동형암호
(블록체인)

https://youtu.be/o8QI0ayuEwk

동형암호

- **동형암호 (Homomorphic Encryption)**
  - 평문과 암호문의 동형(Homomorphic) 성질로 인해 **암호문 상태에서도 임의의 연산을 수행할 수 있는** 암호기술

    → 평문을 **연산하여 암호화**한 결과 = 평문을 **암호화하여 연산**한 결과
  - 동형암호를 활용하면 민감한 정보를 안전하게 보호하면서도 유용하게 데이터를 활용할 수 있음

    → 비밀 데이터인 $\boldsymbol{a}$와 $\boldsymbol{b}$를 노출하지 않으면서 암호화한 상태로 연산 가능

$$Enc(\boldsymbol{a}+\boldsymbol{b}) = Enc(\boldsymbol{a}) + Enc(\boldsymbol{b})$$

$$Enc(\boldsymbol{a} \times \boldsymbol{b}) = Enc(\boldsymbol{a}) \times Enc(\boldsymbol{b})$$

응용분야

- 민감한 정보에 대한 처리 (의료, 금융 등)
- 전자투표
- Outsourced storage and computations (Ex) Cloud
- Privacy-preserving ML (Machine learning)

동형암호 유형

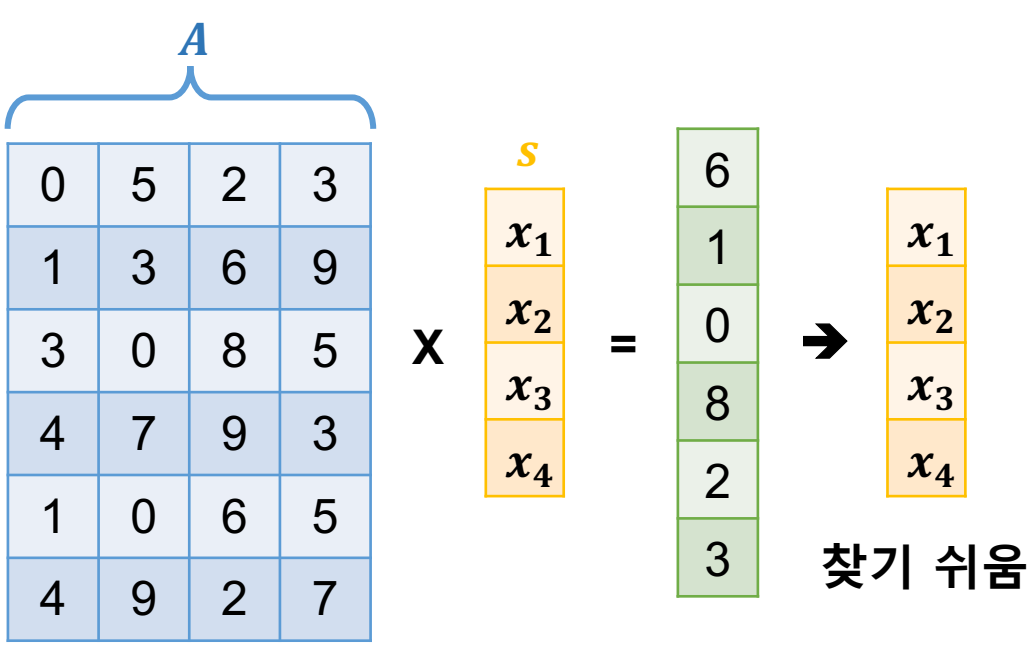
- **Partially Homomorphic Encryption (PHE)**
  - 덧셈 또는 곱셈 중 **하나의 연산에 대해서만** 동형암호 성질 유지
  - RSA-based PHE (1978) : 곱셈 연산만 지원, Schoolbook RSA에서만 동작하므로 안전하지 않음
  - Goldwasser-Micali's PHE (1982) : 이진수의 덧셈 연산만 지원
- **Somewhat Homomorphic Encryption (SHE)**
  - 덧셈과 곱셈 연산을 **모두 지원하지만 제한된 횟수동안** 동형암호 성질 유지
  - 암호문에 Noise를 삽입하는 방식으로 기밀성 유지 → 연산을 반복할수록 Noise 증가 → 원본 메시지 훼손
- **Fully Homomorphic Encryption (FHE)**
  - **모든 유형의 연산에 대해 횟수 제한없이** 동형암호 성질 유지
  - 암호화된 비밀키를 통해 복호화하여 Noise를 없앤 후 재암호화하는 Bootstrapping 기법 → 연산 횟수 제한 극복
  - SHE + Bootstrapping → FHE
  - 연산 부하 발생 가능

