



南开大学
Nankai University

南 开 大 学

计 算 机 学 院

数据安全

SEAL 应用实践

穆禹宸 2012026

年级：2020 级

专业：信息安全、法学双学位班

指导教师：刘哲理

2023 年 3 月 22 日

目录

一、 实验名称	1
二、 实验要求	1
三、 实验过程	1
(一) 环境配置	1
(二) 实验代码	2
1. 前置部分	2
2. 实验核心代码	5
3. 客户端解码	9
(三) 实验结果展示	13
四、 心得体会	14

一、 实验名称

SEAL 应用实践

二、 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 $x^3 + y * z$ 的运算。

三、 实验过程

(一) 环境配置

首先，从 github 上拉取开源的 SEAL 库，代码如下所示：

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

结果如下所示：

```
myc@DESKTOP-2N69J26:~$ git clone https://github.com/microsoft/SEAL
Cloning into 'SEAL'...
remote: Enumerating objects: 17111, done.
remote: Counting objects: 100% (282/282), done.
remote: Compressing objects: 100% (152/152), done.
remote: Total 17111 (delta 141), reused 228 (delta 111), pack-reused 1
6829
Receiving objects: 100% (17111/17111), 5.00 MiB | 596.00 KiB/s, done.
Resolving deltas: 100% (12943/12943), done.
myc@DESKTOP-2N69J26:~$ cd SEAL
myc@DESKTOP-2N69J26:~/SEAL$ cmake .
```

然后输入 cmake .

过程及结果如下所示：

```
myc@DESKTOP-2N69J26:~/SEAL$ cmake .
-- Build type (CMAKE_BUILD_TYPE): Release
-- The CXX compiler identification is GNU 11.3.0
-- The C compiler identification is GNU 11.3.0
-- 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
-- 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
-- Microsoft SEAL debug mode: OFF
-- SEAL_USE_CXX17: ON
-- SEAL_BUILD_DEPS: ON
-- SEAL_USE_MSGSL: ON
-- Microsoft GSL: download ...
-- SEAL_USE_ZLIB: ON
-- ZLIB: download ...
```

```
-- Configuring done
-- Generating done
-- Build files have been written to: /home/myc/SEAL
```

然后输入 make .

过程和结果如下所示：

```
myc@DESKTOP-2N69J26:~/SEAL$ make
[ 1%] Building C object thirdparty/zlib-build/CMakeFiles/zlibstatic.d
ip/adler32.o
[ 2%] Building C object thirdparty/zlib-build/CMakeFiles/zlibstatic.d
ip/compress.o
[ 3%] Building C object thirdparty/zlib-build/CMakeFiles/zlibstatic.d
ip/crc32.o
[ 5%] Building C object thirdparty/zlib-build/CMakeFiles/zlibstatic.d
ip/deflate.o
[ 6%] Building C object thirdparty/zlib-build/CMakeFiles/zlibstatic.d
ip/gzclose.o
[ 6%] Building C object thirdparty/zlib-build/CMakeFiles/zlibstatic.d
ip/gzlib.o
[ 7%] Building C object thirdparty/zlib-build/CMakeFiles/zlibstatic.d
ip/gzread.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
```

最后输入 `sudo make install`

过程和结果如下所示:

```
myc@DESKTOP-2N69JZ6:~/SEAL$ sudo make install
Consolidate compiler generated dependencies of target zlibstatic
[ 18%] Built target zlibstatic
Consolidate compiler generated dependencies of target libzstd_static
[ 54%] Built target libzstd_static
Consolidate compiler generated dependencies of target seal
[100%] Built target seal
Install the project...
-- Install configuration: "Release"
-- Installing: /usr/local/include/SEAL-4.1/seal/util/config.h
-- Installing: /usr/local/lib/libseal-4.1.a
-- Installing: /usr/local/lib/cmake/SEAL-4.1/SEALTargets.cmake
-- Installing: /usr/local/lib/cmake/SEAL-4.1/SEALTargets-release.cmake

-- Installing: /usr/local/include/SEAL-4.1/seal/util/nuc.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
myc@DESKTOP-2N69JZ6:~/SEAL$
```

