Privacy-Preserving Computation with Fully Homomorphic Encryption

31 October // Washington, DC, USA

Robert Podschwadt rpodschw@odu.edu

Daniel Takabi Takabi@odu.edu

About Us

Robert Podschwadt

Daniel Takabi

Old Dominion University

Norfolk, VA

OLD DOMINION UNIVERSITY

About Us

Robert Podschwadt, Ph.D

Research Assistant Professor

School of Cybersecurity

Old Dominion University

Norfolk, VA

Email: rpodschw@odu.edu

About Us

Daniel Takabi, Ph.D

Professor, Department of Electrical & Computer Engineering

Batten Endowed Chair in Cybersecurity

Director, School of Cybersecurity

Director, Coastal Virginia Center for Cyber Innovation

Old Dominion University

- designated a National Center of Academic Excellence in Cyber Research (CAE-R) by NSA

Norfolk, VA

Email: takabi@odu.edu

Our Work on FHE

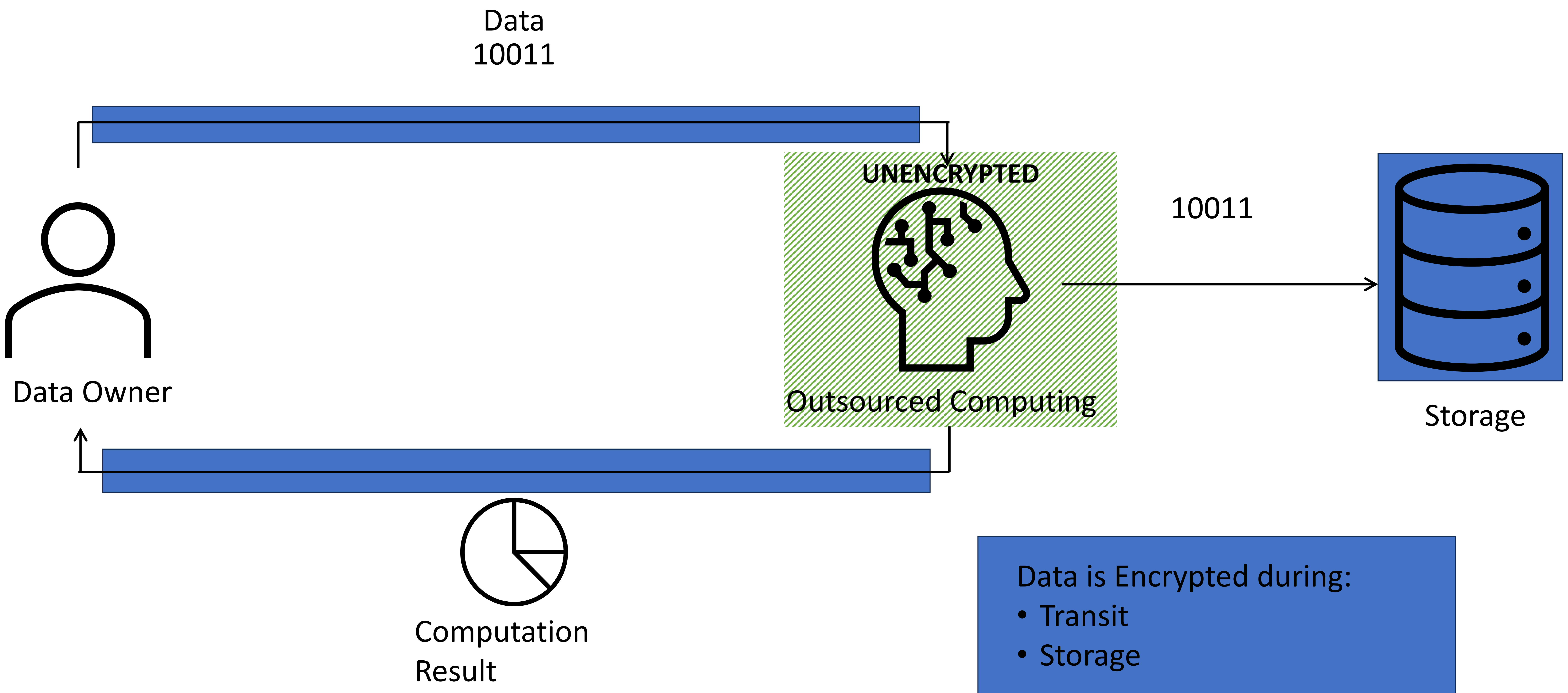
- Since 2015
- Primarily focused on privacy-preserving machine learning (PPML)
  - Convolutional Neural Networks (CNN)
  - Recurrent Neural Networks (RNN)
  - GPU Acceleration
  - Image Transformers
  - Memory Efficiency
  - ML pruning/ compression
  - Ongoing Work: compiler tooling, applications & use cases
- Published 20+ papers

Outline

1. Introduction to Fully Homomorphic Encryption (FHE)
2. Mathematical Background
3. FHE schemes and their Properties
4. CKKS: The Details
5. Computation with FHE
6. Hands-on

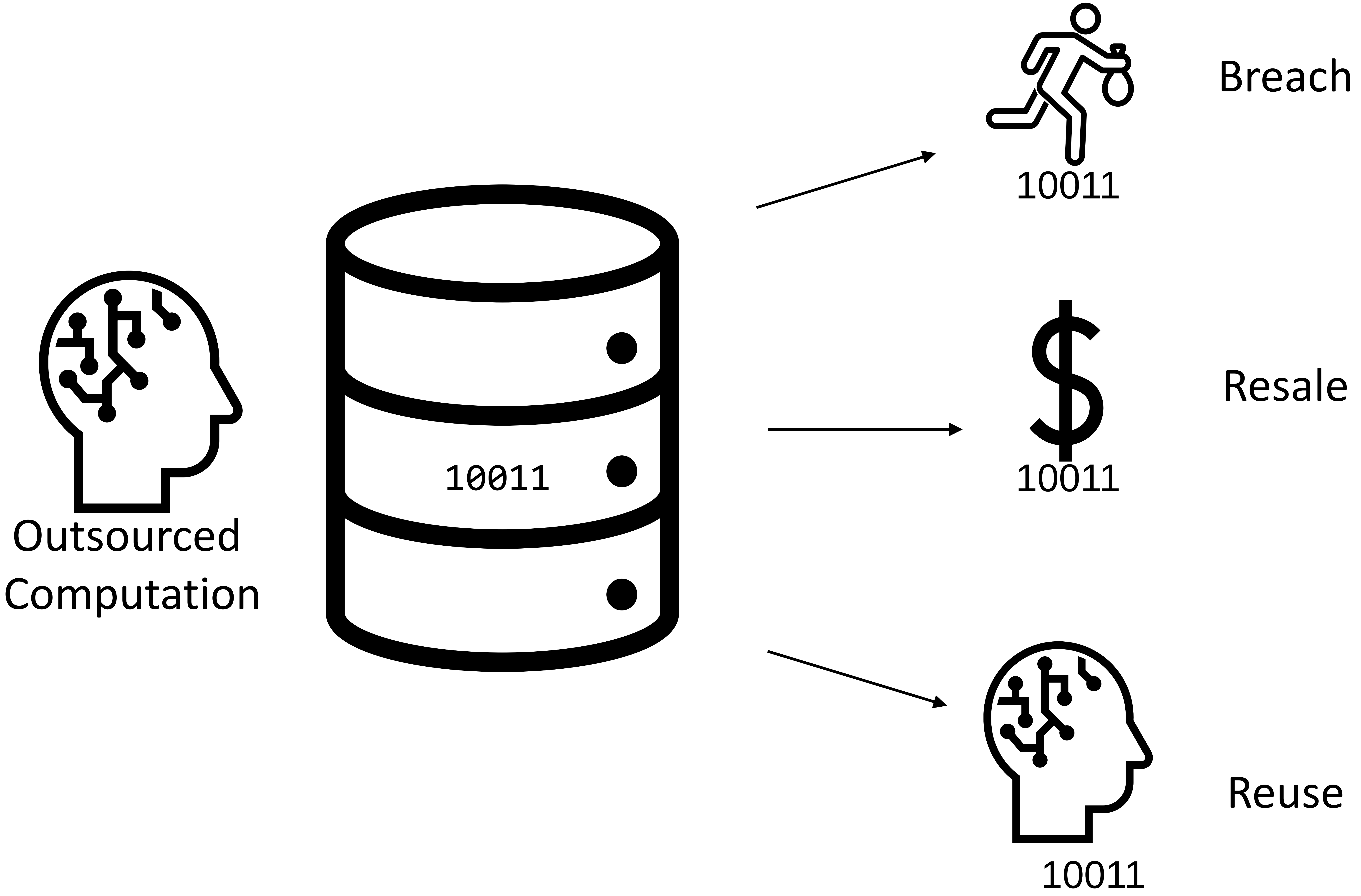
Introduction to Fully Homomorphic Encryption (FHE)

