

# 南开大学

## 数据安全课程实验报告

### SEAL应用实践



学院：网络空间安全学院

专业：信息安全

学号：2113997

姓名：齐明杰

班级：信安2班

# 目录

- 1 实验目的
- 2 实验原理
  - 2.1 开发框架SEAL (Simple Encrypted Arithmetic Library)
  - 2.2 CKKS算法 (Cheon-Kim-Kim-Song)
  - 2.3 标准化构建流程
- 3 实验过程
  - 3.1 SEAL库安装
    - 3.1.1 git clone 加密库资源
    - 3.1.2 编译和安装
  - 3.2 实验代码
  - 3.3 编译运行
- 4 实验心得

# 1 实验目的

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 $x^3 + y * z$ 的运算。

# 2 实验原理

## 2.1 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。

## 2.2 CKKS算法 (Cheon-Kim-Kim-Song)

CKKS算法是一种针对近似数（即实数和复数）计算优化的同态加密方案。由Cheon、Kim、Kim和Song在2017年提出。该算法特别适合处理浮点数运算，使得在加密数据上进行复杂的数学和统计分析成为可能。

主要特点:

- **处理近似数:** CKKS设计之初就考虑了在加密状态下直接处理近似数的能力，这对于科学计算、机器学习等需要大量浮点运算的应用尤其重要。
- **效率和精度:** 通过对近似数的特殊处理，CKKS可以在保持合理计算效率的同时，控制计算过程中的误差积累。
- **适用性:** 适用于需要处理大规模浮点数数据集的场景，如机器学习模型的训练和预测、统计分析等。

应用场景:

- 加密机器学习：在不解密数据的前提下，对机器学习模型进行训练和预测。
- 科学计算：在加密的数据上进行复杂的数学和物理模拟。
- 数据分析：对加密的统计数据进行处理和分析，保护个人隐私。