此时 SEAL 库已经顺利安装了。在本过程中我没有遇到困难。

(二) 实验代码

1. 前置部分

首先, 仿照参考代码进行初始设置, 导入 `examples.h` 头文件, 并生成 `x,y,z` 三个向量

```

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: "<<endl;
15     print_vector(x);
16     cout<<" 原始向量 y 是: "<<endl;
17     print_vector(y);
18     cout<<" 原始向量 z 是: "<<endl;
19     print_vector(z);
20     cout<<endl;
21     ...
22 }

```

然后构建参数容器 `parms`，CKKS 有三个重要参数：

1. `poly_module_degree` (多项式模数)
2. `coeff_modulus` (参数模数)
3. `scale` (规模)

这里我的设置都没有进行改动，均为官方 SEAL 库推荐的设置，如下所示：

```

1  // 构建参数容器 parms
2  EncryptionParameters parms(scheme_type::ckks);
3  // 这里的参数都使用官方建议的
4  size_t poly_modulus_degree = 8192;
5  parms.set_poly_modulus_degree(poly_modulus_degree);
6  parms.set_coeff_modulus(CoeffModulus::Create(poly_modulus_degree, { 60, 40, 40,
   ↪ 60 }));
7  double scale = pow(2.0, 40);

```

对于参数的选择，虽然是选择了官方的建议，但是也要搞清楚其原理。在官方文档的 `examples` 之中，其解释的其原理如下：

假设 CKKS 密码中的比例为 S ，并且当前 `coeff_modulus` (用于密码) 中最后一个素数为 P 。重新调整到下一个级别会将比例更改为 S/P ，并像通常在模量转换时那样从 `coeff_modulus` 中删除 P 。素数数量限制了可以执行多少次重新调整操作，因此限制了计算机深度。

当然，我们可以自由选择初始比例。一种好策略是将初期规模 S 和系数组合 P_i 设置得非常接近彼此。如果在乘法之前密码具有规模 S ，则在乘法之后它们具有规模 S^2 ，在重新调整之后具有规模 S^2/P_i 。如果所有 P_i 都接近于 S ，则 S^2/P_i 再次接近于 S 。这样我们就稳定了各个阶段上的标度值靠近 S 。通常对于深度为 D 的环路，我们需要 D 次重排，也就是说，我们需要能够从系数组合中移除 D 个素数。当系数组合只剩下一个素数时，剩余素必须大于 S 几位以保持明文小数字点前面部分不变。

因此，一般的好策略是按以下方式选择 CKKS 方案的参数：

- (1) 将 60 位质数作为 `coeff_modulus` 中的第一个质数。这样在解密时可以获得最高精度；
- (2) 选择另一个 60 位质数作为 `coeff_modulus` 的最后一个元素，因为它将用作特殊质数，并且应该与其他质数中最大的那个一样大；
- (3) 选择中间质数彼此接近。

我们使用 `CoeffModulus::Create` 生成适当大小的素数。请注意，我们的 `coeff_modulus` 总共有 200 位，在 `poly_modulus_degree` 上限之下这是因为 `CoeffModulus::MaxBitCount(8192)` 返回的是 218。

因此，这也就是官方建议的 60, 40, 40, 60 的来历。

我们选择初始比例为 2^40 。在最后一级，这使我们保留了 $60-40=20$ 位小数点前的精度，并且足够（大约 10-20 位）小数点后的精度。由于我们的中间质数是 40 位（实际上它们非常接近 2^40 ），因此我们可以像上面描述的那样实现比例稳定化。

