```
// RiskCircuit Core Engine
// Production-grade risk analysis with real mathematical rigor

import { ethers } from 'ethers';
import * as tf from '@tensorflow/tfjs';

// ============================================
// ENTROPY ANALYSIS MODULE
// ============================================

class EntropyAnalyzer {
  constructor() {
    this.timeWindows = [3600, 86400, 604800]; // 1hr, 1day, 1week in
seconds
  }

  // Shannon entropy calculation
  calculateShannon(distribution) {
    const normalized = this.normalizeDistribution(distribution);
    return -normalized.reduce((sum, p) => {
      return sum + (p > 0 ? p * Math.log2(p) : 0);
    }, 0);
  }

  // Temporal entropy - measures randomness in transaction timing
  calculateTemporalEntropy(timestamps) {
    if (timestamps.length < 2) return 0;

    // Calculate inter-transaction intervals
    const intervals = [];
    for (let i = 1; i < timestamps.length; i++) {
      intervals.push(timestamps[i] - timestamps[i-1]);
    }

    // Bin intervals logarithmically
    const bins = this.createLogBins(Math.min(...intervals),
Math.max(...intervals), 20);
    const distribution = this.binData(intervals, bins);

    return this.calculateShannon(distribution);
  }

  // Value entropy - measures diversity in transaction amounts
  calculateValueEntropy(values) {
    if (values.length === 0) return 0;

    // Remove outliers using IQR method
    const cleaned = this.removeOutliers(values);
```

```
    if (cleaned.length === 0) return 0;

    // Create logarithmic bins for value distribution
    const minVal = Math.min(...cleaned);
    const maxVal = Math.max(...cleaned);
    const bins = this.createLogBins(minVal, maxVal, 15);
    const distribution = this.binData(cleaned, bins);

    return this.calculateShannon(distribution);
  }

  // Protocol interaction entropy
  calculateProtocolEntropy(interactions) {
    const protocolCounts = {};
    interactions.forEach(int => {
      const protocolName = int.protocol || 'Unknown';
      protocolCounts[protocolName] = (protocolCounts[protocolName] ||
0) + 1;
    });

    const distribution = Object.values(protocolCounts);
    return this.calculateShannon(distribution);
  }

  // Gas price entropy - can indicate bot vs human behavior
  calculateGasEntropy(gasPrices) {
    if (gasPrices.length === 0) return 0;
    const bins = this.createLinearBins(Math.min(...gasPrices),
Math.max(...gasPrices), 10);
    const distribution = this.binData(gasPrices, bins);
    return this.calculateShannon(distribution);
  }

  // Composite entropy score with Lyapunov stability check
  computeCompositeEntropy(walletData) {
    const weights = {
      temporal: 0.3,
      value: 0.25,
      protocol: 0.25,
      gas: 0.2
    };

    const entropies = {
      temporal: this.calculateTemporalEntropy(walletData.timestamps),
      value: this.calculateValueEntropy(walletData.values),
      protocol: this.calculateProtocolEntropy(walletData.protocols),
      gas: this.calculateGasEntropy(walletData.gasPrices)
    };
```

```
    // Weighted average
    const composite = Object.entries(entropies).reduce((sum, [key,
value]) => {
      return sum + (weights[key] * value);
    }, 0);

    // Lyapunov stability check - detect if entropy is changing
rapidly
    const stability =
this.checkLyapunovStability(walletData.historicalEntropies);

    return {
      composite,
      components: entropies,
      stability,
      interpretation: this.interpretEntropy(composite, stability)
    };
  }

  // Lyapunov stability analysis
  checkLyapunovStability(timeSeries) {
    if (timeSeries.length < 10) return { stable: true, exponent: 0,
interpretation: 'stable' };

    // Calculate rate of divergence
    const differences = [];
    for (let i = 1; i < timeSeries.length; i++) {
      differences.push(Math.abs(timeSeries[i] - timeSeries[i-1]));
    }

    // Estimate Lyapunov exponent
    const avgDivergence = differences.reduce((a, b) => a + b) /
differences.length;
