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

实验 3：SEAL 应用实践

姓名：付政烨

学号：2113203

年级：2021 级

专业：信息安全、法学

2024 年 4 月 1 日

目录

一、 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。

二、 实验内容

(一) 环境配置

(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：

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

(2) 编译和安装

输入命令：

```
cd SEAL
cmake .
```

```
-- Configuring done
-- Generating done
-- Build files have been written to: /home/f/Desktop/Lab3/SEAL
f@f-virtual-machine:~/Desktop/Lab3/SEAL$ make
```

```
make
```

```
[100%] Linking CXX static library lib/libseal-4.1.a
[100%] Built target seal
f@f-virtual-machine:~/Desktop/Lab3/SEAL$ sudo make install
```

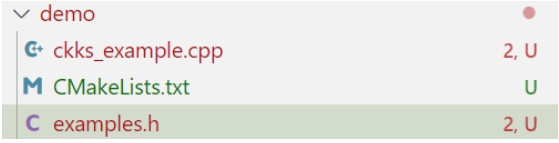
```
sudo make install
```

```
-- Installing: /usr/local/include/SEAL-4.1/seal/util/uintcore.h
-- Installing: /usr/local/include/SEAL-4.1/seal/util/ztools.h
f@f-virtual-machine:~/Desktop/Lab3/SEAL$
```

(二) 应用示例（3.5.2）复现

1. 导入 examples.h 头文件

该文件 examples.h 可在 SEAL/native/examples 目录中找到，并将其复制到 demo 目录下。



2. 编写 CMakeLists.txt 文件

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

```
cmake_minimum_required(VERSION 3.10)
project(demo)
add_executable(he ckks_example.cpp)
add_compile_options(-std=c++17)

find_package(SEAL)
 target_link_libraries (he SEAL::seal)
```

3. 编译和执行 ckks_example 文件

编写完毕后，打开控制台，依次运行：

```
cmake .
make
./he
```

运行结果如下：

```
f@f-virtual-machine:~/Desktop/Lab3/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 ]
```

(三) 实验要求部分

1. 相关参数选择

在 CKKS 密码方案中，选择合适的参数对于保证计算的精度和效率至关重要。其中，多项式模数（poly_modulus_degree）、参数模数（coeff_modulus）和规模（scale）是最关键的三个参数。

- **多项式模数（poly_modulus_degree）**：这个参数决定了多项式的次数，也就是密文中可以表示的值的范围。通常选择一个足够大的值以支持所需的精度和操作。
- **参数模数（coeff_modulus）**：参数模数是一组素数，用于构建多项式的系数。在 CKKS 方案中，它的选择直接影响了计算的效率和精度。通常建议选择一个包含几个素数的序列，这些素数的位数逐渐递减，以便在计算中充分利用同态加密的性质，同时保持足够的精度。
- **规模（scale）**：规模参数用于控制密文中数据的大小范围。它决定了明文与密文之间的比例关系。选择合适的初始比例是至关重要的，因为它直接影响了计算过程中数据的精度和稳定性。

官方建议的参数选择策略基于对 CKKS 方案中参数的深入理解和实验经验：将 60 位素数作为参数模数中的第一个素数，以保证解密时获得最高精度。选择另一个 60 位素数作为参数模数的最后一个素数，用作特殊素数，并且应该与其他素数中的最大素数一样大。选择中间的素数尽可能接近彼此。官方建议使用 CoeffModulus::Create 函数生成适当大小的素数序列。通常，参数模数总共应有大约 200 位，但不超过多项式模数的上限。因此，官方建议的参数选择为 60, 40, 40, 60 的来源在于上述策略（本次实验也采用此策略）。

```
EncryptionParameters parms(scheme_type::ckks);
size_t poly_modulus_degree = 8192;
parms.set_poly_modulus_degree(poly_modulus_degree);
parms.set_coeff_modulus(CoeffModulus::Create(poly_modulus_degree, {60, 40, 40, 60}));
double scale = pow(2.0, 40);
```

