



南開大學
Nankai University

计算机学院和网络空间安全学院
数据安全实验报告

实验 2: SEAL 应用实践

姓名：魏伯繁

学号：2011395

专业：信息安全

2023 年 3 月 31 日

目录

1 实验要求	2
2 实验过程	2
2.1 学习 CKKS 算法基本知识与容错学习相关内容	2
2.2 复现 ckks_example	3
2.2.1 环境搭建	3
2.2.2 代码复现	4
2.3 代码修改	6
3 心得体会	11

实验名称：全同态加密 SEAL 应用实践

1 实验要求

如课本所示，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。所要计算的向量在客户端初始化完成并加密，云服务器需要通过提供的加密后的向量进行计算，在本次实验中，刘哲理老师已经给出在云服务端计算 xyz 的实现代码，需要进行改进，在云服务器改进为求取 x 的三次方加 y 和 z 的乘积

2 实验过程

2.1 学习 CKKS 算法基本知识与容错学习相关内容

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。

图 2.1: 容错学习基本概念

可见，基于容错学习设计的 CKKS 算法的安全性是很高的，并且在 CKKS 算法中还引入了多项式环、线性系数矩阵等内容，对密钥的长度进行了优化，使得使用者可以使用线性空间复杂度的密钥。

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 $\mathbf{s} \in \mathbb{Z}_q[X]/(X^N + 1)$ 和公钥 $\mathbf{p} = (-\mathbf{a} \cdot \mathbf{s} + \mathbf{e}, \mathbf{a})$ 。式中， \mathbf{a}, \mathbf{e} 皆表示多项式环中随机抽取的元素 $\mathbf{a}, \mathbf{e} \in \mathbb{Z}_q[X]/(X^N + 1)$ ，且 \mathbf{e} 为较小噪声。
- 加密函数：对于给定的一个消息 $\mathbf{m} \in \mathbb{C}^{N/2}$ （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 $\mathbf{r} \in \mathbb{Z}[X]/(X^N + 1)$ 。然后，CKKS 使用公钥对 \mathbf{r} 进行如下加密：

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

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

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

$\tilde{\mathbf{r}}$ 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 \mathbf{e} 足够小时，可以获得原消息的近似结果。

图 2.2: CKKS 层次同态加密原理

其中，CKKS 算法的两个最大特点就是再线性化和再缩放，因为在 CKKS 中如果进行了同台乘法，那么密文的大小扩增了一半，所以说每次乘法操作后都需要进行再线性化和再缩放操作。所谓再线性化其实就是将扩增的密文再次恢复为二元对从而允许更多的同台乘法操作。而再缩放就是再每次乘法操作时将密文值除以缩放因子将缩放因子由平方项恢复为一次项。

2.2 复现 ckks_example

复现过程主要由两部分组成，第一部分就是下载 SEAL 库并进行编译搭建本地环境，第二部分是利用 cmake 对已经给好的 cpp 文件进行编译链接。

2.2.1 环境搭建

在环境搭建部分，只需要跟随刘哲理老师课本中的内容一步一步操作即可：

- 1、从 github 上 clone 仓库并进行 cmake 操作，将多个 cpp 和 hpp 整合为一个大工程

```
U -- Performing Test CMAKE_HAVE_LIBC_PTHREAD
-- Performing Test CMAKE_HAVE_LIBC_PTHREAD - Success
b -- Found Threads: TRUE
-- SEAL_BUILD_SEAL_C: OFF
-- SEAL_BUILD_EXAMPLES: OFF
-- SEAL_BUILD_TESTS: OFF
-- SEAL_BUILD_BENCH: OFF
-- Configuring done
-- Generating done
-- Build files have been written to: /home/wbf/dataSecurity/lab3/SEAL-main/SEAL
wbf@wbfubuntu:~/dataSecurity/lab3/SEAL-main/SEAL$
```

图 2.3: git clone

- 2、执行 make 命令完成编译

```
llmod.cpp.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
wbf@wbfubuntu:~/dataSecurity/lab3/SEAL-main/SEAL$
```

图 2.4: make 编译

- 3、执行 make 和 make install 完成开源项目的编译和安装

```
-- 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
wbf@wbfubuntu:~/dataSecurity/lab3/SEAL-main/SEAL$
```

图 2.5: 开源项目编译

```
wbf@wbfubuntu:~/dataSecurity/lab3/SEAL-main/SEAL$ sudo make install
[sudo] wbf 的密码:
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...
```

