



南開大學
Nankai University

南开大学

计算机学院和密码与网络空间安全学院

《数据安全》课程作业

SEAL 应用实践

姓名：陆皓喆

学号：2211044

专业：信息安全

指导教师：刘哲理

2025 年 3 月 12 日

目录

| | |
|--|-----------|
| 1 实验要求 | 2 |
| 2 github 仓库 | 2 |
| 3 实验原理 | 2 |
| 3.1 开发框架 SEAL(Simple Encrypted Arithmetic Library) | 2 |
| 3.2 CKKS 算法 (Cheon-Kim-Kim-Song) | 2 |
| 3.3 标准化构建流程 | 4 |
| 4 实验过程 | 4 |
| 4.1 SEAL 库安装 | 4 |
| 4.1.1 git clone 加密库资源 | 4 |
| 4.1.2 编译和安装 | 5 |
| 4.2 简单测试程序 | 7 |
| 4.3 复现 $x * y * z$ 的运算 | 8 |
| 4.4 完成 $x^3 + y * z$ 的运算 | 12 |
| 5 实验心得与体会 | 19 |

1 实验要求

参考教材实验 2.3，实现将三个数的密文发送到服务器完成 $x^3 + y \times z$ 的运算。

2 github 仓库

本次实验的有关代码和文件，都已经上传至我的个人 github 中。

您可以通过访问[此链接](#)来查阅我的代码文件。

3 实验原理

3.1 开发框架 SEAL(Simple Encrypted Arithmetic Library)

SEAL 是一个开放源代码的、专门用于同态加密的库，它由微软研究院开发。同态加密是一种加密形式，允许用户在加密的数据上直接进行计算，而无需先对数据解密。这种技术对于保护数据隐私而进行的安全计算尤为重要，尤其是在云计算和外包计算场景中。

主要特点：

- **易用性:** SEAL 设计时特别考虑了易用性，尽管同态加密本身是一个复杂的领域，SEAL 提供了一个相对简单的 API，使得非专家也能较容易地实现加密计算。
- **性能:** 通过优化算法和利用现代硬件特性（如多核处理器和 SIMD 指令集），SEAL 能够提供高效的同态加密操作。
- **通用性:** SEAL 支持多种同态加密方案，包括完全同态加密（FHE）和部分同态加密（PHE）方案，使其可以适用于多种不同的应用场景。
- **安全性:** SEAL 在设计和实现时充分考虑了安全性，旨在抵抗包括量子计算机在内的未来潜在威胁。

应用场景：

- **数据隐私保护:** 在云计算环境中安全地处理敏感数据，例如，医疗记录分析、金融数据处理等。
- **安全多方计算:** 多个参与方可以在保持各自数据隐私的同时共同进行计算。
- **加密搜索:** 在加密的数据库上进行搜索，而不暴露搜索内容。

3.2 CKKS 算法 (Cheon-Kim-Kim-Song)

CKKS (Cheon - Kim - Kim - Song) 算法是一种基于格的全同态加密 (Fully Homomorphic Encryption, FHE) 方案，主要用于处理近似算术运算，特别适合于处理浮点数和实数。

该算法的主要流程如下所示：

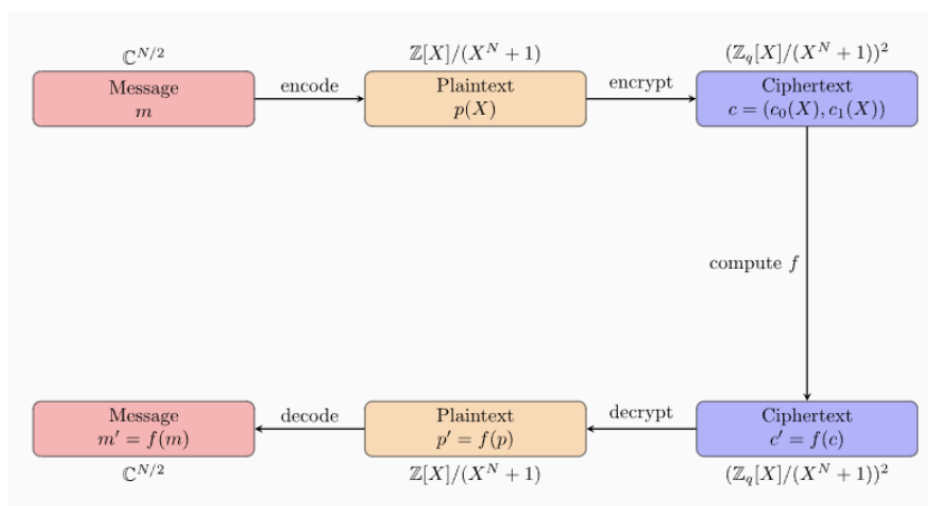


图 3.1: CKKS 算法原理

该算法基于基于 LWE 容错学习中的 RLWE，至少和格中的难题一样困难，从而能够抵抗量子计算机的攻击。

CKKS 作为全同态加密方案，支持密文近似算术运算，适用于浮点数和实数，在多领域有广泛应用：

1. 医疗保健

- **数据研究**：加密患者医疗数据，如健康记录、基因数据，在密文状态下进行联合研究、疾病预测、药物疗效评估，保护隐私。
- **远程医疗**：加密患者生命体征数据，医疗机构在密文状态分析，及时发现健康问题。

2. 金融服务

- **隐私计算**：在加密客户财务、交易等敏感信息上进行风险评估、信用评分等计算。
- **多方计算**：用于金融交易多方联合决策、清算结算，各方加密数据计算，不泄露原始信息。

3. 机器学习与 AI

- **模型训练**：在联邦学习中，各方对加密数据本地训练，聚合加密参数，保护数据隐私。
- **模型推理**：加密用户敏感数据输入模型，密文推理后返回结果再解密。