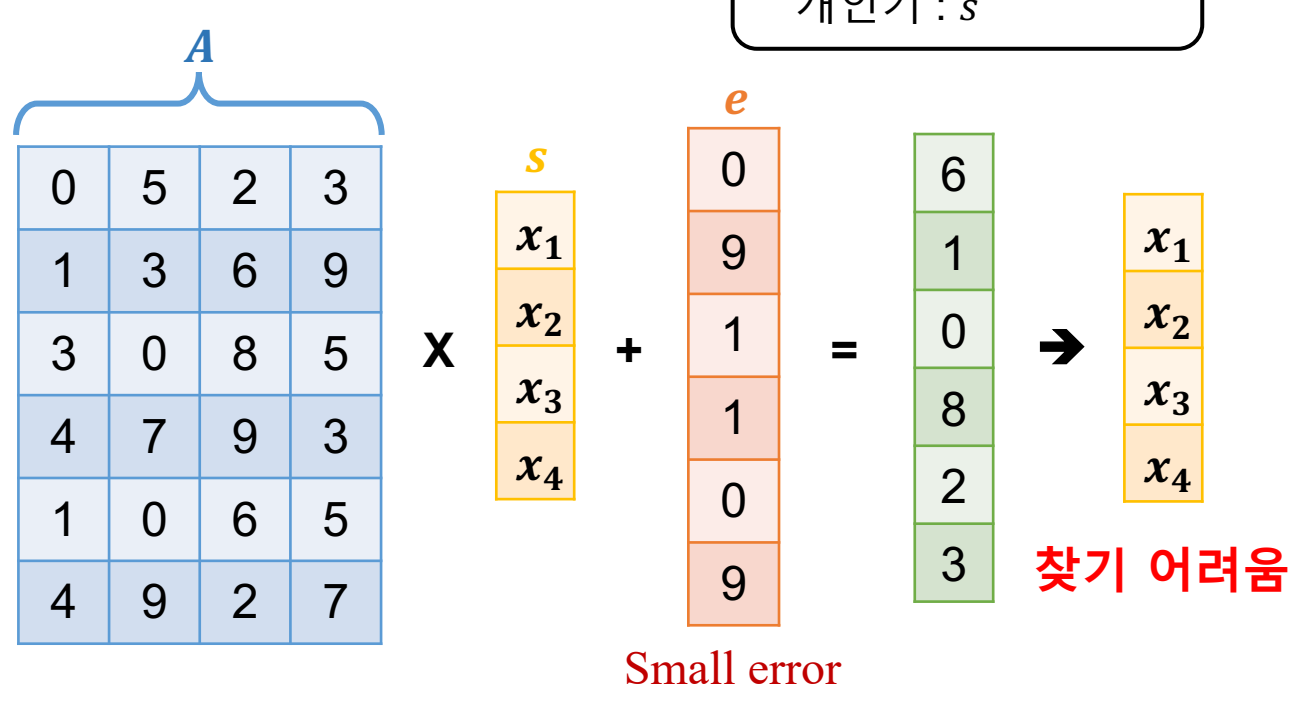
동형암호 주요 스킴

- **성능 이슈** (Bootstrapping으로 인해 느린 속도)로 인해 LWE 기반 동형암호 스킴이 설계되고 있음
- **LWE (Learning With Error) 기반 암호 (2005) : BGV, BFV, CKKS**
  - e (Small error)를 포함한 연립선형방정식의 해를 구하는 문제 (e가 더해지면 s를 찾는 것이 어려워짐)

    → A와 $A \cdot s + e$가 주어졌을 때, **s를 찾는 것의 어려움에 기반**
  - 이를 기반으로 하는 동형암호는 격자기반암호와 **동일한 안전성** 제공

공개키 : A, $A \cdot s + e$
개인키 : s





LWE 기반 동형암호 스킴

- **LWE 기반 동형암호 스킴**
  - **정수 상에서의 HE (exact arithmetic)**
    - 2011·2014 : Brakerski-Gentry-Vaikuntanathan (**BGV**) [1]
    - 2012 : Brakerski / Fan-Vercauteren (**BFV**) [2]
  - **실수 상에서의 HE (approximate arithmetic)**
    - 2017 : Chen-Kim-Kim-Song (**CKKS**) [3]

[1] https://dl.acm.org/doi/10.1145/2633600
[2] https://eprint.iacr.org/2012/144
[3] https://eprint.iacr.org/2016/421

Brakerski-Gentry-Vaikuntanathan (BGV)

- Ring-LWE상에서의 (leveled) FHE
  - 사용자가 정한 **레벨 상에서는 제한 없이 완전 동형 암호** 사용 가능

- **BGV 전체 과정**
  1. **Parameter generation**
     - 보안 레벨, 평문 형태, 평문 및 암호문의 범위
  2. **Key generation**
     개인키, 공개키, EK (=RK, relinearization)
  3. **Encryption**
     비밀키 또는 공개키로 암호화
  4. **Decryption**
     개인키로 복호화
  5. **Evaluation**
     덧셈 및 곱셈에 대한 동형 연산
  6. **Refresh**
     연산 횟수의 제한을 없애기 위함 (Relinearization, Modswitch, Bootstrap)

