# Complete ZKP Biometric Voting System Implementation

## Directory Structure

### Server-Side Directory (VPS/Remote Server)

```
zkp_biometric_voting_server/
│
├── README.md
├── requirements.txt
├── app.py                           # Main Flask application
├── config/
│   ├── server_config.yaml
│   └── database_config.yaml
│
├── core/                            # Original voting system components
│   ├── __init__.py
│   ├── pqc_crypto.py               # Post-quantum cryptography
│   ├── zkp_module.py               # Zero-knowledge proofs
│   ├── biometric_auth.py           # Biometric authentication
│   ├── blockchain_voting.py        # Original blockchain
│   └── voting_system.py            # Main voting system
│
├── server/                          # Server-specific components
│   ├── __init__.py
│   ├── server_real_data.py         # Real fingerprint data handler
│   ├── server_auth.py              # Authentication server
│   ├── server_homomorphic.py       # Homomorphic processing
│   └── enhanced_blockchain.py      # Enhanced blockchain with HE
│
├── testing/                         # Dataset integration
│   ├── __init__.py
│   └── dataset_handler.py          # SOCOFing dataset handler
│
├── dataset/                         # SOCOFing dataset (server side)
│   ├── Real/                       # Only real fingerprints
│   │   ├── 1/
│   │   ├── 2/
│   │   └── ...
│   └── metadata/
│       └── socofing_metadata.db
│
├── storage/                         # Server storage
│   ├── blockchain_data/
│   ├── voter_registry/
│   └── logs/
│
└── scripts/
    ├── setup_server.sh
    └── start_server.py
```

### Client-Side Directory (Local Testing)

```
zkp_biometric_voting_client/
│
├── README.md
├── requirements.txt
├── main.py                         # Main client application
├── config/
│   └── client_config.yaml
│
├── core/                           # Same as server core
│   ├── __init__.py
│   ├── pqc_crypto.py
│   ├── zkp_module.py
│   ├── biometric_auth.py
│   ├── blockchain_voting.py
│   └── voting_system.py
│
├── client/                         # Client-specific components
│   ├── __init__.py
│   ├── client_biometric.py
│   ├── client_zkp.py
│   ├── homomorphic_client.py       # Client homomorphic encryption
│   ├── client_test_data.py
│   ├── enhanced_voting_system.py
│   └── complete_voting_flow.py
│
├── testing/                        # Testing framework
│   ├── __init__.py
│   ├── dataset_handler.py
│   ├── test_voting_system.py
│   ├── test_with_socofing.py
│   └── security_analysis.py
│
├── dataset/                        # SOCOFing dataset (client side)
│   ├── Altered-Easy/
│   ├── Altered-Medium/
│   ├── Altered-Hard/
│   ├── Synthetic/
│   └── metadata/
│
├── results/                        # Test results
│   ├── authentication_results/
│   ├── voting_results/
│   ├── performance_metrics/
│   └── reports/
│
└── scripts/
```

```
    ├── setup_client.sh
    └── run_tests.py
```

## COMPLETE CODE IMPLEMENTATION

## 1. Original Core Components (Both Server & Client)

### core/pqc_crypto.py

```
# pqc_crypto.py - Post-Quantum Cryptographic Implementation
import hashlib
import secrets
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import rsa, padding
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
import numpy as np

class PQCCrypto:
    """Post-Quantum Cryptography implementation using CRYSTALS-Kyber and Dilithium concep

    def __init__(self):
        self.n = 256  # Polynomial degree
        self.q = 3329  # Modulus
        self.eta = 2   # Noise parameter

    def generate_kyber_keypair(self):
        """Generate CRYSTALS-Kyber-like key pair for encryption"""
        # Simplified implementation - in production, use actual CRYSTALS-Kyber
        private_key = np.random.randint(0, self.q, size=self.n)
        public_key = (private_key * 7 + np.random.randint(-self.eta, self.eta+1, size=sel
        return private_key, public_key

    def kyber_encrypt(self, public_key, message):
        """Encrypt using Kyber-like scheme"""
        r = np.random.randint(0, self.q, size=self.n)
        e1 = np.random.randint(-self.eta, self.eta+1, size=self.n)
        e2 = np.random.randint(-self.eta, self.eta+1)

        u = (public_key * r + e1) % self.q
        v = (np.sum(public_key * r) + e2 + message * (self.q // 2)) % self.q
        return u, v

    def kyber_decrypt(self, private_key, ciphertext):
        """Decrypt using Kyber-like scheme"""
        u, v = ciphertext
        message = (v - np.sum(private_key * u)) % self.q
        return 1 if message > self.q // 4 else 0

    def generate_dilithium_keypair(self):
        """Generate CRYSTALS-Dilithium-like key pair for signatures"""
        # Simplified implementation
        private_key = np.random.randint(-2, 3, size=self.n)
        public_key = (private_key * 5) % self.q
        return private_key, public_key

    def dilithium_sign(self, private_key, message):
        """Sign using Dilithium-like scheme"""
        hash_msg = hashlib.sha256(message.encode()).hexdigest()
        # Simplified signature generation
        signature = (private_key * int(hash_msg[:8], 16)) % self.q
        return signature

    def dilithium_verify(self, public_key, message, signature):
        """Verify Dilithium-like signature"""
        hash_msg = hashlib.sha256(message.encode()).hexdigest()
        expected = (public_key * int(hash_msg[:8], 16)) % self.q
        return np.array_equal(signature % self.q, expected % self.q)
```