The Importance of Privacy Preserving Computation

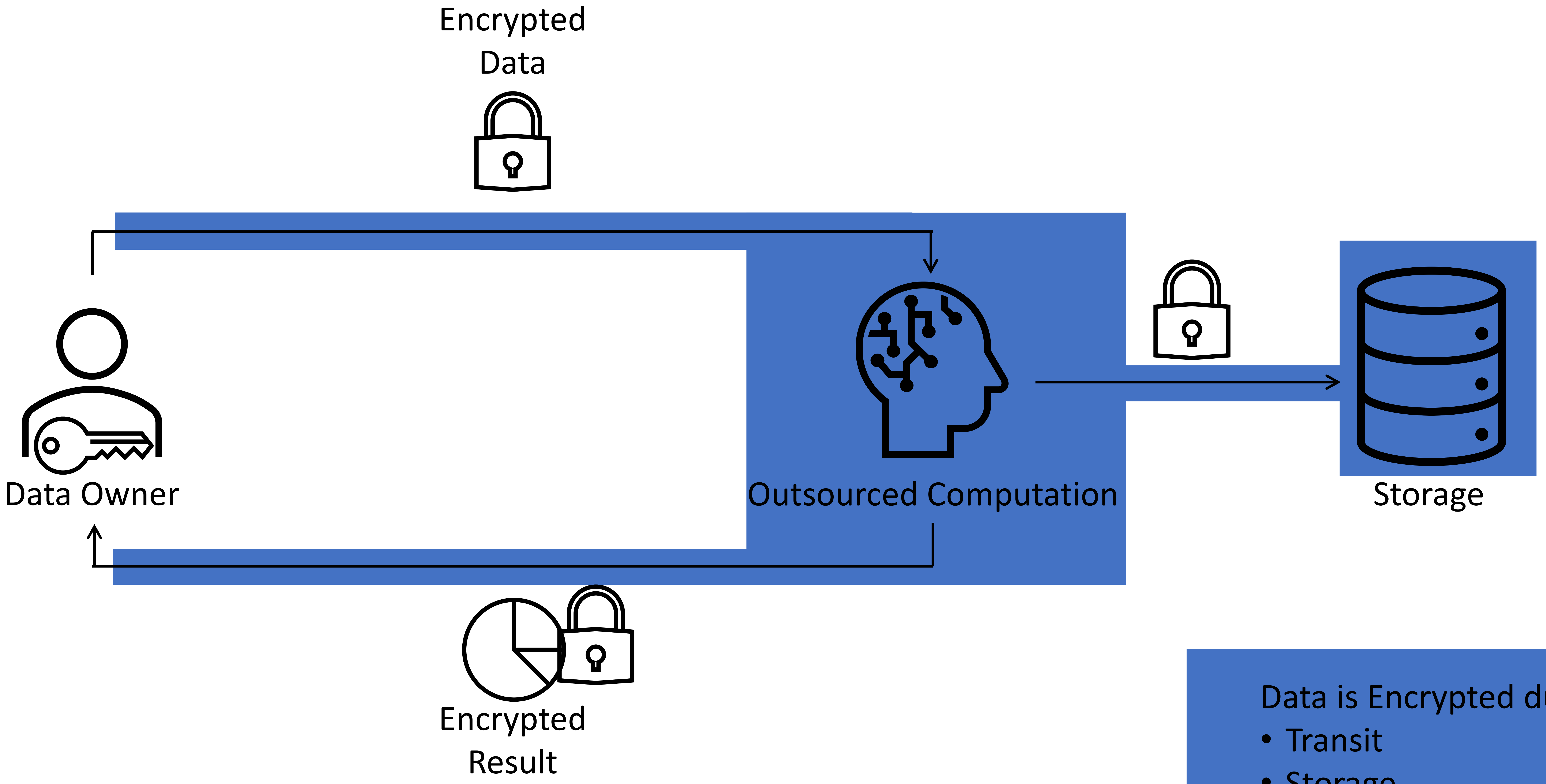


Data is Encrypted during:
- Transit
- Storage

Data Leakage



Encrypted Computation



Data is Encrypted during:
- Transit
- Storage
- **Computation**


TerraUniversal.com

Homomorphic Encryption

- HE allows for operation on the encrypted data without the need for decryption or access to the secret key
- Perfect for outsourced computation…..
- … but there is a catch (multiple catches, actually)
  - Only supports addition and multiplication
  - Noise growth
  - Execution time
  - Size
- Pro: No interactions that are not in the plaintext version
- Con: High computational cost
- Basically trade-off network traffic for CPU load

The High-level View

- Encrypt data by adding noise
- Decryption removes the noise
- Multiplication increases the noise in the Ciphertext
- Too much noise prevents correct decryption
- With leveled FHE we can configure the number of multiplications we can perform
- Once all levels are used we can do no further computation
- With bootstrapping we can refresh the number of levels…
- … unlimited computation!

Homomorphic Encryption - Types

- Partially homomorphic encryption
  - Only support one type of operation like addition or multiplication
- Somewhat homomorphic encryption
  - Support addition and multiplication but not all types of circuits
- Leveled fully homomorphic encryption
  - Support addition and multiplication and circuits with a predefined depth
- Fully homomorphic encryption
  - Support addition and multiplication and circuits of arbitrary depth

Fully Homomorphic Encryption – History 1/2

- First proposed in 1978 after the publication of RSA
  - 30 years of partial results
- First FHE
  - Done by Craig Gentry in 2010
  - Based on ideal lattices
  - Encryption adds noise to the data
  - Operations increase the noise
  - If the noise is too large decryption is impossible
  - Introduction of the bootstrapping trick
    - Use scheme that can evaluate its own decryption function homomorphically
    - This refreshes the noise and allows for further computation

Fully Homomorphic Encryption – History 1/2

IEEE MILCOM IEEE ComSoc

- Improved FHE schemes
  - Mostly based on Ring Learning with Errors Problem (RLWE)
  - BGV https://eprint.iacr.org/2011/277
  - BFV https://eprint.iacr.org/2012/144
  - CKKS https://link.springer.com/chapter/10.1007%2F978-3-319-70694-8_15
  - Support for bootstrapping or leveled FHE
  - Faster bootstrapping
  - TFHE https://tfhe.github.io/tfhe/
    - Supports binary gates
    - High level operations need to pieced together

Open-Source Libraries

IEEE MILCOM IEEE ComSoc

- HELib (https://github.com/HomEnc/HElib)
  - One of the earliest libraries
  - Supports BGV and CKKS
- HEAAN (https://github.com/snucrypto/HEAAN)
  - Original implementation of the CKKS scheme
- TFHE (https://github.com/tfhe/tfhe)
  - Original implementation of the TFHE scheme
- OpenFHE (https://github.com/openfheorg/openfhe-development)
  - Implementation of the most common schemes with bootstrapping and multiparty support
- SEAL (https://github.com/microsoft/SEAL)
  - BFV and CKKS support
  - Used to be popular among researchers
- Lattigo (https://github.com/ldsec/lattigo)
  - Go implementation of various schemes

Mathematical Background

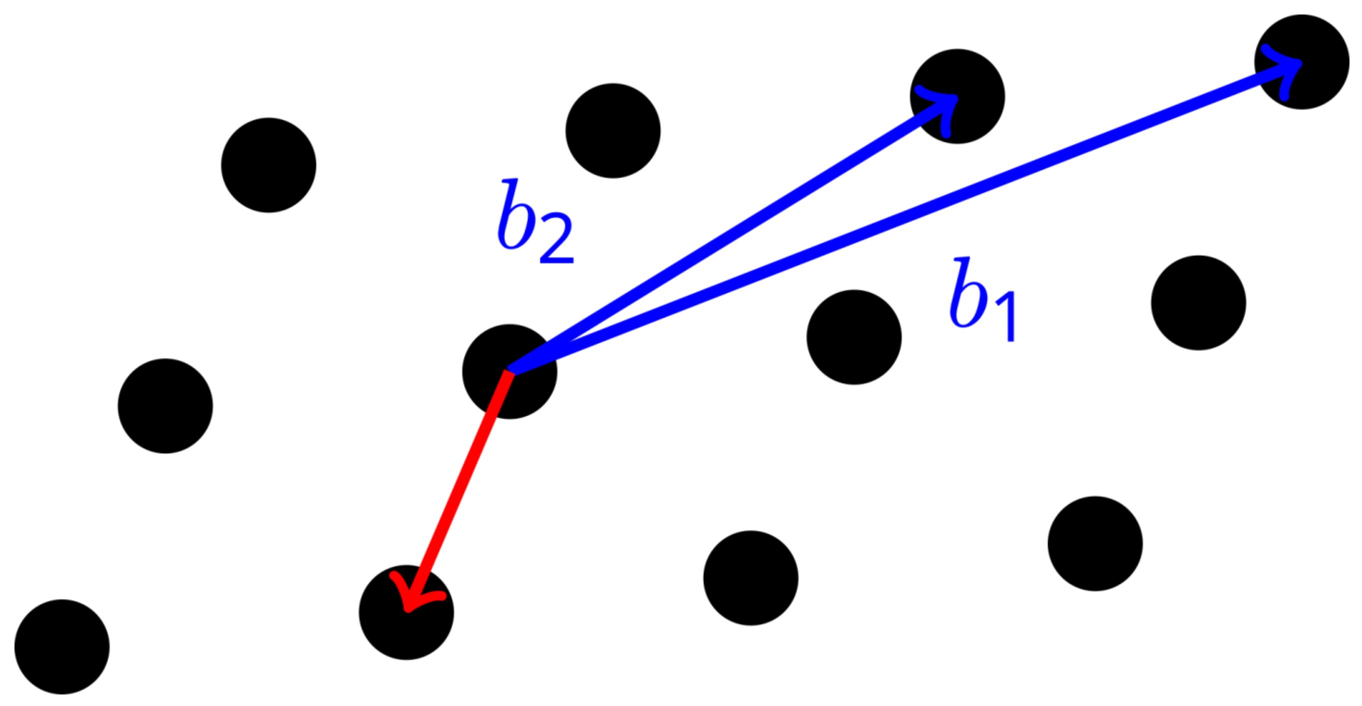
Learning with Errors

- LWE: adding small, random noise to linear equations makes solving them difficult.
- q: a large prime modulus.
- n: the dimension (typically chosen large for security).
- $s \in Z_q^n$: a secret vector.
- $A \in Z_q^{m \times n}$: a randomly chosen matrix.
- $e \in Z_q^m$ a small error vector, where each entry is drawn from a specific error distribution (usually a discrete Gaussian distribution).