根据官方建议设置参数的示例代码，其中多项式模数为 8192，参数模数选择了 $60, 40, 40, 60$ 的素数序列，初始比例为 2^{40}。

2. $x^3 + y \times z$ 计算

该实验的过程可以被描述为以下几个步骤：

1. 计算变量 x 的平方，即 x^2，并将该值存储在变量 x_2 中。
2. 计算常数 1.0 与变量 x 的乘积，即 $1.0 \times x$，并将结果保留在变量 xc 中。
3. 将变量 xc 与 x^2 进行乘法运算，得到 x^3，并将该结果存储在变量 x_3 中。
4. 计算变量 y 与 z 的乘积，并将结果存储在变量 yz 中。
5. 将 x^3 与 yz 相加，即计算 $x^3 + y \times z$，以满足实验的要求。

(1) 计算 x^2 并将该值存储在变量 x_2 中

首先，使用 CKKSEncoder 对向量 x 进行编码转换为 Plaintext 对象，然后使用 Encryptor 对象将其加密为 Ciphertext 对象 xc。此步骤确保了数据的隐私性。使用 Evaluator 对象对 xc（即加密后的 x）执行自乘操作，得到加密的 x^2 结果存储于新的 Ciphertext 对象 $x2$ 中。此操作利用了同态加密的特性，即在密文状态下进行的乘法操作。

```
// 计算 x^2
Ciphertext x2;
evaluator.multiply(xc, xc, x2);
evaluator.relinearize_inplace(x2, relin_keys);
evaluator.rescale_to_next_inplace(x2);
cout << "Calculated x^2." << endl;
```

(2) 计算 $1.0 \times x$ 并将结果保留在变量 xc 中

将常数 1.0 编码成 Plaintext 对象。将编码后的常数 1.0 与 xc 进行同态乘法操作，更新 xc 表示 $1.0 \times x$ 的结果。这一步骤主要是为了保证后续计算的层级（密文中的多项式度数）一致。

```
// 调整层级
Plaintext plain_one;
encoder.encode(1.0, scale, plain_one);
evaluator.multiply_plain_inplace(xc, plain_one);
evaluator.rescale_to_next_inplace(xc);
cout << "Level adjustments done." << endl;

// 计算 x^3
Ciphertext x3;
evaluator.multiply_inplace(x2, xc);
evaluator.relinearize_inplace(x2, relin_keys);
evaluator.rescale_to_next(x2, x3);
cout << "Calculated x^3." << endl;
```

(3) 计算 $xc \times x^2$ 并将该结果存储在变量 x_3 中

将更新后的 xc（即 $1.0 \times x$ 的结果）与 $x2$（即 x^2 的结果）进行同态乘法操作，得到 x^3 的结果存储在新的 Ciphertext 对象 $x3$ 中。

```
// 计算 y*z
Ciphertext yz;
evaluator.multiply(yc, zc, yz);
evaluator.relinearize_inplace(yz, relin_keys);
evaluator.rescale_to_next_inplace(yz);
cout << "Calculated y*z." << endl;
```

(4) 计算 $y \times z$ 并将结果存储在变量 yz 中

类似于 x 的处理，首先对 y 和 z 向量进行编码和加密，得到它们的密文表示 yc 和 zc。使用 Evaluator 对象对 yc 和 zc 进行同态乘法操作，得到 $y \times z$ 的加密结果存储在新的 Ciphertext 对象 yz 中。

```
// 将 x^3 和 y*z 的scale和层级统一
x3.scale() = pow(2.0, 40);
yz.scale() = pow(2.0, 40);
parms_id_type last_parms_id = x3.parms_id();
evaluator.mod_switch_to_inplace(yz, last_parms_id);
cout << "Scale and level of x^3 and y*z are unified." << endl;
```