### core/zkp_module.py

```
# zkp_module.py - Zero-Knowledge Proof Implementation
import hashlib
import random
from typing import Dict, Any, Tuple

class ZKProofSystem:
    """Zero-Knowledge Proof system for voter eligibility and vote privacy"""

    def __init__(self):
        self.commitment_schemes = {}
        self.proof_cache = {}

    def generate_commitment(self, value: int, randomness: int = None) -> Dict[str, Any]:
        """Generate Pedersen commitment for vote privacy"""
        if randomness is None:
            randomness = random.randint(1, 2**256)

        # Simplified Pedersen commitment: C = g^v * h^r
        g = 2  # Generator
        h = 3  # Another generator
        p = 2**256 - 189  # Large prime

        commitment = (pow(g, value, p) * pow(h, randomness, p)) % p

        return {
            'commitment': commitment,
            'randomness': randomness,
            'value': value
        }

    def generate_eligibility_proof(self, voter_id: str, biometric_hash: str) -> Dict[str,
        """Generate ZK proof of voter eligibility without revealing identity"""
        # Create proof that voter is in eligible set without revealing which voter
        nonce = random.randint(1, 2**128)

        # Hash-based proof construction
        proof_input = f"{voter_id}:{biometric_hash}:{nonce}"
        proof_hash = hashlib.sha256(proof_input.encode()).hexdigest()

        return {
            'proof_hash': proof_hash,
            'nonce': nonce,
            'timestamp': hashlib.sha256(str(random.time()).encode()).hexdigest()[:16]
        }

    def generate_vote_proof(self, vote: int, commitment_data: Dict) -> Dict[str, Any]:
        """Generate ZK proof that vote is valid (0 or 1) without revealing the vote"""
        if vote not in [0, 1]:
            raise ValueError("Vote must be 0 or 1")

        # Generate proof that committed value is either 0 or 1
        challenge = random.randint(1, 2**128)

        if vote == 0:
            response_0 = random.randint(1, 2**128)
            response_1 = challenge - response_0
        else:
            response_1 = random.randint(1, 2**128)
            response_0 = challenge - response_1

        return {
            'challenge': challenge,
            'response_0': response_0,
            'response_1': response_1,
            'commitment': commitment_data['commitment']
        }

    def verify_eligibility_proof(self, proof: Dict, voter_registry: set) -> bool:
        """Verify voter eligibility proof"""
        # Simplified verification - in production, use more sophisticated methods
        return proof.get('proof_hash') is not None and len(proof.get('proof_hash', '')) =

    def verify_vote_proof(self, proof: Dict) -> bool:
        """Verify that the vote proof is valid"""
        return (proof.get('challenge') == proof.get('response_0') + proof.get('response_1
```

### core/biometric_auth.py

```
# biometric_auth.py - Biometric Authentication System
import hashlib
import cv2
import numpy as np
from sklearn.metrics import accuracy_score
import pickle

class BiometricAuth:
    """Biometric authentication using fingerprint and iris recognition"""

    def __init__(self):
        self.fingerprint_templates = {}
        self.iris_templates = {}
        self.threshold = 0.85
    def extract_fingerprint_features(self, fingerprint_image):
        """Extract features from fingerprint image"""
        # Simplified feature extraction - in production, use proper minutiae extraction
        if isinstance(fingerprint_image, str):
            # If path provided, load image
            img = cv2.imread(fingerprint_image, cv2.IMREAD_GRAYSCALE)
        else:
            img = fingerprint_image

        # Basic feature extraction using histogram
        hist = cv2.calcHist([img], [0], None, [256], [0, 256])
        features = hist.flatten()

        # Add edge detection features
        edges = cv2.Canny(img, 50, 150)
        edge_hist = cv2.calcHist([edges], [0], None, [256], [0, 256])

        combined_features = np.concatenate([features, edge_hist.flatten()])
        return combined_features / np.linalg.norm(combined_features)

    def extract_iris_features(self, iris_image):
        """Extract features from iris image"""
        if isinstance(iris_image, str):
            img = cv2.imread(iris_image, cv2.IMREAD_GRAYSCALE)
        else:
            img = iris_image

        # Basic iris feature extraction using circular patterns
        center = (img.shape[1]//2, img.shape[0]//2)

        # Extract concentric circles
        features = []
        for radius in range(10, min(img.shape)//2, 5):
            circle_mask = np.zeros(img.shape, dtype=np.uint8)
            cv2.circle(circle_mask, center, radius, 255, 1)
            circle_pixels = img[circle_mask == 255]
            if len(circle_pixels) > 0:
                features.extend([circle_pixels.mean(), circle_pixels.std()])

        return np.array(features)

    def register_biometric(self, user_id: str, fingerprint_data, iris_data):
        """Register biometric data for a user"""
        fp_features = self.extract_fingerprint_features(fingerprint_data)
        iris_features = self.extract_iris_features(iris_data)

        # Hash the features for storage
        fp_hash = hashlib.sha256(fp_features.tobytes()).hexdigest()
        iris_hash = hashlib.sha256(iris_features.tobytes()).hexdigest()

        self.fingerprint_templates[user_id] = {
            'features': fp_features,
            'hash': fp_hash
        }
        self.iris_templates[user_id] = {
            'features': iris_features,
            'hash': iris_hash
        }

        return fp_hash, iris_hash

    def authenticate_biometric(self, fingerprint_data, iris_data) -> Tuple[bool, str]:
        """Authenticate user using biometric data"""
        fp_features = self.extract_fingerprint_features(fingerprint_data)
        iris_features = self.extract_iris_features(iris_data)

        best_match_id = None
        best_score = 0

        for user_id in self.fingerprint_templates:
            # Calculate similarity scores
            fp_similarity = np.dot(fp_features, self.fingerprint_templates[user_id]['feat
            iris_similarity = np.dot(iris_features, self.iris_templates[user_id]['feature

            # Combined score
            combined_score = (fp_similarity + iris_similarity) / 2

            if combined_score > best_score:
                best_score = combined_score
                best_match_id = user_id

        authenticated = best_score >= self.threshold
        return authenticated, best_match_id if authenticated else None

    def generate_biometric_proof(self, user_id: str) -> str:
        """Generate biometric proof for ZKP system"""
        if user_id in self.fingerprint_templates and user_id in self.iris_templates:
            combined_hash = hashlib.sha256(
                f"{self.fingerprint_templates[user_id]['hash']}"
                f"{self.iris_templates[user_id]['hash']}".encode()
            ).hexdigest()
            return combined_hash
        return None
```

