

《数据安全》实验报告

姓名： 费泽锟 学号： 2014074 班级： 信安班

实验名称：

全同态加密 SEAL 应用实践

实验要求：

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，需要根据计算 $x * y * z$ 的实现代码改编为在云服务器实现求取 $x^3 + y * z$ 的密文计算结果的过程。

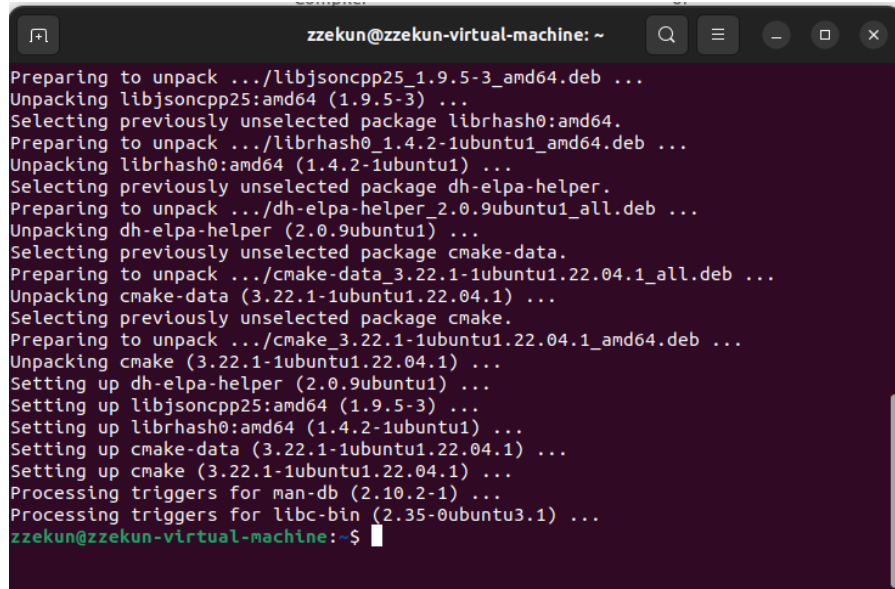
实验过程：

1. 配置实验环境

在进行实验之前，我们首先需要对实验环境进行配置，我们使用的是 Ubuntu22.04 版本，之前已经安装过了 g++编译器，我们只需要继续安装好 cmake 工具即可。使用命令如下：

```
sudo apt install cmake
```

成功安装后结果如下图所示：



```
zzekun@zzekun-virtual-machine: ~  
Preparing to unpack .../libjsoncpp25_1.9.5-3_amd64.deb ...  
Unpacking libjsoncpp25:amd64 (1.9.5-3) ...  
Selecting previously unselected package librhash0:amd64.  
Preparing to unpack .../librhash0_1.4.2-1ubuntu1_amd64.deb ...  
Unpacking librhash0:amd64 (1.4.2-1ubuntu1) ...  
Selecting previously unselected package dh-elpa-helper.  
Preparing to unpack .../dh-elpa-helper_2.0.9ubuntu1_all.deb ...  
Unpacking dh-elpa-helper (2.0.9ubuntu1) ...  
Selecting previously unselected package cmake-data.  
Preparing to unpack .../cmake-data_3.22.1-1ubuntu1.22.04.1_all.deb ...  
Unpacking cmake-data (3.22.1-1ubuntu1.22.04.1) ...  
Selecting previously unselected package cmake.  
Preparing to unpack .../cmake_3.22.1-1ubuntu1.22.04.1_amd64.deb ...  
Unpacking cmake (3.22.1-1ubuntu1.22.04.1) ...  
Setting up dh-elpa-helper (2.0.9ubuntu1) ...  
Setting up libjsoncpp25:amd64 (1.9.5-3) ...  
Setting up librhash0:amd64 (1.4.2-1ubuntu1) ...  
Setting up cmake-data (3.22.1-1ubuntu1.22.04.1) ...  
Setting up cmake (3.22.1-1ubuntu1.22.04.1) ...  
Processing triggers for man-db (2.10.2-1) ...  
Processing triggers for libc-bin (2.35-0ubuntu3.1) ...  
zzekun@zzekun-virtual-machine:~$
```

