

数据安全实验报告

姓名：任意霖 学号：2011897 专业：物联网工程

1 实验名称：SEAL 应用实践（1）

2 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。

3 实验过程

使用 SEAL 同态加密，云提供商永远无法以非加密方式访问他们存储和计算的数据。可以直接对加密的数据执行计算。这种加密计算的结果仍保持加密状态，只能由数据所有者使用机密密钥来解密。本次实验演示的是基于 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。

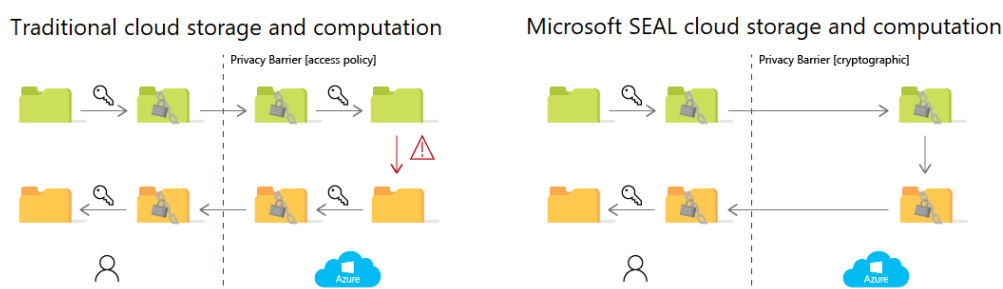


图 3.1: Microsoft 加密算术库 SEAL

3.1 环境配置

SEAL 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，支持基于证书的精确同态运算和基于浮点数的近似同态运算。SEAL 基于 C++ 实现，不需要其他依赖库。

- 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，运行完毕，将在 seal 文件夹下自动建立 SEAL 这个新文件夹。

- 编译和安装

依次输入命令：

```
1 cd SEAL
2 cmake .
3 make
```

该步骤成功后显示如下：

```
-- Performing Test CMAKE_HAVE_LIBC_PTHREAD - Failed
-- Check if compiler accepts -pthread
-- Check if compiler accepts -pthread - yes
-- 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/bigbeauty/seal/SEAL
bigbeauty@bigbeauty-virtual-machine:~/seal/SEAL$
```

图 3.2: SEAL 编译和安装成功实现 1.0

```
1 make
```

```
-- Performing Test CMAKE_HAVE_LIBC_PTHREAD - Failed
-- Check if compiler accepts -pthread
-- Check if compiler accepts -pthread - yes
-- 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/bigbeauty/seal/SEAL
bigbeauty@bigbeauty-virtual-machine:~/seal/SEAL$
```

图 3.3: SEAL 编译和安装成功实现 2.0

```
1 sudo install
```

```
-- Installing: /usr/local/include/SEAL-4.1/seal/util/ntt.h
-- 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
bigbeauty@bigbeauty-virtual-machine:~/seal/SEAL$
```

图 3.4: SEAL 编译和安装成功实现 3.0

3.2 SEAL 应用实践

首先, 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。其次, 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”, 因此还需要引入 evaluator。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), 因此需要引入 encoder。整个构建过程如下所示:

- 选择 CKKS 参数 parms

- 生成 CKKS 框架 context
- 构建 CKKS 模块 keygenerator、encoder、encryptor、evaluator 和 decryptor
- 使用 encoder 将数据 n 编码为明文 m
- 使用 encryptor 将明文 m 加密为密文 c
- 使用 evaluator 对密文 c 运算为密文 c'
- 使用 decryptor 将密文 c' 解密为明文 m'
- 使用 encoder 将明文 m' 解码为数据 n

3.2.1 准备工作

- 设置参数
 - poly-modulus-degree: 多项式模度，为 2 的幂次，值越大计算越慢，但可支持更复杂的加密运算。
 - coeff-modulus: (密文) 系数模量，大整数，素数之积。其长度为其素因子长度之和。值越大，噪声预算越大，加密计算能力越强，但其上限由 poly-modulus-degree 决定；
 - scale 取值任意，这里取了 2 的模数。它决定了明文数据的规模以及乘法计算所消耗的噪声预算。因此，可以尽量取小值来保证效率。
- 创建 SEALcontext
- 构建 keygenerator:Encryptor 只需公钥。
- 生成加密器 encryptor、执行器 evaluator 和解密器 decryptor 密文计算由执行器执行，实际使用中，执行器不会由持有私钥的同一方创建，解密器需要私钥。注意加密需要公钥 pk；解密需要私钥 sk；编码器需要 scale。

