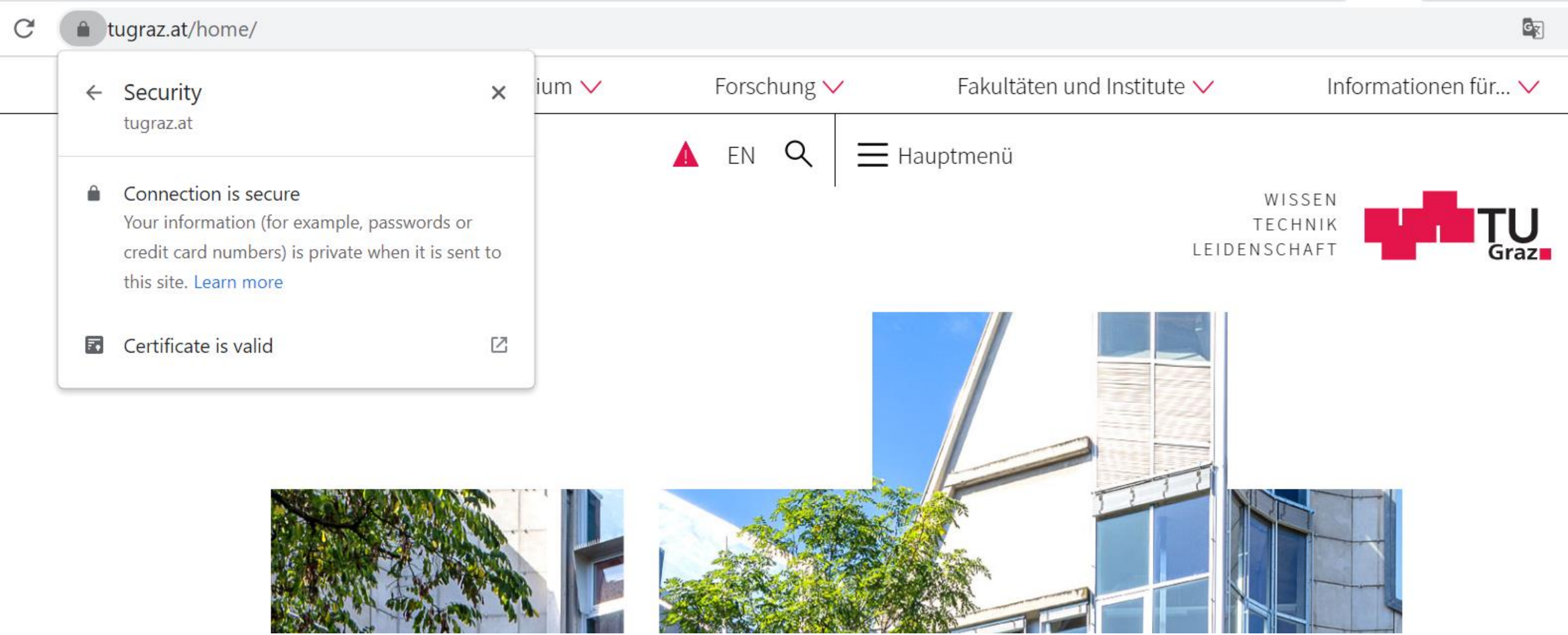
Hardware Implementation of Public-Key Cryptography

Cryptography on Hardware Platform

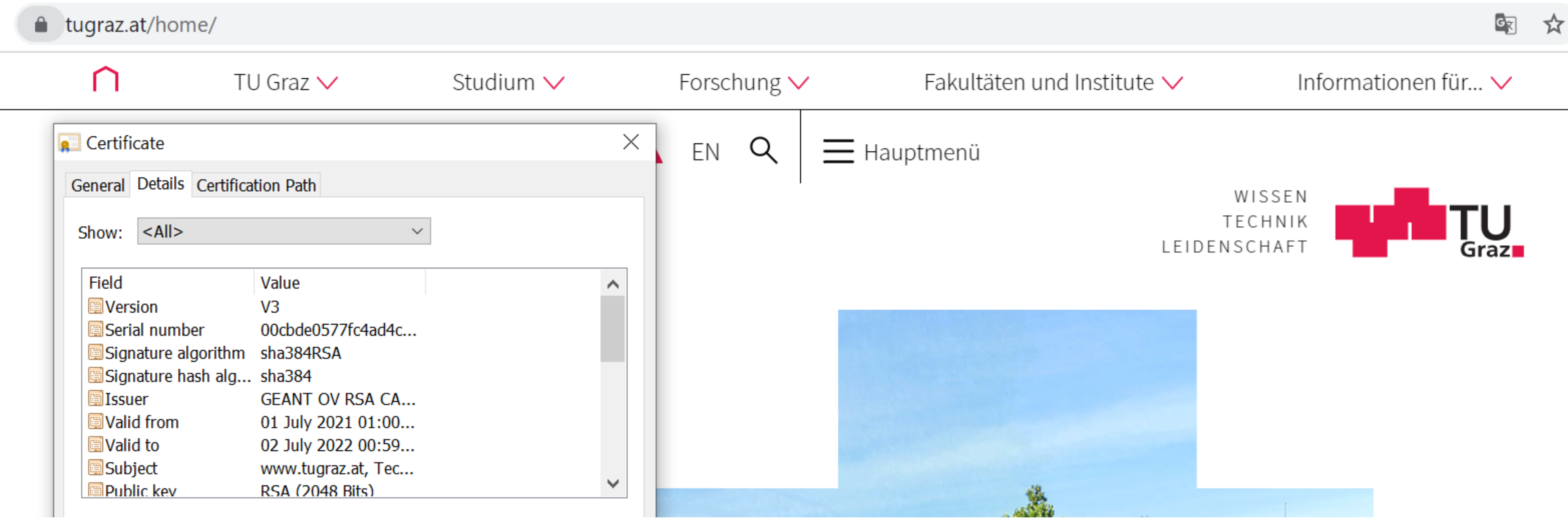
Sujoy Sinha Roy
sujoy.sinharoy@iaik.tugraz.at

Outline

1. Public-key cryptography basics
2. Lattice-based public-key encryption
3. Polynomial arithmetic



tugraz.at/home/
Security
tugraz.at
Connection is secure
Your information (for example, passwords or credit card numbers) is private when it is sent to this site. Learn more
Certificate is valid
ium
Forschung
Fakultäten und Institute
Informationen für...
EN
Hauptmenü
WISSEN
TECHNIK
LEIDENSCHAFT
TU
Graz


tugraz.at/home/
TU Graz
Studium
Forschung
Fakultäten und Institute
Informationen für...
EN
Hauptmenü
WISSEN
TECHNIK
LEIDENSCHAFT
TU
Graz
Certificate
General
Details
Certification Path
Show: <All>
Field
Value
Version
V3
Serial number
00cbde0577fc4ad4c...
Signature algorithm
sha384RSA
Signature hash alg...
sha384
Issuer
GEANT OV RSA CA...
Valid from
01 July 2021 01:00...
Valid to
02 July 2022 00:59...
Subject
www.tugraz.at, Tec...
Public key
RSA (2048 Bits)

Contemporary Cryptographic Primitives (examples)

Public-key Cryptography

- RSA
- Elliptic Curve

Symmetric-key Cryptography

- AES
- SHA-2 or SHA-3

Diffie-Hellman Key Agreement

Public info: Prime p and base g

Secret a

Secret b

$x = g^{a} \bmod p$

$y = g^{b} \bmod p$

Computes $y^{a} \bmod p$
$= g^{ab} \bmod p$

Computes $x^{b} \bmod p$
$= g^{ab} \bmod p$

Security is based on Discrete Log Problem (DLP)

Discrete Logarithm Problem

Given x, g and p, compute the secret a such that

$$x = g^a \bmod p$$

Latest record (Dec 2019) is 795-bit [BGGHTZ'19]
Using Intel Xeon Gold with 6130 CPUs.

Uses Number Field Sieve and takes 3100 core years using 1 CPU.

BBC Sign in News Sport Weather iPlayer Sounds

NEWS

Home UK World Business Politics Tech Science Health Family & Education

Technology

NSA 'developing code-cracking quantum computer'

Share

IBM Quantum Network Technology Resources

Death of public key cryptography???

Googl

The latest news from Gc

Quantum Supremacy Using a Programmable Superconducting Processor

Wednesday, October 23, 2019

Posted by John Martinis, Chief Scientist Quantum Hardware and Sergio Boixo, Chief Scientist Quantum Computing Theory, Google AI Quantum

to search

n

) may

both display "quantum primacy" over classical computers

BY CHARLES Q. CHOI | 06 NOV 2021 | 2 MIN READ

Post Quantum Public Key Cryptography

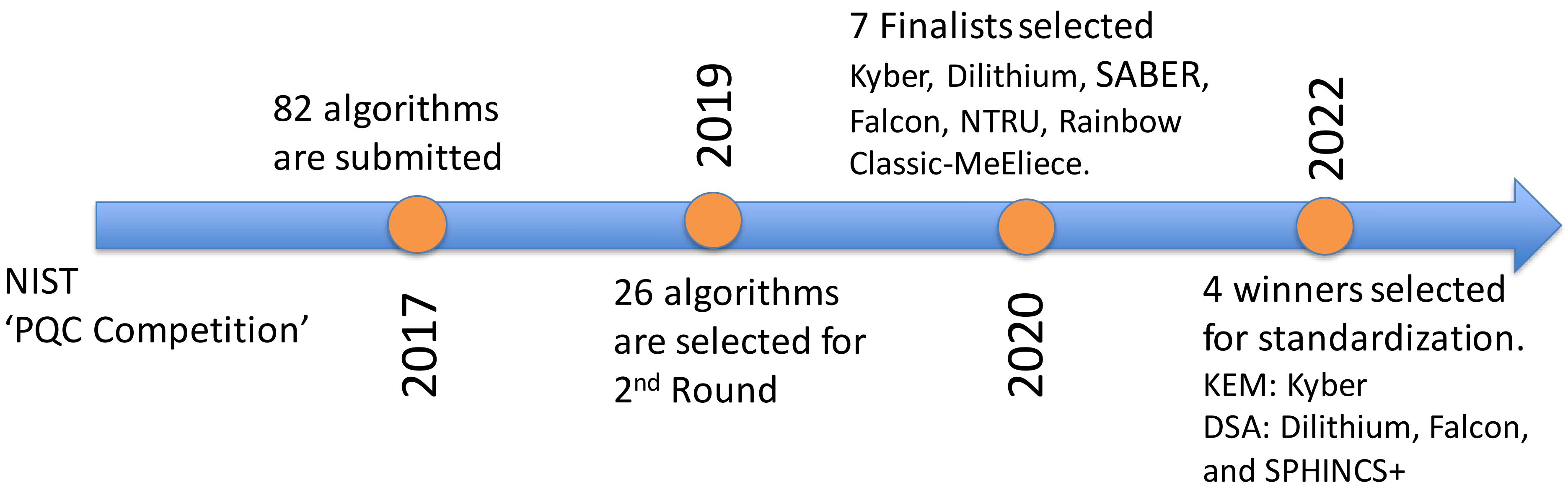
Post-quantum cryptographic (PQC) algorithms are designed using problems that are presumed to be unsolvable using quantum computers.

Currently 5 major problems are used for PQC.

- Lattice-based
- Code-based
- Multivariate-based
- Hash-based
- Isogeny-based

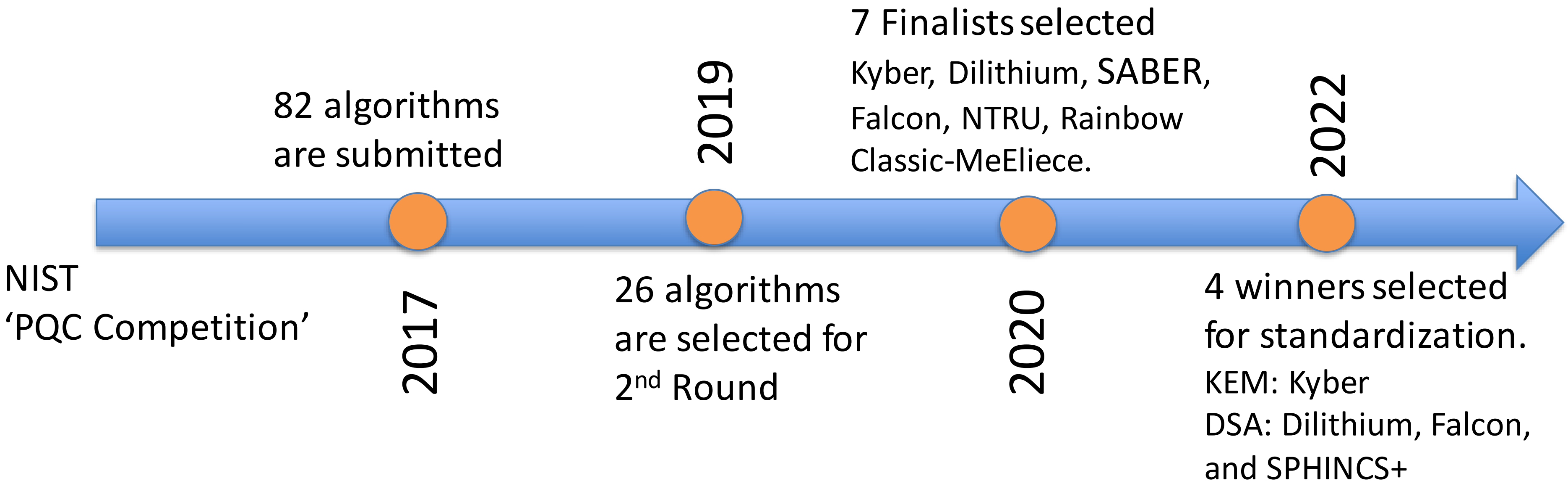
NIST Post Quantum Cryptography Standardization (2016-22)

NIST initiated PQC Standardization in 2016 and called for proposals.



NIST Post Quantum Cryptography Standardization (2016-22)

NIST initiated PQC Standardization in 2016 and called for proposals.



First three winners are all lattice-based. SPHINCS+ is hash-based.

NIST Post Quantum Cryptography Standardization (2022-)

To diversify portfolio of PQC algorithms, NIST called for additional PQC algorithms in 2022. There are around 40 new submissions.

