```
import random
import hashlib

# Set seed for reproducibility
random.seed(42)

# ========================
# Part 1: MPC using Additive Secret Sharing
# ========================
EMPLOYEES = ['Ahmed', 'Ali', 'Rashid']
SALARIES = {'Ahmed': 80, 'Ali': 70, 'Rashid': 70}  # تم التصحيح هنا
TOTAL_SALARY = sum(SALARIES.values())  # 220 سيكون
PRIME = 999983

def create_secret_shares(secret_val, num_people, modulus):
    """Split secret into random additive shares"""
    shares_list = [random.randint(1, modulus - 1) for _ in range(num_people - 1)]
    final_share = (secret_val - sum(shares_list)) % modulus
    shares_list.append(final_share)
    return shares_list

def compute_mpc_total(salary_data, participants, modulus):
    """Compute total using MPC without revealing individual values"""
    shares_distributed = {person: [] for person in participants}

    # Each person splits their salary into shares
    for person, salary in salary_data.items():
        shares = create_secret_shares(salary, len(participants), modulus)
        for idx, receiver in enumerate(participants):
            shares_distributed[receiver].append(shares[idx])

    # Each person computes their partial sum
    partial_results = {}
    for person in participants:
        partial_results[person] = sum(shares_distributed[person]) % modulus

    # Compute global total
    global_total = sum(partial_results.values()) % modulus
    return global_total, shares_distributed, partial_results

# ========================
# Part 2: Shamir's Secret Sharing
# ========================
def extended_gcd(a_val, b_val):
    """Extended Euclidean Algorithm"""
    if a_val == 0:
        return b_val, 0, 1
    gcd_val, x1, y1 = extended_gcd(b_val % a_val, a_val)
    x = y1 - (b_val // a_val) * x1
    y = x1
    return gcd_val, x, y

def calculate_mod_inverse(number, mod):
    """Calculate modular inverse"""
    gcd_val, x_val, _ = extended_gcd(number % mod, mod)
    if gcd_val != 1:
        raise ValueError('Inverse does not exist')
    return x_val % mod

def generate_shamir_shares(secret_data, min_shares, total_shares, prime_num):
    """Generate shares using Shamir's Secret Sharing"""
    coefficients = [random.randint(1, prime_num - 1) for _ in range(min_shares - 1)]

    def polynomial_func(x_point):
        """Evaluate polynomial at given point"""
        result = secret_data
        for exp, coef in enumerate(coefficients, start=1):
            result = (result + coef * pow(x_point, exp, prime_num)) % prime_num
        return result

    return [(i, polynomial_func(i)) for i in range(1, total_shares + 1)]

def recover_secret(share_list, prime_num):
    """Recover secret using Lagrange interpolation"""
    x_points = [s[0] for s in share_list]
    y_points = [s[1] for s in share_list]

    recovered_val = 0
    for j in range(len(share_list)):
        numerator = 1
        denominator = 1
```

```
            denominator = 1
            x_j = x_points[j]

            for m in range(len(share_list)):
                if m == j:
                    continue
                x_m = x_points[m]
                numerator = (numerator * (-x_m)) % prime_num
                denominator = (denominator * (x_j - x_m)) % prime_num

            lagrange_coef = (numerator * calculate_mod_inverse(denominator, prime_num)) % prime_num
            recovered_val = (recovered_val + y_points[j] * lagrange_coef) % prime_num

        return recovered_val

    # ========================
    # Part 3: Garbled AND Gate
    # ========================
    def create_hash_encryption(key_data, message_text):
        """Simulate encryption using hash function"""
        return hashlib.sha256(key_data + message_text.encode()).hexdigest()[:16]

    def generate_random_key():
        """Generate random key bytes"""
        return bytes([random.randint(0, 255) for _ in range(8)])

    def execute_garbled_and(input_a, input_b):
        """Execute garbled AND gate protocol"""
        # Generate keys for inputs and outputs
        key_a0, key_a1 = generate_random_key(), generate_random_key()
        key_b0, key_b1 = generate_random_key(), generate_random_key()
        key_out0, key_out1 = generate_random_key(), generate_random_key()

        # Create garbled table
        garbled_table = {
            create_hash_encryption(key_a0 + key_b0, "00"): key_out0,
