

The 23rd Annual International Conference on Information Security and Cryptology

ICISC 2020

December 2 (Wed) - December 4 (Fri), 2020 | Virtual Conference

Hosted by

Korea Institute of Information Security and Cryptology (KIISC)
National Security Research Institute (NSR)

Designing and Implementing the NIST Post-quantum Public-key Candidate Saber

Sujoy Sinha Roy

s.sinharoy@cs.bham.ac.uk

Centre of Excellence in Cybersecurity,
University of Birmingham, UK



Classical Diffie-Hellman Key Agreement

Public info: Prime p and base g



Secret a


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

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

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



Secret b

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

Why is this secure?

Discrete Logarithm Problem

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

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

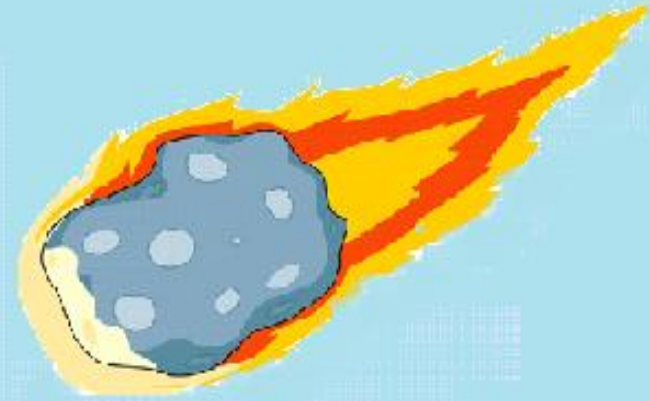
Latest record (Dec 2019) is 795-bit. [BGGHTZ'19]
Using Intel Xeon Gold 6130 CPUs.

Widely Used Public Key Algorithms

RSA cryptosystem
(Integer factorization problem)

Elliptic curve cryptosystem (ECC)
(Elliptic curve discrete logarithm problem)

Shor's Algo. + Quantum Computer



Uh oh...



NEWS

Home UK World Business Politics Tech Science Health Family & Education

Technology

NSA 'developing code-cracking quantum computer'

3 January 2014

f Share



Quantum

Network

Technology

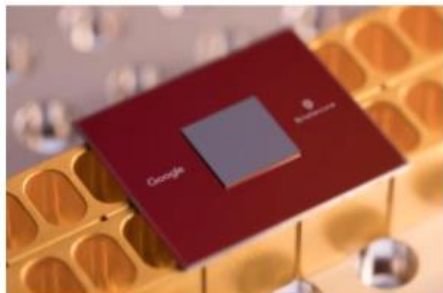
Resources

Quantum Starts Here



Death of public key cryptography???

most powerful supercomputers currently in existence, it will achieve what is known as "quantum supremacy". Google Quantum AI Lab revealed a new gate-based superconducting quantum computing chip called Bristlecone last week with a square array of 72 qubits (a portmanteau for quantum bits). They are going for quantum supremacy, but they may be a few qubits short.



Say hello to Bristlecone, Google's latest quantum processor (left). In the graphic of the device, each "X" shape represents one qubit, with "nearest neighbor connectivity". (Image courtesy of

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

Post Quantum Public Key Cryptography

Existing quantum algorithms cannot solve

- Lattice-based cryptography
 - Multivariate cryptography
 - Hash-based cryptography
 - Code-based cryptography
 - Supersingular elliptic curve isogeny cryptography
- ‘Learning With Errors’ (LWE) problem

Given two linear equations with unknown x and y

$$\begin{array}{l} 3x + 4y = 26 \\ 2x + 3y = 19 \end{array} \quad \text{or} \quad \begin{pmatrix} 3 & 4 \\ 2 & 3 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 26 \\ 19 \end{pmatrix}$$

Find x and y .

Solving system of linear equations

System of linear equations with unknown **s**

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

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

Solving system of linear equations with errors

$$\begin{array}{c} \text{Matrix } \mathbf{A} \\ \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix} \end{array} \cdot \begin{array}{c} \begin{pmatrix} s_1 \\ s_2 \\ \vdots \\ s_n \end{pmatrix} \end{array} + \begin{array}{c} \begin{pmatrix} e_1 \\ e_2 \\ \vdots \\ e_n \\ \vdots \\ e_m \end{pmatrix} \end{array} = \begin{array}{c} \text{Vector } \mathbf{b} \\ \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \\ \vdots \\ b_m \end{pmatrix} \end{array} \pmod{q}$$

- Search **Learning With Errors** (LWE) problem:
Given $(\mathbf{A}, \mathbf{b}) \rightarrow$ computationally infeasible to solve (\mathbf{s}, \mathbf{e})
- Decisional **Learning With Errors** (LWE) problem :
Given $(\mathbf{A}, \mathbf{b}) \rightarrow$ hard to distinguish from random

Diffie-Hellman styled Key Exchange based on LWE

Public matrix **A**

Secret vector **s**
Error vector **e**



$$\mathbf{b} = \mathbf{A} \times \mathbf{s} + \mathbf{e}$$

Secret vector **s'**
Error vector **e'**



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

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

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

Noisy shared secret

FRODO: An example of LWE scheme

FRODO uses a 640-matrix dimension (100-bit security)

FRODO on ARM Cortex M4 @ 24 MHz

Key gen	Encapsulation	Decapsulation
81 M	86 M	87 M

- Around 3.3 sec per operation
- **Slow due to expensive matrix-vector multiplications**

Can we improve the speed?

Standard LWE

$$\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix} * \begin{pmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{pmatrix} + \begin{pmatrix} e_0 \\ e_1 \\ e_2 \\ e_3 \end{pmatrix} \approx \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} \pmod{q}$$

Uniformly random matrix

Ring LWE

$$\begin{pmatrix} a_0 & -a_3 & -a_2 & -a_1 \\ a_1 & a_0 & -a_3 & -a_2 \\ a_2 & a_1 & a_0 & -a_3 \\ a_3 & a_2 & a_1 & a_0 \end{pmatrix} * \begin{pmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{pmatrix} + \begin{pmatrix} e_0 \\ e_1 \\ e_2 \\ e_3 \end{pmatrix} \approx \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} \pmod{q}$$

Matrix from uniformly
random vector

Ring-LWE

$$\begin{pmatrix} a_0 & -a_3 & -a_2 & -a_1 \\ a_1 & a_0 & -a_3 & -a_2 \\ a_2 & a_1 & a_0 & -a_3 \\ a_3 & a_2 & a_1 & a_0 \end{pmatrix} * \begin{pmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{pmatrix} + \begin{pmatrix} e_0 \\ e_1 \\ e_2 \\ e_3 \end{pmatrix} \approx \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} \pmod{q}$$

Matrix from uniformly
random vector



$$a(x) * s(x) + e(x) \approx b(x) \pmod{q} \pmod{x^4 + 1}$$

where

$$a(x) = (a_0 + a_1x + a_2x^2 + a_3x^3)$$

$$s(x) = (s_0 + s_1x + s_2x^2 + s_3x^3)$$

$$e(x) = (e_0 + e_1x + e_2x^2 + e_3x^3)$$

$$b(x) = (b_0 + b_1x + b_2x^2 + b_3x^3)$$

From Standard LWE Key Exchange to Ring-LWE Key Exchange

(Standard) LWE Diffie-Hellman key-exchange

Public matrix **A**

Secret vector **s**
Error vector **e**



$$\mathbf{b} = \mathbf{A} \mathbf{s} + \mathbf{e}$$

Secret vector **s'**
Error vector **e'**



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

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

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

Noisy shared secret

(Efficient) Ring-LWE Diffie-Hellman key-exchange

Public polynomial $a(x)$

Secret poly $s(x)$

Error poly $e(x)$



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

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

Secret poly $s'(x)$

Error poly $e'(x)$



$$\begin{aligned} v(x) &= b'(x) \cdot s(x) \\ &= a(x) \cdot s(x) \cdot s'(x) + e'(x) \cdot s(x) \end{aligned}$$

$$\begin{aligned} v'(x) &= b(x) \cdot s'(x) \\ &= a(x) \cdot s(x) \cdot s'(x) + e(x) \cdot s'(x) \end{aligned}$$

Noisy shared secret poly

Interpolating LWE and ring-LWE: Module LWE

$$\begin{pmatrix}
 \begin{bmatrix} a_0 & -a_3 & -a_2 & -a_1 \\ a_1 & a_0 & -a_3 & -a_2 \\ a_2 & a_1 & a_0 & -a_3 \\ a_3 & a_2 & a_1 & a_0 \end{bmatrix} & \begin{bmatrix} a_8 & -a_{11} & -a_{10} & -a_9 \\ a_9 & a_8 & -a_{11} & -a_{10} \\ a_{10} & a_9 & a_8 & -a_{11} \\ a_{11} & a_{10} & a_7 & a_8 \end{bmatrix} \\
 \begin{bmatrix} a_4 & -a_7 & -a_6 & -a_5 \\ a_5 & a_4 & -a_7 & -a_6 \\ a_6 & a_5 & a_4 & -a_7 \\ a_7 & a_6 & a_5 & a_4 \end{bmatrix} & \begin{bmatrix} a_{12} & -a_{15} & -a_{14} & -a_{13} \\ a_{13} & a_{12} & -a_{15} & -a_{14} \\ a_{14} & a_{13} & a_{12} & -a_{15} \\ a_{15} & a_{14} & a_{13} & a_{12} \end{bmatrix}
 \end{pmatrix}
 * \begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \\ s_6 \\ s_7 \end{bmatrix}
 + \begin{bmatrix} e_0 \\ e_1 \\ e_2 \\ e_3 \\ e_4 \\ e_5 \\ e_6 \\ e_7 \end{bmatrix}
 \approx \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix}$$



$$\begin{pmatrix} a_{0,0}(x) & a_{0,1}(x) \\ a_{1,0}(x) & a_{1,1}(x) \end{pmatrix} * \begin{bmatrix} s_0(x) \\ s_1(x) \end{bmatrix} + \begin{bmatrix} e_0(x) \\ e_1(x) \end{bmatrix} \approx \begin{bmatrix} b_0(x) \\ b_1(x) \end{bmatrix} \pmod{q} \pmod{x^4 + 1}$$

Saber: Module lattice based key exchange, CPA-secure encryption and CCA-secure KEM

Saber is a round 3 finalist for the NIST PQC standardization process.