- Code-based
  - Enhanced pqsigRM
  - FuLeeca
  - LESS
  - MEDS
  - Wave
- Isogenies
  - SQISign
- Lattices
  - EHT
  - EagleSign
  - HAETAE
  - HAWK
  - HuFu
  - Raccoon
  - Squirrels
- MPC-in-the-Head
  - CROSS
  - MIRA
  - MQOM
  - MiRitH
  - PERK
  - RYDE
  - SDitH
- Symmetric
  - AIMer
  - Ascon-Sign
  - FAEST
  - SPHINCS-alpha
- Multivariate
  - 3WISE
  - Biscuit
  - DME-Sign
  - HPPC
  - MAYO
  - PROV
  - QR-UOV
  - SNOVA
  - TUOV
  - UOV
  - VOX
- Other
  - ALTEQ
  - KAZ-Sign
  - PREON
  - Xifrat1-Sign.I
  - eMLE-Sig 2.0

Image source https://datatracker.ietf.org/meeting/117/materials/slides-117-tls-new-post-quantum-signature-algorithms-on-the-horizon-00

Outline

1. Public-key cryptography basics
2. **Lattice-based public-key encryption**
3. Polynomial arithmetic

In this course we will implement a simple lattice-based encryption scheme.

Lattice-based Cryptography – The LWE problem

Given two linear equations with unknown x and y

$$3x + 4y = 26$$
$$2x + 3y = 19$$

or

$$\begin{pmatrix} 3 & 4 \\ 2 & 3 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 26 \\ 19 \end{pmatrix}$$

Find x and y.

Solving System of Linear Equations

For an unknown vector ***s*** of size n

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix} \cdot \begin{pmatrix} s_1 \\ s_2 \\ \vdots \\ s_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \\ \vdots \\ b_m \end{pmatrix}$$

Gaussian elimination solves ***s*** when *the* number of equations $m \geq n$

Solving System of Linear Equations after *Error* is added

$$\underbrace{\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix}}_{\text{Public } \mathbf{A}} \cdot \underbrace{\begin{pmatrix} s_1 \\ s_2 \\ \vdots \\ s_n \end{pmatrix}}_{\text{Secret } s} + \underbrace{\begin{pmatrix} e_1 \\ e_2 \\ \vdots \\ e_n \\ \vdots \\ e_m \end{pmatrix}}_{\text{Error } e} = \underbrace{\begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \\ \vdots \\ b_m \end{pmatrix}}_{\text{Public } \mathbf{b}} \mod q$$

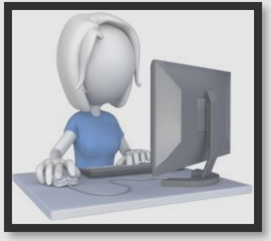
Learning With Errors (LWE) problem:

Given **(A, b)** → computationally infeasible to solve ***s***

Classical → Post-Quantum Diffie-Hellman key agreement

Public info: Prime p and base g

Secret a



$x = g^a \bmod p$ →

← $y = g^b \bmod p$

Secret b

Computes $y^a \bmod p = g^{ab} \bmod p$

Computes $x^b \bmod p = g^{ab} \bmod p$

Can we get a key agreement by replacing dLog with LWE problem?

LWE-based Diffie-Hellman Key-Exchange

Public uniformly random matrix **A** mod q

Small secret vector **[s]**
Small error vector **[e]**

$$b = A \times s + e$$

Small secret vector **[s']**
Small error vector **[e']**

$$b'^{T} = s'^{T} \times A + e'^{T}$$

$$v = b'^{T} \times s$$

Note: All operations are modulo *q*.

$$v' = s'^{T} \times b$$

Noisy **shared secret**

LWE-based Diffie-Hellman Key-Exchange (2)

What to do with the two ‘noisy’ integers?

$$v = b'^{T} \times s$$

$$v' = s'^{T} \times b$$

LWE-based Diffie-Hellman Key-Exchange (2)

What to do with the two ‘noisy’ integers?

This integer I is the same on both sides

v = Integer I + Noise E_1

v’ = Integer I + Noise E_2

E_1 and E_2 are quite small noise elements.

Most significant bit of v and v’ are equal with high probability → You get one key bit.

Ring-LWE problem

Given

$$a(x) * s(x) + e(x) = b(x) \pmod{q} \pmod{f(x)}$$

in a polynomial ring $R_q = \mathbb{Z}_q[x]/\langle f(x) \rangle$ where

$a(x)$: uniformly random public polynomial

$s(x)$: small secret polynomial

$e(x)$: small error polynomial

$b(x)$: output polynomial,

Ring-LWE problem:

Given $(a(x), b(x))$ → computationally infeasible to solve $s(x)$

$$f(x) = x^4 + 1$$

$$\begin{bmatrix} 1 & -4 & -3 & -2 \\ 2 & 1 & -4 & -3 \\ 3 & 2 & 1 & -4 \\ 4 & 3 & 2 & 1 \end{bmatrix}_{4\times 4} \times \begin{bmatrix} 5 \\ 6 \\ 7 \\ 8 \end{bmatrix} = \begin{bmatrix} 5 - 24 - 21 - 16 \\ 10 + 6 - 28 - 24 \\ 15 + 12 + 7 - 32 \\ 20 + 8 + 14 + 8 \end{bmatrix} = \begin{bmatrix} -56 \\ -36 \\ 2 \\ 60 \end{bmatrix}$$

$$-32x^2 - 52x - 61 + 60x^3 + 34x^2 + 16x + 5$$

$$= -56 - 36x + 2x^2 + 60x^3$$

$$a(x) = 1 + 2x + 3x^2 + 4x^3$$

$$b(x) = 5 + 6x + 7x^2 + 8x^3$$

Ring-LWE-based Diffie-Hellman Key-Exchange

Public polynomial a(x)

Small secret poly s(x)

Small error poly e(x)

Small secret poly s’(x)

Small error poly e’(x)

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

$b'(x) = a(x) \cdot s'(x) + e'(x)$

$v(x) = b'(x) \cdot s(x)$
$= a(x) \cdot s(x) \cdot s'(x) + e'(x) \cdot s(x)$

$v'(x) = b(x) \cdot s'(x)$
$= a(x) \cdot s(x) \cdot s'(x) + e(x) \cdot s'(x)$

Decoding v(x) gives n bits.

Decoding v’(x) gives n bits.

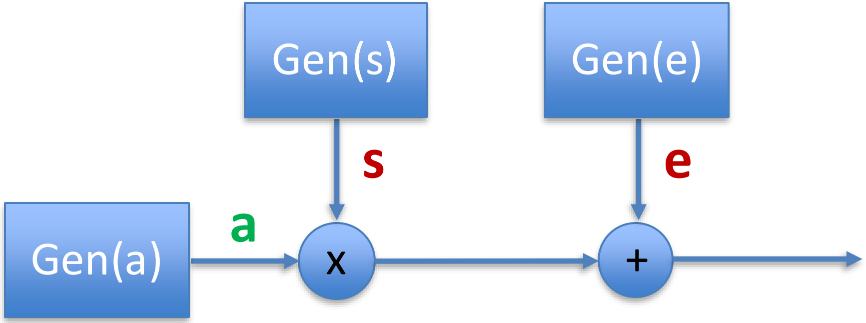
This course: Hardware implementation of Ring-LWE encryption

Ring-LWE (i.e., polynomials) is significantly more efficient than matrix LWE

Assignment 1: We implement ring-LWE public-key encryption (PKE)

Ring LWE-based Public-Key Encryption (PKE)

- **Key Generation:**
  - **Output:** public key (pk), secret key (sk)



b = a*s + e (Ring LWE sample)

Arithmetic operations are performed in a polynomial ring R_q

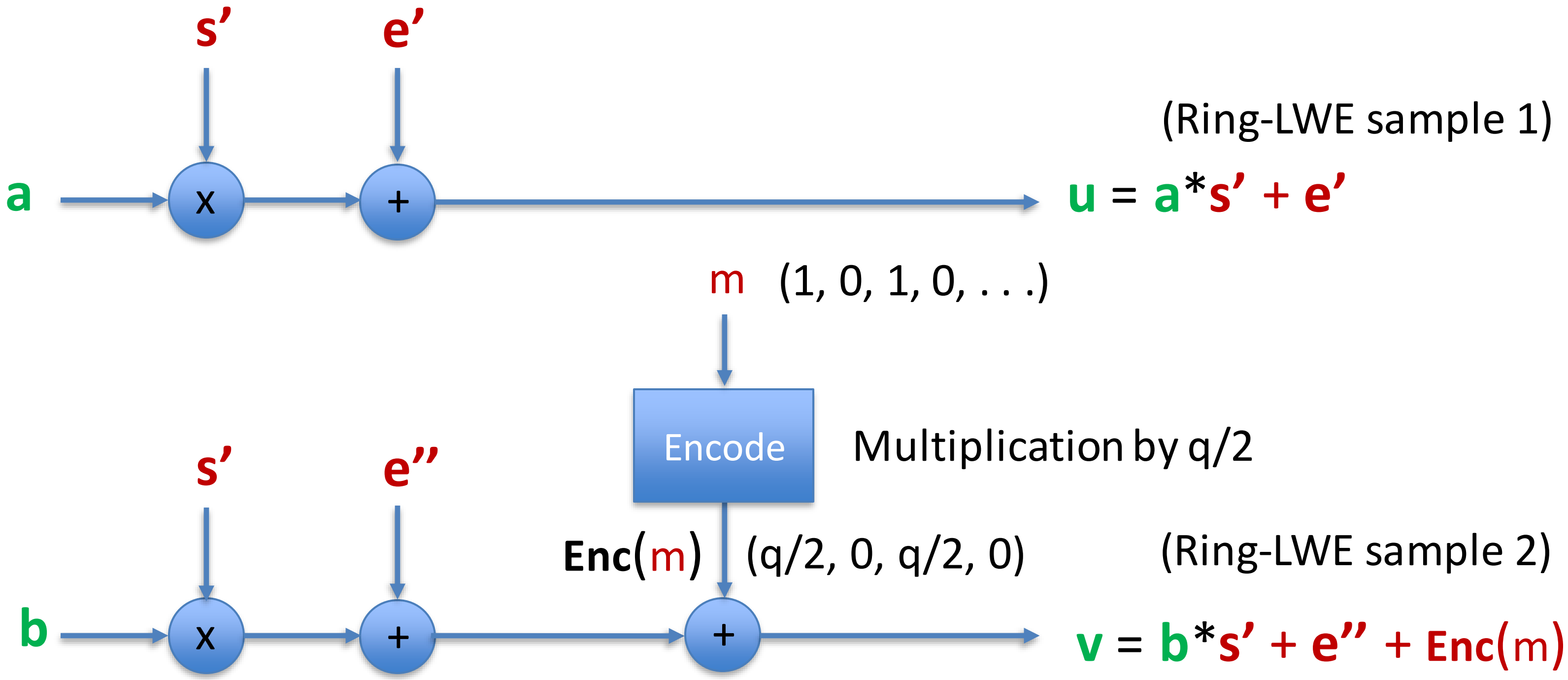
Public Key (pk): (a,b)

Secret Key (sk): (s)

V. Lyubashevsky, C. Peikert, and O. Regev. "On Ideal Lattices and Learning with Errors Over Rings". IACR ePrint 2012/230.

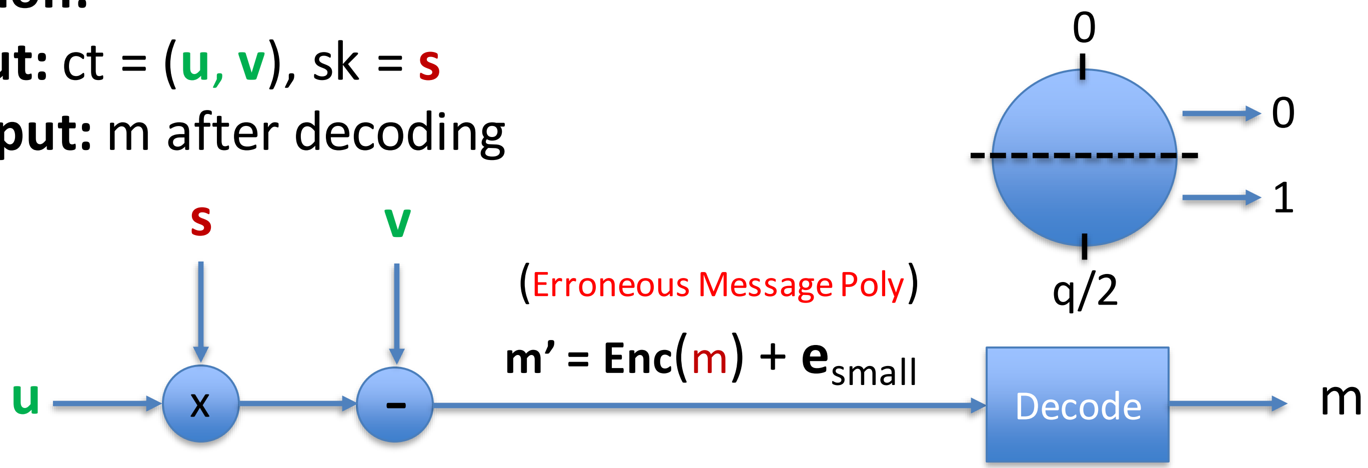
Ring LWE-based Public-Key Encryption (PKE)

- **Encryption:**
  - **Input: pk** = (a,b), message m
  - **Output: ct** = (u,v)



Ring LWE-based Public-Key Encryption (PKE)