    if (avgDivergence <= 0) return { stable: true, exponent:
-Infinity, interpretation: 'stable' };
    const lyapunovExponent = Math.log(avgDivergence);

    return {
      stable: lyapunovExponent < 0.1,
      exponent: lyapunovExponent,
      interpretation: lyapunovExponent > 0.5 ? 'chaotic' :
                     lyapunovExponent > 0.1 ? 'unstable' : 'stable'
    };
  }

  // Utility functions
  normalizeDistribution(distribution) {
```

```
    const sum = distribution.reduce((a, b) => a + b, 0);
    if (sum === 0) return distribution.map(() => 0);
    return distribution.map(val => val / sum);
  }

  createLogBins(min, max, numBins) {
    if (min <= 0) min = 0.0001;
    const logMin = Math.log10(min);
    const logMax = Math.log10(max + 1);
    const step = (logMax - logMin) / numBins;

    return Array.from({length: numBins + 1}, (_, i) =>
      Math.pow(10, logMin + i * step)
    );
  }

  createLinearBins(min, max, numBins) {
    if (min === max) return [min, max];
    const step = (max - min) / numBins;
    return Array.from({length: numBins + 1}, (_, i) => min + i *
step);
  }

  binData(data, bins) {
    const counts = new Array(bins.length - 1).fill(0);
    data.forEach(val => {
      for (let i = 0; i < bins.length - 1; i++) {
        if (val >= bins[i] && val < bins[i+1]) {
          counts[i]++;
          return;
        }
      }
      // Check last bin inclusively
      if(val === bins[bins.length-1]) {
          counts[counts.length-1]++;
      }
    });
    return counts;
  }

  removeOutliers(data) {
    if (data.length < 4) return data;
    const sorted = [...data].sort((a, b) => a - b);
    const q1 = sorted[Math.floor(sorted.length * 0.25)];
    const q3 = sorted[Math.floor(sorted.length * 0.75)];
    const iqr = q3 - q1;
    const lower = q1 - 1.5 * iqr;
    const upper = q3 + 1.5 * iqr;
```

```
    return data.filter(val => val >= lower && val <= upper);
  }

  interpretEntropy(entropy, stability) {
    if (!stability.stable) {
      return `UNSTABLE: Rapidly changing behavior patterns detected
(${stability.interpretation})`;
    }

    if (entropy < 1.5) return 'LOW: Highly predictable, regular
behavior';
    if (entropy < 2.5) return 'MODERATE: Some behavioral diversity';
    if (entropy < 3.5) return 'HIGH: Complex and diverse interaction
patterns';
    return 'VERY HIGH: Potentially chaotic or obfuscated behavior';
  }
}

// ============================================
// ADJACENCY GRAPH ANALYZER
// ============================================

class AdjacencyAnalyzer {
  constructor() {
    // In a real system, this would be populated from a dynamic,
trusted source.
    this.knownBadActors = new Set([
        '0x7c69a6395b283347fce5b3b5a17277e49d6b7b28' // Example: Known
phisher
    ]);
    this.maxDepth = 3;
    this.decayFactor = 0.5; // Risk decays by 50% per hop
  }

  async mapAdjacencyGraph(address, provider, depth = 2) {
    const graph = new Map(); // Maps address -> { interactions: [],
metadata: {} }
    const queue = [{address, depth: 0}];
    const visited = new Set();

    while (queue.length > 0) {
      const {address: current, depth: currentDepth} = queue.shift();

      if (visited.has(current) || currentDepth > depth) continue;
      visited.add(current);

      // Get interactions
```

```
      const interactions = await this.getInteractions(current,
provider);
      graph.set(current, { interactions });

      // Add to queue for next level
      if (currentDepth < depth) {
        interactions.forEach(interaction => {
          if (!visited.has(interaction.address)) {
            queue.push({address: interaction.address, depth:
currentDepth + 1});
          }
        });
      }
    }

    return this.analyzeGraph(graph, address);
  }

  async getInteractions(address, provider) {
    // This is a simplified approach. A production system would use a
dedicated indexing service
    // like Etherscan API, Alchemy Transfers API, or The Graph for
performance and accuracy.