BGV – Parameter and Key generation

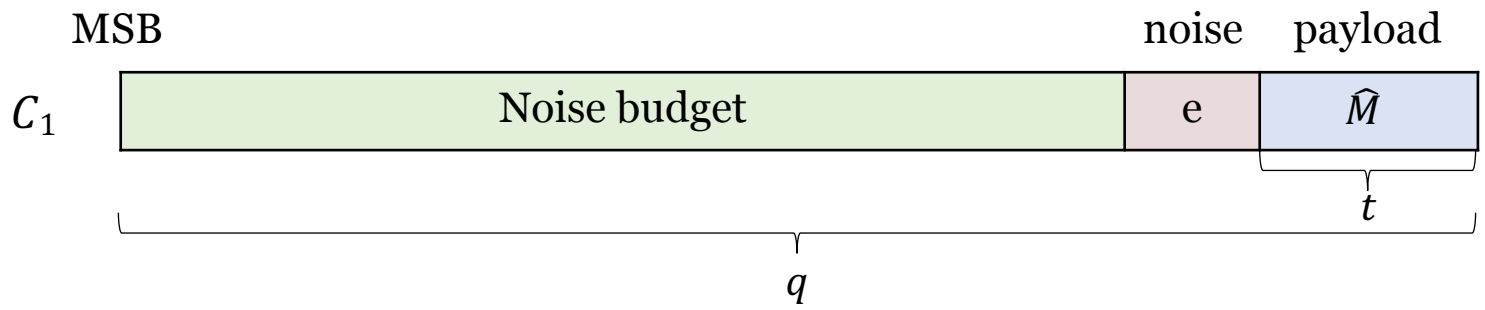
- **Parameter generation**
  - $\boldsymbol{\lambda}$: 보안 레벨
  - $\boldsymbol{PT}$: 평문 형태
    - **정수** : MI (Modular integer), Extension fields (EX)
    - **실수** : Approximate numbers
  - $\boldsymbol{B}$: 최대 곱셈 가능한 depth (BGV에서는 최대 곱셈 범위를 정할 수 있음)
  - **평문 및 암호문의 범위**
    - plaintext ring : $\boldsymbol{R/tR}$ $(mod\ t$ 상의 ring)
    - ciphertext ring : $\boldsymbol{R/qR}$ $(mod\ q$ 상의 ring, $t < q)$

- **Key generation**
  - $\boldsymbol{SK\ (s)}$: 가우시안 분포에서 선택 $(-1, 0, 1$과 같이 매우 작은 값)
  - $\boldsymbol{PK}$: (pk_1, pk_2)와 같이 두 파트로 구성
  - $\boldsymbol{EK\ (= RK)}$: Evaluation key (=Relinearization key), (EK_1, EK_2)과 같이 두 파트로 구성

BGV – Encryption

- Encryption (PK 사용)
  - 암호화 하고자 하는 M를 R/tR로 매핑하여 $\widehat{M}$얻음
    - $\widehat{M}$: 아래 그림의 payload에 해당하며, **t라는 modulus 범위 내에 포함**
    - 다항식 ring이므로 $a_0x_0 + \cdots + a_{n-1}x_{n-1}$형태로 표현되며, **각 계수가 mod t 상에서 표현**

  - **암호문 C를 얻음**
    - $C = (C_1, C_2) = (pk_1u + te_1 + \widehat{M}, pk_2u + te_2)$; 하나의 암호문은 두 파트로 구성
      - **C_1 : 메시지가 임베딩** 된 상태
      - C_2 : 복호화 과정에서 **메시지를 가린 부분을 제외하기 위한 값들로 구성**
    - e: t가 곱해진 값 (te_1, te_2)이므로 payload 다음 범위에 위치
    - **$t < q$** : 나머지 범위는 동형 연산을 수행하면서 **에러가 증가할 수 있는 범위 → noise budget**
      해당 **범위를 초과**할 경우, **refresh를 통해 줄여줘야 함**



BGV – Decryption

- Decryption ($\boldsymbol{SK}$ ($\boldsymbol{s}$) 사용)
  - $\boldsymbol{C'}$ 계산

    $= (C_1 + C_2 s) = \left(pk_1 u + te_1 + \widehat{M} + (pk_2 u + te_2)s\right)$

    → 해당 수식을 계산하면 $\boldsymbol{\widehat{M} + t(-eu + e_1 + e_2)}$

  - $\boldsymbol{C' \bmod t}$
    - t의 배수는 0이 되므로 **에러 값들은 소거**
    - 따라서 $\boldsymbol{\widehat{M}}$을 구할 수 있음

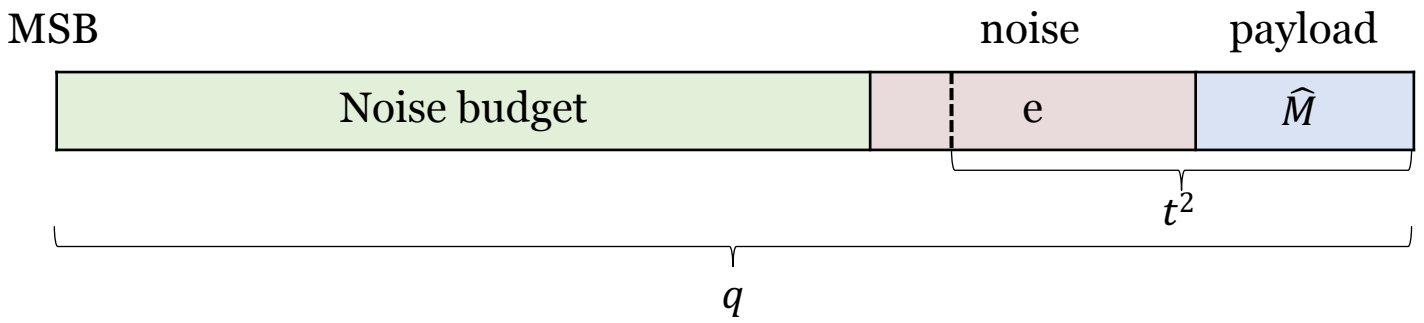
BGV – Evaluation

- 연산 대상인 **각 암호문은 2 파트로 구성** $(C_1 = (C_1{}^{(1)}, C_1{}^{(2)}),\ C_2 = (C_2{}^{(1)}, C_2{}^{(2)}))$
- **덧셈 동형 연산**
  - M_1, M_2에 대한 암호문 2개에 대한 덧셈 $(C_1 + C_2)$
  - M_3에 대한 암호문 형태로 변환됨