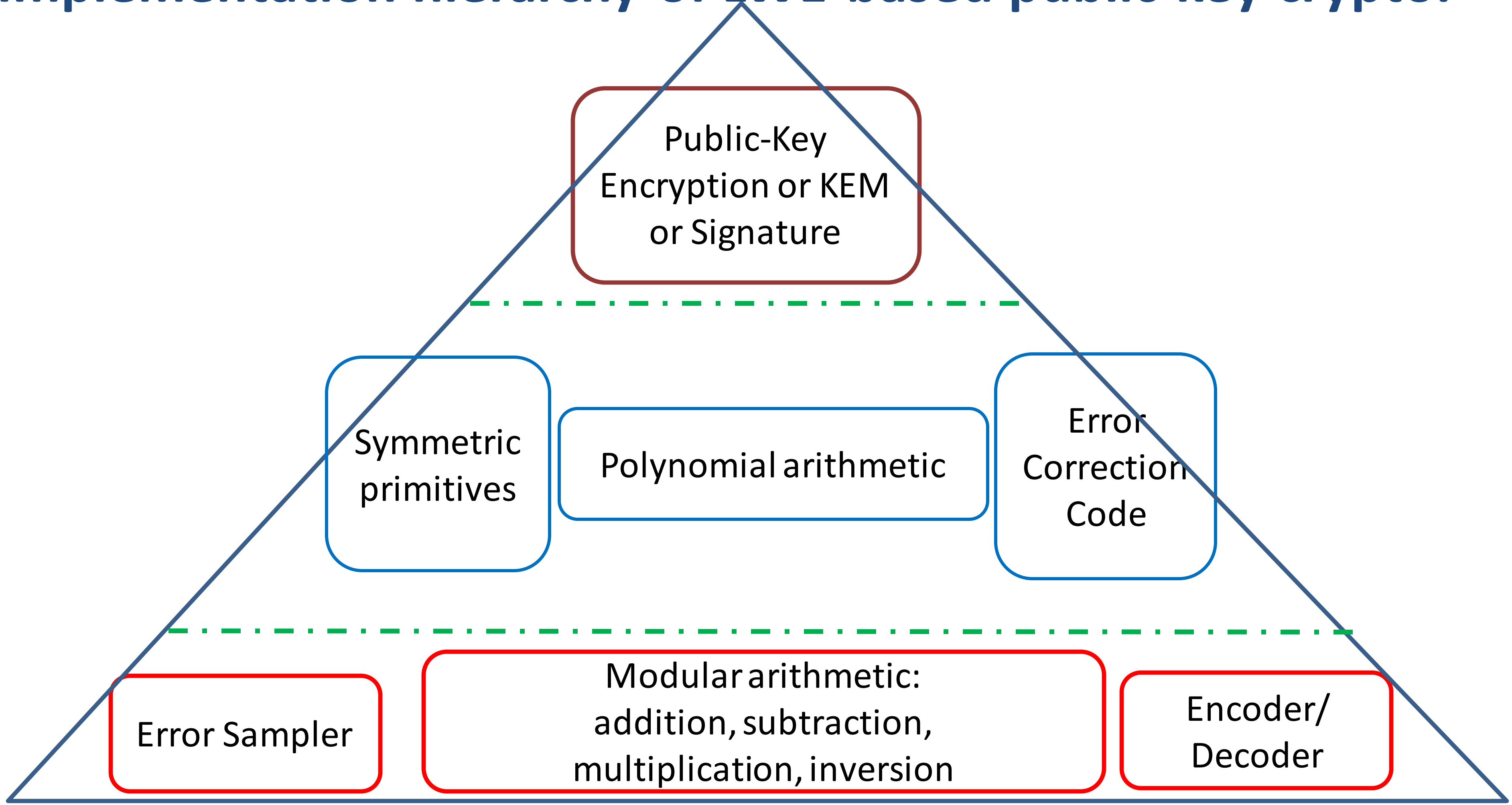
- **Decryption:**
  - **Input:** ct = (u, v), sk = s
  - **Output:** m after decoding



$$\mathbf{v} - \mathbf{u} * \mathbf{s} = \mathbf{m}' = \mathbf{Enc}(m) + (\mathbf{e} * \mathbf{s}' + \mathbf{e}'' + \mathbf{e}' * \mathbf{s})$$
$$= \mathbf{Enc}(m) + \mathbf{e}_{small}$$

Select most significant bit of each coefficient as the message bits

Implementation hierarchy of LWE-based public-key crypto.



Outline

1. Public-key cryptography basics
2. Lattice-based public-key encryption
3. **Polynomial arithmetic**

Mathematical background on Polynomial Arithmetic

Polynomial addition modulo q

Two polynomials are added coefficient-wise modulo q.

Example:

$$
\begin{array}{rl}
 & a(x) = 5x^3 + 4x^2 + 2x + 6 \pmod 7 \\
+ & b(x) = 3x^3 + 2x^2 + 5x + 2 \pmod 7 \\
\hline
\end{array}
$$

Polynomial addition modulo q

Two polynomials are added coefficient-wise modulo q.

Example:

$$
\begin{array}{rl}
 & a(x) = 5x^3 + 4x^2 + 2x + 6 \pmod 7 \\
+ & b(x) = 3x^3 + 2x^2 + 5x + 2 \pmod 7 \\
\hline
 & c(x) = 1x^3 + 6x^2 + 0x + 1 \pmod 7
\end{array}
$$

Polynomial multiplication modulo *q*

Usual way: Multiply each term in one polynomial by each term in the other polynomial and then sum them following the standard way.

$$a(x) = 5x^3 + 4x^2 + 2x + 6 \pmod 7$$

$$*$$

$$b(x) = 3x^3 + 2x^2 + 5x + 2 \pmod 7$$

Polynomial multiplication modulo *q*

Usual way: Multiply each term in one polynomial by each term in the other polynomial and then sum them following the standard way.

$$a(x) = 5x^3 + 4x^2 + 2x + 6 \pmod 7$$

$$* \quad b(x) = 3x^3 + 2x^2 + 5x + 2 \pmod 7$$

$$3x^3 + 1x^2 + 4x + 5$$

Polynomial multiplication modulo *q*

Usual way: Multiply each term in one polynomial by each term in the other polynomial and then sum them following the standard way.

$$
\begin{array}{rl}
 & a(x) = 5x^3 + 4x^2 + 2x + 6 \pmod 7 \\
* & b(x) = 3x^3 + 2x^2 + 5x + 2 \pmod 7 \\
\hline
 & 3x^3 + 1x^2 + 4x + 5 \\
 & 4x^4 + 6x^3 + 3x^2 + 2x
\end{array}
$$

Polynomial multiplication modulo *q*

Usual way: Multiply each term in one polynomial by each term in the other polynomial and then sum them following the standard way.

$$a(x) = 5x^3 + 4x^2 + 2x + 6 \pmod 7$$

$$* \quad b(x) = 3x^3 + 2x^2 + 5x + 2 \pmod 7$$

$$3x^3 + 1x^2 + 4x + 5$$

$$4x^4 + 6x^3 + 3x^2 + 2x$$

$$3x^5 + 1x^4 + 4x^3 + 5x^2$$

Polynomial multiplication modulo *q*

Usual way: Multiply each term in one polynomial by each term in the other polynomial and then sum them following the standard way.

$$a(x) = 5x^3 + 4x^2 + 2x + 6 \pmod 7$$

$$*$$

$$b(x) = 3x^3 + 2x^2 + 5x + 2 \pmod 7$$

$$3x^3 + 1x^2 + 4x + 5$$

$$4x^4 + 6x^3 + 3x^2 + 2x$$

$$3x^5 + 1x^4 + 4x^3 + 5x^2$$

$$1x^5 + 5x^5 + 6x^4 + 4x^3$$

Polynomial multiplication modulo *q*

Usual way: Multiply each term in one polynomial by each term in the other polynomial and then sum them following the standard way.

$$a(x) = 5x^3 + 4x^2 + 2x + 6 \pmod 7$$

$$* \quad b(x) = 3x^3 + 2x^2 + 5x + 2 \pmod 7$$

$$3x^3 + 1x^2 + 4x + 5$$

$$4x^4 + 6x^3 + 3x^2 + 2x$$

$$3x^5 + 1x^4 + 4x^3 + 5x^2$$

$$1x^5 + 5x^5 + 6x^4 + 4x^3$$

$$c(x) = 1x^6 + 1x^5 + 4x^4 + 3x^3 + 2x^2 + 6x + 5 \pmod 7$$

Coefficient-wise addition mod 7

Modular reduction of a polynomial by a polynomial

Let's say, we want to modulo reduce this polynomial

$$c(x) = 1x^6 + 1x^5 + 4x^4 + 3x^3 + 2x^2 + 6x + 5 \pmod 7$$

by the following polynomial

$$f(x) = x^4 + 1 \pmod 7.$$

Modular reduction of a polynomial by a polynomial

Let’s say, we want to modulo reduce this polynomial

$$c(x) = 1x^6 + 1x^5 + 4x^4 + 3x^3 + 2x^2 + 6x + 5 \pmod 7$$

by the following polynomial

$$f(x) = x^4 + 1 \pmod 7.$$

Any term in c(x) with degree ≥ deg(f) will get reduced by f(x) using the congruence relation:

$$x^4 = -1 \pmod 7$$

Modular reduction of a polynomial by a polynomial

Let's say, we want to modulo reduce this polynomial

$$c(x) = 1x^6 + 1x^5 + \boxed{4x^4} + 3x^3 + 2x^2 + 6x + 5 \pmod{7}$$

by the following polynomial

$$f(x) = x^4 + 1 \pmod{7}.$$

Any term in c(x) with degree ≥ deg(f) will get reduced by f(x) using the congruence relation:

$$x^4 = -1 \pmod{7}$$

Example:

$$4x^4 = 4\cdot(-1) \pmod{7}$$
$$= 3 \pmod{7}$$

Modular reduction of a polynomial by a polynomial

Let's say, we want to modulo reduce this polynomial

$$c(x) = \boxed{1x^6} + \boxed{1x^5} + 4x^4 + 3x^3 + 2x^2 + 6x + 5 \pmod 7$$

by the following polynomial

$$f(x) = x^4 + 1 \pmod 7.$$

Any term in c(x) with degree ≥ deg(f) will get reduced by f(x) using the congruence relation:

$$x^4 = -1 \pmod 7$$

Similarly, $1x^5 = 6x \pmod 7$

and $1x^6 = 6x^2 \pmod 7$

Modular reduction of a polynomial by a polynomial

Let's say, we want to modulo reduce this polynomial

$$c(x) = 1x^6 + 1x^5 + 4x^4 + 3x^3 + 2x^2 + 6x + 5 \pmod 7$$

by the following polynomial

$$f(x) = x^4 + 1 \pmod 7.$$

After reduction by $f(x)$

$$6x^2 + 6x + 3$$

Hence, $c(x) \bmod f(x) = (6x^2 + 6x + 3) + (3x^3 + 2x^2 + 6x + 5)$

$$= 3x^3 + 1x^2 + 5x + 1 \pmod 7 \pmod f$$

[Definition] Polynomial ring $R_q = \mathbb{Z}_q[x]/<f(x)>$

- The polynomial ring has its irreducible polynomial $f(x)$ of degree n.
  → Hence all ring-elements are polynomials of degree n-1.

- Closed under polynomial addition and multiplication.
  →For two polynomials $a(x)$ and $b(x)$ $\in R_q$

$$c(x) = a(x) + b(x) \pmod{q} \pmod{f} \in R_q$$

and

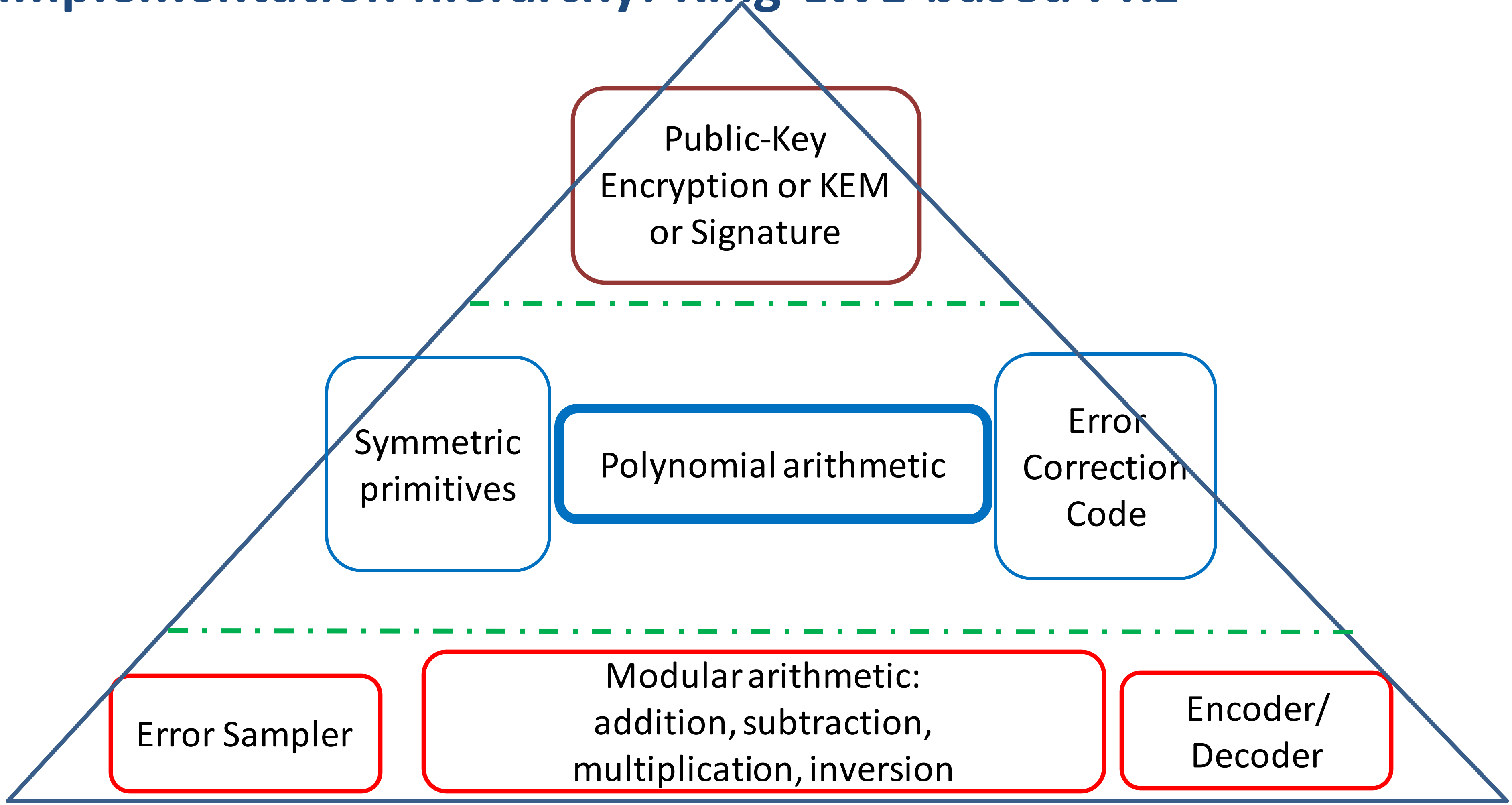
$$c(x) = a(x) * b(x) \pmod{q} \pmod{f} \in R_q$$

- Identity element under the addition rule is the 0-polynomial.
- Identity element under the multiplication rule is the 1-polynomial
- Multiplicative inverse of a polynomial may not exist.

