



南開大學

Nankai University

计算机学院
数据安全实验报告

SEAL 实践应用

姓名：李欣

学号：2011165

专业：计算机科学与技术

2023 年 3 月 25 日

目录

1 实验目的	2
2 实验原理与代码解读	2
2.1 CKKS 算法	2
2.1.1 容错学习	2
2.1.2 方案构造	3
2.1.3 再线性化和再缩放	3
2.2 实验原理	4
2.2.1 标准化构建流程	4
2.3 代码解读	4
2.3.1 注意事项	5
2.3.2 客户端加密	5
2.3.3 服务端计算	6
2.3.4 客户端解密	7
3 实验过程	8
3.1 SEAL 库下载	8
3.2 代码修改	9
3.3 运行结果	11
4 遇到的困难与解决方法	11
4.1 虚拟机无法连接 github 进行 git clone	11
5 心得体会	11

1 实验目的

教材实验演示了一个基于云服务器的算例协助完成客户端的某种运算，示例代码给出的是计算 $x * y * z$ 。本次实验要求实现将三个数的密文发送到服务器，完成 $x * 3 + y * z$ 的计算。

2 实验原理与代码解读

2.1 CKKS 算法

2.1.1 容错学习

容错学习 (Learning with Error, LWE) 是在格的难题上构建出来的问题，可以看作解一个带噪声的线性方程组：给定随机向量 $\mathbf{s} \in \mathbb{Z}_q^n$ 、随机选择线性系数矩阵 $\mathbf{A} \in \mathbb{Z}_q^{n \times n}$ 和随机噪声 $\mathbf{e} \in \mathbb{Z}_q^n$ ，生成矩阵线性运算结果 $(\mathbf{A}, \mathbf{A} \cdot \mathbf{s} + \mathbf{e})$ 。LWE 问题试图从该结果中反推 \mathbf{s} 的值，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。

LWE 问题使得在其上构建的加密系统实现十分简单：

- 密钥生成函数：给定随机向量 $\mathbf{s} \in \mathbb{Z}_q^n$ ，随机选择线性系数矩阵 $\mathbf{A} \in \mathbb{Z}_q^{n \times n}$ 和随机噪声 $\mathbf{e} \in \mathbb{Z}_q^n$ ，将 $(-\mathbf{A} \cdot \mathbf{s} + \mathbf{e}, \mathbf{A})$ 作为公钥， \mathbf{s} 作为私钥。
- 加密函数：对于需要加密的消息 $\mathbf{m} \in \mathbb{Z}_q^n$ ，使用公钥加密为 $(\mathbf{c}_0, \mathbf{c}_1) = (\mathbf{m} - \mathbf{A} \cdot \mathbf{s} + \mathbf{e}, \mathbf{A})$ 。
- 解密函数：使用私钥进行解密，计算 $\mathbf{c}_0 + \mathbf{c}_1 \cdot \mathbf{s} = \mathbf{m} + \mathbf{e}$ ，当噪声 \mathbf{e} 足够小时，能够尽可能地恢复明文 \mathbf{m} 。

2.1.2 方案构造

CKKS 层次同态加密方案即是基于上述 RLWE 问题实现的。具体实现如下：

- 密钥生成函数：给定安全参数，CKKS 生成私钥 $s \in \mathbb{Z}_q[X]/(X^N + 1)$ 和公钥 $p = (-a \cdot s + e, a)$ 。式中， a, e 皆表示多项式环中随机抽取的元素 $a, e \in \mathbb{Z}_q[X]/(X^N + 1)$ ，且 e 为较小噪声。
- 加密函数：对于给定的一个消息 $m \in \mathbb{C}^{N/2}$ （表示为复数向量），CKKS 首先需要对其进行编码，将其映射到多项式环中生成 $r \in \mathbb{Z}[X]/(X^N + 1)$ 。然后，CKKS 使用公钥对 r 进行如下加密：

$$(c_0, c_1) = (r, 0) + p = (r - a \cdot s + e, a)$$

- 解密函数：对密文 (c_0, c_1) ，CKKS 使用密钥进行如下解密：

$$\tilde{r} = c_0 + c_1 * s = r + e$$

\tilde{r} 需要经过解码，从多项式环空间反向映射回向量空间 $\mathbb{C}^{N/2}$ 。当噪声 e 足够小时，可以获得原消息的近似结果。

CKKS 支持浮点运算，为保存消息中的浮点数，在编码过程中 CKKS 设定缩放因子 $\Delta > 0$ ，并将浮点数乘以缩放因子生成整数的多项式项，其浮点值被保存在缩放因子 Δ 中。

