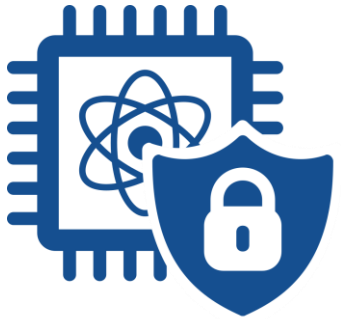
Implementing Post-Quantum Cryptography and Security of Quantum Computers



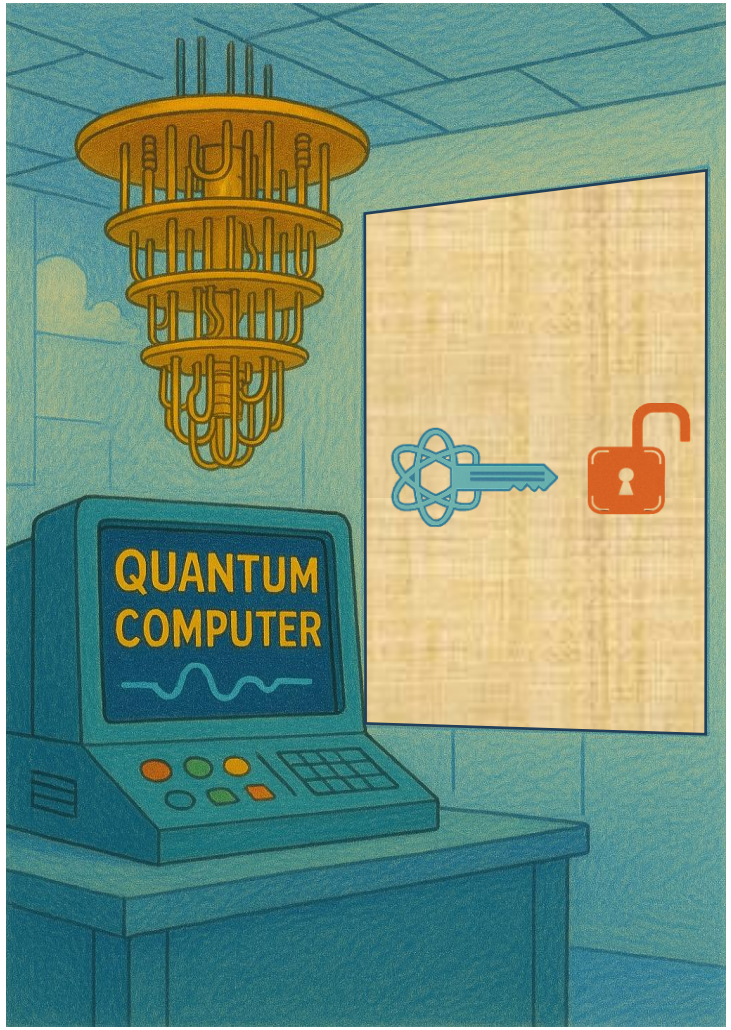
Sanjay Deshpande
Department of Electrical & Computer Engineering
Northwestern University

VIT®
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

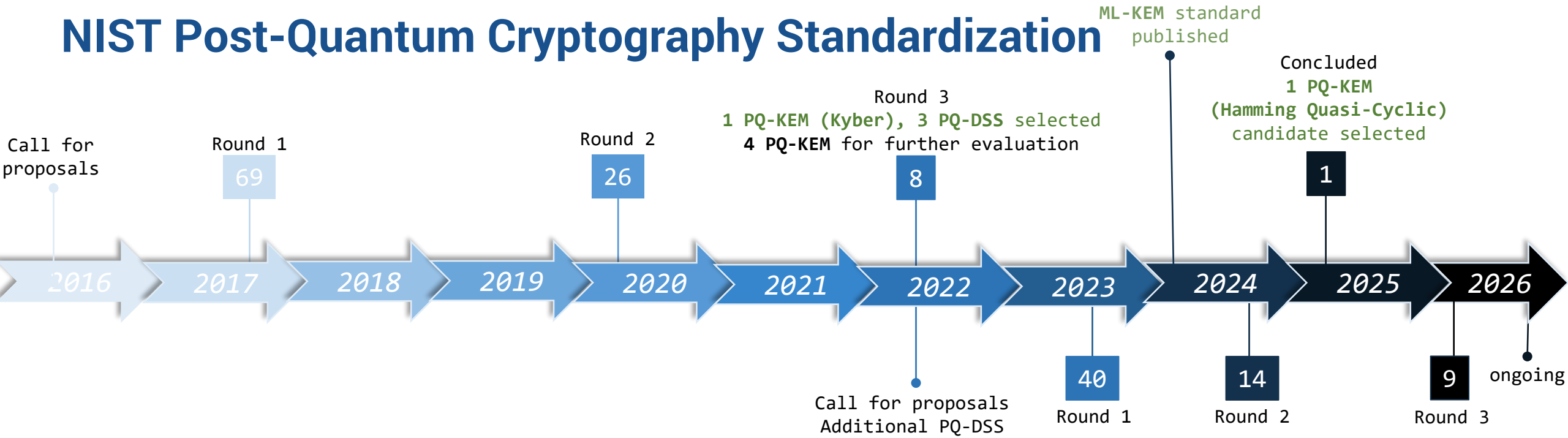
Modern Cryptography

- Symmetric-key cryptography
- Public-key cryptography
  - Shor's algorithm can potentially break existing pre-quantum public-key cryptosystems (e.g., RSA)
- **Quantum-safe Cryptography** to the rescue!
  - Quantum Cryptography
  - Post-quantum cryptography
    - Algorithms run on classical computers
- Post-Quantum Cryptography standardization efforts are in progress around the world

NIST ISO IETF



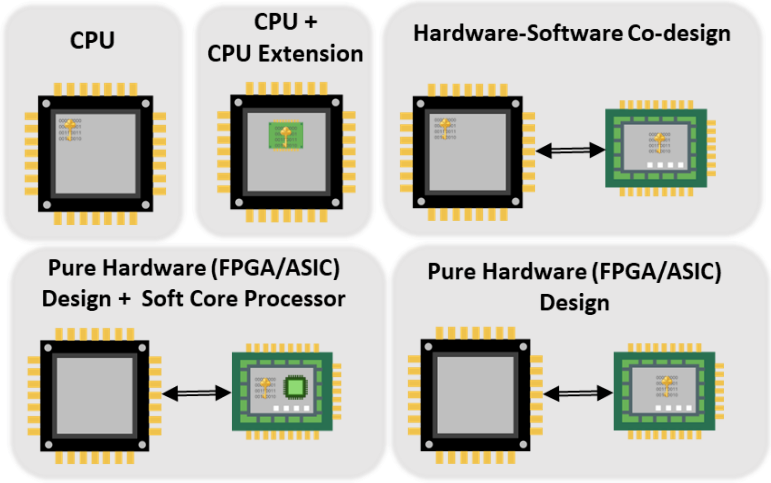
NIST Post-Quantum Cryptography Standardization



- Submitted candidates belong to two categories
  - **PQ-KEM: Post-Quantum Key Encapsulation Mechanism**
  - PQ-DSS: Post-Quantum Digital Signature Scheme
- Evaluation Criteria
  - Security
  - **Hardware and Software implementation efficiency and security**
  - Cost (key size, signature size, etc.)