其代码具体如下：

```

1  /* CKKS有三个重要参数：
2  1.poly_module_degree(多项式模数)
3  2.coeff_modulus (参数模数)
4  3.scale (规模) */
5  size_t poly_modulus_degree = 16384;
6  parms.set_poly_modulus_degree(poly_modulus_degree);
7  parms.set_coeff_modulus(CoeffModulus::Create(poly_modulus_degree, { 60, 40, 40, 40, 40, 40, 40, 60 }));
8  double scale = pow(2.0, 40);
9
10 // (2) 用参数生成CKKS框架context
11 SEALContext context(parms);
12
13 // (3) 构建各模块
14 // 首先构建keygenerator，生成公钥、私钥
15 KeyGenerator keygen(context);
16 auto secret_key = keygen.secret_key();

```

```

17 PublicKey public_key;
18 keygen.create_public_key(public_key);
19
20 // 构建编码器，加密模块、运算器和解密模块
21 // 注意加密需要公钥pk；解密需要私钥sk；编码器需要scale
22 Encryptor encryptor(context, public_key);
23 Decryptor decryptor(context, secret_key);
24
25 CKKSEncoder encoder(context);

```

这里首先定义了模多项式的次数 N 为 16384，同时设置好了模数链的规模。这里创造了大小分别为 60, 40, 40, 40, 40, 40, 40, 60 比特的素数。其中最后一个素数是用来作为计算辅助密钥的特殊模数 P 的，第二个和第三个代表 p_1 和 p_2 ，第一个则是 q_0 。总共 60 位， q_0 可以看做是小数部分的 40 位精度和为原来整数保留的 20 位规模之和。

3.2.2 数据处理

- 对向量 x 、 y 、 z 进行编码
- 对明文 x_p 、 y_p 、 z_p 进行加密

其代码具体如下：

```

1 Plaintext xp, yp, zp;
2 encoder.encode(x, scale, xp);
3 encoder.encode(y, scale, yp);
4 encoder.encode(z, scale, zp);
5 Ciphertext xc, yc, zc;
6 encryptor.encrypt(xp, xc);
7 encryptor.encrypt(yp, yc);
8 encryptor.encrypt(zp, zc);

```

3.2.3 生成重线性密钥和构建环境

```

1 SEALContext context_server(parms);
2 RelinKeys relin_keys;
3 keygen.create_relin_keys(relin_keys);
4 Evaluator evaluator(context_server);

```

3.2.4 计算过程

为了计算 x^3 我们首先需要计算 x^2 ，然后重线性化。再然后进行重缩放，去掉最后一个中间素数，规模减小了等于被调离的素数的因子（40 位素数）。

```

1 evaluator.square(xc, temp1);
2 evaluator.relinearize_inplace(temp1, relin_keys);
3 evaluator.rescale_to_next_inplace(temp1);

```

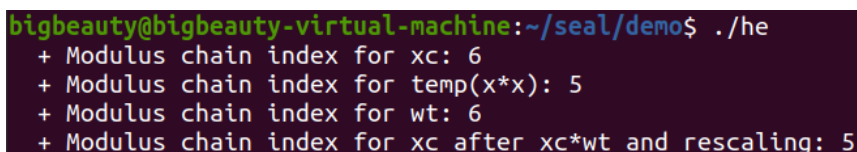
现在由于 $temp1(x^2)$ 与 x 处于不同的层，这使得无法将它们直接相乘来计算。因此可以简单地通过将 x 乘 1，使其切换到模切换链中的下一个参数。同时通过层数输出，展示层数的变化情况。

```

1 Plaintext wt;
2 encoder.encode(1.0, scale, wt);
3 cout<<" + Modulus chain index for xc: "
4     <<context.get_context_data(xc.parms_id())->chain_index()<<endl;
5 cout<<" + Modulus chain index for temp(x*x): "
6     <<context.get_context_data(temp1.parms_id())->chain_index()<<endl;
7 cout<<" + Modulus chain index for wt: "
8     <<context.get_context_data(wt.parms_id())->chain_index()<<endl;
9
10 evaluator.multiply_plain_inplace(xc, wt);
11 evaluator.rescale_to_next_inplace(xc);
12 cout<<" + Modulus chain index for xc after xc*wt and rescaling: "
13     <<context.get_context_data(xc.parms_id())->chain_index()<<endl;

```

其输出结果如下：



```

bigbeauty@bigbeauty-virtual-machine:~/seal/demo$ ./he
+ Modulus chain index for xc: 6
+ Modulus chain index for temp(x*x): 5
+ Modulus chain index for wt: 6
+ Modulus chain index for xc after xc*wt and rescaling: 5

