



南开大学  
Nankai University

南 开 大 学

网 络 空 间 安 全  
学 院

数据安全实验报告

---

SEAL 应用实践

---

2111408 周钰宸

年级：2021 级

专业：信息安全

指导教师：刘哲理

2024 年 3 月 24 日

## 摘要

SEAL(Simple Encrypted Arithmetic Library) 是微软开源的基于 C++ 的同态加密库, 支持 CKKS 方案等多种全同态加密方案, 支持基于整数的精确同态运算和基于浮点数的近似同态运算。该项目采用商业友好的 MIT 许可证在 GitHub 上开源。SEAL 基于 C++ 实现, 不需要其他依赖库。本次实验参照了课本实验 2.3 中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的思路, 实现了将三个数的密文发送到服务器类完成  $x^3 + y \cdot z$  的运算。

**关键字:** 数据安全 SEAL CKKS 全同态加密方案

## 目录

<b>一、 实验要求</b>	<b>1</b>
<b>二、 实验原理</b>	<b>1</b>
(一) CKKS 算法 . . . . .	1
(二) SEAL 开发框架 . . . . .	2
(三) 标准化构建流程 . . . . .	2
<b>三、 实验准备</b>	<b>3</b>
(一) 实验环境 . . . . .	3
(二) SEAL 库安装 . . . . .	3
1. git clone 加密库资源 . . . . .	3
2. 编译与安装 . . . . .	4
(三) 简单测试程序 . . . . .	5
<b>四、 实验过程</b>	<b>7</b>
<b>五、 实验总结与思考</b>	<b>12</b>

## 一、实验要求

1. **实验 2.3:** 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。

本次实验参考上述实验要求，**实现了将三个数的密文发送到服务器，完成了  $x^3 + y^*z$  的运算。**

## 二、实验原理

### (一) CKKS 算法

#### 1. 算法概述:

- CKKS (Cheon-Kim-Kim-Song) 算法是一种基于多项式环的加密方案，用于解决针对实数的同态加密需求。它是针对 BFV 算法的一个扩展，可以支持更加复杂的计算，例如在实数域上进行同态加法 and 同态乘法操作。

#### 2. 核心思想:

- CKKS 算法通过在多项式环中进行同态加密来处理实数数据。它使用了一种称为“脆弱但实用”的同态加密技术，允许在密文域中执行加法和乘法操作，同时保护数据的隐私性。
- 值得注意的是，CKKS 算法基于 LWE 容错学习中的 RLWE，已经证明了 LWE 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。

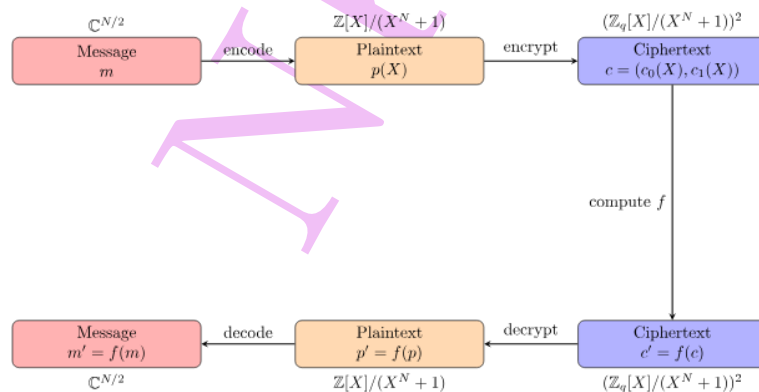


图 1: CKKS 的 HighLevelView

#### 3. 关键特点:

- **支持实数域:** 允许对实数进行同态加密，这使得它在处理实际数据时具有广泛的适用性。
- **同态加法和乘法:** 算法支持在密文域中执行同态加法和同态乘法操作，这为隐私保护数据的计算提供了更大的灵活性。但每次乘法操作后，CKKS 都需要进行再线性化 (Relinearization) 和再缩放 (Rescaling) 操作。
- **扩展性:** CKKS 算法在 BFV 算法的基础上进行了改进，具有更好的性能和扩展性，适用于更广泛的应用场景。

