《数据安全》实验报告

姓名：管昀玫 学号：2013750 班级：计科一班

**实验名称：**

SEAL应用实践

**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，完成的运算。

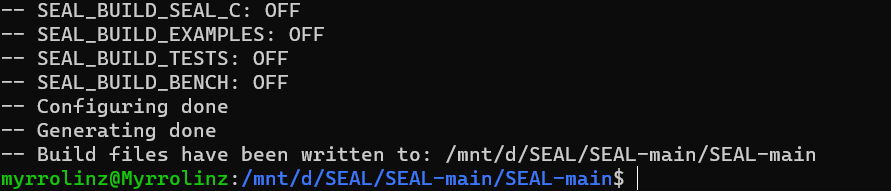
**实验过程：**

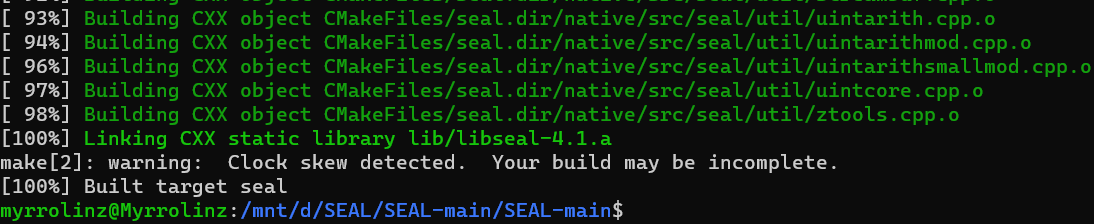
CKKS 是一个公钥加密体系，具有公钥加密体系的一切特点，例如公钥加密、 私钥解密等。代码中需要以下组件：

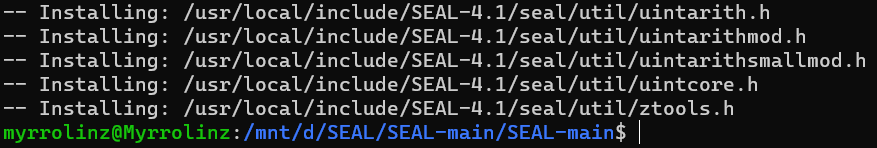
* 密钥生成器 keygenerator
* 加密模块 encryptor
* 解密模块 decryptor
* 密文计算模块 evaluator

1. 安装SEAL库

首先需要git clone相关GitHub repository，然后执行以下命令：

cmake .  


make  


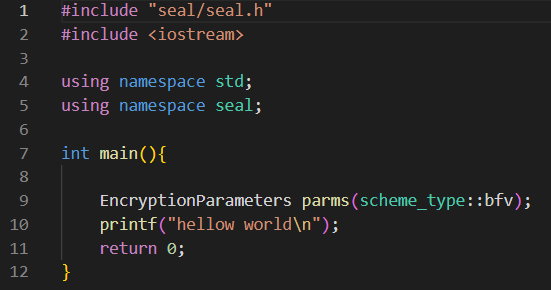
sudo make install  


由于网络原因，cmake时尝试多次才成功。个人经验：科学上网就能顺利安装。

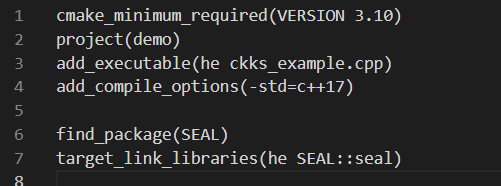
至此，SEAL库安装成功。

1. 简单测试程序

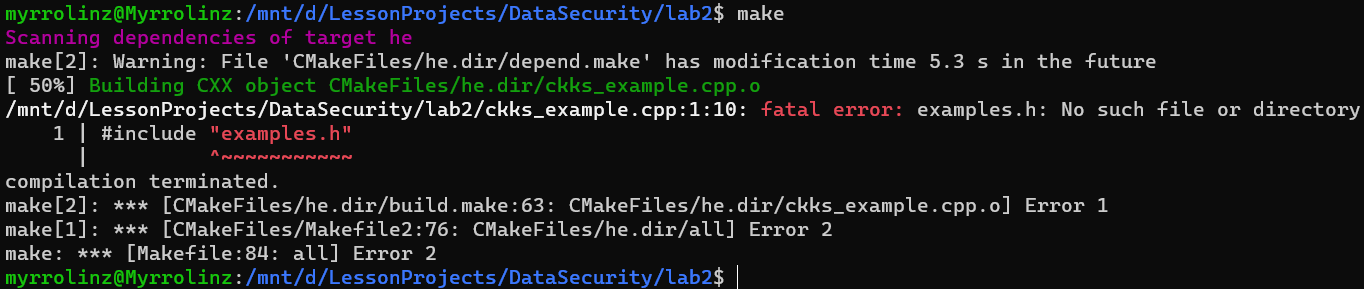
test.cpp如下所示:

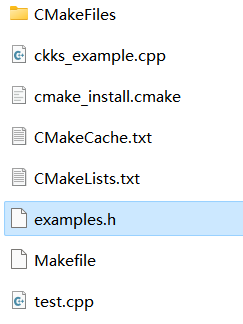
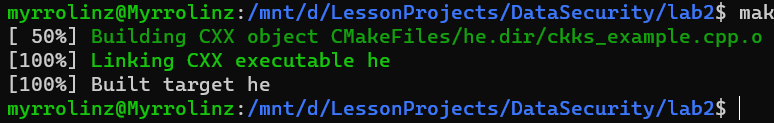
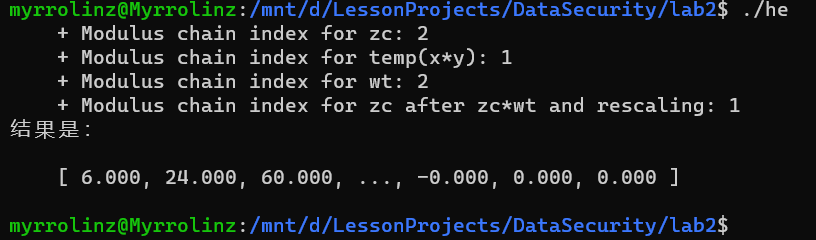


为了完成 test.cpp 的编译和执行，需要编写一个 CMakeLists.txt 文件，内容如下：



之后执行以下命令：

**cmake .**  
  
 **make**  


此处提示没有examples.h，只要从SEAL/native/examples中复制examples.h到我们的文件夹中即可。  
  
  
 **./he**  
  
 发现结果是正确的。

1. 计算

3.1 参数

首先我们需要注意的是，CKKS 是一个（level）全同态加密算法（level 表示其运算深度仍然存在限制），做乘法时两个数必须在同一个level上。

除此之外，我们还需要注意三个参数：

1. **poly\_modulus\_degree（polynomial modulus）**

该参数必须是 2 的幂，如 1024, 2048, 4096, 8192, 16384, 32768...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。

1. **[ciphertext] coefficient modulus**

coeff\_modules 的个数决定了进行 rescaling 的次数，进而决定了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，我在此处选择coeff\_modules\_degree为8194. 如果模数变大，则可以支持更多层级的乘法运算。

1. **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。

此处我使用的参数为：

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 }));

//选用2^40进行编码

double scale = pow(2.0, 40);

3.2 计算

需要注意4个运算规则为：

1.加法可以连续运算，但乘法不能连续运算

2.密文乘法后要进行 relinearize 操作

3.执行乘法后要进行 rescaling 操作

4.进行运算的密文必需执行过相同次数的 rescaling（位于相同 level）

要想把不同 level 的数据拉到同一 level，可以利用乘法单位元 1 把层数较高的操作数拉到较低的 level（如本例），也可以通过内置函数进行直接转换。

因此，合理的顺序为：

1. 计算x\*x
2. 计算x\*1.0
3. 计算(x\*x) \* (x\*1.0)
4. 计算y\*1.0
5. 计算z\*1.0
6. 计算(y\*1.0) \* (z\*1.0)
7. 计算x^3+y\*z

源代码与解释如下所示：

/    Ciphertext temp1;

    Ciphertext result\_c1;

//计算x\*x，密文相乘，要进行relinearize和rescaling操作

    evaluator.multiply(xc,xc,temp1);

    evaluator.relinearize\_inplace(temp1, relin\_keys);

    evaluator.rescale\_to\_next\_inplace(temp1);

//在计算x\*x\*x之前，x没进行过rescaling操作，所以需要对x进行一次乘法和rescaling操作，目的是使得x\*x和x在相同的层

    Plaintext wt;

    encoder.encode(1.0, scale, wt);

