[EX1_bfv_basics.cpp] 다항식 계산을 암호화된 채 BFV Scheme (Brakerski-Fan-Vercauteren)으로 계산

```
#include "examples.h"
using namespace std;
using namespace seal;
void example_bfv_basics(){
    print_example_banner("Example: BFV Basics");
```

```
Example: BFV Basics
```

기본세팅 : **[1] Enc파라미터 [2] Context [3]키 [4]암호기 [5]Evaluator [6]복호기**

[1] Encryption 파라미터 클래스생성 (3가지 파라미터) 안전성, 성능(속도,횟수)등에 영향 이해해야.

①poly_modulus_degree ②coeff_modulus (ciphertext) ③plain_modulus (plaintext ,bfv에만 있음)

- 어떤 계산, 무한횟수로 가능하지 않다. **Noise Budget** (노이즈예산)이라는 제한이 존재함.
- 곱셈할 때마다 노이즈예산 소진 노이즈예산이 0되면! 복호화 불가능 (덧셈 노이즈예산 소진≈0)
  - ➔ 따라서 곱셈의 횟수, 곱셈의 노이즈예산 소진률도 중요 : **multiplicative depth**중요.
  - ➔ 노이즈예산 초기값, 예산소진 비율을 Encryption파라미터가 정한다.
- 수행할 계산에 필요한 "연속된 곱셈횟수"를 파악해서 적절히 multiplicative depth정해야 결과보장.

****multiplicative depth란? 복호불가능한 노이즈쌓일 때까지의 횟수개념. 노이즈예산을 "Ring"으로 보고, 한바퀴 돌아 제자리로 왔을 때, 쌓인 노이즈가 복호 불가능한 시점이 된다.**

① **poly_modulus_degree** = 2^n : n>0의 조건이 있지만 실제론 n≥12 추천
- 클수록 **안전성↑, 연산속도↓**

② **coeff_modulus** = 60bit길이의 prime number들의 곱
- 비트길이 = prime#들 비트길이 합.
- 비트길이상한 = poly_modulus_degree로 정해짐 (표).
  (예) poly.mod.dege=4096 : coeff.mod.길이상한=109[bit]
  36bit길이-소수 3개곱으로 coeff.mod정할 수 있음.
- BFVDefault() : 자동으로 coeff.mod 찾음. poly.mod.deg만 정하면 됨.
- 클수록 **노이즈예산↑, 가능한 계산횟수↑**

poly.mod.deg	상한 of (coeff.mod 비트길이)
2^{10}= 1,024	27
2^{11}= 2,048	54
2^{12}= 4,096	109
2^{13}= 8,192	218
2^{14}= 16,384	438
2^{15}= 32,768	881

③ **plain_modulus** (CKKS에X) = 양의 정수 : 2^n or prime# 일부러 택하기도.
- 평문의 데이터타입 크기결정, 노이즈예산 초기값/소비율 정함.
- 클수록 **노이즈예산↓, 노이즈예산 소비량↑**

$$\textbf{Noise Budget @initial} = \log_2\left(\frac{\text{coeff.mod}}{\text{plain.mod}}\right) \text{ [bit]}$$

$$\textbf{Noise Bedget Consumption} = \log_2(\text{plain.mod}) + \cdots$$

```
    EncryptionParameters parms(scheme_type::bfv);
    size_t poly_modulus_degree = 4096;
    parms.set_poly_modulus_degree(poly_modulus_degree);
    parms.set_coeff_modulus(CoeffModulus::BFVDefault(poly_modulus_degree));
    parms.set_plain_modulus(1024);
```

[2] SEAL Context 생성

- 앞서 설정한 파라미터값 poly.mod.deg & plain.mod (coeff.mod는 자동이니깐 패스)이 유효한지 success/failed로 평가해주고, 파라미터들의 특성/정보를 지니는 obj임.

```
SEALContext context(parms);
    print_line(__LINE__);
    cout << "Set encryption parameters and print" << endl;
    print_parameters(context);
    cout << "Parameter validation (success): " << context.parameter_error_message() << endl;
    cout << endl;
```

```
Line 132 --> Set encryption parameters and print
Encryption parameters :
    scheme                  : BFV
    poly_modulus_degree     : 4096
    coeff_modulus size      : 109 (36 + 36 + 37) bits
    plain_modulus           : 1024
Parameter validation (success): valid
```