    try {
        const history = await provider.getHistory(address);
        const interactions = new Map();

        history.slice(0, 50).forEach(tx => { // Limit to last 50 for
performance
            const counterparty = tx.from.toLowerCase() ===
address.toLowerCase() ? tx.to : tx.from;
            if (!counterparty) return;

            const existing = interactions.get(counterparty) || {
count: 0, value: 0, direction: {in: 0, out: 0} };
            const txValue = parseFloat(ethers.formatEther(tx.value));

            interactions.set(counterparty, {
                count: existing.count + 1,
                value: existing.value + txValue,
                direction: {
                    in: existing.direction.in + (tx.to.toLowerCase()
=== address.toLowerCase() ? txValue : 0),
                    out: existing.direction.out +
(tx.from.toLowerCase() === address.toLowerCase() ? txValue : 0)
                }
            });
        });
```

```
        return Array.from(interactions.entries()).map(([addr, data])
=> ({
            address: addr,
            ...data
        }));
    } catch(e) {
        console.warn(`Could not get history for ${address}:`,
e.message);
        return [];
    }
  }

  analyzeGraph(graph, rootAddress) {
    const riskScores = new Map();
    const predecessors = new Map(); // For path reconstruction

    // BFS to calculate risk propagation
    const queue = [{address: rootAddress, depth: 0, inheritedRisk:
0}];
    const visited = new Set([rootAddress]);
    riskScores.set(rootAddress, this.calculateNodeRisk(rootAddress));

    while (queue.length > 0) {
      const { address: from, depth } = queue.shift();
      const neighbors = graph.get(from)?.interactions || [];

      for (const neighbor of neighbors) {
          const to = neighbor.address;
          const nodeRisk = this.calculateNodeRisk(to);
          const currentRisk = riskScores.get(from);
          // Risk propagates from high-risk nodes to their neighbors
          const propagatedRisk = currentRisk * this.decayFactor;

          if (!riskScores.has(to) || propagatedRisk >
(riskScores.get(to) - nodeRisk)) {
              riskScores.set(to, nodeRisk + propagatedRisk);
              predecessors.set(to, from);
              if(!visited.has(to)) {
                  visited.add(to);
                  queue.push({address: to, depth: depth + 1});
              }
          }
      }
    }

    const { flagged, highestRisk } =
this.findFlaggedConnections(graph, rootAddress);
```

```
    return {
      totalRisk: riskScores.get(rootAddress) || 0,
      riskPath: this.findHighestRiskPath(predecessors, rootAddress,
highestRisk?.address),
      flaggedConnections: flagged
    };
  }

  calculateNodeRisk(address) {
    if (!address) return 0.0;
    // Check against known bad actors
    if (this.knownBadActors.has(address.toLowerCase())) {
      return 1.0; // Maximum risk
    }

    // Check against patterns (simplified)
    if (address.toLowerCase().includes('tornadocash')) return 0.8; //
Known mixer patterns
    if (address.startsWith('0x00000')) return 0.3; // Interaction with
null addresses

    return 0.0; // Default no risk
  }

  findHighestRiskPath(predecessors, startNode, endNode) {
    if (!endNode || !predecessors.has(endNode)) return [];

    const path = [];
    let current = endNode;
    while(current && current !== startNode) {
        path.unshift(current);
        current = predecessors.get(current);
    }
    path.unshift(startNode);
    return path;
  }

  findFlaggedConnections(graph, rootAddress) {
    const flagged = [];
    let highestRisk = { address: null, risk: -1 };
    const visited = new Set();
    const queue = [{address: rootAddress, depth: 0}];

    while (queue.length > 0) {
      const {address, depth} = queue.shift();
      if (visited.has(address)) continue;
      visited.add(address);
```

```
      const risk = this.calculateNodeRisk(address);
      if (risk > 0) {
        flagged.push({ address, depth, risk });
      }
      if (risk > highestRisk.risk) {