图 2.6: 开源项目安装

2.2.2 代码复现

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、私钥解密等因此，我们的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`

```
1 //构建模块
2 //首先构建 keygenerator，生成公钥、私钥
3 KeyGenerator keygen(context);
4 auto secret_key = keygen.secret_key();
5 PublicKey public_key;
6 keygen.create_public_key(public_key);
7
8 //构建编码器，加密模块、运算器和解密模块
9 //注意加密需要公钥 pk；解密需要私钥 sk；编码器需要 scale
10 Encryptor encryptor(context, public_key);
11 Decryptor decryptor(context, secret_key);
```

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)，可以实现数据的“可算不可见”，因此我们还需要引入密文计算模块 `evaluator`

```
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);
```

最后，加密体系都是基于某一数学困难问题构造的，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），因此我们需要引入：编码器 `encoder` 来实现数字和环上元素的相互转换。

```
1 CKKSEncoder encoder(context);
```

总结下来，整个构建过程为：

- i) 选择 CKKS 参数 `parms`
- ii) 生成 CKKS 框架 `context`
- iii) 构建 CKKS 模块 `keygenerator`、`encoder`、`encryptor`、`evaluator` 和 `decryptor`
- iv) 使用 `encoder` 将数据 `n` 编码为明文 `m`
- v) 使用 `encryptor` 将明文 `m` 加密为密文 `c`
- vi) 使用 `evaluator` 对密文 `c` 运算为密文 `c'`
- vii) 使用 `decryptor` 将密文 `c'` 解密为明文 `m'`
- viii) 使用 `encoder` 将明文 `m'` 解码为数据 `n`

```
1  vector<double> x, y, z;
2  x = { 1.0, 2.0, 3.0 };
3  y = { 2.0, 3.0, 4.0 };
4  z = { 3.0, 4.0, 5.0 };
5  //对向量 x、y、z 进行编码
6  Plaintext xp, yp, zp;
7  encoder.encode(x, scale, xp);
8  encoder.encode(y, scale, yp);
9  encoder.encode(z, scale, zp);
10 //对明文 xp、yp、zp 进行加密
11 Ciphertext xc, yc, zc;
12 encryptor.encrypt(xp, xc);
13 encryptor.encrypt(yp, yc);
14 encryptor.encrypt(zp, zc);
```

本次实验实现的时候同态加密算法最直观的应用-云计算，其基本流程为：

- i) 发送方利用公钥 `pk` 加密明文 `m` 为密文 `c`
- ii) 发送方把密文 `c` 发送到服务器
- iii) 服务器执行密文运算，生成结果密文 `c'`
- iv) 服务器将结果密文 `c'` 发送给接收方
- v) 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。

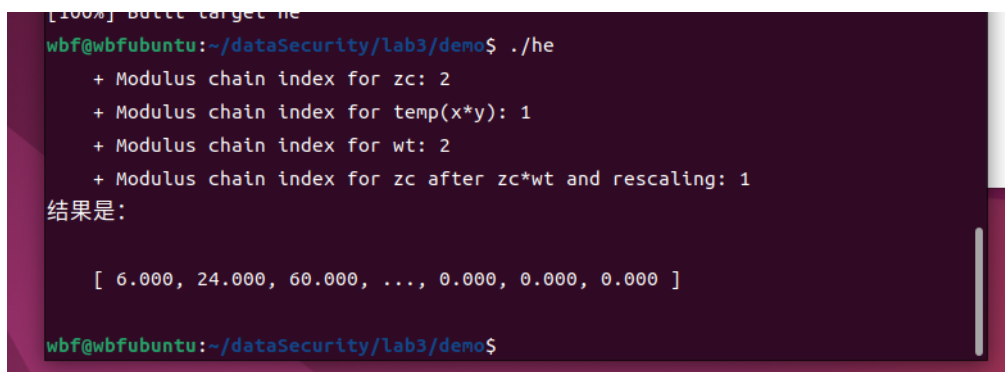
计算过程的代码为：

```

1 //计算  $x*y$ , 密文相乘, 要进行 relinearize 和 rescaling 操作
2 evaluator.multiply(xc,yc,temp);
3 evaluator.relinearize_inplace(temp, relin_keys);
4 evaluator.rescale_to_next_inplace(temp);
5
6 //执行乘法和 rescaling 操作:
7 evaluator.multiply_plain_inplace(zc, wt);
8 evaluator.rescale_to_next_inplace(zc);
9
10 //最后执行  $temp(x*y) * zc(z*1.0)$ 
11 evaluator.multiply_inplace(temp, zc);
12 evaluator.relinearize_inplace(temp, relin_keys);
13 evaluator.rescale_to_next(temp, result_c);

```

接下来使用指令 Cmake make 进行编译链接并执行, 如果程序正确执行可以得到如下结果: 和程序中预测的结果一致, 说明程序编写正确。



```

[100%] Built target he
wbf@wbfubuntu:~/dataSecurity/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 ]

wbf@wbfubuntu:~/dataSecurity/lab3/demo$

