Project Execution and Analysis Report

Course: CS 6530 - Applied Cryptography **Project:** Phase 2 - BFV and CKKS Homomorphic Encryption

1. System Requirements and Installation

To successfully compile and run this project, your system must have the following components installed:

- **A C++17 Compiler:** The code uses C++17 features (`std::optional`). The standard `g++` compiler is recommended.
- **make:** The `make` utility is used to automate the build process using the provided `Makefile`.
- **Microsoft SEAL Library (libseal-dev):** This is the core homomorphic encryption library.
- **(Optional) pkg-config:** Used by the `Makefile` to automatically find the SEAL library paths.

Installation Steps (Ubuntu/Debian-based)

Install Build Tools and SEAL: Open a terminal and run the following command to install `g++`, `make`, and the Microsoft SEAL library:
sudo apt update
sudo apt install build-essential libseal-dev

1. *Note: If you are using the specific `Makefile` version from our discussion that hardcodes the paths (e.g., `/usr/local/include/SEAL-4.1`), you must ensure your SEAL installation is located at that exact path.*

Compile the Project: Navigate to the directory containing all the project files (`main_working.cpp`, `SEAL_Working.h`, `Makefile`). Run the `make` command:
make

2. This will compile the code and create an executable file named `phase2_homomorphic_working`.
   (You can paste a screenshot of the successful compilation output here.)

2. Running the Program and Providing Input

The program is interactive. It will prompt you for inputs on the terminal, but it will write all results, intermediate steps, and logs to a file named `output_log.txt`.

How to Run:

Execute the compiled program from your terminal

1. ./phase2_homomorphic_working
2. The program will immediately create `output_log.txt` and start writing to it.
3. The terminal will wait for your input.

Input Process:

The program will run two demonstrations back-to-back: first for BFV (integers) and then for CKKS (decimals).

For the BFV Demonstration:

1. **Enter Vector Size:** The first prompt is: `Enter the number of elements for the vectors (e.g., 4):` Type a number (e.g., `3`) and press Enter.
2. **Enter Vector 1:** The next prompt is: `Enter 3 space-separated integers for plaintext1:` Type your numbers (e.g., `1 2 3`) and press Enter.
3. **Enter Vector 2:** The final BFV prompt is: `Enter 3 space-separated integers for plaintext2:` Type your numbers (e.g., `4 5 6`) and press Enter.

The program will then immediately run the CKKS demonstration.

For the CKKS Demonstration:

1. **Enter Vector Size:** `Enter the number of elements for the vectors (e.g., 4):` Type a number (e.g., `3`) and press Enter.
2. **Enter Vector 1:** `Enter 3 space-separated doubles for plaintext1:` Type your decimal numbers (e.g., `1.0 2.0 3.0`) and press Enter.
3. **Enter Vector 2:** `Enter 3 space-separated doubles for plaintext2:` Type your numbers (e.g., `4.0 5.0 6.0`) and press Enter.

After the final input, the program will finish all computations and write to the log file. The terminal will display: `Program finished. Output written to output_log.txt`

3. Code Execution Trace and Key Functions

The program's logic is contained in `SEAL_Working.h`. The `main` function simply creates an instance of the `SEAL_Working` class and calls its two main demonstration functions: `demonstrateBFV()` and `demonstrateCKKS()`.

Here is a trace of the execution, matching the output log provided.

BFV Trace (Example: `[1 2 3]` and `[4 5 6]`)

1. **Encoding & Encryption (`demonstrateBFV`):** The input vectors are encoded into polynomials (`ptxt1`) using `bfv_encoder.encode()` and then encrypted using `bfv_encryptor->encrypt()`.

Log Output:

- [INFO] Plaintext 1 encoded to polynomial: 3C6DDx^8191 ...
- [INFO] ctxt1 (after encryption): size = 2, noise budget = 146 bits
- **Explanation:** The log shows the start of the massive, unreadable polynomial (`3C6DDx^...`). The ciphertext `ctxt1` has a `size` of 2 (meaning 2 polynomials) and a full "health bar" or `noise budget` of 146 bits.

2. **Homomorphic Addition (`demonstrateBFV`):** The two ciphertexts (`ctxt1`, `ctxt2`) are added using `bfv_evaluator.add()`.

Log Output:

- Addition (1+4, 2+5, 3+6): 5 7 9
- [INFO] ctxt_sum (after addition): size = 2, noise budget = 146 bits
- **Explanation:** The decrypted result is `5 7 9`, which is correct. Notice the noise budget remains high (146 bits) and the size is still 2. Homomorphic addition is "cheap."