4. 云计算

- **外包计算**：企业加密数据上传云端，云服务提供商密文计算后返回结果，保障数据安全。
- **多租户安全**：确保多租户环境下不同用户加密数据的隔离和计算安全。

5. 物联网 (IoT)

- **数据保护**：加密物联网设备产生的敏感数据，如生活习惯、工业生产数据，传输处理全程保密。
- **边缘计算**：用于边缘设备对加密数据的本地计算，保证隐私安全。

3.3 标准化构建流程

CKKS 算法由五个模块组成：密钥生成器 **keygenerator**、加密模块 **encryptor**、解密模块 **decryptor**、密文计算模块 **evaluator** 和编码器 **encoder**，其中编码器实现数据和环上元素的相互转换。

依据这五个模块，构建同态加密应用的过程为：

1. 选择 CKKS 参数 **parms**
2. 生成 CKKS 框架 **context**
3. 构建 CKKS 模块 **keygenerator**、**encoder**、**encryptor**、**evaluator** 和 **decryptor**
4. 使用 **encoder** 将数据 **n** 编码为明文 **m**
5. 使用 **encryptor** 将明文 **m** 加密为密文 **c**
6. 使用 **evaluator** 对密文 **c** 运算为密文 **c'**
7. 使用 **decryptor** 将密文 **c'** 解密为明文 **m'**
8. 使用 **encoder** 将明文 **m'** 解码为数据 **n**

4 实验过程

4.1 SEAL 库安装

4.1.1 git clone 加密库资源

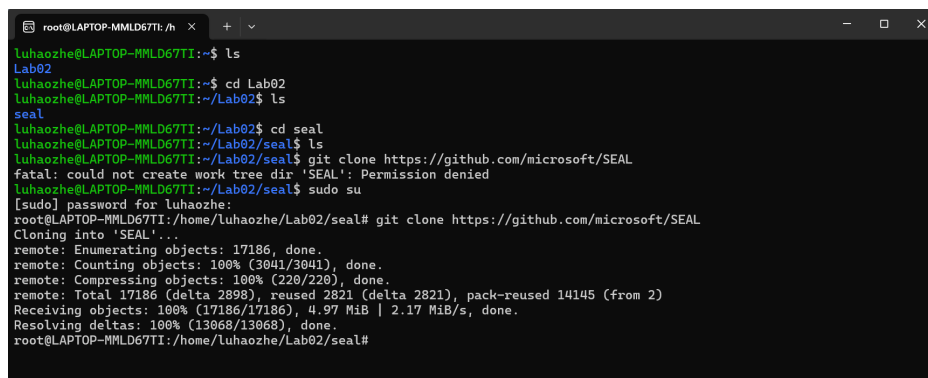
首先，我们在 ubuntu 环境中的 /Desktop/Data_Security 中新建一个文件夹 Lab2，作为本次实验的基础路径。

然后，我们在 Lab2 文件夹中新建一个文件夹 seal，进入该文件夹，打开终端，输入命令：

```
git clone https://github.com/microsoft/SEAL
```

由于不知道为什么，我的 ubuntu 虚拟机死活连接不上 github，一直报错，试了好久也不太行，所以最后我选择重新安装了 WSL 进行实验。

首先，我们输入上面的命令行 `git clone https://github.com/microsoft/SEAL`，安装 SEAL。结果如下所示：



```
root@LAPTOP-MMLD67TI: /h x + v
luhaozhe@LAPTOP-MMLD67TI:~$ ls
Lab02
luhaozhe@LAPTOP-MMLD67TI:~$ cd Lab02
luhaozhe@LAPTOP-MMLD67TI:~/Lab02$ ls
seal
luhaozhe@LAPTOP-MMLD67TI:~/Lab02$ cd seal
luhaozhe@LAPTOP-MMLD67TI:~/Lab02/seal$ ls
luhaozhe@LAPTOP-MMLD67TI:~/Lab02/seal$ git clone https://github.com/microsoft/SEAL
fatal: could not create work tree dir 'SEAL': Permission denied
luhaozhe@LAPTOP-MMLD67TI:~/Lab02/seal$ sudo su
[sudo] password for luhaozhe:
root@LAPTOP-MMLD67TI:/home/luhaozhe/Lab02/seal# git clone https://github.com/microsoft/SEAL
Cloning into 'SEAL'...
remote: Enumerating objects: 17186, done.
remote: Counting objects: 100% (3041/3041), done.
remote: Compressing objects: 100% (220/220), done.
remote: Total 17186 (delta 2898), reused 2821 (delta 2821), pack-reused 14145 (from 2)
Receiving objects: 100% (17186/17186), 4.97 MiB | 2.17 MiB/s, done.
Resolving deltas: 100% (13068/13068), done.
root@LAPTOP-MMLD67TI:/home/luhaozhe/Lab02/seal#
```

图 4.2: git 安装 SEAL

4.1.2 编译和安装

安装完之后，首先我们输入 `cd SEAL`，进入到我们的文件夹中。

然后，我们输入以下命令行：

```
1 cmake .
```

对整个项目进行编译，得到以下的结果：

```
-- x86intrin.h - found
-- SEAL_USE_INTRIN: ON
-- Performing Test SEAL_MEMSET_S_FOUND
-- Performing Test SEAL_MEMSET_S_FOUND - Failed
-- Looking for explicit_bzero
-- Looking for explicit_bzero - found
-- Looking for explicit_memset
-- Looking for explicit_memset - not found
-- SEAL_USE_MEMSET_S: OFF
-- SEAL_USE_EXPLICIT_BZERO: ON
-- SEAL_USE_EXPLICIT_MEMSET: OFF
-- Looking for pthread.h
-- Looking for pthread.h - found
-- Performing Test CMAKE_HAVE_LIBC_PTHREAD
-- Performing Test CMAKE_HAVE_LIBC_PTHREAD - Failed
-- Check if compiler accepts -pthread
-- Check if compiler accepts -pthread - yes
-- Found Threads: TRUE
-- SEAL_BUILD_SEAL_C: OFF
-- SEAL_BUILD_EXAMPLES: OFF
-- SEAL_BUILD_TESTS: OFF
-- SEAL_BUILD_BENCH: OFF
-- Configuring done
-- Generating done
-- Build files have been written to: /home/luhaozhe/Lab02/seal/SEAL
root@LAPTOP-MMLD67TI:/home/luhaozhe/Lab02/seal/SEAL#
```

