DGIST 여름인턴 3주차 주간 보고서

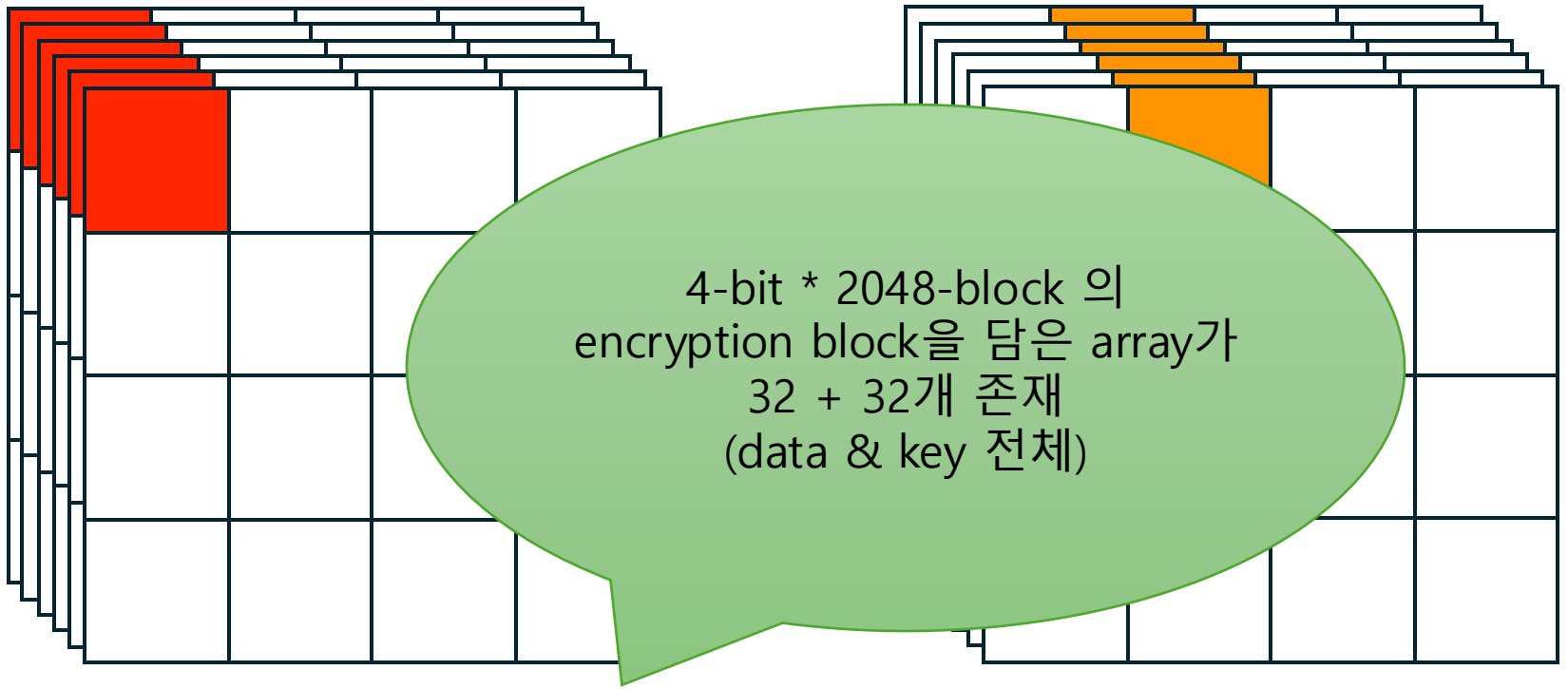
3조 권태완, 조후연, 박준석

3주차: 7/21 – 7/25 진행 상황

Desilofhe로 AES Enc 파트 구현

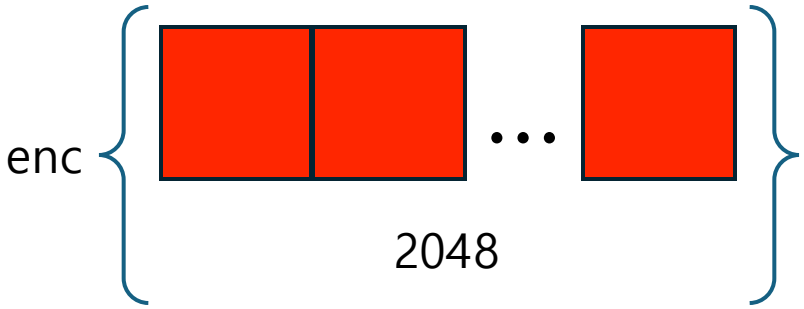
- Key Expansion
  - SubWord
  - **Rcon**
- AddRoundKey >> **XOR**