[3] KeyGenerator 클래스 (키) 생성

- SEAL은 비대칭 암호법으로 "**개인키(for 암호화) /공개키(for 복호화)**" 만들어야함.
  - 개인키 : KeyGenerator클래스를 생성하면 자동으로 개인키는 만들어진다.
  - 공개키 : create_public_key()로 필요할 때마다 여러개 만들 수 있다.

```
    KeyGenerator keygen(context);
    SecretKey secret_key = keygen.secret_key();
    PublicKey public_key;
    keygen.create_public_key(public_key);
```

[4] 암호화기 생성

- 암호화 하는 기능을 지닌 클래스로, 파라미터정보(context)와 공개키(public_key)가 필요함.

```
    Encryptor encryptor(context, public_key);
```

[5] 계산하는 Evaluator 생성

- 암호화된 데이터를 의뢰인으로부터 넘겨받은 service센터에서만 사용한다.
- 계산에 필요한 정보 (노이즈예산 등), 즉 파라미터 정보(context)가 필요함.

```
    Evaluator evaluator(context);
```

[6] 복호화기 생성

- "암호계산결과==평문계산결과"? 확인위해 복호화기 만든다. context/개인키가 필요함.

```
    Decryptor decryptor(context, secret_key);
```

$f(x) = \Sigma a_i x^i = 4x^4+8x^3+8x^2+8x+4 = 4\bullet(x^2+1)\bullet(x+1)^2$: x=6

- f(x) @x=6 : 의뢰인이 보낸 암호화된 데이터 x를 받아서 의뢰받은 계산f(x)를 수행해야함.
- $\{a_i\}$는 1)plaintext-입력으로 본다. 2)plaintext_modulus로 %된 결과이다.
- plaintext는 차수가 N=poly.mod.deg 보다 작은 다항식으로 본다. Ring이론의하면 T=plain.mod일 때 plaintext공간이 $Z_{T[X]}/(X^N+1)$, 즉 polynomial quotient ring이다.
- 일단!!! 의뢰인이 해야할 일 : "메시지(데이터와 원하는 계산식) ➔ 평문 ➔ 암호".
  - ① 메세지➔평문 : "x=6" 을 16진수로 표현, STRING타입으로 평문(x_plain) 만듦.
  - ② 평문 ➔암호 : 암호기로 평문을 암호문(x_encrypted)으로 만듦.
    - 새 암호문size=2확인***
    - 새 암호문(연산(곱셈)하기전 이므로)의 노이즈예산[bits]은 최대일것.
  - ③ 확인 : 암호문을 복호문(x_decrypted)으로 복원했을 때 평문과 일치하는지 확인.

***Encryption size : SEAL에서 암호문은 2개이상의 다항식으로 구성되며, 암호문 구성하는 다항식의 갯수를 size라고함. 새로 암호화된 암호문은 항상 size=2임.

```
print_line(__LINE__);
uint64_t x = 6;
Plaintext x_plain(uint64_to_hex_string(x));
cout << "Express x = " + to_string(x) + " as a plaintext polynomial 0x" + x_plain.to_string() + "."<< endl;
print_line(__LINE__);
Ciphertext x_encrypted;
cout << "Encrypt x_plain to x_encrypted." << endl;
encryptor.encrypt(x_plain, x_encrypted);
cout << "      + size of freshly encrypted x: " << x_encrypted.size() << endl;
cout << "      + noise budget in freshly encrypted x: " <<
                  decryptor.invariant_noise_budget(x_encrypted) << " bits"<< endl;
Plaintext x_decrypted;
cout << "      + decryption of x_encrypted: ";
decryptor.decrypt(x_encrypted, x_decrypted);
cout << "0x" << x_decrypted.to_string() << " ..... Correct." << endl;
```

```
Line 212 --> Express x = 6 as a plaintext polynomial 0x6.
Line 223 --> Encrypt x_plain ➔ x_encrypted.
    + size of freshly encrypted x          : 2
    + noise budget in freshly encrypted x  : 55 bits
    + decryption of x_encrypted            : 0x6 ..... Correct.
```