          highestRisk = { address, risk };
      }

      const neighbors = graph.get(address)?.interactions || [];
      if (depth < this.maxDepth) {
          neighbors.forEach(n => queue.push({address: n.address,
depth: depth + 1}));
      }
    }

    return { flagged, highestRisk };
  }
}

// ============================================
// BAYESIAN RISK NETWORK
// ============================================

class BayesianRiskNetwork {
  constructor() {
    this.nodes = this.initializeNetwork();
    this.evidenceBuffer = [];
  }

  initializeNetwork() {
    return {
      walletAge: {
        states: ['new', 'established', 'veteran'],
        priors: [0.4, 0.4, 0.2]
      },
      fundingSource: {
        states: ['cex', 'mixer', 'defi', 'unlabeled'],
        priors: [0.5, 0.1, 0.3, 0.1]
      },
      riskLevel: {
        states: ['low', 'medium', 'high'],
        parents: ['walletAge', 'fundingSource'],
        cpt: this.initializeRiskCPT()
      }
    };
  }
```

```
  initializeRiskCPT() {
    // CPT for P(Risk | Age, Funding)
    // Structure: [P(low), P(medium), P(high)]
    const cpt = {};
    const ages = this.nodes.walletAge.states;
    const sources = this.nodes.fundingSource.states;

    // Default probabilities
    ages.forEach(age => {
        sources.forEach(source => {
            cpt[`${age},${source}`] = [0.6, 0.3, 0.1]; // Default to
low-ish risk
        });
    });

    // Specific overrides based on heuristics
    Object.assign(cpt, {
        'new,mixer': [0.05, 0.15, 0.8],
        'new,unlabeled': [0.2, 0.5, 0.3],
        'new,defi': [0.4, 0.4, 0.2],
        'new,cex': [0.8, 0.15, 0.05],

        'established,mixer': [0.1, 0.3, 0.6],
        'established,unlabeled': [0.3, 0.4, 0.3],
        'established,defi': [0.6, 0.3, 0.1],
        'established,cex': [0.85, 0.1, 0.05],

        'veteran,mixer': [0.2, 0.5, 0.3],
        'veteran,unlabeled': [0.5, 0.3, 0.2],
        'veteran,defi': [0.7, 0.25, 0.05],
        'veteran,cex': [0.9, 0.08, 0.02],
    });
    return cpt;
  }

  async infer(evidence) {
    // Simplified inference. A full implementation would use a
junction tree or variable elimination.
    // This direct calculation is sufficient for this fixed, simple
network structure.
    const { walletAge, fundingSource } = evidence;

    if (!walletAge || !fundingSource) {
        throw new Error("Insufficient evidence for Bayesian inference.
Need walletAge and fundingSource.");
    }

    const key = `${walletAge},${fundingSource}`;
```

```
    const posterior = this.nodes.riskLevel.cpt[key];

    if (!posterior) {
        console.warn(`No CPT entry for key: ${key}. Using default.`);
        return {
            distribution: { low: 0.33, medium: 0.33, high: 0.34 },
            maxLikelihoodState: 'medium',
            confidence: 0.0
        };
    }

    const posteriorMap = {
        low: posterior[0],
        medium: posterior[1],
        high: posterior[2]
    };

    return {
      distribution: posteriorMap,
      maxLikelihoodState: this.getMaxLikelihoodState(posteriorMap),
      confidence:
this.calculateConfidence(Object.values(posteriorMap))
    };
  }

  getMaxLikelihoodState(distributionMap) {
    return Object.keys(distributionMap).reduce((a, b) =>
distributionMap[a] > distributionMap[b] ? a : b);
  }

  calculateConfidence(distribution) {
    const entropy = -distribution.reduce((sum, p) =>
      sum + (p > 0 ? p * Math.log2(p) : 0), 0
    );
    const maxEntropy = Math.log2(distribution.length);
    if (maxEntropy === 0) return 1;

    return Math.max(0, 1 - (entropy / maxEntropy));
  }

  updateCPTs(evidence, outcome) {
    // This is a placeholder for online learning.
    // A real implementation would require a robust mechanism for
tracking outcomes.
    this.evidenceBuffer.push({evidence, outcome});

    if (this.evidenceBuffer.length >= 100) {
      this.learnFromBuffer();
```

```
      this.evidenceBuffer = [];
    }
  }

  learnFromBuffer() {
    // Implements online learning to update CPTs based on observed
outcomes.