- **곱셈 동형 연산**
  - Ring에서의 텐서 곱이 수행
    - 곱셈 연산 후에는 **암호문이 3파트로 구성**

      → **곱셈을 할수록 항이 늘어나고 에러 범위가 커짐**

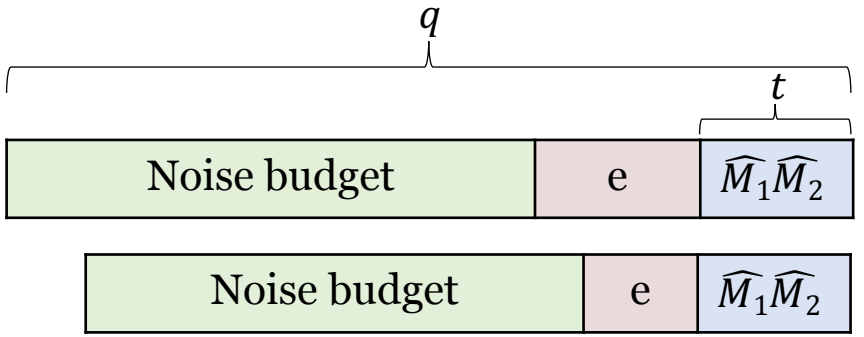
      $(\widehat{M_1}\widehat{M_2} + t(\widehat{M_1}v_2 + \widehat{M_2}v_1) + t^2v_1v_2)$: t^2이상까지 에러 전파
    - 이를 줄이기 위한 과정 (Refresh) 필요



BGV – Refresh

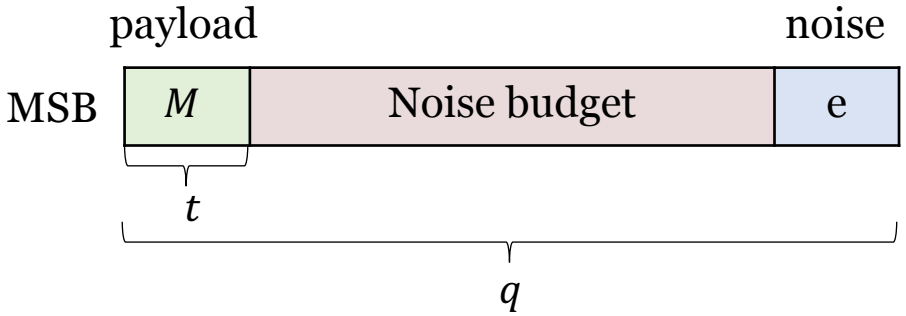
- **곱셈 연산을 반복할 경우 에러가 증가하여 동형 연산을 수행할 수 없는 상황이 발생**
- Relinearization
  - 곱셈 시 **암호문의 항이 늘어나는 문제에 대한 해결 방법**
  - EK를 사용해서 암호문의 항을 **3개에서 2개로 줄임**
  - $\boldsymbol{EK}$ **에는 비밀키 (s)가 사용**
    - $EK = (EK_1, EK_2) = (-(as + 3) + s^2, a)$
  - ${C_1}^*, {C_2}^*, {C_3}^*$는 암호문의 각 항
  - $\widehat{{C_1}^*} = [{C_1}^* + EK_1{C_3}^*]_q, \widehat{{C_2}^*} = [{C_2}^* + EK_2{C_3}^*]_q$

    → $\boldsymbol{{C_3}^*}$**에** $\boldsymbol{EK}$**를 곱하여 2개의 항으로 변환**
- Modswitch
  - 곱셈 시 **에러 값이 지수승으로 증가하는 문제에 대한 해결 방법**
  - Modulus q상의 암호문을 **modulus q'에서 표현** → Modulus를 변경하더라도 **같은 값**을 표현하도록 함
  - $\boldsymbol{q' < q,\ q \equiv q'\ mod\ t}$
    - 이로 인해 **에러 값의 범위를 줄이지만 연산 결과에는 영향 없음**
  - 이때, q'의 값을 **계층적으로 줄여서 여러 번의 곱셈 연산**을 수행할 수 있도록 함
    - q_L (원본 q와 동일) $> q_{L-1} > \cdots > q_0$ → 총 L번만큼의 곱셈 가능



Brakerski/Fan-Vercauteren (BFV)