From now on we assume all multiplications are in $R_q = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$

→ This simplifies modular reduction by $f(x) = x^n + 1$
→ and makes an implementation more efficient

Implementation hierarchy: Ring-LWE-based PKE



How to multiply two polynomials?

We can use the following algorithms and also combinations of them

- Schoolbook multiplication: $O(n^2)$
- Karatsuba multiplication: $O(n^{1.585})$
- Fast Fourier Transform (FFT) multiplication: $O(n \log n)$

Schoolbook method of polynomial multiplication

$$
\begin{array}{rr}
 & a(x) = 5x^3 + 4x^2 + 2x + 6 \pmod 7 \\
* & b(x) = 3x^3 + 2x^2 + 5x + 2 \pmod 7 \\
\hline
 & 3x^3 + 1x^2 + 4x + 5 \\
 & 4x^4 + 6x^3 + 3x^2 + 2x \\
 & 3x^5 + 1x^4 + 4x^3 + 5x^2 \\
 & 1x^5 + 5x^5 + 6x^4 + 4x^3 \\
\hline
\end{array}
$$

$$c(x) = 1x^6 + 1x^5 + 4x^4 + 3x^3 + 2x^2 + 6x + 5 \pmod 7$$

We learnt this method during algebra classes in school.

+ Simple structure makes it easy to implement.

- Time complexity is $O(n^2)$, which is the worst of all three algorithms.

GP/Pari code for Schoolbook polynomial multiplication (1)

```
N = 2^8;     /* Polynomial degree */
q = 7681;  /* Coefficient modulus */
firr = Mod(1, q)*x^N + Mod(1, q);  /* Irreducible polynomial modulus */

schoolbook(a, b) = {

   /* Schoolbook polynomial multiplication  c = a*b has two nested loops */
   c = 0;

      for(i=0, N-1,
         for(j=0, N-1,
            mval = polcoeff(b, j)*polcoeff(a,i) % q;
            c = c + mval*x^(j+i)));

   c = c%firr;

   return (c);
}
```

https://pari.math.u-bordeaux.fr/gp.html

GP/Pari code for Schoolbook polynomial multiplication (2)

```
test() = {
   /* Formation of random polynomial a(x) with coefficients mod q */
   a = 0;
   for(i=0, N-1, a = a + random(q)*x^i);

   /* Formation of random polynomial b(x) with coefficients mod q */
   b = 0;
   for(i=0, N-1, b = b + random(q)*x^i);

   c= schoolbook(a, b);

   /* Native polynomial multiplication d = a*b.    */
   d = a*b % firr;


   print("c = ", c);
   print("d = ", d);
   print("c-d = ", c-d);        /* If correct, then c-d will be 0. */
}

test();
```

https://pari.math.u-bordeaux.fr/gp.html

Architecture for Schoolbook polynomial multiplication

E.g., polynomial degree $N = 256$ and $f(x) = x^{256} + 1$.

Algorithm: Schoolbook algorithm

$acc(x) \leftarrow 0$

for $i = 0; i < 256; i$++ **do**

  for $j = 0; j < 256; j$++ **do**

    $acc[j] = acc[j] + b[j] \cdot a[i]$

  $b = b \cdot x \bmod \langle x^{256} + 1 \rangle$

return acc

How will you implement the algo as an architecture in HW?

Architecture for Schoolbook polynomial multiplication

E.g., polynomial degree $N = 256$ and $f(x) = x^{256} + 1$.

Algorithm: Schoolbook algorithm

$acc(x) \leftarrow 0$

for $i = 0; i < 256; i{+}{+}$ **do**

  for $j = 0; j < 256; j{+}{+}$ **do**

    $acc[j] = acc[j] + b[j] \cdot a[i]$

  $b = b \cdot x \bmod \langle x^{256} + 1 \rangle$

return acc

How will you implement the algo as an architecture in HW?

- What are the fundamental elementary operations?

Architecture for Schoolbook polynomial multiplication

E.g., polynomial degree $N = 256$ and $f(x) = x^{256} + 1$.

Algorithm: Schoolbook algorithm

$acc(x) \leftarrow 0$

for $i = 0; i < 256; i{+}{+}$ **do**

  for $j = 0; j < 256; j{+}{+}$ **do**

    $acc[j] = acc[j] + b[j] \cdot a[i]$

  $b = b \cdot x \bmod \langle x^{256} + 1 \rangle$

return acc

Multiply and Accumulate (MAC)

How will you implement the algo as an architecture in HW?

- What are the fundamental elementary operations?
- Draw an architecture for MAC

Architecture for Schoolbook polynomial multiplication

E.g., polynomial degree $N = 256$ and $f(x) = x^{256} + 1$.

Algorithm: Schoolbook algorithm

$acc(x) \leftarrow 0$

for $i = 0; i < 256; i{+}{+}$ **do**

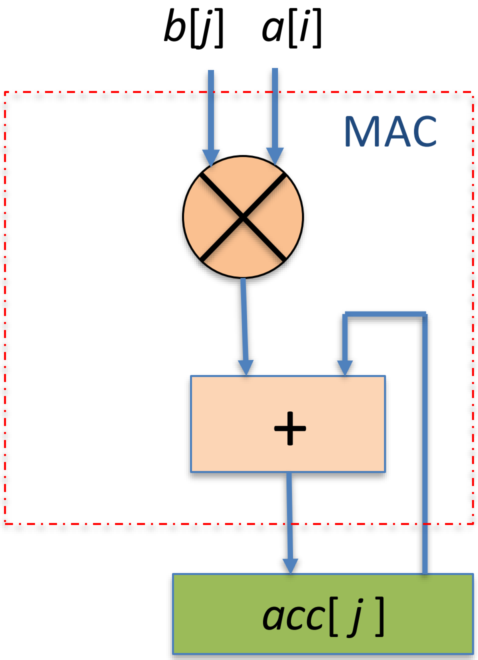
  for $j = 0; j < 256; j{+}{+}$ **do**

    $acc[j] = acc[j] + b[j] \cdot a[i]$

  $b = b \cdot x \bmod \langle x^{256} + 1 \rangle$

return acc

Architecture of MAC unit



Architecture for Schoolbook polynomial multiplication

E.g., polynomial degree $N = 256$ and $f(x) = x^{256} + 1$.

Algorithm: Schoolbook algorithm

$acc(x) \leftarrow 0$

for $i = 0; i < 256; i{+}{+}$ **do**

  for $j = 0; j < 256; j{+}{+}$ **do**

    $acc[j] = acc[j] + b[j] \cdot a[i]$

  $b = b \cdot x \bmod \langle x^{256} + 1 \rangle$

return acc

How to implement this step?

Architecture for Schoolbook polynomial multiplication

E.g., polynomial degree N = 256 and $f(x) = x^{256} + 1$.

Algorithm: Schoolbook algorithm

$acc(x) \leftarrow 0$

for $i = 0; i < 256; i{+}{+}$ **do**

  for $j = 0; j < 256; j{+}{+}$ **do**

    $acc[j] = acc[j] + b[j] \cdot a[i]$

  $b = b \cdot x \bmod \langle x^{256} + 1 \rangle$

return acc

How to implement this step?

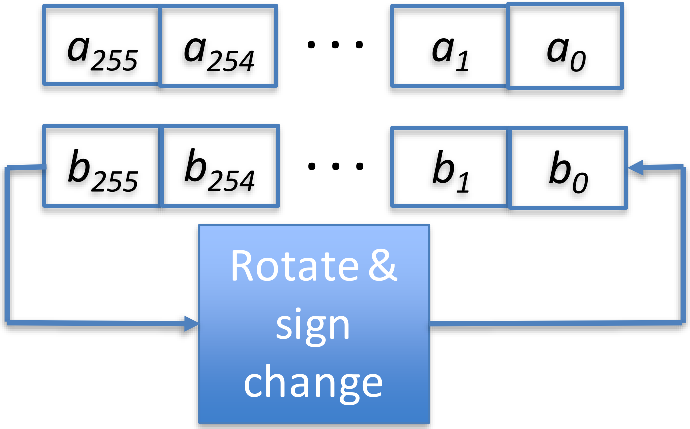
With mod $f(x) = x^n + 1$, we have $x^n \equiv -1$, hence multiplying

$b(x) = b_{n-1}x^{n-1} + \ldots + b_0 \pmod{f(x)}$ by x gives

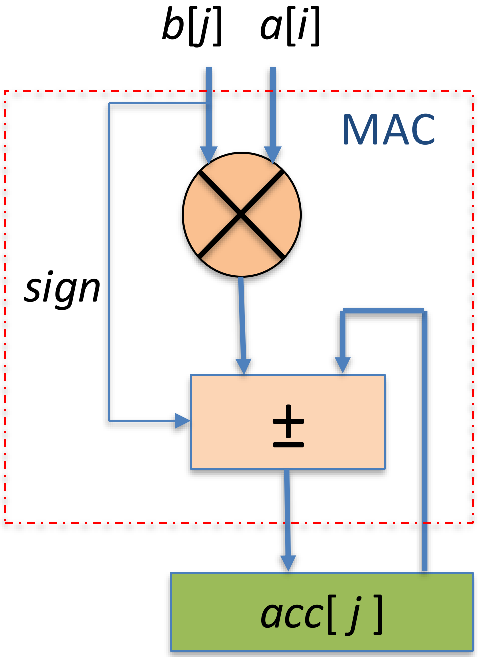
$x \cdot b(x) = b_{n-2}x^{n-1} + \ldots + b_0 x - b_{n-1} \pmod{f(x)}$ →Rotation with sign change.

Architecture for Schoolbook polynomial multiplication

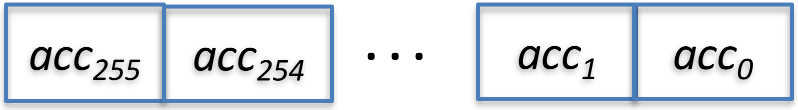
Ring-buffer registers



Apply this MAC( ) one by one.



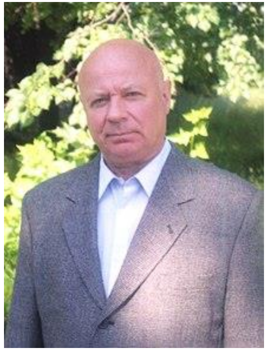
Note: This is just an idea. This may **not** be an optimized architecture!



Karatsuba method of polynomial multiplication

Andrey Kolmogorov
(1903-1987)

In 1960, during a seminar at Moscow State University, Kolmogorov conjectured that multiplying two integers have $O(n^2)$ complexity.



Anatoly Karatsuba
(1937-2008)

Karatsuba, then a 23 years old student, attended the seminar and within a week came up with a divide-and-conquer method for multiplying two integers with $O(n^{\log_2 3})$ complexity.

The method was published in the Proceedings of the USSR Academy of Sciences in 1962.

Karatsuba method of polynomial multiplication (1)

Split each operand into two halve-size polynomials:

$$a(x) = \underbrace{a_{n\text{-}1}\, x^{n\text{-}1} + \;\ldots\; + a_{n/2}\, x^{n/2}}_{a_h(x)} + \underbrace{a_{n/2\text{-}1}\, x^{n/2\text{-}1} + \ldots\; + a_1 x + a_0}_{a_l(x)}$$

Hence, we can write:

$$a(x) = a_h(x)\, x^{n/2} + a_l(x) = a_h x^{n/2} + a_l$$

Karatsuba method of polynomial multiplication (2)

After splitting we have:

$$a(x) = a_h x^{n/2} + a_l$$

$$b(x) = b_h x^{n/2} + b_l$$

Naïve method: We can compute the result using the *Schoolbook* method

$$a(x) * b(x) = a_h b_h x^n + (a_h b_l + a_l b_h) x^{n/2} + a_l b_l$$

It performs 4 multiplication and has a quadratic complexity.

Karatsuba showed how to compute this using 3 multiplications.

Karatsuba method of polynomial multiplication (3)

After splitting we have:

$$a(x) = a_h x^{n/2} + a_l$$

$$b(x) = b_h x^{n/2} + b_l$$

Karatsuba method:

$$a(x) * b(x) = a_h b_h x^n + (a_h b_l + a_l b_h) x^{n/2} + a_l b_l$$

It computes $(a_h b_l + a_l b_h)$ term by performing only one multiplication as:

$$(a_h b_l + a_l b_h) = (a_h + a_l)\cdot(b_h + b_l) - \underbrace{a_h b_h - a_l b_l}$$

These two produces are reused from the above.