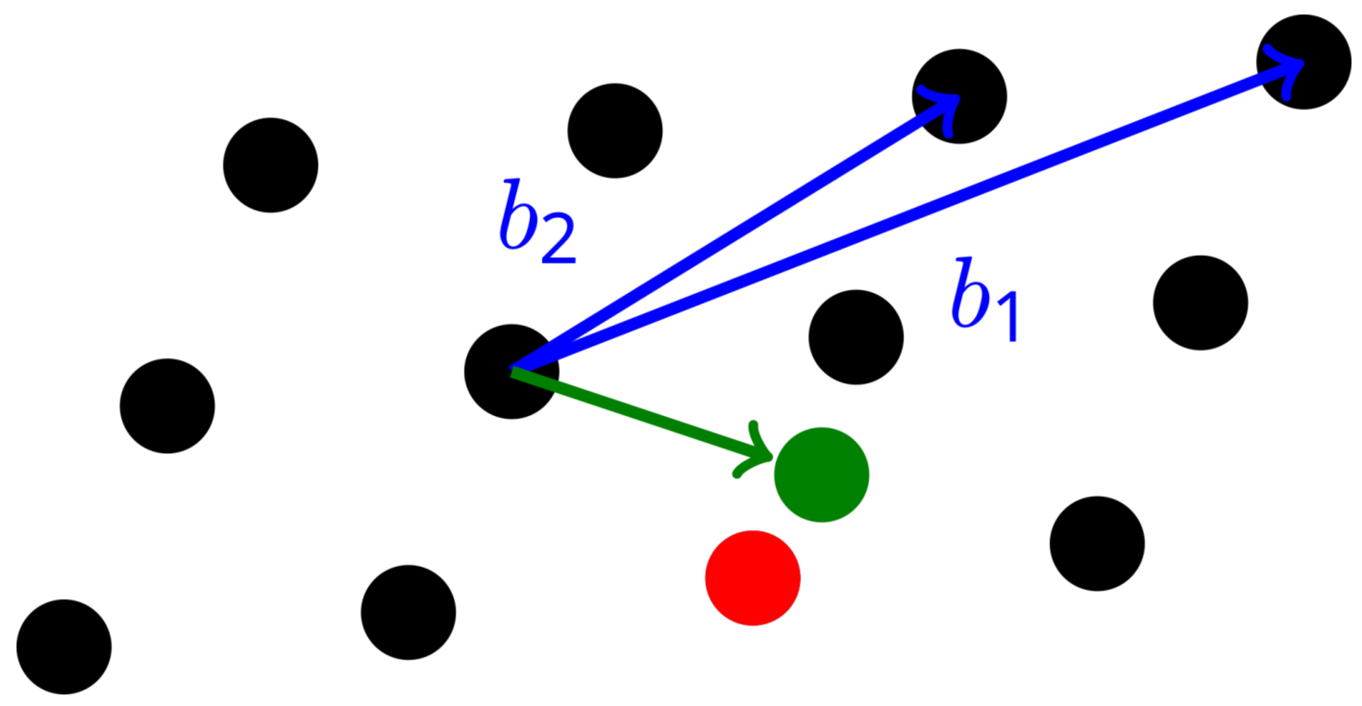
Hardness of Learning with Errors

- Search LWE
  - Given $b = A \cdot s + e \ (mod\ q)$
  - the hard problem is to find s
- Decision LWE
  - Given A and b, decide if b was generated using noise or if it is just a random vector
- Solving these problems can be linked to lattice problems
  - Problems are believed to quantum hard

Shortest Vector Problem



Closest Vector Problem



LWE Example

- Modulus $q = 7$
- Secret $s = [3, 5]$
- Randomly choose $A = \begin{bmatrix} 1 & 4 \\ 2 & 6 \end{bmatrix}$
- Random small vector $e = [1, 0]$
- Compute $b = \begin{bmatrix} 1 & 4 \\ 2 & 6 \end{bmatrix} \cdot \begin{bmatrix} 3 \\ 5 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ mod 7
  - $b = \begin{bmatrix} 1 \cdot 3 + 4 \cdot 5 \\ 2 \cdot 3 + 6 \cdot 5 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ mod 7
  - $b = \begin{bmatrix} 24 \\ 36 \end{bmatrix}$ mod 7
  - $b = \begin{bmatrix} 3 \\ 1 \end{bmatrix}$

Ring Learning With Errors 1/2

- Extension of LWE to polynomial rings
- A polynomial ring is a mathematical structure
  - With addition and multiplication
  - Elements are polynomials
- We work with quotient rings
  - All polynomials are modulo q
  - Cyclotomic polynomial $\mathrm{f}(x)$

$$R_q = \mathbb{Z}_q[X]/(f(x))$$

Is the ring with of polynomials with integer coefficients mod q and polynomials are reduced by $\mathrm{f}(x)$

Ring Learning With Errors 2/2

- The components are similar to the LWE
  - $a(x)$ a random polynomial
  - $s(x)$ a secret polynomial
  - $e(x)$ a small error polynomial
- RLWE is more efficient than LWE

Example:

- $q = 7, R_7 = \mathbb{Z}_7[X]/(x^2 + 1)$
- $a(x) = 4 + x$
- $s(x) = 3 + 2x$
- $e(x) = 1$

$$b(x) = a(x) \cdot s(x) + e(x) \; mod \; 7$$

$$b(x) = (4 + x) \cdot (3 + 2x) + 1 \; mod \; 7$$

Polynomial multiplication:

$$(4 + x) \cdot (3 + 2x) = 12 + 8x + 3x + 2x\text{^}2$$

Reduce modulo $\mathrm{x}^2 + 1$:

$$= 12 + 8x + 3x + 2(-1) = 10 + 11x$$

Reduce coefficients modulo 7 and error $e(x) = 1$

$$= 10 + 11x = 3 + 4x$$

Add error $e(x) = 1$

$$\mathbf{4 + 4x}$$

FHE Schemes

Single Instruction Multiple Data (SIMD)

- Most schemes allow encoding multiple messages into a plaintext/ciphertext
- All operations on the plaintext/ciphertext are performed on all messages encoded at no extra cost
- Max. number of messages is called *slots*
- Number of slots typically >1000

BFV/BGV Schemes

- BFV/BGV Schemes (Brakerski-Fan-Vercauteren/Brakerski-Gentry-Vaikuntanathan):
  - Structure:
    - The BFV/BGV scheme supports operations on integers or fixed-point numbers, making it useful for exact computations. BFV focuses more on optimizations.
  - Use Cases:
    - Best for scenarios where exact arithmetic is needed, such as financial computations (e.g., balance calculations) or voting systems.
  - Properties:
    - Precise results and supports both addition and multiplication on ciphertexts.
    - Supports batching (processing multiple encrypted data points at once)

CKKS Scheme

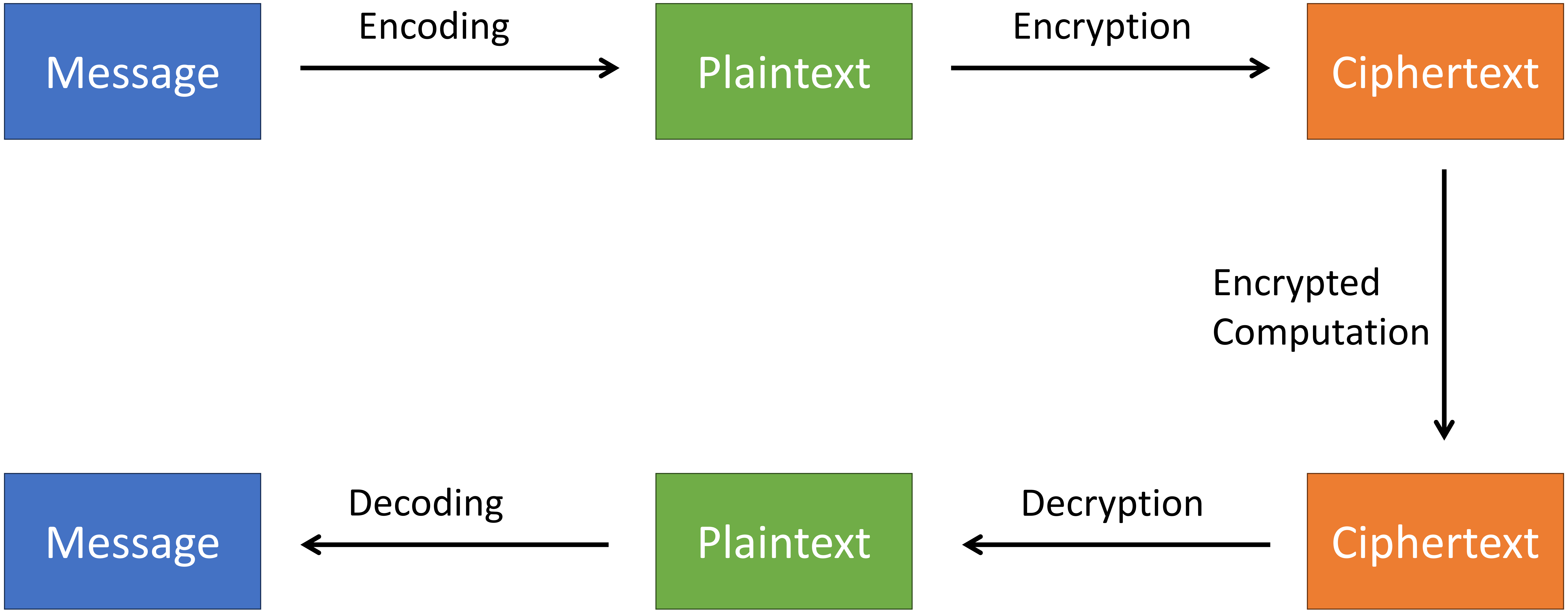
- CKKS (Cheon-Kim-Kim-Song) Scheme
  - Structure:
    - Unlike BFV and BGV, CKKS is designed for computations on real numbers. It enables approximate arithmetic, meaning computations will introduce a small amount of error.
  - Use Cases:
    - Applications requiring computations on real or floating-point numbers, such as machine learning algorithms, image processing, or signal processing. Commonly used in AI/ML, where a high degree of precision is often not needed (approximations are acceptable).
  - Properties:
    - Supports addition and multiplication but introduces a level of approximation (rounding). Provides excellent performance for large-scale computations due to reduced noise accumulation compared to other schemes.
    - Supports batching (processing multiple encrypted data points at once)