### core/blockchain_voting.py

```
# blockchain_voting.py - Blockchain-based Voting System
import hashlib
import json
import time
from typing import List, Dict, Any
import secrets

class Block:
    """Individual block in the voting blockchain"""

    def __init__(self, index: int, transactions: List[Dict], previous_hash: str):
        self.index = index
        self.timestamp = time.time()
        self.transactions = transactions
        self.previous_hash = previous_hash
        self.nonce = 0
        self.hash = self.calculate_hash()

    def calculate_hash(self) -> str:
        """Calculate block hash"""
        block_string = f"{self.index}{self.timestamp}{json.dumps(self.transactions, sort_
        return hashlib.sha256(block_string.encode()).hexdigest()

    def mine_block(self, difficulty: int = 4):
        """Mine block with proof of work"""
        target = "0" * difficulty
        while self.hash[:difficulty] != target:
            self.nonce += 1
            self.hash = self.calculate_hash()

class VotingBlockchain:
    """Blockchain implementation for secure voting"""

    def __init__(self):
        self.chain = [self.create_genesis_block()]
        self.pending_transactions = []
        self.difficulty = 2
        self.voting_results = {}

    def create_genesis_block(self) -> Block:
        """Create the first block in the chain"""
        return Block(0, [], "0")

    def get_latest_block(self) -> Block:
        """Get the most recent block"""
        return self.chain[-1]

    def add_vote_transaction(self, voter_id: str, vote_data: Dict, zkp_proof: Dict, pqc_s
        """Add a vote transaction to pending transactions"""
        transaction = {
            'type': 'VOTE',
            'voter_id_hash': hashlib.sha256(voter_id.encode()).hexdigest(),  # Anonymous
            'vote_commitment': vote_data['commitment'],
            'zkp_proof': zkp_proof,
            'pqc_signature': pqc_signature,
            'timestamp': time.time(),
            'transaction_id': secrets.token_hex(16)
        }
        self.pending_transactions.append(transaction)
        return transaction['transaction_id']

    def mine_pending_transactions(self):
        """Mine pending transactions into a new block"""
        if not self.pending_transactions:
            return None

        block = Block(
            len(self.chain),
            self.pending_transactions.copy(),
            self.get_latest_block().hash
        )

        block.mine_block(self.difficulty)
        self.chain.append(block)
        self.pending_transactions = []
        return block

    def verify_vote_transaction(self, transaction: Dict, zkp_system) -> bool:
        """Verify a vote transaction"""
        # Verify ZKP proof
        if not zkp_system.verify_vote_proof(transaction['zkp_proof']):
            return False

        # Check for double voting
        voter_hash = transaction['voter_id_hash']
        for block in self.chain:
            for tx in block.transactions:
                if tx.get('voter_id_hash') == voter_hash and tx.get('type') == 'VOTE':
                    return False  # Double voting detected

        return True

    def tally_votes(self, zkp_system) -> Dict:
        """Tally votes from the blockchain"""
        vote_count = {'yes': 0, 'no': 0, 'total': 0}

        for block in self.chain[1:]:  # Skip genesis block
            for transaction in block.transactions:
                if transaction.get('type') == 'VOTE':
                    # In a real implementation, votes would be decrypted/revealed here
                    # For demonstration, we'll simulate vote counting
                    vote_count['total'] += 1
                    # This would typically involve homomorphic tallying or MPC

        return vote_count

    def validate_blockchain(self) -> bool:
        """Validate the entire blockchain"""
        for i in range(1, len(self.chain)):
            current_block = self.chain[i]
            previous_block = self.chain[i-1]

            if current_block.hash != current_block.calculate_hash():
                return False

            if current_block.previous_hash != previous_block.hash:
                return False

        return True
```

