Introduction to CKKS

(a.k.a. Approximate Homomorphic Encryption)

Yongsoo Song

Private AI Bootcamp

Microsoft Research, Dec 02

What is CKKS?

Plain Computation

- bool, int (uint64), modulo p
- double (float)

Encrypted Computation

BGV, BFV, TFHE

CKKS

[Cheon-Kim-Kim-Song, Asiacrypt'17] Homomorphic Encryption for Arithmetic of Approximate Numbers (HEAAN)

[Cheon-Han-Kim-Kim-Song, Eurocrypt'18] Bootstrapping for Approximate Homomorphic Encryption

[Cheon-Han-Kim-Kim-Song, SAC'18] A Full RNS Variant of Approximate Homomorphic Encryption

[Chen-Chillotti-Song, Eurocrypt'19] Improved Bootstrapping for Approximate Homomorphic Encryption

…

SEAL/native/examples/4_ckks_basic.cpp

Compute $F(x) = \pi * x^3 + 0.4 * x + 1$ for $x = x_1, x_2, \ldots$

Approximate Arithmetic

Floating-point representation

$$1.01011 = \underbrace{101011}_{\text{significand}} * \underbrace{2^{-5}}_{\text{scaling factor (base}^{\text{exponent}})}$$

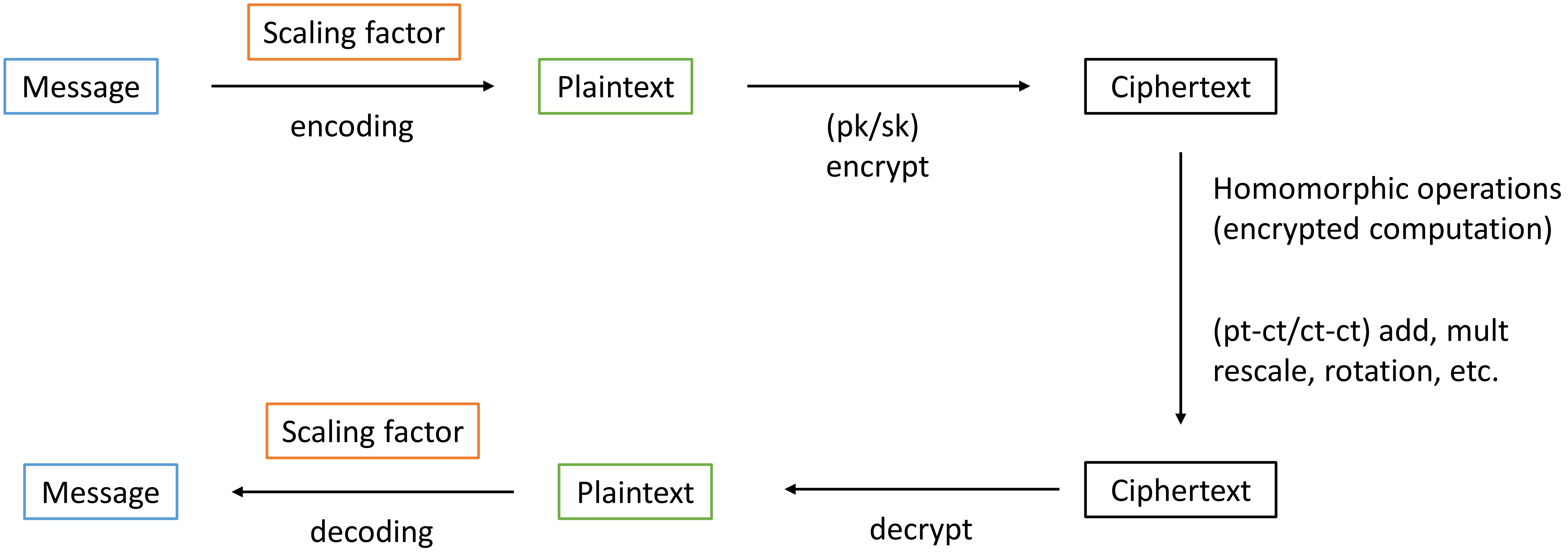
- Floating-point Arithmetic (double, IEEE 754)
  - The significand is assumed to have a binary point to the right of the leftmost bit

$$\left(101011 * 2^{-5}\right) * \left(110111 * 2^{-5}\right) = 100100111101 * 2^{-10} \approx 100101 * 2^{-4}$$

- Fixed-point Arithmetic : more suitable for HE
  - The scaling factor is the same for all values of the same type, and does not change during the entire computation

$$\left(101011 * 2^{-5}\right) * \left(110111 * 2^{-5}\right) = 100100111101 * 2^{-10} \approx 1001010 * 2^{-5}$$

Algorithms in CKKS



Encoding & Decoding

$\mathbb{C}^{n/2} \times \mathbb{R}$ $\underset{\text{decoding}}{\overset{\text{encoding}}{\rightleftarrows}}$ $R = \mathbb{Z}[X]/(X^n + 1)$

z_1

z_2

$\vdots$

$z_{n/2}$

Δ

Message vector Scaling factor

$m(X) \approx$

$\Delta \cdot z_1$

$\Delta \cdot z_2$ i-th slot of plaintext

$\vdots$

$\Delta \cdot z_{n/2}$

Plaintext (Encoded message)

```
double scale = pow(2.0, 40);
```

$m(\zeta_j) \approx \Delta \cdot z_j$ for some roots ζ_j of $X^n + 1 = 0$

Toy example : $n = 4$

$(z_1, z_2) = (1.2 - 3.4i, 5.6 + 7.8i),\ \ \Delta = 2^7 \quad \mapsto \quad m(X) = 435 - 706X + 282X^2 - 308X^3$

$$m(\zeta_1) = 2^7(1.1988\ldots + i * 3.3984\ldots), \quad m(\zeta_2) = 2^7(5.5970\ldots + i * 7.8047\ldots)$$

Encoding of a vector

$$F(x) = \pi * x^3 + 0.4 * x + 1$$

```
vector<double> input;
input.reserve(slot_count);
double curr_point = 0;
double step_size = 1.0 / (static_cast<double>(slot_count) - 1);
for (size_t i = 0; i < slot_count; i++, curr_point += step_size)
{
    input.push_back(curr_point);
}
```

$$\text{slot\_count} = n/2$$

$$\text{input} = (x_1, \dots, x_{n/2})$$

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

$$\text{x\_plain} \approx \begin{array}{|c|} \hline \Delta \cdot x_1 \\ \hline \Delta \cdot x_2 \\ \hline \vdots \\ \hline \Delta \cdot x_{n/2} \\ \hline \end{array}$$