2. 编译安装 SEAL 库

根据教材中的实验指导，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。接着通过 cmake 工具对 SEAL 库进行编译与路径安装。如果在虚拟机之中实在是通过不了 NAT 去访问 GitHub，也可以在本地先 clone 仓库然后通过共享文件夹将其传输进虚拟机之中。

```
zzekun@zzekun-virtual-machine: ~/seal/SEAL
-- x86intrin.h - found
-- SEAL_USE_INTRIN: ON
-- Performing Test SEAL_MEMSET_S_FOUND
-- Performing Test SEAL_MEMSET_S_FOUND - Failed
-- Looking for explicit_bzero
-- Looking for explicit_bzero - found
-- Looking for explicit_memset
-- Looking for explicit_memset - not found
-- SEAL_USE_MEMSET_S: OFF
-- SEAL_USE_EXPLICIT_BZERO: ON
-- SEAL_USE_EXPLICIT_MEMSET: OFF
-- Looking for pthread.h
-- Looking for pthread.h - found
-- Performing Test CMAKE_HAVE_LIBC_PTHREAD
-- Performing Test CMAKE_HAVE_LIBC_PTHREAD - Success
-- 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/zzekun/seal/SEAL
zzekun@zzekun-virtual-machine:~/seal/SEAL$
```

上图显示 cmake 编译链接已经成功，可以进行后续的 make 指令以及 make install 指令的安装过程，接着运行 make 指令与 make install 指令进行安装。

```
.0
[ 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
zzekun@zzekun-virtual-machine:~/seal/SEAL$
```

上图显示 make 命令该步骤成功，最后执行 sudo make install 命令。

```
-- 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
zzekun@zzekun-virtual-machine:~/seal/SEAL$
```

最后显示安装成功，那么至此 SEAL 库的编译与安装就已经完成了。

~~~~~

### 3. 简单测试 SEAL 与 CMakeLists 的正确使用

```
-- Configuring done
-- Generating done
-- Build files have been written to: /home/zzekun/seal/demo
zzekun@zzekun-virtual-machine:~/seal/demo$ make
[ 50%] Building CXX object CMakeFiles/test.dir/test.cpp.o
[100%] Linking CXX executable test
[100%] Built target test
zzekun@zzekun-virtual-machine:~/seal/demo$ ./test
hellow world
zzekun@zzekun-virtual-machine:~/seal/demo$
```

我们分别对测试代码和 CMakeLists.txt 文件之中的信息进行简单的分析，上图为简单样

例测试成功的显示。

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

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

在 CMakeLists.txt 文件之中规定了 cmake 工具的版本要求，要进行编译链接的项目名，确定编译的源文件以及生成的可执行文件名以及编译时需要的库文件。

~~~~~

4. 应用实例复现改编

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

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

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来实现数字和环上元素的相互转换：

- 编码器

总结下来，整个构建过程为：

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`

接着我们需要根据教材之中给出的 $x * y * z$ 的例子进行适当改编, 将其修改为能够进行 $x^3 + y * z$ 的云服务器的计算。我们首先熟悉计算 $x * y * z$ 的流程, 最终能够成功地得到结果:

```
zzekun@zzekun-virtual-machine:~/seal/demo$ cmake .
-- Microsoft SEAL -> Version 4.1.1 detected
-- Microsoft SEAL -> Targets available: SEAL::seal
-- Configuring done
-- Generating done
-- Build files have been written to: /home/zzekun/seal/demo
zzekun@zzekun-virtual-machine:~/seal/demo$ make
[ 50%] Building CXX object CMakeFiles/he.dir/ckks_example.cpp.o
[100%] Linking CXX executable he
[100%] Built target he
zzekun@zzekun-virtual-machine:~/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 ]
zzekun@zzekun-virtual-machine:~/seal/demo$
```

接着我们需要根据实验的主体流程进行分析与改编:

本次实验实现的是同态加密算法最直观的应用——云计算, 其基本流程为:

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。

接着我们通过对源代码的改进, 对源代码和实验流程进行详细的分析与实现:

1. 对参数部分的修改

在 SEAL 框架之中, 我们对乘法运算必须要保持运算的数据均在同一层才能够预算, 所以我们需要对 $x^3 + y * z$ 运算进行适当的修改, 可以修改为 $(x * x) * (x * 1.0) + (y * z)$

1.0) * (z * 1.0)的形式，如果修改为该种形式的情形，那么我们也就会进行两层的乘法运算，所以其实保持原有的参数即可：

```
Poly_module_degree = 8196; coeff_modulus={60,40,40,60};scale = 2^40
```

原有的参数就可以保持进行两层的乘法运算。但是我们也可以使用参数形式如下：

```
Poly_module_degree = 8196; coeff_modulus={50,30,30,30.50};scale = 2^30
```