然后构建各模块，代码如下所示：

```

1 // 用参数生成 CKKS 框架 context
2 SEALContext context(parms);
3
4 // 构建各模块
5 // 生成公钥、私钥和重线性化密钥
6 KeyGenerator keygen(context);
7 auto secret_key = keygen.secret_key();
8 PublicKey public_key;
9 keygen.create_public_key(public_key);
10 RelinKeys relin_keys;
11 keygen.create_relin_keys(relin_keys);
12 // 构建编码器，加密模块、运算器和解密模块
13 // 注意加密需要公钥 pk；解密需要私钥 sk；编码器需要 scale
14 Encryptor encryptor(context, public_key);
15 Evaluator evaluator(context);
16 Decryptor decryptor(context, secret_key);
17 CKKSEncoder encoder(context);

```

这里构建了一个 `SEALContext` 对象 `context(parms)`。这是一个重要的类，它检查我们刚刚设置的参数的有效性和属性。

然后对三个向量进行编码和加密

```

1 // 对向量  $x$ 、 $y$ 、 $z$  进行编码
2 Plaintext xp, yp, zp;
3 encoder.encode(x, scale, xp);
4 encoder.encode(y, scale, yp);
5 encoder.encode(z, scale, zp);
6
7 // 对明文  $x_p$ 、 $y_p$ 、 $z_p$  进行加密
8 Ciphertext xc, yc, zc;
9 encryptor.encrypt(xp, xc);
10 encryptor.encrypt(yp, yc);
11 encryptor.encrypt(zp, zc);

```

2. 实验核心代码

核心代码步骤

我将本次实验拆分为如下五个步骤：

- (1) 计算 $x * x$ ，即 x^2 ，将结果存入变量 $\overline{x2}$ 之中
- (2) 计算 $1.0 * x$ ，结果保存在 \overline{xc} 之中
- (3) 将新的 \overline{xc} 与变量 $\overline{x2}$ 相乘，即 $1.0 * x * x^2$ ，得到 x^3 ，将结果存入变量 $\overline{x3}$ 之中
- (4) 计算 $y * z$ ，将结果存入变量 \overline{yz} 之中
- (5) 将 $\overline{x3} + \overline{yz}$ ，即计算 $x^3 + y * z$ ，完成实验要求

下面说明具体代码：

步骤一

计算 $x * x$ ，即 x^2 ，将结果存入变量 $\overline{x2}$ 之中

首先，计算 x^2 ，较为简单，代码如下所示：

```

1 // 步骤 1, 计算  $x^2$ 
2     print_line(__LINE__);
3     cout << " 计算  $x^2$  ." << endl;
4     Ciphertext x2;
5     evaluator.multiply(xc, xc, x2);
6     // 进行 relinearize 和 rescaling 操作
7     evaluator.relinearize_inplace(x2, relin_keys);
8     evaluator.rescale_to_next_inplace(x2);
9     // 然后查看一下此时  $x^2$  结果的 level
10    print_line(__LINE__);
11    cout << " + Modulus chain index for x2: "
12    << context.get_context_data(x2.parms_id())->chain_index() << endl;

```

这里我们说明以下 level 这个概念。

在 Microsoft SEAL 中，一组加密参数（不包括随机数生成器）通过其 256 位哈希唯一标识。该哈希称为 “parms_id”，可以随时轻松访问和打印。只要更改任何一个参数，哈希就会发生变化。

当从给定的 EncryptionParameters 实例创建 SEALContext 时，Microsoft SEAL 自动创建所谓的“模量切换链”，即从原始集合派生出来的其他加密参数链。模量切换链中的参数与原始参数相同，除了系数模量大小沿着链向下减小之外。更确切地说，在链中每个参数集都试图从前一个集合删除最后一个系数模量素数；直到该参数集不再有效为止（例如 plain_modulus 大于剩余 coeff_modulus）。可以轻松遍历整个链并访问所有参数集。此外，链中每个参数集都有一个 “chain index”，指示其在链中的位置，因此最后一个设置具有索引 0。如果某组加密参数或携带这些加密参数的对象比另一组较高，则说明它们处于较高级别上，并且其 chain index 更大即早期出现在该条线上。