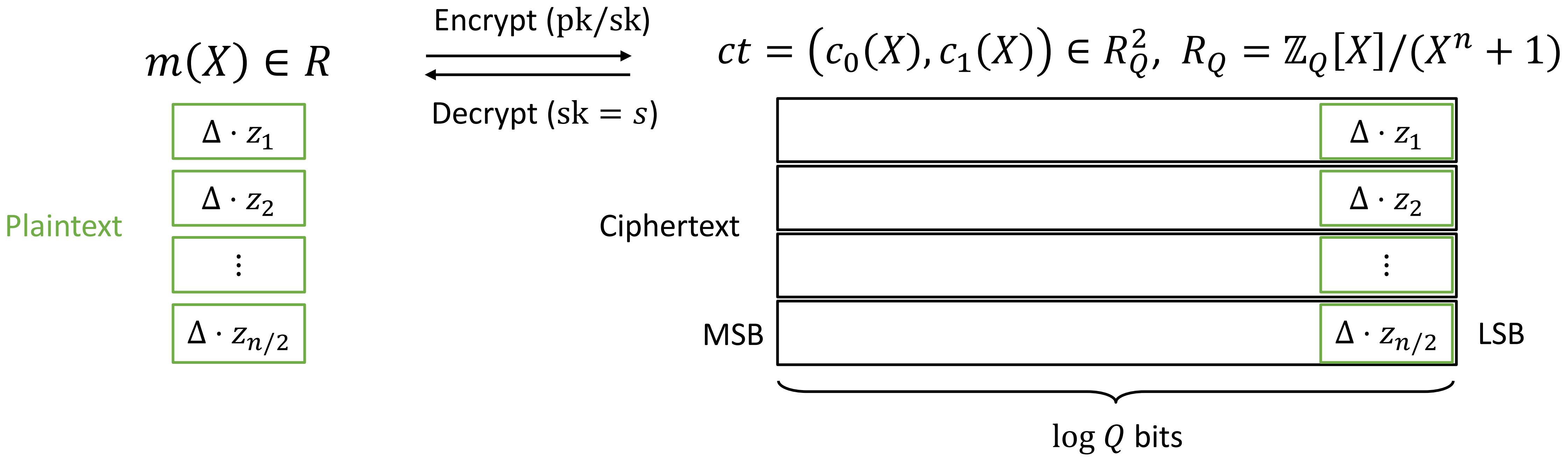
Encoding of a scalar

$$F(x) = \pi * x^3 + 0.4 * x + 1$$

```
Plaintext plain_coeff3, plain_coeff1, plain_coeff0;
encoder.encode(3.14159265, scale, plain_coeff3);
encoder.encode(0.4, scale, plain_coeff1);
encoder.encode(1.0, scale, plain_coeff0);
```

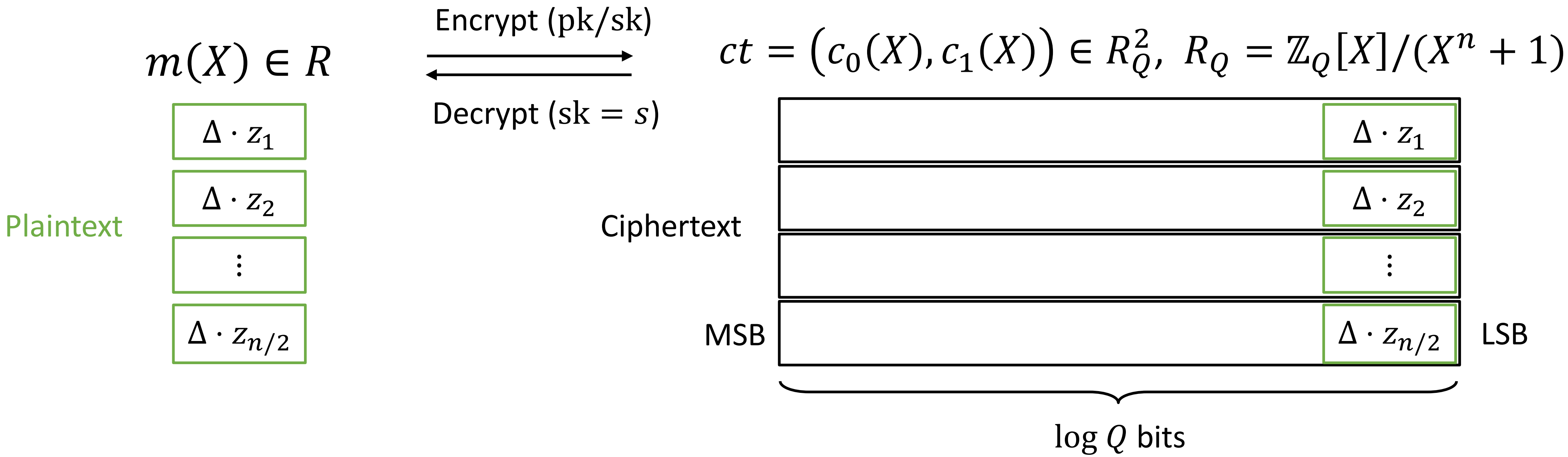
$$\text{plain\_coeff3} \approx \begin{bmatrix} \Delta \cdot \pi \\ \Delta \cdot \pi \\ \vdots \\ \Delta \cdot \pi \end{bmatrix}$$