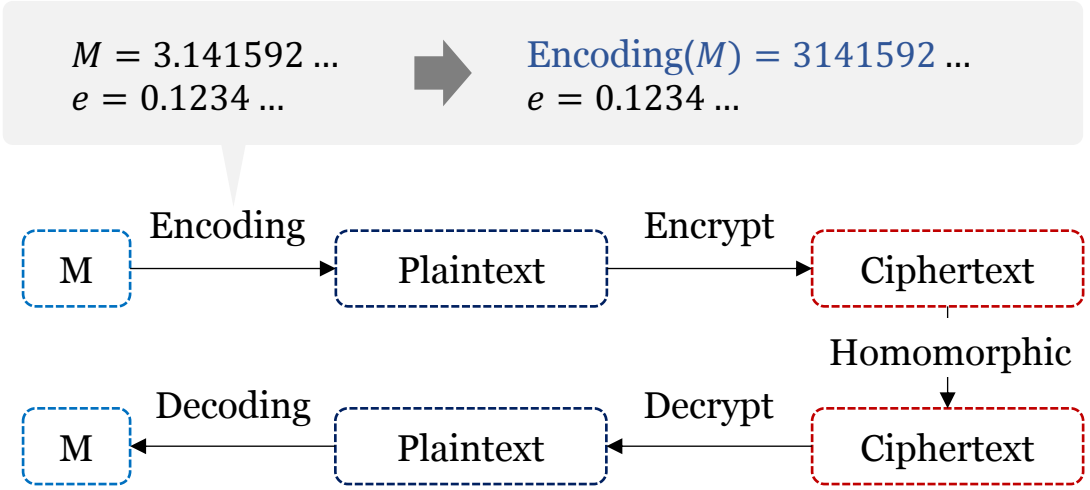
- **BFV**
  - BGV와 유사하지만 암호화 과정의 구조가 다름
  - **Scaling factor를 곱하여 평문의 위치를 조절**
    - **에러 (e)와 메시지 (M)가 떨어져 있도록 하는 과정**
  - 복호화 과정 유사 (Modulus t상에서의 메시지 얻어냄)
  - 동형 덧셈 (BGV와 유사)
  - **동형 곱셈**
    - Scaling 과정이 있었기 때문에 $\frac{t}{q}$ 가 곱해지는 것 외에는 동일
    - **BGV와 마찬가지로 에러가 증가함**
      - 이로 인해 Noise budget를 다 소모하여 **메시지를 훼손시킬 수 있음**
        → Refresh **과정 필요 (Relinearization)**



Cheon-Kim-Kim-Song (CKKS)

- CKKS
  - BGV, BFV와 달리 **실수/복소수에 대한 동형 연산** 제공
  - 동형 연산에 효율적인 고정 소수점 사용

  - **BGV와 유사**하지만 실수/복소수이므로 **인/디코딩필요**
    1. **인코딩하여 정수로 변환** (다항식의 원소로 표현)
    2. 정수 상에서 동형 연산
    3. **디코딩 통해 실수 메시지로 변환**



  - **Encoding 과정에서 Scaling factor (Δ) 곱 수행**
    - **에러 (e)와 메시지(M) 간의 격차를 생성하기 위함**
    - M와 e가 큰 차이가 나지 않는 경우, e 누적 시 M 훼손 가능
    - **M에 대한 Scaling 통해 e가 M에 영향을 주지 않도록 함**
    - 그러나 곱셈 시, M에 곱해진 Δ가 다른 경우 존재
      → **Rescale 필요** (e범위 줄이고 scale 일치 시킴)

동형암호 오픈소스 라이브러리 종류

국가	라이브러리명	개발년도	개발사/기관	비고
미국	**HElib**	2013	IBM	최초 동형암호 라이브러리 BGV 스킴 지원
	SEAL	2015	Microsoft	**BGV, BFV, CKKS** 등의 스킴 지원
	PALISADE 2	2017	MIT	격자 기반 암호 라이브러리
	cuHE	2017	WPI	GPU를 통한 고속화 라이브러리
	cuFHE	2018		
유럽	NFLlib	2016	Sorbonne	유럽 'H2020' HEAT 프로젝트 결과물
	TFHE	2017	KU Leuven	TFHE 스킴 지원 라이브러리
	Lattigo	2019	EPFL	다중 사용자용 라이브러리
한국	**HeaAN**	2017	서울대	**CKKS** 스킴 지원 라이브러리