链接列表是由 SEALContext :: ContextData 对象构成，每个节点可通过特定加密算法 (poly_modulus_degree 保持不变但 coeff_modulus 变化) 的 parms_id 进行标识，在创建 SEALContext 时进行独特预计算并存储于其中。整个链接列表可以很容易地通过 SEALContext 随时访问。

由于模量切换链，5 个质数的顺序很重要。最后一个质数有特殊含义，并称其为“特殊质数”。因此，在模量切换链中设置的第一个参数是唯一涉及到特殊质数的参数。所有密钥对象（如 SecretKey）都在此最高级别创建。所有数据对象（如 Ciphertext）只能处于较低级别。特殊素数应该尽可能大地与其他 coeff_modulus 中最大的素数相同，尽管这不是严格要求。

“模量切换”是一种将密文参数向下更改的技术。函数 Evaluator :: mod_switch_to_next 始终切换到链中下一个级别，而 Evaluator :: mod_switch_to 则切换到与给定 parms_id 相对应的链中下一个参数集。但是，在链条上无法向上切换。

步骤二

计算 $1.0 * x$ ，结果保存在 xc 之中

然后，计算 $1.0 * x$ ，这一步的目的是因为 x^2 和 xc 此时的 level 已经不同，因此需要通过与乘法单位元 “1” 相乘，将 xc 的 level 从 2 变为 1。


```

1 // 步骤 2, 计算  $1.0 * x$ 
2 // 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
3 print_line(__LINE__);
4 cout << " + Modulus chain index for xc: "
5 << context.get_context_data(xc.parms_id())->chain_index() << endl;
6 // 因此, 需要对  $x$  进行一次乘法和 rescaling 操作
7 print_line(__LINE__);
8 cout << " 计算  $1.0 * x$  ." << endl;
9 Plaintext plain_one;
10 encoder.encode(1.0, scale, plain_one);
11 // 执行乘法和 rescaling 操作:
12 evaluator.multiply_plain_inplace(xc, plain_one);
13 evaluator.rescale_to_next_inplace(xc);
14 // 再次查看  $xc$  的层级, 可以发现  $xc$  与  $x^2$  层级变得相同
15 print_line(__LINE__);
16 cout << " + Modulus chain index for xc new: "
17 << context.get_context_data(xc.parms_id())->chain_index() << endl;
18 // 那么, 此时  $xc$  与  $x^2$  层级相同, 二者可以相乘了

```

CKKS 中的乘法会导致密文中的尺度增长。任何密文的尺度都不能太接近 `coeff_modulus` 的总大小, 否则密文就没有足够的空间来存储缩放后的明文。CKKS 方案提供了“重新调整”功能, 可以减小尺度并稳定尺度扩展。

rescale 是一种模数切换操作。与模数切换相同, 它从 `coeff_modulus` 中删除最后一个质数, 并将密文按所删除质数进行缩放。通常情况下, 我们希望完全控制如何改变比例, 在 CKKS 方案中更常见地使用精心选择的系数模量素数。因此, 每一次乘法都要做一次 *rescale*

步骤三

将新的 xc 与变量 x^2 相乘, 即 $1.0 * x * x^2$, 得到 x^3 , 将结果存入变量 x^3 之中

然后, 计算 $1.0 * x * x^2$, 也就是把前两个步骤得到的结果相乘

```

1 // 步骤 3, 计算  $x^3$ , 即  $1 * x * x^2$ 
2 // 先设置新的变量叫  $x^3$ 
3 print_line(__LINE__);
4 cout << " 计算  $1.0 * x * x^2$  ." << endl;
5 Ciphertext x3;
6 evaluator.multiply_inplace(x2, xc);
7 evaluator.relinearize_inplace(x2, relin_keys);
8 evaluator.rescale_to_next(x2, x3);
9 // 此时观察  $x^3$  的层级
10 print_line(__LINE__);
11 cout << " + Modulus chain index for x3: "
12 << context.get_context_data(x3.parms_id())->chain_index() << endl;

