SOT Extensions Architecture

Miner-Contributed Features and Pattern Discovery

Date: 2025-10-25
Status: Extension Proposal
Context: How miners can extend the validator's SOT data with novel features and patterns

---

Executive Summary

While the core miner capabilities (alert scoring, prioritization, cluster assessment) operate on **existing SOT data**, advanced miners can contribute **extensions to the SOT** through:

1. **Proposal 4**: Feature Engineering & Enrichment
2. **Proposal 5**: Anomaly Detection & Pattern Discovery

These extensions create a **bidirectional data flow** where miners not only consume SOT data but also contribute improvements back to it.

Architecture Overview

```
┌─────────────────────────────────────────────────────────────────┐
│                    VALIDATOR (SOT Owner)                        │
├─────────────────────────────────────────────────────────────────┤
│                                                                 │
│  Core SOT Generation (Daily):                                   │
│  ├─ analyzers_alerts (typology detections)                     │
│  ├─ analyzers_alert_clusters (clustering)                      │
│  ├─ analyzers_features (140+ base features)                    │
│  └─ core_money_flows (transaction graph)                       │
│                                                                 │
│  Extended SOT (Miner Contributions):                            │
│  ├─ analyzers_miner_features (enhanced features)               │
│  └─ analyzers_miner_patterns (discovered patterns)             │
│                                                                 │
└─────────────────────────────────────────────────────────────────┘
                     │
                     │
        ┌────────────┴────────────┐
        │                         │
        ▼                         ▼
  ┌──────────┐              ┌──────────┐
  │ Miner A  │              │ Miner B  │
  │          │              │          │
  │ Submits: │              │ Submits: │
  │ Features │              │ Patterns │
  └──────────┘              └──────────┘
        │                         │
```

```
┌────────────────────────┐
│  Consensus Process     │
│  (≥3 miners agree)     │
└────────────────────────┘
           │
           ▼
┌────────────────────────┐
│  SOT Update (Next Day) │
│  New features/patterns │
│  available to all      │
└────────────────────────┘
```

Proposal 4: Feature Engineering Extensions

What Miners Submit

Enhanced Feature Vectors that augment the base `analyzers_features` table.

Architecture Components

4.1 New Schema: `analyzers_miner_features`

```sql
CREATE TABLE analyzers_miner_features (
    -- Time series dimensions
    window_days UInt16,
    processing_date Date,

    -- Identity
    miner_id String,
    feature_version String,

    -- Address identifier
    address String,

    -- Enhanced features (miner-specific)
    graph_embedding Array(Float32),       -- Node2Vec/DeepWalk embeddings
    temporal_derivatives Array(Float32), -- Rate of change features
    domain_signals Array(Float32),       -- AML-specific engineered features

    -- Feature metadata
    feature_names Array(String),         -- Names for each value
    feature_importance Array(Float32),   -- SHAP/importance scores
    computation_time_ms UInt32,

    -- Quality metrics
    correlation_with_base Float32,       -- Max correlation with base features
    information_gain Float32,            -- Mutual information with target
```

```
    coverage Float32,                    -- Non-null rate

    -- Validation
    _version UInt64,
    created_ts UInt64
)
ENGINE = ReplacingMergeTree(_version)
PARTITION BY (window_days, toYYYYMM(processing_date))
ORDER BY (window_days, processing_date, miner_id, address)
SETTINGS index_granularity = 8192;
```

4.2 Feature Submission Process

```python
class MinerFeatureSubmission:
    """
    Miner submits enhanced features for addresses in daily batch
    """

    def compute_enhanced_features(self, addresses, base_features, money_flows):
        """
        Step 1: Miner computes enhanced features
        """
        # Example: Graph embeddings
        embeddings = self.node2vec(money_flows, dimensions=64)

        # Example: Temporal derivatives
        derivatives = self.compute_derivatives(base_features)

        # Example: Domain-specific signals
        aml_signals = self.compute_aml_signals(base_features, money_flows)

        return {
            'graph_embedding': embeddings,
            'temporal_derivatives': derivatives,
            'domain_signals': aml_signals,
            'feature_names': self.get_feature_names(),
            'feature_importance': self.compute_importance()
        }

    def submit_features(self, enhanced_features):
        """
        Step 2: Submit to validator with proof of computation
        """
        submission = {
            'miner_id': self.miner_id,
            'feature_version': self.model_version,
            'features': enhanced_features,
            'manifest': {
                'computation_method': 'Node2Vec + temporal derivatives',
                'dependencies': {'sklearn': '1.3', 'node2vec': '0.4.6'},
                'seed': 42
```

```
            },
            'signature': self.sign(enhanced_features)
        }
        return submission
```