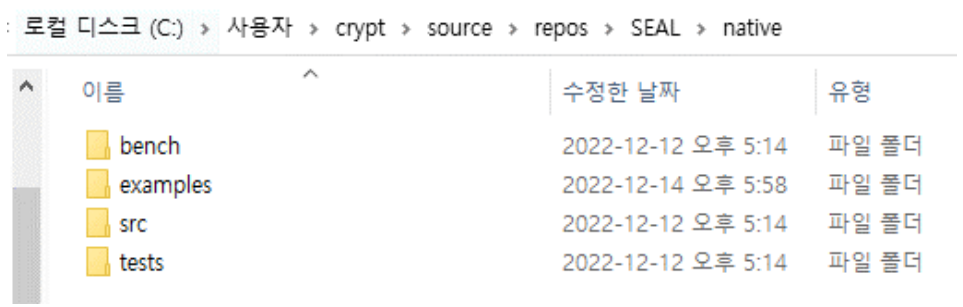
https://seed.kisa.or.kr/kisa/ngc/pec.do

Microsoft SEAL

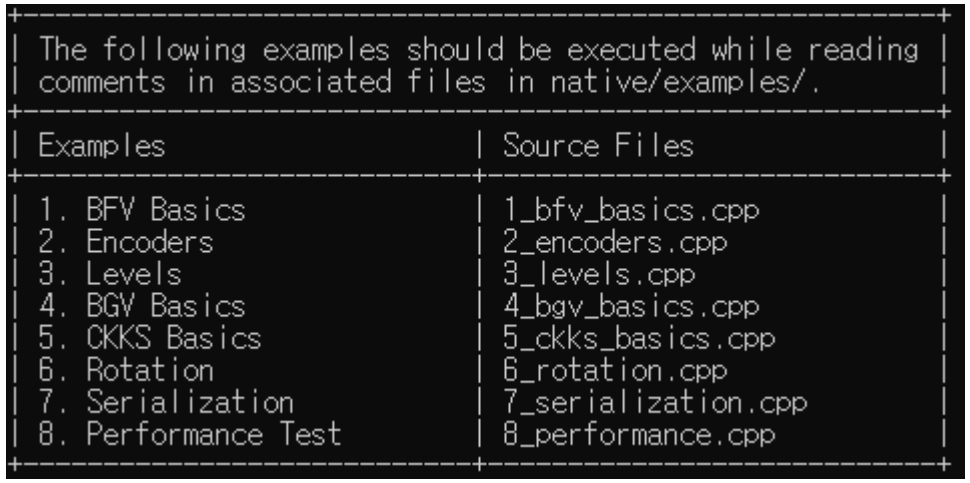
- Microsoft SEAL
  - BGV, BFV, CKKS 등을 지원하는 동형암호 **라이브러리**
  - 다양한 플랫폼 (윈도우, 리눅스, 맥 등)을 지원
  - **임베디드 환경**에서의 라이브러리도 지원

https://github.com/microsoft/SEAL

- **Windows / Visual studio 2019**에서 실행해본 결과 **정상 작동**
  - 다음과 같이 BGV, CKKS 등에 대한 **예제 파일을 제공**
  - **다른 프로그램을 구현할 경우**, 추가적으로 사용할 라이브러리 포함시키면 **해당 코드를 활용 가능**



예제 파일 및 소스 코드 제공



예제 파일 실행 메인 화면

Microsoft SEAL – BGV 예제

- Parameter and key generation

다항식 차수, q, t 설정

```
EncryptionParameters parms(scheme_type::bgv);
size_t poly_modulus_degree = 8192;   //8192차 다항식
parms.set_poly_modulus_degree(poly_modulus_degree);
parms.set_coeff_modulus(CoeffModulus::BFVDefault(poly_modulus_degree)); // 암호문 modulus (q) :218-bit
parms.set_plain_modulus(PlainModulus::Batching(poly_modulus_degree, 20)); // 평문 modulus (t)
SEALContext context(parms);

print_line(__LINE__);
cout << "Set encryption parameters and print" << endl;
print_parameters(context);

// 키 생성 (SK, PK, RK)
KeyGenerator keygen(context);
SecretKey secret_key = keygen.secret_key();
PublicKey public_key;
keygen.create_public_key(public_key);
RelinKeys relin_keys;
keygen.create_relin_keys(relin_keys);
```

Encryption, Decryption, Relinearization에 필요한 **키 생성**

사용할 다항식 차수에 따른 q값

poly_modulus_degree	max coeff_modulus bit-length
1024	27
2048	54
4096	109
8192	218
16384	438
32768	881

```
Line   47 --> Set encryption parameters and print
/
| Encryption parameters :
|   scheme: BGV
|   poly_modulus_degree: 8192
|   coeff_modulus size: 218 (43 + 43 + 44 + 44 + 44) bits
```

실행 화면

Microsoft SEAL – BGV 예제

- Encryption
  - ***PK***를 사용하여 암호화

```
// 암호화
Ciphertext x_encrypted;
print_line(__LINE__);
cout << "Encrypt x_plain to x_encrypted." << endl;
encryptor.encrypt(x_plain, x_encrypted);
cout << "    + noise budget in freshly encrypted x: " << decryptor.invariant_noise_budget(x_encrypted) << " bits"
```

```
Line  90 --> Encrypt x_plain to x_encrypted.
    + noise budget in freshly encrypted x: 146 bits
```

암호화 (동형 연산 X) 후의
noise budget : 146-bits

Microsoft SEAL – BGV 예제

- **동형 연산 수행** (x^2, x^4, x^8에 대한 예제)
  - 암호문 생성 → 동형 연산 → Relinearization

동형 연산

Relinearization

```
//암호화 후, x^2 계산을 위한 과정
print_line(__LINE__);
cout << "Compute and relinearize x_squared (x^2)," << endl;
Ciphertext x_squared;
evaluator.square(x_encrypted, x_squared);
cout << "    + size of x_squared: " << x_squared.size() << endl;

// 곱셈 후 relinearization 수행
// 암호문의 항이 3개에서 2개로 줄어들고,
evaluator.relinearize_inplace(x_squared, relin_keys);
cout << "    + size of x_squared (after relinearization): " << x_squared.size() << endl;
cout << "    + noise budget in x_squared: " << decryptor.invariant_noise_budget(x_squared) << " bits" <
```

```
Line 100 --> Compute and relinearize x_squared (x^2)
    + size of x_squared: 3
    + size of x_squared (after relinearization): 2
    + noise budget in x_squared: 110 bits
    + result plaintext matrix ..... Correct.
```

```
Line 118 --> Compute and relinearize x_4th (x^4),
    + size of x_4th: 3
    + size of x_4th (after relinearization): 2
    + noise budget in x_4th: 36 bits
    + result plaintext matrix ..... Correct.
```

```
Line 134 --> Compute and relinearize x_8th (x^8),
    + size of x_8th: 3
    + size of x_8th (after relinearization): 2
    + noise budget in x_8th: 0 bits
NOTE: Decryption can be incorrect if noise budget is zero
```