    // This is a simplified example using MLE with Laplace smoothing.
    console.log("Updating Bayesian network parameters from new
evidence...");
    const counts = {};
    const totals = {};

    this.evidenceBuffer.forEach(({ evidence, outcome }) => {
        const key = `${evidence.walletAge},${evidence.fundingSource}`;
        if (!counts[key]) {
            counts[key] = { low: 0, medium: 0, high: 0 };
            totals[key] = 0;
        }
        counts[key][outcome]++;
        totals[key]++;
    });

    for (const key in counts) {
        const total = totals[key];
        const riskCounts = counts[key];
        // Update CPT with smoothing
        this.nodes.riskLevel.cpt[key] = [
            (riskCounts.low + 1) / (total + 3),
            (riskCounts.medium + 1) / (total + 3),
            (riskCounts.high + 1) / (total + 3),
        ];
    }
    console.log("Network update complete.");
  }
}

// ============================================
// MAIN RISK ENGINE
// ============================================

export class RiskEngine {
  constructor(provider) {
    this.provider = provider;
    this.entropyAnalyzer = new EntropyAnalyzer();
    this.adjacencyAnalyzer = new AdjacencyAnalyzer();
    this.bayesianNetwork = new BayesianRiskNetwork();
    this.cache = new Map(); // Simple in-memory cache
```

```
    this.knownProtocols = { // Add known addresses for major protocols
        '0x7a250d5630b4cf539739df2c5dacb4c659f2488d': 'Uniswap V2',
        '0x1f9840a85d5af5bf1d1762f925bdaddc4201f984': 'Uniswap Token',
        '0xdac17f958d2ee523a2206206994597c13d831ec7': 'Tether (USDT)',
        '0xa0b86991c6218b36c1d19d4a2e9eb0ce3606eb48': 'USD Coin
(USDC)',
        '0x777777c9898d384f785ee44acfe945efdff5f3e0': 'Tornado.Cash',
        // CEX deposit wallets would be added here
        '0x28c6c06298d514db089934071355e5743bf21d60': 'Binance',
    };
  }

  async analyzeWallet(address) {
    const cached = this.cache.get(address);
    if (cached && Date.now() - cached.timestamp < 3600000) { // 1 hour
cache
      return cached.data;
    }

    try {
      // Fetch wallet data in parallel
      const [transactionData, adjacencyData, onChainMetadata] = await
Promise.all([
        this.fetchTransactionData(address),
        this.adjacencyAnalyzer.mapAdjacencyGraph(address,
this.provider),
        this.fetchOnChainMetadata(address)
      ]);

      // Compute entropy metrics
      const entropyScore =
this.entropyAnalyzer.computeCompositeEntropy({
        timestamps: transactionData.timestamps,
        values: transactionData.values,
        protocols: transactionData.protocols,
        gasPrices: transactionData.gasPrices,
        historicalEntropies: [] // Would be populated from historical
analysis
      });

      // Bayesian risk inference
      const bayesianRisk = await this.bayesianNetwork.infer({
        walletAge:
this.categorizeWalletAge(onChainMetadata.firstSeen),
        fundingSource:
this.categorizeFundingSource(transactionData.fundingSources)
      });
```

```
      // Combine all risk factors
      const overallRisk = this.combineRiskFactors({
        entropy: entropyScore,
        adjacency: adjacencyData,
        bayesian: bayesianRisk
      });

      // Generate report
      const report = {
        address,
        timestamp: Date.now(),
        overallRisk,
        components: {
          entropy: entropyScore,
          adjacency: adjacencyData,
          bayesian: bayesianRisk
        },
        interpretation: this.interpretRisk(overallRisk, { adjacency:
adjacencyData, bayesian: bayesianRisk }),
        recommendations: this.generateRecommendations(overallRisk, {
entropyScore, adjacencyData, bayesianRisk }),
        visualizations: {
          entropyGlyph: this.generateEntropyGlyph(entropyScore),
          riskTimeline: this.generateRiskTimeline(transactionData,
overallRisk.score)
        }
      };