Naïve Way **[1]x^2+1 [2] $(x+1)^2$ [3] $f=4\bullet(x^2+1)$ [4]$f=f\bullet(x+1)^2$**

- 인수분해해서 multicative depth(곱셈횟수)를 줄일 수 있응 형태로 만든다.
  (예) 이 예제에서 f(x)의 곱셈횟수=10번은 인수분해시 4번으로 줄어들 수 있다.

[1] x_sq_plus_1 = x^2+1

- 부분결과 역시 ciphertext이다. 선언부터 go.
- 계산 ① $(x\_encrypted)^2$: evaluator.square에 In=x_encrypted, Out=x_sq_plus_1 인자적용.
  ② $(x\_encrypted)^2+1\_plain$: evaluator.add_plain_inplace에 In=1_plain, IO=x_sq_plus_1적용.
- 곱셈계산 후 달라진 점 찾기 : 1)노이즈예산 줄어드는 것(Δ=-22[bit]), 2)암호문size가 더이상 2아닌것.
- 계산결과 확인 : x_sq_plus_1을 decrypt해서 $x^2+1=37=32+4+1=2^5+2^2+1^0='b0010\_0101=$**0x25**

```
print_line(__LINE__);
cout << "Compute x_sq_plus_one (x^2+1)." << endl;
Ciphertext x_sq_plus_one;
evaluator.square(x_encrypted, x_sq_plus_one);
Plaintext plain_one("1");
evaluator.add_plain_inplace(x_sq_plus_one, plain_one);
cout << "      + size of x_sq_plus_one: " << x_sq_plus_one.size() << endl;
cout << "      + noise budget in x_sq_plus_one: " <<
                  decryptor.invariant_noise_budget(x_sq_plus_one) << " bits"<< endl;
Plaintext decrypted_result;
cout << "      + decryption of x_sq_plus_one: ";
decryptor.decrypt(x_sq_plus_one, decrypted_result);
cout << "0x" << decrypted_result.to_string() << " ..... Correct." << endl;
```

```
Line 270 --> Compute x_sq_plus_one  : x^2+1
    + size of x_sq_plus_one          : 3
    + noise budget in x_sq_plus_one  : 33 bits
    + decryption of x_sq_plus_one    : 0x25 ..... Correct.
```

[2] x_plus_1_sq = $(x+1)^2$

- evaluate>size/noise>복호결과 : $(x+1)^2=49=32+16+1=2^5+2^4+2^0=b'0011\_0001=$**0x31**

```
print_line(__LINE__);
cout << "Compute x_plus_one_sq ((x+1)^2)." << endl;
Ciphertext x_plus_one_sq;
evaluator.add_plain(x_encrypted, plain_one, x_plus_one_sq);
evaluator.square_inplace(x_plus_one_sq);
cout << "      + size of x_plus_one_sq: " << x_plus_one_sq.size() << endl;
cout << "      + noise budget in x_plus_one_sq: " <<
                  decryptor.invariant_noise_budget(x_plus_one_sq) << " bits"<< endl;
cout << "      + decryption of x_plus_one_sq: ";
```

```
            decryptor.decrypt(x_plus_one_sq, decrypted_result);
    cout << "0x" << decrypted_result.to_string() << " ..... Correct." << endl;
```

```
Line 300 --> Compute x_plus_one_sq : (x+1)²
    + size of x_plus_one_sq                 : 3
    + noise budget in x_plus_one_sq         : 33 bits
    + decryption of x_plus_one_sq           : 0x31 ..... Correct.
```

