数据安全实验报告

Lab3 SEAL 应用实践
网络空间安全学院 信息安全专业
2112492 刘修铭 1036

1 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。

2 实验原理

2.1 开发框架 SEAL

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。

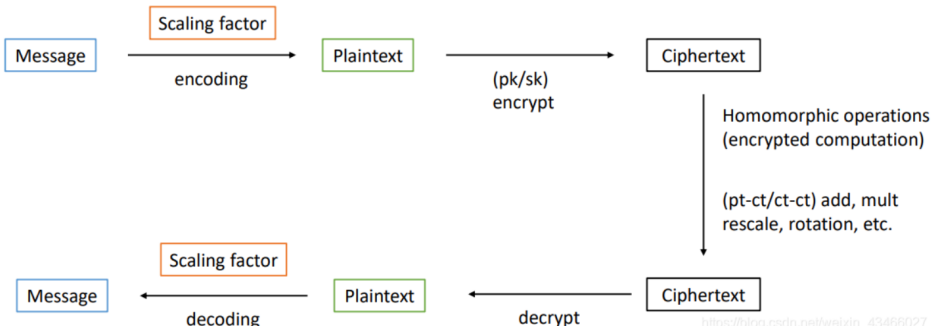
SEAL 基于 C++ 实现，不需要其他依赖库。

2.2 CKKS 算法

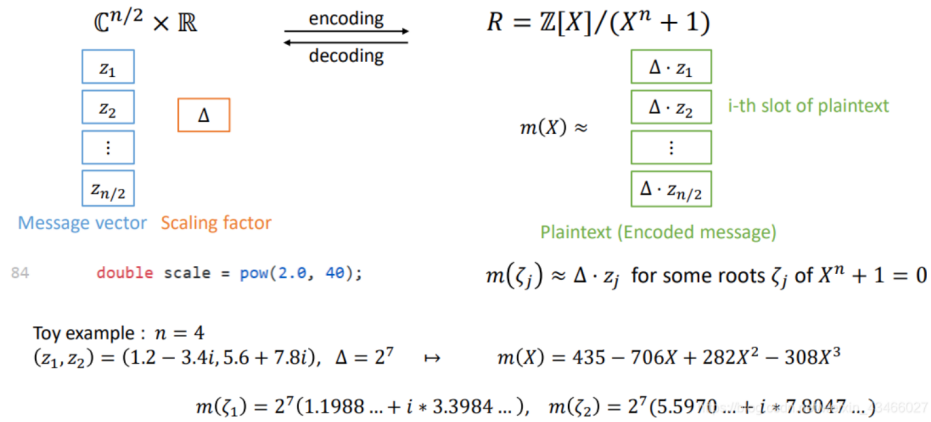
CKKS 是 2017 年提出的同态加密方案。它支持浮点向量在密文空间的加减乘运算并保持同态，但是只支持有限次乘法的运算。

如图是 CKKS 的一个大概流程。先对消息（向量）进行编码，然后再加密，在密文空间进行一些运算后再解密，最后解码成运算后的消息（向量）。

注意，这里的编码指的是将复数向量映射成为多项式，是为了方便下面进一步的加密，



如图，这是 CKKS 编、解码的过程。



对密文进行计算时，需要遵守如下原则：

- 加法可以连续运算，但惩罚不能连续运算
- 密文乘法后要进行 relinearize 操作
- 执行乘法后要进行 rescaling 操作
- 进行运算的密钥必须执行过相同次数的 rescaling，即位于相同 level

因此，在每次进行运算前，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 只能降低而无法升高，所以需要小心设计计算的先后顺序。

3 实验过程（含主要源代码）

3.1 实验环境配置

运行命令 `git clone https://github.com/microsoft/SEAL`，克隆加密库资源。

```
lxm@lxmliu2002:~$ git clone https://github.com/microsoft/SEAL
Cloning into 'SEAL'...
remote: Enumerating objects: 17111, done.
remote: Counting objects: 100% (17111/17111), done.
remote: Compressing objects: 100% (3939/3939), done.
remote: Total 17111 (delta 13019), reused 16913 (delta 12958), pack-reused 0
Receiving objects: 100% (17111/17111), 4.91 MiB | 3.20 MiB/s, done.
Resolving deltas: 100% (13019/13019), done.
```

接着运行 `cmake`，进行编译环境的配置。

> 此处本人一次成功，但舍友一直失败，经过探索，确定是由于网络问题导致，需要科学上网。而虚拟机对于网络的配置较为玄学，WSL 的优势得以显现。

```
-- SEAL_USE_MEMSET_S: OFF
-- SEAL_USE_EXPLICIT_BZERO: ON
-- SEAL_USE_EXPLICIT_MEMSET: OFF
-- Performing Test CMAKE_HAVE_LIBC_PTHREAD
-- 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/lxm/SEAL
```