TFHE

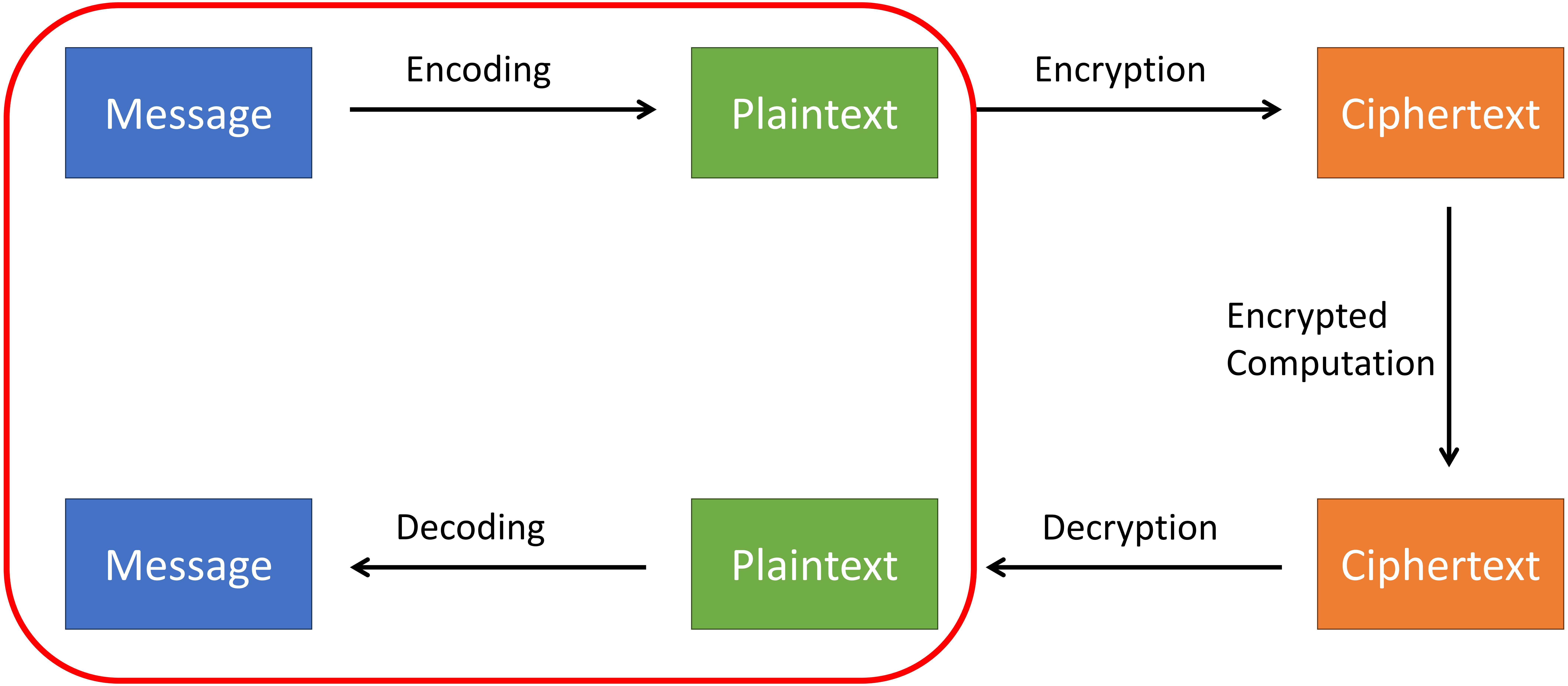
- TFHE (Torus Fully Homomorphic Encryption)
  - AKA. CGGI (Chillotti-Gama-Georgieva-Izabachène)
  - Structure:
    - TFHE is optimized for fast and efficient boolean circuit evaluation. It uses operations over the torus, which makes binary computations faster.
  - Use Cases:
    - Ideal for applications that require boolean logic, such as circuits performing logical AND/OR/NOT operations. Examples include secure voting systems, logic gate-based algorithms, or encrypted control systems.
  - Properties:
    - Low Latency: TFHE allows for fast, low-latency operations, making it one of the most efficient schemes for binary data.
    - Binary Gates: Supports the evaluation of encrypted logic gates, making it particularly useful for cryptographic protocols and arbitrary computation.

CKKS: The Details

CKKS Overview



CKKS Overview



CKKS Encoding

Encoding: complex vector $\mathbb{C}^{\frac{N}{2}} \rightarrow m(x) \in \mathbb{Z}[X]/(X^N + 1)$

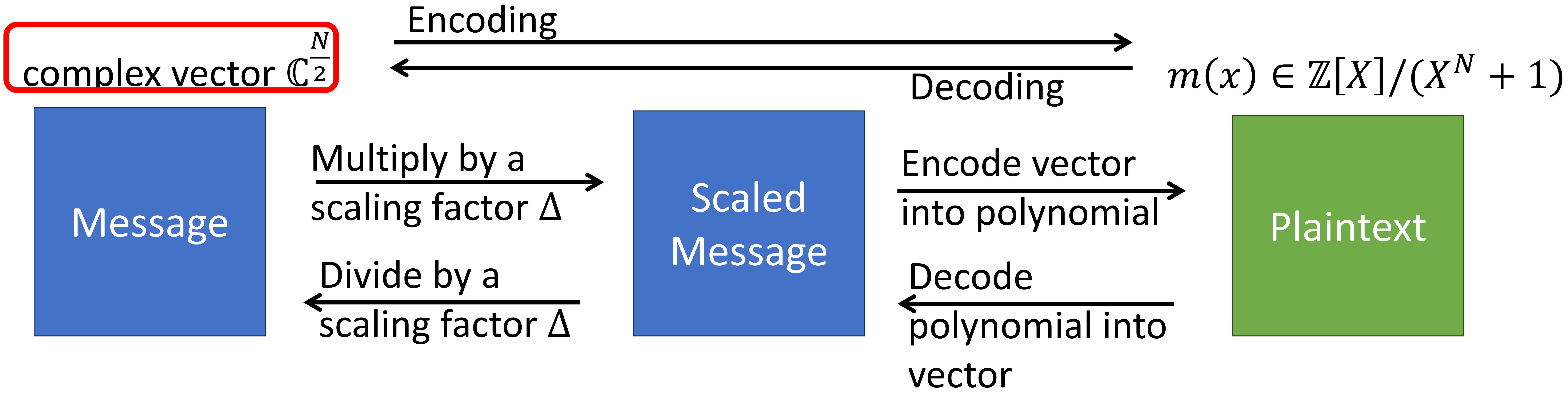
Decoding: $m(x) \in \mathbb{Z}[X]/(X^N + 1) \rightarrow$ complex vector $\mathbb{C}^{\frac{N}{2}}$

Message $\xrightarrow{\text{Multiply by a scaling factor } \Delta}$ Scaled Message $\xrightarrow{\text{Encode vector into polynomial}}$ Plaintext

Plaintext $\xrightarrow{\text{Decode polynomial into vector}}$ Scaled Message $\xrightarrow{\text{Divide by a scaling factor } \Delta}$ Message

- Message is the vector $z = [z_1, z_2, \ldots, z_{N/2}]$
- In the encoded plaintext $m(x)$ the slot i contains the value Δz_i (approximately)
- $m(\zeta_i) \approx \Delta \cdot z_i$ for root ζ_i of $X^N + 1 = 0$

CKKS Encoding



- Message is the vector $z = [z_1, z_2, \dots, z_{N/2}]$
- In the encoded plaintext $m(x)$ the slot i contains the value Δz_i (approximately)
- $m(\zeta_i) \approx \Delta \cdot z_i$ for root ζ_i of $X^N + 1 = 0$

CKKS Encoding

IEEE MILCOM IEEE ComSoc

Encoding

complex vector $\mathbb{C}^{\frac{N}{2}}$

Decoding

$m(x) \in \mathbb{Z}[X]/(X^N + 1)$

Message

Multiply by a scaling factor Δ

Divide by a scaling factor Δ

Scaled Message

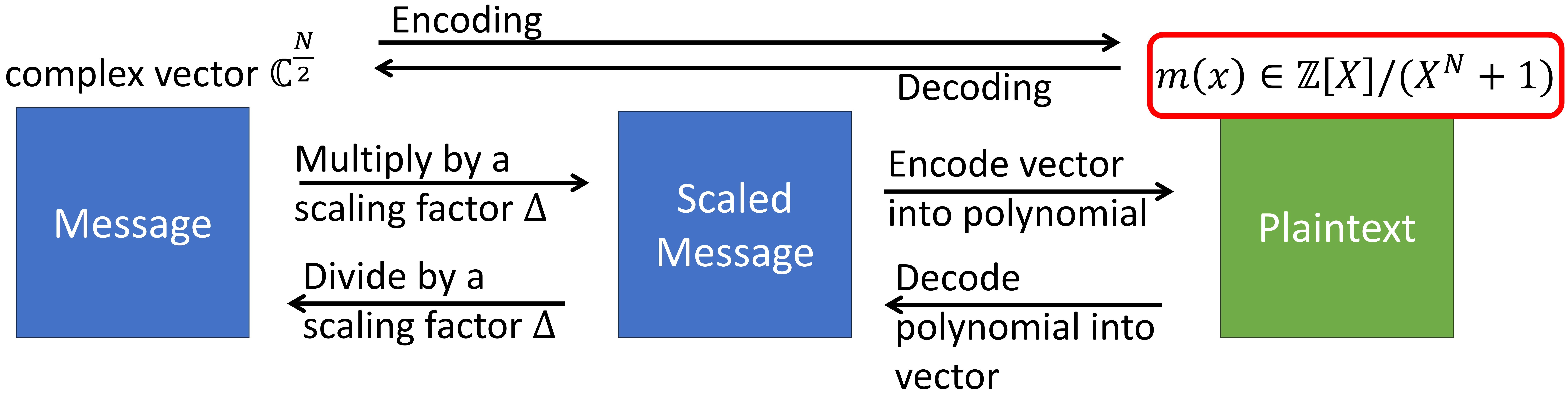
Encode vector into polynomial

Decode polynomial into vector

Plaintext

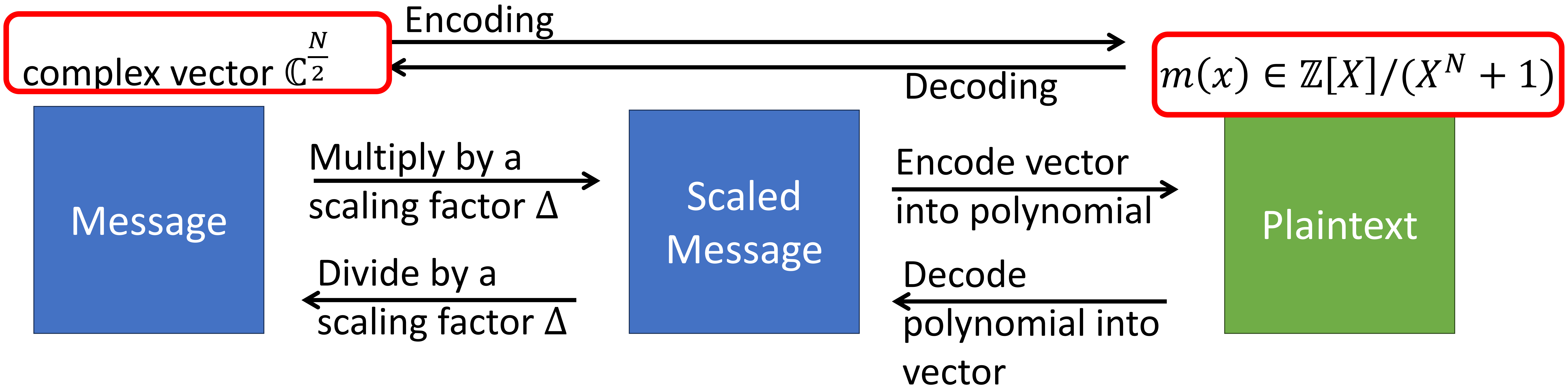
- Message is the vector $z = [z_1, z_2, \dots, z_{N/2}]$
- In the encoded plaintext $m(x)$ the slot i contains the value $\boxed{\Delta z_i}$ (approximately)
- $m(\zeta_i) \approx \Delta \cdot z_i$ for root ζ_i of $X^N + 1 = 0$

CKKS Encoding



- Message is the vector $z = [z_1, z_2, \dots, z_{N/2}]$
- In the encoded plaintext $m(x)$ the slot i contains the value Δz_i (approximately)
- $m(\zeta_i) \approx \Delta \cdot z_i$ for root ζ_i of $X^N + 1 = 0$