[3] result = 4 • (x²+1) ==4•x_sq_plus_1

[4] result = {4•(x²+1)} • (x+1)² ==result• x_plus_1_sq

```
print_line(__LINE__);
    cout << "Compute encrypted_result (4(x^2+1)(x+1)^2)." << endl;
    Ciphertext encrypted_result;
    Plaintext plain_four("4");
    evaluator.multiply_plain_inplace(x_sq_plus_one, plain_four);
    evaluator.multiply(x_sq_plus_one, x_plus_one_sq, encrypted_result);
    cout << "      + size of encrypted_result: " << encrypted_result.size() << endl;
    cout << "      + noise budget in encrypted_result: " <<
            decryptor.invariant_noise_budget(encrypted_result) << " bits" << endl;
    cout << "NOTE: Decryption can be incorrect if noise budget is zero." << endl;
    cout << endl;
```

```
Line 315 --> Compute encrypted_result   : 4(x²+1)(x+1)²
    + size of encrypted_result              : 5
    + noise budget in encrypted_result      : 4 bits
NOTE: Decryption can be incorrect if noise budget is zero.
```

<결과 정리>

	x_encrypted	x•x+1	(x+1)•(x+1)	4•(x²+1)•(x+1)²
곱셈수	0	1	1	2
size	2	3	3	5
noise	55	33	33	4

➔ size는 곱셉할 때마다 늘어나고 노이즈예산은 점점 소비되는 것을 알수있음.

➔ (x²+1), (x+1)² 의 size가 3으로 늘어서 노이즈소비량도 그만큼 커진다.

using Relinearization **[0]relin키생성 [1]x² +1 [2] (x+1)² [3] f=4•(x²+1) [4]f=f•(x+1)²**

- 앞에서의 방식은 계산종료시 노이즈예산이 55➔4[bit]까지 소비되었다.
- 곱셈연산이 진행될수록 암호문size가 길어지는데 이는 더 많은 노이즈예산 소비를 가져온다.
- Relinearization : 곱셈연산마다 커진 암호문 size를 fresh상태의 size=2로 줄이는 과정이다.
    - Relinearization역시 계산이라서 비용이 들지만 커진암호로 인한 비용보단 적다.
    - Relinearization은 relin키 필요하고, KeyGen으로 쉽게 생성된다.

[0] relin키생성

```
    print_line(__LINE__);
    cout << "Generate relinearization keys." << endl;
    RelinKeys relin_keys;
    keygen.create_relin_keys(relin_keys);
```

```
Line 353 --> Generate relinearization keys.
```

[1] x_sq_plus_1 = x²+1

- x•x계산 > size출력 > relinearize(x²) >size출력 > x²+1계산 >노이즈체크 >복호결과확인

```
    print_line(__LINE__);
    cout << "Compute and relinearize x_squared (x^2)," << endl;
    cout << string(13, ' ') << "then compute x_sq_plus_one (x^2+1)" << endl;
    Ciphertext x_squared;
    evaluator.square(x_encrypted, x_squared);
    cout << "      + size of x_squared: " << x_squared.size() << endl;
    evaluator.relinearize_inplace(x_squared, relin_keys);
    cout << "      + size of x_squared (after relinearization): " << x_squared.size() << endl;
    evaluator.add_plain(x_squared, plain_one, x_sq_plus_one);
    cout << "      + noise budget in x_sq_plus_one: " <<
                        decryptor.invariant_noise_budget(x_sq_plus_one) << " bits"<< endl;
    cout << "      + decryption of x_sq_plus_one: ";
                        decryptor.decrypt(x_sq_plus_one, decrypted_result);
    cout << "0x" << decrypted_result.to_string() << " ..... Correct." << endl;
```

```
Line 361 --> Compute and relinearize x_squared (x²),
              then compute x_sq_plus_one (x²+1)
    + size of x_squared                     : 3
    + size of x_squared (after relinearization)     : 2
    + noise budget in x_sq_plus_one         : 33 bits
    + decryption of x_sq_plus_one           : 0x25 ..... Correct.
```