NIST reported that

“SABER is one of the most promising KEM schemes to be considered for standardization at the end of the third round.”

Saber is based on

Module Learning with Rounding (MLWR)

+ Flexibility

+ no generation of errors e , e' etc.

+ efficient bandwidth usage

Learning with error (LWE) vs Learning with rounding (LWR)

LWE:

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix} \cdot \begin{pmatrix} s_1 \\ s_2 \\ \vdots \\ s_n \end{pmatrix} + \begin{pmatrix} e_1 \\ e_2 \\ \vdots \\ e_n \\ \vdots \\ e_m \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \\ \vdots \\ b_m \end{pmatrix} \pmod{q}$$

LWR:

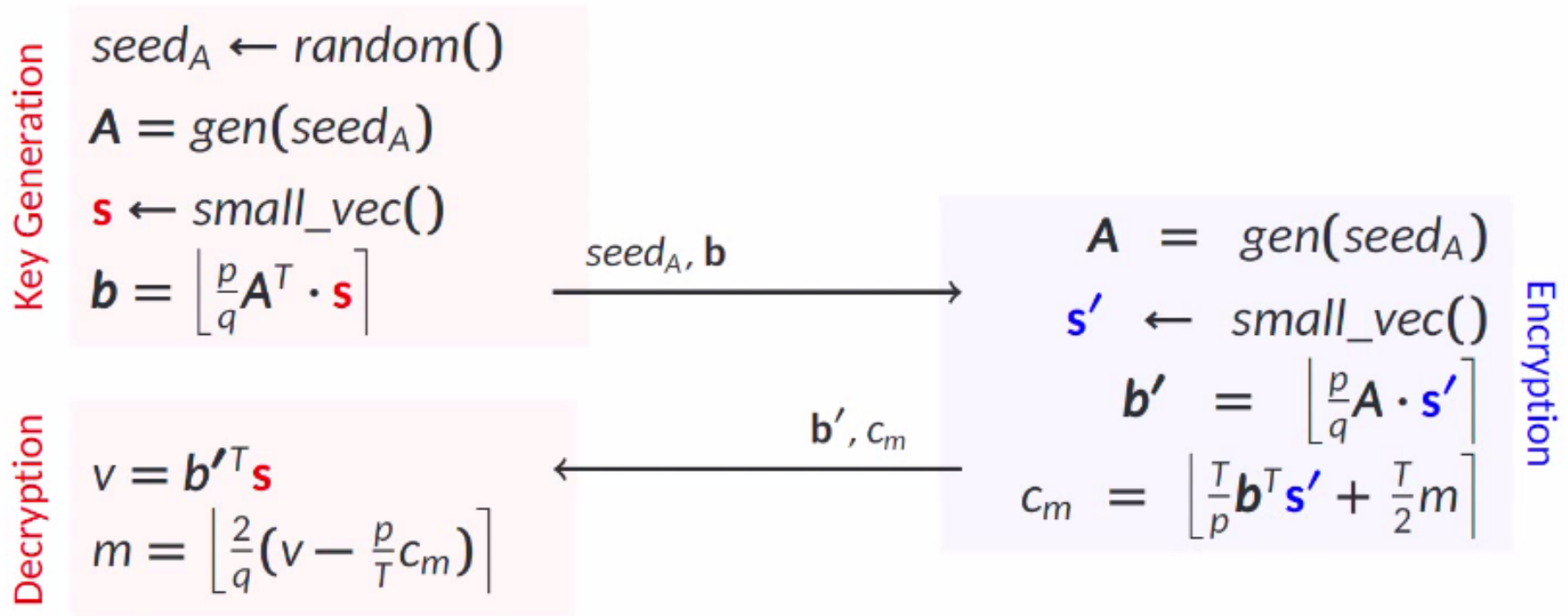
$$\left\lfloor \frac{p}{q} \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix} \cdot \begin{pmatrix} s_1 \\ s_2 \\ \vdots \\ s_n \end{pmatrix} \right\rfloor = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \\ \vdots \\ b_m \end{pmatrix} \pmod{p}$$

Advantages of LWR

where $p < q$

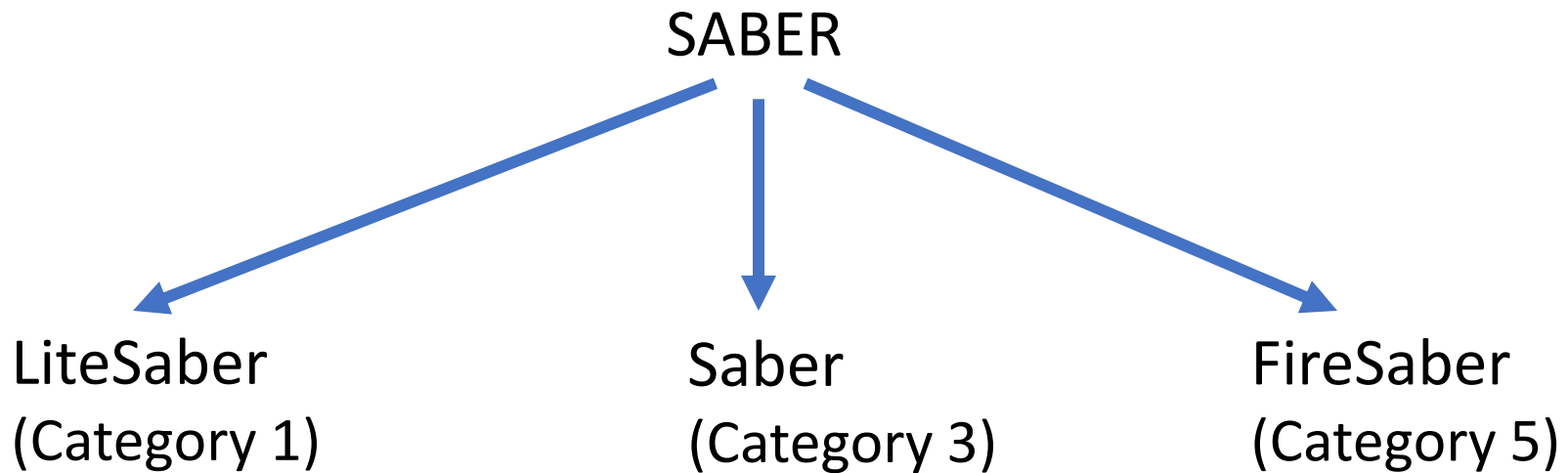
- + no generation of errors e
- + efficient bandwidth usage

The Saber protocol



Saber.KEM is obtained via the Fujisaki-Okamoto transform.

The three security levels of Saber



All polynomials have degree 255

Module dim $k=2$

Module dim $k=3$

Module dim $k=4$

How to choose p , q and secret s ?

$$\left[\frac{p}{q} \begin{pmatrix} a_{0,0}(x) & \dots & a_{0,k-1}(x) \\ & \dots & \\ a_{k-1,0}(x) & \dots & a_{k-1,k-1}(x) \end{pmatrix} * \begin{pmatrix} s_0(x) \\ \dots \\ s_{k-1}(x) \end{pmatrix} \right] \approx \begin{pmatrix} b_0(x) \\ \dots \\ b_{k-1}(x) \end{pmatrix} \pmod{x^{256} + 1}$$

How to choose p , q and secret s ?

For a given module dimension (k) and polynomial degree (n), the parameters p , q , and secret s influence:

- Security
- Decryption failure
- Performance
- Physical security



Needs investigating
implementation aspects

**(next part of
this talk)**

The Saber protocol: building blocks

Key Generation

$seed_A \leftarrow \text{random}()$

$A = \text{gen}(seed_A)$

$\mathbf{s} \leftarrow \text{small_vec}()$

$\mathbf{b} = \left\lfloor \frac{p}{q} \mathbf{A}^T \cdot \mathbf{s} \right\rfloor$

$seed_A, \mathbf{b}$

Decryption

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

$m = \left\lfloor \frac{2}{q} \left(v - \frac{p}{T} c_m \right) \right\rfloor$

\mathbf{b}', c_m

$A = \text{gen}(seed_A)$

$\mathbf{s}' \leftarrow \text{small_vec}()$

$\mathbf{b}' = \left\lfloor \frac{p}{q} \mathbf{A} \cdot \mathbf{s}' \right\rfloor$

$c_m = \left\lfloor \frac{T}{p} \mathbf{b}^T \mathbf{s}' + \frac{T}{2} m \right\rfloor$

Encryption

Building blocks:

- Polynomial addition, subtraction, multiplication
- Rounding
- Sampling of secret
- Hashing and Pseudo-random string generation

How to multiply two polynomials?

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

Memory access during NTT

Simplified NTT loops

A[n-1]
A[n-2]
A[3]
A[2]
A[1]
A[0]

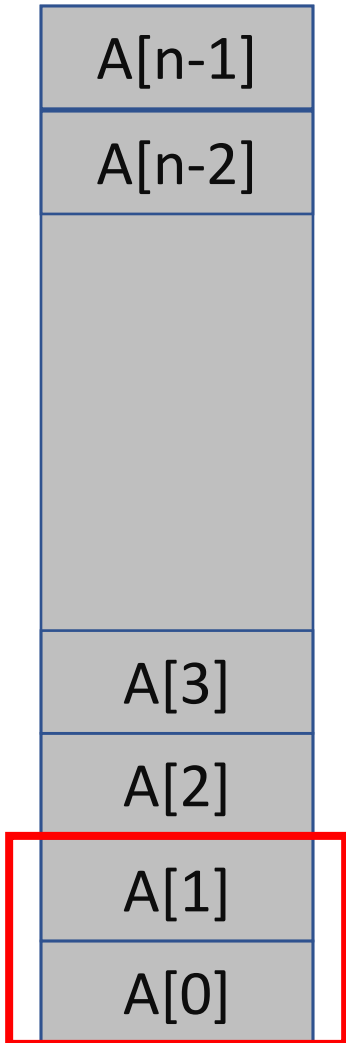
```
for (m=2; m<=n; m=2m)
{
    for (j=0; j<=m/2-1; j++)
    {
        for (k=0; j<n; k=k+m)
        {
            index = f(m, j, k);
            Butterfly(A[index], A[index+m/2]);
        }
    }
}
```


Memory access during NTT

Simplified NTT loops

```
for (m=2; m<=n; m=2*m)
{
    for (j=0; j<=m/2-1; j++)
    {
        for (k=0; j<n; k=k+m)
        {
            index = f(m, j, k);
            Butterfly(A[index], A[index+m/2]);
        }
    }
}
```