- **SubBytes**
- ShiftRows
- ~~MixColumns~~

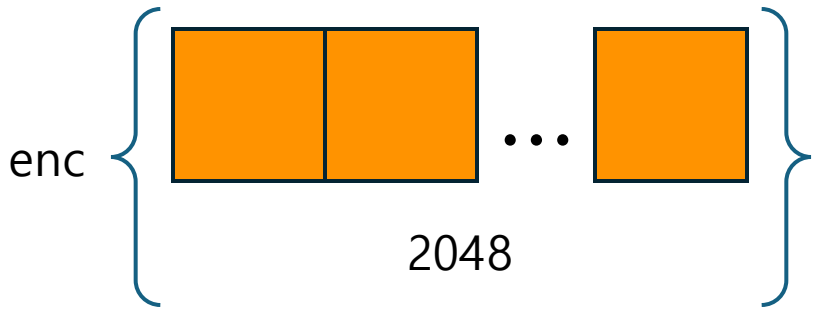
데이터 전처리 방법 1



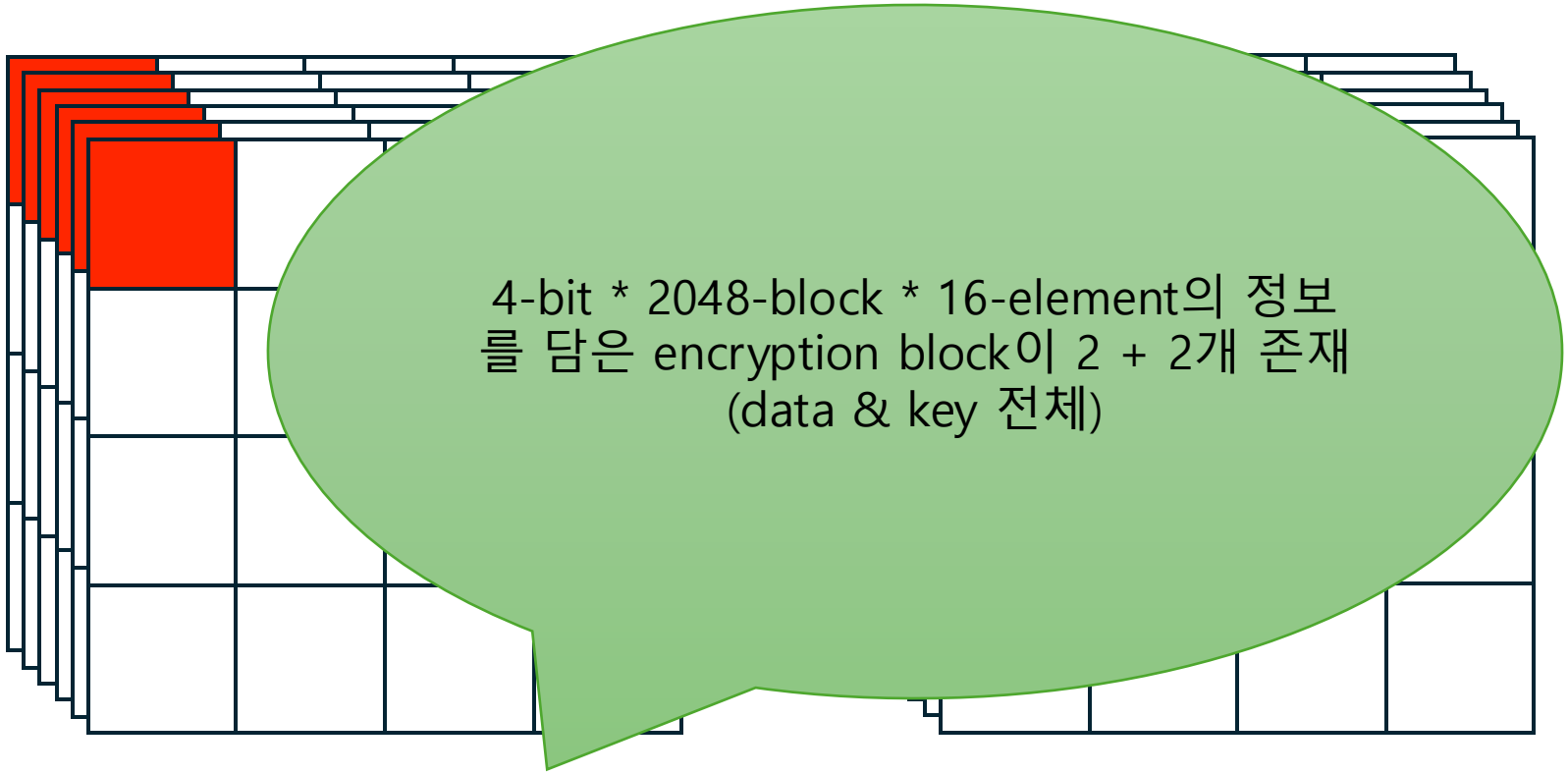
4-bit * 2048-block 의
encryption block을 담은 array가
32 + 32개 존재
(data & key 전체)