### core/voting_system.py

```
# voting_system.py - Main voting system integration
import json
from typing import Dict, Any, Optional
import logging
import hashlib
import time

class SecureVotingSystem:
    """Main secure voting system integrating all components"""

    def __init__(self):
        from pqc_crypto import PQCCrypto
        from zkp_module import ZKProofSystem
        from biometric_auth import BiometricAuth
        from blockchain_voting import VotingBlockchain

        self.pqc_crypto = PQCCrypto()
        self.zkp_system = ZKProofSystem()
        self.biometric_auth = BiometricAuth()
        self.blockchain = VotingBlockchain()
        self.registered_voters = set()
        self.election_active = False

        # Generate system keys
        self.system_private_key, self.system_public_key = self.pqc_crypto.generate_dilith

        logging.basicConfig(level=logging.INFO)
        self.logger = logging.getLogger(__name__)

    def register_voter(self, voter_id: str, fingerprint_data, iris_data) -> Dict[str, Any
        """Register a new voter with biometric data"""
        try:
            # Register biometric data
            fp_hash, iris_hash = self.biometric_auth.register_biometric(voter_id, fingerp

            # Add to registered voters
            self.registered_voters.add(voter_id)

            # Generate voter credential
            biometric_proof = self.biometric_auth.generate_biometric_proof(voter_id)
            eligibility_proof = self.zkp_system.generate_eligibility_proof(voter_id, biom

            self.logger.info(f"Voter {voter_id} registered successfully")

            return {
                'status': 'success',
                'voter_id': voter_id,
                'eligibility_proof': eligibility_proof,
                'fingerprint_hash': fp_hash,
                'iris_hash': iris_hash
            }

        except Exception as e:
            self.logger.error(f"Voter registration failed: {str(e)}")
            return {'status': 'error', 'message': str(e)}

    def cast_vote(self, fingerprint_data, iris_data, vote: int) -> Dict[str, Any]:
        """Cast a vote with biometric authentication"""
        if not self.election_active:
            return {'status': 'error', 'message': 'Election is not active'}

        try:
            # Authenticate voter
            authenticated, voter_id = self.biometric_auth.authenticate_biometric(fingerpr

            if not authenticated:
                return {'status': 'error', 'message': 'Biometric authentication failed'}

            # Generate vote commitment and proof
            commitment_data = self.zkp_system.generate_commitment(vote)
            vote_proof = self.zkp_system.generate_vote_proof(vote, commitment_data)

            # Generate eligibility proof
            biometric_proof = self.biometric_auth.generate_biometric_proof(voter_id)
            eligibility_proof = self.zkp_system.generate_eligibility_proof(voter_id, biom

            # Create PQC signature
            vote_message = f"{voter_id}:{commitment_data['commitment']}:{vote_proof['chal
            pqc_signature = self.pqc_crypto.dilithium_sign(self.system_private_key, vote_

            # Add to blockchain
            transaction_id = self.blockchain.add_vote_transaction(
                voter_id,
                commitment_data,
                vote_proof,
                {'signature': pqc_signature.tolist(), 'message': vote_message}
            )

            # Mine the block
            block = self.blockchain.mine_pending_transactions()

            self.logger.info(f"Vote cast successfully by voter {voter_id}")

            return {
                'status': 'success',
                'transaction_id': transaction_id,
                'block_hash': block.hash if block else None,
                'message': 'Vote cast successfully'
            }

        except Exception as e:
            self.logger.error(f"Vote casting failed: {str(e)}")
            return {'status': 'error', 'message': str(e)}

    def start_election(self) -> Dict[str, str]:
        """Start the election process"""
        self.election_active = True
        self.logger.info("Election started")
        return {'status': 'success', 'message': 'Election started'}
    def end_election_and_tally(self) -> Dict[str, Any]:
        """End election and tally votes"""
        self.election_active = False

        # Tally votes
        results = self.blockchain.tally_votes(self.zkp_system)

        # Validate blockchain integrity
        blockchain_valid = self.blockchain.validate_blockchain()

        self.logger.info("Election ended and votes tallied")

        return {
            'status': 'success',
            'results': results,
            'blockchain_valid': blockchain_valid,
            'total_blocks': len(self.blockchain.chain),
            'message': 'Election completed'
        }
```

## 2. Testing Framework Components

### testing/dataset_handler.py

```
# dataset_handler.py - SOCOFing dataset integration
import os
import cv2
import numpy as np
import json
import sqlite3
from pathlib import Path
import logging

logger = logging.getLogger(__name__)

class SOCOFingDatasetHandler:
    """Handler for SOCOFing dataset with real/altered/synthetic fingerprint classificatio

    def __init__(self, dataset_path: str):
        self.dataset_path = Path(dataset_path)
        self.db_path = self.dataset_path / "metadata" / "socofing_metadata.db"
        self.setup_database()

        # SOCOFing dataset structure
        self.categories = {
            'Real': 'Real',           # Original fingerprints
            'Altered-Easy': 'Altered',    # Slightly altered
            'Altered-Medium': 'Altered', # Moderately altered
            'Altered-Hard': 'Altered',   # Heavily altered
            'Synthetic': 'Synthetic'      # Completely synthetic
        }

    def setup_database(self):
        """Setup SQLite database for fingerprint metadata"""
        os.makedirs(self.db_path.parent, exist_ok=True)

        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()

        cursor.execute('''
        CREATE TABLE IF NOT EXISTS fingerprint_data (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            subject_id TEXT,
            finger_id TEXT,
            category TEXT,  -- Real, Altered, Synthetic
            subcategory TEXT, -- Easy, Medium, Hard for altered
            file_path TEXT,
            features_hash TEXT,
            is_server_data BOOLEAN,  -- True for real data on server
            created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
        )
        ''')

        cursor.execute('''
        CREATE TABLE IF NOT EXISTS subject_mapping (
            subject_id TEXT PRIMARY KEY,
            voter_id TEXT,
            real_fingerprints_count INTEGER,
            altered_fingerprints_count INTEGER,
            synthetic_fingerprints_count INTEGER
        )
        ''')

        conn.commit()
        conn.close()

    def parse_socofing_structure(self):
        """Parse SOCOFing dataset directory structure"""
        fingerprint_data = []

        for category_dir in self.dataset_path.iterdir():
            if not category_dir.is_dir() or category_dir.name == 'metadata':
                continue

            category_name = category_dir.name
            if category_name not in self.categories:
                continue

            for subject_dir in category_dir.iterdir():
                if not subject_dir.is_dir():
                    continue

                subject_id = subject_dir.name

                for fp_file in subject_dir.glob("*.png"):
                    # SOCOFing naming: subject_finger_session.png
                    filename_parts = fp_file.stem.split('_')
                    if len(filename_parts) >= 2:
                        finger_id = filename_parts[1] if len(filename_parts) > 1 else "ur
                        fingerprint_data.append({
                            'subject_id': subject_id,
                            'finger_id': finger_id,
                            'category': self.categories[category_name],
                            'subcategory': category_name,
                            'file_path': str(fp_file),
                            'is_server_data': category_name == 'Real'  # Only real data g
                        })

        return fingerprint_data

    def store_dataset_metadata(self):
        """Store dataset metadata in database"""
        fingerprint_data = self.parse_socofing_structure()

        if not fingerprint_data:
            logger.warning("No fingerprint data found in dataset")
            return

        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()

        # Clear existing data
        cursor.execute('DELETE FROM fingerprint_data')

        for fp_data in fingerprint_data:
            cursor.execute('''
            INSERT INTO fingerprint_data
            (subject_id, finger_id, category, subcategory, file_path, is_server_data)
            VALUES (?, ?, ?, ?, ?, ?)
            ''', (
                fp_data['subject_id'],
                fp_data['finger_id'],
                fp_data['category'],
                fp_data['subcategory'],
                fp_data['file_path'],
                fp_data['is_server_data']
            ))

        conn.commit()
        conn.close()

        logger.info(f"Stored {len(fingerprint_data)} fingerprint records in database")

    def get_category_counts(self):
        """Get count of fingerprints by category"""
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()

        cursor.execute('''
        SELECT category, subcategory, COUNT(*)
        FROM fingerprint_data
        GROUP BY category, subcategory
        ''')

        results = cursor.fetchall()
        conn.close()

        return {f"{cat}_{subcat}": count for cat, subcat, count in results}
```

