proposer Activity:** Proposers with unusually high batch failure rates or those attempting to push through problematic transactions.
- **Compromised Validators:** Validators whose voting patterns deviate significantly from the norm or from the eventual outcome of batches, potentially indicating they are compromised or acting maliciously.

2. Data Sources for Phase 2 FL Clients

FL clients (run by participating organizations like Primary Nodes/Validators and Secondary Nodes/Proposers) will need to extract data primarily from on-chain events and state related to `BatchProcessing.sol`.

- **Smart Contract Events (from `BatchProcessing.sol` and related contracts):**
    - `BatchProposed(uint256 batchId, address indexed proposer, address[] selectedValidators, uint256 proposalTime)` : Key for identifying who proposed what, and who was selected to validate.
    - `BatchValidated(uint256 batchId, address indexed validator, bool vote, uint256 validationTime)` : Shows individual validator votes.

    - `BatchCommitted(uint256 batchId, address indexed committer, uint256 commitTime)`: Indicates successful batch outcome.
    - `BatchFlagged(uint256 batchId, address indexed committer, uint256 flagTime)`: Indicates failed/problematic batch outcome.
    - `NodeReputationUpdated(address indexed node, int256 change, uint256 newReputation)` (from `NodeManagement.sol`): To track reputation changes resulting from batch processing actions.
- **Smart Contract State (queried programmatically):**
    - `getBatchDetails(uint256 batchId)`: To get the status, votes, proposer, validators, etc., for a specific batch.
    - `nodeReputation(address node)` (from `NodeManagement.sol`): To get current reputations.
    - `MIN_VALIDATORS_FOR_BATCH`, `superMajorityFraction` (from `BatchProcessing.sol`): System parameters.

3. Feature Engineering for Phase 2

Each FL client will locally compute features for nodes (validators, proposers) based on the data sources above. These features will be used to train local models.

For Validators:

1. `validator_vote_consistency_rate (0-1)`: Percentage of times a validator's vote (approve/reject) matches the final outcome of the batch (committed/flagged) over a recent period (e.g., last N batches they validated).
2. `validator_agreement_with_majority_rate (0-1)`: Percentage of times a validator's vote aligns with the majority vote of other validators for the same batch.
3. `validator_reputation_change_per_batch (float)`: Average reputation change (positive or negative) this validator experiences per batch they participate in.
4. `validator_batches_participated_count (int)`: Total number of batches validated recently.
5. `validator_solo_dissent_rate (0-1)`: Percentage of times a validator was the only one (or one of very few) to vote against a batch that was ultimately committed, or for a batch that was ultimately flagged.
6. `validator_approval_rate_for_low_rep_proposers (0-1)`: How often this validator approves batches from proposers with reputation below a certain threshold.

For Proposers (Secondary Nodes):

1. `proposer_batch_success_rate (0-1)`: Percentage of batches proposed by this node that get successfully committed.

2. `proposer_batch_flagged_rate (0-1)` : Percentage of batches proposed by this node that get flagged.
3. `proposer_avg_validators_approving (float)` : Average number/percentage of selected validators who approve batches from this proposer.
4. `proposer_reputation_change_per_proposal (float)` : Average reputation change for this proposer per batch they propose.
5. `proposer_batches_proposed_count (int)` : Total number of batches proposed recently.

Labels for Training:

This is more challenging than Sybil detection. Initially, labels might be derived heuristically or through semi-supervised methods: * **Anomalous Behavior (Label=1):** * Validators with very low `vote_consistency_rate` or `agreement_with_majority_rate` . * Proposers with very high `batch_flagged_rate` . * Nodes experiencing consistent, significant negative reputation changes due to batch activities. * Clusters of validators frequently voting together in a way that seems statistically unlikely or benefits a specific proposer repeatedly, especially if those batches are borderline. * **Normal Behavior (Label=0):** Nodes exhibiting expected patterns.

Alternatively, an **unsupervised anomaly detection** approach could be used first, where the FL model learns "normal" behavior, and deviations are flagged as anomalous. This avoids the need for explicit initial labeling.