...





데이터 전처리 방법 2



4-bit * 2048-block * 16-element의 정보
를 담은 encryption block이 2 + 2개 존재
(data & key 전체)

...

enc { concat { ... } 2048 ... } 2048 ...

데이터 전처리 방법의 비교

비교	Method 1	Method 2
장점	암호화된 각각의 원소의 위치 정보를 가지고 있으므로 shiftrows, mixcolumn 연산들의 구현이 간단 (∵ np.roll, np.reshape 사용 가능)	XOR, enc, dec 연산을 한 번에 처리 가능
단점	XOR 연산에서 데이터 블록 32 by 32를 일대일로 처리하기 때문에 많은 overhead 발생	Shiftrows, Mixcolumns 연산 구현이 **매우** 복잡해짐 (masking & rotation 필수)

데이터 전처리 방법의 비교

비교	M	Method 2
장점	암호화된 각각의 와 shiftrows, mixcol (∵ np.roll, np.resha	번에 처리 가능
단점	XOR 연산에서 데이터 블록 하기 때문에 많은 overhead 발생	iftrows, Mixcolumns 연산 구현이 **매우** 복잡해짐 (masking & rotation 필수)

XOR 연산을 많이 하니까 두 번째 방법을 시도하자!

ζ- 위상에서의 연산

- XOR, SubBytes 같은 비선형 연산은 정수 위상에서 다항식 보간법으로 구현하기 어렵다는 판단.
- 동형암호는 Enc(a+b) = Enc(a) + Enc(b) 을 가능케 한다는 것에서
  Enc(f(ζ(a), ζ(b))) = f(Enc(ζ (a)),Enc(ζ(b))) 을 수행할 수 있는 보간 다항식을 구현하기로 하였음.