图 4.3: cmake 得到的结果

说明我们成功完成了 cmake 的编译。接下来，我们输入 `make`，得到以下结果：

```

[ 53%] Building C object thirdparty/zlib-build/CMakeFiles/zlibstatic.dir/zutil.o
[ 54%] Linking C static library ../lib/libz.a
[ 54%] Built target zlibstatic
Scanning dependencies of target seal
[ 55%] Building CXX object CMakeFiles/seal.dir/native/src/seal/batchencoder.cpp.o
[ 55%] Building CXX object CMakeFiles/seal.dir/native/src/seal/ciphertext.cpp.o
[ 56%] Building CXX object CMakeFiles/seal.dir/native/src/seal/ckks.cpp.o
[ 58%] Building CXX object CMakeFiles/seal.dir/native/src/seal/context.cpp.o
[ 59%] Building CXX object CMakeFiles/seal.dir/native/src/seal/decryptor.cpp.o
[ 60%] Building CXX object CMakeFiles/seal.dir/native/src/seal/encryptionparams.cpp.o
[ 62%] Building CXX object CMakeFiles/seal.dir/native/src/seal/encryptor.cpp.o
[ 63%] Building CXX object CMakeFiles/seal.dir/native/src/seal/evaluator.cpp.o
[ 64%] Building CXX object CMakeFiles/seal.dir/native/src/seal/keygenerator.cpp.o
[ 65%] Building CXX object CMakeFiles/seal.dir/native/src/seal/kswitchkeys.cpp.o
[ 67%] Building CXX object CMakeFiles/seal.dir/native/src/seal/memorymanager.cpp.o
[ 67%] Building CXX object CMakeFiles/seal.dir/native/src/seal/modulus.cpp.o
[ 68%] Building CXX object CMakeFiles/seal.dir/native/src/seal/plaintext.cpp.o
[ 69%] Building CXX object CMakeFiles/seal.dir/native/src/seal/randomgen.cpp.o
[ 70%] Building CXX object CMakeFiles/seal.dir/native/src/seal/serialization.cpp.o
[ 72%] Building CXX object CMakeFiles/seal.dir/native/src/seal/valcheck.cpp.o
[ 73%] Building C object CMakeFiles/seal.dir/native/src/seal/util/blake2b.c.o
[ 74%] Building C object CMakeFiles/seal.dir/native/src/seal/util/blake2xb.c.o
[ 75%] Building CXX object CMakeFiles/seal.dir/native/src/seal/util/clipnormal.cpp.o
[ 77%] Building CXX object CMakeFiles/seal.dir/native/src/seal/util/common.cpp.o
[ 78%] Building CXX object CMakeFiles/seal.dir/native/src/seal/util/croots.cpp.o
[ 78%] Building C object CMakeFiles/seal.dir/native/src/seal/util/fips202.c.o
[ 79%] Building CXX object CMakeFiles/seal.dir/native/src/seal/util/globals.cpp.o
[ 81%] Building CXX object CMakeFiles/seal.dir/native/src/seal/util/galois.cpp.o
[ 82%] Building CXX object CMakeFiles/seal.dir/native/src/seal/util/hash.cpp.o
[ 83%] Building CXX object CMakeFiles/seal.dir/native/src/seal/util/iterator.cpp.o
[ 84%] Building CXX object CMakeFiles/seal.dir/native/src/seal/util/mempool.cpp.o
[ 86%] Building CXX object CMakeFiles/seal.dir/native/src/seal/util/numth.cpp.o
[ 87%] Building CXX object CMakeFiles/seal.dir/native/src/seal/util/polyarithsmallmod.cpp.o
[ 88%] Building CXX object CMakeFiles/seal.dir/native/src/seal/util/rlwe.cpp.o
[ 89%] Building CXX object CMakeFiles/seal.dir/native/src/seal/util/rns.cpp.o
[ 89%] Building CXX object CMakeFiles/seal.dir/native/src/seal/util/scalingvariant.cpp.o
[ 91%] Building CXX object CMakeFiles/seal.dir/native/src/seal/util/ntt.cpp.o
[ 92%] Building CXX object CMakeFiles/seal.dir/native/src/seal/util/streambuf.cpp.o
[ 93%] Building CXX object CMakeFiles/seal.dir/native/src/seal/util/uintarith.cpp.o
[ 94%] Building CXX object CMakeFiles/seal.dir/native/src/seal/util/uintarithmod.cpp.o
[ 96%] Building CXX object CMakeFiles/seal.dir/native/src/seal/util/uintarithsmallmod.cpp.o
[ 97%] Building CXX object CMakeFiles/seal.dir/native/src/seal/util/uintcore.cpp.o
[ 98%] Building CXX object CMakeFiles/seal.dir/native/src/seal/util/ztools.cpp.o
[100%] Linking CXX static library lib/libseal-4.1.a
[100%] Built target seal
root@LAPTOP-MMLD67TI:/home/Luhaozhe/Lab02/seal/SEAL#