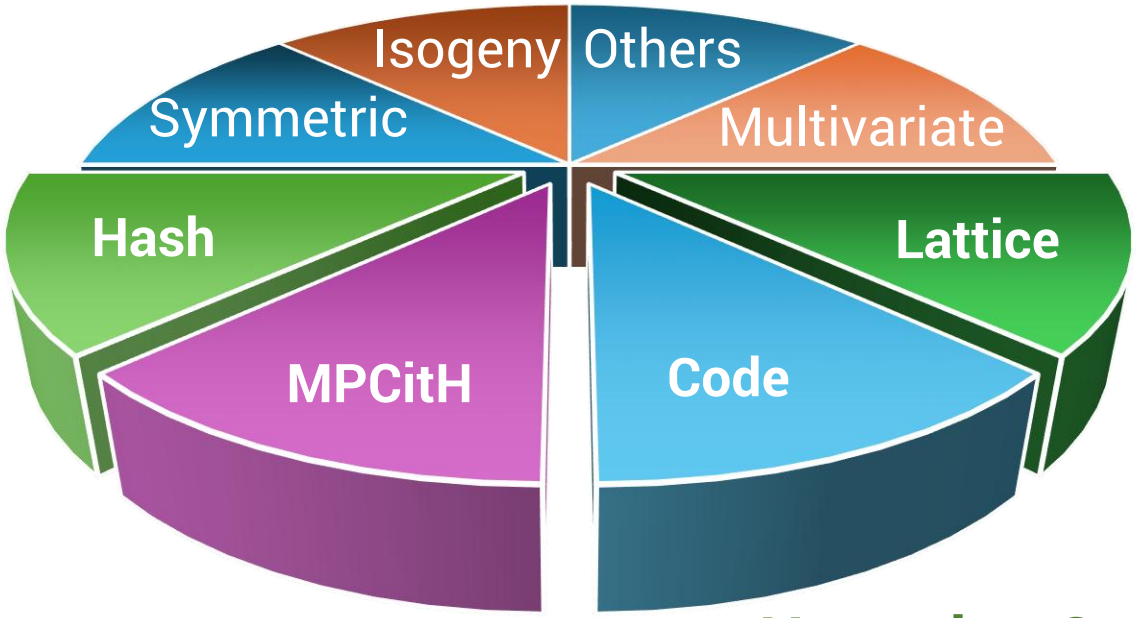
Secure and Efficient Implementations of Cryptographic Algorithms

- Target platforms:
  - CPU
  - Microcontrollers
  - **Field-Programmable Gate Array (FPGA)**
  - Application Specific Integrated Circuit (ASIC)
- Essential in improving security, performance and energy efficiency.



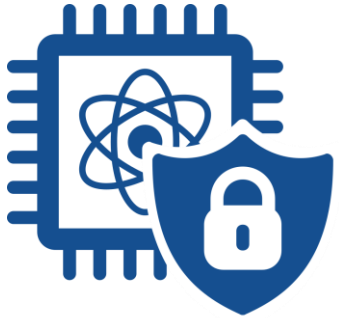
Today's Talk

Selected for standardization by NIST



Hamming Quasi-Cyclic (HQC)

Hamming Quasi-Cyclic (HQC)



Key Encapsulation Mechanism

Alice's Public Key

$(A_{PK}, A_{SK}) \leftarrow KeyGen(seed)$

Alice's Secret Key

$K' \leftarrow Decap(A_{SK}, C)$

Decapsulated Key

A_{PK}

C

Encapsulated Key

$(K, C) \leftarrow Encap(A_{PK})$

Ciphertext

Secure Channel

Symmetric Encryption

Hamming Quasi Cyclic (HQC)

- A code-based KEM scheme based on the syndrome decoding hard problem.
- Construction is based on concatenated codes:
  - Shortened Reed Solomon code and
  - Duplicated Reed-Muller code

Parameter Set	Security	*n*	*w*	|*ek*|	|*dk*|	|*ct*|
HQC-1	128	17,669	66	2,241B	2,321B	4,433B
HQC-3	192	35,851	100	4,514B	4,602B	8,978B
HQC-5	256	57,637	131	7,237B	7,333B	14,421B
X25519	128	–	–	32B	32B	32B

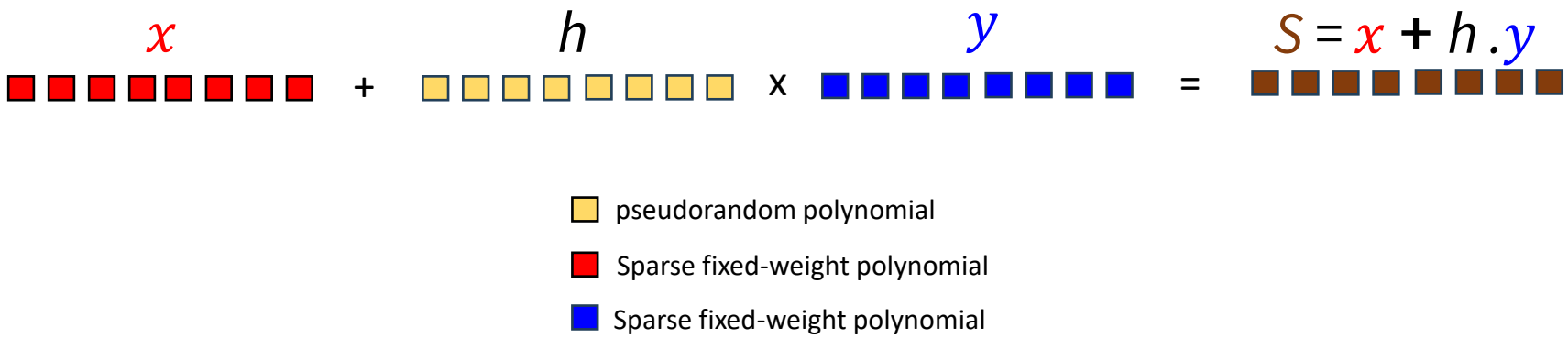
n - degree of the polynomial
w - weight
keypair – (ek,dk)
ct - ciphertext

☐ *Quantum-safe*

☐ *Not Quantum-safe*

Syndrome Decoding Problem

- h, x, y are polynomials in $\mathbb{F}_2[X]/(X^n-1)$
- Setup:
  - h – sampled from a pseudo-random number generator (PRNG)
  - x, y – fixed weight vectors (sparse polynomials) whose locations are sampled from a PRNG



h and S are public
x and y are secret
Given h and S, finding x and y is hard!

Key Generation

HQC-KEM.Keygen()

Input: None.

Output: Encapsulation key $\mathsf{ek}_{\mathsf{KEM}}$, decapsulation key $\mathsf{dk}_{\mathsf{KEM}}$.

▷ Sample $\mathsf{seed}_{\mathsf{KEM}}$

1. $\mathsf{seed}_{\mathsf{KEM}} \leftarrow\$ \mathcal{B}^{|\mathsf{seed}|}$

▷ Compute $\mathsf{seed}_{\mathsf{PKE}}$ and randomness σ

2. $\mathsf{ctx}_{\mathsf{KEM}} \leftarrow \mathsf{XOF.Init}(\mathsf{seed}_{\mathsf{KEM}})$
3. $(\mathsf{ctx}_{\mathsf{KEM}}, \mathsf{seed}_{\mathsf{PKE}}) \leftarrow \mathsf{XOF.GetBytes}(\mathsf{ctx}_{\mathsf{KEM}}, |\mathsf{seed}|)$
4. $(\mathsf{ctx}_{\mathsf{KEM}}, \sigma) \leftarrow \mathsf{XOF.GetBytes}(\mathsf{ctx}_{\mathsf{KEM}}, |k|)$