## 2.3 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 $n$

## 3 实验过程

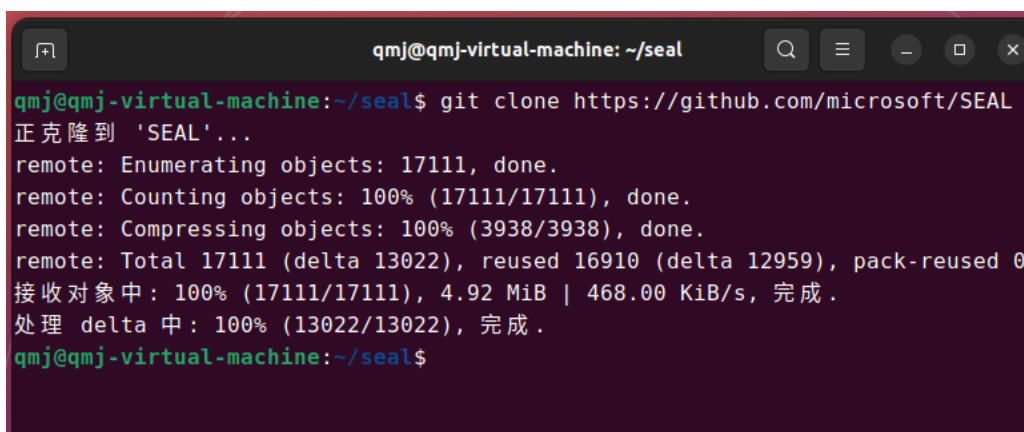
### 3.1 SEAL库安装

#### 3.1.1 git clone 加密库资源

在Ubuntu的home文件夹下建立文件夹seal，进入该文件夹后，打开终端，输入命令：

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

运行完毕，将在seal文件夹下自动建立SEAL这个新文件夹。

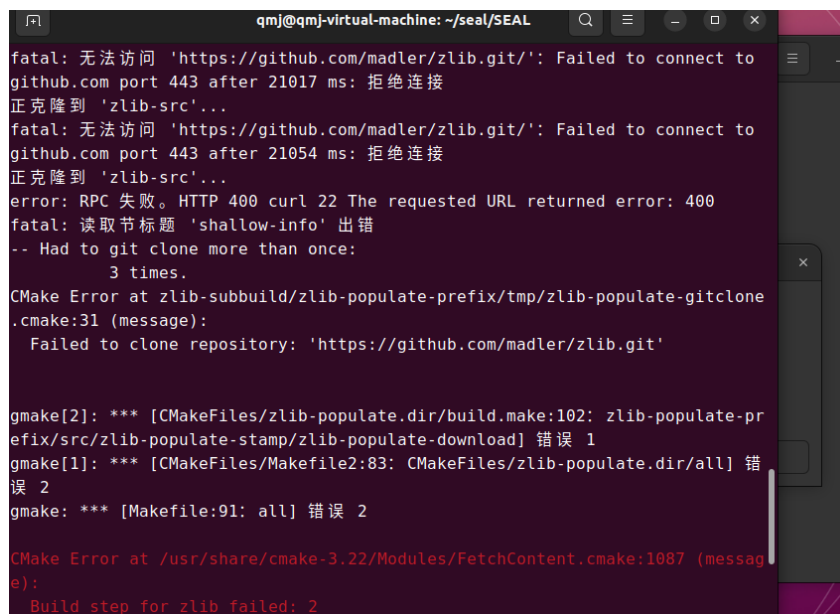


```
qmj@qmj-virtual-machine: ~/seal
qmj@qmj-virtual-machine:~/seal$ git clone https://github.com/microsoft/SEAL
正克隆到 'SEAL'...
remote: Enumerating objects: 17111, done.
remote: Counting objects: 100% (17111/17111), done.
remote: Compressing objects: 100% (3938/3938), done.
remote: Total 17111 (delta 13022), reused 16910 (delta 12959), pack-reused 0
接收对象中: 100% (17111/17111), 4.92 MiB | 468.00 KiB/s, 完成.
处理 delta 中: 100% (13022/13022), 完成.
qmj@qmj-virtual-machine:~/seal$
```

#### 3.1.2 编译和安装

输入命令：

```
1 | cd SEAL
2 | cmake .
```



```
qmj@qmj-virtual-machine: ~/seal/SEAL
fatal: 无法访问 'https://github.com/madler/zlib.git/': Failed to connect to
github.com port 443 after 21017 ms: 拒绝连接
正克隆到 'zlib-src'...
fatal: 无法访问 'https://github.com/madler/zlib.git/': Failed to connect to
github.com port 443 after 21054 ms: 拒绝连接
正克隆到 'zlib-src'...
error: RPC 失败。HTTP 400 curl 22 The requested URL returned error: 400
fatal: 读取节标题 'shallow-info' 出错
-- Had to git clone more than once:
    3 times.
CMake Error at zlib-subbuild/zlib-populate-prefix/tmp/zlib-populate-gitclone
.cmake:31 (message):
  Failed to clone repository: 'https://github.com/madler/zlib.git'

gmake[2]: *** [CMakeFiles/zlib-populate.dir/build.make:102: zlib-populate-pr
efix/src/zlib-populate-stamp/zlib-populate-download] 错误 1
gmake[1]: *** [CMakeFiles/Makefile2:83: CMakeFiles/zlib-populate.dir/all] 错
误 2
gmake: *** [Makefile:91: all] 错误 2

CMake Error at /usr/share/cmake-3.22/Modules/FetchContent.cmake:1087 (messag
e):
  Build step for zlib failed: 2
```

网络原因可能会报错，多尝试几次。该步骤成功后显示如下：

```

- 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 - Success
- 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/qmj/seal/SEAL
qmj@qmj-virtual-machine: ~/seal/SEAL$

```

1 | `make`

显示如下:

```

[ 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/uintarithsmallmod.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
qmj@qmj-virtual-machine: ~/seal/SEAL$

```

1 | `sudo make install`

最后一步成功后显示如下:

```

-- Installing: /usr/local/include/SEAL-4.1/seal/util/nas.h
-- Installing: /usr/local/include/SEAL-4.1/seal/util/hestdparms.h
-- 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
qmj@qmj-virtual-machine: ~/seal/SEAL$