3. **Homomorphic Multiplication (The "Middle Change"):** This is the most important part. `bfv_evaluator.multiply()` is called.

Log Output:

- [INFO] ctxt_mult (after multiply): size = 3, noise budget = 114 bits
- **Explanation:** This is the *intermediate ciphertext*. As predicted by FHE theory, multiplying two size-2 ciphertexts results in a size-3 ciphertext. This operation is "expensive," which is proven by the noise budget dropping from 146 to 114 bits.

4. **Relinearization (`demonstrateBFV`):** The size-3 ciphertext is unusable for further multiplications, so `bfv_evaluator.relinearize_inplace()` is called to shrink it.

Log Output:

- [INFO] ctxt_mult (after relinearize):size = 2, noise budget = 114 bits
- **Explanation:** The ciphertext size is now back to 2, making it standard again. The noise budget remains 114 (relinearization adds its own small noise, but the main cost was the multiplication itself). The final decrypted result is `4 10 18`, which is correct.

CKKS Trace (Example: `[1.0 2.0 3.0]` and `[4.0 5.0 6.0]`)

1. **Encoding & Encryption (`demonstrateCKKS`):** The `double` vectors are encoded using `ckks_encoder.encode()` with a large scale factor (e.g., 2^{40}).

Log Output:

- [INFO] ctxt1 (after encryption): size = 2, level = 2, scale = 40.0 bits
- **Explanation:** For CKKS, we track `level` (which modulus we are on) and `scale` (the precision). We start at the highest level (level 2 in this log) with a scale of 40 bits.

2. **Homomorphic Multiplication (`demonstrateCKKS`):** `ckks_evaluator.multiply()` is called.

Log Output:

- [INFO] ctxt_mult (after multiply): size = 3, level = 2, scale = 80.0 bits
- **Explanation:** Just like BFV, the size grows to 3. Critically, the scales are multiplied ($2^{40} * 2^{40} = 2^{80}$), so the new scale is `80.0 bits`. This high scale/noise must be fixed.

3. **Relinearization (`demonstrateCKKS`):** `ckks_evaluator.relinearize_inplace()` fixes the size.

Log Output:

- [INFO] ctxt_mult (after relinearize):size = 2, level = 2, scale = 80.0 bits
- **Explanation:** Size is back to 2. The scale and level are untouched.

4. **Rescaling (The Key CKKS Step):** `ckks_evaluator.rescale_to_next_inplace()` is called. This is the "magic" of CKKS.

Log Output:

- [INFO] ctxt_mult (after rescale): size = 2, level = 1, scale = 40.0 bits
- **Explanation:** This single operation "consumes" a level (dropping from 2 to 1) to "burn off" the extra scale, bringing it back down from 80 to 40 bits. This manages the noise and precision, allowing for another multiplication at this new level. The final decrypted result is `4.000 10.000 18.000`, which is correct.

4. Verification Process

The program proves its own correctness by running two computations in parallel:

1. **Plaintext Computation:** Inside the "Verification" section of each function, the code calculates the expected answer using the original, unencrypted input vectors.
   - **BFV:** `expected_sum.push_back(plaintext1[i] + plaintext2[i]);`
   - **CKKS:** `expected_mult.push_back(plaintext1[i] * plaintext2[i]);`
2. **Homomorphic Computation:** This is the `sum_result` or `mult_result` vector that is obtained by decrypting the final ciphertext.

The verification check then compares these two results:

For BFV (Exact):

- A simple `==` is used. The decrypted result must be *identical* to the plaintext calculation.
  bool sum_correct = (sum_result == expected_sum);
- // Log output: Addition verification: PASS

For CKKS (Approximate):

- We cannot use `==` because CKKS is approximate. A special helper function, `verifyVectors()`, is used.
  bool mult_correct = verifyVectors(expected_mult, mult_result, 0.01);
- // Log output: Multiplication verification: PASS
- This function (in `SEAL_Working.h`) checks if the absolute difference between the expected and actual values is less than a small tolerance (e.g., `0.01`). This is the standard method for verifying approximate homomorphic schemes.

The "PASS" messages in the log are the final proof that the homomorphic operations were successful.