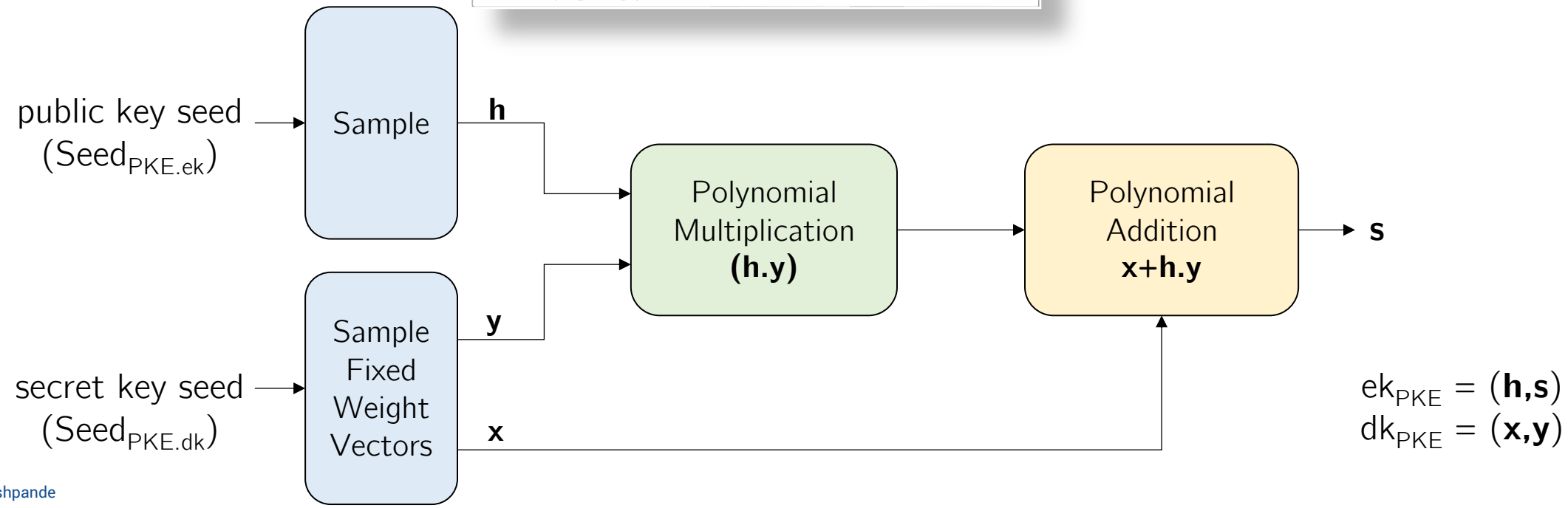
▷ Compute HQC-PKE keypair

5. $(\mathsf{ek}_{\mathsf{PKE}}, \mathsf{dk}_{\mathsf{PKE}}) \leftarrow \mathsf{HQC\text{-}PKE.Keygen}(\mathsf{seed}_{\mathsf{PKE}})$

▷ Compute HQC-KEM keypair

6. $\mathsf{ek}_{\mathsf{KEM}} \leftarrow \mathsf{ek}_{\mathsf{PKE}}$
7. $\mathsf{dk}_{\mathsf{KEM}} \leftarrow (\mathsf{ek}_{\mathsf{KEM}}, \mathsf{dk}_{\mathsf{PKE}}, \sigma, \mathsf{seed}_{\mathsf{KEM}})$

8. **return** $(\mathsf{ek}_{\mathsf{KEM}}, \mathsf{dk}_{\mathsf{KEM}})$



Encapsulation

HQC-KEM.Encaps($\mathsf{ek}_{\mathsf{KEM}}$)

Input: Encapsulation key $\mathsf{ek}_{\mathsf{KEM}}$.
Output: Shared secret key K, ciphertext $\mathbf{c}_{\mathsf{KEM}}$.

▷ Sample message $\mathbf{m}$ and salt
1. $\mathbf{m} \xleftarrow{\$} \mathcal{B}^{|k|}$
2. $\mathsf{salt} \xleftarrow{\$} \mathcal{B}^{|\mathsf{salt}|}$

▷ Compute shared key K and ciphertext $\mathbf{c}_{\mathsf{KEM}}$
3. $(K, \theta) \leftarrow \mathcal{G}(\mathcal{H}(\mathsf{ek}_{\mathsf{KEM}}) \| \mathbf{m} \| \mathsf{salt})$
4. $\mathbf{c}_{\mathsf{PKE}} \leftarrow \mathsf{HQC\text{-}PKE.Encrypt}(\mathsf{ek}_{\mathsf{KEM}}, \mathbf{m}, \theta)$
5. $\mathbf{c}_{\mathsf{KEM}} \leftarrow (\mathbf{c}_{\mathsf{PKE}}, \mathsf{salt})$
6. **return** $(K, \mathbf{c}_{\mathsf{KEM}})$

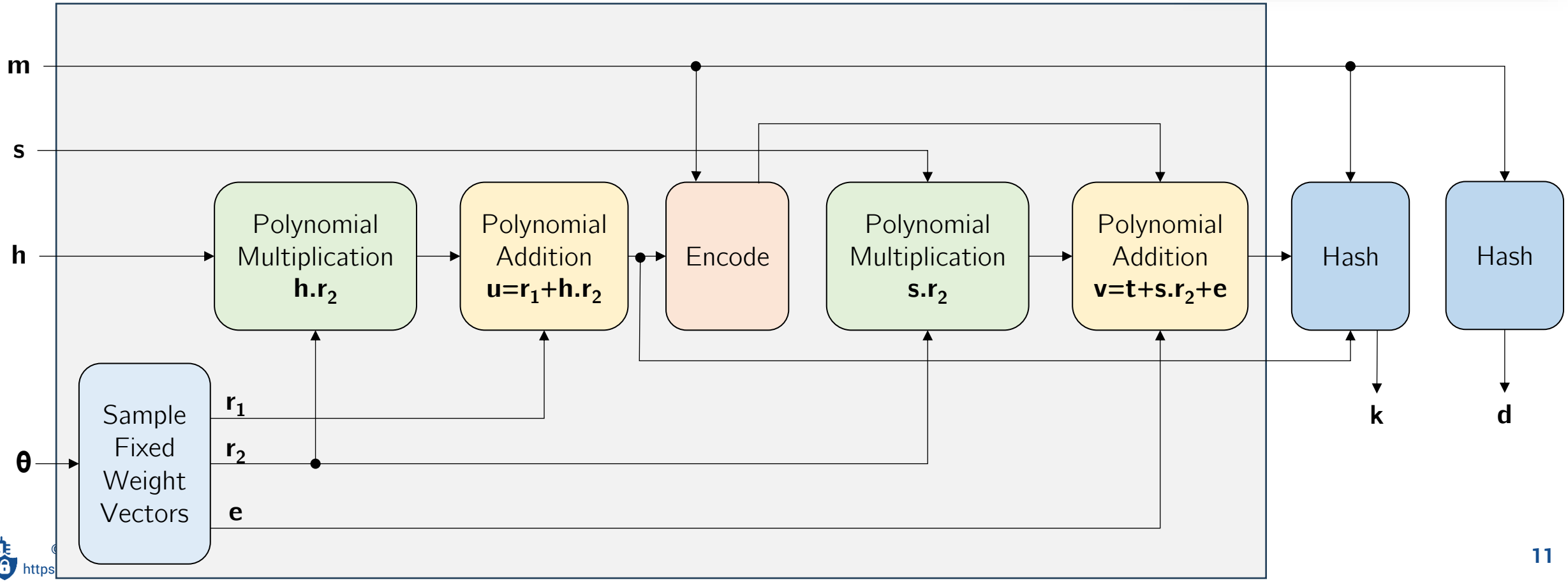
HQC-PKE.Encrypt($\mathsf{ek}_{\mathsf{PKE}}, \mathbf{m}, \theta$)

Input: Encryption key $\mathsf{ek}_{\mathsf{PKE}}$, message $\mathbf{m}$, randomness θ.
Output: Ciphertext $\mathbf{c}_{\mathsf{PKE}}$.

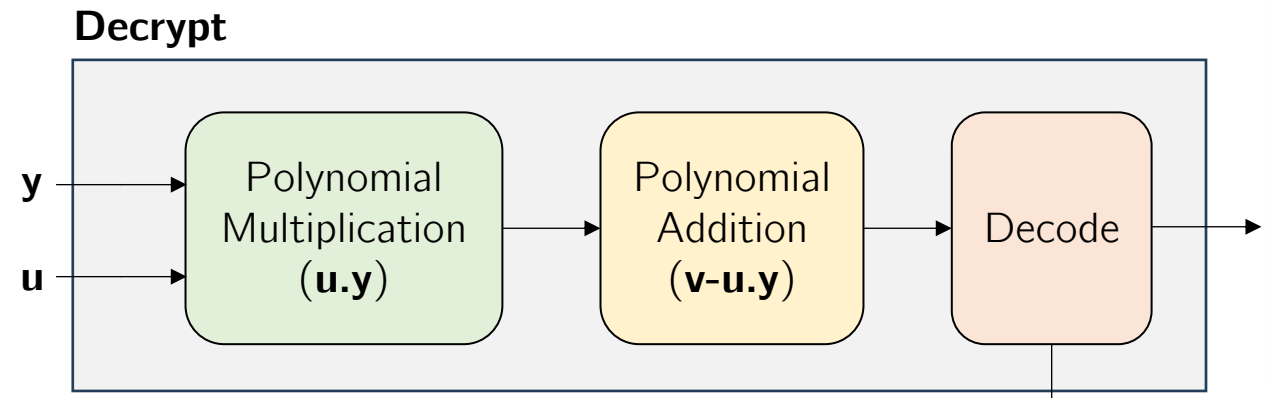
▷ Parse encryption key $\mathsf{ek}_{\mathsf{PKE}}$
1. $\mathsf{seed}_{\mathsf{PKE.ek}} \leftarrow \mathsf{ek}_{\mathsf{PKE}}[0 : |\mathsf{seed}|]$
2. $\mathsf{ctx}_{\mathsf{PKE.ek}} \leftarrow \mathsf{XOF.Init}(\mathsf{seed}_{\mathsf{PKE.ek}})$
3. $(\mathsf{ctx}_{\mathsf{PKE.ek}}, \mathbf{h}) \leftarrow \mathsf{SampleVect}(\mathsf{ctx}_{\mathsf{PKE.ek}}, \mathcal{R})$
4. $\mathbf{s} \leftarrow \mathsf{ek}_{\mathsf{PKE}}[|\mathsf{seed}| : |\mathsf{seed}| + |\mathbf{s}|]$