Karatsuba method of polynomial multiplication (3)

After splitting we have:

$$a(x) = a_h x^{n/2} + a_l$$

$$b(x) = b_h x^{n/2} + b_l$$

Karatsuba method:

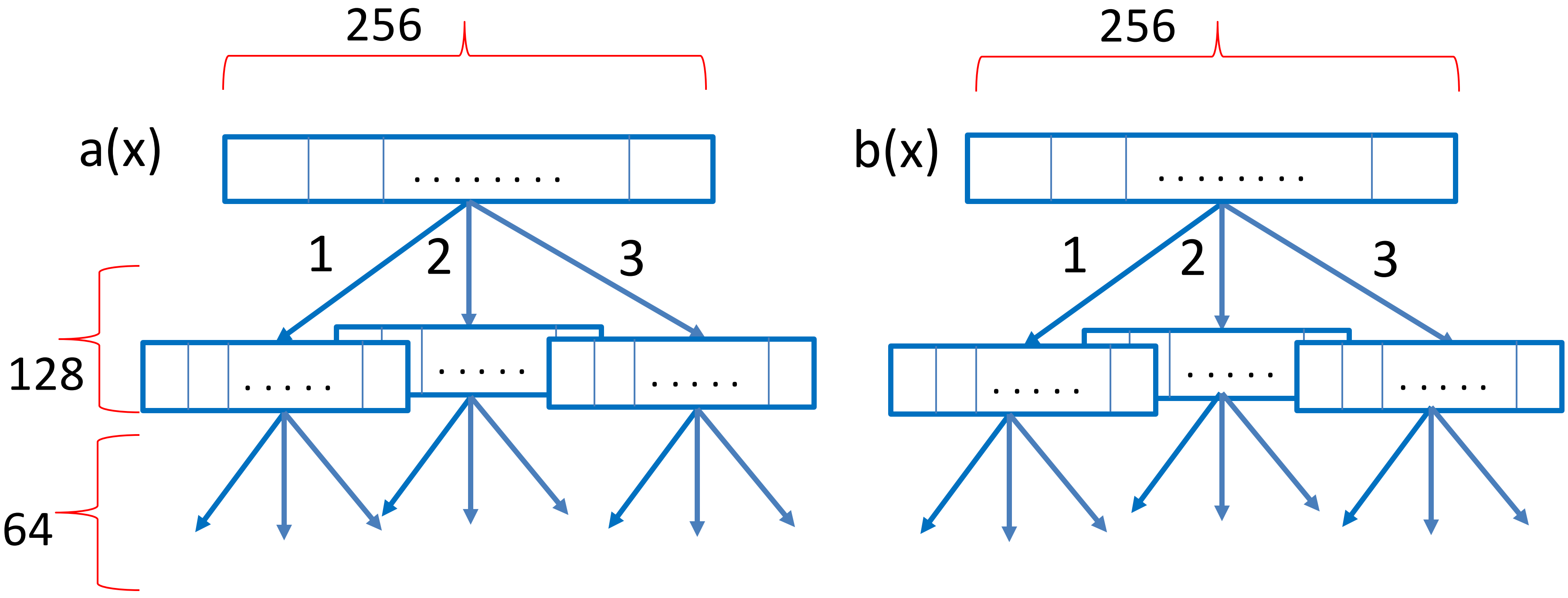
$$a(x) * b(x) = a_h b_h x^n + (a_h b_l + a_l b_h) x^{n/2} + a_l b_l$$

It computes $(a_h b_l + a_l b_h)$ term by performing only one multiplication as:

$$(a_h b_l + a_l b_h) = (a_h + a_l) \cdot (b_h + b_l) - a_h b_h - a_l b_l$$

Hence, the three multiplications are:
$a_h b_h$, $a_l b_l$, and $(a_h + a_l) \cdot (b_h + b_l)$.

Divide-and-Conquer approach: Karatsuba tree



- Recursively apply divide-and-conquer strategy
- When the polynomials are of sufficiently-small size, multiply them
- And return to the higher levels

Complexity of Karatsuba polynomial multiplication

Let, T_n be the time for multiplication two *n*-coefficient polynomials.

$$
\begin{aligned}
T_n &= 3T_{n/2} \\
&= 3^2 T_{n/4} \\
&= 3^3 T_{n/8} \\
&= \ldots \\
&= 3^{\log_2 n} T_1
\end{aligned}
$$

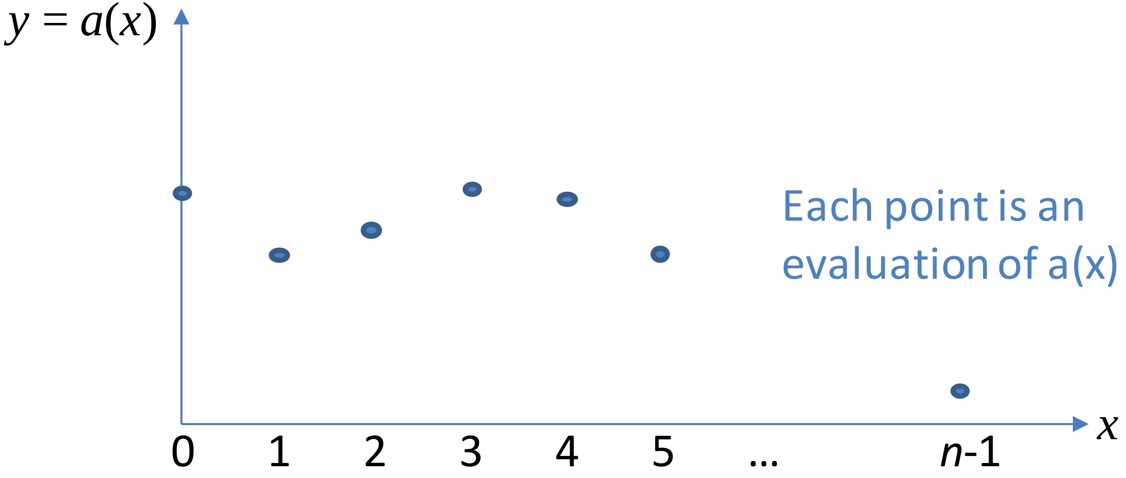
Hence, the complexity = $O(3^{\log_2 n}) = O(n^{\log_2 3}) \approx O(n^{1.585})$

The idea of FFT

Representation: Polynomial ↔ Point values

Given a polynomial a(x) we can easily compute its evaluations at *n* points

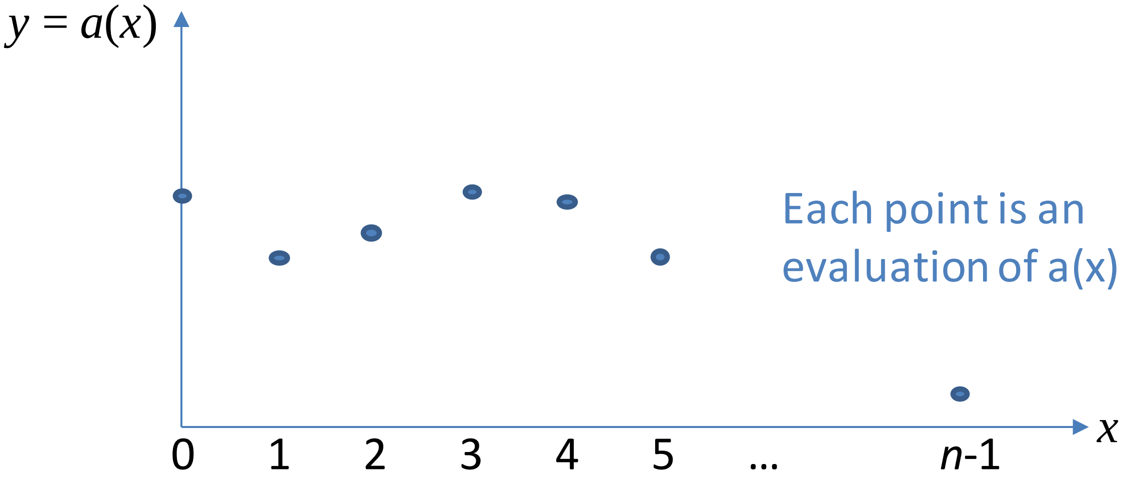
$$a(x) = a_{n\text{-}1}\, x^{n\text{-}1} + \ldots + a_1 x + a_0$$



Representation: Polynomial ↔ Point values

Given *n* distinct evaluation points $y_0, y_1, \ldots, y_{n-1}$ can we get $a(x)$?

$$a(x) = ?$$



Representation: Polynomial ⟷ Point values

What we have as $y_0, y_1, \ldots, y_{n-1}$ are:

$$y_0 = a(0) = a_{n-1}\, 0^{n-1} + \ldots + a_2 0^2 + a_1 0 + a_0$$

$$y_1 = a(1) = a_{n-1}\, 1^{n-1} + \ldots + a_2 1^2 + a_1 1 + a_0$$

$$\ldots$$

$$y_{n-1} = a(n-1) = a_{n-1}\, (n-1)^{n-1} + \ldots + a_2(n-1)^2 + a_1(n-1) + a_0$$

$$\underbrace{\begin{bmatrix} 0^0 & 0^1 & 0^2 & \ldots & 0^{n-1} \\ 1^0 & 1^1 & 1^2 & \ldots & 1^{n-1} \\ 2^0 & 2^1 & 2^2 & \ldots & 2^{n-1} \\ & & \ldots & & \end{bmatrix}}_{V} * \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \ldots \end{bmatrix} = \begin{bmatrix} a(0) \\ a(1) \\ a(2) \\ \ldots \end{bmatrix}$$

(V^{-1} performs the opposite)

Polynomial → Point values

$$\begin{pmatrix} a(0) \\ a(1) \\ a(2) \\ \dots \\ a(n\text{-}1) \end{pmatrix} = \begin{pmatrix} 0^0 & 0^1 & 0^2 & \dots & 0^{n\text{-}1} \\ 1^0 & 1^1 & 1^2 & \dots & 1^{n\text{-}1} \\ 2^0 & 2^1 & 2^2 & \dots & 2^{n\text{-}1} \\ & & \dots & & \\ (n\text{-}1)^0 & & & & (n\text{-}1)^{n\text{-}1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \dots \\ a_{n\text{-}1} \end{pmatrix}$$

Points

Polynomial coefficients

Given a polynomial, calculating the *n* distinct points is called ‘evaluation’.

Point values → Polynomial

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \dots \\ a_{n-1} \end{bmatrix} = \begin{pmatrix} 0^0 & 0^1 & 0^2 & \dots & 0^{n-1} \\ 1^0 & 1^1 & 1^2 & \dots & 1^{n-1} \\ 2^0 & 2^1 & 2^2 & \dots & 2^{n-1} \\ & & \dots & & \\ (n-1)^0 & & & & (n-1)^{n-1} \end{pmatrix}^{-1} \begin{bmatrix} a(0) \\ a(1) \\ a(2) \\ \dots \\ a(n-1) \end{bmatrix}$$

Polynomial coefficients

Points

Given n distinct points, calculating the polynomial is called ‘interpolation’.

Rules: Polynomial ↔ Point values

1. Interpolation will succeed in obtaining a(x) only if there are n distinct evaluations $y_0, \ldots, y_{n-1}$.

2. You can choose any values for x as long as you get n distinct y_i.

Application of DFT in polynomial multiplication

$$a(x) = a_0 + a_1 x + \ldots + a_{n\text{-}1} x^{n\text{-}1}$$
$$b(x) = b_0 + b_1 x + \ldots + b_{n\text{-}1} x^{n\text{-}1} \qquad \times$$

$$c(x) = a(x)*b(x) = c_0 + c_1 x + \ldots + c_{n\text{-}1} x^{n\text{-}1} + \ldots + c_{2n\text{-}2} x^{2n\text{-}2}$$

Polynomial c(x) has degree 2*n*-2.

→ Therefore c(x) can be represented as 2*n*-1 discrete points.

Application of DFT in polynomial multiplication

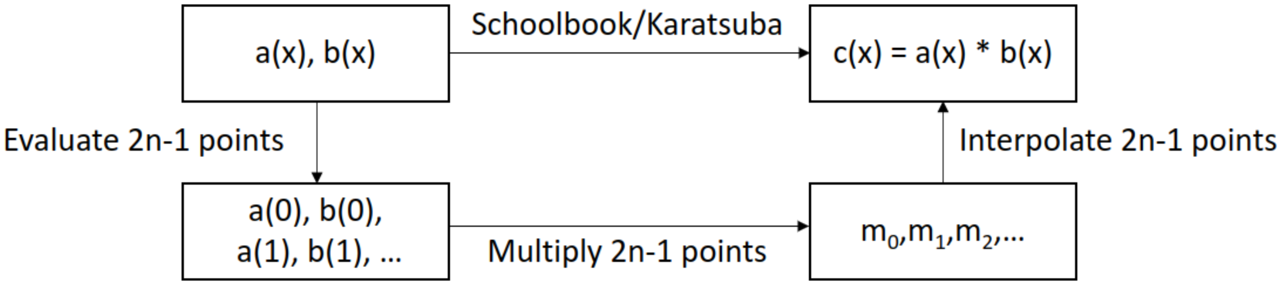
- For c(x) = a(x) * b(x) where a(x) and b(x) have degree of n-1:
  - Evaluate a(x) and b(x) at 2n-1 points
  - Multiply evaluated points $m_i = a(i).b(i)$
  - Use Lagrange's interpolating polynomials to reconstruct c(x)

$$c(x) = a(x) * b(x) = \sum_{i=0}^{2n-2} i.\, L_i(x) \text{ where } L_i(x) = \prod_{i \neq j} \frac{x-j}{i-j}$$

Application of DFT in polynomial multiplication

- For c(x) = a(x) * b(x) where a(x) and b(x) have degree of n-1:
  - Evaluate a(x) and b(x) at 2n-1 points
  - Multiply evaluated points m_i = a(i).b(i)
  - Use Lagrange's interpolating polynomials to reconstruct c(x)

$$c(x) = a(x) * b(x) = \sum_{i=0}^{2n-2} i.L_i(x) \quad \text{where} \quad L_i(x) = \prod_{i \neq j} \frac{x-j}{i-j}$$



Application of DFT in polynomial multiplication

- Observation: If we can perform evaluation and interpolation operations fast, then we can multiply two polynomials fast.
  - Can we use DFT to perform these operations?

Application of DFT in polynomial multiplication

- Observation: If we can perform evaluation and interpolation operations fast, then we can multiply two polynomials fast.
  - Can we use DFT to perform these operations?