```

图 4.4: make 得到的结果

说明我们 make 成功。接下来，我们输入命令行：

```
1 sudo make install
```

得到以下结果：

```

-- Installing: /usr/local/include/SEAL-4.1/seal/util/iterator.h
-- Installing: /usr/local/include/SEAL-4.1/seal/util/locks.h
-- Installing: /usr/local/include/SEAL-4.1/seal/util/mempool.h
-- Installing: /usr/local/include/SEAL-4.1/seal/util/msvc.h
-- Installing: /usr/local/include/SEAL-4.1/seal/util/numth.h
-- Installing: /usr/local/include/SEAL-4.1/seal/util/pointer.h
-- Installing: /usr/local/include/SEAL-4.1/seal/util/polyarithsmallmod.h
-- Installing: /usr/local/include/SEAL-4.1/seal/util/polycore.h
-- Installing: /usr/local/include/SEAL-4.1/seal/util/rlwe.h
-- Installing: /usr/local/include/SEAL-4.1/seal/util/rns.h
-- Installing: /usr/local/include/SEAL-4.1/seal/util/scalingvariant.h
-- Installing: /usr/local/include/SEAL-4.1/seal/util/ntt.h
-- Installing: /usr/local/include/SEAL-4.1/seal/util/streambuf.h
-- Installing: /usr/local/include/SEAL-4.1/seal/util/uintarith.h
-- Installing: /usr/local/include/SEAL-4.1/seal/util/uintarithmod.h
-- Installing: /usr/local/include/SEAL-4.1/seal/util/uintarithsmallmod.h
-- Installing: /usr/local/include/SEAL-4.1/seal/util/uintcore.h
-- Installing: /usr/local/include/SEAL-4.1/seal/util/ztools.h
root@LAPTOP-MMLD67TI:/home/Luhaozhe/Lab02/seal/SEAL#

```

图 4.5: sudo make install 得到的结果

显示如下，说明我们的安装就完成了！

4.2 简单测试程序

为了验证 SEAL 的 C++ 库已经完成安装，我们进行一下简单的测试。首先，我们在 SEAL 同级下建立 demo 文件夹，写入代码如下所示，并保存为 test.cpp 文件。

```
1  #include "seal/seal.h"
2  #include <iostream>
3
4  using namespace std;
5  using namespace seal;
6
7  int main(){
8
9      EncryptionParameters parms(scheme_type::bfv);
10     printf("hellow world\n");
11     return 0;
12 }
```

通过分析代码可得，代码通过调用 EncryptionParameters，即 SEAL 头文件中的类来验证功能。由于此句在 hellow world 之前，因此如果 SEAL 库成功安装，就会正常输出 hellow world，而不报任何错误。

然后，为了完成 test.cpp 的编译和执行，需要编写一个 CMakeLists.txt 文件，内容如下：

```
1  cmake_minimum_required(VERSION 3.10)
2  project(demo)
3  add_executable(test test.cpp)
4  add_compile_options(-std=c++17)
5  find_package(SEAL)
6  target_link_libraries(test SEAL::seal)
```

编写完毕后，打开控制台，依次运行：

```
1  cmake .
2  make
3  ./test
```

运行结果如下所示：


```
root@LAPTOP-MMLD67TI:/home/luhaozhe/Lab02/seal/demo# cmake .
-- The C compiler identification is GNU 9.4.0
-- The CXX compiler identification is GNU 9.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Looking for pthread.h
-- Looking for pthread.h - found
-- Performing Test CMAKE_HAVE_LIBC_PTHREAD
-- Performing Test CMAKE_HAVE_LIBC_PTHREAD - Failed
-- Check if compiler accepts -pthread
-- Check if compiler accepts -pthread - yes
-- Found Threads: TRUE
-- Microsoft SEAL -> Version 4.1.2 detected
-- Microsoft SEAL -> Targets available: SEAL::seal
-- Configuring done
-- Generating done
-- Build files have been written to: /home/luhaozhe/Lab02/seal/demo
root@LAPTOP-MMLD67TI:/home/luhaozhe/Lab02/seal/demo# make
Scanning dependencies of target test
[ 50%] Building CXX object CMakeFiles/test.dir/test.cpp.o
[100%] Linking CXX executable test
[100%] Built target test
root@LAPTOP-MMLD67TI:/home/luhaozhe/Lab02/seal/demo# ./test
hellow world
root@LAPTOP-MMLD67TI:/home/luhaozhe/Lab02/seal/demo#
```