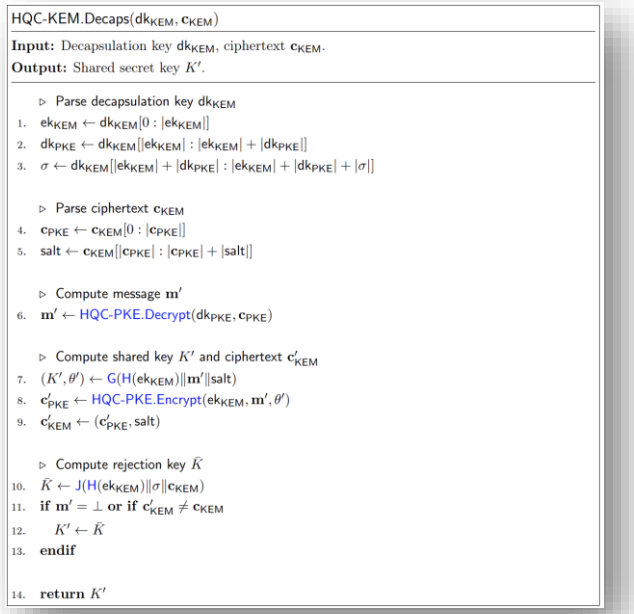
▷ Compute ciphertext $\mathbf{c}_{\mathsf{PKE}}$
5. $\mathsf{ctx}_\theta \leftarrow \mathsf{XOF.Init}(\theta)$
6. $(\mathsf{ctx}_\theta, \mathbf{r}_2) \leftarrow \mathsf{SampleFixedWeightVect}(\mathsf{ctx}_\theta, \mathcal{R}_{\omega_r})$
7. $(\mathsf{ctx}_\theta, \mathbf{e}) \leftarrow \mathsf{SampleFixedWeightVect}(\mathsf{ctx}_\theta, \mathcal{R}_{\omega_e})$
8. $(\mathsf{ctx}_\theta, \mathbf{r}_1) \leftarrow \mathsf{SampleFixedWeightVect}(\mathsf{ctx}_\theta, \mathcal{R}_{\omega_r})$
9. $\mathbf{u} \leftarrow \mathbf{r}_1 + \mathbf{h} \cdot \mathbf{r}_2$
10. $\mathbf{v} \leftarrow \mathcal{C}.\mathsf{Encode}(\mathbf{m}) + \mathsf{Truncate}(\mathbf{s} \cdot \mathbf{r}_2 + \mathbf{e}, \ell)$
11. $\mathbf{c}_{\mathsf{PKE}} \leftarrow (\mathbf{u}, \mathbf{v})$
12. **return** $\mathbf{c}_{\mathsf{PKE}}$

Encrypt



Decapsulation





HQC-KEM.Decaps(dk_{KEM}, c_{KEM})

Input: Decapsulation key dk_{KEM}, ciphertext c_{KEM}.
Output: Shared secret key K'.

▷ Parse decapsulation key dk_{KEM}
$ek_{KEM} \leftarrow dk_{KEM}[0 : |ek_{KEM}|]$
$dk_{PKE} \leftarrow dk_{KEM}[|ek_{KEM}| : |ek_{KEM}| + |dk_{PKE}|]$
$\sigma \leftarrow dk_{KEM}[|ek_{KEM}| + |dk_{PKE}| : |ek_{KEM}| + |dk_{PKE}| + |\sigma|]$

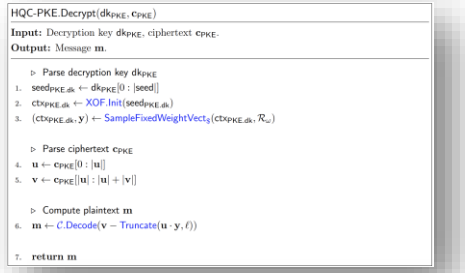
▷ Parse ciphertext c_{KEM}
$c_{PKE} \leftarrow c_{KEM}[0 : |c_{PKE}|]$
$salt \leftarrow c_{KEM}[|c_{PKE}| : |c_{PKE}| + |salt|]$

▷ Compute message m'
$m' \leftarrow$ HQC-PKE.Decrypt(dk_{PKE}, c_{PKE})

▷ Compute shared key K' and ciphertext c'_{KEM}
$(K', \theta') \leftarrow G(H(ek_{KEM}) \| m' \| salt)$
$c'_{PKE} \leftarrow$ HQC-PKE.Encrypt(ek_{KEM}, m', θ')
$c'_{KEM} \leftarrow (c'_{PKE}, salt)$

▷ Compute rejection key $\bar{K}$
$\bar{K} \leftarrow J(H(ek_{KEM}) \| \sigma \| c_{KEM})$
if $m' = \perp$ **or if** $c'_{KEM} \neq c_{KEM}$
  $K' \leftarrow \bar{K}$
endif

return K'



HQC-PKE.Decrypt(dk_{PKE}, c_{PKE})

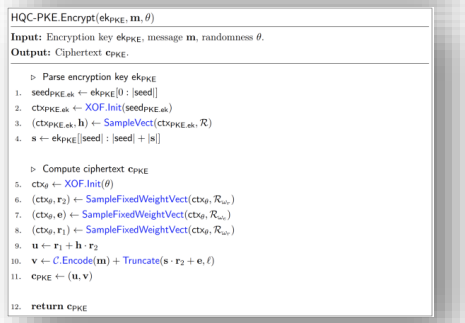
Input: Decryption key dk_{PKE}, ciphertext c_{PKE}.
Output: Message m.

▷ Parse decryption key dk_{PKE}
$seed_{PKE.dk} \leftarrow dk_{PKE}[0 : |seed|]$
$ctx_{PKE.dk} \leftarrow$ XOF.Init($seed_{PKE.dk}$)
$(ctx_{PKE.dk}, y) \leftarrow$ SampleFixedWeightVect$_3$($ctx_{PKE.dk}$, $\mathcal{R}_\omega$)

▷ Parse ciphertext c_{PKE}
$u \leftarrow c_{PKE}[0 : |u|]$
$v \leftarrow c_{PKE}[|u| : |u| + |v|]$