- 이렇게 했을 때 XOR의 다변수 다항식 보간 계수는 ifft2D를 통해 매우 sparse한 상태로 가공할 수 있었음.
- SubByte 또한 ifft2D를 사용하면 다변수 다항식의 계수를 빠르고 쉽게 얻을 수 있었음.

```python
def _pre_encode_coeffs(engine: Any) -> Tuple[np.ndarray, np.ndarray]:
    print("\nINFO: Pre-encoding S-Box coefficients for the first time...")
    pt_c_hi, pt_c_lo = np.empty_like(C_hi, dtype=object), np.empty_like(C_lo, dtype=object)
    slot_count = 16 * 2048
    for i in range(_DEG + 1):
        for j in range(_DEG + 1):
            if abs(C_hi[i, j]) >= _EPS: pt_c_hi[i, j] = engine.encode(np.full(slot_count, C_hi[i, j]))
            if abs(C_lo[i, j]) >= _EPS: pt_c_lo[i, j] = engine.encode(np.full(slot_count, C_lo[i, j]))
    print("INFO: Coefficient pre-encoding complete.")
    return pt_c_hi, pt_c_lo
```

XOR 구현 계획 – reference 참고

16차 단위근 (16th root of unity) 집합에서 다변수 다항식의 계수를 라그랑주 기법을 통해 구하는 것이 핵심

$P(x, y) = \Sigma_{p=0}^{15} \Sigma\_q_{=0}^{15} C[p,q] \cdot x^p \cdot y^\varphi$ 의 계수 C[p,q]를 구하기

$P(\zeta^a, \zeta^b) = \zeta^{ a \text{ xor } b }$ ($\forall$ a,b $\in \{0,\ldots,15\}$), $\zeta = \exp(-2\pi i/16)$ 는 16차 원시복소근

$P(\zeta^a, \zeta^b) = \Sigma_p \Sigma\_q\ C[p,q] \cdot (\zeta^a)^p \cdot (\zeta^b)^\varphi = \Sigma_p \Sigma\_q\ C[p,q] \cdot \zeta^{ap + bq}$
따라서 행렬식으로 쓰면
$f[a,b] = \Sigma_p \Sigma\_q\ C[p,q] \cdot \zeta^{ap} \cdot \zeta^{bq}$
이 식은 2차원 DFT와 동일하다.
즉, 주어진 f[a,b] 로부터 C[p.q]를 얻으려면 2차원 역-DFT를 수행하면 된다

Amortized Large Look-up Table Evaluation with Multivariate Polynomials for Homomorphic Encryption

Heewon Chung[1], Hyojun Kim[1], Young-Sik Kim[2], and Yongwoo Lee[3]

XOR 계수 생성

1. 4-bit (a^b) 행렬에 대한 2-D IFFT 수행하여 계수 만들어냄

```python
def compute_xor_mono_coeffs_zeta(M: int = 16) -> np.ndarray:
    """Return complex matrix C[p,q] (shape M×M)."""
    zeta = np.exp(-2j * np.pi / M)
    # f[a,b] = ζ^(a xor b)
    f = np.fromfunction(lambda a, b: zeta ** (a.astype(int) ^ b.astype(int)), (M, M), dtype=int)
    C = np.fft.ifft2(f)
    return C
```

2. zeta**a, zeta**b에 대한 verify 함수로 오차 확인

3. 절대값이 1e-12 이상인 값만 추려 (p,q,re,im) 형식으로 리스트화 하여 json 파일에 정리. 이때 sparse한 표현으로 저장

```python
def compress_coefficients(C: np.ndarray, tol: float = 1e-12) -> List[Tuple[int, int, float, float]]:
    M, N = C.shape
    entries: List[Tuple[int, int, float, float]] = []
    for p in range(M):
        for q in range(N):
            if abs(C[p, q]) > tol:
                entries.append((p, q, float(C[p, q].real), float(C[p, q].imag)))
    return entries
```