//此时，我们可以查看框架中不同数据的层级：

cout << "    + Modulus chain index for xc: "

<< context\_server.get\_context\_data(xc.parms\_id())->chain\_index() << endl;

cout << "    + Modulus chain index for temp(x\*x): "

<< context\_server.get\_context\_data(temp1.parms\_id())->chain\_index() << endl;

cout << "    + Modulus chain index for wt: "

<< context\_server.get\_context\_data(wt.parms\_id())->chain\_index() << endl;

//执行乘法和rescaling操作：

    evaluator.multiply\_plain\_inplace(xc, wt);

    evaluator.rescale\_to\_next\_inplace(xc);

//再次查看xc的层级，可以发现xc与temp1层级变得相同

cout << "    + Modulus chain index for zc after zc\*wt and rescaling: "

<< context\_server.get\_context\_data(xc.parms\_id())->chain\_index() << endl;

//最后执行temp（x\*x）\* xc（x\*1.0）

    evaluator.multiply\_inplace(temp1, xc);

    evaluator.relinearize\_inplace(temp1,relin\_keys);

    evaluator.rescale\_to\_next(temp1, result\_c1);

    Ciphertext result\_c2;

//在计算x\*x\*x+y\*z之前，需要对y\*Z进行一次乘法和rescaling操作，目的是使得x\*x\*x和y\*z在相同的层

    Plaintext wt1;

    encoder.encode(1.0, scale, wt1);

//此时，我们可以查看框架中不同数据的层级：

cout << "    + Modulus chain index for yc: "

<< context\_server.get\_context\_data(yc.parms\_id())->chain\_index() << endl;

cout << "    + Modulus chain index for zc: "

<< context\_server.get\_context\_data(zc.parms\_id())->chain\_index() << endl;

cout << "    + Modulus chain index for wt1: "

<< context\_server.get\_context\_data(wt1.parms\_id())->chain\_index() << endl;

//执行乘法和rescaling操作：

    evaluator.multiply\_plain\_inplace(yc, wt1);

    evaluator.rescale\_to\_next\_inplace(yc);

//执行乘法和rescaling操作：

    evaluator.multiply\_plain\_inplace(zc, wt1);

    evaluator.rescale\_to\_next\_inplace(zc);

//最后执行(xc\*1.0)\*(yc\*1.0)

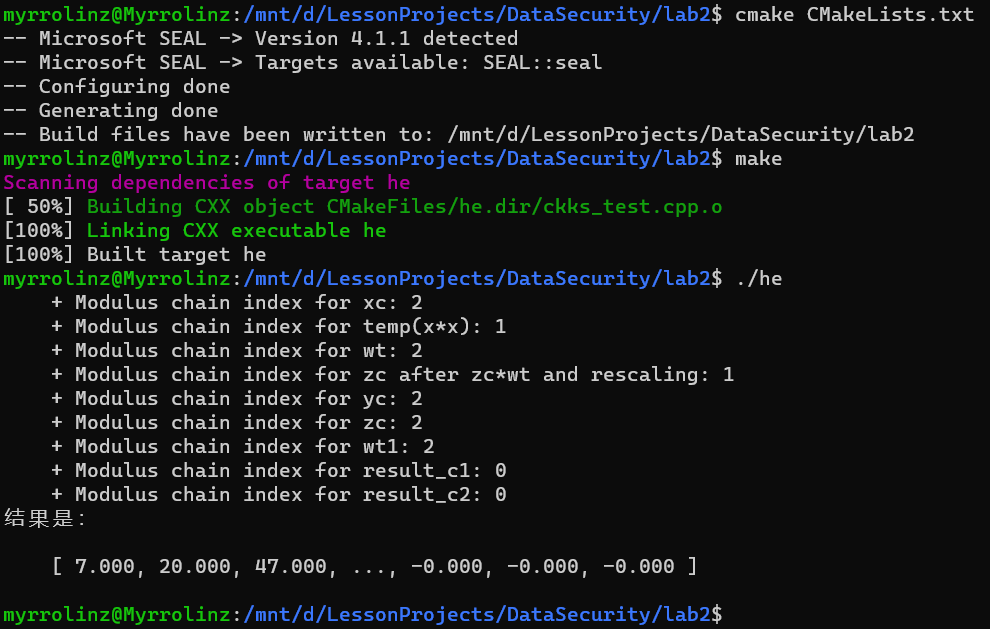
//计算y\*z，密文相乘，要进行relinearize和rescaling操作

    evaluator.multiply(yc,zc,result\_c2);

    evaluator.relinearize\_inplace(result\_c2, relin\_keys);

    evaluator.rescale\_to\_next\_inplace(result\_c2);