4.3 Validator Feature Validation

```python
class FeatureValidator:
    """
    Validator evaluates submitted features for quality and utility
    """

    def validate_submission(self, submission, base_features, ground_truth):
        """
        Multi-stage validation of miner features
        """
        checks = {}

        # Stage 1: Format and integrity
        checks['format'] = self.check_format(submission)
        checks['signature'] = self.verify_signature(submission)
        checks['coverage'] = self.check_coverage(submission.features)

        # Stage 2: Quality metrics
        checks['redundancy'] = self.check_redundancy(
            submission.features,
            base_features,
            threshold=0.95  # Max correlation with any base feature
        )
        checks['information_gain'] = self.compute_mutual_information(
            submission.features,
            ground_truth
        )

        # Stage 3: Utility testing
        checks['ablation'] = self.ablation_test(
            submission.features,
            base_features,
            ground_truth
        )

        return checks

    def ablation_test(self, enhanced_features, base_features, targets):
        """
        Test if enhanced features improve model performance
        """
        # Baseline model with base features only
        baseline_score = self.train_and_evaluate(base_features, targets)

        # Enhanced model with base + miner features
```

```
        enhanced_score = self.train_and_evaluate(
            concat(base_features, enhanced_features),
            targets
        )

        improvement = enhanced_score - baseline_score

        return {
            'baseline_auc': baseline_score,
            'enhanced_auc': enhanced_score,
            'improvement': improvement,
            'significant': improvement > 0.01  # Minimum threshold
        }
```

4.4 Consensus and SOT Integration

```
class FeatureConsensusManager:
    """
    Manages consensus across multiple miners for feature adoption
    """

    def evaluate_consensus(self, feature_submissions):
        """
        Determine which features to add to SOT
        """
        # Requirement: ≥3 miners submit similar features
        consensus_features = []

        for feature_type in ['graph_embedding', 'temporal_derivatives']:
            miners_with_feature = [
                s for s in feature_submissions
                if feature_type in s.features
            ]

            if len(miners_with_feature) >= 3:
                # Compute average feature values
                avg_features = self.average_features(miners_with_feature)

                # Validate consensus quality
                variance = self.compute_variance(miners_with_feature)
                utility = self.compute_utility(avg_features)

                if variance < 0.1 and utility > 0.01:
                    consensus_features.append({
                        'type': feature_type,
                        'values': avg_features,
                        'contributors': [m.miner_id for m in miners_with_feature],
                        'variance': variance,
                        'utility': utility
                    })
```

```
        return consensus_features

    def integrate_into_sot(self, consensus_features):
        """
        Add consensus features to next day's SOT batch
        """
        # Update schema to include new feature columns
        schema_update = self.generate_schema_update(consensus_features)

        # Compute features for all addresses
        for address in self.all_addresses:
            enhanced_values = self.compute_consensus_features(
                address,
                consensus_features
            )
            self.feature_repository.update(address, enhanced_values)

        # Log feature addition
        self.log_sot_extension({
            'date': self.processing_date,
            'features_added': len(consensus_features),
            'contributors': set([c for f in consensus_features for c in
f.contributors]),
            'utility_improvement': sum(f.utility for f in consensus_features)
        })
```

Feature Lifecycle

```
Day 1: Miner A discovers useful graph embedding
├─ Computes embeddings for addresses
├─ Submits to validator with proof
└─ Validator validates but waits for consensus

Day 2-3: Miners B, C also submit similar embeddings
├─ Validator detects consensus (3+ miners)
├─ Runs ablation tests
└─ Confirms utility improvement

Day 4: Feature added to SOT
├─ Validator computes embedding for all addresses
├─ New column added to analyzers_features
├─ All miners can now use this feature
└─ Original contributors get discovery bonus
```

Proposal 5: Pattern Discovery Extensions

What Miners Submit

Novel anomaly patterns not detected by existing typology rules, which can become new alerts in future SOT batches.