(5) 计算 $x^3 + y \times z$。

将 $x3$（即 x^3 的结果）与 yz（即 $y \times z$ 的结果）进行同态加法操作，得到最终结果 $x^3 + y \times z$ 的加密形式。使用 Decryptor 对象对最终结果进行解密，再通过 CKKSEncoder 解码回明文的浮点数向量，以便查看计算结果。

```
// 将 x^3 和 y*z 相加得到最终结果
Ciphertext encrypted_result;
evaluator.add(x3, yz, encrypted_result);
cout << "Homomorphic computation of x^3 + y*z is complete." << endl;

// 解密和解码结果
Plaintext result_p;
decryptor.decrypt(encrypted_result, result_p);
vector<double> result;
encoder.decode(result_p, result);
cout << "Decryption and decoding completed, result is:\n" << endl;
print_vector(result, 3 /* precision */);
```

3. 重线性化

在同态加密计算中，尤其是在执行多步骤的算术操作时，重线性化（Relinearization）是一个关键过程，确保了计算的高效性和密文大小的可控性。这个过程通过使用预先生成的重线性化密钥（RelinKeys）来实现。

(1) 为什么需要重线性化

当我们在同态加密框架中进行乘法操作时，比如将两个密文相乘，结果密文的项数（也可以理解为多项式的度数）会增加。这个增加的项数不仅会使后续的计算变得更加复杂和耗时，而且还会增加密文的大小，导致更高的存储和传输成本。

重线性化操作的目的是将这个因乘法操作而增加的项数减少回原来的水平。具体来说，如果没有进行重线性化，乘法后的密文可能需要多个密文系数来表示，重线性化可以将这些密文压缩成更简洁的形式，减少密文系数的数量，而不改变密文所代表的值。

(2) 实现重线性化

在同态乘法中，两个密文相乘会导致结果的密文项数增加，这不仅会增加后续计算的复杂性，还会增加存储和传输的成本。通过使用重线性化键（RelinKeys），可以将这个增加的项数减少回原来的水平，而不改变密文的实际值。

```
Evaluator evaluator(context);
Ciphertext encrypted_result;
evaluator.multiply(encrypted_x, encrypted_y, encrypted_result);

// 重线性化操作，将乘法后的结果重线性化以减少其项数
evaluator.relinearize_inplace(encrypted_result, relin_keys);
```

在上述代码段中，multiply 函数将两个密文相乘，并将结果存储在 encrypted_result 中。然后，relinearize_inplace 函数使用预先生成的重线性化密钥 relin_keys 来处理 encrypted_result，减少其项数。

(3) 重线性化的好处

通过重线性化，我们可以保持后续计算的效率和密文的大小可控。这对于执行复杂的同态加密算法尤其重要，因为这些算法可能涉及多步骤的乘法和加法操作。重线性化确保了这些操作能够在不牺牲性能和安全性的情况下顺利进行。

4. 缩放操作

缩放操作是在同态乘法之后用于减少密文比例因子（scale）的过程。CKKS 方案中的每次乘法都会导致结果的比例因子增大，这可能会导致精度问题或溢出。通过缩放操作，可以降低比例因子，从而维持数值的稳定性和计算的准确性。

```
double scale = pow(2.0, 40); // 假设原始比例因子是2^40
Ciphertext encrypted_result;
evaluator.multiply(encrypted_x, encrypted_y, encrypted_result); // 执行乘法
evaluator.relinearize_inplace(encrypted_result, relin_keys); // 重线性化
evaluator.rescale_to_next_inplace(encrypted_result); // 缩放操作
```

在乘法和重线性化之后，rescale_to_next_inplace 函数对 encrypted_result 进行缩放操作，以减少其比例因子。这个函数实际上将密文的比例因子除以一个与当前密文使用的最小模数相对应的值，从而降低比例因子的大小，并确保后续操作中数值的稳定性。

5. 层级调整

(1) 同态加密中的模数链

在 CKKS 方案中，每个密文都与一个模数链相关联，这个链定义了可以在密文上执行的操作数量。每次进行乘法操作后，都需要执行缩放（rescaling）操作来减少密文的模数（即在模数链中向下移动一级），这有助于控制精度和避免溢出。由于这个原因，执行了不同数量乘法操作的两个密文可能处于模数链中的不同层级。