### testing/test_voting_system.py

```
# test_voting_system.py - Comprehensive testing suite
import unittest
import numpy as np
import tempfile
import cv2
import sys
import os

sys.path.append(os.path.join(os.path.dirname(__file__), '..', 'core'))

from voting_system import SecureVotingSystem

class TestSecureVotingSystem(unittest.TestCase):
    """Test suite for the secure voting system"""

    def setUp(self):
        """Set up test environment"""
        self.voting_system = SecureVotingSystem()

        # Create mock biometric data
        self.mock_fingerprint = np.random.randint(0, 255, (200, 200), dtype=np.uint8)
        self.mock_iris = np.random.randint(0, 255, (150, 150), dtype=np.uint8)

    def test_pqc_key_generation(self):
        """Test post-quantum cryptographic key generation"""
        private_key, public_key = self.voting_system.pqc_crypto.generate_kyber_keypair()
        self.assertEqual(len(private_key), 256)
        self.assertEqual(len(public_key), 256)

        # Test encryption/decryption
        message = 1
        ciphertext = self.voting_system.pqc_crypto.kyber_encrypt(public_key, message)
        decrypted = self.voting_system.pqc_crypto.kyber_decrypt(private_key, ciphertext)
        self.assertEqual(message, decrypted)

    def test_biometric_registration(self):
        """Test biometric voter registration"""
        result = self.voting_system.register_voter("voter001", self.mock_fingerprint, sel
        self.assertEqual(result['status'], 'success')
        self.assertIn('voter001', self.voting_system.registered_voters)

    def test_zkp_generation_and_verification(self):
        """Test zero-knowledge proof generation and verification"""
        # Test commitment generation
        commitment = self.voting_system.zkp_system.generate_commitment(1)
        self.assertIn('commitment', commitment)
        self.assertIn('randomness', commitment)

        # Test vote proof
        vote_proof = self.voting_system.zkp_system.generate_vote_proof(1, commitment)
        verification = self.voting_system.zkp_system.verify_vote_proof(vote_proof)
        self.assertTrue(verification)

    def test_blockchain_integrity(self):
        """Test blockchain integrity and mining"""
        # Add some mock transactions
        self.voting_system.blockchain.add_vote_transaction(
            "voter001",
            {'commitment': 12345},
            {'challenge': 1, 'response_0': 0, 'response_1': 1},
            {'signature': [1, 2, 3], 'message': 'test'}
        )

        # Mine block
        block = self.voting_system.blockchain.mine_pending_transactions()
        self.assertIsNotNone(block)
        self.assertTrue(block.hash.startswith('0' * self.voting_system.blockchain.difficu

        # Validate blockchain
        self.assertTrue(self.voting_system.blockchain.validate_blockchain())

    def test_complete_voting_flow(self):
        """Test complete voting flow"""
        # Register voter
        registration = self.voting_system.register_voter("voter002", self.mock_fingerprir
        self.assertEqual(registration['status'], 'success')

        # Start election
        start_result = self.voting_system.start_election()
        self.assertEqual(start_result['status'], 'success')

        # Cast vote
        vote_result = self.voting_system.cast_vote(self.mock_fingerprint, self.mock_iris,
        self.assertEqual(vote_result['status'], 'success')

        # End election
        end_result = self.voting_system.end_election_and_tally()
        self.assertEqual(end_result['status'], 'success')
        self.assertTrue(end_result['blockchain_valid'])

if __name__ == '__main__':
    unittest.main()
```