```

图 2.7: 查看结果

2.3 代码修改

本次实验的重点是修改代码以实现:

$$x^3 + y * z$$

需要注意的一点是 x 的三次方下降了两个 level, 而 $y*z$ 只是下降了一个 level, 所以我的思路是不如将 yz 的运算改写为 $(y1.0)(z1.0)$, 这样计算结束后 y 与 z 的乘积也下降了两个 level, 可以直接和 x 的三次方做运算。具体代码如下:

```

1 //初始化一个常量
2 Plaintext wt;
3 encoder.encode(1.0, scale, wt);

```

```

4
5 //let the xc and yc `s index becom 5
6 evaluator.multiply_plain_inplace(zc, wt);
7 evaluator.rescale_to_next_inplace(zc);
8 evaluator.multiply_plain_inplace(yc, wt);
9 evaluator.rescale_to_next_inplace(yc);
10
11 //计算 y*z, 密文相乘, 要进行 relinearize 和 rescaling 操作
12 //let temp`s size become 4
13 evaluator.multiply(zc, yc, temp);
14 evaluator.relinearize_inplace(temp, relin_keys);
15 evaluator.rescale_to_next_inplace(temp);

```

而在计算 x 的三次方时的思路是首先使用 xc 与自己相乘, 其次再做一次 xc 乘 1, 然后再将两者相乘得到结果。

```

1 // accomplish  $x**3$  cal
2 Ciphertext temp2;
3 evaluator.multiply(xc, xc, temp2);
4 evaluator.relinearize_inplace(temp2, relin_keys);
5 evaluator.rescale_to_next_inplace(temp2);
6
7 Ciphertext temp3;
8 evaluator.multiply_plain_inplace(xc, wt);
9 evaluator.rescale_to_next_inplace(xc);
10
11 evaluator.multiply(temp2, xc, temp3);
12 evaluator.relinearize_inplace(temp3, relin_keys);
13 evaluator.rescale_to_next_inplace(temp3);

```

然后贴上全部的源代码, 这里要注意的是前面的位置关于 `coeff_modules` 的最大位数与 `poly_modules` 我也做了修改, 改为了教材上提供的 16384 和 `coeff_modules = 60, 40, 40, 40, 40, 40, 40, 60`, 这里也是需要注意的。然后贴上我的源代码:

```

1 #include "examples.h"
2 /* 该文件可以在 SEAL/native/example 目录下找到 */
3 #include <vector>
4 using namespace std;
5 using namespace seal;
6 #define N 3
7 //本例目的: 给定  $x, y, z$  三个数的密文, 让服务器计算  $x**3 + y*z$ 
8

```



```
9  int main(){
10
11  //初始化要计算的原始数据
12  vector<double> x, y, z;
13  x = { 1.0, 2.0, 3.0 };
14  y = { 2.0, 3.0, 4.0 };
15  z = { 3.0, 4.0, 5.0 };
16
17  /******
18  客户端的视角：生成参数、构建环境和生成密文
19  *****/*
20  // (1) 构建参数容器 parms
21  EncryptionParameters parms(scheme_type::ckks);
22  /*CKKS 有三个重要参数：
23  1.poly_module_degree(多项式模数)
24  2.coeff_modulus (参数模数)
25  3.scale (规模) */
26
27  //change here because we need to mul 3 times
28  size_t poly_modulus_degree = 16384;
29  parms.set_poly_modulus_degree(poly_modulus_degree);
30  parms.set_coeff_modulus(CoeffModulus::Create(poly_modulus_degree, { 60, 40, 40, 40, 40, 40, 40, 40 }));
31  //选用 2~40 进行编码
32  double scale = pow(2.0, 40);
33
34  // (2) 用参数生成 CKKS 框架 context
35  SEALContext context(parms);
36
37  // (3) 构建各模块
38  //首先构建 keygenerator, 生成公钥、私钥
39  KeyGenerator keygen(context);
40  auto secret_key = keygen.secret_key();
41  PublicKey public_key;
42  keygen.create_public_key(public_key);
43
44
45  //构建编码器，加密模块、运算器和解密模块
46  //注意加密需要公钥 pk; 解密需要私钥 sk; 编码器需要 scale
47  Encryptor encryptor(context, public_key);
48  Decryptor decryptor(context, secret_key);
49
50  CKKSEncoder encoder(context);
```

```


51
52 //对向量  $x$ 、 $y$ 、 $z$  进行编码
53 Plaintext xp, yp, zp;
54 encoder.encode(x, scale, xp);
55 encoder.encode(y, scale, yp);
56 encoder.encode(z, scale, zp);
57 //对明文  $xp$ 、 $yp$ 、 $zp$  进行加密
58 Ciphertext xc, yc, zc;
59 encryptor.encrypt(xp, xc);
60 encryptor.encrypt(yp, yc);
61 encryptor.encrypt(zp, zc);
62
63 //至此，客户端将  $pk$ 、 $CKKS$  参数发送给服务器，服务器开始运算
64 /*****
65 服务器的视角：生成重线性密钥、构建环境和执行密文计算
66 *****/
67 //生成重线性密钥和构建环境
68 SEALContext context_server(parms);
69 RelinKeys relin_keys;
70 keygen.create_relin_keys(relin_keys);
71 Evaluator evaluator(context_server);
72
73 /* 对密文进行计算，要说明的原则是：
74 -加法可以连续运算，但乘法不能连续运算
75 -密文乘法后要进行 relinearize 操作
76 -执行乘法后要进行 rescaling 操作
77 -进行运算的密文必需执行过相同次数的 rescaling (位于相同 level) */
78 Ciphertext temp;
79 Ciphertext result_c;
80
81 //初始化一个常量
82 Plaintext wt;
83 encoder.encode(1.0, scale, wt);
84
85 //let the xc and yc `s index becom 5
86 evaluator.multiply_plain_inplace(zc, wt);
87 evaluator.rescale_to_next_inplace(zc);
88
89 evaluator.multiply_plain_inplace(yc, wt);
90 evaluator.rescale_to_next_inplace(yc);
91
92