```

## 3.2 实验代码

本次实验我们需要客户端将一组数据的计算外包到云服务器上，云服务器不能得知密文相关的信息，客户端可以正确完成密文运算结果的解密。

```
1  #include "examples.h"
2  #include <vector>
3  using namespace std;
4  using namespace seal;
5  #define N 3
6  int main()
7  {
8
9      // 客户端的视角：要进行计算的数据
10     vector<double> x, y, z;
11     x = { 1.0, 2.0, 3.0 };
12     y = { 2.0, 3.0, 4.0 };
13     z = { 3.0, 4.0, 5.0 };
14     cout<<"原始向量x是： ";
15     print_vector(x);
16     cout<<"原始向量y是： ";
17     print_vector(y);
18     cout<<"原始向量z是： ";
19     print_vector(z);
20     cout<<endl;
21     // 构建参数容器 parms
22     EncryptionParameters parms(scheme_type::ckks);
23     // 这里的参数都使用官方建议的
24     size_t poly_modulus_degree = 8192;
25     parms.set_poly_modulus_degree(poly_modulus_degree);
26     parms.set_coeff_modulus(CoeffModulus::Create(poly_modulus_degree, {
60, 40, 40, 60 }));
27     double scale = pow(2.0, 40);
28
29     // 用参数生成 CKKS 框架 context
30     SEALContext context(parms);
31
32     // 构建各模块
33     // 生成公钥、私钥和重线性化密钥
34     KeyGenerator keygen(context);
35     auto secret_key = keygen.secret_key();
36     PublicKey public_key;
37     keygen.create_public_key(public_key);
38     RelinKeys relin_keys;
39     keygen.create_relin_keys(relin_keys);
40     // 构建编码器，加密模块、运算器和解密模块
```

```

41 // 注意加密需要公钥 pk; 解密需要私钥 sk; 编码器需要 scale
42 Encryptor encryptor(context, public_key);
43 Evaluator evaluator(context);
44 Decryptor decryptor(context, secret_key);
45 CKKSEncoder encoder(context);
46
47 // 对向量 x、y、z 进行编码
48 Plaintext xp, yp, zp;
49 encoder.encode(x, scale, xp);
50 encoder.encode(y, scale, yp);
51 encoder.encode(z, scale, zp);
52
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 /*
61 下面进入本次实验的核心内容
62 计算 $x^3+y*z$ 
63 */
64 // 步骤1, 计算 $x^2$ 
65     print_line(__LINE__);
66     cout << "计算  $x^2$  ." << endl;
67     Ciphertext x2;
68     evaluator.multiply(xc, xc, x2);
69     // 进行 relinearize 和 rescaling 操作
70     evaluator.relinearize_inplace(x2, relin_keys);
71     evaluator.rescale_to_next_inplace(x2);
72     // 然后查看一下此时 $x^2$ 结果的level
73     print_line(__LINE__);
74     cout << " + Modulus chain index for x2: "
75 << context.get_context_data(x2.parms_id())->chain_index() << endl;
76
77     // 步骤2, 计算 $1.0*x$ 
78     // 此时xc本身的层级应该是2, 比 $x^2$ 高, 因此这一步解决层级问题
79     print_line(__LINE__);
80     cout << " + Modulus chain index for xc: "
81 << context.get_context_data(xc.parms_id())->chain_index() << endl;
82     // 因此, 需要对 x 进行一次乘法和 rescaling操作
83     print_line(__LINE__);
84     cout << "计算  $1.0*x$  ." << endl;
85     Plaintext plain_one;
86     encoder.encode(1.0, scale, plain_one);

```



```

87 // 执行乘法和 rescaling 操作:
88 evaluator.multiply_plain_inplace(xc, plain_one);
89 evaluator.rescale_to_next_inplace(xc);
90 // 再次查看 xc 的层级, 可以发现 xc 与 x^2 层级变得相同
91 print_line(__LINE__);
92 cout << " + Modulus chain index for xc new: "
93 << context.get_context_data(xc.parms_id())->chain_index() << endl;
94 // 那么, 此时xc与x^2层级相同, 二者可以相乘了
95
96 // 步骤3, 计算x^3, 即1*x*x^2
97 // 先设置新的变量叫x3
98     print_line(__LINE__);
99     cout << "计算 1.0*x*x^2 ." << endl;
100     Ciphertext x3;
101     evaluator.multiply_inplace(x2, xc);
102     evaluator.relinearize_inplace(x2, relin_keys);
103     evaluator.rescale_to_next(x2, x3);
104     // 此时观察x^3的层级
105     print_line(__LINE__);
106     cout << " + Modulus chain index for x3: "
107     << context.get_context_data(x3.parms_id())->chain_index() << endl;
108
109
110 // 步骤4, 计算y*z
111 print_line(__LINE__);
112 cout << "计算 y*z ." << endl;
113 Ciphertext yz;
114 evaluator.multiply(yz, yc, zc);
115 // 进行 relinearize 和 rescaling 操作
116 evaluator.relinearize_inplace(yz, relin_keys);
117 evaluator.rescale_to_next_inplace(yz);
118 // 然后查看一下此时y*z结果的level
119 print_line(__LINE__);
120 cout << " + Modulus chain index for yz: "
121 << context.get_context_data(yz.parms_id())->chain_index() << endl;
122
123 // 注意, 此时问题在于scales的不统一, 可以直接重制。
124 print_line(__LINE__);
125 cout << "Normalize scales to 2^40." << endl;
126 x3.scale() = pow(2.0, 40);
127 yz.scale() = pow(2.0, 40);
128 // 输出观察, 此时的scale的大小已经统一了!
129 print_line(__LINE__);
130 cout << " + Exact scale in 1*x^3: " << x3.scale() << endl;
131 print_line(__LINE__);
132 cout << " + Exact scale in y*z: " << yz.scale() << endl;