NTT starts with $m=2$
Butterfly($A[0]$, $A[1]$)

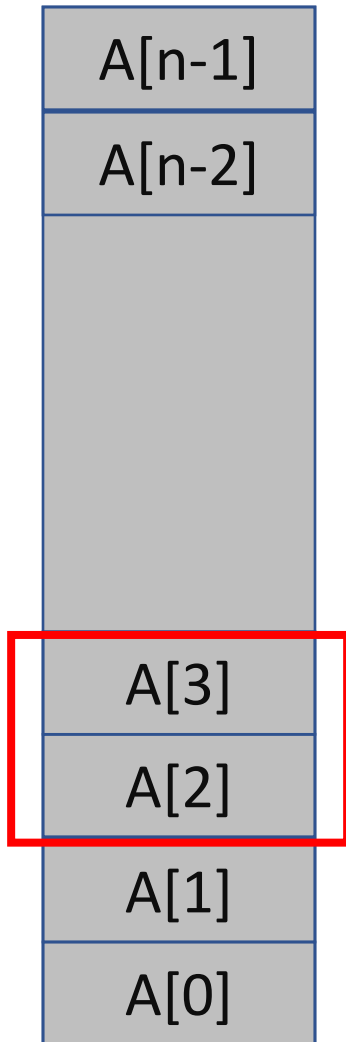


Memory access during NTT

Simplified NTT loops

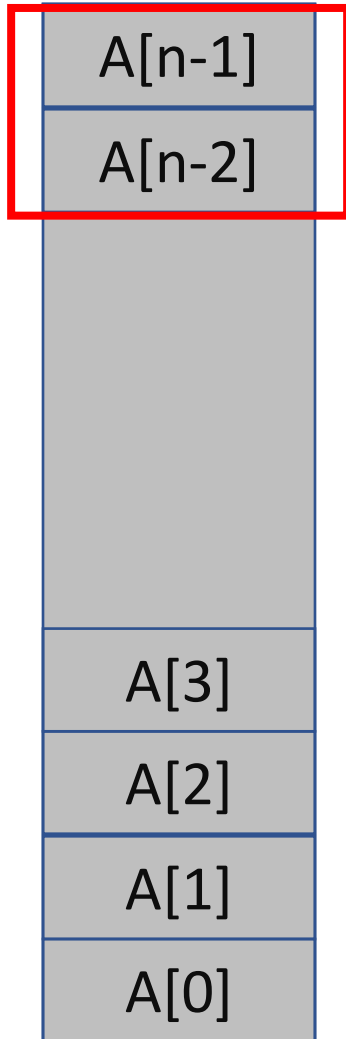
```
for (m=2; m<=n; m=2*m)
{
    for (j=0; j<=m/2-1; j++)
    {
        for (k=0; j<n; k=k+m)
        {
            index = f(m, j, k);
            Butterfly(A[index], A[index+m/2]);
        }
    }
}
```

NTT starts with $m=2$
Butterfly($A[2]$, $A[3]$)



Memory access during NTT

Simplified NTT loops



```
for (m=2; m<=n; m=2*m)
{
    for (j=0; j<=m/2-1; j++)
    {
        for (k=0; j<n; k=k+m)
        {
            index = f(m, j, k);
            Butterfly(A[index], A[index+m/2]);
        }
    }
}
```

NTT starts with **m**=2

Butterfly(A[n-2], A[n-1])

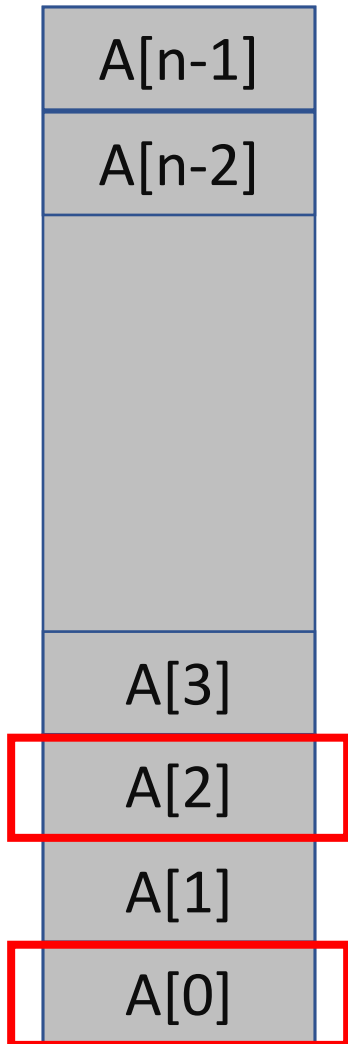
Memory access during NTT

Simplified NTT loops

```
for (m=2; m<=n; m=2*m)
{
    for (j=0; j<=m/2-1; j++)
    {
        for (k=0; j<n; k=k+m)
        {
            index = f(m, j, k);
            Butterfly(A[index], A[index+m/2]);
        }
    }
}
```

Next, m increments to **m**=4.

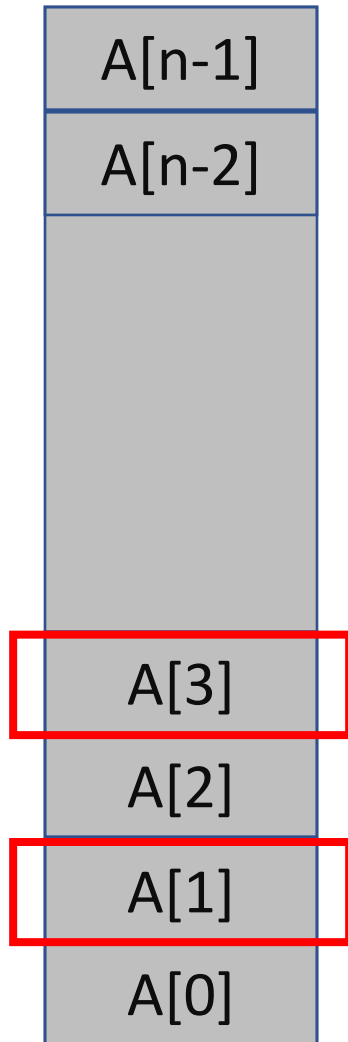
Butterfly(A[0], A[2]), Butterfly(A[4], A[6]) ...



Memory access during NTT

Simplified NTT loops

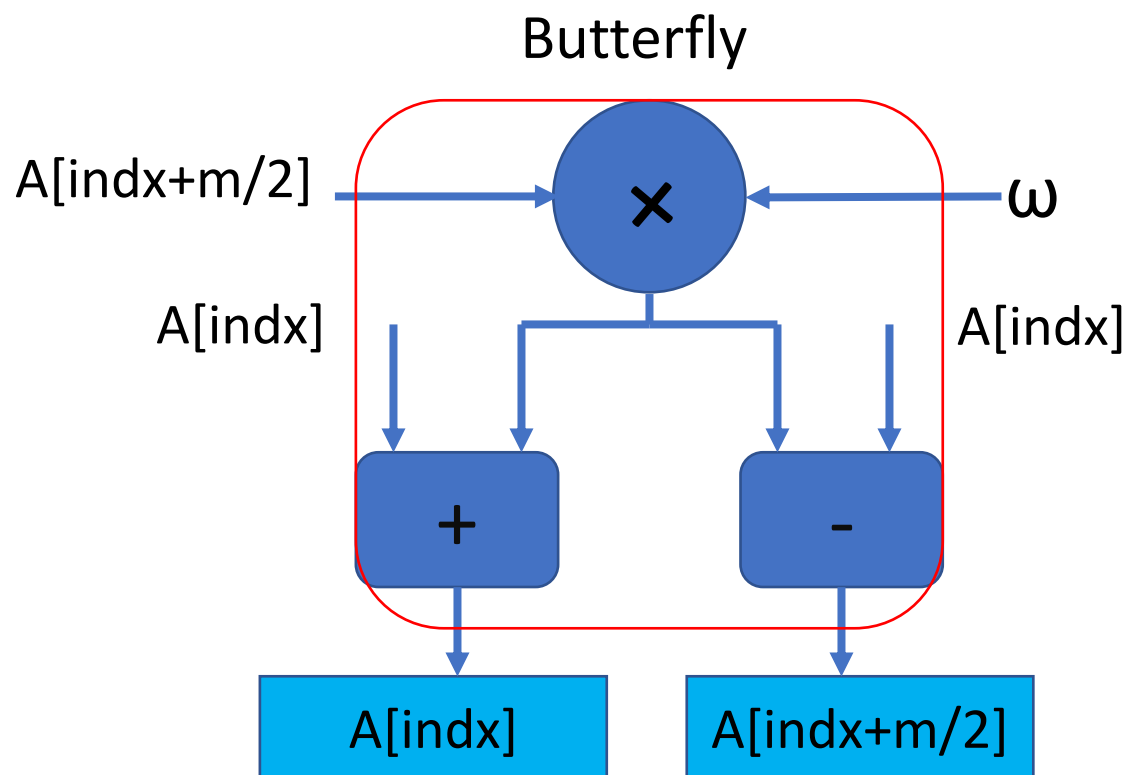
```
for (m=2; m<=n; m=2*m)
{
    for (j=0; j<=m/2-1; j++)
    {
        for (k=0; j<n; k=k+m)
        {
            index = f(m, j, k);
            Butterfly(A[index], A[index+m/2]);
        }
    }
}
```



Next, m increments to **m**=4.

Butterfly(A[1], A[3]), Butterfly(A[5], A[7]) ...

$A[n-1]$
$A[n-2]$
$A[3]$
$A[2]$
$A[1]$
$A[0]$



NTT-based polynomial multiplication: summary

- Asymptotically fastest algorithm for polynomial multiplication
- Implementation effort is needed for making it fast
 - Variable memory access pattern increases access overhead
 - Parallelization requires extra design effort

NewHope, Kyber, Dilithium make NTT integral part.

Polynomial multiplication choices for Saber

Saber uses Learning with rounding (LWR)

$$\left[\frac{p}{q} \text{ Uniform in } [0, q-1] \right] \quad \text{where } p < q$$

and performs polynomial arithmetic modulo p and q .

Choice 1: prime p and q .