#### 4. 应用领域:

- 机器学习和数据分析: 可以用于在加密状态下进行机器学习模型的训练和推断, 从而保护用户的隐私数据。
- 金融和医疗保健: 用于在保护数据隐私的同时进行金融数据分析和医疗数据共享。
- 云计算和边缘计算: 可以用于在云端和边缘设备之间安全地传输和处理实数数据。

## (二) SEAL 开发框架

#### 1. 框架概述:

- SEAL (Simple Encrypted Arithmetic Library) 是一种用于同态加密计算的开发框架, 由微软研究院开发。它提供了一套简单而高效的工具, 用于在加密状态下执行各种算术操作。

#### 2. 核心思想:

- SEAL 框架的核心思想是将加密数据表示为多项式, 并使用多项式运算来执行同态加密计算。它使用了一种称为 BFV (Brakerski-Pan-Vercauteren) 的方案, 可以支持同态加法和同态乘法操作。

#### 3. 关键特点:

- 灵活性: 框架提供了灵活的 API, 支持各种加密参数的设置和定制, 使得用户可以根据实际需求进行配置和优化。
- 高效性: SEAL 框架采用了高效的多项式运算和优化技术, 能够在密文域中实现高效的同态加密计算。
- 安全性: 提供了一系列安全性保障措施, 包括数据保护、密钥管理和安全协议, 确保加密数据的安全性和隐私性。

#### 4. 应用领域:

- 数据隐私保护: 用于在保护数据隐私的同时进行数据分析和计算, 例如在医疗保健和金融领域中。
- 机器学习和人工智能: 可以支持在加密状态下进行机器学习模型的训练和推断, 从而保护用户的隐私数据。
- 云计算和边缘计算: 在云端和边缘设备之间安全地传输和处理加密数据, 保护数据的隐私和安全。