```

```

133
134 // 但是，此时还有一个问题，就是我们的x^3和yz的层级还不统一！
135 // 在官方 examples 中，给出了一个简便的变换层级的方法，如下所示：
136 parms_id_type last_parms_id = x3.parms_id();
137 evaluator.mod_switch_to_inplace(yz, last_parms_id);
138 print_line(__LINE__);
139 cout << " + Modulus chain index for yz new: "
140 << context.get_context_data(yz.parms_id())->chain_index() << endl;
141
142 // 步骤5, x^3+y*z
143 print_line(__LINE__);
144 cout << "计算 x^3+y*z ." << endl;
145 Ciphertext encrypted_result;
146 evaluator.add(x3, yz, encrypted_result);
147
148 // 计算完毕，服务器把结果发回客户端
149 Plaintext result_p;
150 decryptor.decrypt(encrypted_result, result_p);
151
152 // 注意要解码到一个向量上
153 vector<double> result;
154 encoder.decode(result_p, result);
155
156 // 输出结果
157 print_line(__LINE__);
158 cout << "结果是: " << endl;
159 print_vector(result, 3 /*precision*/);
160
161 return 0;
162 }

```

### 参数解释:

#### (1) poly\_modulus\_degree (polynomial modulus)

该参数必须是2的幂，如1024, 2048, 4096, 8192, 16384, 32768，当然再大点也没问题。

更大的poly\_modulus\_degree会增加密文的尺寸，这会让计算变慢，但也能让代码执行更复杂的计算。

#### (2) [ciphertext] coefficient modulus

这是一组重要参数，因为rescaling操作依赖于coeff\_modules。

简单来说，coeff\_modules的个数决定了你能进行rescaling的次数，进而决定了你能执行的乘法操作的次数。

coeff\_modules的最大位数与poly\_modules有直接关系。

本文例子中的{60, 40, 40, 60}有以下含义：

- ① coeff\_modules总位长200 (60+40+40+60) 位
- ② 最多进行两次（两层）乘法操作

该系列数字的选择不是随意的，有以下要求：

- ① 总位长不能超过上表限制
- ② 最后一个参数为特殊模数，其值应该与中间模数的最大值相等
- ③ 中间模数与scale尽量相近

注意：如果将模数变大，则可以支持更多层级的乘法运算，比如poly\_modulus为16384则可以支持coeff\_modules= { 60, 40, 40, 40, 40, 40, 40, 60 }，也就是6层的运算。

### (3) Scale

Encoder利用该参数对浮点数进行缩放，每次相乘后密文的scale都会翻倍，因此需要执行rescaling操作约减一部分，约模的大素数位长由coeff\_modules中的参数决定。

Scale不应太小，虽然大的scale会导致运算时间增加，但能确保噪声在约模的过程中被正确地舍去，同时不影响正确解密。

因此，两组推荐的参数为：

Poly\_module\_degree = 8196; coeff\_modulus={60,40,40,60};scale =  $2^{40}$

Poly\_module\_degree = 8196; coeff\_modulus={50,30,30,30.50};scale =  $2^{30}$

**在这里，我选用的参数与示例代码相同。**

**代码的核心部分**集中在执行同态加密计算，具体涉及计算  $x^3 + y \cdot z$  的步骤。下面是对这部分的重点解释：

#### 步骤1：计算 $x^2$

- 使用 `evaluator.multiply(xc, xc, x2);` 计算加密的向量 `x` 自乘得到 `x^2`，结果保存在 `x2` 中。
- `relinearize_inplace(x2, relin_keys);` 和 `rescale_to_next_inplace(x2);` 分别用于重线性化和重新缩放 `x2`，使其适应后续的乘法操作。这两步是必要的，因为乘法操作增加了密文的大小和层级。