4. FL Model Adaptation for Phase 2 (TFF with Keras)

- **Model Input:** The number of features will likely increase compared to Phase 1. The `NUM_FEATURES` in `data_preparation.py` and the input layer of the Keras model in `model_definition.py` will need to be updated.
- **Model Architecture (`model_definition.py`):**
    - The simple Keras Sequential model from Phase 1 can be a starting point.
    - You might need to increase its capacity (more layers/units) if the feature set is richer.
    - For detecting collusion, more advanced architectures like Graph Neural Networks (GNNs) could be considered in the long term if you can represent validator-proposer interactions as graphs, but this significantly increases complexity with TFF.
    - Initially, focus on per-node anomaly scores. If a node (validator or proposer) gets a high anomaly score from the FL model, it warrants investigation.

- **Output:** The model could output:
    - A **risk score (0-1)** for each node (validator/proposer) indicating the likelihood of them being involved in anomalous/collusive batch processing activity.
    - Or, if using classification, a probability of belonging to the "anomalous" class.

5. TFF Implementation Sketch for Phase 2

Assume you create a new subdirectory, e.g., `ChainFLIP_FL_Dev/tff_batch_monitoring/`, or adapt the Phase 1 code.

1. **`data_preparation.py` (for Batch Monitoring):**

    - Modify/create `load_local_data_for_client`:
        - Implement logic to connect to the blockchain (e.g., using `web3.py`).
        - Query events and state from `BatchProcessing.sol` and `NodeManagement.sol`.
        - Perform the feature engineering described in Section 3 for each node the client is interested in or has data for.
        - The `ELEMENT_SPEC` will need to be updated to reflect the new number of features.
    - The simulation of non-IID data should reflect different behavioral patterns of validators/proposers across clients.

2. **`model_definition.py` (for Batch Monitoring):**

    - Update `NUM_FEATURES` based on the new feature set.
    - Adjust the Keras model architecture if needed (e.g., input layer size, complexity).
    - The `tff_model_fn` will use this updated Keras model.

3. **`federated_training.py`:**

    - The `build_fed_avg_process` function can likely remain the same, as it takes the `tff_model_fn` as an argument.

4. **`run_simulation.py` (for Batch Monitoring):**

    - Update client IDs and data generation calls to use the new data preparation logic.
    - The interpretation of metrics (loss, accuracy/AUC if applicable, or anomaly scores) will be specific to the batch monitoring task.

6. Integration with Admin Dashboard for Phase 2

- The prediction service (using the trained global FL model for batch monitoring) will take features of a specific validator or proposer and output a risk/anomaly score.
- **Dashboard Display:**
    - List nodes (validators, proposers) with their FL-derived risk scores related to batch processing.
    - Highlight nodes exceeding certain risk thresholds.
    - Potentially visualize voting patterns or proposer success rates alongside FL scores.
- **Admin Actions:**
    - **Investigation:** Admins use high risk scores as a trigger to investigate specific nodes or batches more closely (e.g., reviewing on-chain history, IPFS data linked to problematic batches).
    - **Reputation Adjustment:** If an investigation confirms malicious/collusive behavior, admins can use `SupplyChainNFT.sol#adminUpdateReputation` or `adminPenalizeNode`.
    - **Temporary Suspension (Future):** For very high-risk, repeatedly flagged nodes, a mechanism to temporarily suspend their ability to validate or propose batches could be considered (would require smart contract changes).

7. Smart Contract Considerations for Phase 2

- The existing events in `BatchProcessing.sol` and `NodeManagement.sol` are quite comprehensive and should provide most of the necessary data for feature engineering.
- No immediate new functions or events seem strictly necessary for Phase 2 data collection, but this should be re-evaluated during detailed feature engineering.

8. Testing Phase 2

- **Structured Synthetic Data:** Similar to Phase 1, create synthetic data in `data_preparation.py` that simulates:
    - Normal validators/proposers.
    - Colluding validators (e.g., a group that always approves a specific bad proposer).
    - A consistently failing proposer.
    - A validator that always votes against the majority.

- **Metrics:** Monitor if the FL model learns to assign higher risk scores to these simulated malicious/anomalous entities.

This design provides a roadmap for Phase 2. The next step would be to implement the data extraction and feature engineering, adapt the TFF scripts, and then develop testing assets.