- + Fast NTT-based multiplication
- Expensive rounding
- Rounding bias

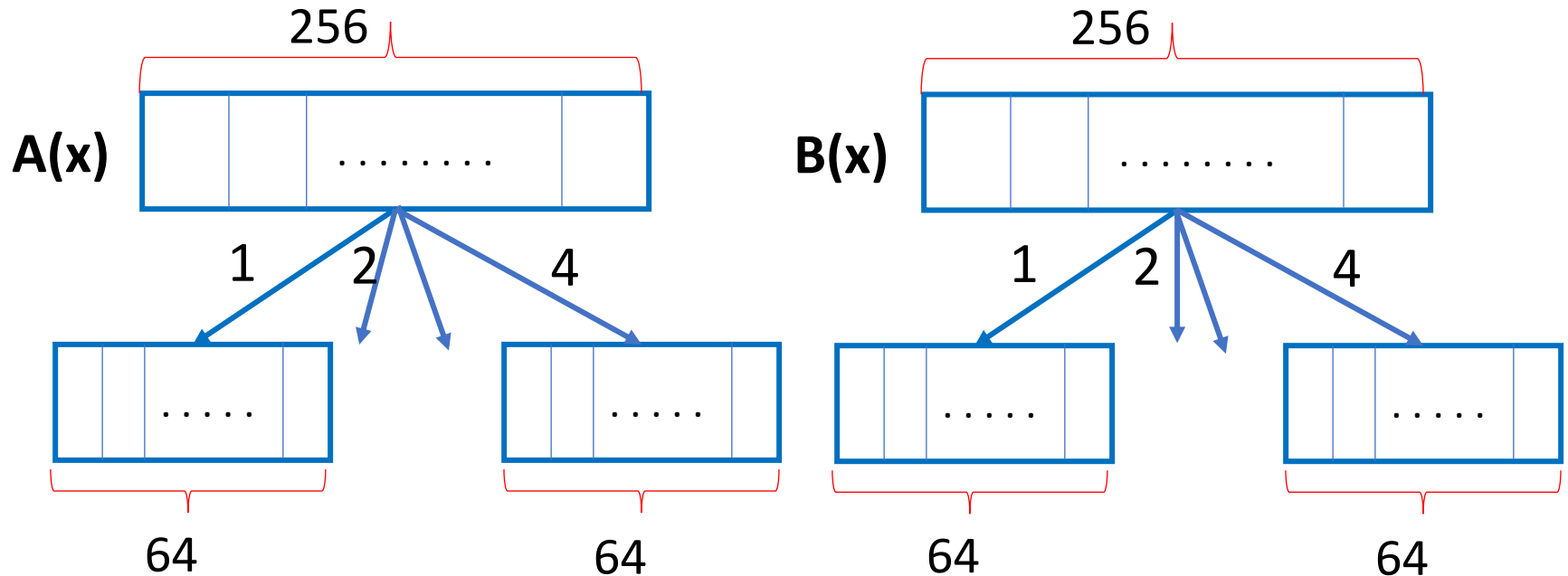
Choice 2: pow-2 p and q .

- No NTT-based multiplication
- + Free rounding
- + No Rounding bias
- + Generic polynomial mult.
- + Easier masking against SCA
- + more ...

Saber went for Choice 2

Toom-Cook polynomial multiplication algorithms

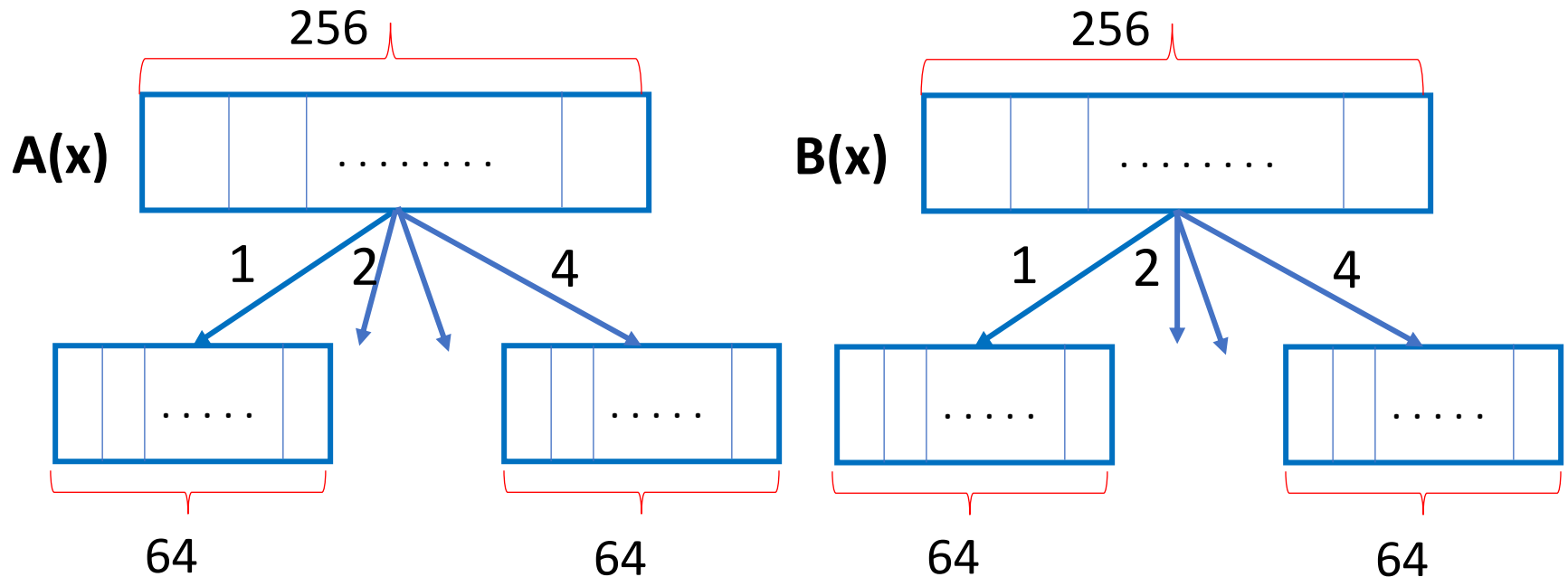
- Toom-Cook multiplication



Toom-Cook 4 Way needs 7 multiplications

Karatsuba would need 9 multiplications

Toom-Cook 4 Way: step-by-step: splitting



Splitting operand into 4 polynomials

Take $y = x^{64}$

$$A(y) = A_3 y^3 + A_2 y^2 + A_1 y + A_0$$

$$B(y) = B_3 y^3 + B_2 y^2 + B_1 y + B_0$$

Toom-Cook 4 Way: step-by-step: evaluation

$$\begin{aligned}w_1 &= A(\infty) * B(\infty) = A_3 * B_3 \\w_2 &= A(2) * B(2) = (A_0 + 2 \cdot A_1 + 4 \cdot A_2 + 8 \cdot A_3) * (B_0 + 2 \cdot B_1 + 4 \cdot B_2 + 8 \cdot B_3) \\w_3 &= A(1) * B(1) = (A_0 + A_1 + A_2 + A_3) * (B_0 + B_1 + B_2 + B_3) \\w_4 &= A(-1) * B(-1) = (A_0 - A_1 + A_2 - A_3) * (B_0 - B_1 + B_2 - B_3) \\w_5 &= A\left(\frac{1}{2}\right) * B\left(\frac{1}{2}\right) = (8 \cdot A_0 + 4 \cdot A_1 + 2 \cdot A_2 + A_3) * (8 \cdot B_0 + 4 \cdot B_1 + 2 \cdot B_2 + B_3) \\w_6 &= A\left(\frac{-1}{2}\right) * B\left(\frac{-1}{2}\right) = (8 \cdot A_0 - 4 \cdot A_1 + 2 \cdot A_2 - A_3) * (8 \cdot B_0 - 4 \cdot B_1 + 2 \cdot B_2 - B_3) \\w_7 &= A(0) * B(0) = A_0 * B_0\end{aligned}$$

Linear operations

+

Seven multiplications are computed

Toom-Cook 4 Way: step-by-step: interpolation

```
// Interpolation
```

```
 $w_2 = w_2 + w_5$ 
```

```
 $w_6 = w_6 - w_5$ 
```

```
 $w_4 = (w_4 - w_3)/2$ 
```

```
 $w_5 = w_5 - w_1 - 64 \cdot w_7$ 
```

```
 $w_3 = w_3 + w_4$ 
```

```
 $w_5 = 2 \cdot w_5 + w_6$ 
```

```
 $w_2 = w_2 - 65 \cdot w_3$ 
```

```
 $w_3 = w_3 - w_7 - w_1$ 
```

```
 $w_2 = w_2 + 45 \cdot w_3$ 
```

```
 $w_5 = (w_5 - 8 \cdot w_3)/24$ 
```

```
 $w_6 = w_6 + w_2$ 
```

```
 $w_2 = (w_2 + 16 \cdot w_4)/18$ 
```

```
 $w_3 = w_3 - w_5$ 
```

```
 $w_4 = -(w_4 + w_2)$ 
```

```
 $w_6 = (30 \cdot w_2 - w_6)/60$ 
```

```
 $w_2 = w_2 - w_6$ 
```

```
return  $w_1 \cdot y^6 + w_2 \cdot y^5 + w_3 \cdot y^4 + w_4 \cdot y^3 + w_5 \cdot y^2 + w_6 \cdot y + w_7;$ 
```