Encrypt & Decrypt



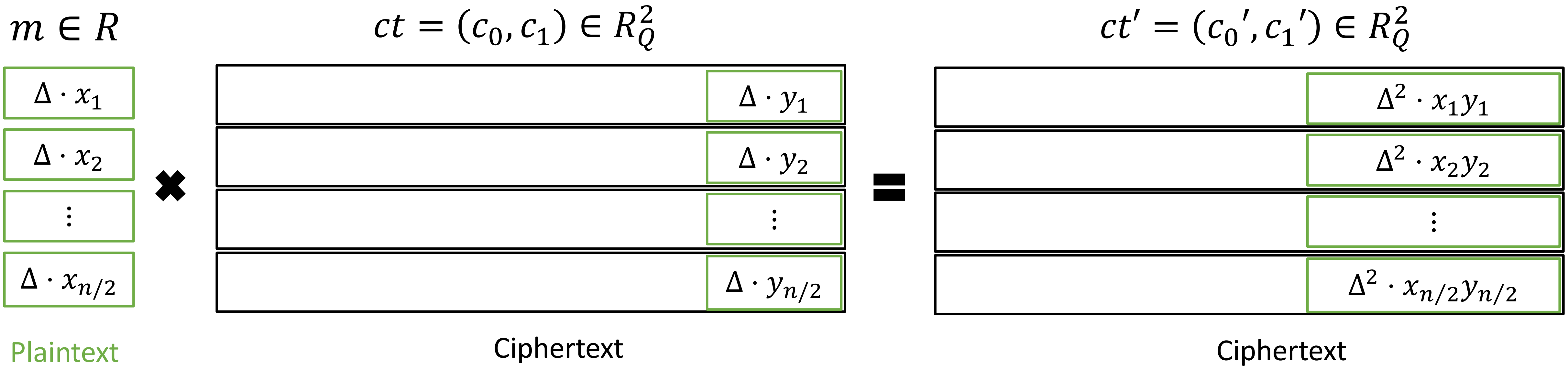
- Encrypt: $m(X) \mapsto ct = \big(c_0(X), c_1(X)\big) \in R_Q^2$ such that $c_0 + c_1 s \approx m \ (mod\ Q)$
  - Correctness: $\|m\| < Q$.
  - Notation: $ct(S) = c_0 + c_1 S \in R_Q[S]$
- Warning: An encryption of m is not decrypted to exactly m but $m + e$ for some error e such that $|e| < bound$

Encrypt & Decrypt



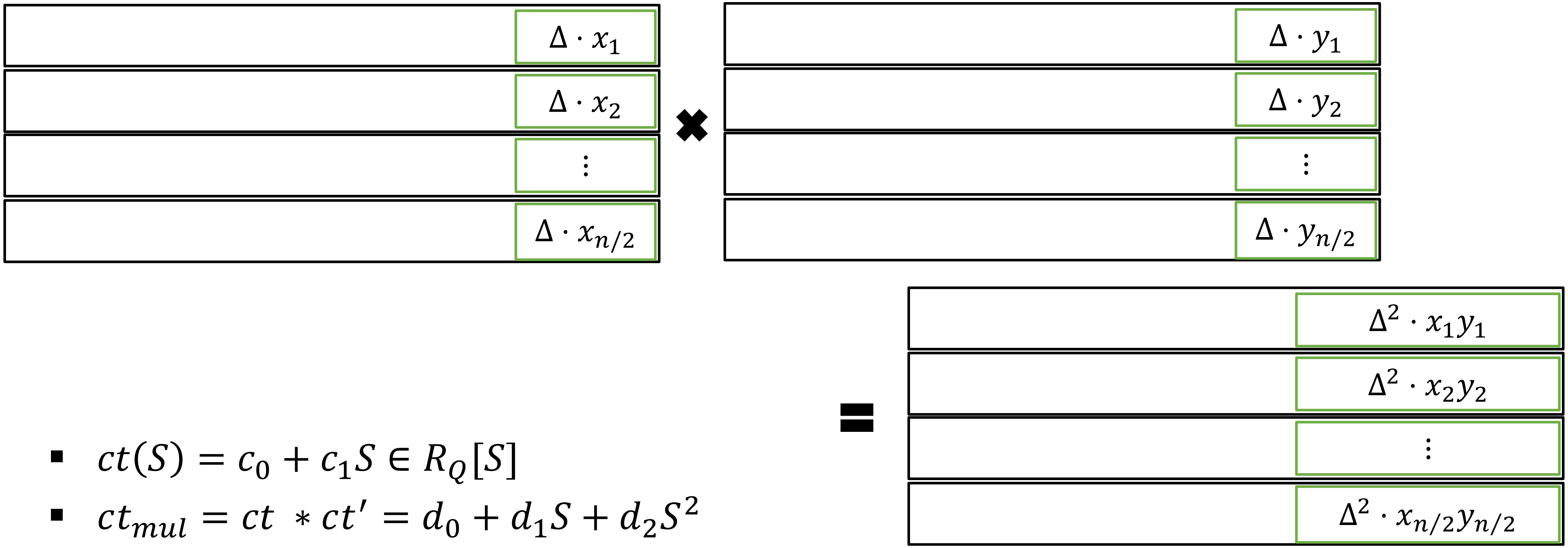

```
Ciphertext x1_encrypted;
encryptor.encrypt(x_plain, x1_encrypted);
```