上述的参数形式能够维持三层的乘法运算，实验之中我们采用的是第二种参数形式，虽然这样会降低密文规模。

2. 源代码分析与修改

我们接下来对源代码进行分析与修改：

```
home > zzekun > seal > demo > G+ ckks_example.cpp
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*y*z
8
9  int main() {
10     //初始化要计算的原始数据
11     vector<double> x, y, z;
12     x = { 1.0, 2.0, 3.0 };
13     y = { 2.0, 3.0, 4.0 };
14     z = { 3.0, 4.0, 5.0 };
15 }
```

首先我们需要使用 seal 的命名空间，并且因为计算过程之中只有 3 个数的密文，所以需要定义好 N 为 3，在 seal 框架之中数据的运算可以是 vector 容器类的运算。

```
19 // (1) 构建参数容器 parms
20 EncryptionParameters parms(scheme_type::ckks);
21 /*CKKS有三个重要参数：
22 1.poly_module_degree(多项式模数)
23 2.coeff_modulus (参数模数)
24 3.scale (规模) */
25
26 size_t poly_modulus_degree = 8192;
27 parms.set_poly_modulus_degree(poly_modulus_degree);
28 // 这个参数调整之后正好可以进行3层运算
29 parms.set_coeff_modulus(CoeffModulus::Create(poly_modulus_degree, { 50, 30, 30, 30, 50 }));
30 //选用2^40进行编码，修改为2^30
31 double scale = pow(2.0, 30);
```

接下来就是对 CKKS 算法的参数进行设置，其中：

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. 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

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。

接下来我们需要生成 CKKS 框架 `context`，然后去生成基于格的公钥与私钥，并对 `vector` 的浮点数进行编码与加密的操作，其中编码的目的在于将明文编码进向量空间。

```
33 // (2) 用参数生成CKKS框架context
34 SEALContext context(parms);
35
36 // (3) 构建各模块
37 //首先构建keygenerator，生成公钥、私钥
38 KeyGenerator keygen(context);
39 auto secret_key = keygen.secret_key();
40 PublicKey public_key;
41 keygen.create_public_key(public_key);
42
43 //构建编码器，加密模块、运算器和解密模块
44 //注意加密需要公钥pk；解密需要私钥sk；编码器需要scale
45 Encryptor encryptor(context, public_key);
46 Decryptor decryptor(context, secret_key);
47
48 CKKSEncoder encoder(context);
49 //对向量x、y、z进行编码
50 Plaintext xp, yp, zp;
51 encoder.encode(x, scale, xp);
52 encoder.encode(y, scale, yp);
53 encoder.encode(z, scale, zp);
54 //对明文xp、yp、zp进行加密
55 Ciphertext xc, yc, zc;
56 encryptor.encrypt(xp, xc);
57 encryptor.encrypt(yp, yc);
58 encryptor.encrypt(zp, zc);
59
```

我们还需要生成重线性密钥并且进一步构建环境和运算器 `evaluator`，其中重线性过程需要重线性密钥来进行配合运算。

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

我们已经构建好了参数、密钥以及进行密文计算的环境，那么接下来我们就只需要设计好合理的计算流程就能够成功地进行同态计算了。我们需要获得的计算流程为 $x^3 + y * z$ ，

我们首先可以进行 $x * x$ 的计算，这一步计算为同层计算可以直接正常的进行，但是进而进行 $(x * x) * x$ 的计算时，此时的 x^2 的计算结果与 x 不处于同一层之中，需要进行再线性化与再缩放，此时我们只需要将 x 转化为 $x * 1.0$ ，即可将这两部分数据转化为同层次的数据进行计算。

在这一部分之中，我们需要注意的一点是：我们需要先进行 $x * x$ 的计算，因为如果我们先进行 $x * 1.0$ 的计算并且不使用 `temp` 数据对其结果进行存储的话，可能会导致 x 原本处于的运算层数的降低，具体的计算过程如下：

```

84 // 使得x和x^2在同一层
85 Plaintext wt;
86 encoder.encode(1.0, scale, wt);
87 //此时，我们可以查看框架中不同数据的层级：
88 cout << "    + Modulus chain index for xc: "
89 |<< context_server.get_context_data(xc.parms_id())->chain_index() << endl;
90 cout << "    + Modulus chain index for temp(x*x): "
91 |<< context_server.get_context_data(temp.parms_id())->chain_index() << endl;
92 cout << "    + Modulus chain index for wt: "
93 |<< context_server.get_context_data(wt.parms_id())->chain_index() << endl;
94
95 //执行乘法和rescaling操作：
96 evaluator.multiply_plain_inplace(xc, wt);
97 evaluator.rescale_to_next_inplace(xc);
98
99 //为了保证xc在计算之中的层级，需要先在xc最高层的时候进行xc*xc计算，后续的xc*1.0的计算会降低xc的层级
100 //再次查看xc的层级，可以发现xc与temp层级变得相同
101 cout << "    + Modulus chain index for xc after xc*wt and rescaling: "
102 |<< context_server.get_context_data(xc.parms_id())->chain_index() << endl;
103
104 //最后执行temp (x*x) * xc (x*1.0)
105 evaluator.multiply_inplace(temp, xc);
106 evaluator.relinearize_inplace(temp, relin_keys);
107 evaluator.rescale_to_next_inplace(temp);
108 // evaluator.rescale_to_next(temp, result_c);