图 4.6: 验证

我们发现，成功输出“hellow world”，说明 SEAL 安装成功！

4.3 复现 $x * y * z$ 的运算

首先，为了更好地了解同态加密的原理和代码部分的实现，我们先尝试复现课本中提供的 $x * y * z$ 的运算。代码如下所示：

```
1  #include "examples.h"
2  /* 该文件可以在 SEAL/native/example 目录下找到 */
3  #include <vector>
4  using namespace std;
5  using namespace seal;
6  #define N 3
7  //本例目的：给定 x, y, z 三个数的密文，让服务器计算 x*y*z
8
9  int main(){
10  //初始化要计算的原始数据
11  vector<double> x, y, z;
12      x = { 1.0, 2.0, 3.0 };
13      y = { 2.0, 3.0, 4.0 };
14      z = { 3.0, 4.0, 5.0 };
```

```
15
16 /*****
17 客户端的视角：生成参数、构建环境和生成密文
18 *****/
19 // (1) 构建参数容器 parms
20 EncryptionParameters parms(scheme_type::ckks);
21 /*CKKS 有三个重要参数：
22 1.poly_module_degree(多项式模数)
23 2.coeff_modulus (参数模数)
24 3.scale (规模) */
25
26 size_t poly_modulus_degree = 8192;
27 parms.set_poly_modulus_degree(poly_modulus_degree);
28 parms.set_coeff_modulus(CoeffModulus::Create(poly_modulus_degree, { 60, 40, 40, 60
    ↪ }));
29 //选用 2~40 进行编码
30 double scale = pow(2.0, 40);
31
32 // (2) 用参数生成 CKKS 框架 context
33 SEALContext context(parms);
34
35 // (3) 构建各模块
36 //首先构建 keygenerator, 生成公钥、私钥
37 KeyGenerator keygen(context);
38 auto secret_key = keygen.secret_key();
39 PublicKey public_key;
40     keygen.create_public_key(public_key);
41
42 //构建编码器, 加密模块、运算器和解密模块
43 //注意加密需要公钥 pk; 解密需要私钥 sk; 编码器需要 scale
44     Encryptor encryptor(context, public_key);
45     Decryptor decryptor(context, secret_key);
46
47     CKKSEncoder encoder(context);
48 //对向量 x、y、z 进行编码
49     Plaintext xp, yp, zp;
50     encoder.encode(x, scale, xp);
51     encoder.encode(y, scale, yp);
52     encoder.encode(z, scale, zp);
53 //对明文 xp、yp、zp 进行加密
54     Ciphertext xc, yc, zc;
55     encryptor.encrypt(xp, xc);
```

```
56     encryptor.encrypt(yp, yc);
57     encryptor.encrypt(zp, zc);
58
59
60 //至此，客户端将 pk、CKKS 参数发送给服务器，服务器开始运算
61 /*****
62 服务器的视角：生成重线性密钥、构建环境和执行密文计算
63 *****/
64 //生成重线性密钥和构建环境
65 SEALContext context_server(parms);
66     RelinKeys relin_keys;
67     keygen.create_relin_keys(relin_keys);
68     Evaluator evaluator(context_server);
69
70 /* 对密文进行计算，要说明的原则是：
71 -加法可以连续运算，但乘法不能连续运算
72 -密文乘法后要进行 relinearize 操作
73 -执行乘法后要进行 rescaling 操作
74 -进行运算的密文必需执行过相同次数的 rescaling (位于相同 level) */
75     Ciphertext temp;
76     Ciphertext result_c;
77 //计算  $x*y$ ，密文相乘，要进行 relinearize 和 rescaling 操作
78     evaluator.multiply(xc, yc, temp);
79     evaluator.relinearize_inplace(temp, relin_keys);
80     evaluator.rescale_to_next_inplace(temp);
81
82 //在计算  $x*y * z$  之前， $z$  没有进行过 rescaling 操作，所以需要对其进行一次乘法和 rescaling 操作，
83 ↪ 目的是使得  $x*y$  和  $z$  在相同的层
84     Plaintext wt;
85     encoder.encode(1.0, scale, wt);
86 //此时，我们可以查看框架中不同数据的层级：
87 cout << "    + Modulus chain index for zc: "
88 << context_server.get_context_data(zc.parms_id())->chain_index() << endl;
89 cout << "    + Modulus chain index for temp( $x*y$ ): "
90 << context_server.get_context_data(temp.parms_id())->chain_index() << endl;
91 cout << "    + Modulus chain index for wt: "
92 << context_server.get_context_data(wt.parms_id())->chain_index() << endl;
93
94 //执行乘法和 rescaling 操作：
95     evaluator.multiply_plain_inplace(zc, wt);
96     evaluator.rescale_to_next_inplace(zc);
```

```

97 //再次查看 zc 的层级，可以发现 zc 与 temp 层级变得相同
98 cout << "      + Modulus chain index for zc after zc*wt and rescaling: "
99 << context_server.get_context_data(zc.parms_id())->chain_index() << endl;
100
101 //最后执行 temp (x*y) * zc (z*1.0)
102 evaluator.multiply_inplace(temp, zc);
103 evaluator.relinearize_inplace(temp, relin_keys);
104 evaluator.rescale_to_next(temp, result_c);
105
106
107 //计算完毕，服务器把结果发回客户端
108 /*****
109 客户端的视角：进行解密和解码
110 *****/
111 //客户端进行解密
112 Plaintext result_p;
113 decryptor.decrypt(result_c, result_p);
114 //注意要解码到一个向量上
115 vector<double> result;
116 encoder.decode(result_p, result);
117 //得到结果，正确的话将输出：{6.000, 24.000, 60.000, ..., 0.000, 0.000, 0.000}
118 cout << " 结果是: " << endl;
119 print_vector(result, 3, 3);
120 return 0;
121 }

```

将 seal 下的 native 下的 examples 下的 example.h 复制到 Demo 文件夹下；这个头文件定义了使用 seal 的常见头文件，并定义了一些输出函数。

然后，我们定义文件 ckks_example.cpp，并将源代码复制到该文件。

我们还需要更改 CMakeLists.txt 内容：

```

1 cmake_minimum_required(VERSION 3.10)
2 project(demo)
3 add_executable(he ckks_example.cpp)
4 add_compile_options(-std=c++17)
5
6 find_package(SEAL)
7 target_link_libraries(he SEAL::seal)

```

编写完毕后，打开控制台，依次运行三条语句即可。运行结果如下所示：

```
root@LAPTOP-MMLD67TI:/home/luhaozhe/Lab02/seal/demo# cmake .
-- Microsoft SEAL -> Version 4.1.2 detected
-- Microsoft SEAL -> Targets available: SEAL::seal
-- Configuring done
-- Generating done
-- Build files have been written to: /home/luhaozhe/Lab02/seal/demo
root@LAPTOP-MMLD67TI:/home/luhaozhe/Lab02/seal/demo# make
Scanning dependencies of target he
[ 50%] Building CXX object CMakeFiles/he.dir/ckks_example.cpp.o
[100%] Linking CXX executable he
[100%] Built target he
root@LAPTOP-MMLD67TI:/home/luhaozhe/Lab02/seal/demo# ./he
+ Modulus chain index for zc: 2
+ Modulus chain index for temp(x*y): 1
+ Modulus chain index for wt: 2
+ Modulus chain index for zc after zc*wt and rescaling: 1
结果是:

[ 6.000, 24.000, 60.000, ..., 0.000, 0.000, -0.000 ]

root@LAPTOP-MMLD67TI:/home/luhaozhe/Lab02/seal/demo#
```