Plain – Cipher mult



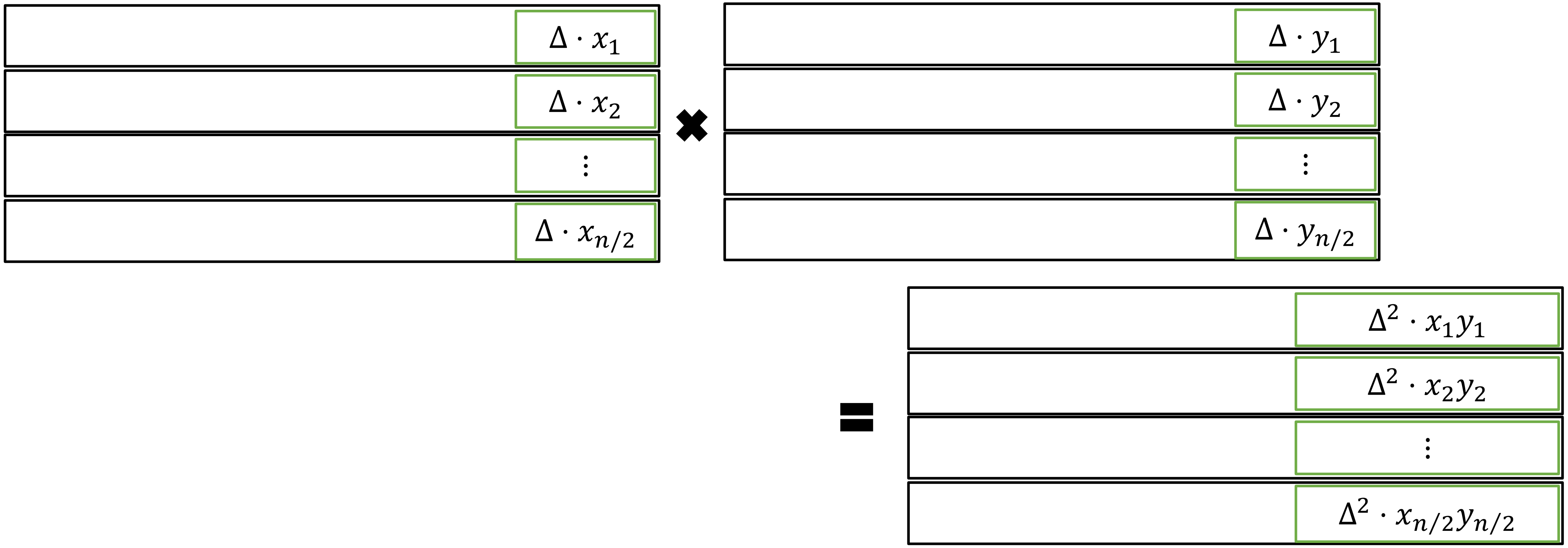

```
evaluator.multiply_plain(x1_encrypted, plain_coeff3, x1_encrypted_coeff3);
```

Cipher – Cipher mult & Relinearization



- $ct(S) = c_0 + c_1 S \in R_Q[S]$
- $ct_{mul} = ct \ * ct' = d_0 + d_1 S + d_2 S^2$
- $relinearize$: $ct_{mul} \mapsto ct'_{mul} = d'_0 + d'_1 S$
    - change the format of ciphertext while (almost) preserving encrypted plaintext
    - (almost always) performed after cipher-cipher multiplication

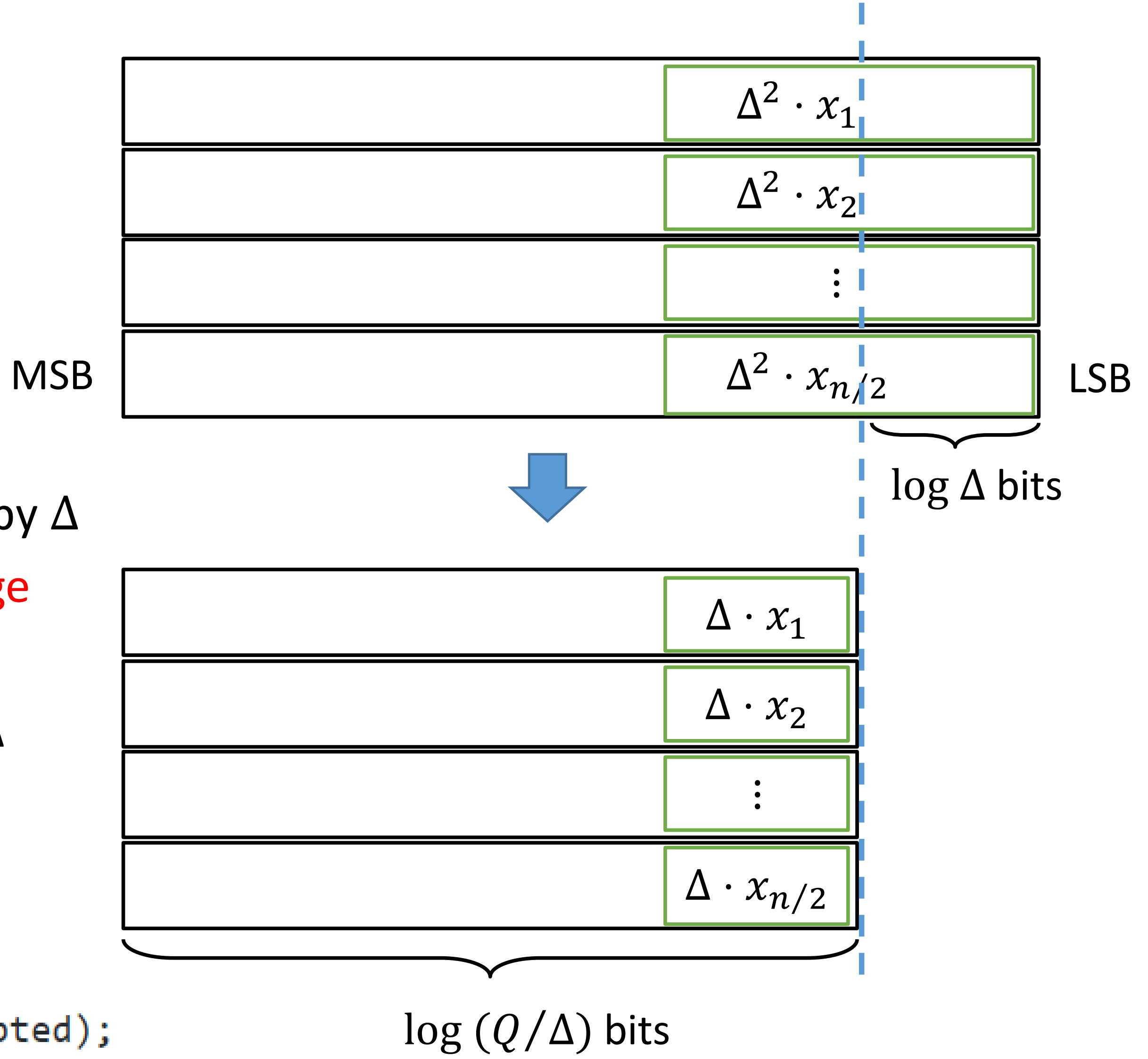
Cipher – Cipher mult & Relinearization




```
evaluator.square(x1_encrypted, x3_encrypted);
evaluator.relinearize_inplace(x3_encrypted, relin_keys);
```