### 3. Server-Side Components

### server/app.py

```
# server/app.py - Main Flask application
from flask import Flask, request, jsonify
from flask_cors import CORS
import logging
import os
import sys
import base64
import numpy as np
from datetime import datetime

# Add paths
sys.path.append(os.path.join(os.path.dirname(__file__), '..', 'core'))
sys.path.append(os.path.join(os.path.dirname(__file__), '..', 'testing'))

from voting_system import SecureVotingSystem
from dataset_handler import SOCOFingDatasetHandler
from server_real_data import ServerRealDataHandler
from server_homomorphic import HomomorphicVoteProcessor
from enhanced_blockchain import EnhancedVotingBlockchain

app = Flask(__name__)
CORS(app)

# Configuration
app.config['DATASET_PATH'] = os.environ.get('DATASET_PATH', './dataset')
app.config['DEBUG'] = os.environ.get('DEBUG', 'False').lower() == 'true'

# Setup logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# Global instances
voting_system = None
dataset_handler = None
server_handler = None
enhanced_blockchain = None

def initialize_system():
    """Initialize all system components"""
    global voting_system, dataset_handler, server_handler, enhanced_blockchain

    try:
        voting_system = SecureVotingSystem()
        dataset_handler = SOCOFingDatasetHandler(app.config['DATASET_PATH'])
        dataset_handler.store_dataset_metadata()

        server_handler = ServerRealDataHandler(dataset_handler, voting_system.biometric_a
        enhanced_blockchain = EnhancedVotingBlockchain()

        logger.info("Server system initialized successfully")
        return True

    except Exception as e:
        logger.error(f"System initialization failed: {str(e)}")
        return False

@app.before_first_request
def setup_system():
    """Setup system before first request"""
    os.makedirs('./storage/logs', exist_ok=True)
    if not initialize_system():
        logger.critical("Failed to initialize system")
```

```
@app.route('/health', methods=['GET'])
def health_check():
    """Health check endpoint"""
    return jsonify({
        'status': 'healthy',
        'timestamp': datetime.now().isoformat()
    })

@app.route('/api/authenticate_fingerprint', methods=['POST'])
def authenticate_fingerprint():
    """API endpoint for fingerprint authentication"""
    try:
        data = request.json
        if not data or 'features' not in data:
            return jsonify({'error': 'Missing fingerprint features'}), 400

        features_b64 = data['features']
        features_bytes = base64.b64decode(features_b64)
        features = np.frombuffer(features_bytes, dtype=np.float64)

        authenticated, voter_id = server_handler.authenticate_against_real_data(features)

        return jsonify({
            'authenticated': authenticated,
            'voter_id': voter_id,
            'server_data_source': 'SOCOFing_Real'
        })

    except Exception as e:
        return jsonify({'error': str(e)}), 400

@app.route('/api/register_voter', methods=['POST'])
def register_voter():
    """API endpoint for voter registration"""
    try:
        data = request.json
        if not data or 'subject_id' not in data or 'finger_id' not in data:
            return jsonify({'error': 'Missing subject_id or finger_id'}), 400

        result = server_handler.register_real_voter_from_dataset(
            data['subject_id'], data['finger_id']
        )
        return jsonify(result)

    except Exception as e:
        return jsonify({'error': str(e)}), 400

@app.route('/api/cast_vote', methods=['POST'])
def cast_vote():
    """API endpoint for casting encrypted vote"""
    try:
        data = request.json
        required_fields = ['voter_proof', 'encrypted_vote', 'zkp_proofs']
        if not all(field in data for field in required_fields):
            return jsonify({'error': 'Missing required fields'}), 400
        # Add to enhanced blockchain
        transaction_id = enhanced_blockchain.add_homomorphic_vote_transaction(
            data['voter_proof'],
            data['encrypted_vote'],
            data['zkp_proofs']
        )

        # Mine block
        block = enhanced_blockchain.mine_pending_transactions()

        return jsonify({
            'status': 'success',
            'transaction_id': transaction_id,
            'block_hash': block.hash if block else None
        })

    except Exception as e:
        return jsonify({'error': str(e)}), 400

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000, debug=app.config['DEBUG'])
```

### server/server_real_data.py

```
# server/server_real_data.py - Server-side real fingerprint data handler
import sqlite3
import hashlib
import numpy as np
import cv2
import os
import logging
from typing import Dict, Tuple, Optional

logger = logging.getLogger(__name__)

class ServerRealDataHandler:
    """Server-side handler for real fingerprint data storage and processing"""

    def __init__(self, dataset_handler, original_biometric_auth):
        self.dataset_handler = dataset_handler
        self.biometric_auth = original_biometric_auth
        self.real_templates = {}
        self.registered_voters = set()
        self.load_real_fingerprint_data()

    def load_real_fingerprint_data(self):
        """Load only real fingerprint data on server"""
        try:
            conn = sqlite3.connect(self.dataset_handler.db_path)
            cursor = conn.cursor()

            cursor.execute('''
            SELECT subject_id, finger_id, file_path
            FROM fingerprint_data
            WHERE category = 'Real' AND is_server_data = 1
            ''')

            real_data = cursor.fetchall()
            conn.close()

            logger.info(f"Loading {len(real_data)} real fingerprint templates")

            for subject_id, finger_id, file_path in real_data:
                try:
                    if not os.path.exists(file_path):
                        continue

                    img = cv2.imread(file_path, cv2.IMREAD_GRAYSCALE)
                    if img is not None:
                        features = self.biometric_auth.extract_fingerprint_features(img)

                        voter_id = f"voter_{subject_id}_{finger_id}"
                        self.real_templates[voter_id] = {
                            'features': features,
                            'hash': hashlib.sha256(features.tobytes()).hexdigest(),
                            'subject_id': subject_id,
                            'finger_id': finger_id
                        }

                except Exception as e:
                    logger.warning(f"Error processing {file_path}: {e}")

            logger.info(f"Successfully loaded {len(self.real_templates)} real templates")

        except Exception as e:
            logger.error(f"Failed to load real fingerprint data: {str(e)}")
            raise

    def authenticate_against_real_data(self, client_features: np.ndarray) -> Tuple[bool,
        """Authenticate client fingerprint against real server data"""
        try:
            best_match_id = None
            best_score = 0
            threshold = 0.85

            # Normalize features
            if np.linalg.norm(client_features) > 0:
                client_features = client_features / np.linalg.norm(client_features)

            for voter_id, template_data in self.real_templates.items():
                server_features = template_data['features']

                if np.linalg.norm(server_features) > 0:
                    server_features = server_features / np.linalg.norm(server_features)

                similarity = np.dot(client_features, server_features)

                if similarity > best_score:
                    best_score = similarity
                    best_match_id = voter_id
            authenticated = best_score >= threshold
            return authenticated, best_match_id if authenticated else None

        except Exception as e:
            logger.error(f"Authentication error: {str(e)}")
            return False, None

    def register_real_voter_from_dataset(self, subject_id: str, finger_id: str) -> Dict:
        """Register a real voter using dataset data"""
        try:
            voter_id = f"voter_{subject_id}_{finger_id}"

            if voter_id in self.real_templates:
                self.registered_voters.add(voter_id)

                return {
                    'status': 'success',
                    'voter_id': voter_id,
                    'template_hash': self.real_templates[voter_id]['hash'],
                    'data_source': 'SOCOFing_Real'
                }
            else:
                return {
                    'status': 'error',
                    'message': f'Real data not found for subject {subject_id}, finger {fi
                }

        except Exception as e:
            return {'status': 'error', 'message': str(e)}
```