#### 步骤2：调整层级

- 由于在CKKS方案中，每次乘法后都会增加密文的层级，为了让 `x^2` 能与 `x` 进行乘法操作，需要将 `x` 的层级调整至与 `x^2` 相同。
- 通过将 `x` 乘以1 (`multiply_plain_inplace(xc, plain_one);`) 并执行重新缩放 (`rescale_to_next_inplace(xc);`)，`x` 的层级被降低，使其与 `x^2` 的层级相匹配。

### 步骤3: 计算 $x^3$

- 现在 `x` 和 `x^2` 的层级相同，可以将它们相乘得到 `x^3`，保存在 `x3` 中。

### 步骤4: 计算 $y \cdot z$

- 同样地，将 `y` 和 `z` 相乘得到 `yz`。这一步也涉及重线性化和重新缩放操作，以确保操作后的密文保持合适的形式。

### 步骤5: 规范化尺度和层级，计算 $x^3 + y \cdot z$

- 为了将 `x^3` 和 `yz` 相加，首先需要确保它们的尺度(scale)相同，因此将两者的尺度统一设置为 `pow(2.0, 40)`。
- 然后，为了确保它们位于同一层级，对 `yz` 执行模数切换 (`mod_switch_to_inplace(yz, last_parms_id);`)，使其层级与 `x^3` 相匹配。
- 最后，将 `x^3` 和 `yz` 相加，得到最终的加密结果。

## 3.3 编译运行

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。

定义文件 `code.cpp`，并将源代码复制到该文件。

更改 `CMakeLists.txt` 内容：

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

编写完毕后，打开控制台，依次运行：

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

得到如下结果：

```

qmj@qmj-virtual-machine:~/seal/Demo$ cmake .
- The C compiler identification is GNU 11.4.0
- The CXX compiler identification is GNU 11.4.0
- Detecting C compiler ABI info
- Detecting C compiler ABI info - done
- Check for working C compiler: /usr/bin/cc - skipped
- Detecting C compile features
- Detecting C compile features - done
- Detecting CXX compiler ABI info
- Detecting CXX compiler ABI info - done
- Check for working CXX compiler: /usr/bin/c++ - skipped
- 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 - Success
- Found Threads: TRUE
- Microsoft SEAL -> Version 4.1.1 detected
- Microsoft SEAL -> Targets available: SEAL::seal
- Configuring done
- Generating done
- Build files have been written to: /home/qmj/seal/Demo

```

```

qmj@qmj-virtual-machine:~/seal/Demo$ make
[ 50%] Building CXX object CMakeFiles/he.dir/code.cpp.o
[100%] Linking CXX executable he
[100%] Built target he

```

```

qmj@qmj-virtual-machine:~/seal/Demo$ ./he
原始向量 x 是：
[ 1.000, 2.000, 3.000 ]

原始向量 y 是：
[ 2.000, 3.000, 4.000 ]

原始向量 z 是：
[ 3.000, 4.000, 5.000 ]

Line 65 --> 计算 x^2 .
Line 73 --> + Modulus chain index for x2: 1
Line 79 --> + Modulus chain index for xc: 2
Line 83 --> 计算 1.0*x .
Line 91 --> + Modulus chain index for xc new: 1
Line 98 --> 计算 1.0*x*x^2 .
Line 105 --> + Modulus chain index for x3: 0
Line 111 --> 计算 y*z .
Line 119 --> + Modulus chain index for yz: 1
Line 124 --> Normalize scales to 2^40.
Line 129 --> + Exact scale in 1*x^3: 1.09951e+12
Line 131 --> + Exact scale in y*z: 1.09951e+12
Line 138 --> + Modulus chain index for yz new: 0
Line 143 --> 计算 x^3+y*z .
Line 157 --> 结果是：

[ 7.000, 20.000, 47.000, ..., 0.000, 0.000, 0.000 ]

```

答案符合公式  $x^3 + y \cdot z$ 。

## 4 实验心得

通过这次实验，我深刻体会到了同态加密技术的强大和实用性，特别是在保护数据隐私的同时执行复杂计算的能力。使用Microsoft SEAL库进行加密计算不仅加深了我对同态加密原理的理解，也提升了我的编程技能和解决实际问题的能力。我特别感兴趣的是如何管理加密数据的层级和尺度，这对于确保计算精确性和效率至关重要。这次实验是我信息安全学习旅程中的重要一步，它激发了我进一步探索同态加密和其他加密技术在真实世界应用中的可能性的兴趣。