Rescale

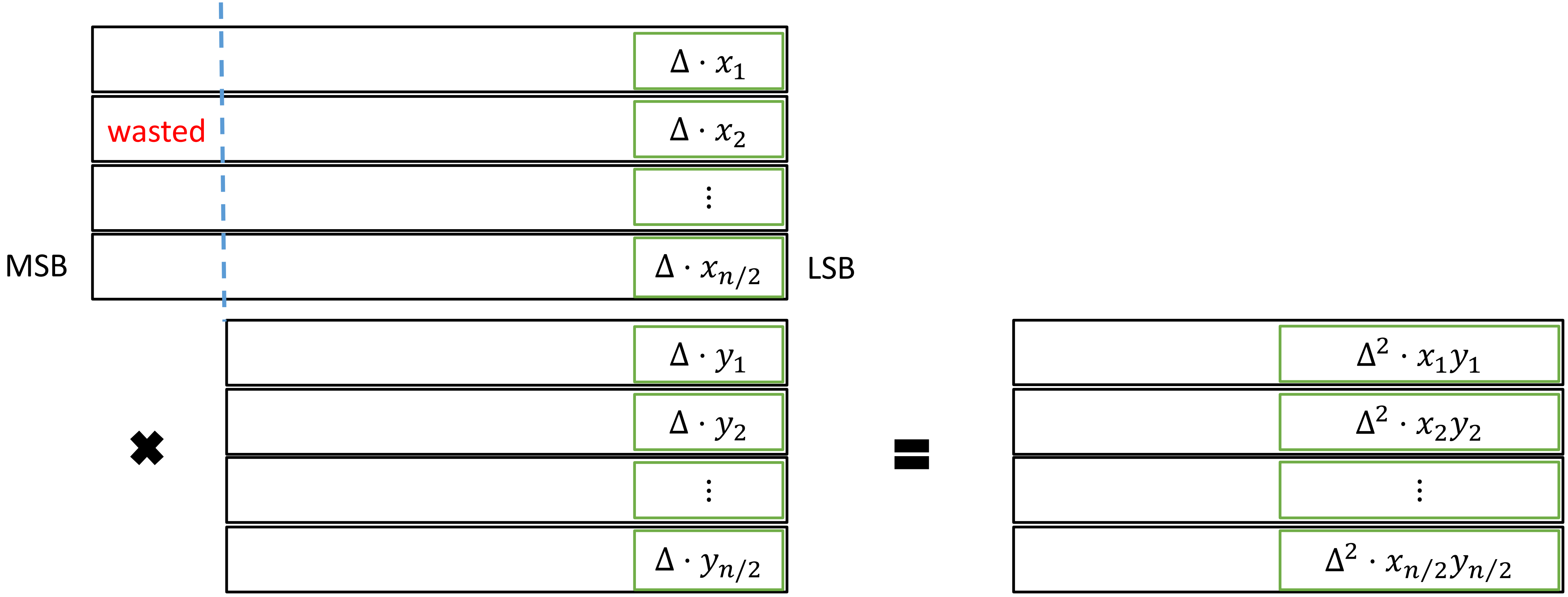
- Usually performed after multiplication
- Rescale $ct \in R_Q^2 \mapsto ct' \in R_{Q'}^2$, for $Q' < Q$
  - Ciphertext & plaintext are (approximately) divided by Δ
  - Input & output are encryptions of the same message with different representations
  - Scaling factor $\Delta^2 \mapsto \Delta$, ctx modulus $Q \mapsto Q' = Q/\Delta$



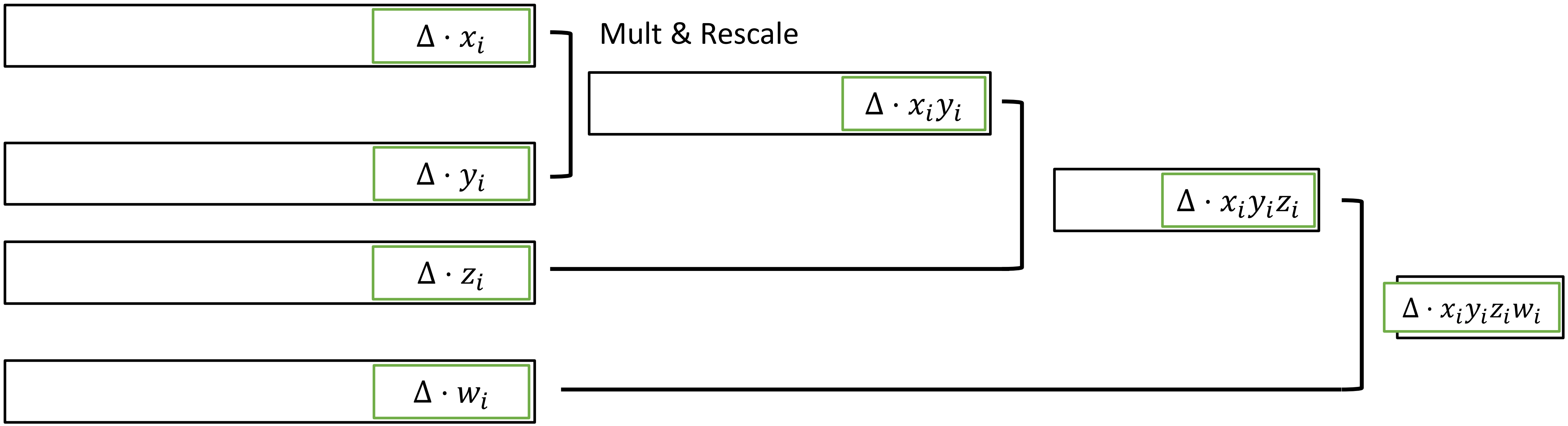

```
evaluator.rescale_to_next_inplace(x3_encrypted);

evaluator.rescale_to_next_inplace(x1_encrypted_coeff3);
```