2.1.3 再线性化和再缩放

CKKS 支持同态加法和同态乘法。给定两个密文 ct_1 和 ct_2 ，其对应的同态加法如下：

$$ct_1 + ct_2 = (c_0, c_1) + (c'_0, c'_1) = (c_0 + c'_0, c_1 + c'_1)$$

对应的同态乘法操作如下：

$$ct_1 \cdot ct_2 = (c_0, c_1) \cdot (c'_0, c'_1) = (c_0 \cdot c'_0, c_0 \cdot c'_1 + c_1 \cdot c'_0, c_1 \cdot c'_1)$$

可见，在进行同态乘法操作后，密文的大小扩增了一半。因此，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。

再线性化。再线性化技术能够将扩增的密文 $(c_0 \cdot c'_0, c_0 \cdot c'_1 + c_1 \cdot c'_0, c_1 \cdot c'_1)$ 再次恢复为二元对 (d_0, d_1) ，从而允许进行更多的同态乘法操作。

再缩放。另外，因为在编码消息的过程中使用了缩放因子 Δ ，在进行同态乘法操作时，两个缩放因子皆为 Δ 的密文相乘，其结果的缩放因子变为 Δ^2 。如果连续使用同态乘法，缩放因子将会呈指数级上升。所以，每次乘法操作之后，CKKS 都会进行再缩放的操作，将密文值除以 Δ 以将缩放因子从 Δ^2 恢复到 Δ 。在不断再缩放除以 Δ 的过程中，表示密文值得可用比特每次会下降 $\log(\Delta)$ 比特，直到最终用尽。此时，无法再继续进行同态乘法。

在 CKKS 的加密、解密、再线性化和再缩放的过程中，积累增加的噪声会影响最终解密消息的精度和准度。所以，CKKS 支持浮点运算的同时，对结果的准确性做出了一定的牺牲。CKKS 适用于允许一定误差的、基于浮点数的计算应用，比如机器学习任务。

自举。CKKS 中的再线性化和再缩放是为了保证缩放因子不变，同时降低噪音，但会造成密文模数减少，所以只能构成有限级全同态方案。CKKS 的自举操作能提高密文模数，以支持无限次数的全同态，但是自举成本很高，在满足需求的时候，甚至不需要执行自举操作，后来有一些研究针对 CKKS 方案的自举操作做了精度和效率的提升。

2.2 实验原理

2.2.1 标准化构建流程

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，由于 CKKS 是一个 Level 全同态加密算法，且 CKKS 所基于的数学困难问题在一个“多项式环”上（环上的元素与实数并不相同），所以我们的代码需要以下五个组件：

- 密钥生成器 keygenerator
- 加密模块 encryptor
- 解密模块 decryptor
- 密文计算模块 evaluator
- 编码器 encoder

综上，CKKS 的总体构建过程如下

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

2.3 代码解读

同态加密算法在云计算中的应用基本流程如下：

1. 发送方利用公钥 pk 加密明文 m 为密文 c

2. 发送方把密文 c 发送到服务器
3. 服务器执行密文运算，生成结果密文 c'
4. 服务器将结果密文 c' 发送给接收方
5. 接收方利用私钥 sk 解密密文 c' 为明文结果 m'

当发送方与接收方相同时，则该客户利用全同态加密算法完成了一次安全计算，即既利用了云计算的算力，又保障了数据的安全性，这对云计算的安全应用有重要意义。

本例目的：给定 x, y, z 三个数的密文，让服务器计算 $x * y * z$ 。在这里，我们不会对每一行代码进行解读，仅解读关键部分。

2.3.1 注意事项

如示例代码中所述，每次进行运算前，要保证参与运算的数据位于同一“level”上。

加法不需要进行 rescaling 操作，因此不会改变数据的 level。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。

可以通过输出 `p.scale()`、`p.parms_id()` 以及 `context->get_context_data(p.parms_id())->chain_index()` 来确认即将进行的操作的数据满足如下计算条件：