XOR 연산 수행

```python
def _xor_operation(engine_context, enc_alpha, enc_beta):
    engine = engine_context.engine
    relin_key = engine_context.relinearization_key
    conjugate_key = engine_context.conjugation_key

    # 1. Build power bases
    base_x = build_power_basis(engine, enc_alpha, relin_key, conjugate_key)
    base_y = build_power_basis(engine, enc_beta,  relin_key, conjugate_key)

    # 2. Pre-encoded polynomial coefficients
    coeff_pts = _get_coeff_plaintexts(engine)

    # 3. Evaluate polynomial securely
    cipher_res = engine.multiply(enc_alpha, 0.0)

    for (i, j), coeff_pt in coeff_pts.items():
        term = engine.multiply(base_x[i], base_y[j], relin_key)
        term_res = engine.multiply(term, coeff_pt)
        cipher_res = engine.add(cipher_res, term_res)

    return cipher_res
```

XOR 연산 수행 - details

1. make_power_basis 연산 사용 및 단위원 구조의 특성을 사용하여 빠른 거듭제곱 수행

```
# Positive powers 1..8
pos_basis = engine.make_power_basis(ct, 8, relin_key)  # list length 8

basis: Dict[int, object] = {}
basis[0] = ones_cipher(engine, ct)

for idx, c in enumerate(pos_basis, start=1):
    basis[idx] = c  # exponents 1..8

# Negative powers: ct^-k  (k=1..7) → ct^(16-k)
for k in range(1, 8):
    conj_ct = engine.conjugate(pos_basis[k - 1], conj_key)  # ct^-k
    basis[16 - k] = conj_ct  # exponents 15..9
```

XOR 연산 수행 - details

2. term 계산 및 계수와의 곱 수행. sparse한 성질을 활용하여 적은 반복

```
for (i, j), coeff_pt in coeff_pts.items():
    term = engine.multiply(base_x[i], base_y[j], relin_key)
    term_res = engine.multiply(term, coeff_pt)
    cipher_res = engine.add(cipher_res, term_res)
```

XOR 연산 결과

```
enc_alpha = engine.encrypt(alpha, public_key, level=5)
enc_beta = engine.encrypt(beta, public_key, level=5)

# 2. Evaluate XOR operation
start_time = time.time()
cipher_res = _xor_operation(engine_context, enc_alpha, enc_beta)

bootstrap_ct = engine.bootstrap(cipher_res, relinearization_key, conjugation_key, bootstrap_key)
end_time = time.time()
print(f"XOR time taken: {end_time - start_time} seconds")

start_time = time.time()
```

level 5 소모,
level=7, thread개수 16, bootstrapping 사용 기준
xor에 3.6초 bootstrap에 6초

총 xor 연산 시간 9.6초

SubBytes

Problem: AES의 유일한 비선형(Non-linear) 연산으로,
단순 덧셈/곱셈만 지원하는 FHE에서 구현하기 가장 어려운 부분 중 하나

도전 과제: 8비트 입력을 받아 8비트 출력을 내는 복잡한 Lookup Table(LUT)을 암호문 상태에서 100% 정확하게 구현해야 함.

핵심 아이디어: '다변수 다항식 LUT' 방식을 채택.

Amortized Large Look-up Table Evaluation with Multivariate Polynomials for Homomorphic Encryption

Heewon Chung[1], Hyojun Kim[1], Young-Sik Kim[2], and Yongwoo Lee[3]

SubBytes - Paterson-Stockmeyer 알고리즘 적용

- 차수가 높은 다항식 P(x) (예: 15차)를 직접 계산 X
- 대신, 새로운 변수 y=x^4 를 도입하여 저차 다항식으로 재구성.
- 이 방식은 "baby-steps" (계수 다항식 계산)와 "giant-steps" (y=x^k를 곱하는 연산)로 나누어, 전체 곱셈 횟수를 O(n) -> O(n^(1/2)) 로 줄임.