- Discrete Fourier Transform (DFT)
  - A transformation $(a_0, a_1, \ldots, a_{n-2}, a_{n-1})$ --> $(A_0, A_1, \ldots, A_{n-2}, A_{n-1})$

$$A_k = \sum_{j=0}^{n-1} a_j . e^{\left(-\frac{(2i\pi)}{n}\right).k.j}$$

  - $\omega = e^{-i2\pi/n}$ is n-th primitive root of 1 (unity) which satisfies $\omega^n = 1$

$\omega^k \neq 1$ for $1 \leq k < n$

Application of DFT in polynomial multiplication

- We can choose our evaluation points as powers of ω

$$\underbrace{\begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \dots & \omega^{0\text{-}1} \\ \omega^0 & \omega^1 & \omega^2 & \dots & \omega^{n-1} \\ \omega^0 & \omega^2 & \omega^4 & \dots & \omega^{2n-2} \\ & & \dots & & \\ & & & & \end{bmatrix}}_{V(\omega)} * \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \\ \dots \end{bmatrix} = \begin{bmatrix} a(\omega^0) \\ a(\omega^1) \\ a(\omega^2) \\ \\ \dots \end{bmatrix}$$

$$V(\omega) * V(\omega^{-1}) = n * I$$

$$V(\omega^{-1}) = n * V(\omega)^{-1}$$

$$V(\omega)^{-1} = (1/n) * V(\omega^{-1})$$

With V(ω) (**DFT**), we compute *evaluation*

With V(ω)-1 or (1/n) * $V(\omega^{-1})$ (**IDFT**), we compute *interpolation*

Application of DFT in polynomial multiplication

- We can choose our evaluation points as powers of ω

$$\underbrace{\begin{bmatrix} \omega^0\ \omega^0\ \omega^0 & \dots\ \omega^{0-1} \\ \omega^0\ \omega^1\ \omega^2 & \dots\ \omega^{n-1} \\ \omega^0\ \omega^2\ \omega^4 & \dots\ \omega^{2n-2} \\ & \dots \end{bmatrix}}_{V(\omega)} * \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \dots \end{bmatrix} = \begin{bmatrix} a(\omega^0) \\ a(\omega^1) \\ a(\omega^2) \\ \dots \end{bmatrix}$$

$$V(\omega) * V(\omega^{-1}) = n * I$$

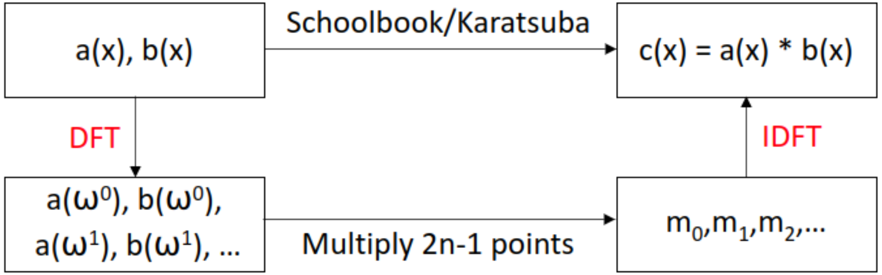
$$V(\omega^{-1}) = n * V(\omega)^{-1}$$

$$V(\omega)^{-1} = (1/n) * V(\omega^{-1})$$

With V(ω) (**DFT**), we compute *evaluation*

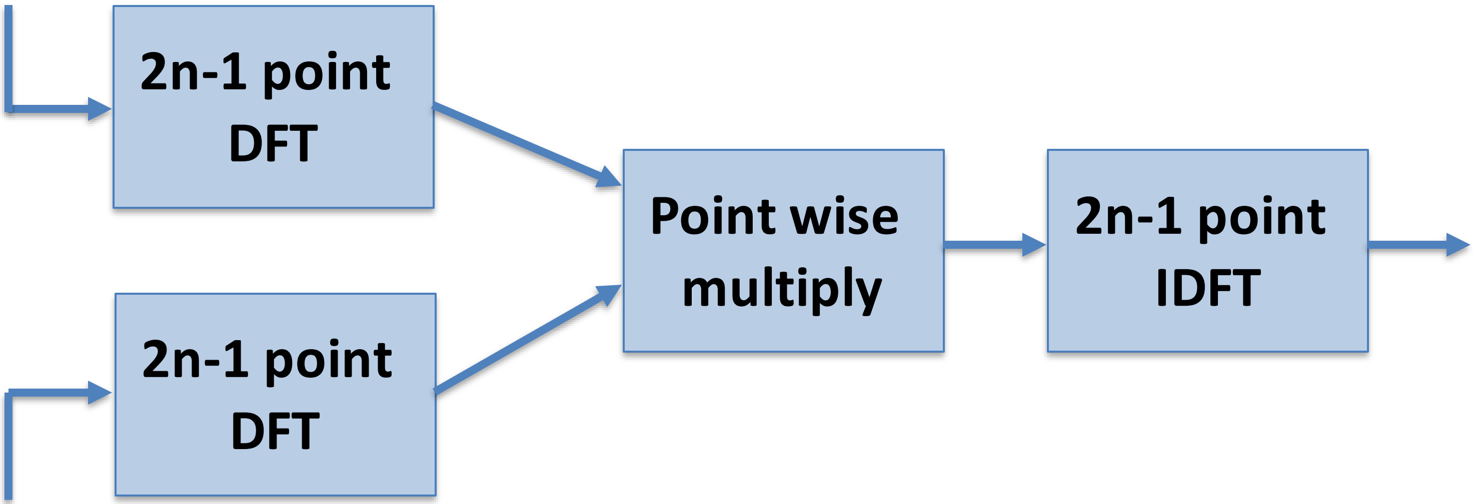
With V(ω)-1 or (1/n) * $V(\omega^{-1})$ (**IDFT**), we compute *interpolation*

- We can use DFT and IDFT for evaluation and interpolation.



Summary: DFT-base polynomial multiplication

$a(x) = a_{n\text{-}1}\, x^{n\text{-}1} + \ldots + a_0$



$b(x) = b_{n\text{-}1}\, x^{n\text{-}1} + \ldots + b_0$

$c(x) = c_{2n\text{-}2}\, x^{2n\text{-}2} + \ldots + c_0$

What is the complexity of Discrete Fourier Transform (DFT) ?

Answer: $O(n^2)$

Fast Fourier Transform (FFT) computes it ‘fast’ in $O(n \log n)$

Fast Fourier Transform (FFT)

The n-point FFT evaluates $a(x) = a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$

at n *special* points: $x = \omega_n^k = e^{-i2\pi k/n}$ for $k = 0, \ldots, n-1$ where $\omega_n = e^{-i2\pi/n}$ is the n^{th} primitive root of 1 i.e., $\omega_n^n = 1$.

With these special points, we can **reuse intermediate values** to do fewer computation in total.

Fast Fourier Transform (FFT)

The n-point FFT evaluates $a(x) = a_{n-1}x^{n-1} + \ldots + a_1x + a_0$

at n *special* points: $x = \omega_n^k = e^{-i2\pi k/n}$ for $k = 0, \ldots, n\text{-}1$ where $\omega_n = e^{-i2\pi/n}$ is the n^{th} primitive root of 1.

Interesting mathematical property FFT uses:

$$\omega_n^{n/2} = -1$$

Fast Fourier Transform (FFT)

The n-point FFT evaluates $a(x) = a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$

at n *special* points: $x = \omega_n^k = e^{-i2\pi k/n}$ for $k = 0, \ldots, n-1$ where $\omega_n = e^{-i2\pi/n}$ is the n^{th} primitive root of 1.

Interesting mathematical property FFT uses:

$$\omega_n^{n/2} = -1$$

We can rewrite

$$\begin{aligned} a(x) &= a_{n-1}x^{n-1} + \ldots + a_1 x + a_0 \\ &= (\ldots + a_4 x^4 + a_2 x^2 + a_0) + (\ldots + a_5 x^4 + a_3 x^2 + a_1)x \\ &= a_{even}(x^2) + x a_{odd}(x^2) \end{aligned}$$

Fast Fourier Transform (FFT)

Interesting mathematical property FFT uses:

$$\omega_n^{n/2} = -1$$

We can rewrite

$$a(x) = a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$$
$$= (\ldots +a_4x^4 + a_2x^2 + a_0) + (\ldots +a_5x^4 + a_3x^2 + a_1)x$$
$$= a_{even}(x^2) + x a_{odd}(x^2)$$

Based on the above,

$$y_k = a(\omega^k) = a_{even}(\omega^{2k}) + \omega^k a_{odd}(\omega^{2k})$$

and

$$y_{k+n/2} = a(\omega^{k+n/2}) = a_{even}(\omega^{2k+n}) + \omega^{k+n/2} a_{odd}(\omega^{2k+n})$$
$$= a_{even}(\omega^{2k}) - \omega^k a_{odd}(\omega^{2k})$$

Fast Fourier Transform (FFT)

Interesting mathematical property FFT uses:

$$\omega_n^{n/2} = -1$$

We can rewrite

$$a(x) = a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$$
$$= (\ldots + a_4 x^4 + a_2 x^2 + a_0) + (\ldots + a_5 x^4 + a_3 x^2 + a_1)x$$
$$= a_{even}(x^2) + x a_{odd}(x^2)$$

Based on the above,

FFT reuses them

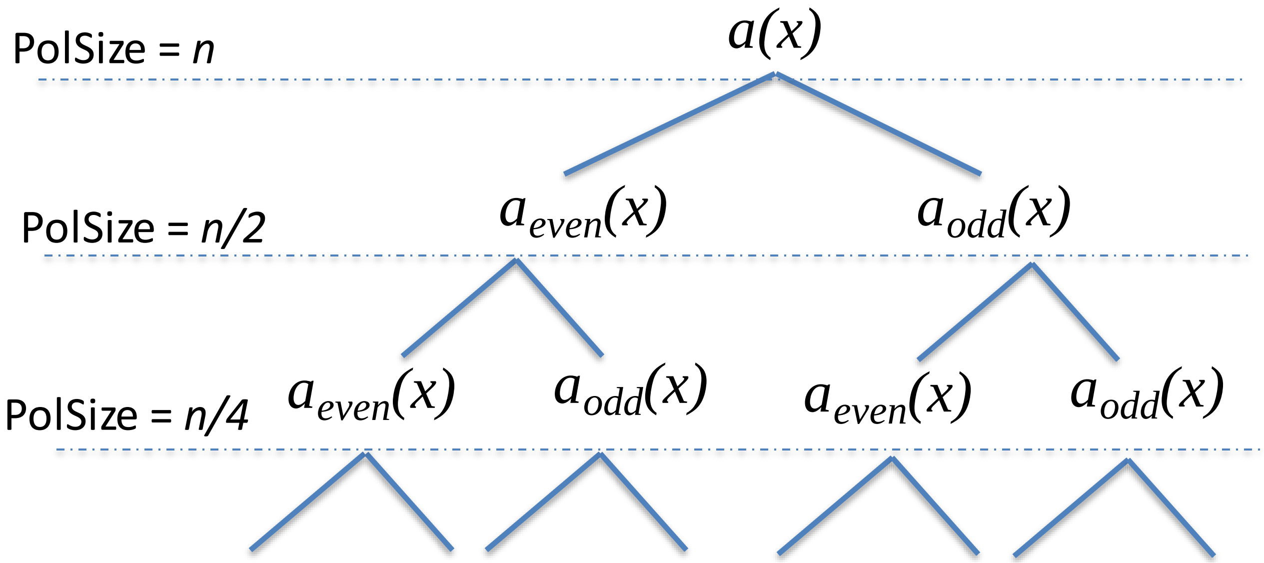
$$y_k = a(\omega^k) = \boxed{a_{even}(\omega^{2k})} + \boxed{\omega^k a_{odd}(\omega^{2k})}$$

and

$$y_{k+n/2} = a(\omega^{k+n/2}) = a_{even}(\omega^{2k+n}) + \omega^{k+n/2} a_{odd}(\omega^{2k+n})$$
$$= \boxed{a_{even}(\omega^{2k})} - \boxed{\omega^k a_{odd}(\omega^{2k})}$$

Complexity of FFT

Uses divide and conquer approach



Each level in the tree has O(n) cost. There are log(n) levels.
Total cost = O(n log n)

FFT to Number Theoretic Transform (NTT)

- FFT involves arithmetic of real numbers

  It evaluates at powers of $e^{-i2\pi/n}$ where $e^{-i2\pi/n}$ is the complex n^{th} primitive root of the unity.

- Number Theoretic Transform (NTT)

  NTT replaces $e^{-i2\pi/n}$ by an n^{th} primitive root of the unity modulo q where q is a prime satisfying $q \equiv 1 \bmod n$ and n is a power-of-2.

→ Only ***integer arithmetic*** modulo q

Number Theoretic Transform (NTT)

- An n-point NTT takes a(x) as an input and generates:

$$\mathbf{a}(x) = \sum_{i=0}^{n-1} \mathcal{A}_i . x^i \quad \text{where} \quad \mathcal{A}_i = \sum_{j=0}^{n-1} a_j . \omega^{i.j}$$

ω: n^{th} root of unity (**twiddle factor**) satisfying $\omega^n \equiv 1 (\text{mod } q)$

$\omega^i \neq 1 (\text{mod } q) \; \forall i < n$

$q \equiv 1 \; (\text{mod } n)$

Number Theoretic Transform (NTT)

- An n-point NTT takes a(x) as an input and generates:

$$\mathbf{a}(x) = \sum_{i=0}^{n-1} \mathcal{A}_i . x^i \quad \text{where} \quad \mathcal{A}_i = \sum_{j=0}^{n-1} a_j . \omega^{i.j}$$

ω: n^{th} root of unity (**twiddle factor**) satisfying $\omega^n \equiv 1 (\text{mod } q)$

$\omega^i \neq 1 (\text{mod } q) \; \forall i < n$

$q \equiv 1 \; (\text{mod } n)$

- Inverse NTT (INTT) operation uses a similar formula.

$$\text{a}(x) = \sum_{i=0}^{n-1} a_i . x^i \quad \text{where} \quad a_i = \frac{1}{n} . \sum_{j=0}^{n-1} \mathcal{A}_j . \omega^{-i.j}$$

Number Theoretic Transform (NTT)

- Example (NTT for n=4):

$$\mathcal{A}_0 = a_0 + a_1 + a_2 + a_3$$

$$\mathcal{A}_1 = a_0 + a_1.\omega^1 + a_2.\omega^2 + a_3.\omega^3$$

$$\mathcal{A}_2 = a_0 + a_1.\omega^2 + a_2.\omega^4 + a_3.\omega^6$$

$$\mathcal{A}_3 = a_0 + a_1.\omega^3 + a_2.\omega^6 + a_3.\omega^9$$

Using $\omega^4 = 1$

$\omega^2 = -1$

Number Theoretic Transform (NTT)