```

我们在第一步之中已经完成了第一部分的运算，接下来需要设计的就是第二部分也就是 $y * z$ 部分的计算，这里我们的变换策略为对 y 和 z 数据均去乘 1.0 ，降低层数，这时 $y * z$ 运算的结果与 $x * x * x$ 计算结果的层数就相同了，接着就可以直接进行加法运算。

```

110 //继续进行x^3+y*z的计算
111 //先把y变换成y*1.0, z变换成z*1.0
112 evaluator.multiply_plain_inplace(y, wt);
113 evaluator.rescale_to_next_inplace(y);
114
115 evaluator.multiply_plain_inplace(z, wt);
116 evaluator.rescale_to_next_inplace(z);
117
118 cout << "    + Modulus chain index for yc after yc*wt and rescaling: "
119 |<< context_server.get_context_data(yc.parms_id())->chain_index() << endl;
120 cout << "    + Modulus chain index for zc after zc*wt and rescaling: "
121 |<< context_server.get_context_data(zc.parms_id())->chain_index() << endl;
122
123 Ciphertext temp1;
124 evaluator.multiply(y, z, temp1);
125 evaluator.relinearize_inplace(temp1, relin_keys);
126 evaluator.rescale_to_next_inplace(temp1);
127
128 //最后执行temp (x*x*x) + temp1
129 // evaluator.multiply_inplace(temp, xc);
130 evaluator.add_inplace(temp, temp1);
131 // evaluator.relinearize_inplace(temp, relin_keys);
132 // evaluator.rescale_to_next_inplace(temp);
133 // evaluator.rescale_to_next(temp, result_c);
134 result_c = temp;

```


在得到了密文的计算结果之后,在客户端只需要进行解密已经从明文域的解码操作就能够成功地获得最终的明文运算结果了。

```
139 //计算完毕,服务器把结果发回客户端
140 /*****
141 客户端的视角:进行解密和解码
142 *****/
143 //客户端进行解密
144 Plaintext result_p;
145 decryptor.decrypt(result_c, result_p);
146 //注意要解码到一个向量上
147 vector<double> result;
148 encoder.decode(result_p, result);
149 //得到结果,正确的话将输出:{6.000, 24.000, 60.000, ..., 0.000, 0.000, 0.000}
150 cout << "结果是:" << endl;
151 print_vector(result, 3, 3);
152 return 0;
```

最终我们能够成功地得到最后的结果,也就是 $x^3 + y * z$ 的计算结果,得到的最后结果为:
[7.000, 20.000, 47.000],但是对于 vector 的最后部分出现了-0.000 的原因仍需继续研究。

```
zzekun@zzekun-virtual-machine:~/seal/demo$ cmake .
-- Microsoft SEAL -> Version 4.1.1 detected
-- Microsoft SEAL -> Targets available: SEAL::seal
-- Configuring done
-- Generating done
-- Build files have been written to: /home/zzekun/seal/demo
zzekun@zzekun-virtual-machine:~/seal/demo$ make
Consolidate compiler generated dependencies of target he
[100%] Built target he
zzekun@zzekun-virtual-machine:~/seal/demo$ ./he
+ Modulus chain index for xc: 3
+ Modulus chain index for temp(x*x): 2
+ Modulus chain index for wt: 3
+ Modulus chain index for xc after xc*wt and rescaling: 2
+ Modulus chain index for yc after yc*wt and rescaling: 2
+ Modulus chain index for zc after zc*wt and rescaling: 2
结果是:
[ 7.000, 20.000, 47.000, ..., 0.000, -0.000, -0.000 ]
zzekun@zzekun-virtual-machine:~/seal/demo$
```

心得体会

通过本次实验,我们学习了全同态加密算法的原理,实验中给出了一个非常贴近现实的使用 CKKS 的场景,我们完成了最终密文计算,明文解密的过程。

经过学习与实践,将在密码学课上学习的知识和数据安全学习的内容结合在了一起,更好的理解了 CKKS 算法和 SEAL 框架的特性及其适合的应用场景。