```

```

93 //计算 y*z, 密文相乘, 要进行 relinearize 和 rescaling 操作
94 //let temp`s size become 4
95 evaluator.multiply(zc,yc,temp);
96 evaluator.relinearize_inplace(temp, relin_keys);
97 evaluator.rescale_to_next_inplace(temp);
98
99 // accomplish x**3 cal
100 Ciphertext temp2;
101 evaluator.multiply(xc,xc,temp2);
102 evaluator.relinearize_inplace(temp2, relin_keys);
103 evaluator.rescale_to_next_inplace(temp2);
104
105 Ciphertext temp3;
106 evaluator.multiply_plain_inplace(xc, wt);
107 evaluator.rescale_to_next_inplace(xc);
108
109 evaluator.multiply(temp2,xc,temp3);
110 evaluator.relinearize_inplace(temp3, relin_keys);
111 evaluator.rescale_to_next_inplace(temp3);
112
113
114 //最后执行加法
115 evaluator.add_inplace(temp3, temp);
116 //evaluator.relinearize_inplace(temp3, relin_keys);
117 //evaluator.rescale_to_next(temp3, result_c);
118 result_c=temp3;
119
120 //计算完毕, 服务器把结果发回客户端
121 /*****
122 客户端的视角: 进行解密和解码
123 *****/
124 //客户端进行解密
125 Plaintext result_p;
126 decryptor.decrypt(result_c, result_p);
127 //注意要解码到一个向量上
128 vector<double> result;
129 encoder.decode(result_p, result);
130 //得到结果, 正确的话将输出: {7.000, 20.000, 47.000, ..., 0.000, 0.000, 0.000}
131 cout << " 结果是: " << endl;
132 print_vector(result,3,3);
133 return 0;
134 }

```



```
29 parms.set_coeff_modulus(CoeffModulus::Create(poly_modulus_degree, 2048));
问题 4 输出 调试控制台 终端
[ 50%] Building CXX object CMakeFiles/extend_ckk.dir/extend_ckk.cpp.o
[100%] Linking CXX executable extend_ckk
[100%] Built target extend_ckk
• wbf@wbfubuntu:~/dataSecurity/lab3/extend_demo$ ./extend_ckk
+ Modulus chain index for temp(z*y): 4
here
结果是:
[ 7.000, 20.000, 47.000, ..., -0.000, -0.000, 0.000 ]
○ wbf@wbfubuntu:~/dataSecurity/lab3/extend_demo$
```

图 2.8: 实验结果

3 心得体会

通过本次实验，我第一次接触到了一个新的密码学领域：同态加密以及 SEAL 库和 CKKS 算法，初步了解了格概念带给密码学的改变，理解了基于格的难解性问题。并且动手实践实现了新的对于密文的计算要求，进一步加深了我对 CKKS 算法的理解。

在编写代码的过程中，我进一步理解了再线性化和再缩放两个重要的概念，通过程序调试以及程序报错一步一步的理解 level 的概念，查阅资料了解为什么两个需要计算的密文应该处于同一 level 上，并且也明白了这样的设计带给编程的复杂性

通过本次实验，我对自举操作有了更加深入的理解和体会，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。