- Example (NTT for n=4):

$$\mathcal{A}_0 = a_0 + a_1 + a_2 + a_3$$
$$\mathcal{A}_1 = a_0 + a_1.\omega^1 + a_2.\omega^2 + a_3.\omega^3$$
$$\mathcal{A}_2 = a_0 + a_1.\omega^2 + a_2.\omega^4 + a_3.\omega^6$$
$$\mathcal{A}_3 = a_0 + a_1.\omega^3 + a_2.\omega^6 + a_3.\omega^9$$

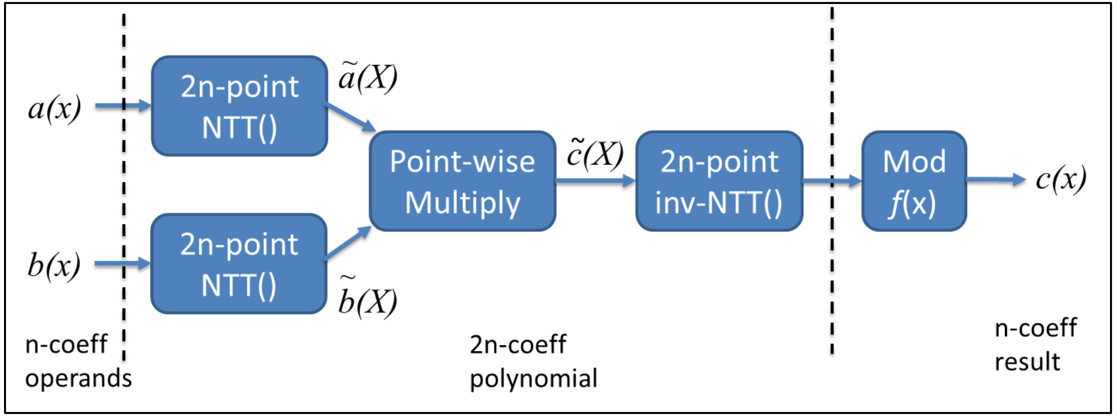
$$\mathcal{A}_0 = a_0 + a_1 + a_2 + a_3$$
$$\mathcal{A}_1 = a_0 + a_1.\omega^1 - a_2 - a_3.\omega^2$$
$$\mathcal{A}_2 = a_0 - a_1 + a_2 - a_3$$
$$\mathcal{A}_3 = a_0 - a_1.\omega - a_2 + a_3.\omega^1$$

Using $\omega^4 = 1$

$\omega^2 = -1$

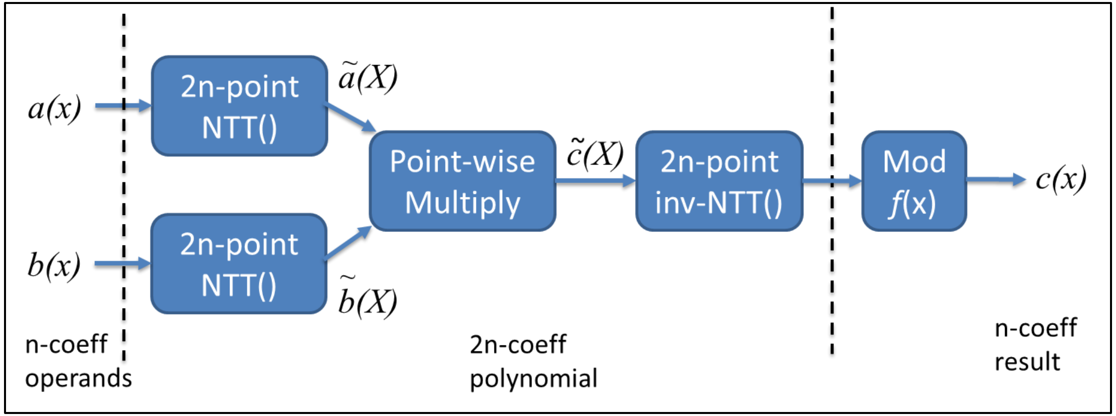
An optimization in NTT: Negative-wrapped convolution

Polynomial multiplication in $R_q = \mathbb{Z}_q[x]/\langle f(x) \rangle$ where q is a prime satisfying $q \equiv 1 \pmod{n}$ is as follows:

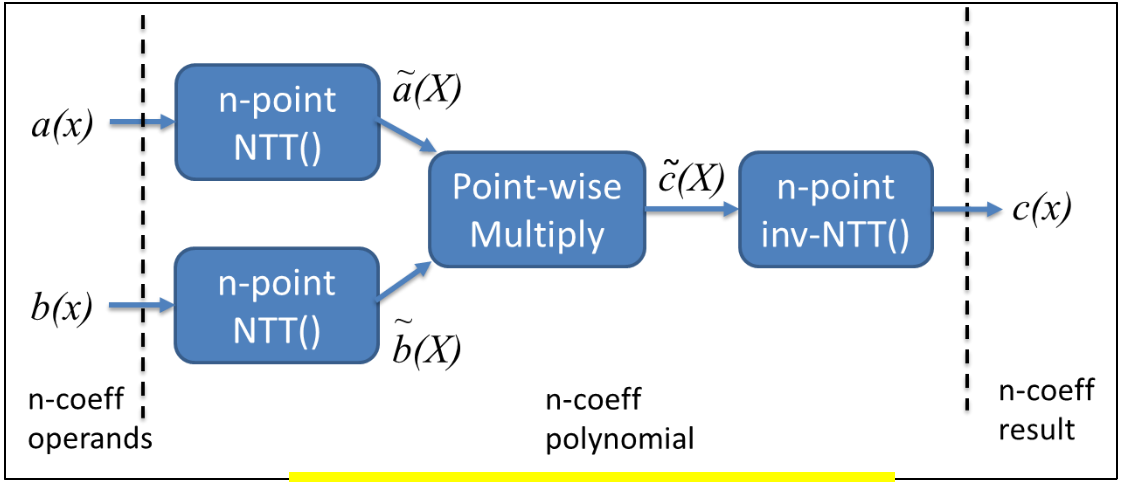


An optimization in NTT: Negative-wrapped convolution

Polynomial multiplication in $R_q = \mathbb{Z}_q[x]/\langle f(x) \rangle$ where q is a prime satisfying $q \equiv 1 \pmod{n}$ is as follows:



Polynomial multiplication in $R_q = \mathbb{Z}_q[x]/\langle f(x) \rangle$ where q is a prime satisfying $q \equiv 1 \pmod{2n}$, and $f(x) = x^n + 1$ is as follows:

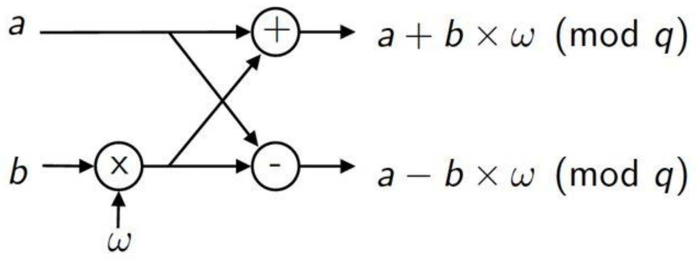


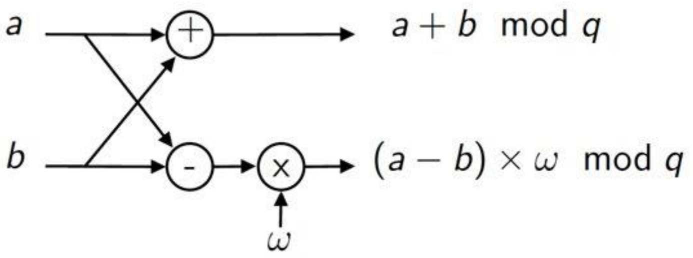
Negative-wrapped convolution

An optimization in NTT: Negative-wrapped convolution

- Two main approaches to perform fast NTT:
  - Decimation-in-time (DIT) with Cooley-Tukey butterfly structure
  - Decimation-in-frequency (DIF) with Gentleman-Sande butterfly structure

- For n-pt NTT, there are log(n) stages where each stage performs n/2 butterfly operations





Explaining NTT using the Chinese Remainder Theorem (CRT)

https://electricdusk.com/ntt.html

(Optional study material. Not essential for this course)

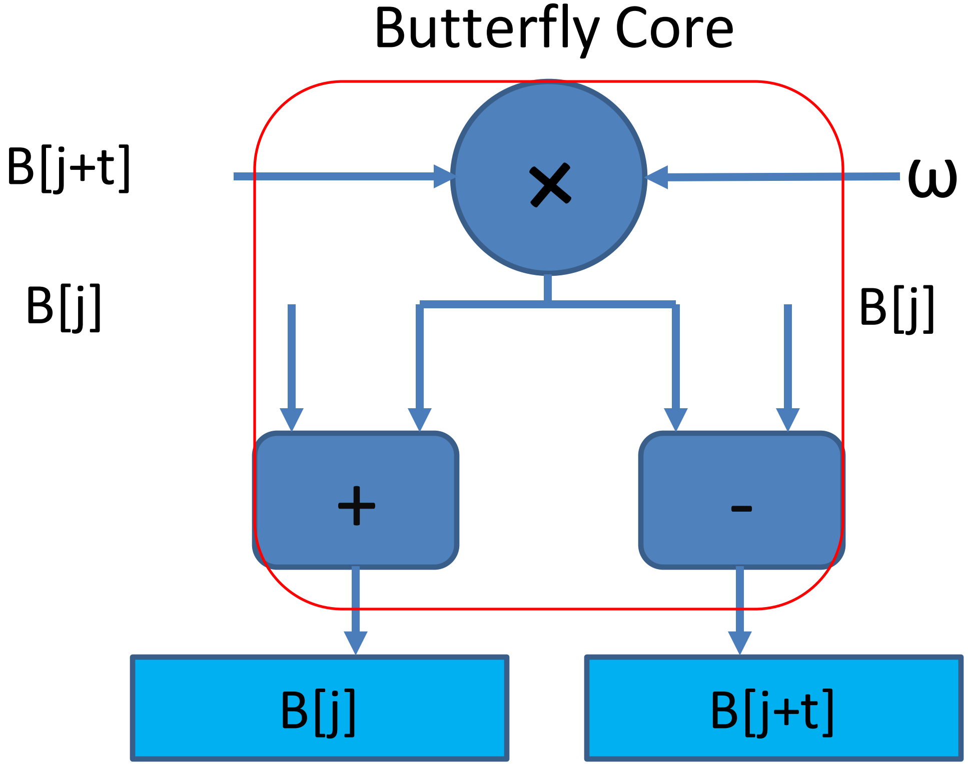
Python code of NTT-based multiplication is available on the course page.

Forward NTT Pseudocode

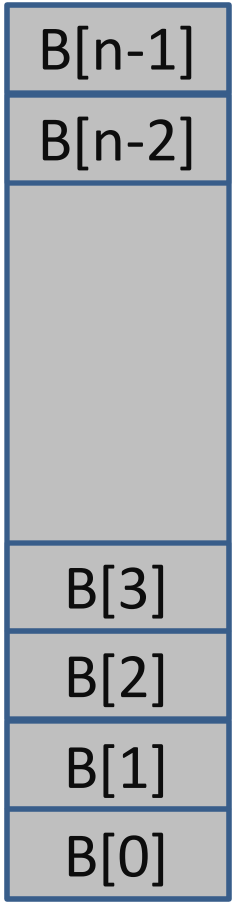
```
fntt(B[ ] of size N):
   t = N
   m = 1
   while(m<N):
       t = int(t/2)
       for i in range(m):
               j1 = 2*i*t
               j2 = j1 + t - 1
               psi_pow = int_bitreverse(m+i)  # Bits in the reverse order

               W = psi_table[psi_pow]

               for j in range(j1,j2+1):              # Cooley-Tukey butterfly operation
                       U = B[j]
                       V = (B[j+t]*W)  % q
                       B[j]   = (U+V) % q
                       B[j+t] = (U-V) % q
       m = 2*m
return B
```

Butterfly circuit for forward NTT

```
# Cooley-Tukey butterfly operation

for j in range(j1,j2+1):
        U = B[j]
        V = (B[j+t]*W) % q
        B[j]   = (U+V) % q
        B[j+t] = (U-V) % q
```



NTT and Memory access



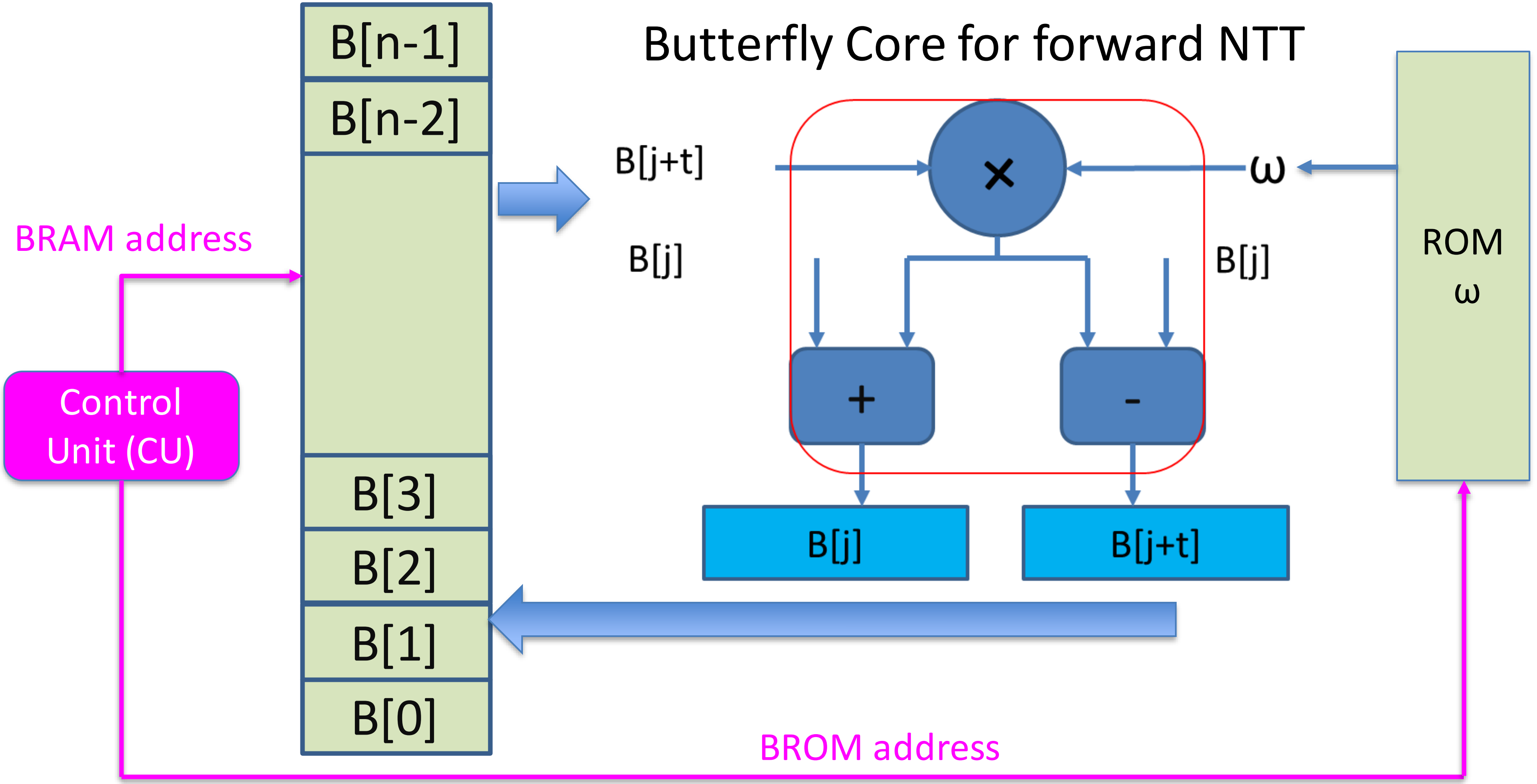
Simplified NTT loops

```
Loop m {
   Loop i {
      Loop j {
            Butterfly(B[j],B[j+t]);
      }
   }
}
```

Butterfly() reads two coefficients from memory.