图 4.7: 计算结果验证

我们发现，计算出的结果都是正确的，说明程序运行成功！

4.4 完成 $x^3 + y * z$ 的运算

在完成了 $x * y * z$ 的程序验证之后，我们开始实现 $x^3 + y * z$ 的运算。

```
1  #include "examples.h"
2  #include <iostream>
3  #include <vector>
4  #include <cmath>
5  #include "seal/seal.h"
6
7  using namespace std;
8  using namespace seal;
9
10 // 定义常量 N
11 const int N = 3;
12
13 // 打印向量函数
14 void printVector(const vector<double>& vec, int precision = 3) {
15     cout << fixed << setprecision(precision);
16     cout << "{ ";
17     for (size_t i = 0; i < vec.size(); ++i) {
18         cout << vec[i];
19         if (i < vec.size() - 1) {
20             cout << ", ";
21         }
22     }
23 }
```

```
23     cout << " }" << endl;
24 }
25
26 // 打印当前行号及信息
27 void printCurrentLine(int line, const string& message = "") {
28     cout << "Line " << line << ": " << message << endl;
29 }
30
31 int main() {
32     // 初始化待计算的向量
33     vector<double> vectorX = {1.0, 2.0, 3.0};
34     vector<double> vectorY = {2.0, 3.0, 4.0};
35     vector<double> vectorZ = {3.0, 4.0, 5.0};
36
37     // 输出原始向量信息
38     cout << " 原始向量 x 为: " << endl;
39     printVector(vectorX);
40     cout << " 原始向量 y 为: " << endl;
41     printVector(vectorY);
42     cout << " 原始向量 z 为: " << endl;
43     printVector(vectorZ);
44     cout << endl;
45
46     // 配置加密参数
47     EncryptionParameters encryptionParams(scheme_type::ckks);
48     size_t polynomialModulusDegree = 8192;
49     encryptionParams.set_poly_modulus_degree(polynomialModulusDegree);
50     encryptionParams.set_coeff_modulus(CoeffModulus::Create(polynomialModulusDegree,
51     ↪ e, {60, 40, 40,
52     ↪ 60}));
53     double encodingScale = pow(2.0, 40);
54
55     // 创建 CKKS 上下文
56     SEALContext sealContext(encryptionParams);
57
58     // 生成密钥
59     KeyGenerator keyGenerator(sealContext);
60     auto privateKey = keyGenerator.secret_key();
61     PublicKey publicKey;
62     keyGenerator.create_public_key(publicKey);
63     RelinKeys relinearizationKeys;
64     keyGenerator.create_relin_keys(relinearizationKeys);
```

```
63
64 // 构建加密、解密、编码和评估器
65 Encryptor encryptor(sealContext, publicKey);
66 Evaluator evaluator(sealContext);
67 Decryptor decryptor(sealContext, privateKey);
68 CKKSEncoder encoder(sealContext);
69
70 // 对向量进行编码
71 Plaintext plainX, plainY, plainZ;
72 encoder.encode(vectorX, encodingScale, plainX);
73 encoder.encode(vectorY, encodingScale, plainY);
74 encoder.encode(vectorZ, encodingScale, plainZ);
75
76 // 对编码后的明文进行加密
77 Ciphertext encryptedX, encryptedY, encryptedZ;
78 encryptor.encrypt(plainX, encryptedX);
79 encryptor.encrypt(plainY, encryptedY);
80 encryptor.encrypt(plainZ, encryptedZ);
81
82 // 计算  $x^2$ 
83 printCurrentLine(__LINE__, " 开始计算  $x^2$ ");
84 Ciphertext squaredX;
85 evaluator.multiply(encryptedX, encryptedX, squaredX);
86 evaluator.relinearize_inplace(squaredX, relinearizationKeys);
87 evaluator.rescale_to_next_inplace(squaredX);
88 printCurrentLine(__LINE__, " $x^2$  的模数链索引为: " +
89     to_string(sealContext.get_context_data(squaredX.parms_id())->chain_index)
90     ↪ );
91
92 // 调整 encryptedX 的层级
93 printCurrentLine(__LINE__, "encryptedX 的模数链索引为: " +
94     to_string(sealContext.get_context_data(encryptedX.parms_id())->chain_index)
95     ↪ );
96 printCurrentLine(__LINE__, " 开始计算  $1.0 * x$  以调整层级");
97 Plaintext plainOne;
98 encoder.encode(1.0, encodingScale, plainOne);
99 evaluator.multiply_plain_inplace(encryptedX, plainOne);
100 evaluator.rescale_to_next_inplace(encryptedX);
101 printCurrentLine(__LINE__, " 调整后 encryptedX 的模数链索引为: " +
    to_string(sealContext.get_context_data(encryptedX.parms_id())->chain_index)
    ↪ );
```

```
102 // 计算  $x^3$ 
103 printCurrentLine(__LINE__, " 开始计算  $x^3$ ");
104 Ciphertext cubedX;
105 evaluator.multiply(squaredX, encryptedX, cubedX);
106 evaluator.relinearize_inplace(cubedX, relinearizationKeys);
107 evaluator.rescale_to_next_inplace(cubedX);
108 printCurrentLine(__LINE__, " $x^3$  的模数链索引为: " +
109     to_string(sealContext.get_context_data(cubedX.parms_id())->chain_index()));
110
111 // 计算  $y * z$ 
112 printCurrentLine(__LINE__, " 开始计算  $y * z$ ");
113 Ciphertext productYZ;
114 evaluator.multiply(encryptedY, encryptedZ, productYZ);
115 evaluator.relinearize_inplace(productYZ, relinearizationKeys);
116 evaluator.rescale_to_next_inplace(productYZ);
117 printCurrentLine(__LINE__, " $y * z$  的模数链索引为: " +
118     to_string(sealContext.get_context_data(productYZ.parms_id())->chain_index(
119         ↪ )));
120
121 // 统一 scale
122 printCurrentLine(__LINE__, " 将  $x^3$  和  $y * z$  的 scale 统一为  $2^{40}$ ");
123 cubedX.scale() = encodingScale;
124 productYZ.scale() = encodingScale;
125 printCurrentLine(__LINE__, " $x^3$  的精确 scale 为: " +
126     ↪ to_string(cubedX.scale()));
127 printCurrentLine(__LINE__, " $y * z$  的精确 scale 为: " +
128     ↪ to_string(productYZ.scale()));
129
130 // 调整  $y * z$  的层级与  $x^3$  一致
131 parms_id_type targetParamsId = cubedX.parms_id();
132 evaluator.mod_switch_to_inplace(productYZ, targetParamsId);
133 printCurrentLine(__LINE__, " 调整后  $y * z$  的模数链索引为: " +
134     to_string(sealContext.get_context_data(productYZ.parms_id())->chain_index(
135         ↪ )));
136
137 // 计算  $x^3 + y * z$ 
138 printCurrentLine(__LINE__, " 开始计算  $x^3 + y * z$ ");
139 Ciphertext encryptedResult;
140 evaluator.add(cubedX, productYZ, encryptedResult);
141
142 // 解密结果
143 Plaintext plainResult;
```



```
140     decryptor.decrypt(encryptedResult, plainResult);
141
142     // 解码结果
143     vector<double> result;
144     encoder.decode(plainResult, result);
145
146     // 输出最终结果
147     printCurrentLine(__LINE__, " 计算结果为: ");
148     print_vector(result, 3 /*precision*/);
149
150     return 0;
151 }
```