运行 `make` 进行编译

```
[ 83%] Building CXX object CMakeFiles/seal.dir/native/src/se
[ 84%] Building CXX object CMakeFiles/seal.dir/native/src/se
[ 86%] Building CXX object CMakeFiles/seal.dir/native/src/se
[ 87%] Building CXX object CMakeFiles/seal.dir/native/src/se
[ 88%] Building CXX object CMakeFiles/seal.dir/native/src/se
[ 89%] Building CXX object CMakeFiles/seal.dir/native/src/se
[ 89%] Building CXX object CMakeFiles/seal.dir/native/src/se
[ 91%] Building CXX object CMakeFiles/seal.dir/native/src/se
[ 92%] Building CXX object CMakeFiles/seal.dir/native/src/se
[ 93%] Building CXX object CMakeFiles/seal.dir/native/src/se
[ 94%] Building CXX object CMakeFiles/seal.dir/native/src/se
[ 96%] Building CXX object CMakeFiles/seal.dir/native/src/se
[ 97%] Building CXX object CMakeFiles/seal.dir/native/src/se
[ 98%] Building CXX object CMakeFiles/seal.dir/native/src/se
[100%] Linking CXX static library lib/libseal-4.1.a
[100%] Built target seal
```

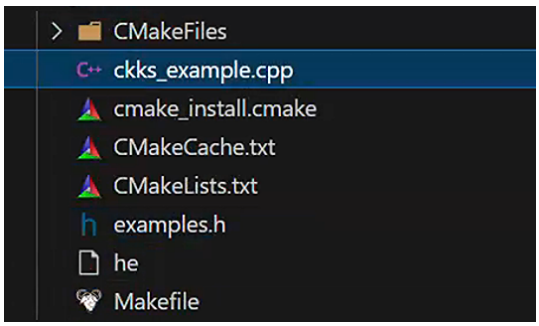
最后，`sudo make install`，复制相关文件到指定文件夹中

```
-- Installing: /usr/local/include/SEAL-4.1/seal/util/polyarithsmallmod.h
-- Installing: /usr/local/include/SEAL-4.1/seal/util/polycore.h
-- Installing: /usr/local/include/SEAL-4.1/seal/util/rlwe.h
-- Installing: /usr/local/include/SEAL-4.1/seal/util/rns.h
-- Installing: /usr/local/include/SEAL-4.1/seal/util/scalingvariant.h
-- 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
lxm@lxmliu2002:~/SEAL$
```

按照实验手册说明，创建 `demo` 文件夹，并写入 `test.cpp` 和 `CMakeLists.txt` 文件，以进行安装测试。编写完成后，在控制台依次运行 `cmake .`、`make` 和 `./test` 进行测试。

3.2 CKKS 应用示例

按照实验手册说明，将对应的代码复制到对应的文件夹中。



将 `example.h` 复制到对应文件夹，并修改 `CMakeLists.txt` 文件。

```
cmake_minimum_required(VERSION 3.10)
project(demo)
add_executable(he ckks_example.cpp)
add_compile_options(-std=c++17)
find_package(SEAL)
target_link_libraries(he SEAL::seal)
```

接着打开控制台，依次运行 `cmake .`、 `make` 和 `./he`，对项目进行编译并运行。

```
lxm@lxmliu2002:~/seal/test$ 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/lxm/seal/test
lxm@lxmliu2002:~/seal/test$ make
[100%] Built target he
```

3.3 CKKS 改写

该部分参考教材实验 2.3 完成。

首先复制 `example.h` 文件到该文件夹目录下，用于后续编程。下面对 CKKS 代码进行改写。

- 设定好要进行计算的数据，初始化原始向量

```
// 客户端的视角：要进行计算的数据
vector<double> x, y, z;
x = {1.0, 2.0, 3.0};
y = {2.0, 3.0, 4.0};
z = {3.0, 4.0, 5.0};
cout << "原始向量x是：" << endl;
print_vector(x);
cout << "原始向量y是：" << endl;
print_vector(y);
cout << "原始向量z是：" << endl;
print_vector(z);
cout << endl;
```

- 接着对一些参数进行设置，本次实验中，均按照官方建议进行参数设置。从示例代码中得知，CKKS 有三个重要参数：poly_module_degree(多项式模数)、coeff_modulus（参数模数）和 scale（规模）。