## (三) 标准化构建流程

本次实验将会基于 CKKS 方案构建一个基于云服务器的算力协助完成客户端的某种运算。

具体构建流程如下: 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  $n$

### 三、 实验准备

#### (一) 实验环境

首先本次实验由于需要用到 Linux 操作系统，这里说明一下我所使用的虚拟机和对应的操作环境：

虚拟机软件	VMware Workstation 17 Pro
宿主机	Windows 11 家庭中文版
虚拟机操作系统	Ubuntu

表 1: 本次实验环境及工具

#### (二) SEAL 库安装

##### 1. git clone 加密库资源

这里依照课本的 2.4.1 完成对 SEAL 库的安装部署。首先通过如下命令：

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

从开源的 GitHub 仓库将 SEAL 库的所需资源都 clone 下来，结果如下：

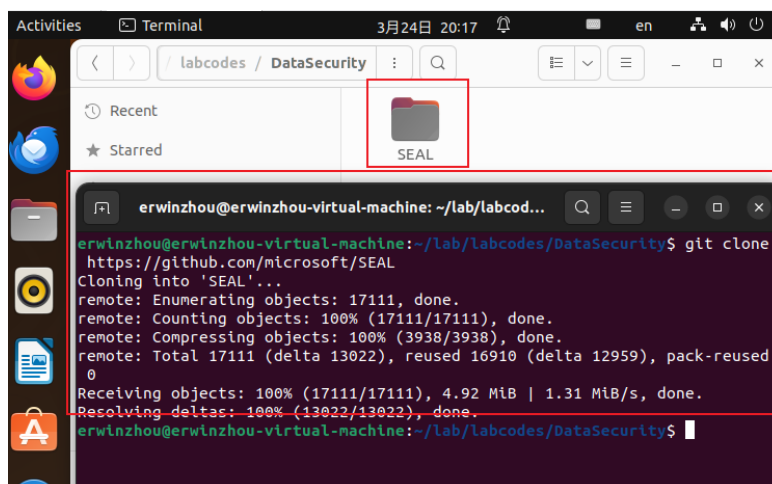


图 2: Git Clone 加密库资源

图2可以看到已经成功从 GitHub 对应仓库中复制了资源，名为 SEAL 的文件夹自动建立了。

## 2. 编译与安装

接下来会通过高级编译配置工具 cmake，来快速实现开源项目的编译和安装。

首先第一步是通过命令：

```
1 erwinzhou@erwinzhou-virtual-machine:~/lab/labcodes/DataSecurity/SEAL$ cmake
```

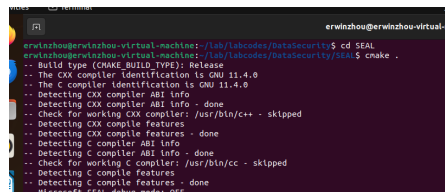


图 3: cmake 结果 1

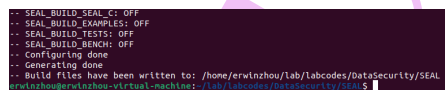


图 4: cmake 结果 2

图3和4显示了 cmake 的结果。然后通过以下命令：

```
1 erwinzhou@erwinzhou-virtual-machine:~/lab/labcodes/DataSecurity/SEAL$ make
2 erwinzhou@erwinzhou-virtual-machine:~/lab/labcodes/DataSecurity/SEAL$ sudo make install
```

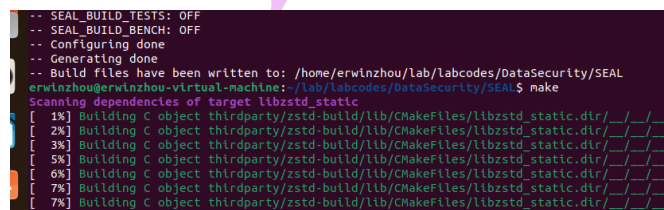


图 5: cmake 结果 3

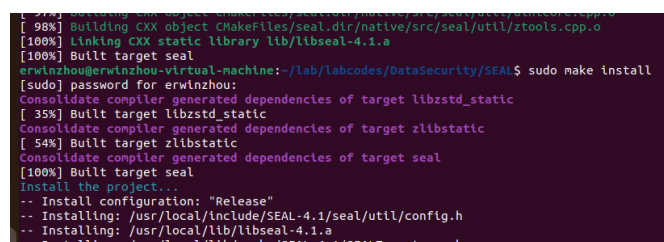


图 6: cmake 结果 4

图5和6显示了**安装完毕成功的结果**。其中在 make install 步骤，相关的头文件和静态库都被安装复制到了/usr/local 文件夹下的 include 和 lib 文件夹中，以便系统里其它应用程序可以查找到并使用。

### (三) 简单测试程序

之后为了验证 SEAL 的 C++ 库已经正常安装，这里进行一个简单的测试。首先在 SEAL 同级下建立 demo 文件夹，然后在其中的一个 test.cpp 中写入如下代码：

```
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 }
```

以及对应的 MakeList:

```
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)
```

这里比较重要的代码是**通过调用 EncryptionParameters 即 SEAL 头文件中的类来验证功能**。由于此句在 hello world 之前，因此如果 SEAL 库成功安装，就会正常输出 helloworld，而不报任何错误。