Linear operations

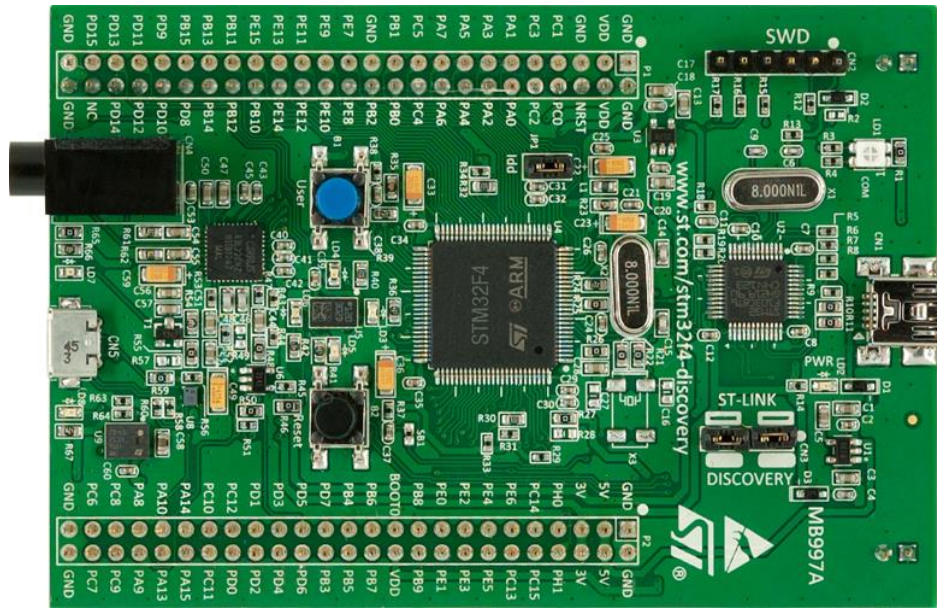
→ This number has a role to play

Linear operations

```
__m256i _mm256_abs_epi16 (__m256i a)
__m256i _mm256_add_epi16 (__m256i a, __m256i b)
__m256i _mm256_adds_epi16 (__m256i a, __m256i b)
__m256i _mm256_blend_epi16 (__m256i a, __m256i b, const int imm8)
__m128i _mm_broadcastw_epi16 (__m128i a)
__m256i _mm256_broadcastw_epi16 (__m128i a)
__m256i _mm256_cmpeq_epi16 (__m256i a, __m256i b)
__m256i _mm256_cmpgt_epi16 (__m256i a, __m256i b)
__m256i _mm256_cvtepi16_epi32 (__m128i a)
__m256i _mm256_cvtepi16_epi64 (__m128i a)
__m256i _mm256_cvtepi8_epi16 (__m128i a)
__m256i _mm256_cvtepu8_epi16 (__m128i a)
int _mm256_extract_epi16 (__m256i a, const int index)
__m256i _mm256_hadd_epi16 (__m256i a, __m256i b)
__m256i _mm256_hadds_epi16 (__m256i a, __m256i b)
__m256i _mm256_hsub_epi16 (__m256i a, __m256i b)
__m256i _mm256_hsubs_epi16 (__m256i a, __m256i b)
```

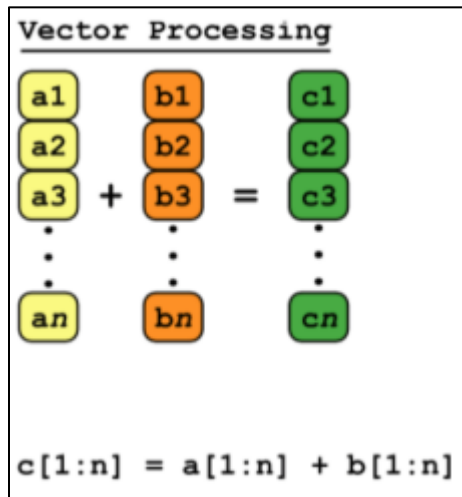
Vectorized instructions for 16-bit operands

DSP instructions



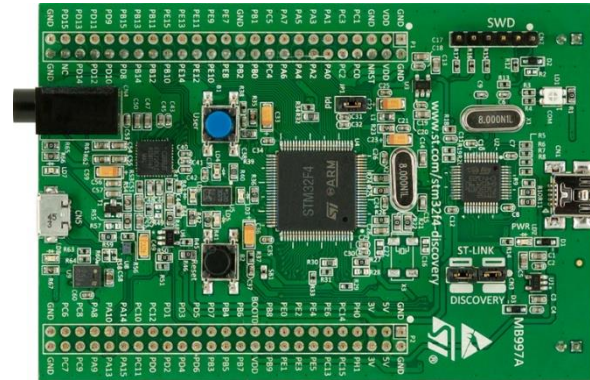
ARM Cortex-M4

- Popular 32-bit microcontroller
- Has DSP instructions for half-word operations



AVX

+



Microcontroller with DSP

Keep coefficients smaller/equal to 16 bits to use

- `_epi16()` AVX intrinsics in high-end platforms
- DSP instructions in low-end microcontrollers

Options for q : 2^{16} , 2^{15} , 2^{14} , 2^{13} ..., etc.



Toom-Cook 4 Way: step-by-step: interpolation

```
// Interpolation
w2 = w2 + w5
w6 = w6 - w5
w4 = (w4 - w3)/2
w5 = w5 - w1 - 64 · w7
w3 = w3 + w4
w5 = 2 · w5 + w6
w2 = w2 - 65 · w3
w3 = w3 - w7 - w1
w2 = w2 + 45 · w3
w5 = (w5 - 8 · w3)/24 → This number has a role
                        to play
w6 = w6 + w2
w2 = (w2 + 16 · w4)/18
w3 = w3 - w5
w4 = -(w4 + w2)
w6 = (30 · w2 - w6)/60
w2 = w2 - w6
return w1 · y6 + w2 · y5 + w3 · y4 + w4 · y3 + w5 · y2 + w6 · y + w7;
```

Linear operations

Division by 24 in Toom-Cook Interpolation

$$w_5 = (w_5 - 8 \cdot w_3)/24$$

- $24 = 8 \cdot 3$
- We are working in R_q where $q = 2^i$
- 3 has inverse in mod q
E.g. $3^{-1} \bmod 2^{15} \rightarrow 10923$
- So, division by 3 is same as multiplying by $3^{-1} \bmod q$

Division by 24 in Toom-Cook Interpolation

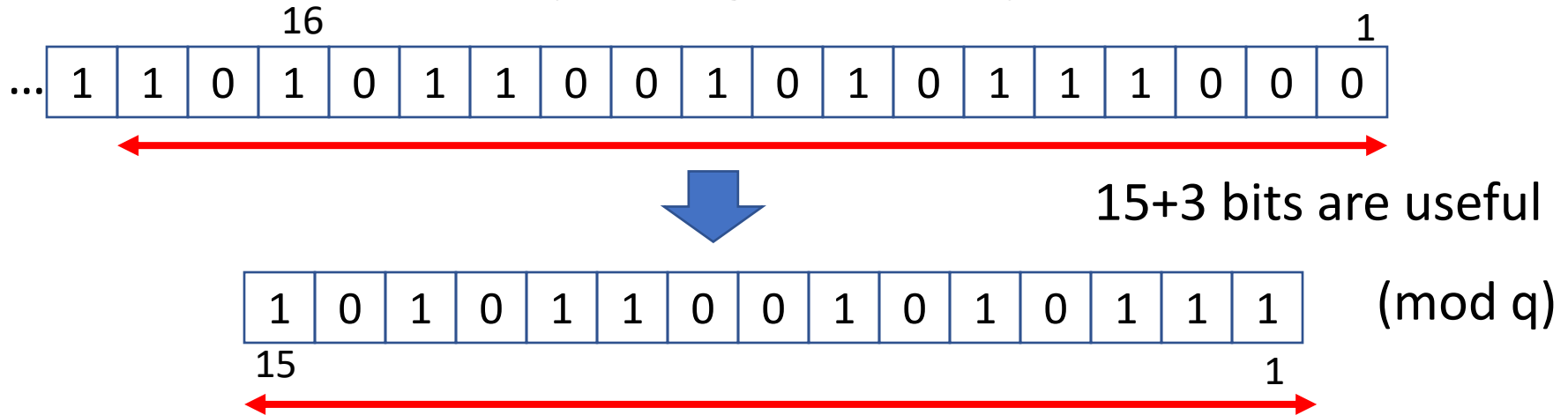
$$w_5 = (w_5 - 8 \cdot w_3)/24$$

- $24 = 8 \cdot 3$
- We are working in R_q where $q = 2^i$
- But, 8 does not have inverse in mod q

Only option: do actual division

Working with $q = 2^{15}$

Example: integer division by $8=2^3$

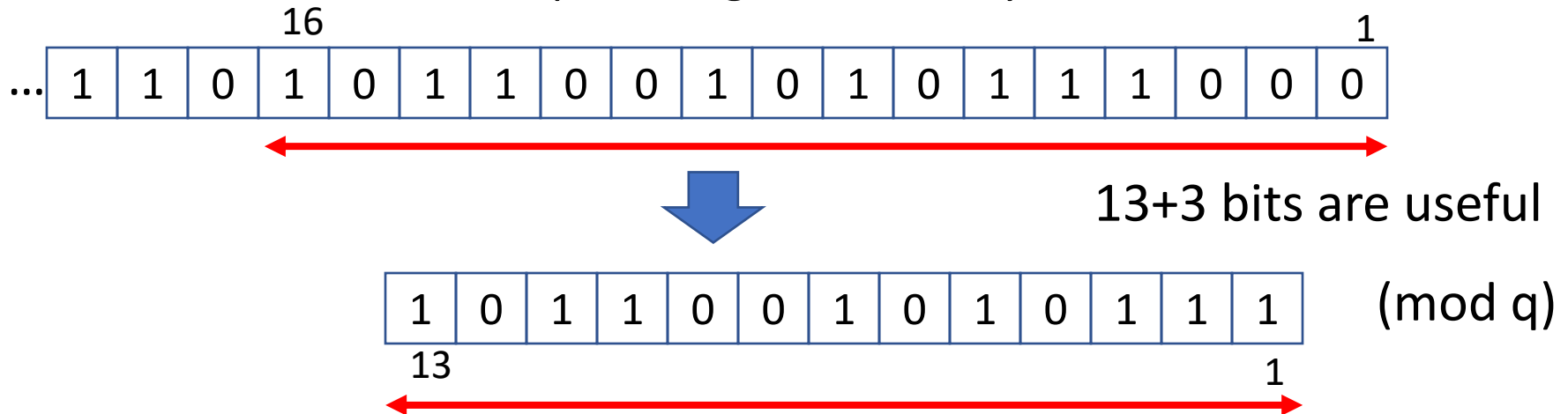


In 16-bit Computer:

- Requires careful arithmetic of two words
- Slower arithmetic

Working with $q = 2^{13}$

Example: integer division by $8=2^3$



Fits in 16-bit words ☺

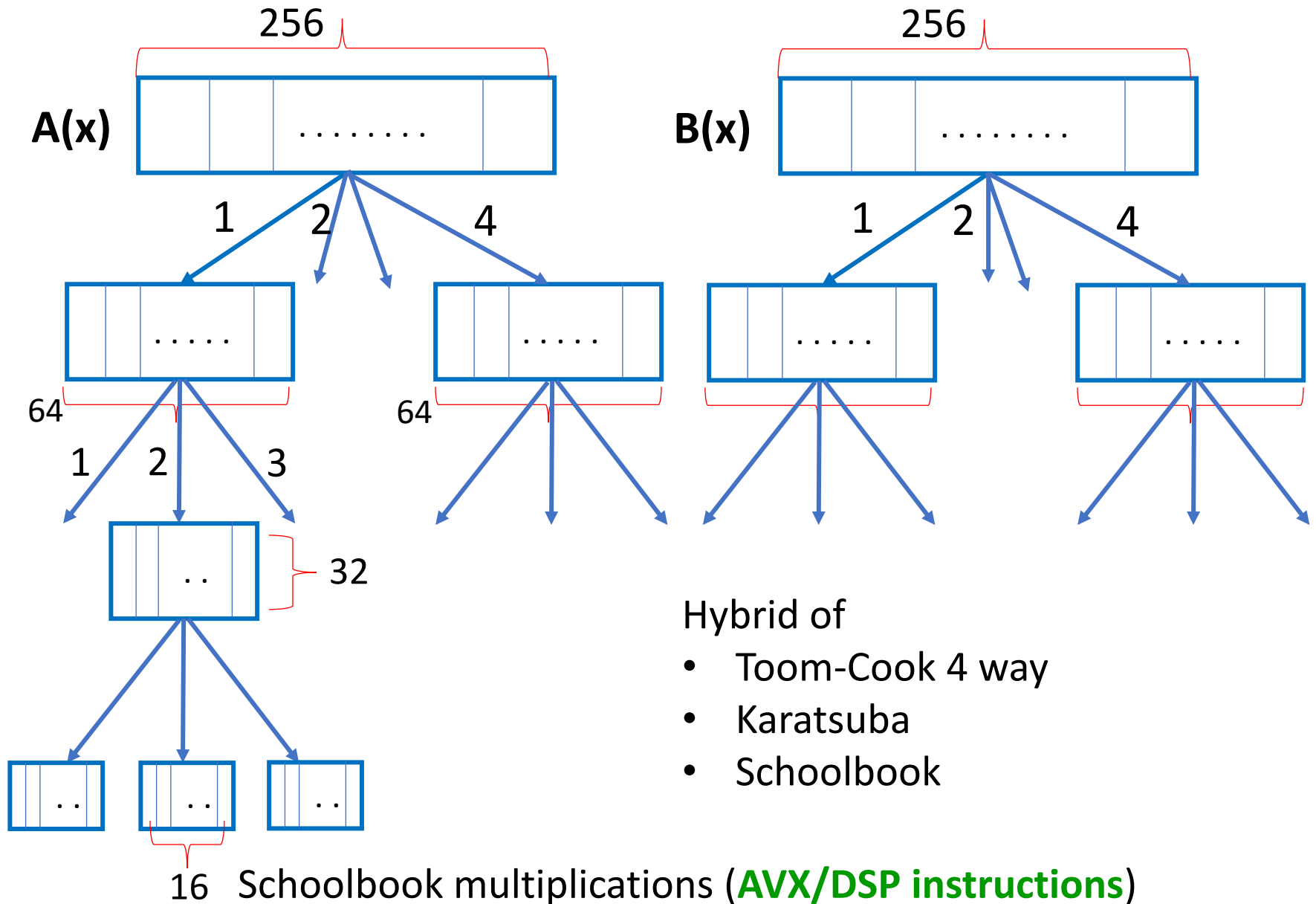
In 16-bit Computer:

- **Easy** to implement
- **Less** complicated arithmetic

Saber Parameters

- Polynomial length $n = 256$
- $q = 2^{13}$
- $p = 2^{10}$

Polynomial multiplication in Saber



Polynomial multiplication using DSP instructions

Cortex-M4: STM32F4-discovery by STMicroelectronics

- 16-bit DSP instructions
- Cross-half-word multiplication possible



A. Karmakar, J.M. Bermudo Mera, S. Sinha Roy and I. Verbauwhede.
"Saber on ARM", CHES 2018

For 16x16 Schoolbook multiplication → 37.5% reduction overall

... more SW optimizations

- Saber in RSA coprocessor

B. Wang, X. Gu and Y. Yang. “*Saber on ESP32*”, ACNS 2020.

uses 2048-bit integer multiplier to accelerate polynomial multiplication in Saber.

→ Benefits from pow-2 moduli p and q .

- Improved Toom-Cook multiplication in SW

J.M. Bermudo Mera, A. Karmakar, and I. Verbauwhede.
“*Time-memory trade-off in Toom-Cook multiplication*”, CHES 2020

proposes SW optimization techniques.

Results for PQC finalists (NIST category III security)

Size in bytes

Scheme	Secret Key	Public key	Ciphertext
Saber	1,344	992	1,088
Kyber768	2,400	1,184	1,088
NTRUhrss701	1,450	1,138	1,138
McEliece460896	13,568	52,4160	188

Speed in SW Intel Xeon E3-1220, hiphop, supercop-20200906

Scheme	Keygen	Encaps	Decaps
Saber	80,340	103,204	103,092
Kyber768	53,588	74,092	64,000
NTRUhrss701	269,864	26,596	64,164
McEliece460896	179,358,620	76,472	267,728

SaberX4 is a batched implementation of Saber for higher operations/sec.

Very recent implementation uses NTT-based multiplication and reports ~20% speedup.

Saber in Hardware [CHES 2020]

Performance bottlenecks

Two 'big' building blocks

- SHA/SHAKE
 - Keccak is slow in SW
 - But fast in HW (26 cycles per permutation)
 - Saber protocol uses serialized Keccak calls
 - We use one Keccak core
 - Simplifies HW implementation
- Polynomial multiplication

Performance bottlenecks

Two 'big' building blocks

- SHA/SHAKE
 - Keccak is slow in SW
 - But fast in HW (26 cycles per permutation)
 - Saber protocol uses serialized Keccak calls
 - We use one Keccak core
 - Simplifies HW implementation
- Polynomial multiplication
 - Saber protocol allows *any* polynomial mul. algo.
 - So, choose the best for the target HW

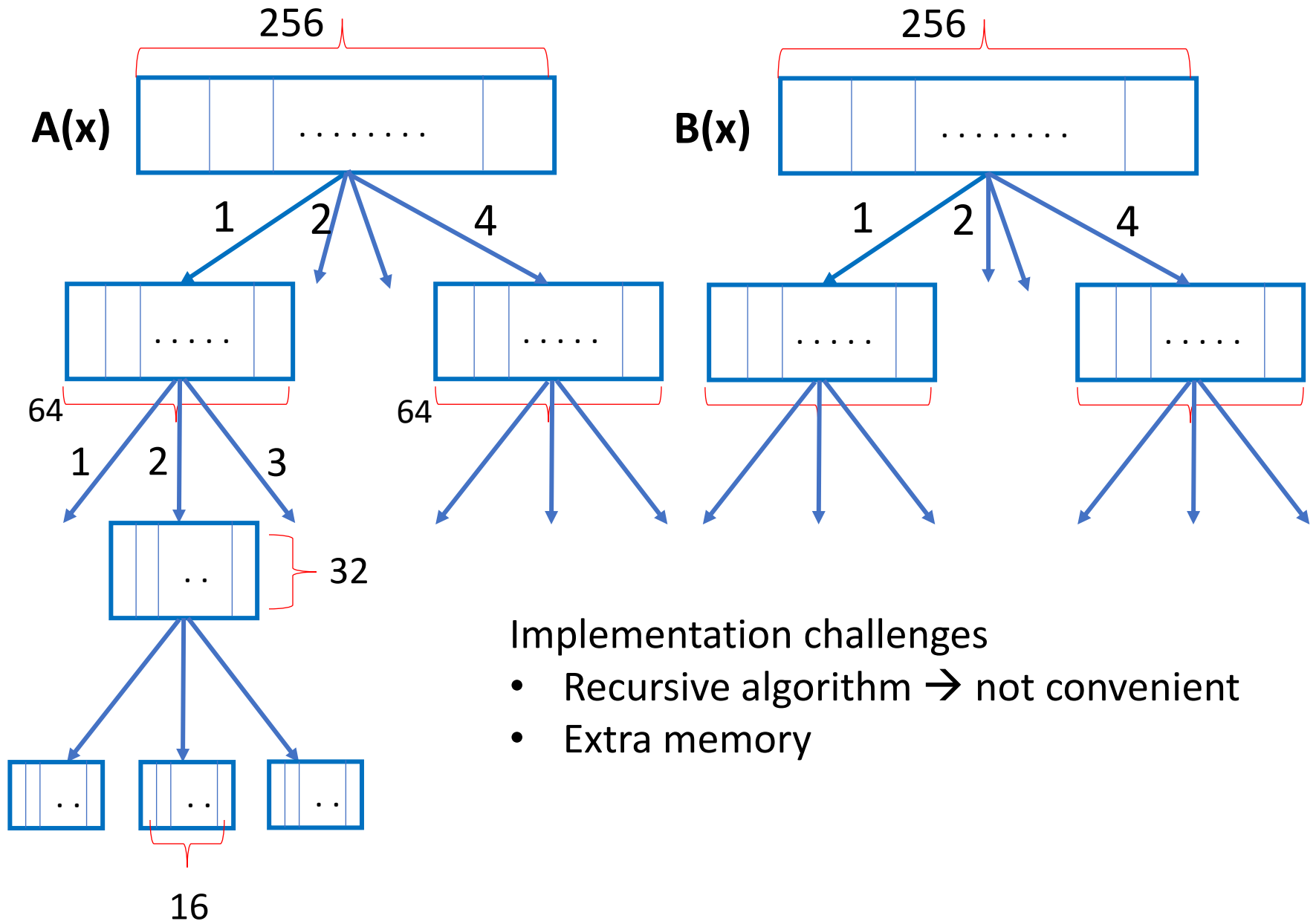
Polynomial multiplication(s) in Saber

Interesting features:

1. Polynomials are of (small) degree 255
2. Moduli $p = 2^{10}$ and $q = 2^{13}$
 - No modular reduction circuit
3. For any $A(x) * B(x)$
 - $A(x)$ is always a secret polynomial where coefficients are small $[-3, 3]$, $[-4, 4]$ or $[-5, 5]$.
 - $B(x)$ is either modulo p or q

Can we design a hardware that benefits from above features?

Toom-Cook in HW?