- 多项式模数的度数（`poly_modulus_degree`）：可以提供足够的加密强度，同时又能够保持较高的性能。
- 系数模数（`coeff_modulus`）：系数模数的选择对于 CKKS 方案的性能和安全性至关重要。其中，60 位和 40 位的系数模数用于提供较高的加密强度，而 40 位的系数模数则用于提供较高的计算效率。这样的组合可以在保证加密安全性的前提下，尽可能地提高计算的效率。
- 缩放参数（`scale`）：缩放参数决定了加密结果的范围，从而影响了计算的精度和安全性。在这个例子中，选择了 `pow(2.0, 40)` 作为缩放参数，可以提供较高的精度和安全性。

```
// 构建参数容器 parms
EncryptionParameters parms(scheme_type::ckks);
// 这里的参数都使用官方建议的
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}));
double scale = pow(2.0, 40);

// 用参数生成 CKKS 框架 context
SEALContext context(parms);
```

- 生成 CKKS 框架，对各个模块进行构建。

```
// 构建各模块
// 生成公钥、私钥和重线性化密钥
KeyGenerator keygen(context);
auto secret_key = keygen.secret_key();
PublicKey public_key;
keygen.create_public_key(public_key);
RelinKeys relin_keys;
keygen.create_relin_keys(relin_keys);
// 构建编码器，加密模块、运算器和解密模块
// 注意加密需要公钥 pk；解密需要私钥 sk；编码器需要 scale
Encryptor encryptor(context, public_key);
Evaluator evaluator(context);
Decryptor decryptor(context, secret_key);
CKKSEncoder encoder(context);
```

- 接着使用编码器对向量进行编码，使用加密模块对向量进行加密。

```
// 对向量 x、y、z 进行编码
Plaintext xp, yp, zp;
encoder.encode(x, scale, xp);
encoder.encode(y, scale, yp);
encoder.encode(z, scale, zp);

// 对明文 xp、yp、zp 进行加密
Ciphertext xc, yc, zc;
encryptor.encrypt(xp, xc);
encryptor.encrypt(yp, yc);
encryptor.encrypt(zp, zc);
```

- 下面进入本次实验核心部分

1. 计算 x^2，将 $xc \times xc$ 的结果存入 $x2$。

```
print_line(__LINE__);
cout << "计算 x^2 ." << endl;
Ciphertext x2;
evaluator.multiply(xc, xc, x2);
// 进行 relinearize 和 rescaling 操作
evaluator.relinearize_inplace(x2, relin_keys);
evaluator.rescale_to_next_inplace(x2);
// 然后查看一下此时x^2结果的level
print_line(__LINE__);
cout << " + Modulus chain index for x2: "
    << context.get_context_data(x2.parms_id())->chain_index() << endl;
```

2. 计算 $1.0 \times x$，将 x 的 level 与 x^2 的 level 保持一致，使后续计算能够进行。

```
// 此时xc本身的层级应该是2，比x^2高，因此这一步解决层级问题
print_line(__LINE__);
cout << " + Modulus chain index for xc: "
     << context.get_context_data(xc.parms_id())->chain_index() << endl;
// 因此，需要对 x 进行一次乘法和 rescaling操作
print_line(__LINE__);
cout << "计算 1.0*x ." << endl;
Plaintext plain_one;
encoder.encode(1.0, scale, plain_one);
// 执行乘法和 rescaling 操作：
evaluator.multiply_plain_inplace(xc, plain_one);
evaluator.rescale_to_next_inplace(xc);
// 再次查看 xc 的层级，可以发现 xc 与 x^2 层级变得相同
print_line(__LINE__);
cout << " + Modulus chain index for xc new: "
     << context.get_context_data(xc.parms_id())->chain_index() << endl;
// 那么，此时xc与x^2层级相同，二者可以相乘了
```

3. 计算 x^3 ，即 $1 \times x \times x^2$

```
// 先设置新的变量叫x3
print_line(__LINE__);
cout << "计算 1.0*x*x^2 ." << endl;
Ciphertext x3;
evaluator.multiply_inplace(x2, xc);
evaluator.relinearize_inplace(x2, relin_keys);
evaluator.rescale_to_next(x2, x3);
// 此时观察x^3的层级
print_line(__LINE__);
cout << " + Modulus chain index for x3: "
     << context.get_context_data(x3.parms_id())->chain_index() << endl;
```