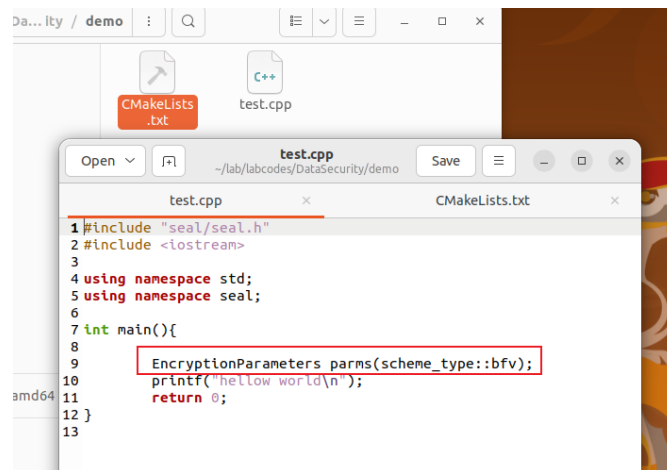


图 7: SEAL 库安装验证

之后再依次通过 `cmake`, `make` 和 `.` 命令完成编译和对可执行文件的运行。结果如下:

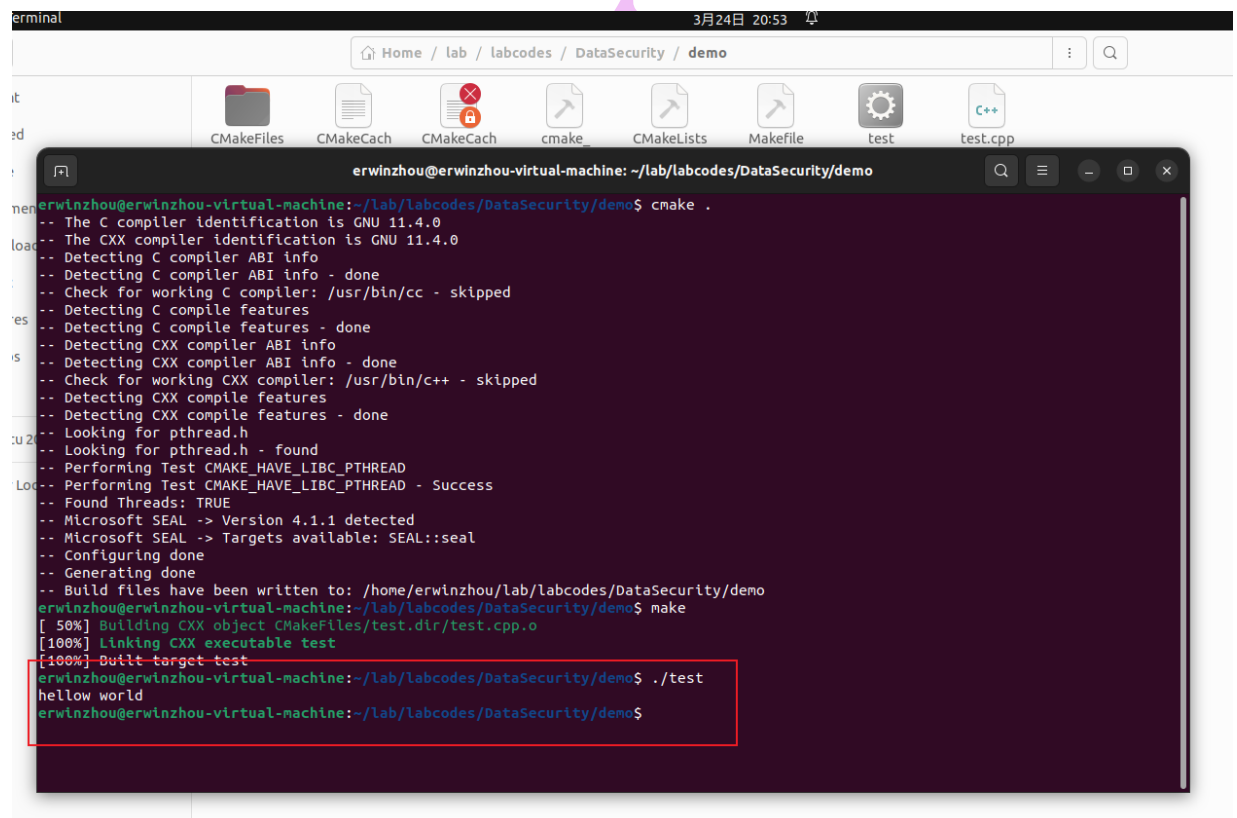


图 8: SEAL 库验证结果

图8可以看到编译完成后运行输出了”hello world”，代表了实验简单测试验证成功！接下来就可以进入正式的应用中了。



## 四、 实验过程

本次实验我们需要以课本中**实验 2.3** 中实现  $x^*y^*z$  的代码作为参考。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思路，实现了将三个数的密文发送到服务器类完成  $x^3 + y^*z$  的运算。实验代码如下：

```
1  #include "examples.h"
2  #include <vector>
3  using namespace std;
4  using namespace seal;
5  #define N 3
6  int main()
7  {
8      // Client's perspective: data to be computed
9      vector<double> x, y, z;
10     x = {1.0, 2.0, 3.0};
11     y = {2.0, 3.0, 4.0};
12     z = {3.0, 4.0, 5.0};
13     cout << "Original vector x is: ";
14     print_vector(x);
15     cout << "Original vector y is: ";
16     print_vector(y);
17     cout << "Original vector z is: ";
18     print_vector(z);
19     cout << endl;
20     // Build parameter container parms
21     EncryptionParameters parms(scheme_type::ckks);
22     // Use recommended parameters
23     size_t poly_modulus_degree = 8192;
24     parms.set_poly_modulus_degree(poly_modulus_degree);
25     parms.set_coeff_modulus(CoeffModulus::Create(poly_modulus_degree, {60, 40,
26         40, 60}));
27     double scale = pow(2.0, 40);
28     // Generate CKKS framework context with the parameters
29     SEALContext context(parms);
30     // Build modules
31     // Generate public key, secret key, and relinearization keys
32     // Which will be used frequently later
33     KeyGenerator keygen(context);
34     auto secret_key = keygen.secret_key();
35     PublicKey public_key;
36     keygen.create_public_key(public_key);