CKKS Encoding



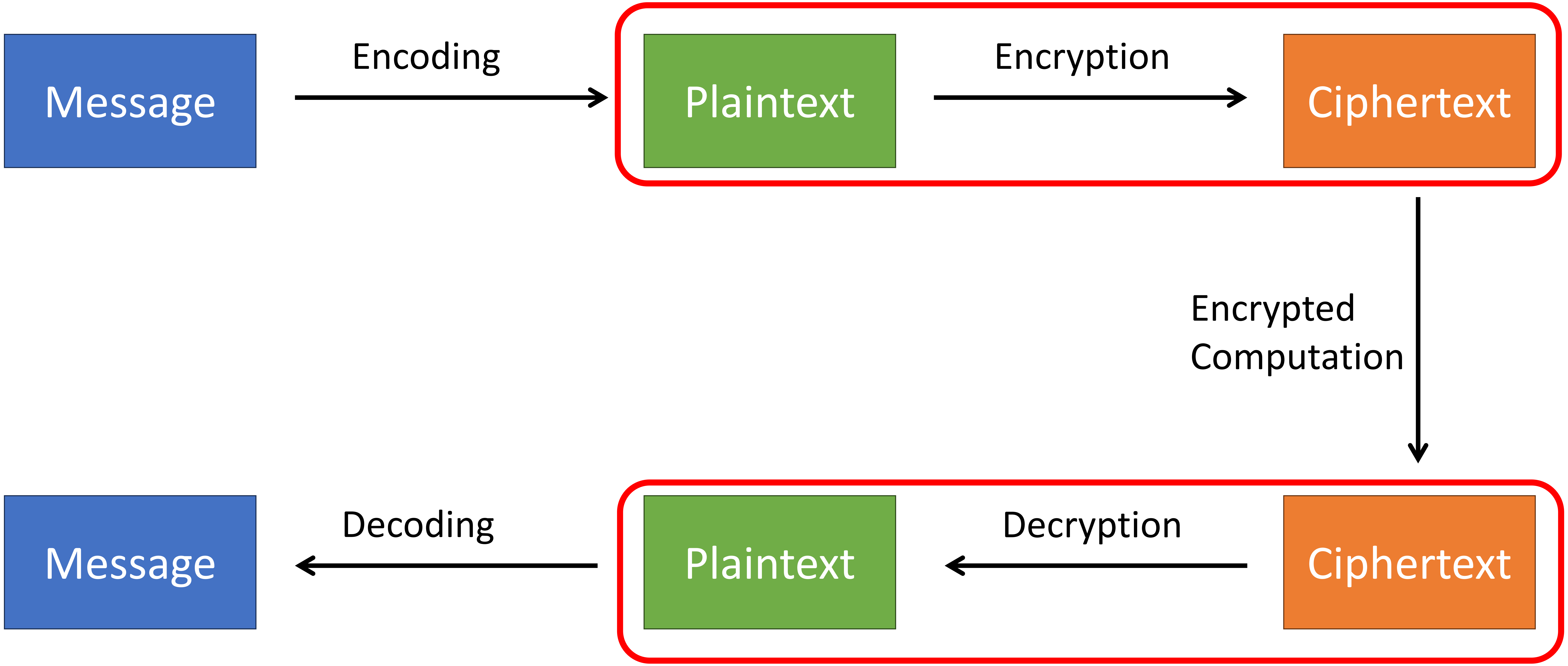
The message vector z is encoded in the plaintext polynomial $m(x)$

CKKS Encoding Example

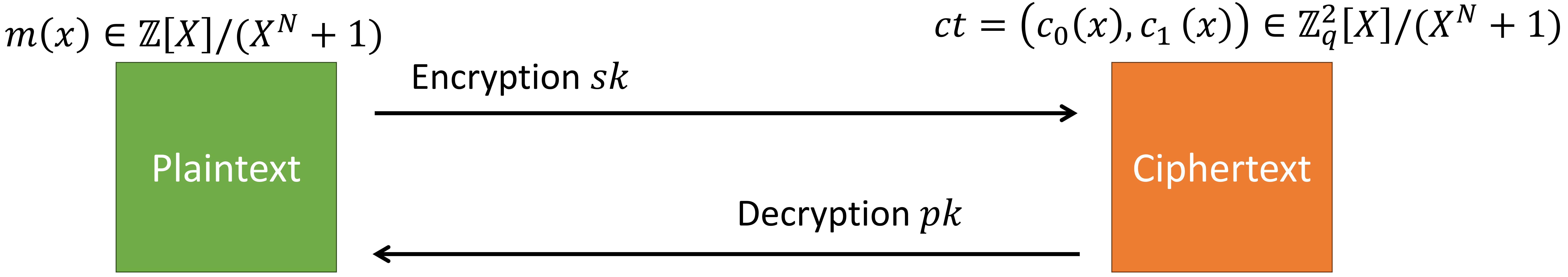
- $\mathrm{N} = 4, \Delta = 2^7$
- Message: $(z_1, z_2) = (1.2 - 3.4i, 5.6 + 7.8i)$
- Encoded message: $m(x) = 435 - 706x + 282x^2 - 308x^3$
- Complex roots of $X^4 + 1 = 0$
  - $\zeta_1 = (1 + i)/\sqrt{2}$
  - $\zeta_2 = -(1 + i)/\sqrt{2}$
  - …
- $m(\zeta_1) \approx 153.5 - i \cdot 435.0$
- $m(\zeta_1)/\Delta \approx 1.998 - i \cdot 3.398$

- $m(\zeta_2) \approx 716.4 + i \cdot 999.0$
- $m(\zeta_2)/\Delta \approx 5.597 + i \cdot 7.805$

CKKS Overview



CKKS Secret and Public Key



- Sample three polynomials from $\mathbb{Z}_q[X]/(X^N + 1)$: a, sk, e
  - Secret key: sk
  - Public key: $pk = (-a \cdot sk + e, a)$
  - A small error polynomial e

CKKS Encryption

$m(x) \in \mathbb{Z}[X]/(X^N + 1)$

$c = \left(c_0(x), c_1(x)\right) \in R_q^2$

- Encryption:
  - Encrypt the message polynomial m into two polynomials c_0, c_1

$$Enc(m) =$$

$$(m, 0) + pk =$$

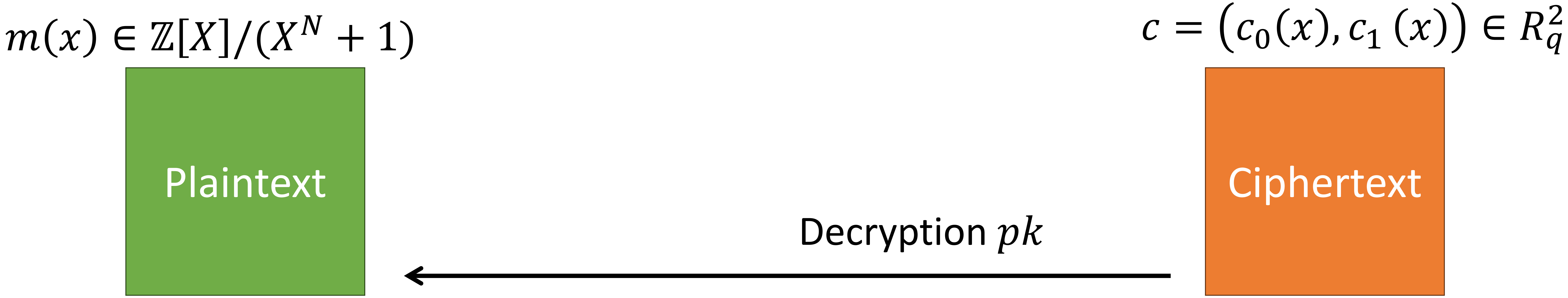
$$(m - a \cdot sk + e, a) =$$

$$(c_0, c_1) = c$$

(we use m and c_i instead of $m(x), c_i(x)$

for simplicity)

CKKS Decryption



- Decryption:
  - Decrypt the ciphertext polynomials c_0, c_1 into the message polynomial m

Recall:
$c_0, c_1 = (m - a \cdot sk + e, a)$

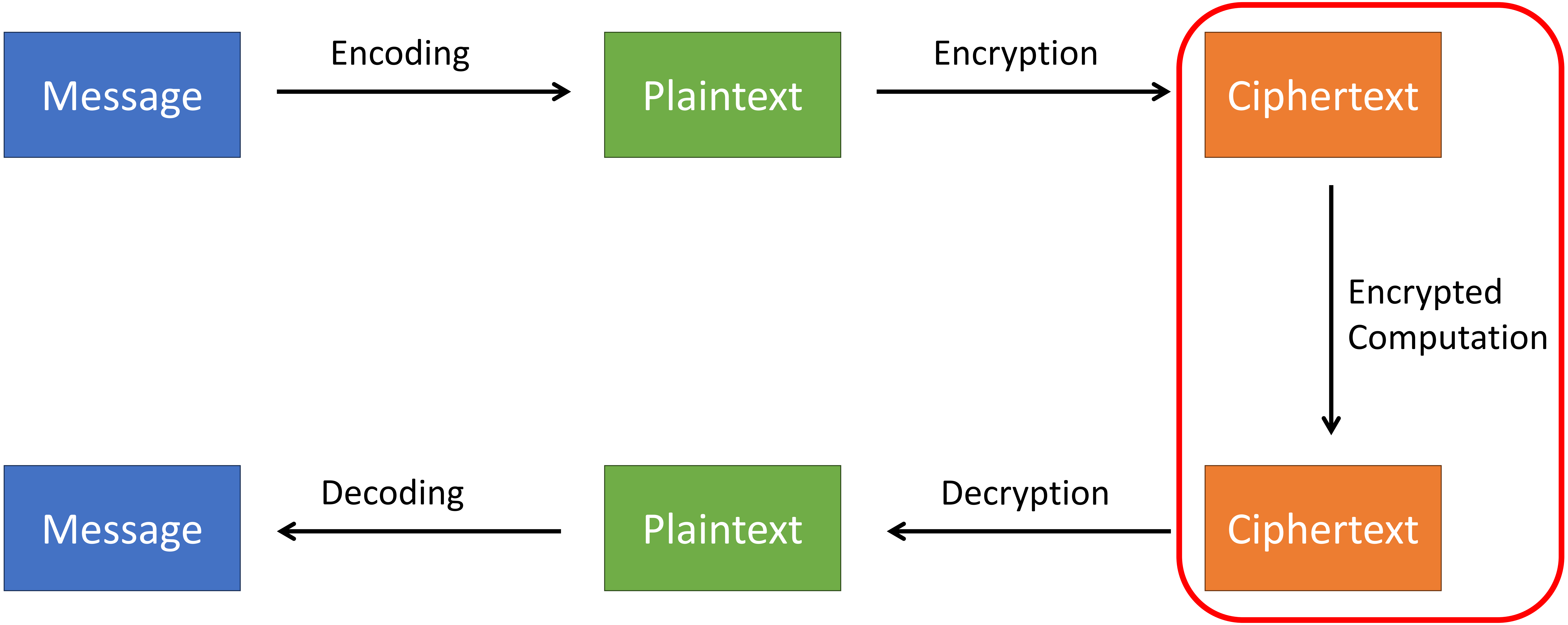
$$Dec(c) =$$

$$c_0 + c_1 sk =$$

$$m - \cancel{a \cdot sk} + e + \cancel{a \cdot sk} =$$

$$m + e \approx m$$

CKKS Overview



CKKS Addition

- Given two ciphertexts $c = (c_0, c_1), d = (d_0, d_1)$
- Addition:
  - Straight forward. Add polynomials of both chiphertexts

