Shamir's Secret Sharing Algorithm - Complete Implementation

Overview

Shamir's Secret Sharing (SSS) is a cryptographic algorithm that splits a secret into multiple shares, where a minimum threshold of shares is required to reconstruct the original secret. This implementation handles high-precision numbers, share verification, and wrong share detection.

Core Algorithm Components

1. Polynomial Generation

The secret is embedded as the constant term (a_0) of a polynomial:

$f(x) = a_0 + a_1x + a_2x^2 + ... + a_{k-1}x^{(k-1)}$

Where:

- a_0 = secret
- $a_1, a_2, ..., a_{k-1}$ = random coefficients
- k = minimum shares required (threshold)

2. Share Generation Process

```
For i = 1 to n:
  share_i = (i, f(i))
  hash_i = SHA256(i + "," + f(i))
```

3. Secret Reconstruction using Lagrange Interpolation

secret = $f(0) = \Sigma(i=1 \text{ to } k)\ y_i \times L_i(0)$

Where Lagrange basis polynomial:

$L_i(0) = \Pi(j=1 \text{ to } k, j \neq i)\ (-x_j) / (x_i - x_j)$

Implementation Steps

Step 1: High-Precision Arithmetic Functions

javascript

```javascript
// Custom precision operations
function preciseAdd(a, b) {
```

```
    return parseFloat((parseFloat(a) + parseFloat(b)).toFixed(15));
}

function preciseMultiply(a, b) {
    return parseFloat((parseFloat(a) * parseFloat(b)).toFixed(15));
}

function preciseDivide(a, b) {
    if (Math.abs(b) < 1e-15) throw new Error("Division by zero");
    return parseFloat((parseFloat(a) / parseFloat(b)).toFixed(15));
}
```

Step 2: Polynomial Evaluation

javascript
```
function evaluatePolynomial(coefficients, x) {
    let result = 0;
    for (let i = 0; i < coefficients.length; i++) {
        result += coefficients[i] * Math.pow(x, i);
    }
    return parseFloat(result.toFixed(15));
}
```

Step 3: Share Generation

javascript
```
function generateShares(secret, n, k) {
    // Generate k-1 random coefficients
    const coefficients = [parseFloat(secret)];
    for (let i = 1; i < k; i++) {
        coefficients.push(Math.random() * 1000);
    }

    // Generate n shares
    const shares = [];
    for (let x = 1; x <= n; x++) {
        const y = evaluatePolynomial(coefficients, x);
        const hash = createHash(x, y);
        shares.push({x: x, y: y, hash: hash});
    }
    return shares;
}
```

Step 4: Lagrange Interpolation for Reconstruction

javascript
```
function lagrangeInterpolation(shares) {
```

```
    let secret = 0;
    const n = shares.length;

    for (let i = 0; i < n; i++) {
        let xi = shares[i].x;
        let yi = shares[i].y;

        // Calculate Lagrange basis polynomial Li(0)
        let li = 1;
        for (let j = 0; j < n; j++) {
            if (i !== j) {
                let xj = shares[j].x;
                li = preciseMultiply(li, preciseDivide(-xj, xi - xj));
            }
        }
        secret = preciseAdd(secret, preciseMultiply(yi, li));
    }
    return secret;
}
```

Share Verification Methods

1. Vandermonde Matrix Method

javascript

```
function createVandermondeMatrix(xValues, k) {
    const matrix = [];
    for (let i = 0; i < xValues.length; i++) {
        const row = [];
        for (let j = 0; j < k; j++) {
            row.push(Math.pow(xValues[i], j));
        }
        matrix.push(row);
    }
    return matrix;
}

function calculateDeterminant(matrix) {
    // Recursive determinant calculation
    const n = matrix.length;
    if (n === 1) return matrix[0][0];
    if (n === 2) return matrix[0][0] * matrix[1][1] - matrix[0][1] * matrix[1][0];

    let det = 0;
    for (let i = 0; i < n; i++) {
        const subMatrix = getSubMatrix(matrix, 0, i);
        const cofactor = matrix[0][i] * calculateDeterminant(subMatrix);
```

```
        det += (i % 2 === 0) ? cofactor : -cofactor;
    }
    return det;
}
```

2. Wrong Share Detection Algorithm

```javascript
function findWrongShares(shares, k) {
    const combinations = getCombinations(shares, k);
    const reconstructionResults = [];

    // Test all k-combinations
    for (let combo of combinations) {
        try {
            const secret = lagrangeInterpolation(combo);
            reconstructionResults.push({
                shares: combo,
                secret: secret.toFixed(10)
            });
        } catch (error) {
            // Invalid combination
        }
    }

    // Find consensus
    const secretCounts = {};
    for (let result of reconstructionResults) {
        if (!secretCounts[result.secret]) {
            secretCounts[result.secret] = [];
        }
        secretCounts[result.secret].push(result);
    }

    // Identify most common result
    let mostCommon = null;
    let maxCount = 0;
    for (let [secret, results] of Object.entries(secretCounts)) {
        if (results.length > maxCount) {
            maxCount = results.length;
            mostCommon = secret;
        }
    }

    return {
        correctSecret: mostCommon,
        validCombinations: maxCount,
```

```
        wrongShares: identifyWrongShares(shares, secretCounts[mostCommon])
    };
}
```

Configuration Parameters

Standard Configuration (n=4, k=3):

- **Total Shares (n):** 4
- **Minimum Required (k):** 3
- **Secret Precision:** Up to 20 digits with decimals
- **Hash Algorithm:** SHA-256

JSON Share Format:

```json
[
    {"x": 1, "y": 125.789012345678901},
    {"x": 2, "y": 251.234567890123456},
    {"x": 3, "y": 402.567890123456789},
    {"x": 4, "y": 579.901234567890123}
]
```

Error Detection Logic

Combination Testing Process:

1. Generate all possible k-combinations from available shares
2. Reconstruct secret for each combination using Lagrange interpolation
3. Group results by reconstructed secret value (rounded to 10 decimal places)
4. Find the most frequent result (consensus)
5. Mark shares not participating in consensus combinations as potentially wrong

Validation Criteria:

- **Matrix Determinant ≠ 0:** Ensures shares are linearly independent
- **Consensus Threshold:** Majority of combinations produce same result
- **Hash Verification:** SHA-256 integrity check for each share
- **Precision Tolerance:** Allow small floating-point errors (< 1e-10)

Usage Process

1. Generate Shares:

Input: secret = "123.456789012345"

n = 4, k = 3
Process: Generate polynomial, evaluate at x=1,2,3,4

Output: 4 shares with x,y coordinates and hashes

2. Verify and Reconstruct:

Input: JSON array of shares
Process: Apply Lagrange interpolation

Output: Reconstructed secret with error analysis

3. Detect Wrong Shares:

Input: Potentially corrupted shares
Process: Test all k-combinations, find consensus

Output: Identification of valid/invalid shares

Mathematical Foundation

Polynomial Mathematics:

- **Degree:** k-1 (one less than threshold)
- **Uniqueness:** Any k points uniquely determine a polynomial of degree k-1
- **Security:** Fewer than k shares reveal no information about the secret

Linear Algebra:

- **Vandermonde Matrix:** Ensures share uniqueness
- **Determinant Test:** Non-zero determinant confirms linear independence
- **Matrix Inversion:** Required for coefficient recovery

Precision Handling:

- **Floating Point:** 15-decimal precision to handle 20-digit secrets
- **Error Tolerance:** 1e-10 threshold for equality comparisons
- **Rounding Strategy:** Consistent rounding to avoid accumulation errors