- 1) 用同一组参数进行加密；
- 2) 位于 (chain) 上的同一 level；
- 3) scale 相同。

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。

目前，SEAL 提供了 reverse、square 等有限的计算操作，大部分复杂运算需要自己编写代码实现，在实现过程中要根据数据量把握好精度和性能的取舍。

2.3.2 客户端加密

客户端负责生成参数、构建环境和生成密文

- 构建参数容器 `parms`

CKKS 有三个重要参数：`poly_module_degree` (多项式模数)、`coeff_modulus` (参数模数)、`scale` (规模)，选用 2^{40} 进行编码，代码实现中参数的选择会在代码修改中给出。

```

1  EncryptionParameters parms(scheme_type::ckks);
2  size_t poly_modulus_degree = 8192;
3  parms.set_poly_modulus_degree(poly_modulus_degree);
4  parms.set_coeff_modulus(CoeffModulus::Create(poly_modulus_degree,
5  { 60, 40, 40, 60 }));
6  double scale = pow(2.0, 40);

```

- 用参数生成 CKKS 框架 `context`

```
1 SEALContext context(parms);
```

- 构建各模块

```
1 //首先构建 keygenerator, 生成公钥、私钥
2 KeyGenerator keygen(context);
3 auto secret_key = keygen.secret_key();
4 PublicKey public_key;
5     keygen.create_public_key(public_key);
6
7 //构建编码器, 加密模块、运算器和解密模块
8 //注意加密需要公钥 pk; 解密需要私钥 sk; 编码器需要 scale
9 Encryptor encryptor(context, public_key);
10 Decryptor decryptor(context, secret_key);
11 CKKSEncoder encoder(context);
12 //对向量 x、y、z 进行编码 (仅展示 xp)
13 Plaintext xp;
14 encoder.encode(x, scale, xp);
15 //对明文 xp、yp、zp 进行加密 (仅展示 xc)
16 Ciphertext xc;
17 encryptor.encrypt(xp, xc);
```

至此, 客户端将 pk、CKKS 参数发送给服务器, 服务器开始运算。

2.3.3 服务端计算

服务器主要负责生成重线性密钥、构建环境和执行密文计算

- 生成重线性密钥和构建环境
- 对密文进行计算
 - 加法可以连续运算, 但乘法不能连续运算
 - 密文乘法后要进行 relinearize 操作
 - 执行乘法后要进行 rescaling 操作
 - 进行运算的密文必需执行过相同次数的 rescaling (位于相同 level)

```
1 //生成重线性密钥和构建环境
2 SEALContext context_server(parms);
3 RelinKeys relin_keys;
4 keygen.create_relin_keys(relin_keys);
5 Evaluator evaluator(context_server);
```

```

6
7     Ciphertext temp;
8     Ciphertext result_c;
9     //计算  $x*y$ , 密文相乘, 要进行 relinearize 和 rescaling 操作
10    evaluator.multiply(xc, yc, temp);
11    evaluator.relinearize_inplace(temp, relin_keys);
12    evaluator.rescale_to_next_inplace(temp);
13
14    //在计算  $x*y * z$  之前,  $z$  没有进行过 rescaling 操作, 所以需要对  $z$  进行一次乘法
15    //和 rescaling 操作,
16    //目的是使得  $x*y$  和  $z$  在相同的层
17    Plaintext wt;
18    encoder.encode(1.0, scale, wt);
19    //此时, 我们可以查看框架中不同数据的层级: (省略代码)
20
21    //执行乘法和 rescaling 操作:
22    evaluator.multiply_plain_inplace(zc, wt);
23    evaluator.rescale_to_next_inplace(zc);
24
25    //再次查看  $zc$  的层级, 可以发现  $zc$  与  $temp$  层级变得相同
26    cout << "      + Modulus chain index for zc after zc*wt and rescaling: "
27    << context_server.get_context_data(zc.parms_id())->chain_index() << endl;
28
29    //最后执行  $temp (x*y) * zc (z*1.0)$ 
30    evaluator.multiply_inplace(temp, zc);
31    evaluator.relinearize_inplace(temp, relin_keys);
32    evaluator.rescale_to_next(temp, result_c);