      // Cache result
      this.cache.set(address, { timestamp: Date.now(), data: report
});
      return report;

    } catch (error) {
      console.error('Risk analysis failed:', error);
      throw new Error(`Failed to analyze wallet ${address}:
${error.message}`);
    }
  }

  async fetchTransactionData(address) {
    const history = await this.provider.getHistory(address);
    if (!history) return { timestamps: [], values: [], protocols: [],
gasPrices: [], fundingSources: [] };

    const transactions = history.slice(0, 100); // Analyze last 100
txs
```

```
    const values = [];
    const timestamps = [];
    const protocols = [];
    const gasPrices = [];

    for(const tx of transactions) {
        if(tx.from.toLowerCase() !== address.toLowerCase()) continue;
// Only consider outgoing transactions for behavior

        values.push(parseFloat(ethers.formatEther(tx.value)));
        if (tx.timestamp) timestamps.push(tx.timestamp);
        protocols.push({ protocol: this.identifyProtocol(tx.to) });
        if(tx.gasPrice)
gasPrices.push(parseFloat(ethers.formatUnits(tx.gasPrice, 'gwei')));
    }

    return {
      timestamps,
      values,
      protocols,
      gasPrices,
      fundingSources: this.identifyFundingSources(history, address)
    };
  }

  async fetchOnChainMetadata(address) {
    const [balance, txCount, firstTx] = await Promise.all([
        this.provider.getBalance(address),
        this.provider.getTransactionCount(address, 'latest'),
        this.provider.getHistory(address, 0, 1) // Fetch the very
first transaction
    ]);

    const firstSeen = firstTx.length > 0 && firstTx[0].timestamp
        ? firstTx[0].timestamp * 1000 // Convert seconds to ms
        : Date.now();

    return {
      balance: parseFloat(ethers.formatEther(balance)),
      transactionCount: txCount,
      firstSeen
    };
  }

  identifyProtocol(address) {
    if (!address) return 'EOA Interaction';
    return this.knownProtocols[address.toLowerCase()] || 'Unknown
Contract';
```

```
  }

  identifyFundingSources(transactions, walletAddress) {
    const sources = new Set();
    // Look at the first 5 incoming transactions
    transactions.filter(tx => tx.to?.toLowerCase() ===
walletAddress.toLowerCase())
        .slice(0, 5)
        .forEach(tx => {
            sources.add(this.identifyProtocol(tx.from));
        });
    return Array.from(sources);
  }

  categorizeWalletAge(firstSeenTimestamp) {
    const ageInDays = (Date.now() - firstSeenTimestamp) / (1000 * 60 *
60 * 24);
    if (ageInDays < 30) return 'new';
    if (ageInDays < 365) return 'established';
    return 'veteran';
  }

  categorizeFundingSource(sources) {
    if (sources.some(s => s.toLowerCase().includes('tornado'))) return
'mixer';
    if (sources.some(s => s.toLowerCase().includes('binance') ||
s.toLowerCase().includes('coinbase'))) return 'cex';
    if (sources.some(s => s.toLowerCase().includes('uniswap') ||
s.toLowerCase().includes('aave'))) return 'defi';
    return 'unlabeled';
  }

  combineRiskFactors({ entropy, adjacency, bayesian }) {
    // Convert Bayesian state to a numeric score (0-1)
    const bayesianScoreMap = { low: 0.1, medium: 0.5, high: 0.9 };
    const bayesianScore =
bayesianScoreMap[bayesian.maxLikelihoodState];

    // Normalize composite entropy (assuming a practical max of ~4.5)
    const normalizedEntropy = Math.min(entropy.composite / 4.5, 1.0);

    // Adjacency risk is already 0-1
    const adjacencyRisk = Math.min(adjacency.totalRisk, 1.0);

    // Define weights for each component
    const weights = {
      bayesian: 0.5,
      adjacency: 0.3,
```

```
      entropy: 0.2
    };

    // Calculate final weighted score
    let score = (bayesianScore * weights.bayesian) +
                (adjacencyRisk * weights.adjacency) +
                (normalizedEntropy * weights.entropy);

    // If there's a direct link to a known bad actor, elevate risk