$$Add(c, d) = (c_0 + d_0, c_1 + d_1)$$

CKKS Multiplication

- Given two ciphertexts $c = (c_0, c_1), d = (d_0, d_1)$
- Multiplication
  - What we want is: $\mathrm{Dec(cd)} = \ Dec(c) \cdot Dec(d)$

$$(c_0 + c_1 \cdot sk) \cdot (d_0 + d_1 \cdot sk) =$$

$$c_0 d_0 + (c_0 d_1 + d_0 c_1) \cdot sk + c_1 d_1 \cdot sk^2$$

  - We can compute the product as

$$cd = (c_0 d_0, c_0 d_1 + d_0 c_1, c_1 d_1)$$

  - BUT the ciphertext consist of three polynomials now

CKKS Relinearization

- After multiplication the ciphertext consists of three parts
- To bring it back down to two we use relinearization
  - Create a relinearization key:
    - e_0 small random polynomial
    - a_0 random polynomial
    - v a large integer
  - relinearization key $rk = (-a_o sk + e_0 + sk^2, a_0)\ mod\ vq$
  - $(-a_o sk + e_0 + sk^2, a_0)$ decrypts to $e_0 + sk^2$
  - p is used to control the noise introduced
  - $Relin(Mult(c, d), rk) = (c_0 d_0, c_0 d_1 + d_0 c_1) + \lfloor \frac{c_1 d_1 \cdot rk}{v} \rceil$

CKKS Rescaling Motivation

- Recall:
  - The ciphertext c encrypts some message z scaled by Δ
  - $c \cdot c$ encrypts $z^2\Delta^2$
  - Multiplication causes the scale Δ to grow quadratically
- We want to keep the scale Δ the same size after multiplication to prevent overflow
- Rescaling allows us to reduce the size of Δ after a multiplication

Residue Number System 1/2

- Time for a small detour
- Residue Number System (RNS), related to the Chinese Remainder Theorem
- Given a set of “small” coprime numbers we can represent a large integer as a set of smaller integers
- Given n coprimes $c_1, \dots, c_n$ we can represent numbers between 0 and $-1 + \prod_{i=1}^{n} c_i$
- We represent a number x as an n tuple where each element is the remainder of $\frac{x}{c_i}$

Residue Number System 2/2

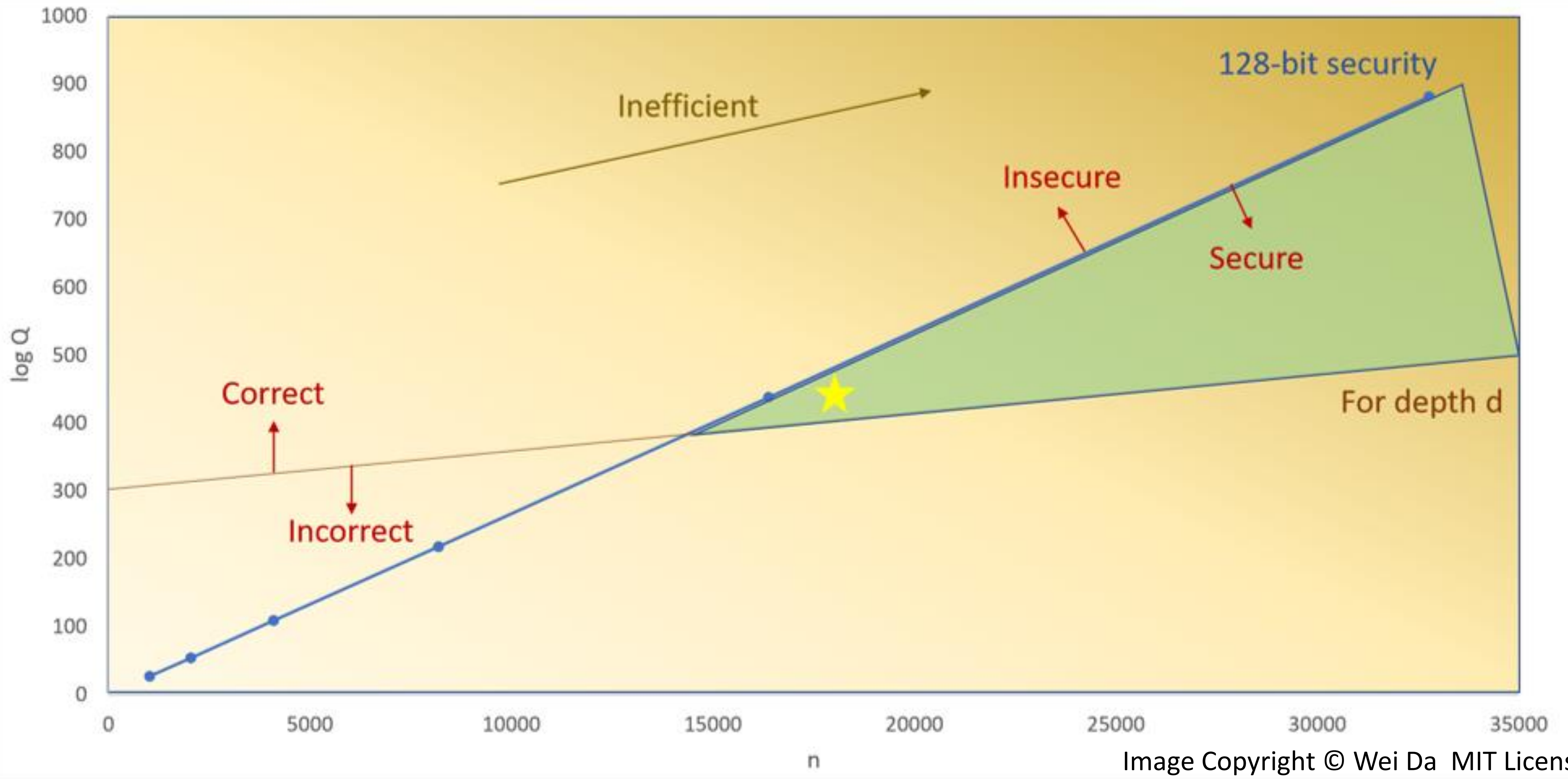
- Almost back on track
- Addition and multiplication is element-wise
- Example:
  - Co-primes: 3, 5, 11
  - Can represent numbers between 0 and 164
  - 16 -> (1,1,5), 9 -> (0, 4, 9)
  - 16+9 = 25 -> (1,0,3) = (1,1,5) + (0,4,9) = (1+0, 1+4, 5+9) = (1,0,3)
  - 16*9 = 144 -> (0,4,1) = (1,1,5) * (0,4,9) = (1*0, 4*1, 5*9) = (0,4,1)
- Why do we need this?
  - Numbers can get 100s of bits large
  - Working with numbers larger than a word (64bit) is slow

CKKS Rescaling

- We can select the ciphertext modulus q as the product of multiple smaller (less than word-size) primes p_l and a prime q_0
- L is the number of the smaller primes p_l
- Select L primes $p_1, \ldots p_L$, each $p_l \approx \Delta$, and a prime $q_0 > \Delta$
- After each multiplication, we can “discard” one of the primes
  - Ciphertext c is now $c' \in R_{q'}^2$, with $q' = \frac{q}{\Delta}$
  - Scaling factor Δ^2 is reduced to Δ
  - Doesn’t change the encrypted message only the representation
- We can only preform L multiplications -> leveled HE

Security Parameters

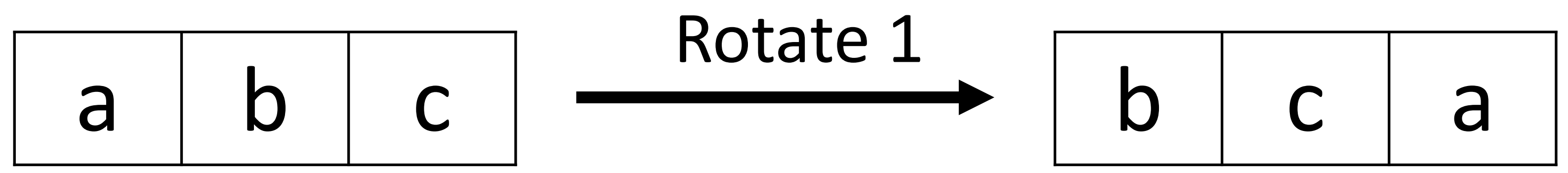
- For security increase n
- For more levels increase q
- Security of the scheme relies on $\frac{n}{q}$
  - as q increases so must n
- Larger values increase the computational cost
- The HE standard provides values for n and q that provide 128bit security



N	log q
1024	29
2048	56
4096	111
8192	220
16384	440
32768	880

Additional CKKS operations

- Rotation
  - Ciphertexts are encryptions of vectors
  - We can rotate the elements in the vector with wrap around
    - Requires rotation (galois) keys



- Bootstrapping
  - Resets the level of the Ciphertext to allow additional computation
  - Expensive operation

Computation with FHE

Constraints

- No “random” access to slots in the encrypted vector
  - Can’t do $c[i]$
- No inter slot operations
  - Can’t do $c[i] + c[j]$
- With CKKS we can only evaluate Polynomial functions
- Given a ciphertext c we can’t (easily) compute, e.g.:
  - $\max(y, c)$
  - Sigmoid: $\frac{1}{1+e^{-c}}$
  - $\sqrt{c}$
  - y^c
  - $\frac{y}{c}$
  - ….