最终结果为：



**心得体会：**

通过此次实验，我学习了使用SEAL基于CKKS方案构建一个运算。

在此次实验中，我踩了许多坑，比如没有设计好运算顺序，导致y\*z算完了之后想计算1.0\*1.0再与y\*z相乘，以把他们拉到与x\*x\*x\*1.0的同一层，但是这个方法一直在报参数错误的问题，所以我第二次就改变了运算顺序，先算y\*1.0和z\*1.0，然后再把他们相乘，这样就能与x\*x\*x\*1.0在同一层而不报错了。

第二个坑是参数问题。我之前写代码的时候由于逻辑错误报错，因此尝试了另一组参数，后来才排查出来是逻辑问题。另一组参数为：

size\_t poly\_modulus\_degree = 8192;

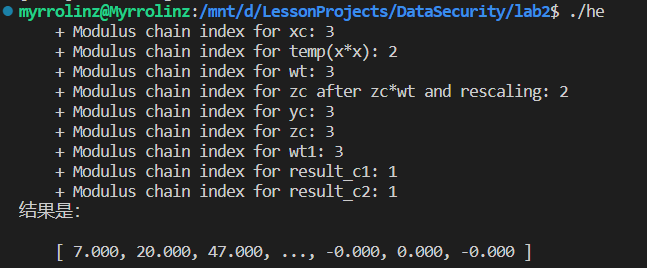
parms.set\_poly\_modulus\_degree(poly\_modulus\_degree);

parms.set\_coeff\_modulus(CoeffModulus::Create(poly\_modulus\_degree, { 50,30,30,30,50 }));

//选用2^40进行编码

double scale = pow(2.0, 30);

在逻辑正确的前提下替换另一组参数，此时仍然work：



我对三个参数比较好奇，因此查阅了资料：

在 *SEAL* 中，用于加密的参数集合中的每个元素(处理随机数生成器)都可以使用一个 *256*-bits 的参数哈希来唯一表示。该哈希值被称为 parms\_id，并可以被随时访问和输出。该哈希值会随着参数的改变而改变。

给定 EncryptionParameters 去创建 SEALContext 对象时，*SEAL* 自动去创建一个 *modulus switching chain*，该 *chain* 是从最开始的集合中获得的加密参数。*modulus switching chain* 中的参数和最开始的参数相同，只是模系数的大小会沿着 *chain* 逐渐减少。准确来说，每次进行 *switching* 时，会从 *chain* 中移除最后一个模素数，该过程直到参数集合非法为止。同时，*chain* 的参数集合中的每个参数有自己的 *chain index*，以表示该参数在 *chain* 中的位置，因此最后一个参数的下标为 0。如果，某一个 *chain* 中的下标值更大的话，那么我们可以认为该加密参数处于一个更高的 *level*，例如最开始的参数。

对于 *chain* 上的参数集合，当 SEALContext 对象被创建时，这些参数会进行一些唯一的预计算并把其结果保存在 SEALContext::ContextData 对象中。而 *chain* 作为 SEALContext::ContextData 对象基本链表，可以通过 SEALContext::ContextData 进行访问。而对于链表中的每个节点可以通过 parms\_id 识别 (poly\_modulus\_degree 保持不变，而coeff\_modulus 随着 switching 而改变)。

除此之外，我还了解了更多有关modulus switching的知识：

*进行 modulus switching 的好处在于：因为密文的大小和 coeff\_modulus 中素数的数量呈正相关，如果不需要对给定的密文执行任何进一步的计算的话，我们最好在将其发送回密钥持有者进行解密之前，将其切换到链中最小（最后）的参数集。如果我们处理正确的话，损失的噪声预算对结果没有丝毫影响。即便移除了一个模数，也不会影响计算。可能为了在小模数下以获得计算效率，越早向下一层进行 switching 越好，事实上该过程会消耗一些噪声预算。可以从结果看出，理想情况下应该在噪声预算降至 25-bits 时向下一个 level 进行切换。*