### server/server_homomorphic.py

```
# server/server_homomorphic.py - Server-side homomorphic processing
import tenseal as ts
import numpy as np
import logging
from typing import Dict, Any

logger = logging.getLogger(__name__)

class HomomorphicVoteProcessor:
    """Server-side homomorphic vote processing for private tallying"""

    def __init__(self):
        self.context = None
        self.encrypted_tally = None
        self.vote_count = 0

    def initialize_context(self, public_context_bytes: bytes):
        """Initialize homomorphic context"""
        try:
            self.context = ts.context_from(public_context_bytes)
            self.encrypted_tally = ts.ckks_vector(self.context, [0.0])
            logger.info("Homomorphic context initialized")
            return True
        except Exception as e:
            logger.error(f"Context initialization failed: {str(e)}")
            return False

    def add_encrypted_vote(self, encrypted_vote_data: Any) -> bool:
        """Add encrypted vote to running tally"""
        try:
            if isinstance(encrypted_vote_data, dict) and 'serialized' in encrypted_vote_c
                encrypted_vote = ts.ckks_vector_from(self.context, encrypted_vote_data['s
            else:
                encrypted_vote = ts.ckks_vector_from(self.context, encrypted_vote_data)

            if self.encrypted_tally is None:
                self.encrypted_tally = encrypted_vote
            else:
                self.encrypted_tally = self.encrypted_tally + encrypted_vote

            self.vote_count += 1
            logger.info(f"Added encrypted vote {self.vote_count}")
            return True

        except Exception as e:
            logger.error(f"Failed to add encrypted vote: {str(e)}")
            return False

    def get_encrypted_results(self) -> Dict:
        """Return encrypted tallying results"""
        try:
            if self.encrypted_tally is None:
                return {'error': 'No encrypted tally available'}

            return {
                'encrypted_tally': self.encrypted_tally.serialize(),
                'total_votes': self.vote_count,
                'ready_for_decryption': True
            }

        except Exception as e:
            logger.error(f"Failed to get results: {str(e)}")
            return {'error': str(e)}

    def reset_tally(self):
        """Reset tally for new election"""
        if self.context:
            self.encrypted_tally = ts.ckks_vector(self.context, [0.0])
        self.vote_count = 0
        logger.info("Tally reset")

    def get_processing_stats(self) -> Dict:
        """Get processing statistics"""
        return {
            'total_votes_processed': self.vote_count,
            'context_initialized': self.context is not None,
            'tally_initialized': self.encrypted_tally is not None
        }
```

### server/enhanced_blockchain.py

```
# server/enhanced_blockchain.py - Enhanced blockchain with homomorphic support
import hashlib
import json
import time
import secrets
import logging
from typing import Dict, List, Any, Optional
import sys
import os

sys.path.append(os.path.join(os.path.dirname(__file__), '..', 'core'))

from blockchain_voting import VotingBlockchain, Block
from server_homomorphic import HomomorphicVoteProcessor

logger = logging.getLogger(__name__)

class EnhancedBlock(Block):
    """Enhanced block with homomorphic data"""

    def __init__(self, index: int, transactions: List[Dict], previous_hash: str, homomorp
        super().__init__(index, transactions, previous_hash)
        self.homomorphic_data = homomorphic_data or {}
        self.hash = self.calculate_enhanced_hash()

    def calculate_enhanced_hash(self) -> str:
        """Calculate hash including homomorphic data"""
        block_string = (f"{self.index}{self.timestamp}"
                        f"{json.dumps(self.transactions, sort_keys=True)}"
                        f"{self.previous_hash}{self.nonce}"
                        f"{json.dumps(self.homomorphic_data, sort_keys=True)}")
        return hashlib.sha256(block_string.encode()).hexdigest()

class EnhancedVotingBlockchain(VotingBlockchain):
    """Enhanced blockchain with homomorphic encryption support"""

    def __init__(self):
        super().__init__()
        self.homomorphic_processor = HomomorphicVoteProcessor()
        self.election_active = False
        self.chain = [self.create_enhanced_genesis_block()]
        logger.info("Enhanced blockchain initialized")

    def create_enhanced_genesis_block(self) -> EnhancedBlock:
        """Create enhanced genesis block"""
        genesis_data = {
            'homomorphic_enabled': True,
            'encryption_scheme': 'CKKS'
        }
        return EnhancedBlock(0, [], "0", genesis_data)

    def add_homomorphic_vote_transaction(self, voter_proof: Dict, encrypted_vote: Any, zk
        """Add homomorphically encrypted vote"""
        try:
            if hasattr(encrypted_vote, 'serialize'):
                vote_data = encrypted_vote.serialize()
            elif isinstance(encrypted_vote, dict):
                vote_data = encrypted_vote
            else:
                vote_data = str(encrypted_vote)

            transaction = {
                'type': 'HOMOMORPHIC_VOTE',
                'voter_id_hash': voter_proof.get('voter_hash',
                    hashlib.sha256(str(voter_proof).encode()).hexdigest()),
                'encrypted_vote': vote_data,
                'zkp_proofs': zkp_proofs,
                'timestamp': time.time(),
                'transaction_id': secrets.token_hex(16)
            }

            # Add to homomorphic tally
            success = self.homomorphic_processor.add_encrypted_vote(vote_data)
            transaction['added_to_tally'] = success

            self.pending_transactions.append(transaction)
            logger.info(f"Added homomorphic vote: {transaction['transaction_id']}")

            return transaction['transaction_id']

        except Exception as e:
            logger.error(f"Failed to add homomorphic vote: {str(e)}")
            raise

    def mine_pending_transactions(self) -> Optional[EnhancedBlock]:
        """Mine enhanced block"""
        if not self.pending_transactions:
            return None

        try:
            homomorphic_data = {
                'votes_in_tally': self.homomorphic_processor.vote_count,
                'processing_stats': self.homomorphic_processor.get_processing_stats()
            }

            block = EnhancedBlock(
                len(self.chain),
                self.pending_transactions.copy(),
                self.get_latest_block().hash,
                homomorphic_data
            )

            block.mine_block(self.difficulty)
            self.chain.append(block)
            self.pending_transactions = []

            logger.info(f"Mined enhanced block {block.index}")
            return block

        except Exception as e:
            logger.error(f"Mining failed: {str(e)}")
            return None

    def perform_homomorphic_tally(self) -> Dict:
        """Perform homomorphic tallying"""
        try:
            encrypted_results = self.homomorphic_processor.get_encrypted_results()

            tally_transaction = {
                'type': 'ENCRYPTED_TALLY',
                'encrypted_results': encrypted_results.get('encrypted_tally', ''),
                'total_votes': encrypted_results.get('total_votes', 0),
                'timestamp': time.time(),
                'requires_threshold_decryption': True,
                'transaction_id': secrets.token_hex(16)
            }

            self.pending_transactions.append(tally_transaction)
            final_block = self.mine_pending_transactions()

            return {
                'status': 'success',
                'encrypted_results': encrypted_results,
                'final_block_hash': final_block.hash if final_block else None
            }

        except Exception as e:
            logger.error(f"Homomorphic tally failed: {str(e)}")
            return {'status': 'error', 'error': str(e)}
```