37     RelinKeys relin_keys;
38     keygen.create_relin_keys(relin_keys);
39     // Build encoder, encryptor, evaluator, and decryptor modules
```

```

39
40 // Note tha:
41 // (1)encryption requires public key pk
42 // (2)decryption requires secret key sk
43 // (3)encoder requires scale
44 Encryptor encryptor(context, public_key);
45 Evaluator evaluator(context);
46 Decryptor decryptor(context, secret_key);
47 CKKSEncoder encoder(context);
48 // Encode vectors x, y, z
49 Plaintext xp, yp, zp;
50 encoder.encode(x, scale, xp);
51 encoder.encode(y, scale, yp);
52 encoder.encode(z, scale, zp);
53 // Encrypt plaintexts xp, yp, zp
54 Ciphertext xc, yc, zc;
55 encryptor.encrypt(xp, xc);
56 encryptor.encrypt(yp, yc);
57 encryptor.encrypt(zp, zc);
58 /**
59  * For this part, the server will perform the following operations:
60  *   x^3+y*z
61  * 1. Calculate x^3.
62  * 2. Calculate y*z.
63  * 3. Normalize the scales of x^3 and y*z.
64  * 4. Transform the levels of x^3 and y*z.
65  * 5. Calculate x^3+y*z.
66  * 6. Send the result back to the client.
67  */
68 // To calculate x^2
69 print_line(__LINE__);
70 cout << "-----Calculating x^2-----" << endl;
71 Ciphertext x2;
72 evaluator.multiply(xc, xc, x2);
73 // Perform relinearization and rescaling operations
74 evaluator.relinearize_inplace(x2, relin_keys);
75 evaluator.rescale_to_next_inplace(x2);
76 // Check the level of x^2 at this point
77 print_line(__LINE__);
78 cout << " + Modulus chain index for x2: "
79       << context.get_context_data(x2.parms_id())->chain_index() << endl;
80 // To calculate 1.0*x
81 // At this point, the level of xc should be 2, higher than x^2, so we need
    to solve the level problem