Add/Mult between ctxs with different moduli

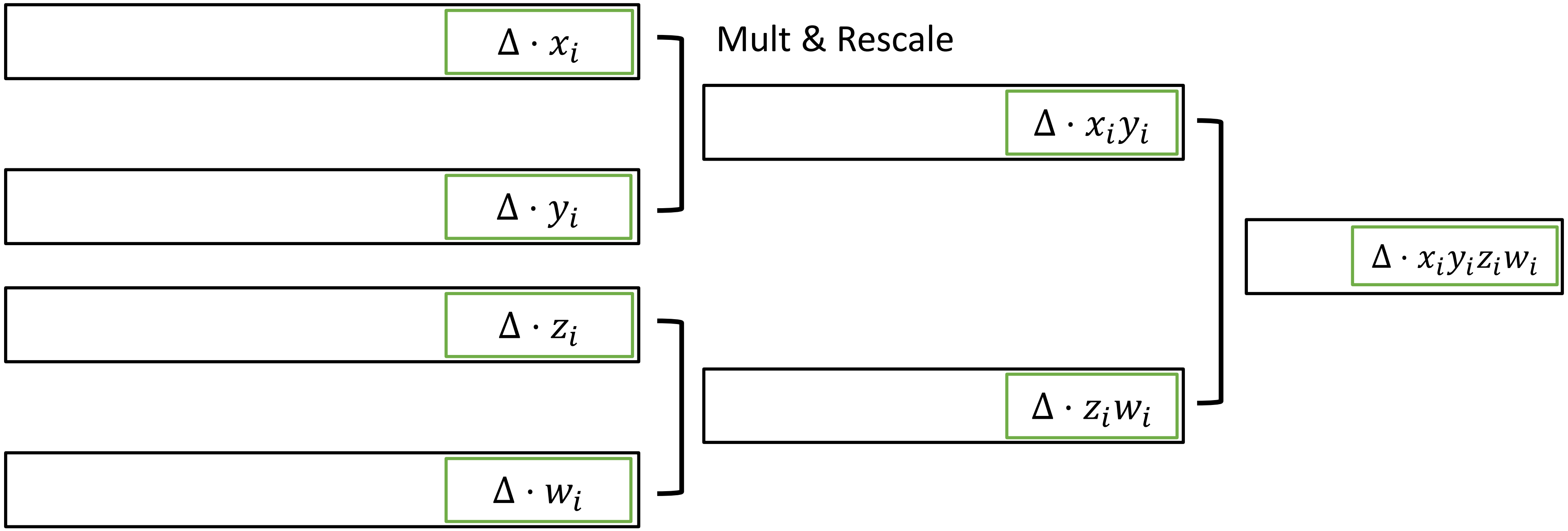


$((xy)z)w$ vs $(xy)(zw)$



Ciphertext modulus : $Q \mapsto Q' = Q/\Delta \mapsto Q'' = Q/\Delta^2 \mapsto Q''' = Q/\Delta^3$

$((xy)z)w$ vs $(xy)(zw)$



Ciphertext modulus : $Q \quad \mapsto \quad Q' = Q/\Delta \quad \mapsto \quad Q'' = q/\Delta^2$

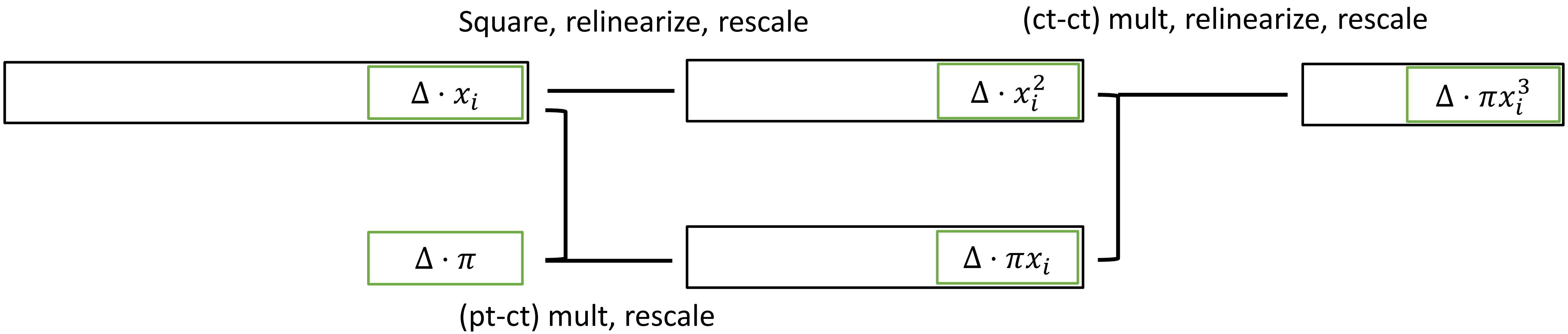
Ciphertext level

- $Q = q_0 \cdot \Delta^L$
  - q_0 : base modulus (which is usually set to be >> Δ)
  - $Q_\ell = q_0 \cdot \Delta^\ell$
  - "Ciphertext level is ℓ" = "Ciphertext modulus is Q_ℓ"

- Level = Computational capability
  - Ciphertext level decreases as the computation progresses
  - No more (multiplicative) arithmetic is allowed for 0-level ciphertexts but decryption

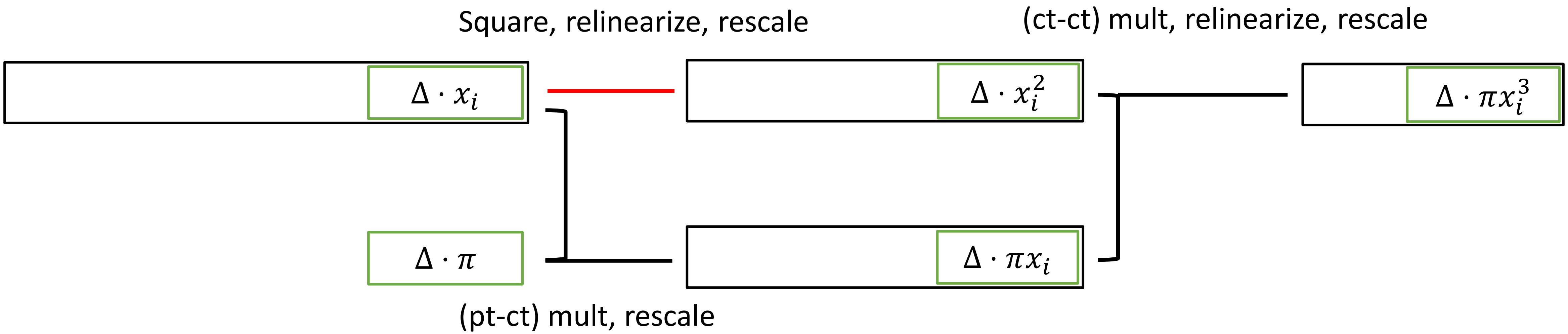
- <Multiplication> $(ct, \ell, \Delta), (ct', \ell, \Delta) \mapsto (ct_{mul}, \ell, \Delta^2)$ product of plaintexts & scaling factors
- <Relinearization> $\left(ct_{mul}, \ell, \Delta^2\right) \mapsto (ct'_{mul}, \ell, \Delta^2)$
- <Rescale> $\left(ct'_{mul}, \ell, \Delta^2\right) \mapsto (ct''_{mul}, \ell - 1, \Delta)$ change the scale (plaintext)