▷ Compute plaintext m
$m \leftarrow \mathcal{C}$.Decode($v - $ Truncate($u \cdot y, \ell$))

return m



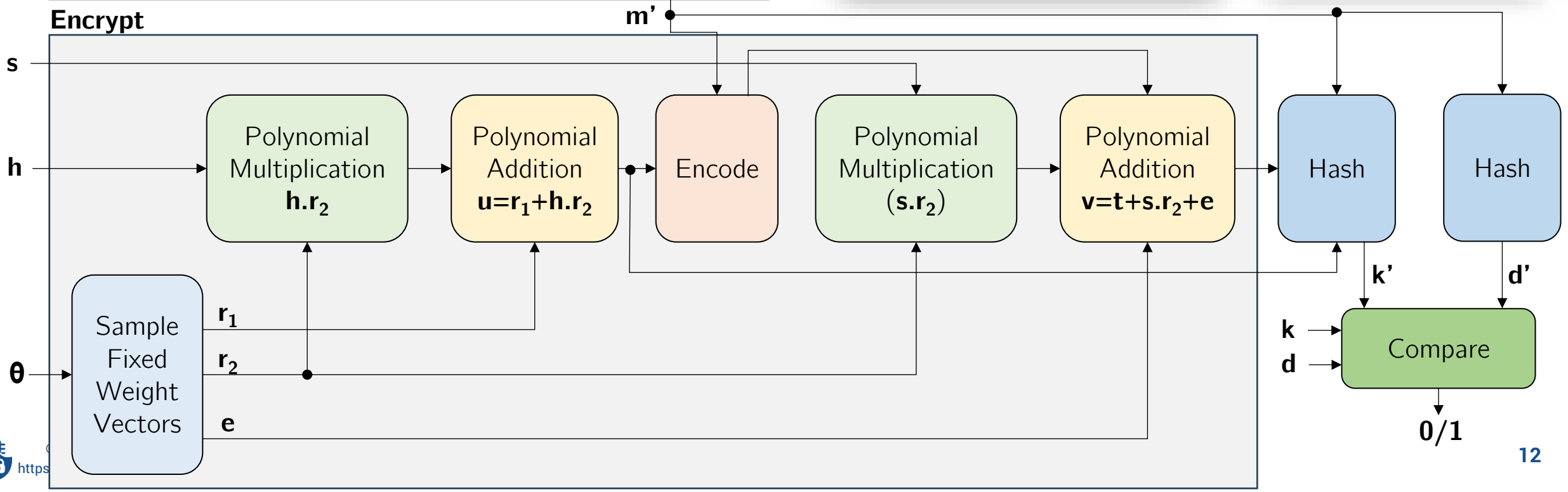
HQC-PKE.Encrypt(ek_{PKE}, m, θ)

Input: Encryption key ek_{PKE}, message m, randomness θ.
Output: Ciphertext c_{PKE}.

▷ Parse encryption key ek_{PKE}
$seed_{PKE.ek} \leftarrow ek_{PKE}[0 : |seed|]$
$ctx_{PKE.ek} \leftarrow$ XOF.Init($seed_{PKE.ek}$)
$(ctx_{PKE.ek}, h) \leftarrow$ SampleVect($ctx_{PKE.ek}$, $\mathcal{R}$)
$s \leftarrow ek_{PKE}[|seed| : |seed| + |s|]$

▷ Compute ciphertext c_{PKE}
$ctx_\theta \leftarrow$ XOF.Init(θ)
$(ctx_\theta, r_2) \leftarrow$ SampleFixedWeightVect(ctx_θ, $\mathcal{R}_\omega$)
$(ctx_\theta, e) \leftarrow$ SampleFixedWeightVect(ctx_θ, $\mathcal{R}_\omega$)
$(ctx_\theta, r_1) \leftarrow$ SampleFixedWeightVect(ctx_θ, $\mathcal{R}_\omega$)
$u \leftarrow r_1 + h \cdot r_2$
$v \leftarrow \mathcal{C}$.Encode(m) + Truncate($s \cdot r_2 + e, \ell$)
$c_{PKE} \leftarrow (u, v)$

return c_{PKE}



Performance Comparison HQC vs Elliptic Curve Cryptography

Server

Client

$K' \leftarrow Decap(A_{SK}, C)$

K'-Decapsulated Key

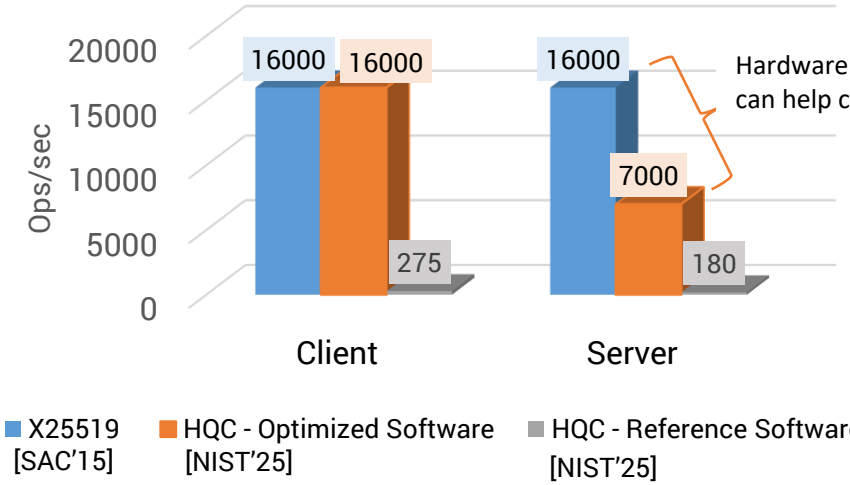
*Key generation is performed offline on the server side

C

$(K, C) \leftarrow Encap(A_{PK})$

K-Encapsulated Key
C-Ciphertext

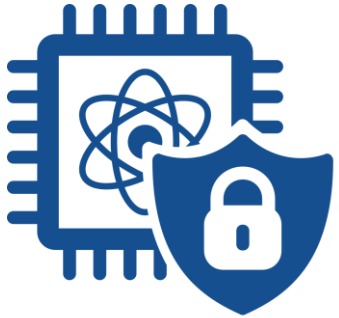
Operations Per Second Comparison on a CPU



Operating frequency:
CPU: Intel Core i7-11850H – 2.5 GHz

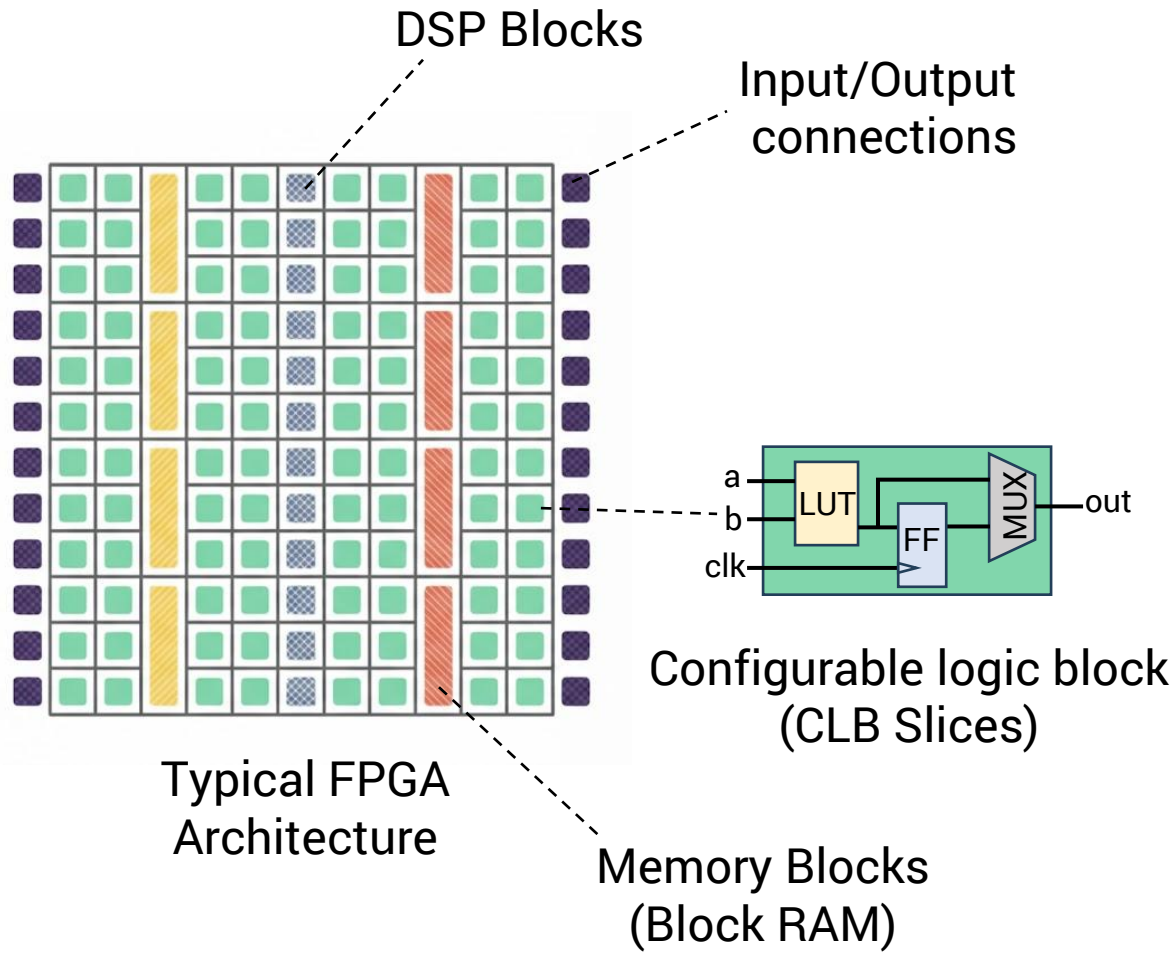
*Performance varies considerably by hardware platform and implementation constraints, and should be taken as a rough indication only.