```

步骤四

计算 $y * z$ ，将结果存入变量 \underline{yz} 之中

然后，计算 $y * z$ ，这一步是直接相乘，较为简单，代码如下所示：

```

1 // 步骤 4, 计算  $y*z$ 
2 print_line(__LINE__);
3 cout << " 计算  $y*z$  ." << endl;
4 Ciphertext yz;
5 evaluator.multiply(yz, zc, yz);
6 // 进行 relinearize 和 rescaling 操作
7 evaluator.relinearize_inplace(yz, relin_keys);
8 evaluator.rescale_to_next_inplace(yz);
9 // 然后查看一下此时  $y*z$  结果的 level
10 print_line(__LINE__);
11 cout << " + Modulus chain index for yz: "
12 << context.get_context_data(yz.parms_id())->chain_index() << endl;

```

步骤五

将 $\underline{x3} + \underline{yz}$ ，即计算 $x^3 + y * z$ ，完成实验要求

此时，我们分别得到了变量 $\underline{x3}$ (x^3) 和变量 \underline{yz} ($y * z$)，它们的 *level* 和 *scale* 都是不同的，此时他们无法直接相加。为了解决这两个问题，需要进行如下操作：

```

1 // 注意，此时问题在于 scales 的不统一，可以直接重制。
2 print_line(__LINE__);
3 cout << "Normalize scales to  $2^{40}$ ." << endl;
4 x3.scale() = pow(2.0, 40);
5 yz.scale() = pow(2.0, 40);
6 // 输出观察，此时的 scale 的大小已经统一了！
7 print_line(__LINE__);
8 cout << " + Exact scale in  $1*x^3$ : " << x3.scale() << endl;
9 print_line(__LINE__);
10 cout << " + Exact scale in  $y*z$ : " << yz.scale() << endl;
11
12 // 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！
13 // 在官方 examples 中，给出了一个简便的变换层级的方法，如下所示：
14 parms_id_type last_parms_id = x3.parms_id();
15 evaluator.mod_switch_to_inplace(yz, last_parms_id);
16 print_line(__LINE__);
17 cout << " + Modulus chain index for yz new: "
18 << context.get_context_data(yz.parms_id())->chain_index() << endl;

```

这里的原理同样来自于官方样例：

由于 P_2 和 P_1 非常接近于 2^{40} ，我们可以简单地“欺骗”Microsoft SEAL 并将比例尺设置

为相同。将 $1.0 * x * x^2$ 的比例尺更改为 2^{40} 仅意味着我们通过 $2^{120} / (P_2^2 * P_1)$ 缩放 $1.0 * x * x^2$ 的值, 这非常接近于 1。这不应导致任何明显的错误。注意, 这只是其中一种方法, 但是这种方法最简单。

而对于 level 的切换, 更加简单, CKKS 支持模数切换, 就像 BFV 方案一样, 使我们能够在不需要时切换系数模数的部分。

最后, 计算 $x^3 + y * z$

```
1 // 步骤 5,  $x^3+y*z$ 
2     print_line(__LINE__);
3     cout << " 计算  $x^3+y*z$  ." << endl;
4     Ciphertext encrypted_result;
5     evaluator.add(x3, yz, encrypted_result);
```

注意, 加法并不需要 rescale 操作。因此直接相加后直接处理即可!