Relinearization 통해 **암호문 항 개수 감소** (3개 → 2개)

Noise budget : **146-bits → 110-bits**

연산을 반복함에 따라 **noise budget 감소**

Noise budget : **110-bits → 36-bits**

Noise budget : **36-bits → 0-bits**

Noise budget을 모두 소모하여 x^8에 대한 **정상 동작 실패**

Microsoft SEAL – BGV 예제

- **동형 연산 수행** (Relinearization + **Modswitch**)

동형 연산

Relinearization

Modswitch

```
// x^2
print_line(__LINE__);
cout << "Compute and relinearize x_squared (x^2)," << endl;
cout << "    + noise budget in x_squared (previously): " << decryptor.invariant_noise_budget(x_squared) << " bits"
     << endl;
evaluator.square(x_encrypted, x_squared);

//relinearization
evaluator.relinearize_inplace(x_squared, relin_keys);
// modswitch (에러 범위 줄임)
// 곱셈 연산 후, relinearization + modswitch 적용 -> noise budget이 적게 줄어듦 (=더 많이 남음) -> 에러 범위 감소
evaluator.mod_switch_to_next_inplace(x_squared);
cout << "    + noise budget in x_squared (with modulus switching): " << decryptor.invariant_noise_budget(x_squared)
```

```
Line 196 --> Compute and relinearize x_8th (x^8),
    + noise budget in x_8th (previously): 0 bits
    + noise budget in x_8th (with modulus switching): 15 bits
    + result plaintext matrix ..... Correct.
```

Relinearization만 적용한 경우보다 **noise budget이 많이 남음**
(에러 범위 감소)

Noise budget : 기존 0 bits → Modswitch 적용 후 15-bits

이로 인해 x^8**에 대해서도 정상 연산 가능**

Microsoft SEAL – CKKS 예제

- BGV와 유사
  - 정수로의 인코딩 과정 및 곱셈 후의 rescaling 부분은 다름

```
Plaintext x_plain;
print_line(__LINE__);
cout << "Encode input vectors." << endl;
encoder.encode(input, scale, x_plain);
Ciphertext x1_encrypted;
encryptor.encrypt(x_plain, x1_encrypted);
```

```
print_line(__LINE__);
cout << "Compute, relinearize, and rescale (PI*x)*x^2." << endl;
evaluator.multiply_inplace(x3_encrypted, x1_encrypted_coeff3);
evaluator.relinearize_inplace(x3_encrypted, relin_keys);
cout << "    + Scale of PI*x^3 before rescale: " << log2(x3_encrypted.scale()) << " bits" << endl;
evaluator.rescale_to_next_inplace(x3_encrypted);
cout << "    + Scale of PI*x^3 after rescale: " << log2(x3_encrypted.scale()) << " bits" << endl;
```

Encoding
(실수/복소수 → 정수)

동형 연산 → **Relinearization**
암호문 항의 수 감소

→ **Rescale**
서로 다른 Scale factor 일치시킴

Encoding

Rescale

```
Evaluating polynomial PI*x^3 + 0.4x + 1 ...
Line 129 --> Encode input vectors.
Line 140 --> Compute x^2 and relinearize:
    + Scale of x^2 before rescale: 80 bits
Line 152 --> Rescale x^2.
    + Scale of x^2 after rescale: 40 bits
Line 165 --> Compute and rescale PI*x.
    + Scale of PI*x before rescale: 80 bits
    + Scale of PI*x after rescale: 40 bits
Line 180 --> Compute, relinearize, and rescale (PI*x)*x^2.
    + Scale of PI*x^3 before rescale: 80 bits
    + Scale of PI*x^3 after rescale: 40 bits
Line 192 --> Compute and rescale 0.4*x.
    + Scale of 0.4*x before rescale: 80 bits
    + Scale of 0.4*x after rescale: 40 bits
```

연구 사례 1

SmartFHE

Private 계정 **Public 계정**

컨트랙트 및 트랜잭션 (Public, Private)

동형암호 및 영지식 증명 사용

Microsoft의 SEAL 사용

완전 동형 암호화를 통해 개인 정보를 보호하는
스마트 컨트랙트를 지원하는 최초의 프레임 워크

컨트랙트에 사용되는 지갑 주소 및 다양한 개인 정보 노출 방지

단일 사용자 / **여러 사용자**에 대해 동일 기능 지원

BGV (벤치마킹은 BFV 사용) Multi key HE (다자간 계산 수행)

사용자의
실행 시간 측면에서
높은 성능 달성

트랜잭션	거래 금액 동형 암호화	영지식 증명	설명
Public → Public / Private	X	X	기존의 이더리움 트랜잭션과 동일
Private → Private	O	O	거래 금액 암호화 + 이체 금액 이상의 잔액 증명
Private → Public	X	O	이체 금액 이상의 잔액 증명