[2] x_ plus_1_sq = (x+1)²

- x+1계산 > (x+1)•(x+1)계산 >size출력 > relinearize((x+1)²) > size출력 > 노이즈체크 >복호결과확인

```
    print_line(__LINE__);
    Ciphertext x_plus_one;
    cout << "Compute x_plus_one (x+1)," << endl;
    cout << string(13, ' ') << "then compute and relinearize x_plus_one_sq ((x+1)^2)." << endl;
    evaluator.add_plain(x_encrypted, plain_one, x_plus_one);
    evaluator.square(x_plus_one, x_plus_one_sq);
    cout << "      + size of x_plus_one_sq: " << x_plus_one_sq.size() << endl;
    evaluator.relinearize_inplace(x_plus_one_sq, relin_keys);
    cout << "      + size of x_plus_one_sq (after relinearization): " << x_plus_one_sq.size() << endl;
    cout << "      + noise budget in x_plus_one_sq: " <<
                    decryptor.invariant_noise_budget(x_plus_one_sq) << " bits"<< endl;
    cout << "      + decryption of x_plus_one_sq: ";
    decryptor.decrypt(x_plus_one_sq, decrypted_result);
    cout << "0x" << decrypted_result.to_string() << " ..... Correct." << endl;
```

```
Line 376 --> Compute x_plus_one (x+1),
              then compute and relinearize x_plus_one_sq ((x+1)²).
    + size of x_plus_one_sq                    : 3
    + size of x_plus_one_sq (after relinearization) : 2
    + noise budget in x_plus_one_sq            : 33 bits
    + decryption of x_plus_one_sq              : 0x31 ..... Correct.
```

[3] result = 4 • (x²+1) ==4•x_sq_plus_1

[4] result = {4•(x²+1)} • (x+1)² ==result• x_plus_1_sq

```
print_line(__LINE__);
    cout << "Compute and relinearize encrypted_result (4(x^2+1)(x+1)^2)." << endl;
    evaluator.multiply_plain_inplace(x_sq_plus_one, plain_four);
    evaluator.multiply(x_sq_plus_one, x_plus_one_sq, encrypted_result);
    cout << "      + size of encrypted_result: " << encrypted_result.size() << endl;
    evaluator.relinearize_inplace(encrypted_result, relin_keys);
    cout << "      + size of encrypted_result (after relinearization): "<<encrypted_result.size() << endl;
    cout << "      + noise budget in encrypted_result: " <<
                    decryptor.invariant_noise_budget(encrypted_result) << " bits"<< endl;
    cout << endl;
    cout << "NOTE: Notice the increase in remaining noise budget." << endl;
    print_line(__LINE__);
    cout << "Decrypt encrypted_result (4(x^2+1)(x+1)^2)." << endl;
    decryptor.decrypt(encrypted_result, decrypted_result);
    cout<<"      + decryption of 4(x^2+1)(x+1)^2 = 0x"<<decrypted_result.to_string()<<" ..... Correct."<<endl;
    cout << endl;
```

```
Line 390 --> Compute and relinearize encrypted_result (4(x^2+1)(x+1)^2).
    + size of encrypted_result                 : 3
    + size of encrypted_result (after relinearization) : 2
    + noise budget in encrypted_result         : 10 bits
NOTE: Notice the increase in remaining noise budget.
Line 407 --> Decrypt encrypted_result (4(x^2+1)(x+1)^2).
    + decryption of 4(x^2+1)(x+1)^2      = 0x54 ..... Correct.
```

```
// Wrong Parameter EX
    print_line(__LINE__);
    cout << "An example of invalid parameters" << endl;
    parms.set_poly_modulus_degree(2048);
    context = SEALContext(parms);
    print_parameters(context);
    cout << "Parameter validation (failed): " << context.parameter_error_message() << endl << endl;
```

```
Line 425 --> An example of invalid parameters
Encryption parameters :
    scheme                      : BFV
    poly_modulus_degree         : 2048
    coeff_modulus size          : 109 (36 + 36 + 37) bits
    plain_modulus               : 1024
Parameter validation (failed): parameters are not compliant with HomomorphicEncryption.org security standard
```