```

图 3.5: 数据的层数变化情况

最后由于， $temp1(x^2)$ 与 x 具有相同的精确规模并使用相同的加密参数，因此可以执行乘法操作。在计算过程中，将结果写入 $temp2$ ，然后重新线性化和重新缩放，从而完成了对 x^3 的计算。

```

1 evaluator.multiply(temp1, xc, temp2);
2 evaluator.relinearize_inplace(temp2, relin_keys);
3 evaluator.rescale_to_next_inplace(temp2);

```

接下来计算 $y \times z$ 的结果，其中需要注意的是，为了保证后续的 $temp3(y \times z)$ 可以与 $temp2 x^3$ 相加，因此需要保证二者在同一层数。因此在执行 $y * z$ 的计算之前，首先通过对 y 和 z 分别乘以 1，从而实现对两者层数的调整，再进行乘法运算，并将结果写入 $temp3$ 。其具体代码如下：

```

1 // change yz, zc index
2 evaluator.multiply_plain_inplace(yz, wt);
3 evaluator.rescale_to_next_inplace(yz);
4 evaluator.multiply_plain_inplace(zc, wt);
5 evaluator.rescale_to_next_inplace(zc);
6
7 // 计算y*z, 密文相乘, 要进行relinearize和rescaling操作
8 evaluator.multiply(zc, yz, temp3);
9 evaluator.relinearize_inplace(temp3, relin_keys);
10 evaluator.rescale_to_next_inplace(temp3);

```

最后完成 $temp2(x^3)$ 和 $temp3(y \times z)$ 的相加部分。为了确保其加密参数可以匹配，因此通过输出对各项层数进行观察和比较，从而确保后续的相加可以完成，其结果如下所示：

```
+ Modulus chain index for yc after yc*wt and rescaling: 5
+ Modulus chain index for zc after zc*wt and rescaling: 5
+ Modulus chain index for temp(y*z): 4
+ Modulus chain index for temp(x*x*x): 4
```

图 3.6: 数据的层数变化情况

从上述图片中可知，所有的密文都相容，可以相加。其具体代码如下所示：

```
1 // add
2 evaluator.add_inplace(temp3, temp2);
3 result_c=temp3;
```

计算完毕后，服务器把结果发回客户端。

3.2.5 解密和解码

其具体代码如下所示：

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

打印结果如下所示：

```
bigbeauty@bigbeauty-virtual-machine:~/seal/demo$ ./he
+ Modulus chain index for xc: 6
+ Modulus chain index for temp(x*x): 5
+ Modulus chain index for wt: 6
+ Modulus chain index for xc after xc*wt and rescaling: 5
+ Modulus chain index for yc after yc*wt and rescaling: 5
+ Modulus chain index for zc after zc*wt and rescaling: 5
+ Modulus chain index for temp(y*z): 4
+ Modulus chain index for temp(x*x*x): 4
Result is:
[ 7.000, 20.000, 47.000, ..., -0.000, -0.000, 0.000 ]
```

图 3.7: 输出结果

4 心得体会

通过本次实验对 CKKS 同态加密算法有了更加深刻的了解。在实验过程中，由于对象之间的层数不一致导致无法进行运算的报错问题，使我更加清楚的了解到同态加密算法的基本原理，并在不断调试中加深了理解。为了方便对层数进行观察，对于每一次的运算也通过输出层数，使得各对象的 level 级别更加的清晰可观。在本次实验中，我感受到前沿密码学还是很深奥很有趣的，并愿意不断学习下去。