Hardware Acceleration
Using FPGAs



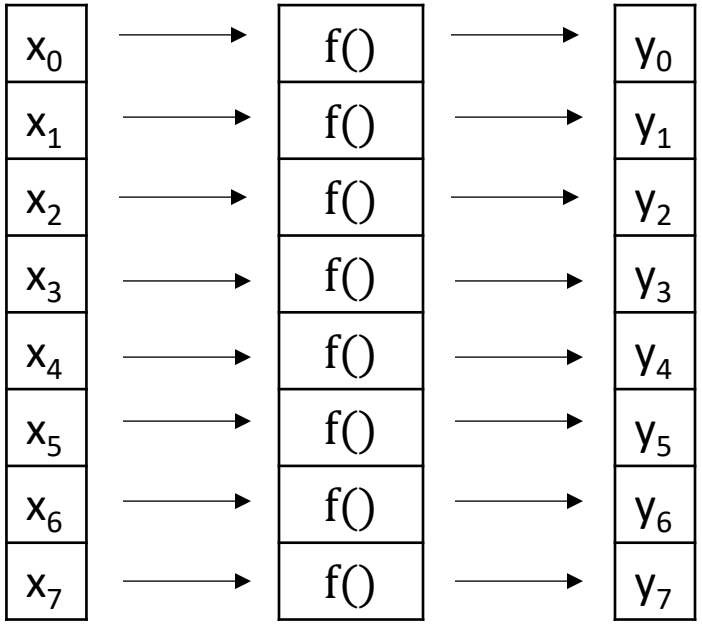
Field Programmable Gate Array (FPGA)

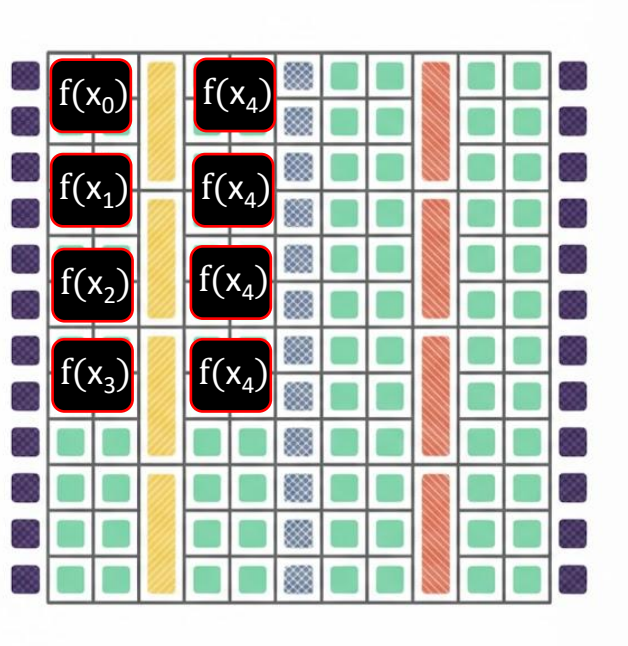
- FPGA is a reconfigurable substrate
- **Reconfigurable** functions, interconnection of functions, input/output
- Depending on application, they can offer **higher throughput**



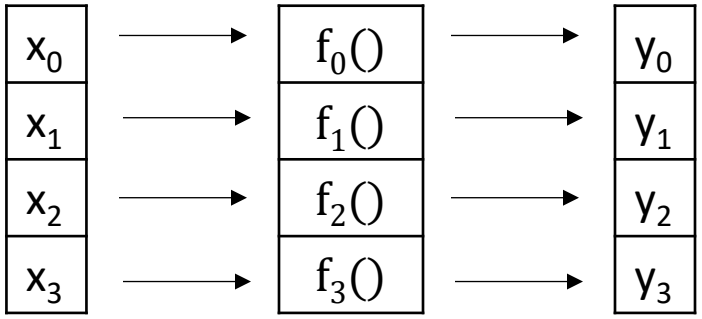
FPGA Throughput – Example 1

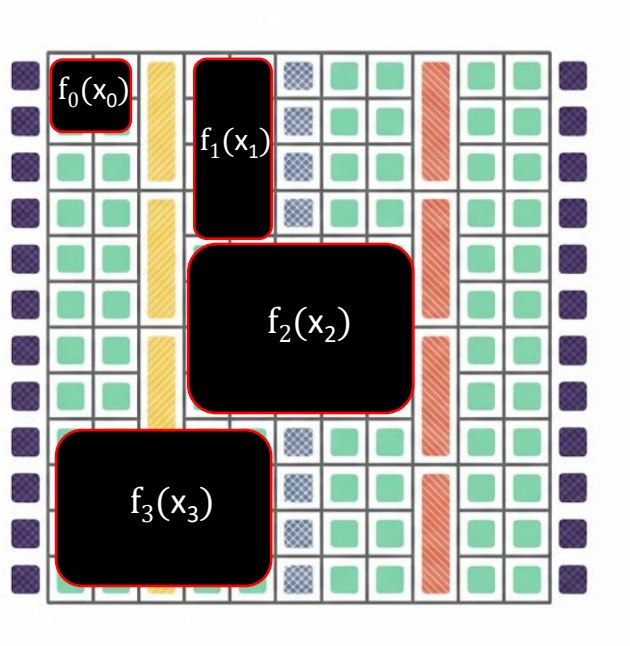
- Compute $y = f(x)$ for $x = [x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7]$