3. 客户端解码

将计算完成的结果从服务器重新返回客户端, 客户端进行解码, 并进行结果展示, 代码如下所示:

```
1 // 计算完毕, 服务器把结果发回客户端
2 Plaintext result_p;
3 decryptor.decrypt(encrypted_result, result_p);
4
5 // 注意要解码到一个向量上
6 vector<double> result;
7 encoder.decode(result_p, result);
8
9 // 输出结果
10 print_line(__LINE__);
11 cout << " 结果是: " << endl;
12 print_vector(result, 3 /*precision*/);
13
14 return 0;
```

至此, 本次实验所编写的代码和整个流程全部结束。

实验完整代码如下所示:

实验完整代码

```
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: "<<endl;
15 print_vector(x);
16 cout<<"原始向量y是: "<<endl;
17 print_vector(y);
18 cout<<"原始向量z是: "<<endl;
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,
27     40, 40, 60 }));
28 double scale = pow(2.0, 40);
29
30 // 用参数生成 CKKS 框架 context
31 SEALContext context(parms);
32
33 // 构建各模块
34 // 生成公钥、私钥和重线性化密钥
35 KeyGenerator keygen(context);
36 auto secret_key = keygen.secret_key();
37 PublicKey public_key;
38 keygen.create_public_key(public_key);
39 RelinKeys relin_keys;
40 keygen.create_relin_keys(relin_keys);
41 // 构建编码器，加密模块、运算器和解密模块
42 // 注意加密需要公钥 pk；解密需要私钥 sk；编码器需要 scale
43 Encryptor encryptor(context, public_key);
44 Evaluator evaluator(context);
45 Decryptor decryptor(context, secret_key);
46 CKKSEncoder encoder(context);
47
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
54 // 对明文 xp、yp、zp 进行加密
55 Ciphertext xc, yc, zc;
56 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, zc, yz);
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 }
```

(三) 实验结果展示

编写 CMakeLists.txt, 内容如下所示:

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

这段 CMakeListx.txt 的内容是: CMake 最低版本要求是 3.10, 项目名称是 demo, 源文件是 homework.cpp。编译选项是使用 C++17 标准。同时, 使用 find_package 命令查找 SEAL 库, 并将其链接到可执行文件 he 中。

然后, 运行如下命令, 如下所示:

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

最终输出的结果如下所示:

```

myc@DESKTOP-2N69J26:~/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 ]

myc@DESKTOP-2N69J26:~/demo$ 

```

可以看到，我们的结果其中 $7 = 1^3 + 1 * 3$, $20 = 2^3 + 3 * 4$, $47 = 3^3 + 4 * 5$ ，完全满足 $result = x^3 + y * z$ ，因此可以证明，本次实验取得圆满成功！

四、 心得体会

本次实验，我学习了 SEAL 库的使用。并且在掌握课上知识的基础上，通过自己手动编写 SEAL 的程序，实现了 $x^3 + y * z$ 的运算。这其中的过程还是比较艰辛的，困难主要有两方面，一方面是如何妥善地设计乘法的顺序，因为数据的 level 只能降低无法升高，所以要小心设计计算的先后顺序，我的设计是先进行 $1.0 * x$ ，然后进行 $(1.0 * x) * x^2$ ，这样得到了 x^3 ，同时也解决了 level 的问题，最后再加 $y * z$ 得到最终结果；另一方面则是在加法的过程中，我们的 scale 实际上已经变了，因此如果直接相加，会提示 mismatch，即无法直接相加，解决这个问题我花费了一定的时间，最终在 SEAL 库所给出的官方 example 中，找到了可行的解决办法，即由于我们的 P_1 和 P_2 非常接近，可以直接设置他们的 scale (原文为：set the scales to be the same)，然后才能直接相加，这样我们才能得到最终的结果。

在仔细研究了 SEAL 库之中所给的 example 的代码和注释，我对 SEAL 库有了更深刻的理解，同时对课上理论的理解也更加透彻了，尤其是在参数选择那里的解释使我醍醐灌顶！

本次实验我在编写工程和参考 SEAL 库的官方样例的时候，也进一步熟悉了 CMAKE 这个强大的工具。借此机会，我感觉我在信息安全数学基础、密码学课上学到的理论知识有了进一步的加深，而且我在操作系统、编译原理课程上所学到的对 Linux 和编译的操作有了更加深刻的理解。本次实验不光锻炼了我的代码能力，也使我之前所学知识有了整体的贯通和融合！