4. 计算 $y \times z$

```
print_line(__LINE__);
cout << "计算 y*z ." << endl;
Ciphertext yz;
evaluator.multiply(yc, zc, yz);
// 进行 relinearize 和 rescaling 操作
evaluator.relinearize_inplace(yz, relin_keys);
evaluator.rescale_to_next_inplace(yz);
// 然后查看一下此时y*z结果的level
print_line(__LINE__);
cout << " + Modulus chain index for yz: "
     << context.get_context_data(yz.parms_id())->chain_index() << endl;
```

5. 计算 $x^3 + y \times z$

(a) 完全前面的计算后，现有的两个待求和的变量的 level 和 scales 都不统一。在此，模仿给定样例进行重制。

```
// 注意，此时问题在于scales的不统一，可以直接重制。
print_line(__LINE__);
cout << "Normalize scales to 2^40." << endl;
x3.scale() = pow(2.0, 40);
yz.scale() = pow(2.0, 40);
// 输出观察，此时的scale的大小已经统一了！
print_line(__LINE__);
cout << " + Exact scale in 1*x^3: " << x3.scale() << endl;
print_line(__LINE__);
cout << " + Exact scale in  y*z: " << yz.scale() << endl;

// 但是，此时还有一个问题，就是我们的x^3和yz的层级还不统一！
// 在官方 examples 中，给出了一个简便的变换层级的方法，如下所示：
parms_id_type last_parms_id = x3.parms_id();
evaluator.mod_switch_to_inplace(yz, last_parms_id);
print_line(__LINE__);
cout << " + Modulus chain index for yz new: "
    << context.get_context_data(yz.parms_id())->chain_index() << endl;
```

(b) 处理后，进行求和运算即可

```
print_line(__LINE__);
cout << "计算 x^3+y*z ." << endl;
Ciphertext encrypted_result;
evaluator.add(x3, yz, encrypted_result);
```

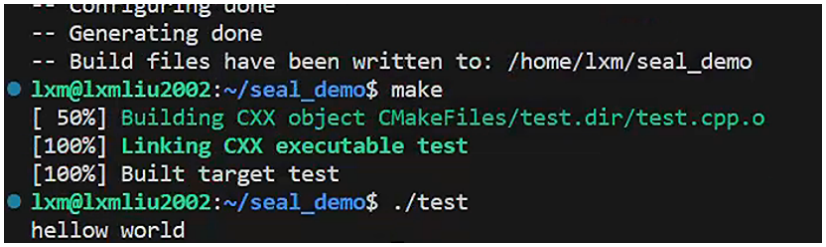
6. 客户端解码

```cpp
// 计算完毕，服务器把结果发回客户端
Plaintext result_p;
decryptor.decrypt(encrypted_result, result_p);

// 注意要解码到一个向量上
vector<double> result;
encoder.decode(result_p, result);

// 输出结果
print_line(__LINE__);
cout << "结果是: " << endl;
print_vector(result, 3 /*precision*/);
```

4 实验结果及分析

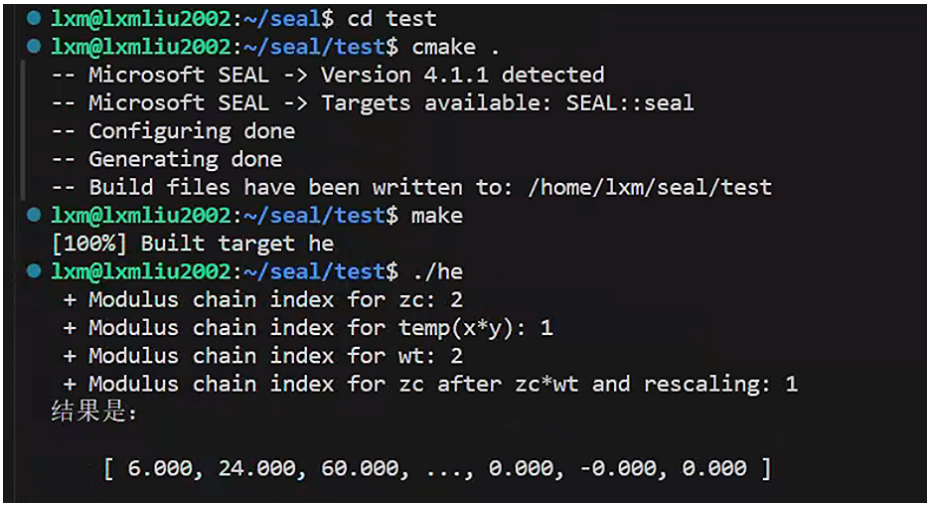
4.1 实验环境配置

在测试程序中，得到如下结果。可以看到，成功输出 hwllow world，说明环境配置成功。



4.2 CKKS 应用示例

CKKS 应用示例部分，编译并运行后，得到如下结果。



可以确认，$6 = 1 \times 2 \times 3$，$24 = 2 \times 3 \times 4$，$60 = 3 \times 4 \times 5$，说明程序逻辑正确，运行成功。