significantly
    if (adjacency.flaggedConnections.some(c => c.risk === 1.0 &&
c.depth === 1)) {
        score = Math.max(score, 0.95);
    }

    return {
      score: Math.min(score, 1.0) * 100, // Scale to 0-100
    };
  }

  interpretRisk(overallRisk, { adjacency, bayesian }) {
    const score = overallRisk.score;
    if (score > 85) return `CRITICAL RISK: Wallet exhibits strong
indicators of illicit activity. Direct link to sanctioned or known
fraudulent addresses likely.`;
    if (score > 60) return `HIGH RISK: Behavior is highly anomalous or
shows significant connection to high-risk counterparties
(${bayesian.maxLikelihoodState} risk profile).`;
    if (score > 35) return `MODERATE RISK: Wallet shows some unusual
patterns or indirect links to risky addresses. Caution is advised.`;
    if (score > 10) return `LOW RISK: Wallet behavior appears normal
with no significant risk factors detected.`;
    return `VERY LOW RISK: Wallet is established and consistently
interacts with reputable protocols.`;
  }

  generateRecommendations(overallRisk, { entropyScore, adjacencyData,
bayesianRisk }) {
    const recommendations = new Set();
    const score = overallRisk.score;

    if (score > 60) {
        recommendations.add("URGENT: Manual review of transaction
history is required.");
        recommendations.add("Consider freezing funds or flagging the
address pending investigation.");
    } else if (score > 35) {
        recommendations.add("Monitor address for further suspicious
```

```
activity.");
        recommendations.add("Advise user to be cautious with
interactions from this address.");
    } else {
        recommendations.add("No immediate action required. Continue
standard monitoring.");
    }

    if (adjacencyData.flaggedConnections.length > 0) {
        recommendations.add(`Review flagged connections:
${adjacencyData.flaggedConnections.map(c =>
c.address).slice(0,2).join(', ')}...`);
    }

    if (entropyScore.composite > 3.5) {
        recommendations.add("High entropy suggests potential
obfuscation techniques; investigate transaction patterns.");
    }

    if (bayesianRisk.maxLikelihoodState === 'high') {
        recommendations.add(`Bayesian analysis indicates a high-risk
profile, likely funded via mixer or unlabeled sources.`);
    }

    return Array.from(recommendations);
  }

  generateEntropyGlyph(entropyScore) {
    // Generates an SVG string for a simple radar chart (glyph)
    const components = entropyScore.components;
    const size = 100;
    const center = size / 2;
    const maxEntropy = 4; // Normalized max value for visualization
    const points = Object.values(components).map((value, i, arr) => {
        const angle = (i / arr.length) * 2 * Math.PI;
        const normalizedValue = Math.min(value / maxEntropy, 1.0);
        const x = center + center * 0.8 * normalizedValue *
Math.cos(angle - Math.PI / 2);
        const y = center + center * 0.8 * normalizedValue *
Math.sin(angle - Math.PI / 2);
        return `${x},${y}`;
    }).join(' ');

    return `<svg width="${size}" height="${size}" viewBox="0 0 ${size}
${size}">
        <circle cx="${center}" cy="${center}" r="${center*0.8}"
fill="none" stroke="#4a5568" stroke-width="1"/>
        <circle cx="${center}" cy="${center}" r="${center*0.4}"
```

```
fill="none" stroke="#4a5568" stroke-width="0.5"/>
        <polygon points="${points}" fill="#e53e3e" fill-opacity="0.6"
stroke="#c53030" stroke-width="1"/>
    </svg>`;
  }

  generateRiskTimeline(transactionData, finalRisk) {
    // Creates a simplified time series of value flow
    const timeline = transactionData.timestamps.map((ts, i) => ({
      time: ts,
      value: transactionData.values[i],
      type: 'transaction'
    }));
    timeline.push({ time: Date.now() / 1000, value: finalRisk, type:
'current_risk_score' });
    return timeline.sort((a,b) => a.time - b.time);
  }
}
```