Vector Computation

- Element-wise operations are simple

a	b	c	+	d	e	f	=	a+d	b+e	c+f

- But what if we want to compute the inner product?
  - The first part is simple

a	b	c	x	d	e	f	=	ad	be	cf

  - But how do we sum up the rest?

$$\underbrace{\boxed{ad}\boxed{be}\boxed{cf}}_{\Sigma}$$

- Simple Solution:
  - More ciphertexts:

a	a	a	x	d	d	d	=	ad	ad	ad
b	b	b	x	e	e	e	=	be	be	be
c	c	c	x	f	f	f	=	cf	cf	cf

+

ad+be+cf	ad+be+cf	ad+be+cf

Better Vector Computation

- Using multiple ciphertexts is not very efficient
- Better way:
  - Use rotations

a	b	c

x

d	e	f

=

ad	be	cf

ad	be	cf

Rotate by 1 →

be	cf	ad

ad	be	cf

+

be	cf	ad

=

ad+be	be+cf	ad+cf

ad	be	cf

Rotate by 2 →

cf	ad	be

ad+be	be+cf	ad+cf

+

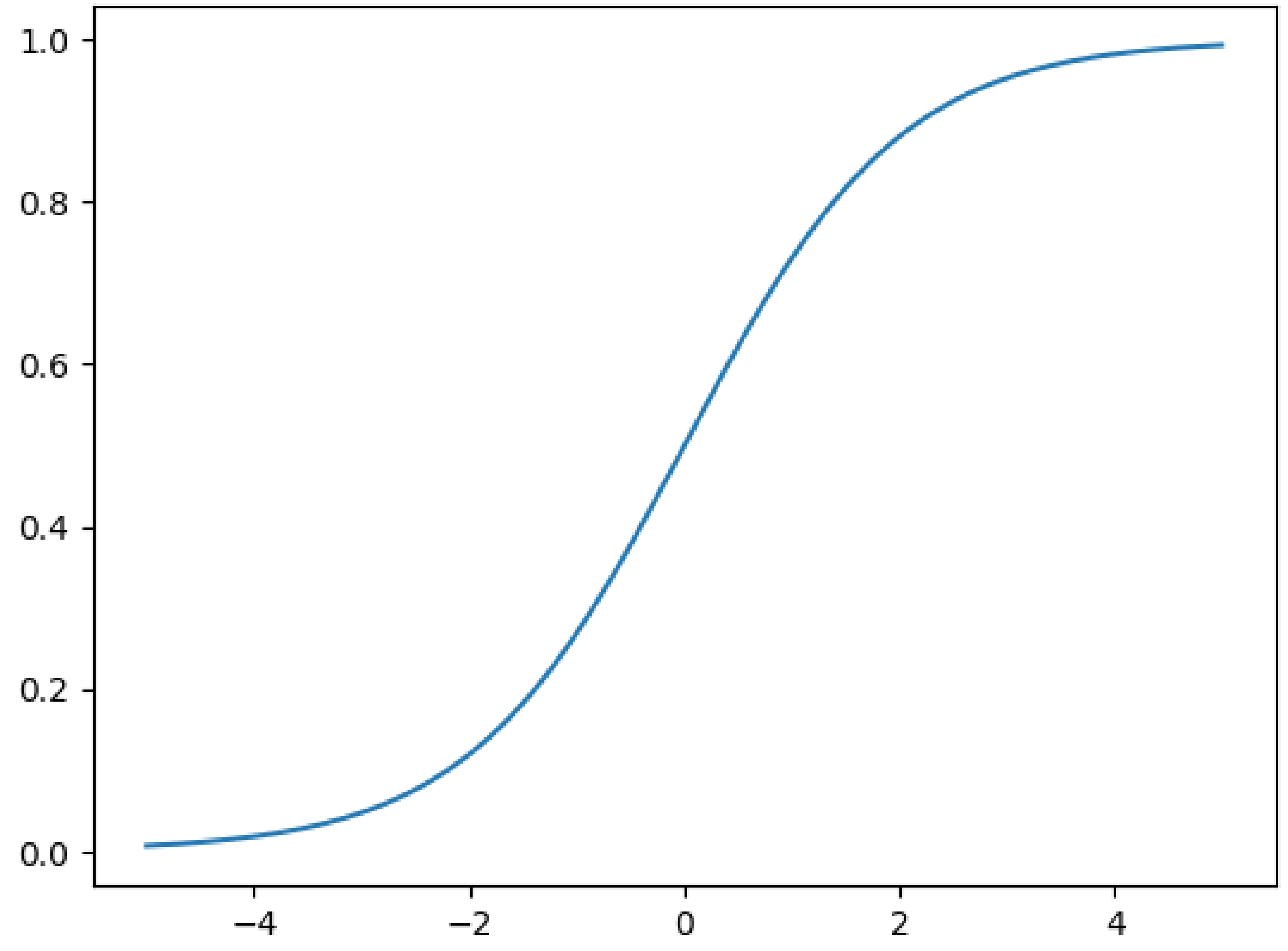
cf	ad	be

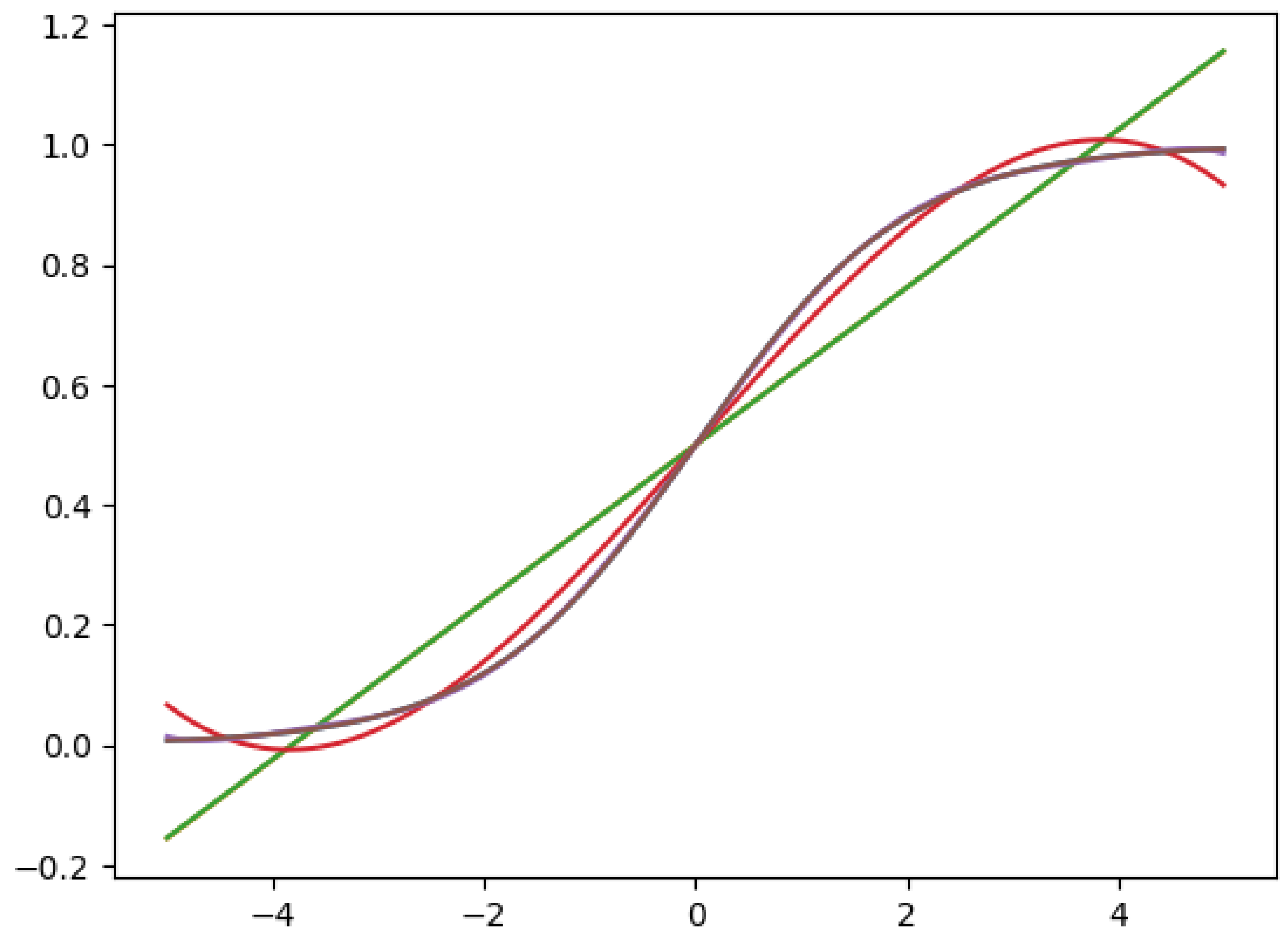
=

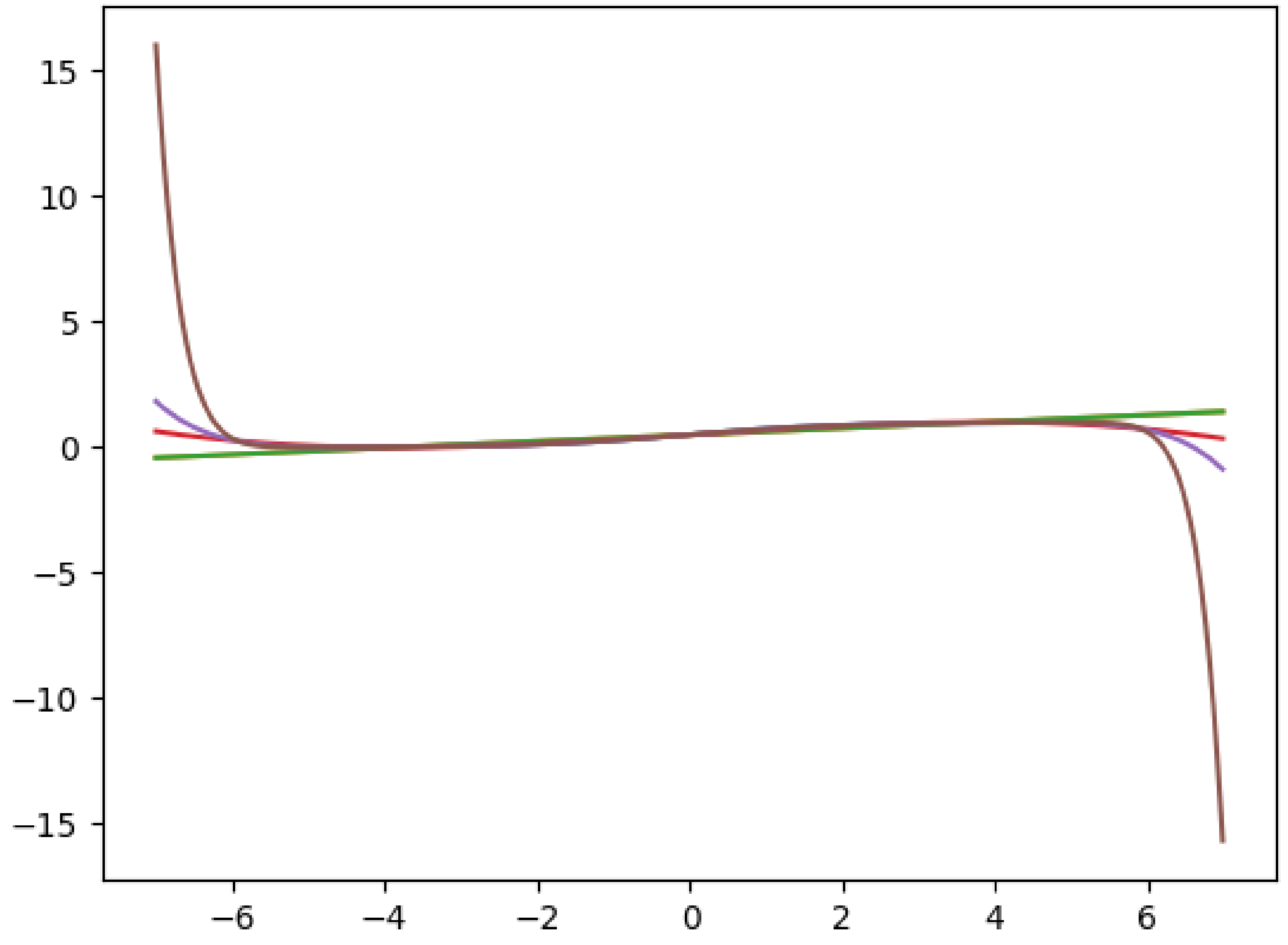
ad+be+cf	ad+be+cf	ad+be+cf

Evaluating Non-Polynomial Functions

- What if we want to evaluate functions that are not easily expressed as polynomials?
- Example:
- $S(x) = \frac{1}{1+e^{-x}}$
- We can approximate the function using polynomials
- Trade-off between accuracy and complexity
- We need to carefully consider the interval. Polynomials can get out of hand quickly

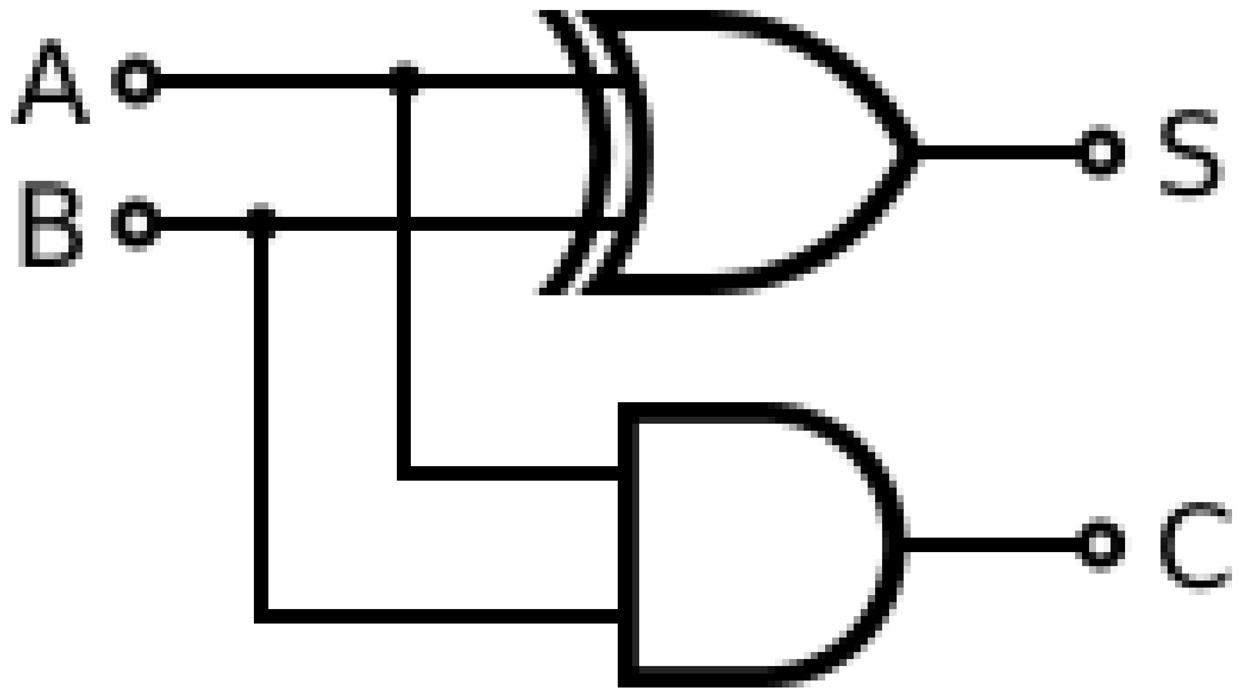






Binary Gate Computation

- FHEW/TFHE support binary gate evaluation
  - AND, OR, NAND, NOR, XOR, XNOR
- We can use the binary gates to (theoretically) express any function or program



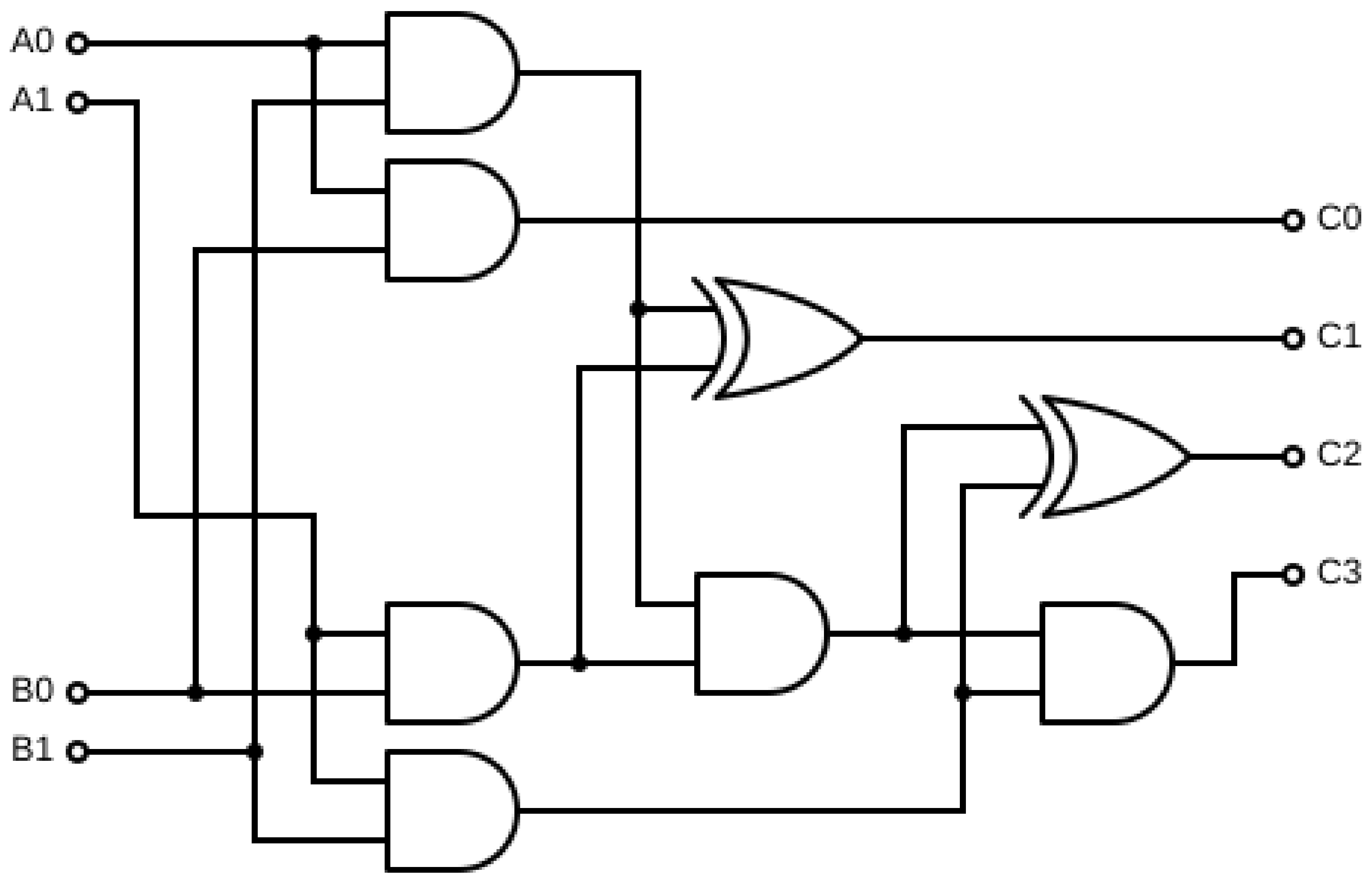