FPGA Throughput – Example 2

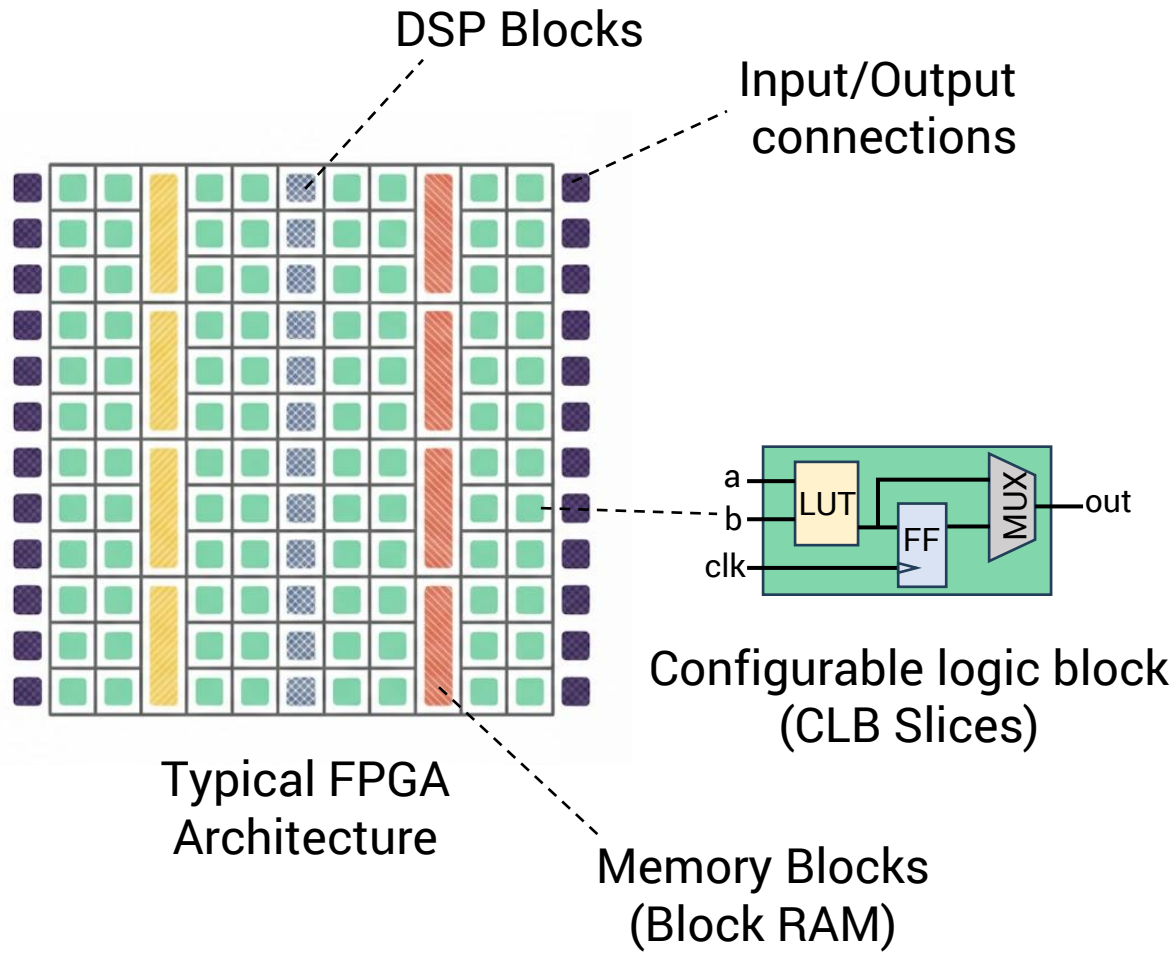




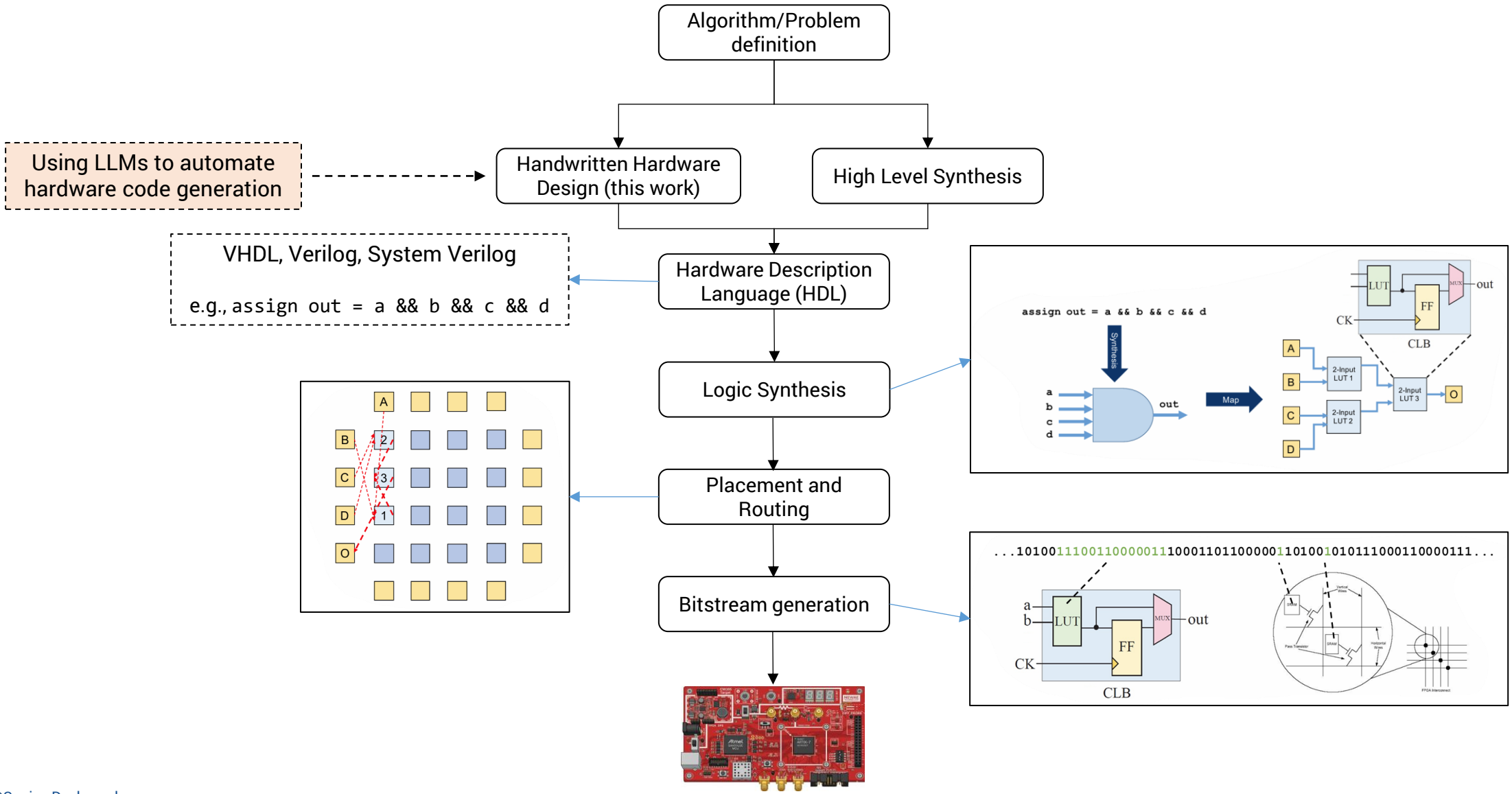
FPGA Implementation Performance and Evaluation Metrics

- Area
- Time
- Time-Area product
  - Time * max{LUT/4, FF/8} + 128 ∗ BRAM + 100 ∗ DSP}

- For reference (AMD Artix 7 200 T FPGA):
  - Look up Tables (LUTs) = 215,360
  - Digital Signal Processing Blocks (DSPs) = 740
  - Flipflops (FFs) = 269,200
  - Block RAM (BRAM) = 365 (each block can store 36 Kb)

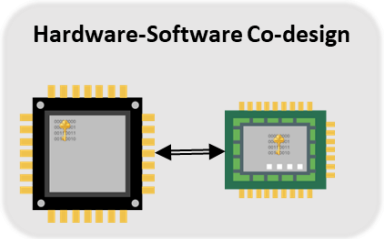


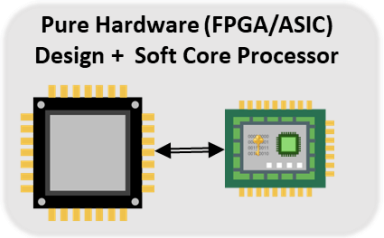
FPGA Design Flow

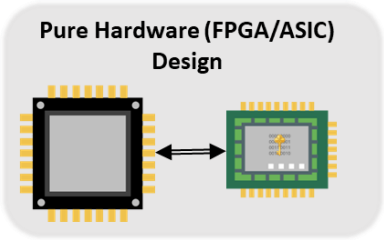


Hardware Platform Configurations and Interfaces

- ⟺ represents the interface — the link between the CPU and the hardware accelerator
- Communication protocol depends on placement:
  - AXI (Advanced eXtensible Interface)
  - PCIe (Peripheral Component Interconnect Express)
  - UART (Universal Asynchronous Receiver/Transmitter)
  - …
- Standardized protocols enable IP reuse and easier verification across different systems