- 이를 통해 15차 다항식 평가에 필요한 FHE 곱셈 깊이 최소화

ALGORITHM B.

$$
\begin{aligned}
a_{km-1}x^{km-1} + \cdots + a_1x + a_0 \\
&= (\cdots((a_{km-1}x^{k-1} + \cdots + a_{k(m-1)})x^k \\
&\quad + a_{k(m-1)-1}x^{k-1} + \cdots + a_{k(m-2)})x^k \\
&\qquad \cdot \quad \cdot \quad \cdot \\
&\quad + a_{2k-1}x^{k-1} + \cdots + a_{k+1}x + a_k)x^k \\
&\quad + a_{k-1}x^{k-1} + \cdots + a_1x + a_0.
\end{aligned}
$$

Reference: M. S. Paterson and L. J. Stockmeyer. "On the number of nonscalar multiplications necessary to evaluate polynomials." SIAM J. Comput., 1973.

SubBytes – 주의사항

```
for m in range(num_chunks):
    baby_step_terms_hi, baby_step_terms_lo = [], []
    for i in range(k):
        inner_terms_hi, inner_terms_lo = [], []
        for j in range(_DEG + 1):
            idx_y = i + m * k
            ct_hi_pow = hi_basis[j]
            if abs(C_hi[j, idx_y]) >= _EPS:
                inner_terms_hi.append(engine.multiply(ct_hi_pow, pt_c_hi[j, idx_y]))
            if abs(C_lo[j, idx_y]) >= _EPS:
                inner_terms_lo.append(engine.multiply(ct_hi_pow, pt_c_lo[j, idx_y]))

        inner_sum_hi = _sum_terms_tree(inner_terms_hi, engine_context)
        inner_sum_lo = _sum_terms_tree(inner_terms_lo, engine_context)

        ct_lo_pow = lo_basis[i]
        baby_step_terms_hi.append(engine.multiply(inner_sum_hi, ct_lo_pow, rlk))
        baby_step_terms_lo.append(engine.multiply(inner_sum_lo, ct_lo_pow, rlk))

    chunk_results_hi.append(_sum_terms_tree(baby_step_terms_hi, engine_context))
    chunk_results_lo.append(_sum_terms_tree(baby_step_terms_lo, engine_context))

final_result_hi = chunk_results_hi[0]
final_result_lo = chunk_results_lo[0]
for m in range(1, num_chunks):
    ct_lo_pow_k = lo_basis[m * k]
    term_hi = engine.multiply(chunk_results_hi[m], ct_lo_pow_k, rlk)
    final_result_hi = engine.add(final_result_hi, term_hi)
    term_lo = engine.multiply(chunk_results_lo[m], ct_lo_pow_k, rlk)
    final_result_lo = engine.add(final_result_lo, term_lo)

# 지금까지의 결과는 정수 형태이므로, 이를 ζ 형태로 변환한다.
final_result_hi = int_cipher_to_zeta_cipher(engine_context, final_result_hi)
final_result_lo = int_cipher_to_zeta_cipher(engine_context, final_result_lo)

return final_result_hi, final_result_lo
```

- 우리가 만든 코드의 경우 입력을 zeta 위상에서 받지만, 출력은 정수 위상으로 변환시킨다. 그래서 이를 다시 zeta 위상으로 변환하는 작업을 마지막에 거쳐야한다.

- $\zeta \rightarrow \mathbb{Z} \rightarrow \zeta$

구현 모듈

- xor 과 SubBytes 를 바탕으로 key scheduling에 필요한 RotWord, SubWord, RconXor, Xor 구현을 완료하였으며, 오버헤드를 최소화 할 수 있는 key scheduling 방법을 찾아 전체 구현은 거의 완료하였음

- ShiftRows 단독 연산을 구현 완료, MixColumns는 구현에 있어 난항을 겪고 있음.

- 이번주 목요일 전까지는 전체 연산에 대한 구현을 완료하고 시간이 되는대로 복호화에도 구현을 시도하게 될 것.

4주차: -07/31

Desilofhe로 AES Enc 파트 구현

- MixColumns

Desilofhe로 AES Dec 파트 구현

…

끝