我们首先来分析一下我们编写的代码。实际上我们只需要修改核心部分的代码，也就是修改算式的计算过程即可。

本段代码的主要流程如下所示：

1. 数据初始化

```
1 vector<double> vectorX = {1.0, 2.0, 3.0};
2 vector<double> vectorY = {2.0, 3.0, 4.0};
3 vector<double> vectorZ = {3.0, 4.0, 5.0};
```

我们创建并初始化三个向量 vectorX、vectorY、vectorZ，作为后续待计算的原始数据。

2. 加密相关准备工作

- 参数配置

```
1 EncryptionParameters encryptionParams(scheme_type::ckks);
2 size_t polynomialModulusDegree = 8192;
3 encryptionParams.set_poly_modulus_degree(polynomialModulusDegree);
4 encryptionParams.set_coeff_modulus(CoeffModulus::Create(polynomialModulusDegree,
5   ↪ Degree, {60, 40, 40,
6   ↪ 60}));
7 double encodingScale = pow(2.0, 40);
```

选择 CKKS 同态加密方案，设置多项式模数的度数为 8192，系数模数以及编码缩放因子 encodingScale 为 2^{40} ，这些参数决定了加密计算的精度和性能。

- 创建上下文和密钥

```
1 SEALContext sealContext(encryptionParams);
2 KeyGenerator keyGenerator(sealContext);
3 auto privateKey = keyGenerator.secret_key();
4 PublicKey publicKey;
5 keyGenerator.create_public_key(publicKey);
```

```
6 RelinKeys relinearizationKeys;  
7 keyGenerator.create_relin_keys(relinearizationKeys);
```

根据配置的参数创建加密上下文 sealContext，并利用 KeyGenerator 生成私钥 privateKey、公钥 publicKey 和重线性化密钥 relinearizationKeys。

- 构建加密模块

```
1 Encryptor encryptor(sealContext, publicKey);  
2 Evaluator evaluator(sealContext);  
3 Decryptor decryptor(sealContext, privateKey);  
4 CKKSEncoder encoder(sealContext);
```

分别构建加密器 encryptor、评估器（用于密文运算）evaluator、解密器 decryptor 和编码器 encoder，为后续操作提供支持。

3. 数据编码与加密

- 编码

```
1 Plaintext plainX, plainY, plainZ;  
2 encoder.encode(vectorX, encodingScale, plainX);  
3 encoder.encode(vectorY, encodingScale, plainY);  
4 encoder.encode(vectorZ, encodingScale, plainZ);
```

使用 encoder 将向量 vectorX、vectorY、vectorZ 编码为明文 plainX、plainY、plainZ，编码过程中使用了之前设定的 encodingScale。

- 加密

```
1 Ciphertext encryptedX, encryptedY, encryptedZ;  
2 encryptor.encrypt(plainX, encryptedX);  
3 encryptor.encrypt(plainY, encryptedY);  
4 encryptor.encrypt(plainZ, encryptedZ);
```

通过 encryptor 利用公钥 publicKey 将明文 plainX、plainY、plainZ 加密为密文 encryptedX、encryptedY、encryptedZ。

4. 密文计算

下面就是最关键的密文计算部分了。主要步骤如下所示：

- 计算 x^2

```
1 Ciphertext squaredX;  
2 evaluator.multiply(encryptedX, encryptedX, squaredX);  
3 evaluator.relinearize_inplace(squaredX, relinearizationKeys);  
4 evaluator.rescale_to_next_inplace(squaredX);
```

使用评估器 evaluator 将密文 encryptedX 自乘得到 x^2 的密文 squaredX。由于乘法操作可能导致密文结构变得复杂，通过重线性化操作（relinearize_inplace）简化密文，再进行缩放操作（rescale_to_next_inplace），确保后续计算的可行性。