当想要对两个处于不同层级的密文执行加法或其他操作时，直接操作是不可能的，因为它们的模数不匹配。这就像是试图直接相加两个使用不同基数系统的数字一样——在没有适当转换的情况下，操作是没有意义的。因此，我们需要一种方法来“同步”这些密文的层级，以便它们可以参与同一操作。

(2) 层级调整的实现

mod_switch_to_inplace 函数正是用于这种层级同步的工具。它调整一个密文的层级，使之与另一个给定层级的密文匹配。在调整层级时，它实际上是在减小密文的模数，以匹配目标层级的模数。

```
Ciphertext encrypted_result;
// 假设encrypted_result和encrypted_other处于不同层级
evaluator.mod_switch_to_inplace(encrypted_other, encrypted_result.parms_id());
```

在这个例子中：encrypted_result 是我们的基准密文，我们想要将另一个密文 encrypted_other 调整到这个密文的层级。evaluator.mod_switch_to_inplace(encrypted_other, encrypted_result.parms_id()) 这行代码执行了实际的层级调整。parms_id() 方法返回了 encrypted_result 密文的参数 ID，这个 ID 唯一标识了其层级和模数链的位置。执行这个操作后，encrypted_other 的层

级被调整，使其与 encrypted_result 相同，从而允许之后在它们之间执行加法或其他兼容的操作。

(3) 层级调整的影响

- **数据精度**：层级调整通过减小模数来实现，这可能会影响密文中表示的数据的精度。因此，在设计算法时需要考虑这一点，以确保最终结果的准确性。
- **操作能力**：每次层级调整都相当于在模数链上向下移动一级，这减少了可以在密文上执行的操作数量。因此，合理规划操作顺序和次数对于维持计算能力至关重要。

6. 实验结果

完整代码详见附件（代码一）

```
-------------------------
[SEAL Context Initialized] Encryption parameters are set.
[Keys Generated] Public and secret keys, and relinearization keys are ready.
[Vectors Encrypted] x, y, and z vectors are now encrypted.
~~ Homomorphic Computation Begins ~~
Calculating: x^3 + y*z
Calculated x^2.
Level adjustments done.
Calculated x^3.
Calculated y*z.
Scale and level of x^3 and y*z are unified.
Homomorphic computation of x^3 + y*z is complete.
Decryption and decoding completed, result is:


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

f@f-virtual-machine:~/Desktop/Lab3/demo$
```

三、 实验心得与体会

在本次实验中，我对同态加密技术的潜力和挑战有了更深刻的认识。CKKS 方案允许在加密数据上进行复杂的算术运算，而不需要解密，这对于保护数据隐私来说是一个巨大的进步。通过实验，我学习到了如何合理选择密文参数（如多项式模数、参数模数和规模），以确保既能满足计算精度又能保证效率。我意识到参数的选择对于实验结果的精度和效率有着至关重要的影响。选择合适的多项式模数、参数模数序列和规模不仅影响着计算的可行性，还直接关系到计算的安全性和效率。通过实践，我深刻体会到了在实际应用中，如何根据计算需要和安全要求来选择合适的参数。

实验中的编程实践也让我对重线性化和缩放操作有了更直观的理解。重线性化是优化加密计算过程的重要步骤，它通过减少乘法操作后密文的项数来维持计算的高效性。缩放操作则是确保计算精度的关键，通过调整比例因子来防止精度损失。此外，层级调整的概念和操作使我认识到，在执行多步骤运算时，需要精心设计计算过程，以保证密文在整个计算过程中的有效性和一致性。