```

```

82     print_line(__LINE__);
83     cout << " + Modulus chain index for xc: "
84           << context.get_context_data(xc.parms_id())->chain_index() << endl;
85     // Therefore, we need to perform one multiplication and rescaling
      operation on x
86     print_line(__LINE__);
87     cout << "-----Calculating 1.0*x-----" << endl;
88     Plaintext plain_one;
89     encoder.encode(1.0, scale, plain_one);
90     // Perform multiplication and rescaling operations:
91     evaluator.multiply_plain_inplace(xc, plain_one);
92     evaluator.rescale_to_next_inplace(xc);
93     // Check the level of xc again, and we can see that xc and x^2 have the
      same level now
94     print_line(__LINE__);
95     cout << " + Modulus chain index for xc new: "
96           << context.get_context_data(xc.parms_id())->chain_index() << endl;
97     // Now, xc and x^2 have the same level, so they can be multiplied
98     // To calculate x^3, which is 1*x*x^2
99     // Restored in x3
100    print_line(__LINE__);
101    cout << "-----Calculating 1.0*x*x^2-----" << endl;
102    Ciphertext x3;
103    evaluator.multiply_inplace(x2, xc);
104    evaluator.relinearize_inplace(x2, relin_keys);
105    evaluator.rescale_to_next(x2, x3);
106    // Check the level of x^3 at this point
107    print_line(__LINE__);
108    cout << " + Modulus chain index for x3: "
109          << context.get_context_data(x3.parms_id())->chain_index() << endl;
110    // To calculate y*z
111    print_line(__LINE__);
112    cout << "-----Calculating y*z-----" << endl;
113    Ciphertext yz;
114    evaluator.multiply(yz, zc, yz);
115    // Perform relinearization and rescaling operations
116    evaluator.relinearize_inplace(yz, relin_keys);
117    evaluator.rescale_to_next_inplace(yz);
118    // Check the level of y*z at this point
119    print_line(__LINE__);
120    cout << " + Modulus chain index for yz: "
121          << context.get_context_data(yz.parms_id())->chain_index() << endl;
122    // Note that the problem now is the inconsistency of scales, which can be
      directly reset

```

```

123     print_line(__LINE__);
124     cout << "-----Normalize Scales-----" << endl;
125     // Normalize scales to 2^40
126     x3.scale() = pow(2.0, 40);
127     yz.scale() = pow(2.0, 40);
128     // The scale sizes is unified now
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     // The levels of x^3 and y*z are not consistent, so we need to solve the
        level problem
134     parms_id_type last_parms_id = x3.parms_id();
135     evaluator.mod_switch_to_inplace(yz, last_parms_id);
136     print_line(__LINE__);
137     cout << " + Modulus chain index for yz new: "
138         << context.get_context_data(yz.parms_id())->chain_index() << endl;
139     // Step 5, x^3+y*z
140     print_line(__LINE__);
141     cout << "-----Calculating x^3+y*z-----" << endl;
142     Ciphertext encrypted_result;
143     evaluator.add(x3, yz, encrypted_result);
144     // Calculation completed, the server sends the result back to the client
145     Plaintext result_p;
146     decryptor.decrypt(encrypted_result, result_p);
147     // Decode to a vector
148     vector<double> result;
149     encoder.decode(result_p, result);
150     // Output the result
151     print_line(__LINE__);
152     cout << "The result is:" << endl;
153     print_vector(result, 3 /*precision*/);
154     return 0;
155 }

```

以下是对代码中一些核心部分的重点解释：

#### 1. 重点参数：

- **poly\_modulus\_degree (polynomial modulus)：**该参数必须是 2 的幂，更大的 poly\_modulus\_degree 会增加密文的尺寸，这会让计算变慢，但也能执行更复杂的计算。
- **[ciphertext] coefficient modulus：**这些参数至关重要，因为 rescaling 操作依赖于 coeff\_modules 的数量。简单来说，coeff\_modules 的个数决定了可以进行 rescaling 的次数，从而决定了可以执行的乘法操作次数。