Architecture Components

5.1 New Schema: `analyzers_miner_patterns`

```
CREATE TABLE analyzers_miner_patterns (
    -- Time series dimensions
    window_days UInt16,
    processing_date Date,

    -- Pattern identity
    pattern_id String,
    miner_id String,
    pattern_version String,

    -- Pattern definition
    pattern_type String,                -- e.g., 'circular_flow',
'timing_cluster'
    pattern_description String,
    detection_logic String,             -- Algorithm description

    -- Detected instances
    addresses Array(String),            -- Addresses matching pattern
    confidence_scores Array(Float32),   -- Confidence per address
    evidence_json Array(String),        -- Evidence per address

    -- Pattern characteristics
    min_participants UInt32,
    max_hops UInt32,
    temporal_window_hours UInt32,
    amount_range_usd Tuple(Decimal128(18), Decimal128(18)),

    -- Quality metrics
    prevalence Float32,                 -- % of addresses matching
    novelty_score Float32,              -- Similarity to existing patterns
    discriminative_power Float32,       -- Correlation with illicit outcomes

    -- Validation status
    consensus_count UInt32,             -- Number of miners detecting same
pattern
    confirmed_illicit_rate Float32,     -- T+τ confirmation rate
    false_positive_rate Float32,

    _version UInt64,
    created_ts UInt64
)
ENGINE = ReplacingMergeTree(_version)
PARTITION BY (window_days, toYYYYMM(processing_date))
ORDER BY (window_days, processing_date, pattern_id, miner_id)
SETTINGS index_granularity = 8192;
```

5.2 Pattern Discovery Process

```
class PatternDiscoveryMiner:
    """
    Miner discovers novel anomaly patterns in money flows
    """

    def discover_patterns(self, money_flows, base_features):
        """
        Apply unsupervised learning to find new patterns
        """
        patterns = []

        # Method 1: Graph motif mining
        circular_flows = self.detect_circular_flows(money_flows)
        if len(circular_flows) > 0:
            patterns.append({
                'type': 'circular_flow',
                'instances': circular_flows,
                'description': 'Money returns to source within N hops'
            })

        # Method 2: Temporal clustering
        timing_clusters = self.detect_timing_anomalies(money_flows)
        if len(timing_clusters) > 0:
            patterns.append({
                'type': 'timing_cluster',
                'instances': timing_clusters,
                'description': 'Burst of transactions in narrow time window'
            })

        # Method 3: Amount patterns
        value_laddering = self.detect_amount_patterns(money_flows)
        if len(value_laddering) > 0:
            patterns.append({
                'type': 'value_laddering',
                'instances': value_laddering,
                'description': 'Sequential transactions with increasing amounts'
            })

        return patterns

    def detect_circular_flows(self, flows):
        """
        Find cycles in transaction graph
        """
        G = self.build_graph(flows)
        cycles = []

        for node in G.nodes():
```

```
            # BFS to find paths back to node
            paths = self.find_cycles_from_node(G, node, max_hops=6)

            for path in paths:
                if self.is_suspicious_cycle(path):
                    cycles.append({
                        'addresses': path,
                        'total_volume': sum(G[a][b]['volume'] for a,b in
zip(path[:-1], path[1:])),
                        'hops': len(path) - 1,
                        'time_span': self.compute_time_span(path),
                        'confidence': self.compute_cycle_confidence(path)
                    })

        return cycles

    def submit_pattern(self, pattern):
        """
        Submit discovered pattern to validator
        """
        submission = {
            'pattern_id': self.generate_pattern_id(pattern),
            'miner_id': self.miner_id,
            'pattern_version': self.version,
            'pattern_type': pattern['type'],
            'description': pattern['description'],
            'detection_logic': self.get_algorithm_description(),
            'addresses': pattern['instances'],
            'confidence_scores': [i['confidence'] for i in pattern['instances']],
            'evidence_json': [self.create_evidence(i) for i in
pattern['instances']],
            'characteristics': {
                'min_participants': min(len(i['addresses']) for i in
pattern['instances']),
                'max_hops': max(i.get('hops', 0) for i in pattern['instances']),
                'temporal_window_hours': self.compute_temporal_window(pattern)
            },
            'signature': self.sign(pattern)
        }
        return submission
```