5. Full Execution Log Example :

```
tyche@t-HP-Pavilion-Laptop-14-dv2xxx:~/Documents/Applied Cryptography/Project$ make
g++ -std=c++17 -Wall -Wextra -O2 -g -I/usr/local/include/SEAL-4.1 -o phase2_homomorphic_working main_working.cpp -L/usr/local/lib -lseal-4.1
Build completed successfully!
tyche@t-HP-Pavilion-Laptop-14-dv2xxx:~/Documents/Applied Cryptography/Project$ ./phase2_homomorphic_working
   Enter the number of elements for the vectors (e.g., 4): 3
   Enter 3 space-separated integers for plaintext1: 1 2 3
   Enter 3 space-separated integers for plaintext2: 4 5 6
   Enter the number of elements for the vectors (e.g., 4): 3
   Enter 3 space-separated doubles for plaintext1: 1.0 2.0 3.0
   Enter 3 space-separated doubles for plaintext2: 4.0 5.0 6.0
Program finished. Output written to output_log.txt
tyche@t-HP-Pavilion-Laptop-14-dv2xxx:~/Documents/Applied Cryptography/Project$
```

```
BFV (Brakerski-Fan-Vercauteren) with Microsoft SEAL
============================================================

1. Key Generation:
   Key generation completed in constructor (0 microseconds)

2. Encryption:
   Vector size chosen: 3
   User input for plaintext1: 1 2 3
   User input for plaintext2: 4 5 6
   Plaintext 1: 1 2 3
   Plaintext 2: 4 5 6
      [INFO] Plaintext 1 encoded to polynomial: 3C6DDx^8191 + C0707x^8190 + 6BF9Ex^8189 ...
   Encryption completed in 12440 microseconds
      [INFO] ctxt1 (after encryption):     size = 2, noise budget = 146 bits
      [INFO] ctxt2 (after encryption):     size = 2, noise budget = 146 bits

3. Homomorphic Operations:
   Addition (1+4, 2+5, 3+6): 5 7 9
   Addition operation took 697 microseconds
      [INFO] ctxt_sum (after addition):    size = 2, noise budget = 146 bits
   Multiplication operation took 19178 microseconds
      [INFO] ctxt_mult (after multiply):   size = 3, noise budget = 114 bits
   Relinearization operation took 3283 microseconds
      [INFO] ctxt_mult (after relinearize):size = 2, noise budget = 114 bits
   Multiplication (1*4, 2*5, 3*6): 4 10 18
   Scalar multiplication (1*2, 2*2, 3*2): 2 4 6
   Scalar multiplication took 3856 microseconds
      [INFO] ctxt_scalar (after plain_mult):size = 2, noise budget = 121 bits

4. Verification:
   Addition verification: PASS
   Multiplication verification: PASS
   Scalar multiplication verification: PASS

✅ BFV with Microsoft SEAL: ALL TESTS PASSING!
   - Proper noise management
   - Relinearization after multiplication
   - Modulus switching for noise reduction
```

```
CKKS (Cheon-Kim-Kim-Song) with Microsoft SEAL
============================================================

1. Key Generation:
   Key generation completed in constructor (0 microseconds)

2. Encryption:
   Vector size chosen: 3
   User input for plaintext1: 1.0 2.0 3.0
   User input for plaintext2: 4.0 5.0 6.0
   Plaintext 1: 1 2 3
   Plaintext 2: 4 5 6
   Encryption completed in 10290 microseconds
      [INFO] ctxt1 (after encryption):     size = 2, level = 2, scale = 40.0 bits
      [INFO] ctxt2 (after encryption):     size = 2, level = 2, scale = 40.0 bits

3. Homomorphic Operations:
   Addition result: 5.000 7.000 9.000
   Addition operation took 485 microseconds
      [INFO] ctxt_sum (after addition):    size = 2, level = 2, scale = 40.0 bits
   Multiplication operation took 1250 microseconds
      [INFO] ctxt_mult (after multiply):   size = 3, level = 2, scale = 80.0 bits
   Relinearization operation took 4870 microseconds
      [INFO] ctxt_mult (after relinearize):size = 2, level = 2, scale = 80.0 bits
   Rescaling operation took 991 microseconds
      [INFO] ctxt_mult (after rescale):    size = 2, level = 1, scale = 40.0 bits
   Multiplication result: 4.000 10.000 18.000
   Scalar multiplication (2.0x) result: 2.000 4.000 6.000
   Scalar multiplication took 1631 microseconds
      [INFO] ctxt_scalar (after plain_mult):size = 2, level = 1, scale = 40.0 bits
   Plaintext addition result: 5.000 7.000 9.000
   Plaintext addition took 315 microseconds
      [INFO] ctxt_add_plain (after add_plain):size = 2, level = 2, scale = 40.0 bits

4. Verification:
   Addition verification: PASS
   Multiplication verification: PASS
   Scalar multiplication verification: PASS
   Plaintext addition verification: PASS
```