Image Credit: Jooja Creative Commons Attribution-Share Alike 4.0 International

Lookup Tables

- Another way to compute functions is lookup tables
- Lookup up tables store a mapping from input to output
- Example: Sigmoid Function

x	f(x)
-1	0.26894142,
-0.5	0.37754067
0	0.
0.5	0.62245933
1	0.7310585786300049

- Lookup tables can be efficiently computed using binary schemes

Scheme Switching

- If the only tool you have is a hammer every problem looks like nail
- Schemes are great a different things
  - BFV,BGV, CKKS are great for arithmetic
  - TFHE/FHEW are great for binary computation
- We can use the best scheme for the operation
- From CKKS -> FHEW
  - Perform the decoding homomorphically
  - One CKKS ciphertext into multiple FHEW ciphertexts
- From FHEW -> CKKS
  - Homomorphically evaluate the decryption function
  - Multiple FHEW into one CKKS ciphertexts

Tools For An Easier Life

- LWE/Latice Estimator
  - https://github.com/malb/lattice-estimator
  - Use to estimate security of parameters
- TenSEAL
  - https://github.com/OpenMined/TenSEAL
  - Tensor library build on top of SEAL
- HEIR
  - https://heir.dev/
  - Compiler Toolchain for FHE
- Concrete and Concrete-ML
  - https://github.com/zama-ai/concrete and https://github.com/zama-ai/concrete-ml
  - Concrete is a TFHE compiler
  - Concrete-ML is a machine learning built on top of Concrete

Hands On

Getting Ready

- Here is what you need:
  - A browser with internet access
  - A Google account

github.com/podschwadt/fhe_tutorial