$$F(x) = \pi * x^3 + 0.4 * x + 1$$



Square, relinearize, rescale
(ct-ct) mult, relinearize, rescale
$\Delta \cdot x_i$
$\Delta \cdot x_i^2$
$\Delta \cdot \pi x_i^3$
$\Delta \cdot \pi$
$\Delta \cdot \pi x_i$
(pt-ct) mult, rescale

$$F(x) = \pi * x^3 + 0.4 * x + 1$$

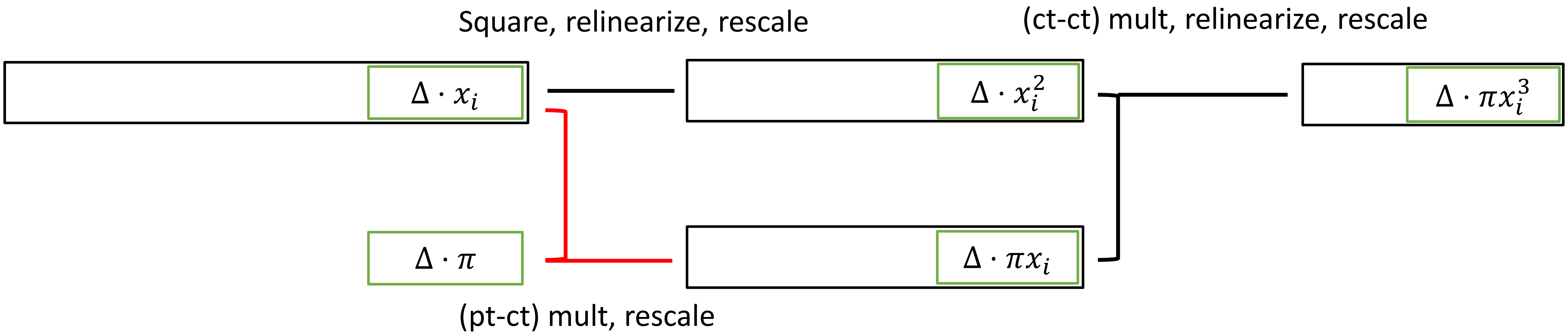


Ciphertext modulus : $Q \mapsto Q' = Q/\Delta$

```
evaluator.square(x1_encrypted, x3_encrypted);
evaluator.relinearize_inplace(x3_encrypted, relin_keys);

evaluator.rescale_to_next_inplace(x3_encrypted);
```

$$F(x) = \pi * x^3 + 0.4 * x + 1$$

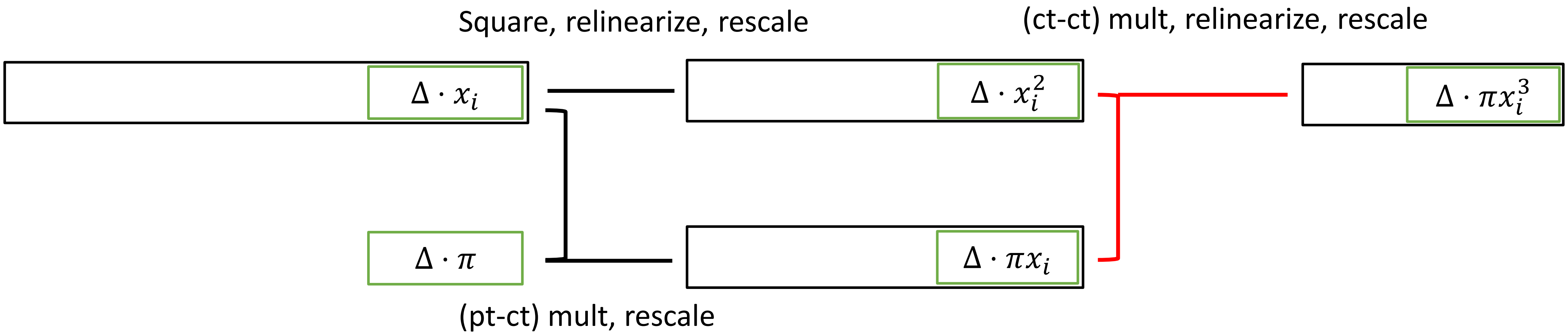


Ciphertext modulus : $Q \quad \mapsto \quad Q' = Q/\Delta$

```
evaluator.multiply_plain(x1_encrypted, plain_coeff3, x1_encrypted_coeff3);

evaluator.rescale_to_next_inplace(x1_encrypted_coeff3);
```

$$F(x) = \pi * x^3 + 0.4 * x + 1$$



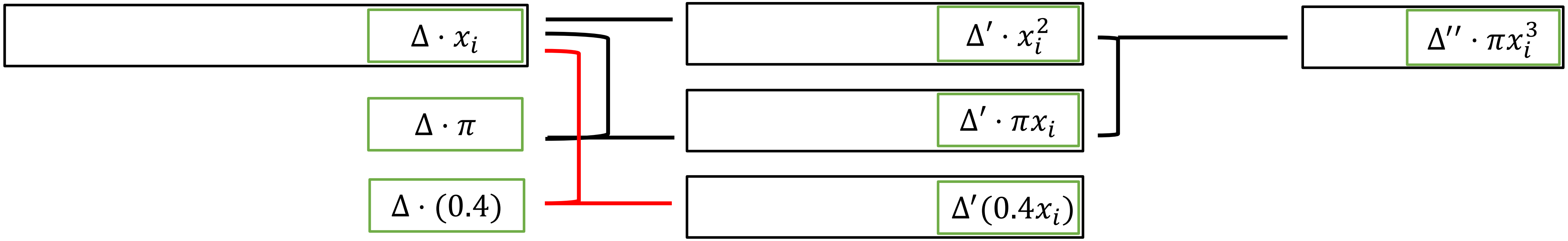
Ciphertext modulus : $Q \mapsto Q' = Q/\Delta \mapsto Q'' = Q/\Delta^2$

```
evaluator.multiply_inplace(x3_encrypted, x1_encrypted_coeff3);
evaluator.relinearize_inplace(x3_encrypted, relin_keys);

evaluator.rescale_to_next_inplace(x3_encrypted);
```