- 调整 encryptedX 层级

```
1 Plaintext plainOne;  
2 encoder.encode(1.0, encodingScale, plainOne);  
3 evaluator.multiply_plain_inplace(encryptedX, plainOne);  
4 evaluator.rescale_to_next_inplace(encryptedX);
```

我们发现 encryptedX 与 squaredX 的层级不一致, 为使两者能够相乘计算 x^3 , 先将 1.0 编码为明文 plainOne, 然后让 encryptedX 与 plainOne 相乘, 并进行缩放操作, 降低 encryptedX 的层级, 使其与 squaredX 层级相同。

- 计算 x^3

```
1 Ciphertext cubedX;  
2 evaluator.multiply(squaredX, encryptedX, cubedX);  
3 evaluator.relinearize_inplace(cubedX, relinearizationKeys);  
4 evaluator.rescale_to_next_inplace(cubedX);
```

将调整层级后的 encryptedX 与 squaredX 相乘得到 x^3 的密文 cubedX, 同样进行重线性化和缩放操作, 保证密文处于可处理状态。

- 计算 $y * z$

```
1 Ciphertext productYZ;  
2 evaluator.multiply(encryptedY, encryptedZ, productYZ);  
3 evaluator.relinearize_inplace(productYZ, relinearizationKeys);  
4 evaluator.rescale_to_next_inplace(productYZ);
```

使用评估器将密文 encryptedY 和 encryptedZ 相乘得到 $y * z$ 的密文 productYZ, 随后进行重线性化和缩放操作。

- 统一 scale 与调整层级

```
1 cubedX.scale() = encodingScale;  
2 productYZ.scale() = encodingScale;  
3  
4 parms_id_type targetParamsId = cubedX.parms_id();  
5 evaluator.mod_switch_to_inplace(productYZ, targetParamsId);
```

为了能够对 x^3 和 $y * z$ 的密文进行加法运算, 先统一它们的 scale 为 encodingScale, 再通过 mod_switch_to_inplace 操作, 将 productYZ 的层级调整为与 cubedX 一致。

- 计算 $x^3 + y * z$

```
1 Ciphertext encryptedResult;  
2 evaluator.add(cubedX, productYZ, encryptedResult);
```

使用评估器将调整好的 cubedX 和 productYZ 相加, 得到最终计算结果的密文 encryptedResult。

5. 解密与结果输出

```
1 Plaintext plainResult;  
2 decryptor.decrypt(encryptedResult, plainResult);  
3 vector<double> result;  
4 encoder.decode(plainResult, result);
```

利用解密器 decryptor 和私钥 privateKey 对最终密文 encryptedResult 进行解密, 得到明文 plainResult, 再通过编码器 encoder 解码得到计算结果向量 result 并输出。

然后, 我们重新进行 cmake、make 的操作, 得到 he 可执行文件。执行 ./he, 得到的结果如下所示:

```
root@LAPTOP-MMLD67TI:/home/luhaozhe/Lab02/seal/demo# cmake .  
-- Microsoft SEAL -> Version 4.1.2 detected  
-- Microsoft SEAL -> Targets available: SEAL::seal  
-- Configuring done  
-- Generating done  
-- Build files have been written to: /home/luhaozhe/Lab02/seal/demo  
root@LAPTOP-MMLD67TI:/home/luhaozhe/Lab02/seal/demo# make  
Scanning dependencies of target he  
[ 50%] Building CXX object CMakeFiles/he.dir/ckks_example.cpp.o  
[100%] Linking CXX executable he  
[100%] Built target he  
root@LAPTOP-MMLD67TI:/home/luhaozhe/Lab02/seal/demo# ./he  
原始向量 x 为:  
{ 1.000, 2.000, 3.000 }  
原始向量 y 为:  
{ 2.000, 3.000, 4.000 }  
原始向量 z 为:  
{ 3.000, 4.000, 5.000 }  
  
Line 83: 开始计算 x^2  
Line 88: x^2 的模数链索引为: 1  
Line 92: encryptedX 的模数链索引为: 2  
Line 94: 开始计算 1.0 * x 以调整层级  
Line 99: 调整后 encryptedX 的模数链索引为: 1  
Line 103: 开始计算 x^3  
Line 108: x^3 的模数链索引为: 0  
Line 112: 开始计算 y * z  
Line 117: y * z 的模数链索引为: 1  
Line 121: 将 x^3 和 y * z 的 scale 统一为 2^40  
Line 124: x^3 的精确 scale 为: 1099511627776.000000  
Line 125: y * z 的精确 scale 为: 1099511627776.000000  
Line 130: 调整后 y * z 的模数链索引为: 0  
Line 134: 开始计算 x^3 + y * z  
Line 147: 计算结果为:  
  
[ 7.000, 20.000, 47.000, ..., 0.000, -0.000, 0.000 ]  
  
root@LAPTOP-MMLD67TI:/home/luhaozhe/Lab02/seal/demo#
```

图 4.8: 修改后得到的结果

易得: $7 = 1 \times 1 + 2 \times 3$, $20 = 2 \times 2 + 3 \times 4$, $47 = 3 \times 3 + 4 \times 5$, 与程序输出的结果相同, 说明我们的程序验证成功!

5 实验心得与体会

本次实验, 我基于教材中的步骤, 完成了实验。主要有以下收获:

- 安装了 WSL 来完成实验，并且对 WSL 更加熟悉了；
- 熟悉了官方库 SEAL 的加密方式和调用加密方式；
- 基于课本上的计算范例，自己完成了任务的编写，让我对同态加密的原理理解更加深刻了。

总的来说，本次实验我收获颇丰，希望在后续实验中可以学到更多的数据安全的知识。