附 件

代码一

```
#include "examples.h"
#include <vector>
using namespace std;
using namespace seal;

#define N 3

int main() {
    // 初始化向量数据
    vector<double> x, y, z;
    x = {1.0, 2.0, 3.0};
    y = {2.0, 3.0, 4.0};
    z = {3.0, 4.0, 5.0};
    cout << "Initial vector x: " << endl;
    print_vector(x);
    cout << "Initial vector y: " << endl;
    print_vector(y);
    cout << "Initial vector z: " << endl;
    print_vector(z);
    cout << "------------------------" << endl;

    // 设置加密参数
    EncryptionParameters parms(scheme_type::ckks);
    size_t poly_modulus_degree = 8192;
    parms.set_poly_modulus_degree(poly_modulus_degree);
    parms.set_coeff_modulus(CoeffModulus::Create(poly_modulus_degree, {60, 40, 40, 60}));
    double scale = pow(2.0, 40);

    // 初始化SEAL上下文
    SEALContext context(parms);
    cout << "[SEAL Context Initialized] Encryption parameters are set." << endl;

    // 生成密钥
    KeyGenerator keygen(context);
    auto secret_key = keygen.secret_key();
    PublicKey public_key;
    keygen.create_public_key(public_key);
    RelinKeys relin_keys;
    keygen.create_relin_keys(relin_keys);
    cout << "[Keys Generated] Public and secret keys, and relinearization
```

```
41          keys are ready." << endl;
42
43      // 加密向量
44      Encryptor encryptor(context, public_key);
45      Evaluator evaluator(context);
46      Decryptor decryptor(context, secret_key);
47      CKKSEncoder encoder(context);
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      Ciphertext xc, yc, zc;
54      encryptor.encrypt(xp, xc);
55      encryptor.encrypt(yp, yc);
56      encryptor.encrypt(zp, zc);
57      cout << "[Vectors Encrypted] x, y, and z vectors are now encrypted." << endl;
58
59      // 进行同态计算
60      cout << "~~Homomorphic Computation Begins~~\nCalculating: x^3 + y*z" << endl;
61      // 计算 x^2
62      Ciphertext x2;
63      evaluator.multiply(xc, xc, x2);
64      evaluator.relinearize_inplace(x2, relin_keys);
65      evaluator.rescale_to_next_inplace(x2);
66      cout << "Calculated x^2." << endl;
67
68      // 调整层级
69      Plaintext plain_one;
70      encoder.encode(1.0, scale, plain_one);
71      evaluator.multiply_plain_inplace(xc, plain_one);
72      evaluator.rescale_to_next_inplace(xc);
73      cout << "Level adjustments done." << endl;
74
75      // 计算 x^3
76      Ciphertext x3;
77      evaluator.multiply_inplace(x2, xc);
78      evaluator.relinearize_inplace(x2, relin_keys);
79      evaluator.rescale_to_next(x2, x3);
80      cout << "Calculated x^3." << endl;
81
82      // 计算 y*z
83      Ciphertext yz;
84      evaluator.multiply(yc, zc, yz);
```

```
85      evaluator.relinearize_inplace(yz, relin_keys);
86      evaluator.rescale_to_next_inplace(yz);
87      cout << "Calculated y*z." << endl;
88
89      // 将 x^3 和 y*z 的scale和层级统一
90      x3.scale() = pow(2.0, 40);
91      yz.scale() = pow(2.0, 40);
92      parms_id_type last_parms_id = x3.parms_id();
93      evaluator.mod_switch_to_inplace(yz, last_parms_id);
94      cout << "Scale and level of x^3 and y*z are unified." << endl;
95
96      // 将 x^3 和 y*z 相加得到最终结果
97      Ciphertext encrypted_result;
98      evaluator.add(x3, yz, encrypted_result);
99      cout << "Homomorphic computation of x^3 + y*z is complete." << endl;
100
101     // 解密和解码结果
102     Plaintext result_p;
103     decryptor.decrypt(encrypted_result, result_p);
104     vector<double> result;
105     encoder.decode(result_p, result);
106     cout << "Decryption and decoding completed, result is:\n" << endl;
107     print_vector(result, 3 /* precision */);
108
109     return 0;
110 }
```