5.3 Pattern Validation and Consensus

```
class PatternConsensusValidator:
    """
    Validates and builds consensus around discovered patterns
    """

    def validate_pattern_submission(self, submission):
        """
        Multi-stage pattern validation
```

```
        """
        validation = {}

        # Stage 1: Novelty check
        validation['novelty'] = self.check_novelty(
            submission.pattern_type,
            submission.detection_logic,
            existing_patterns=self.get_existing_patterns()
        )

        # Stage 2: Significance check
        validation['prevalence'] = len(submission.addresses) /
self.total_addresses
        validation['rare_enough'] = validation['prevalence'] < 0.05  # Must be
rare

        # Stage 3: Reproducibility
        validation['reproducible'] = self.reproduce_pattern(
            submission.detection_logic,
            submission.addresses
        )

        return validation

    def build_consensus(self, pattern_submissions):
        """
        Determine if multiple miners found same pattern
        """
        # Group submissions by pattern similarity
        pattern_groups = self.group_similar_patterns(pattern_submissions)

        consensus_patterns = []
        for group in pattern_groups:
            if len(group) >= 3:  # Require 3+ miners
                # Check address overlap
                address_overlap = self.compute_address_overlap(group)

                if address_overlap > 0.7:  # 70% of addresses must match
                    consensus_patterns.append({
                        'pattern_type': group[0].pattern_type,
                        'contributors': [p.miner_id for p in group],
                        'consensus_addresses': self.intersect_addresses(group),
                        'avg_confidence': np.mean([p.confidence_scores for p in
group]),
                        'agreement_score': address_overlap
                    })

        return consensus_patterns

    def schedule_pattern_confirmation(self, consensus_pattern, tau_days=21):
        """
        Schedule T+τ validation for pattern
        """
        confirmation_task = {
```

```
            'pattern_id': consensus_pattern.pattern_id,
            'addresses': consensus_pattern.consensus_addresses,
            'check_date': self.processing_date + timedelta(days=tau_days),
            'validation_logic': 'check_if_addresses_became_illicit'
        }

        self.confirmation_scheduler.schedule(confirmation_task)
```

5.4 Pattern Confirmation (T+τ)

```
class PatternConfirmationEngine:
    """
    Validates patterns against actual outcomes after τ days
    """

    def confirm_pattern(self, pattern_id, tau_days=21):
        """
        Check if pattern predictions were accurate
        """
        pattern = self.get_pattern(pattern_id)
        original_date = pattern.processing_date
        current_date = original_date + timedelta(days=tau_days)

        # Get actual outcomes for addresses
        outcomes = []
        for address in pattern.addresses:
            outcome = self.check_address_outcome(
                address,
                start_date=original_date,
                end_date=current_date
            )
            outcomes.append(outcome)

        # Compute confirmation metrics
        confirmation = {
            'pattern_id': pattern_id,
            'total_addresses': len(pattern.addresses),
            'confirmed_illicit': sum(o.illicit for o in outcomes),
            'confirmed_clean': sum(o.clean for o in outcomes),
            'inconclusive': sum(o.inconclusive for o in outcomes),
            'confirmed_illicit_rate': sum(o.illicit for o in outcomes) /
len(outcomes),
            'false_positive_rate': sum(o.clean for o in outcomes) / len(outcomes),
            'avg_days_to_confirmation': np.mean([o.days_to_event for o in outcomes
if o.illicit])
        }

        # Update pattern status
        if confirmation['confirmed_illicit_rate'] > 0.3:  # 30% threshold
            self.promote_pattern_to_typology(pattern, confirmation)
```

```
        return confirmation

    def promote_pattern_to_typology(self, pattern, confirmation):
        """
        Add validated pattern as new typology rule in SOT
        """
        new_typology = {
            'typology_type': pattern.pattern_type,
            'description': pattern.description,
            'detection_logic': pattern.detection_logic,
            'discovered_by': pattern.contributors,
            'discovery_date': pattern.processing_date,
            'confirmation_rate': confirmation.confirmed_illicit_rate,
            'status': 'active'
        }

        # Add to typology detector configuration
        self.typology_config.add_rule(new_typology)

        # Reward pattern discoverers
        self.reward_discoverers(
            pattern.contributors,
            bonus_multiplier=2.0  # 2x bonus for validated discovery
        )

        # Log SOT extension
        self.log_sot_extension({
            'type': 'new_typology',
            'pattern_type': pattern.pattern_type,
            'contributors': pattern.contributors,
            'confirmed_illicit_rate': confirmation.confirmed_illicit_rate
        })
```