Implementation challenges

- Recursive algorithm \rightarrow not convenient
- Extra memory

Schoolbook in HW?

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

Disadvantages

- $O(n^2)$ complexity
- But $n = 256$ (small)

Advantages

- Simple structure
- Easier to implement
- Optimal memory
- High flexibility

Our hardware architecture uses schoolbook. [CHES 2020]

Schoolbook polynomial multiplier

Algorithm: Schoolbook algorithm

$acc(x) \leftarrow 0$

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

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

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

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

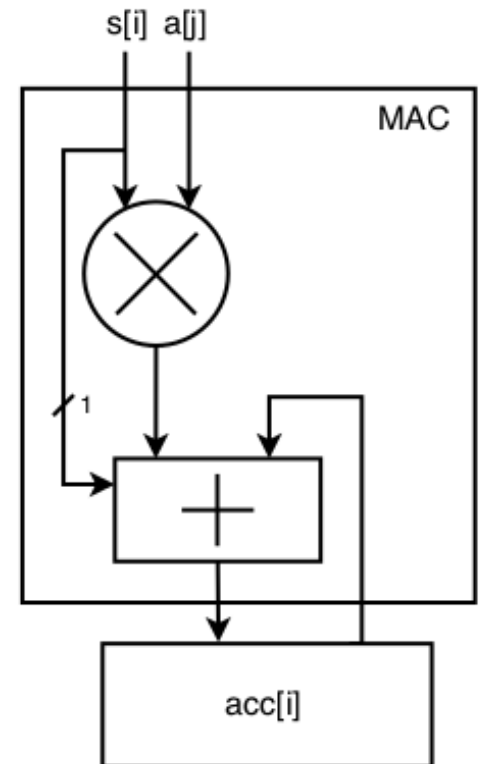
return acc

Multiply and Accumulate
(MAC)

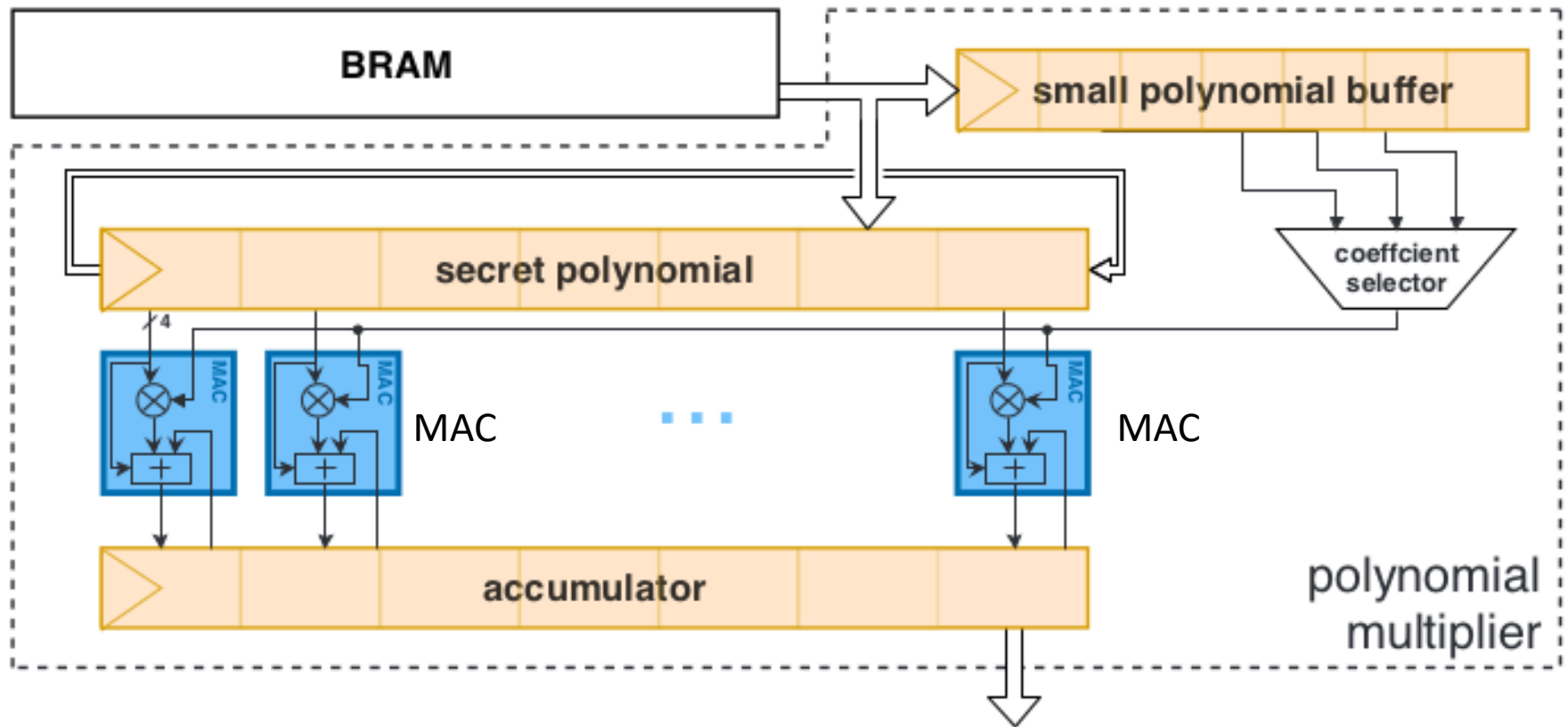
- $s[i]$ are small $[-3, 3]$, $[-4, 4]$ or $[-5, 5]$
- $a[i]$ are modulo $p=2^{10}$ or $q=2^{13}$
- No modular reduction



MAC unit requires little area (50 LUTs)

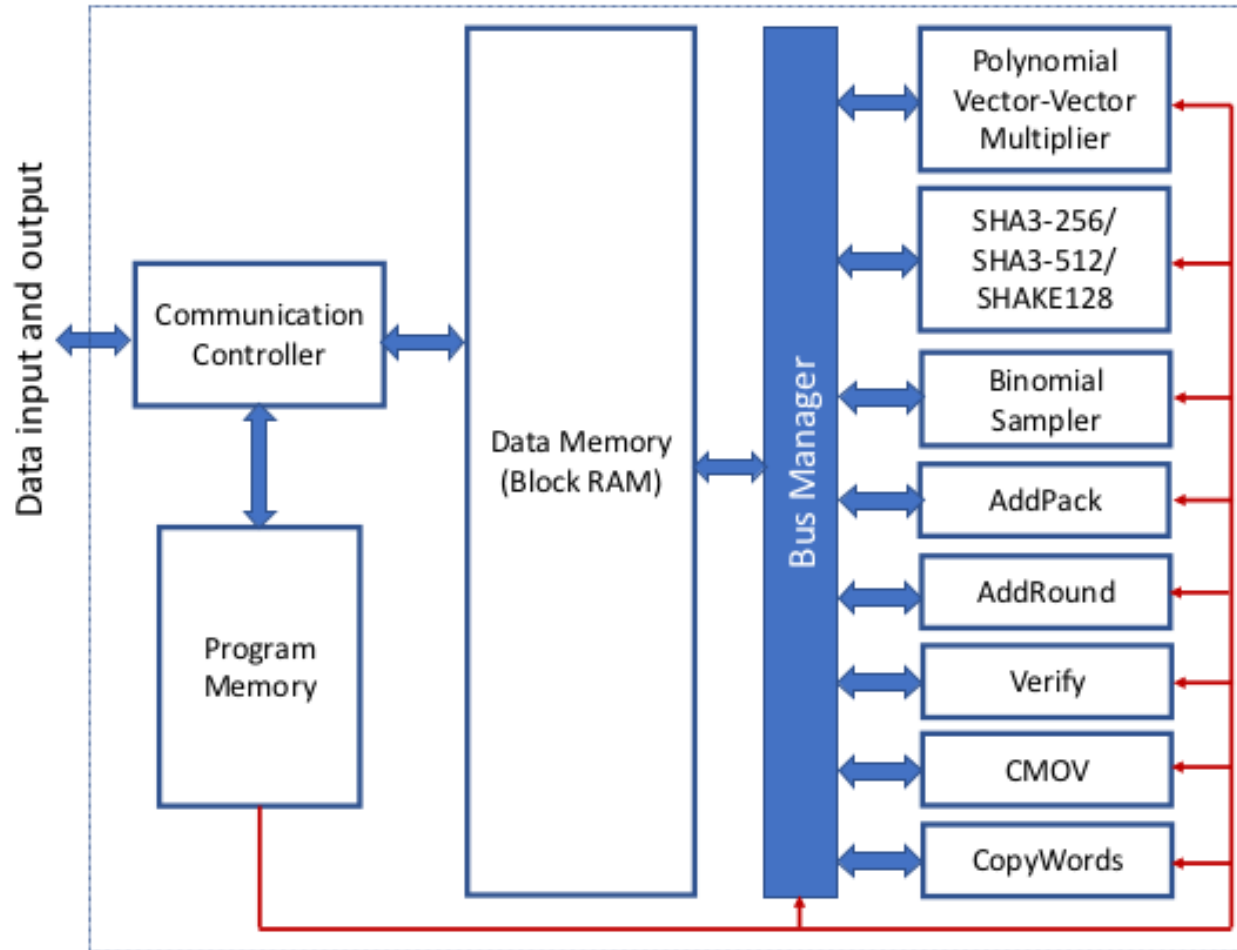


High-speed schoolbook polynomial multiplier



- 256 parallel MACs are used
- Secret and result polynomials are stored in registers
- one polynomial multiplication requires only 256 cycles
- Small control logic

