```python
import random
from sympy import mod_inverse

# ElGamal key generation
def generate_keys(p):
    """Generate ElGamal public and private keys."""
    g = random.randint(2, p - 2)  # Generator
    x = random.randint(1, p - 2)  # Private key
    y = pow(g, x, p)  # Public key
    return (p, g, y), x  # Public key: (p, g, y), Private key: x

# ElGamal encryption
def encrypt(public_key, message):
    p, g, y = public_key
    k = random.randint(1, p - 2)  # Random integer k
    c1 = pow(g, k, p)
    c2 = (message * pow(y, k, p)) % p
    return c1, c2

# ElGamal decryption (partial and final)
def decrypt_partial(private_key, public_key, ciphertext):
    p, g, y = public_key
    c1, c2 = ciphertext
    s = pow(c1, private_key, p)  # Shared secret
    return c2, s  # Partially decrypted message with the shared secret

def decrypt_final(partially_decrypted, shared_secret, p):
    c2, s = partially_decrypted
    s_inv = mod_inverse(s, p)  # Modular inverse of shared secret
    message = (c2 * s_inv) % p
    return message

# Secure addition based on the protocol
def secure_addition(p, a, b, public_key, private_key_a, private_key_b):
    """Perform secure addition using the protocol described."""
    # Step 1: Party A encrypts all possible sums (a + b) for b in [0, 9]
    random_r = random.randint(1, p - 1)  # Random number R
    encrypted_messages = []
    for b_guess in range(10):
        m_a = a + b_guess + random_r  # Add random R to each sum
        encrypted_messages.append(encrypt(public_key, m_a))

    # Party A sends the encrypted messages (in order) to Party B
    # Step 2: Party B selects the correct encrypted message based on its value of b
    selected_ciphertext = encrypted_messages[b]

    # Party B re-randomizes the selected ciphertext
    c1, c2 = selected_ciphertext
    random_s = random.randint(1, p - 1)  # Random value S
    randomized_c1 = (c1 * pow(public_key[1], random_s, p)) % p
    randomized_c2 = (c2 * pow(public_key[2], random_s, p)) % p
    randomized_encrypted = (randomized_c1, randomized_c2)

    # Step 3: Party B sends the re-randomized ciphertext back to Party A
    partially_decrypted_c2, shared_secret_a = decrypt_partial(private_key_a, public_key, randomized_encrypted)

    # Step 4: Party A partially decrypts and sends the result back to Party B
    final_message_b = decrypt_final((partially_decrypted_c2, shared_secret_a), random_s, p)

    # Party B subtracts random R to reveal the sum
    final_sum = final_message_b - random_r
    return final_sum

# Example usage
p = 467  # A prime number
a = 3  # Sensitive number of Party A
b = 6  # Sensitive number of Party B

# Generate keys for both parties
public_key, private_key_a = generate_keys(p)
_, private_key_b = generate_keys(p)

# Perform secure addition
result = secure_addition(p, a, b, public_key, private_key_a, private_key_b)
print("The securely added result is:", result)
```