Theory to Practice

- HE parameter: $\log Q$ > (Depth of circuit L) * ($\log \Delta$)
  - Arithmetic operations modulo a large integer are very expensive
  - Set $Q_\ell = q_0 \cdot q_1 q_2 \ldots q_\ell$, $1 \le \ell \le L$ for distinct primes $q_1, \ldots, q_L$ and use the CRT representation
- <Rescale> ciphertext modulus from Q_ℓ down to $Q_{\ell-1} = Q_\ell / q_\ell$
  - The scaling factor is divided by $q_\ell \neq \Delta$
  - Updates the scaling factor of a ciphertext along the computation (double ciphertext.scale())

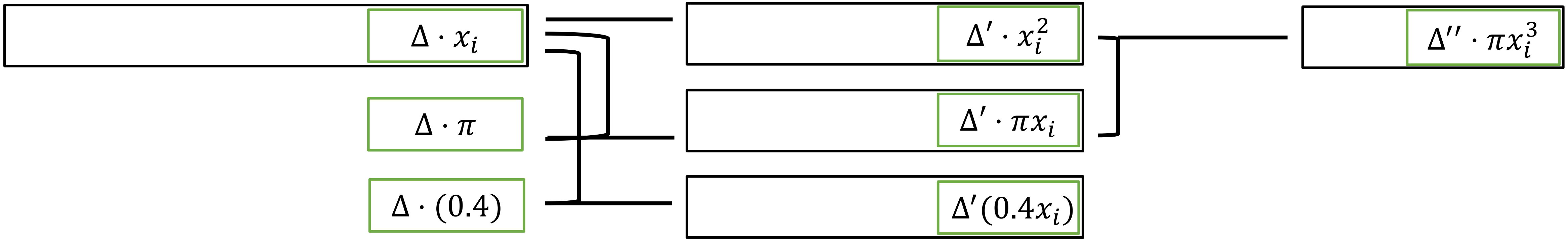


Ciphertext modulus:	Q_2	$Q_1 = Q_2/q_2$	$Q_0 = Q_1/q_1$
Plaintext scaling factor:	$\Delta = 2^{40}$	$\Delta' = \Delta^2/q_2$	$\Delta'' = {\Delta'}^2/q_1$

Theory to Practice

- How can we add ciphertexts with different scales?
  - Simple: set ciphertext.scale() = Δ ($q_\ell \approx \Delta$ for the stability of scaling factors $\Delta \approx \Delta' \approx \Delta''$)
  - Complex (accurate): TMI

- Precision?
  - Basic operations: $\log \Delta - \log(\text{noise})$ bits of precision, $\log(\text{noise}) \approx 10{\sim}15$
  - Complex circuit: need for numerical analysis



Ciphertext modulus:	Q_2	$Q_1 = Q_2/q_2$	$Q_0 = Q_1/q_1$
Plaintext scaling factor:	$\Delta = 2^{40}$	$\Delta' = \Delta^2/q_2$	$\Delta'' = {\Delta'}^2/q_1$

Parameter setting

$$F(x) = \pi * x^3 + 0.4 * x + 1$$

- Modulus switching
  - Special prime (modulus) : q_{L+1}
  - public key, relinearization key, rotation key : modulus $Q_{L+1} = q_0 \cdot q_1 \dots q_L \cdot q_{L+1}$
  - Requirement: $q_{L+1} \geq q_i, \forall i$

```
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 }));

```

$n = 2^{13}$

(security: $\log Q_{L+1} = \sum_i \log q_i \leq 218$)

$\lceil \log q_0 \rceil = 60$

$\lceil \log q_1 \rceil = \lceil \log q_2 \rceil = 40 = \log \Delta$

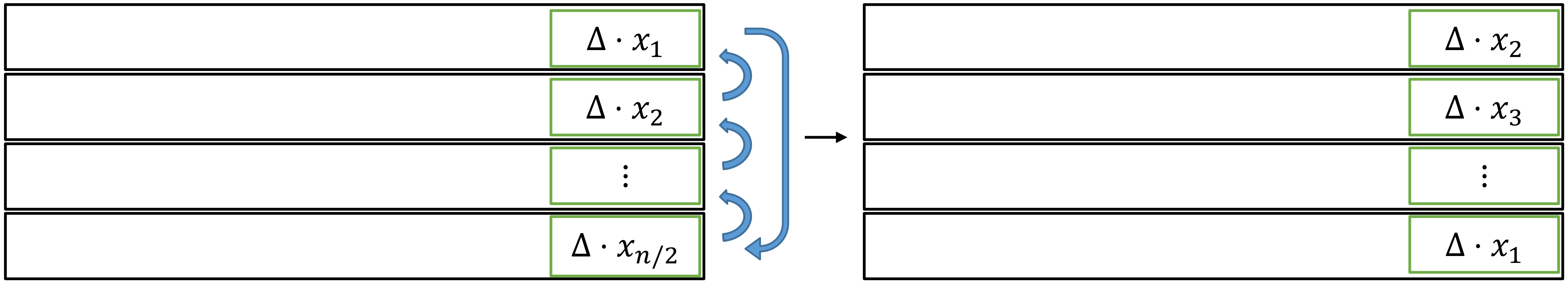
$\lceil \log q_3 \rceil = 60$

Level 2 HE system, (roughly) 30-bit precision

Correct decryption if $res < 2^{20}$

+ Rotation (slot shifting)

$ct = (c_0, c_1) \in R_Q^2$ $\qquad$ $ct' = ({c_0}', {c_1}') \in R_Q^2$



Ciphertext $\qquad$ Ciphertext

5_rotation.cpp

Bootstrapping

- Raise the level of a ciphertext
  - Recover the computational capability
  - Overcome the limitation of leveled HE system
  - Very expensive (seconds ~ minutes)