스마트 컨트랙트	개인 정보 동형 암호화	영지식 증명	설명
Private → Private	O	O	개인 정보 (컨트랙트 수행 조건)를 암호화하여 연산 + 해당 조건들이 실제로 충족되는지 검증

연구 사례 2

에스크로에서의 데이터 보안 문제를 해결하기 위한 블록체인 기반의 엣지 컴퓨팅

블록체인을 통한 무결성 보장 + 엣지 컴퓨팅을 통한 다자간 계산 = **효율적이며 안전한 다자간 계산 가능**

에스크로 클라이언트

모든 데이터 암호화

Edge 노드

Edge 노드 … **Edge 노드**

동형 암호를 통해 데이터에 대한 연산 후
결과를 체인에 기록, 클라이언트에게 반환

Paillier 암호 시스템 (덧셈 동형)

데이터 동형 암호화를 통해 프라이버시 보호 + 블록체인 활용을 통해 무결성 보장

그러나 동형암호화 후 연산 수행 시 **클라이언트 부담 증가** + 블록체인 상에서 전부 동작 시 **체인의 효율성 감소**

동형암호 + 블록체인에 엣지 컴퓨팅 구조 적용

연구 사례 3

블록체인 및 동형암호 기반의 스마트 그리드 시스템

사용자의 전기 사용량 데이터를 안전하게 집계하고 예측에 활용

동형암호화를 통한 개인 정보 보호 및 기밀성 유지

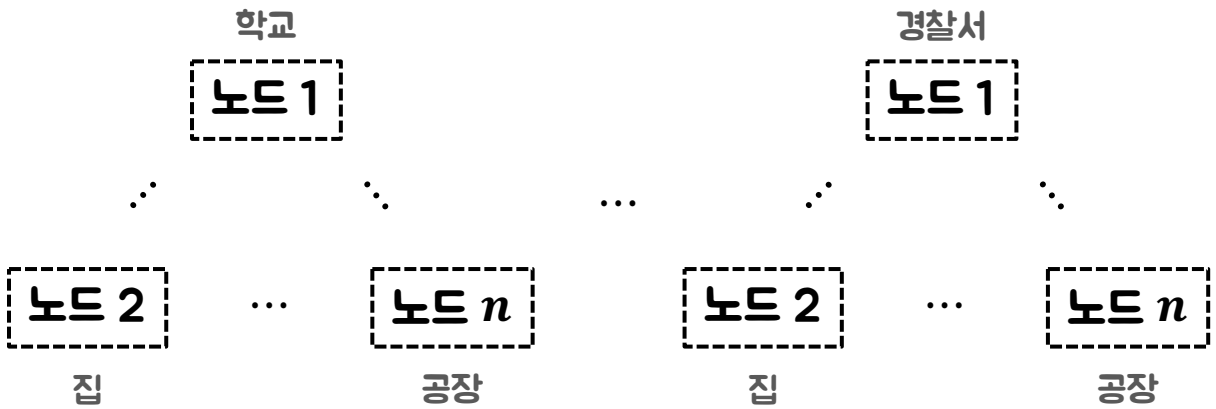
모든 과정에서 집계자는 데이터를 볼 수 없으나
정확한 데이터를 전력 센터에 제공 가능

집계 모듈을 블록체인에 배포

집계 + 결과를 체인에 저장하여 무결성 보장

스마트 그리드 네트워크

통합된 딥러닝 모델 및 체인 배포



클라우드 서버

블록체인 노드에 집계 모듈 배포

1. 로컬 데이터 암호화 후 클라우드에 업로드

2. 암호화 된 상태로 데이터 집계 후 로컬에 전송

3. 데이터 복호화 한 후, 학습시켜 모델 갱신

*이외에도 데이터 주입 공격, 가로채기 등을 방지하기 위한 인증 메커니즘 도입하여 개인 정보 보호

연구사례 종합 (?)

블록체인 상에서의 데이터 프라이버시 보호를 위해 동형암호와 영지식 증명이 활용

동형암호

- **블록 체인의 거래 데이터** (금액, 계좌 등)을 **숨긴 채 지불 가능**
- **빅데이터에 대한 집계 시스템**에 대한 **안전한 설계** 가능
- 최근 연구는 속도나 계산 오버헤드 면에서 높은 성능을 달성
  → 블록체인 **확장성을 저하시키지 X**
- **보안성 및 확장성을 모두 만족**하는 연구 사례 다수

영지식 증명

- **데이터를 노출하지 않으면서, 필요한 값을 가지고 있음을 증명 가능**
- **동형 암호화와 함께** 쓰일 수 있음
  → 암호문에 대한 연산을 하지만, 조건을 만족하는 값임을 증명

여기는 그냥 동향 발표 때 적었던 결론입니다..
블록체인에 어떻게 활용하시려고 하는지 몰라서 말씀해주시면 삭제하거나 수정하겠습니다..

안전한 집계 시스템, 신뢰성 있는 데이터 공유 및 딥러닝 학습 등 **다양한 분야에 응용 가능한 상황**

그러나 **대부분의 연구가 집계 시스템을 대상**으로 함

향후, **다양한 타겟에 대한 연구가 필요**할 것으로 생각되며, 여러 기술과 결합된다면 **더 넓은 분야에 확장** 가능할 것으로 예상