## 4. Client-Side Components

### client/homomorphic_client.py

```
# client/homomorphic_client.py - Client-side homomorphic encryption
import tenseal as ts
import numpy as np
import logging
import hashlib
from typing import Dict, Any, List

logger = logging.getLogger(__name__)

class HomomorphicEncryption:
    """Client-side homomorphic encryption for private vote tallying"""

    def __init__(self, poly_modulus_degree: int = 8192):
        self.context = None
        self.public_context = None
        self.poly_modulus_degree = poly_modulus_degree
        self.setup_context()

    def setup_context(self):
        """Setup CKKS context for homomorphic encryption"""
        try:
            self.context = ts.context(
                ts.SCHEME_TYPE.CKKS,
                poly_modulus_degree=self.poly_modulus_degree,
                coeff_mod_bit_sizes=[60, 40, 40, 60]
            )

            self.context.generate_galois_keys()
            self.context.global_scale = 2**40
            self.public_context = self.context.serialize(save_secret_key=False)

            logger.info("Homomorphic encryption context initialized")

        except Exception as e:
            logger.error(f"Context setup failed: {str(e)}")
            raise

    def encrypt_vote(self, vote: int) -> Dict[str, Any]:
        """Encrypt single vote"""
        try:
            if vote not in [0, 1]:
                raise ValueError("Vote must be 0 or 1")

            vote_vector = [float(vote)]
            encrypted_vote = ts.ckks_vector(self.context, vote_vector)

            return {
                'serialized': encrypted_vote.serialize(),
                'vote_hash': hashlib.sha256(str(vote).encode()).hexdigest()
            }

        except Exception as e:
            logger.error(f"Vote encryption failed: {str(e)}")
            raise

    def encrypt_batch_votes(self, votes: List[int]) -> Dict[str, Any]:
        """Encrypt multiple votes"""
        try:
            for vote in votes:
                if vote not in [0, 1]:
                    raise ValueError(f"All votes must be 0 or 1, got {vote}")

            vote_floats = [float(v) for v in votes]
            encrypted_batch = ts.ckks_vector(self.context, vote_floats)

            return {
                'serialized': encrypted_batch.serialize(),
                'batch_size': len(votes),
                'batch_hash': hashlib.sha256(str(votes).encode()).hexdigest()
            }

        except Exception as e:
            logger.error(f"Batch encryption failed: {str(e)}")
            raise

    def get_public_context(self) -> bytes:
        """Get public context for server"""
        if self.public_context is None:
            raise ValueError("Public context not initialized")
        return self.public_context

    def verify_encryption(self, encrypted_data: Dict[str, Any], original_vote: int = None
        """Verify encryption integrity"""
        try:
            encrypted_vote = ts.ckks_vector_from(self.context, encrypted_data['serialized

            if encrypted_vote.size() == 0:
                return False

            if original_vote is not None:
                decrypted = encrypted_vote.decrypt()
                if len(decrypted) > 0 and abs(decrypted[0] - float(original_vote)) < 0.1:
                    return True
                return False

            return True

        except Exception as e:
            logger.warning(f"Verification failed: {str(e)}")
            return False

    def get_context_info(self) -> Dict[str, Any]:
        """Get context information"""
        return {
            'scheme': 'CKKS',
            'poly_modulus_degree': self.poly_modulus_degree,
            'global_scale': int(self.context.global_scale) if self.context else None,
            'context_initialized': self.context is not None
        }
```

### client/client_zkp.py

```
# client/client_zkp.py - Client-side ZK proof generation
import hashlib
import random
from
```