- 60, 40, 40, 60 具有重要意义: coeff\_modules 总位长 200 (60+40+40+60) 位; 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: 总位长不能超过上表限制; 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; 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。

- **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 =  $2^{30}$

## 2. 核心代码:

- **步骤 1: 计算  $x^2$** : 首先对  $x$  进行同态乘法操作, 得到  $x^2$ 。通过 evaluator.multiply(xc, xc, x2); 实现此操作。然后执行重线性化和尺度调整操作, 确保结果的正确性。
- **步骤 2: 计算  $1.0 \times x$** : 由于  $x$  的层级高于  $x^2$ , 因此对  $x$  进行乘法和尺度调整操作, 使其与  $x^2$  的层级相匹配。此处使用了 evaluator.multiply\_plain\_inplace(xc, plain\_one); 和 evaluator.rescale\_to\_next\_inplace(xc)。
- **步骤 3: 计算  $x^3$** : 通过将  $x^2$  与  $x$  进行同态乘法操作, 得到  $x^3$ 。并再次执行重线性化和尺度调整操作, 以确保结果的正确性。使用了 evaluator.multiply\_inplace(x2, xc) 和 evaluator.rescale\_to\_next(x2, x3)。
- **步骤 4: 计算  $y \times z$** : 对  $y$  和  $z$  进行同态乘法操作, 得到结果。然后执行重线性化和尺度调整操作, 确保结果的正确性。使用了 evaluator.multiply(yz, zc, yz);。
- **步骤 5: 计算  $x^3 + y \times z$** : 将  $x^3$  和  $y \times z$  进行同态加法操作, 得到最终结果的密文。此处使用了 evaluator.add(x3, yz, encrypted\_result);。

这里需要额外的重点就是尺度和层级匹配: 为了确保加法操作的正确性, 首先需要将两个操作数的尺度统一设置为 pow(2.0, 40)。然后, 为了确保两个密文位于相同的层级, 对 yz 执行模数切换操作, 使其层级与相应的密文匹配。

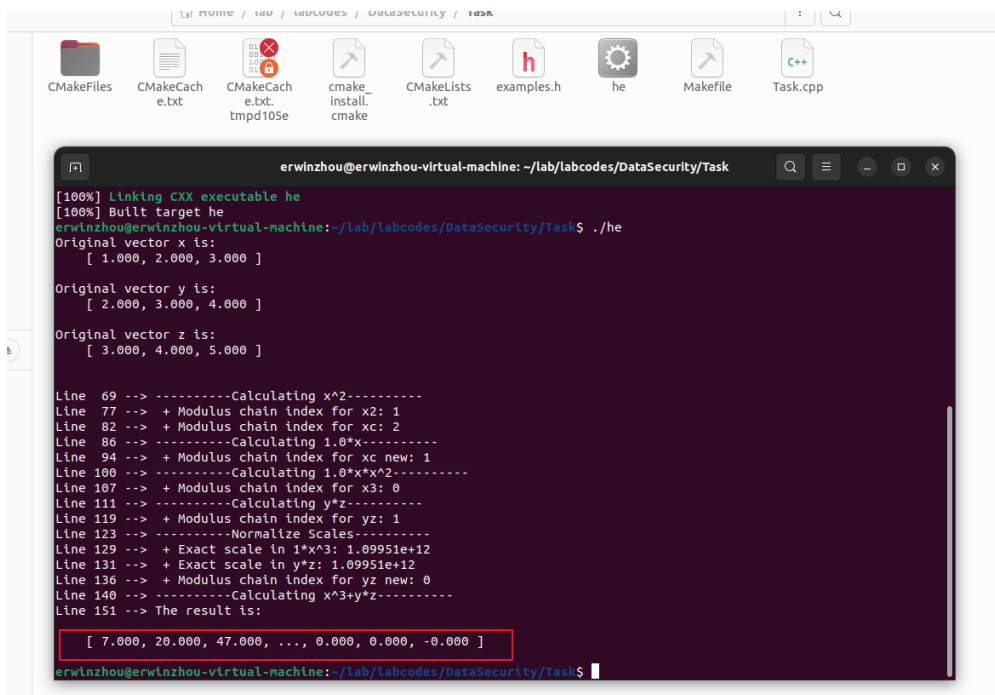
- **步骤 6: 解密和解码**: 将结果密文解密并解码成明文, 然后输出结果。