附件

CKKS 同态加密完整代码

```
1 #include "examples.h"
2 #include <vector>
3 using namespace std;
4 using namespace seal;
5
6 #define N 3
7
8 int main(){
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
14    // parms
15    EncryptionParameters parms(scheme_type::ckks);
16
17    /* CKKS有三个重要参数:
18    1.poly_module_degree(多项式模数)
19    2.coeff_modulus (参数模数)
20    3.scale (规模) */
21    size_t poly_module_degree = 16384;
22    parms.set_poly_module_degree(poly_module_degree);
23    parms.set_coeff_modulus(CoeffModulus::Create(poly_module_degree, { 60, 40, 40, 40, 40, 40, 40, 60 }));
24    double scale = pow(2.0, 40);
25
26    // (2) 用参数生成CKKS框架context
27    SEALContext context(parms);
28
29    // (3) 构建各模块
30    // 首先构建keygenerator, 生成公钥、私钥
31    KeyGenerator keygen(context);
32    auto secret_key = keygen.secret_key();
33    PublicKey public_key;
34    keygen.create_public_key(public_key);
35
36    // 构建编码器, 加密模块、运算器和解密模块
37    // 注意加密需要公钥pk; 解密需要私钥sk; 编码器需要scale
38    Encryptor encryptor(context, public_key);
39    Decryptor decryptor(context, secret_key);
40
41    CKKSEncoder encoder(context);
42
43    // 对向量x、y、z进行编码
```

```

44 Plaintext xp, yp, zp;
45 encoder.encode(x, scale, xp);
46 encoder.encode(y, scale, yp);
47 encoder.encode(z, scale, zp);
48
49 // 对明文xp、yp、zp进行加密
50 Ciphertext xc, yc, zc;
51 encryptor.encrypt(xp, xc);
52 encryptor.encrypt(yp, yc);
53 encryptor.encrypt(zp, zc);
54
55 // 至此，客户端将pk、CKKS参数发送给服务器，服务器开始运算
56 /*****
57 服务器的视角：生成重线性密钥、构建环境和执行密文计算
58 *****/
59 // 生成重线性密钥和构建环境
60 SEALContext context_server(parms);
61 RelinKeys relin_keys;
62     keygen.create__relin_keys(relin_keys);
63 Evaluator evaluator(context_server);
64
65 Ciphertext temp1;
66 Ciphertext temp2;
67 Ciphertext temp3;
68 Ciphertext result_c;
69
70 // 初始化一个常量
71 Plaintext wt;
72 encoder.encode(1.0, scale, wt);
73
74 // caculate x**3
75 evaluator.multiply(xc, xc, temp1);
76 evaluator.relinearize__inplace(temp1, relin_keys);
77 evaluator.rescale__to__next__inplace(temp1);
78
79 cout<<" + Modulus chain index for xc: "
80     <<context.get_context_data(xc.parms_id())->chain_index()<<endl;
81 cout<<" + Modulus chain index for temp(x*x): "
82     <<context.get_context_data(temp1.parms_id())->chain_index()<<endl;
83 cout<<" + Modulus chain index for wt: "
84     <<context.get_context_data(wt.parms_id())->chain_index()<<endl;
85
86 evaluator.multiply__plain__inplace(xc, wt);
87 evaluator.rescale__to__next__inplace(xc);
88 // ensure xc level
89 cout<<" + Modulus chain index for xc after xc*wt and rescaling: "
90     <<context.get_context_data(xc.parms_id())->chain_index()<<endl;
91
92 // caculate x*x*x

```



```

93     evaluator.multiply(temp1, xc, temp2);
94     evaluator.relinearize_inplace(temp2, relin_keys);
95     evaluator.rescale_to_next_inplace(temp2);
96
97     // change yz, zc index
98     evaluator.multiply_plain_inplace(yc, wt);
99     evaluator.rescale_to_next_inplace(yc);
100    evaluator.multiply_plain_inplace(zc, wt);
101    evaluator.rescale_to_next_inplace(zc);
102
103    // 计算y*z, 密文相乘, 要进行relinearize和rescaling操作
104    cout<<" + Modulus chain index for yc after yc*wt and rescaling: "
105         <<context.get_context_data(yc.parms_id())->chain_index()<<endl;
106    cout<<" + Modulus chain index for zc after zc*wt and rescaling: "
107         <<context.get_context_data(zc.parms_id())->chain_index()<<endl;
108
109    evaluator.multiply(zc,yc,temp3);
110    evaluator.relinearize_inplace(temp3, relin_keys);
111    evaluator.rescale_to_next_inplace(temp3);
112
113    // add
114    cout<<" + Modulus chain index for temp(y*z): "
115         <<context.get_context_data(temp3.parms_id())->chain_index()<<endl;
116    cout<<" + Modulus chain index for temp(x*x*x): "
117         <<context.get_context_data(temp2.parms_id())->chain_index()<<endl;
118    evaluator.add_inplace(temp3, temp2);
119    result_c=temp3;
120
121    //计算完毕, 服务器把结果发回客户端
122    /*****
123    客户端的视角: 进行解密和解码
124    *****/
125    //客户端进行解密
126    Plaintext result_p;
127    decryptor.decrypt(result_c, result_p);
128
129    vector<double> result;
130    encoder.decode(result_p, result);
131    cout << "Result is: " << endl;
132    print_vector(result,3,3);
133    return 0;
134 }

```

CMAKELISTS 文件完整代码

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