            create_hash_encryption(key_a0 + key_b1, "01"): key_out0,
            create_hash_encryption(key_a1 + key_b0, "10"): key_out0,
            create_hash_encryption(key_a1 + key_b1, "11"): key_out1,
        }

        # Select keys based on inputs
        selected_key_a = key_a1 if input_a else key_a0
        selected_key_b = key_b1 if input_b else key_b0

        # Evaluate the circuit
        encrypted_value = create_hash_encryption(selected_key_a + selected_key_b, f"{input_a}{input_b}")
        output_key = garbled_table.get(encrypted_value)

        if output_key is None:
            return None
        return 1 if output_key == key_out1 else 0

    # ========================
    # Main Execution
    # ========================
    def main():
        print("SECURE MULTI-PARTY COMPUTATION - LAB 04")
        print("=" * 50)

        # ---------- Part 1: MPC Sum ----------
        final_total, distribution, partial_sums = compute_mpc_total(SALARIES, EMPLOYEES, PRIME)

        print("\nPART 1: MPC WITH ADDITIVE SECRET SHARING")
        print("-" * 40)
        print("Employee Salaries:", SALARIES)
        print("\nShares Distributed:")
        for emp in EMPLOYEES:
            print(f"  {emp}: {distribution[emp]}")
        print("\nPartial Sums Calculated by Each Employee:")
        for emp in EMPLOYEES:
            print(f"  {emp}: {partial_sums[emp]}")
        print(f"\nFinal Computed Total: {final_total}")
        print(f"Actual Total: {TOTAL_SALARY}")
        print(f"Verification: {'SUCCESS' if final_total == TOTAL_SALARY else 'FAILED'}")

        # ---------- Part 2: Shamir Secret Sharing ----------
        MIN_SHARES_NEEDED = 3
        TOTAL_SHARES_GEN = 5
        ORIGINAL_SECRET = 123
        PRIME_USED = 1013

        generated_shares = generate_shamir_shares(ORIGINAL_SECRET, MIN_SHARES_NEEDED, TOTAL_SHARES_GEN, PRIME_USED)
```

```
    secret_recovered = recover_secret(generated_shares[:MIN_SHARES_NEEDED], PRIME_USED)
    failed_recovery = recover_secret(generated_shares[:MIN_SHARES_NEEDED-1], PRIME_USED)

    print("\nPART 2: SHAMIR'S SECRET SHARING")
    print("-" * 40)
    print(f"Original Secret: {ORIGINAL_SECRET}")
    print(f"Prime Used: {PRIME_USED}")
    print(f"Minimum Shares Required (k): {MIN_SHARES_NEEDED}")
    print(f"\nGenerated Shares (n={TOTAL_SHARES_GEN}):")
    for idx, (x, y) in enumerate(generated_shares):
        print(f"  Share {idx+1}: ({x}, {y})")

    print(f"\nUsing {MIN_SHARES_NEEDED} shares for recovery:")
    print(f"Recovered Secret: {secret_recovered}")
    print(f"Using {MIN_SHARES_NEEDED-1} shares (should fail):")
    print(f"Recovered Value: {failed_recovery}")
    print(f"Verification: {'SUCCESS' if secret_recovered == ORIGINAL_SECRET and failed_recovery != ORIGINAL_SECRET else 'FAI

    # ---------- Part 3: Garbled Circuit ----------
    INPUT_X = 1
    INPUT_Y = 0
    and_result = execute_garbled_and(INPUT_X, INPUT_Y)

    print("\nPART 3: GARBLED AND GATE")
    print("-" * 40)
    print(f"First Input (Omar): {INPUT_X}")
    print(f"Second Input (Nancy): {INPUT_Y}")
    print(f"Garbled Circuit Output: {and_result}")
    print(f"Expected AND Result: {INPUT_X & INPUT_Y}")
    print(f"Verification: {'SUCCESS' if and_result == (INPUT_X & INPUT_Y) else 'FAILED'}")

if __name__ == "__main__":
    main()
```

```
SECURE MULTI-PARTY COMPUTATION - LAB 04
================================================

PART 1: MPC WITH ADDITIVE SECRET SHARING
----------------------------------------
Employee Salaries: {'Ahmed': 80, 'Ali': 70, 'Rashid': 70}

Shares Distributed:
  Ahmed: [670488, 26226, 288390]
  Ali: [116740, 777573, 256788]
  Rashid: [212835, 196254, 454875]

Partial Sums Calculated by Each Employee:
  Ahmed: 985104
  Ali: 151118
  Rashid: 863964

Final Computed Total: 220
Actual Total: 220
Verification: SUCCESS

PART 2: SHAMIR'S SECRET SHARING
----------------------------------------
Original Secret: 123
Prime Used: 1013
Minimum Shares Required (k): 3

Generated Shares (n=5):
  Share 1: (1, 495)
  Share 2: (2, 140)
  Share 3: (3, 71)
  Share 4: (4, 288)
  Share 5: (5, 791)

Using 3 shares for recovery:
Recovered Secret: 123
Using 2 shares (should fail):
Recovered Value: 850
Verification: SUCCESS

PART 3: GARBLED AND GATE
----------------------------------------
First Input (Omar): 1
Second Input (Nancy): 0
Garbled Circuit Output: 0
Expected AND Result: 0
Verification: SUCCESS
```