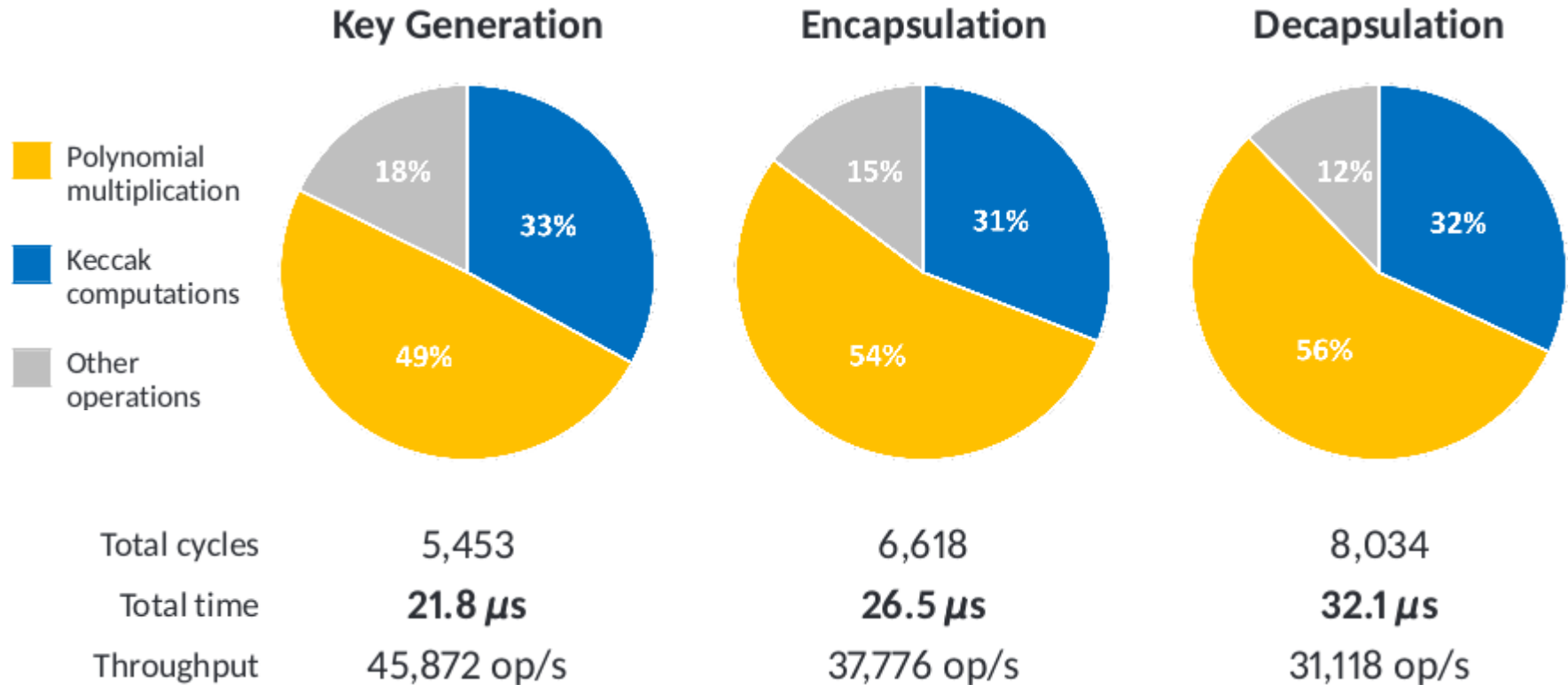
Instruction Set Coprocessor for Saber



- Full HW for CCA-secure Saber KEM
- Flexibility → unified architecture for three Saber variants
- Generic framework → can be followed by other schemes

ISA Saber: Performance results

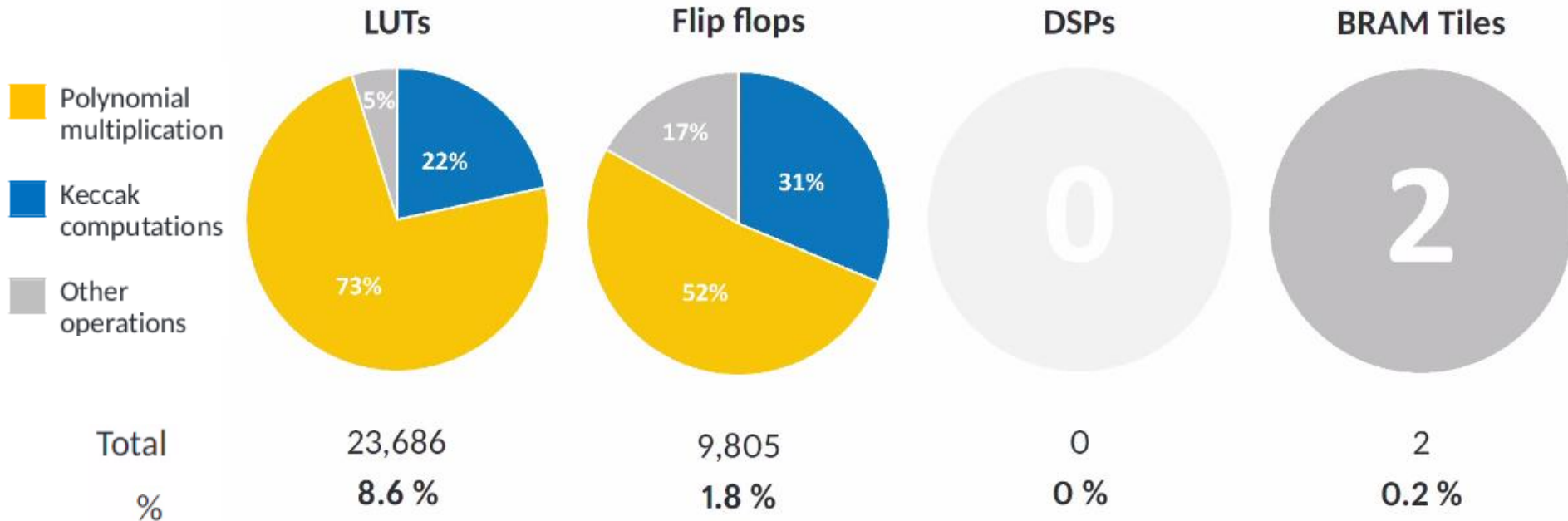
Target platform Ultrascale+ XCZU9EG-2FFVB1156 FPGA



Slide courtesy: Andrea Basso [CHES 2020 talk]

ISA Saber: Area results

Target platform Ultrascale+ XCZU9EG-2FFVB1156 FPGA



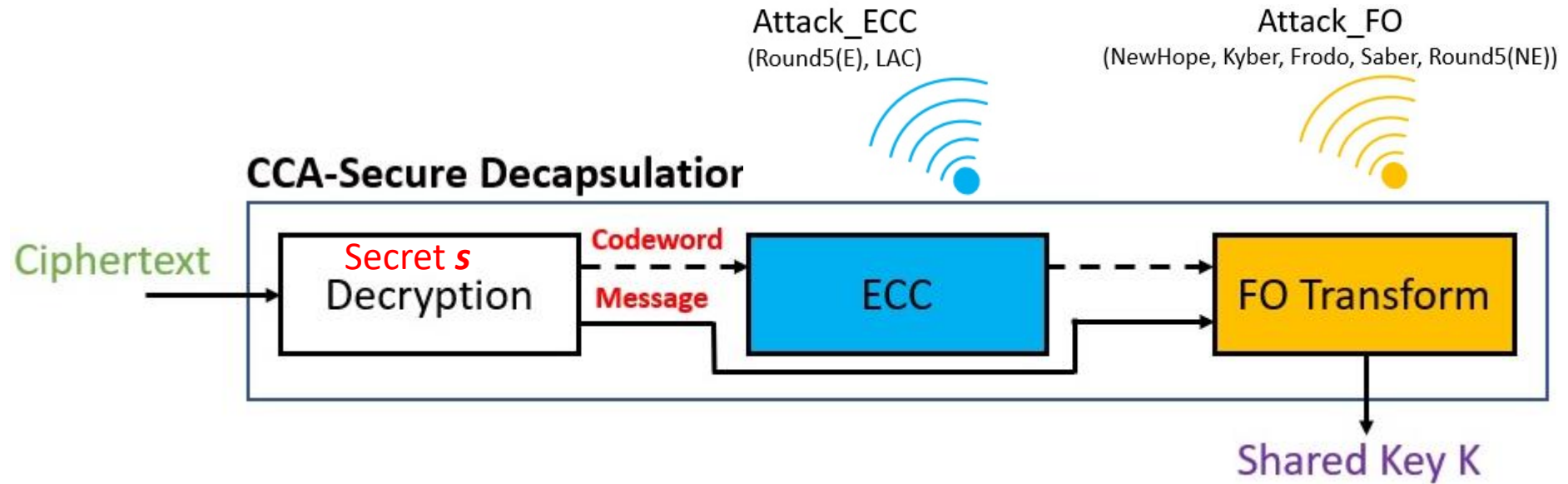
In one FPGA: 11 coprocessors can be fit → 504 K / 416 K / 342 K KEMs per sec

Slide courtesy: Andrea Basso [CHES 2020 talk]

... the game continues ...

Making PQC Side Channel Resistant

Side Channel Analysis of Lattice-based Crypto



Assumption: the secret s is static.

Attacker's goal: know the secret s

P. Ravi, S. Sinha Roy, A. Chattopadhyay, S. Bhasin. "Generic Side-channel attacks on CCA-secure encapsulation schemes" in CHES 2020.

Z. Xu, O. Pemberton, S. Sinha Roy and D. Oswald.

"Magnifying Side-Channel Leakage of Lattice-Based Cryptosystems with Chosen Ciphertexts: The Case Study of Kyber." IACR ePrint 2020/912.

Masking Saber

Two *unique* advantages for Saber

1. Use of power-of-2 moduli makes '*Arithmetic to Boolean*' conversion a lot more efficient.
2. Use of Learning with rounding (LWR) eliminates need for error sampling
 - No need for masked error sampler
 - Reduction in randomness requirement

Cycle counts on ARM Cortex-M4

Scheme	Unmasked	Masked
Saber (MLWR with pow-2 modulus)	1,123,280	2,833,348 (2.52x)
Ring LWE with prime modulus	4,416,918	25,334,493 (5.74x)

M. Van Beirendonck, JP. D'Anvers, A. Karmakar, J. Balasch, and I. Verbauwhede.
"*A Side-Channel Resistant Implementation of SABER*" in IACR ePrint 2020/733.

Conclusions

- Saber targets high security, flexibility, efficiency, and simplicity
- Use of LWR results
 - Less randomness requirement
 - Lower communication bandwidth
- Use of power-of-2 moduli results in
 - Simpler and efficient implementation
 - Easier masking against SCA
- Use of generic polynomial multiplication
 - Gives freedom to implementors
 - Platform-dependent implementation strategy
 - AVX, M4, RSA card, FPGA, ASIC, ...

Future works

- More efficient implementations
- Lightweight hardware architectures
- Side channel and fault attack resistant HW and SW
- Study Saber's compatibility with lattice-based signature schemes Dilithium and Falcon.