4.3 CKKS 改写

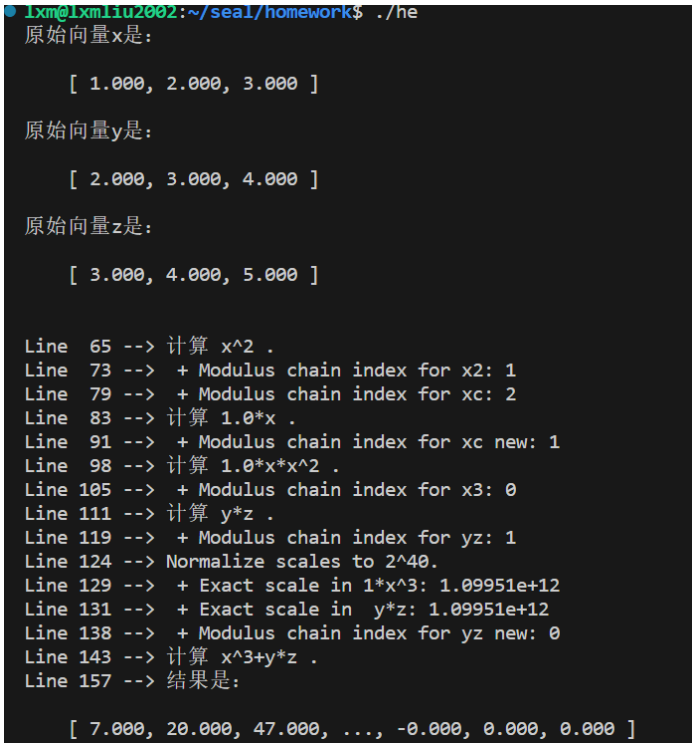
完成代码的编写后，修改 `CMakeLists.txt` 文件。

```
cmake_minimum_required(VERSION 3.10)
project(demo)
add_executable(he homework.cpp)
add_compile_options(-std=c++17)
find_package(SEAL)
target_link_libraries(he SEAL::seal)
```

然后运行 `cmake .` 和 `make`，对程序进行编译。

```
lxm@lxmliu2002:~/seal/homework$ cmake .
-- The C compiler identification is GNU 9.4.0
-- The CXX compiler identification is GNU 9.4.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Performing Test CMAKE_HAVE_LIBC_PTHREAD
-- Performing Test CMAKE_HAVE_LIBC_PTHREAD - Failed
-- Check if compiler accepts -pthread
-- Check if compiler accepts -pthread - yes
-- Found Threads: TRUE
-- Microsoft SEAL -> Version 4.1.1 detected
-- Microsoft SEAL -> Targets available: SEAL::seal
-- Configuring done
-- Generating done
-- Build files have been written to: /home/lxm/seal/homework
lxm@lxmliu2002:~/seal/homework$ make
[ 50%] Building CXX object CMakeFiles/he.dir/homework.cpp.o
[100%] Linking CXX executable he
[100%] Built target he
```

接着输入 `./he` 运行程序。



可以确认，$7 = 1 \times 1 + 2 \times 3$，$20 = 2 \times 2 + 3 \times 4$，$47 = 3 \times 3 + 4 \times 5$，说明程序逻辑正确，运行成功。

5 文件组织说明

本次实验中用到的所有代码均已置于 `./codes` 文件中。

- `./codes/seal_demo` 为测试 SEAL 的 demo 程序
- `./codes/test` 为 CKKS 应用示例程序
- `./codes/homework` 为改写的 CKKS 应用程序
- `./2112492 刘修铭 SEAL应用实践.pdf` 为本次实验的实验报告

```
.
├──codes
│   ├── homework
│   │   ├── CMakeLists.txt
│   │   ├── Makefile
│   │   ├── examples.h
│   │   ├── he
│   │   └── homework.cpp
│   ├── seal_demo
│   │   ├── CMakeLists.txt
│   │   ├── Makefile
│   │   ├── test
│   │   └── test.cpp
│   └── test
│       ├── CMakeLists.txt
```

```
16 │       ├── Makefile
17 │       ├── ckks_example.cpp
18 │       ├── examples.h
19 │       └── he
20 └── 2112492 刘修铭 SEAL应用实践.pdf
```

6 参考

本次实验主要参考教材内容完成。