Butterfly() writes two coefficients to memory.

NTT in HW

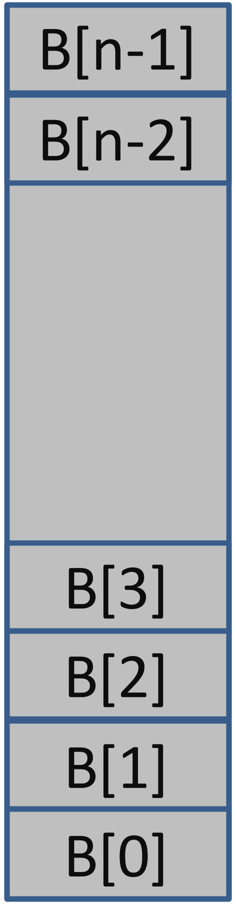


B[n-1]
B[n-2]
B[3]
B[2]
B[1]
B[0]
Butterfly Core for forward NTT
B[j+t]
B[j]
×
ω
B[j]
+
-
B[j]
B[j+t]
ROM
ω
BRAM address
Control Unit (CU)
BROM address

Inverse NTT Pseudocode

```
intt(B[ ] of size N):
    t = 1
    m = N
     while(m>1):
        j1 = 0
        h = int(m/2)
        for i in range(h):
            j2 = j1 + t - 1
            psi_pow = int_bitreverse(h+i,l)
            W = psi_inv_table[psi_pow]

            for j in range(j1,j2+1):
                # Gentleman-Sande butterfly operation
                U = B[j]
                V = B[j+t]
                B[j]   = (U+V) % q
                B[j+t] = (U-V)*W % q
            j1 = j1 + 2*t
        t = 2*t
        m = int(m/2)
        # ... (Division by N)
    return B
```

NTT and Memory access



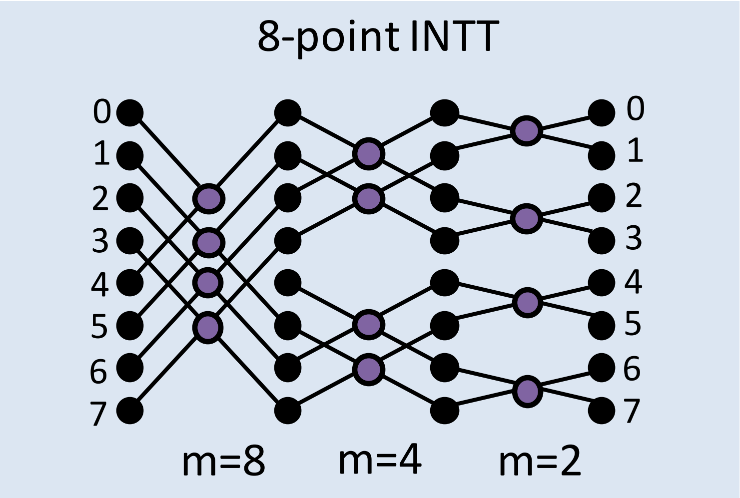
Simplified NTT loops

```
Loop m {
   Loop i {
      Loop j {
            Butterfly(B[j],B[j+m/2]);
      }
   }
}
```

Butterfly() reads two coefficients from memory.

Butterfly() writes two coefficients to memory.

NTT and Memory access

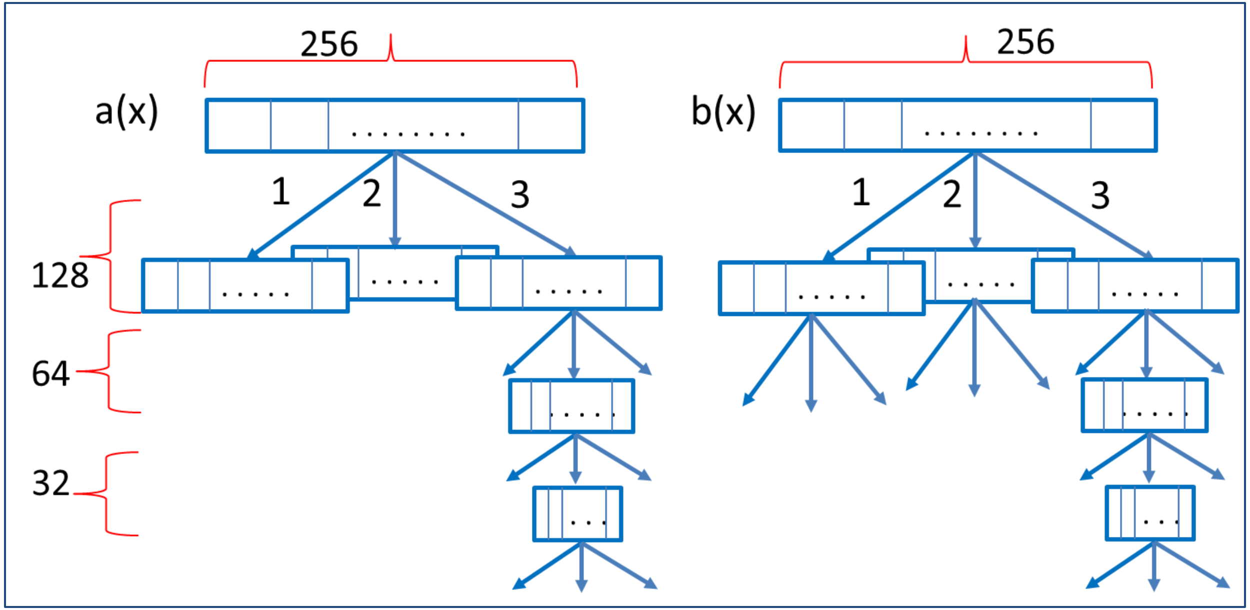


NTT and Memory access

```
--- MFNTT_DIT_NR (N=8)
A_index=0, B_index=4, psi_pow=4
A_index=1, B_index=5, psi_pow=4
A_index=2, B_index=6, psi_pow=4
A_index=3, B_index=7, psi_pow=4
---
A_index=0, B_index=2, psi_pow=2
A_index=1, B_index=3, psi_pow=2
A_index=4, B_index=6, psi_pow=6
A_index=5, B_index=7, psi_pow=6
---
A_index=0, B_index=1, psi_pow=1
A_index=2, B_index=3, psi_pow=5
A_index=4, B_index=5, psi_pow=3
A_index=6, B_index=7, psi_pow=7
```

```
--- MINTT_DIF_RN (N=8)
A_index=0, B_index=1, psi_pow=1
A_index=2, B_index=3, psi_pow=5
A_index=4, B_index=5, psi_pow=3
A_index=6, B_index=7, psi_pow=7
---
A_index=0, B_index=2, psi_pow=2
A_index=1, B_index=3, psi_pow=2
A_index=4, B_index=6, psi_pow=6
A_index=5, B_index=7, psi_pow=6
---
A_index=0, B_index=4, psi_pow=4
A_index=1, B_index=5, psi_pow=4
A_index=2, B_index=6, psi_pow=4
A_index=3, B_index=7, psi_pow=4
```

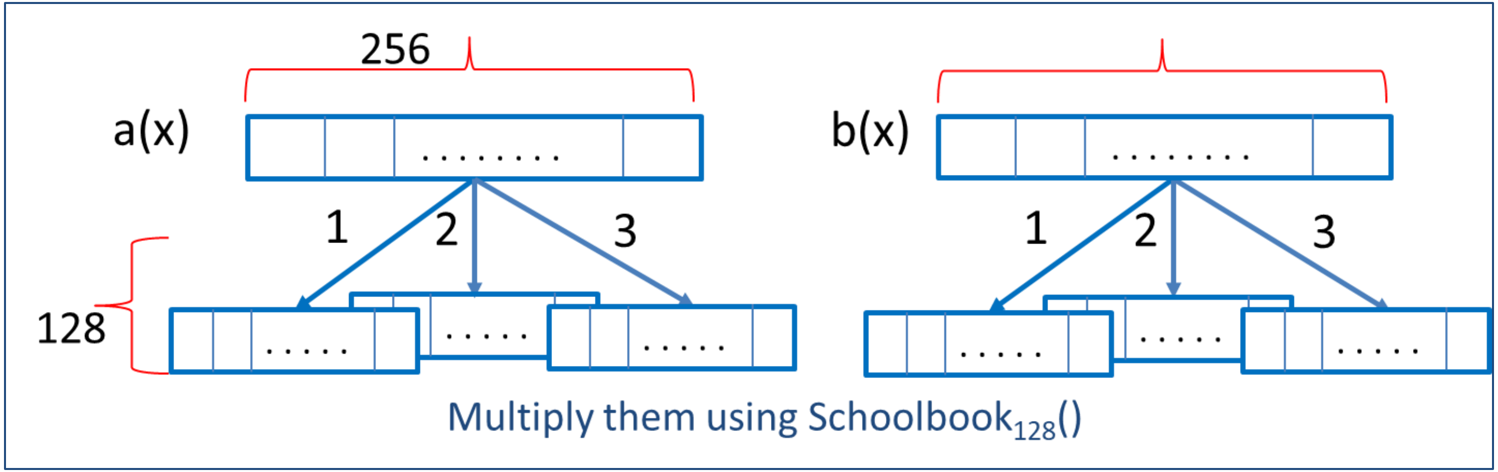
Karatsuba multiplier in HW?



- Karatsuba uses divide-and-conquer recursively.
- Recursion is easy to implement in SW → Call the function recursively.
- Full recursion is '*difficult*' to implement in HW (**my** *personal opinion*)

But, a few levels of recursions is easy to implement. (see next slide)

E.g., 1 level of Karatsuba then Schoolbook

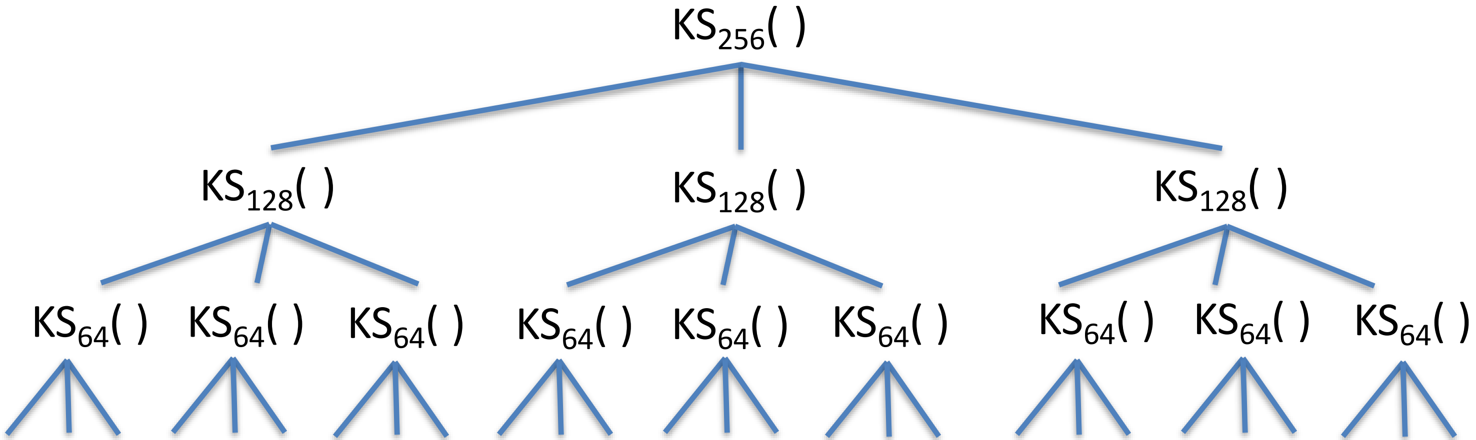


Some ideas:

1. Use HW/SW co-design approach. Perform splitting and joining in SW and compute the Schoolbook multiplications in HW.
   → Easy to implement. But many rounds of HW <--> SW communications.

2. Do everything in HW. → More efficient.

HW/SW co-design of the Karatsuba method



1. SW: Since recursion is challenging to implement in HW, perform all the recursive function calls in SW.

2. HW: When the recursion tree reaches a 'threshold', perform the actual schoolbook multiplications in HW.

3. SW: Read the partial results from HW and combine them in SW.

HW/SW co-design of the Karatsuba method: example