SOT Extension Lifecycle

Complete Flow

```
┌─────────────────────────────────────────────────────────────┐
│ Day 0: Pattern Discovery                                    │
├─────────────────────────────────────────────────────────────┤
│ Miner A: Detects circular flow pattern in 50 addresses     │
│ Miner B: Detects same pattern in 45 addresses (40 overlap)│
│ Miner C: Detects timing cluster in 30 addresses            │
└─────────────────────────────────────────────────────────────┘
                         ↓
┌─────────────────────────────────────────────────────────────┐
│ Day 1: Consensus Building                                   │
├─────────────────────────────────────────────────────────────┤
│ Validator: Detects consensus on circular flow (A & B)      │
│ Validator: No consensus on timing cluster (only C)         │
```

```
│ Validator: Validates novelty and significance        │
│ Validator: Schedules T+21 confirmation               │
└──────────────────────────────────────────────────────┘
                        ↓
┌──────────────────────────────────────────────────────┐
│ Day 2-20: Monitoring                                 │
├──────────────────────────────────────────────────────┤
│ Pattern tracked in analyzers_miner_patterns          │
│ More miners may discover same pattern (strengthen)   │
│ Pattern not yet in SOT (provisional status)          │
└──────────────────────────────────────────────────────┘
                        ↓
┌──────────────────────────────────────────────────────┐
│ Day 21: Pattern Confirmation                         │
├──────────────────────────────────────────────────────┤
│ Validator checks outcomes for 40 consensus addresses │
│ Results: 15 confirmed illicit (37.5% rate)           │
│ Threshold met: Pattern promoted to typology          │
└──────────────────────────────────────────────────────┘
                        ↓
┌──────────────────────────────────────────────────────┐
│ Day 22+: SOT Integration                             │
├──────────────────────────────────────────────────────┤
│ New typology added to detector configuration         │
│ Future daily batches include this pattern detection  │
│ Original discoverers (A & B) receive bonus rewards   │
│ Pattern becomes standard feature in analyzers_alerts │
└──────────────────────────────────────────────────────┘
```

Reward Structure for SOT Extensions

Feature Engineering Rewards

```
feature_reward = (
    base_reward
    * utility_multiplier          # 1.0 - 2.0 based on ablation test
    * consensus_bonus             # 1.5x if ≥3 miners agree
    * adoption_bonus              # 2.0x if added to SOT
    * longevity_multiplier        # Ongoing reward if feature remains useful
)
```

Pattern Discovery Rewards

```
pattern_reward = (
    base_reward
    * novelty_multiplier          # 2.0x for truly novel patterns
    * confirmation_rate_multiplier # 1.0 - 3.0 based on illicit rate
    * consensus_bonus             # 1.5x if ≥3 miners agree
```

```
        * first_discoverer_bonus        # 3.0x for first miner
        * adoption_bonus                # 5.0x if promoted to typology
)
```

Implementation Considerations

Database Impact

```sql
-- New tables required
analyzers_miner_features      -- ~10-50 GB/month (if 10 miners × 500K addresses)
analyzers_miner_patterns      -- ~1-5 GB/month (sparse, consensus only)

-- Extended existing tables
ALTER TABLE analyzers_features ADD COLUMN miner_embeddings Array(Float32);
-- Add consensus features to base schema

-- New materialized view for consensus tracking
CREATE MATERIALIZED VIEW analyzers_feature_consensus_mv ...
```

Performance Considerations

- Feature computation must be efficient (≤5 min for 500K addresses)
- Pattern detection batched/sampled to prevent exponential complexity
- Consensus building runs daily, not real-time
- T+τ confirmation runs weekly for efficiency

Security Considerations

- Prevent miner coordination to game consensus (diversity checks)
- Validate computation claims through spot audits
- Rate limit submissions to prevent spam
- Require stake or reputation threshold for submissions

Conclusion

Proposals 4 & 5 create a **collaborative intelligence network** where:

1. **Validators maintain canonical SOT** (authoritative data)
2. **Miners contribute extensions** (features, patterns)
3. **Consensus determines adoption** (≥3 miners agreement)
4. **Time validates quality** (T+τ confirmation)
5. **Successful extensions become SOT** (permanent integration)

This architecture enables continuous improvement of the AML system through decentralized innovation while maintaining data quality through rigorous validation.