```

计算完毕, 服务器把结果发回客户端

2.3.4 客户端解密

客户端将结果解码到一个向量上

```

1  Plaintext result_p;
2  decryptor.decrypt(result_c, result_p);
3  vector<double> result;
4  encoder.decode(result_p, result);

```

结果如图所示, 通过饰演的结果, 我们可以看到, 当进行计算时, 两个运算数时处于同一 level 的, 如计算 $x*y$ 时, x 和 y 都为 level1, 计算的结果 wt 为 level2, 而 z 不能与 wt 直接进行运算, 这是由于 z 为 level1 而 wt 为 level2, 二者不在同一层, 故需要将 z 变成 level2, 即 z 与常数 1 相乘, zc 为 level2, 至此, wt 可以与 zc 进行运算, 计算结果经验证是正确的。


```
gloria01@gloria01-virtual-machine:~/Desktop/data_security/seal/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 ]
```

图 2.1: $x*y*z$ 的运算结果

3 实验过程

3.1 SEAL 库下载

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 (<https://github.com/microsoft/SEAL>) 开源。SEAL 基于 C++ 实现, 不需要其他依赖库。

SEAL 库的安装步骤如下:

1. 打开终端, 输入命令: `git clone https://github.com/microsoft/SEAL`

```
gloria01@gloria01-virtual-machine:~/Desktop/data_security$ git clone https://git
hub.com/microsoft/SEAL
正克隆到 'SEAL'...
remote: Enumerating objects: 17111, done.
remote: Counting objects: 100% (282/282), done.
remote: Compressing objects: 100% (153/153), done.
remote: Total 17111 (delta 141), reused 229 (delta 110), pack-reused 16829
接收对象中: 100% (17111/17111), 5.00 MiB | 3.21 MiB/s, 完成.
处理 delta 中: 100% (12941/12941), 完成.
```

图 3.2: git clone

2. `cmake .`

```
-- SEAL_BUILD_BENCH: OFF
-- Configuring done
-- Generating done
-- Build files have been written to: /home/gloria01/Desktop/data_security/SEAL
gloria01@gloria01-virtual-machine:~/Desktop/data_security/SEAL$
```

图 3.3: cmake

3. `make`

```
[ 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
gloria01@gloria01-virtual-machine:~/Desktop/data_security/SEAL$
```

图 3.4: make

4. sudo make install

```
-- 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
gloria01@gloria01-virtual-machine:~/Desktop/data_security/SEAL$
```

图 3.5: make install

3.2 代码修改

本次实验的目的时计算 $x^3 + yz$ ，我们知道，在 CKKS 计算中，两两运算时，需要使操作数处于相同的 level。主要的困难为 x^3 与 yz 处于不同的 level，为使二者处于相同的 level，我们可以将式子改写成以下形式，便于理解和实现 $(x * x) * (x * 1) + (z * 1) * (y * 1)$

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，但也能让你执行更复杂的计算。

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，列表如下：

poly_modulus_degree	max coeff_module bit-length
1024	27
2048	54
4096	109
8192	218
16384	438
32768	881

表 1: coeff_modules 的最大位数和 poly_modules 的关系

本文例子中的 {60, 40, 40, 60} 有以下含义：

- (a) coeff_modules 总位长 200 (60+40+40+60) 位
- (b) 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：

- (a) 总位长不能超过上表限制
- (b) 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
- (c) 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`

在本次实验中，我们要计算 $x^*3 + z*y$ ，适当的将模数变大，以支持更大层级的乘法运算（尽管基础实验和本次实验的层级是一样的，均为四层），实验指导手册中提到的支持六层运算的 `poly_modulus` 为 16384，参数 `coeff_modules` 如代码所示。

修改参数

```
1 //mul 3 times
2 size_t poly_modulus_degree = 16384;
3 parms.set_poly_modulus_degree(poly_modulus_degree);
4 parms.set_coeff_modulus(CoeffModulus::Create(poly_modulus_degree, { 60, 40, 40,
    40, 40, 40, 40, 60 }));
```

计算 $x*x$ 和 $x*1$

```
1 // 1. calculate x*x and x*1
2 evaluator.multiply(xc,xc,tempxx);
3 evaluator.relinearize_inplace(tempxx, relin_keys);
4 evaluator.rescale_to_next_inplace(tempxx);
5
6 evaluator.multiply_plain_inplace(xc, wt);
7 evaluator.rescale_to_next_inplace(xc);
```