最后将 SEAL 下的 Native 下的 examples 下的 example.h 放入, 其中包含了很多 SEAL 的工具函数, 用于输出和打印等。

代码文件命名为 Task.cpp, 并重新编写 CMakeList.txt 内容如下:

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

之后再次通过 `cmake`, `make` 并运行。结果如下:



```
erwinzhou@erwinzhou-virtual-machine: ~/lab/labcodes/DataSecurity/Task
[100%] Linking CXX executable he
[100%] Built target he
erwinzhou@erwinzhou-virtual-machine:~/lab/labcodes/DataSecurity/Task$ ./he
Original vector x is:
[ 1.000, 2.000, 3.000 ]

Original vector y is:
[ 2.000, 3.000, 4.000 ]

Original vector z is:
[ 3.000, 4.000, 5.000 ]

Line 69 --> -----Calculating x^2-----
Line 77 --> + Modulus chain index for x2: 1
Line 82 --> + Modulus chain index for xc: 2
Line 86 --> -----Calculating 1.0*x-----
Line 94 --> + Modulus chain index for xc new: 1
Line 100 --> -----Calculating 1.0*x*x^2-----
Line 107 --> + Modulus chain index for x3: 0
Line 111 --> -----Calculating y*z-----
Line 119 --> + Modulus chain index for yz: 1
Line 123 --> -----Normalize Scales-----
Line 129 --> + Exact scale in 1*x^3: 1.09951e+12
Line 131 --> + Exact scale in y*z: 1.09951e+12
Line 136 --> + Modulus chain index for yz new: 0
Line 140 --> -----Calculating x^3+y*z-----
Line 151 --> The result is:
[ 7.000, 20.000, 47.000, ..., 0.000, 0.000, -0.000 ]
erwinzhou@erwinzhou-virtual-machine:~/lab/labcodes/DataSecurity/Task$
```

图 9: 实验结果

图9中可以看到通过编译生成的文件运行后的结果。如图所示, 每一部分在不同的位置正在进行的操作都是可以看到的, 能够清楚地看到中间整理统一 level, 计算对应值的过程。最后的结果也完美符合  $x^3 + y \cdot z$  的公式。

于是我们就实现了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。

**到此实验全部结束, 非常成功!**

## 五、 实验总结与思考

本次实验我通过将课堂上老师讲授的全同态的加密知识在课后进一步进行了复习, 其中格的相关知识十分有趣令我印象深刻, 如何在保证数据隐私的基础上完成复杂的运算。跟着参考资料和老师讲解视频的思路, 对参考代码进行了研读。

由此我基于教材中的例子, 完成了本次的实验内容。工作包括:

1. 熟悉了 Linux 环境中使用强大的 CMakeLists 来进行编译配置的便捷。
2. 熟悉了 Microsoft SEAL 库进行加密计算的方式。
3. 基于参考代码, 自己编写完成了对应的实验练习, 解决了遇到的问题和困难。让我对同态加密的原理理解更加深刻。

总的来说, 本次实验我收获颇丰, 希望在后面的实验中继续努力, 将数据安全领域的知识熟记于心, 并在实验中反复巩固, 不断探索。

## 参考文献

NIKU