计算 $(x*x)*(x*1)$

```
1 // 2. calculate (x*x)*(x*1)
2 evaluator.multiply(tempxx,xc,add1);
3 evaluator.relinearize_inplace(add1, relin_keys);
4 evaluator.rescale_to_next_inplace(add1);
5 cout << "      + Modulus chain index for (x*x)*(x*1): "
6 << context_server.get_context_data(add1.parms_id())->chain_index() << endl;
```

计算 $(y*1)$ 和 $(z*1)$

```
1 // 3. calculate y*1 and z*1
2 evaluator.multiply_plain_inplace(y, wt);
```

```

3 evaluator.rescale_to_next_inplace(y);
4
5 evaluator.multiply_plain_inplace(zc, wt);
6 evaluator.rescale_to_next_inplace(zc);

```

计算 $(y*1)*(z*1)$

```

1 // 4. calculate (y*1)*(z*1)
2 evaluator.multiply(zc, yc, add2);
3 evaluator.relinearize_inplace(add2, relin_keys);
4 evaluator.rescale_to_next_inplace(add2);
5 cout << "      + Modulus chain index for (y*1)*(z*1): "
6 << context_server.get_context_data(add2.parms_id())->chain_index() << endl;

```

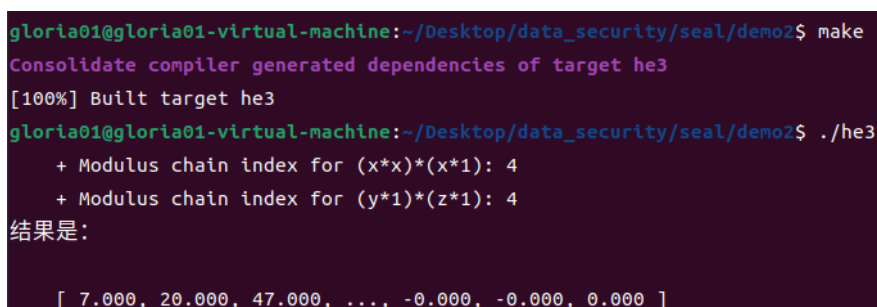
计算 $add1+add2$

```

1 // 5. calculate add1 + add2
2 evaluator.add_inplace(add1, add2);
3 result_c=add1;

```

3.3 运行结果



```

gloria01@gloria01-virtual-machine:~/Desktop/data_security/seal/demo2$ make
Consolidate compiler generated dependencies of target he3
[100%] Built target he3
gloria01@gloria01-virtual-machine:~/Desktop/data_security/seal/demo2$ ./he3
      + Modulus chain index for (x*x)*(x*1): 4
      + Modulus chain index for (y*1)*(z*1): 4
结果是：

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

```

图 3.6: $x**3+y*z$

4 遇到的困难与解决方法

4.1 虚拟机无法连接 github 进行 git clone

根据实验指导，多试几次也无法下载，于是参考网上的教程修改了虚拟机的网络设置，将虚拟机的网络与主机 vpn 连接，最后成功 ping 通 github，git clone 成功。

5 心得体会

1. 原理的学习：通过本次实验，我学习了 CKKS 算法，了解了 CKKS 算法的基本原理，能够掌握容错学习，CKKS 层次同态加密方案的方案构造，同时理解了再线性化和再缩放。CKKS 中的再线性化和再缩放是为了保证缩放因子不变，同时降低噪音，但会造成密文模数减少，所以只能构成有限级

全同态方案。CKKS 的自举操作能提高密文模数，以支持无限次数的全同态，但是自举成本很高，在满足需求的时候，甚至不需要执行自举操作，后来有一些研究针对 CKKS 方案的自举操作做了精度和效率的提升。同时学会使用了 SEAL 库来辅助编程。

2. 代码编程：代码要修改的地方其实并不是很多，需要注意的是，参与计算的两个密文必须处于同一层级下，否则难以进行运算。加法可以连续运算，但乘法不能连续运算，密文乘法后要进行 relinearize 操作，执行乘法后要进行 rescaling 操作。

3. 环境配置：配置环境是实验的基础，必须要正确地配置环境！