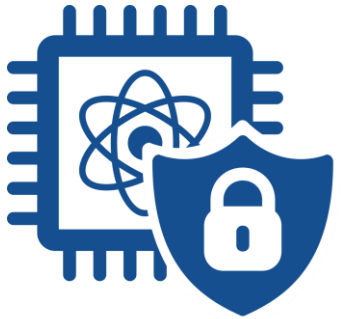


Hardware vs Software Development

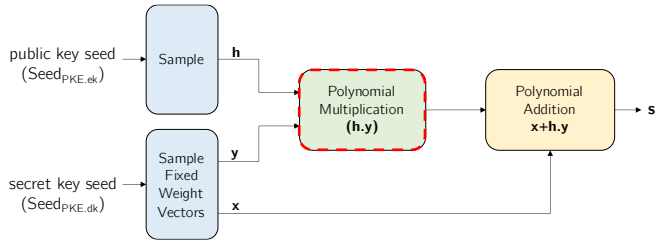
	CPU	GPU	FPGA	ASIC
Engineering time	short	medium	long	very long
Compilation time	short	short	long	very long
Debugging	easy	medium	hard	very hard
Maintainability	easy	medium	hard	very hard

Hardware Implementation

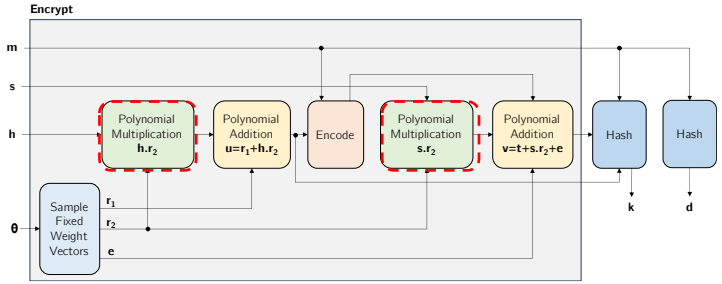
HQC



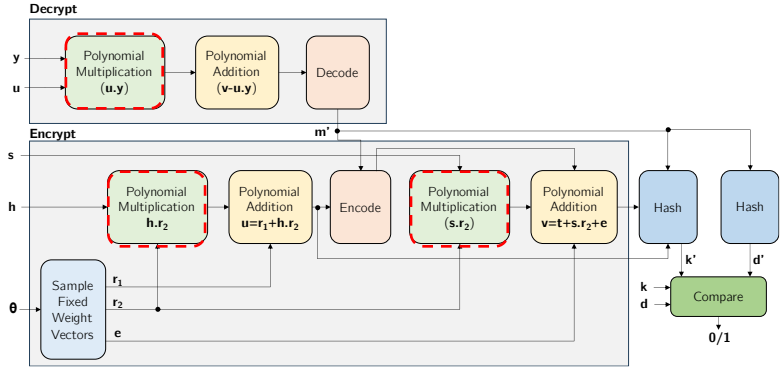
HQC Primitives



Key Generation



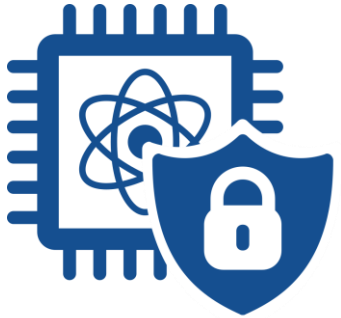
Encapsulation



Decapsulation

Performance Bottleneck.
95-96% across different parameter sets

Performance Bottleneck
Polynomial Multiplication



Polynomial Multiplication

A & B are polynomials in $\mathbb{F}_2[X]/(X^n-1)$

$$A(x)=a_0+a_1x+a_2x^2+\cdots+a_nx^n \qquad B(x)=b_0+b_1x+b_2x^2+\cdots+b_nx^n$$

$$c(x) = A(x).B(x) = \sum_{k=0}^{n+n} c_k x^k \qquad \text{Where each} \quad c_k = \sum_{i=0}^{k} a_i b_{k\_i}$$

Polynomial Multiplication – Schoolbook Technique

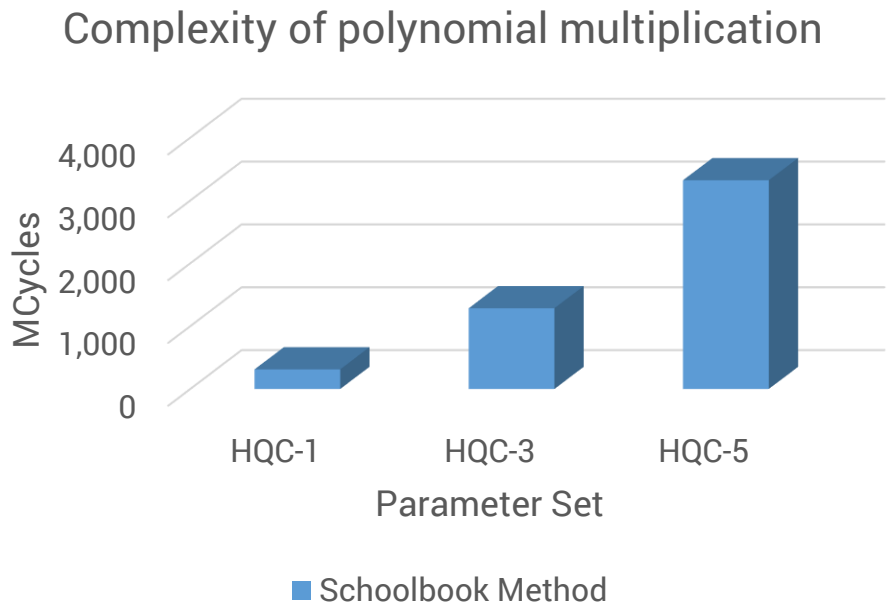
A & B are polynomials in $\mathbb{F}_2[X]/(X^n-1)$

$$A(x)=a_0+a_1x \qquad B(x)=b_0+b_1x$$

$$C(x)=a_1b_1x^2+(a_1b_0+a_0b_1)x+a_0b_0$$

Overall complexity = $O(n^2)$

Parameter Set	*n*	*w*
HQC-1	17,669	66
HQC-3	35,851	100
HQC-5	57,637	131



Polynomial Multiplication – Karatsuba Algorithm

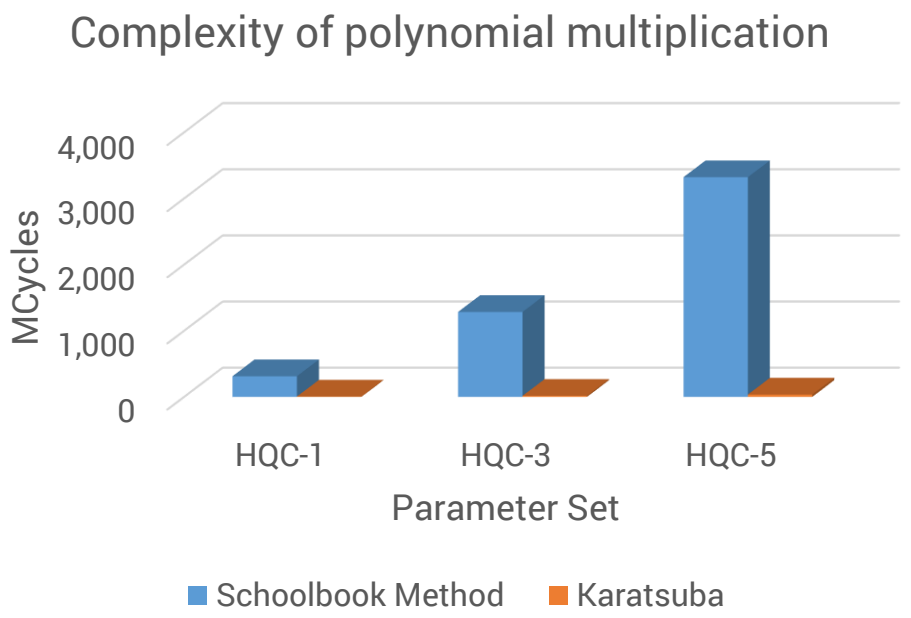
A & B are polynomials in $\mathbb{F}_2[X]/(X^n-1)$

$$A(x)=a_0+a_1x \qquad B(x)=b_0+b_1x$$

$$C(x)=\underbrace{a_1b_1}_{c_2}x^2+\underbrace{(a_1b_0+a_0b_1)}x+\underbrace{a_0b_0}_{c_0}$$

$$c_1=(a_0+a_1)(b_0+b_1)-c_0-c_2$$

Parameter Set	*n*	*w*
HQC-1	17,669	66
HQC-3	35,851	100
HQC-5	57,637	131



Schoolbook = $O(n^2)$

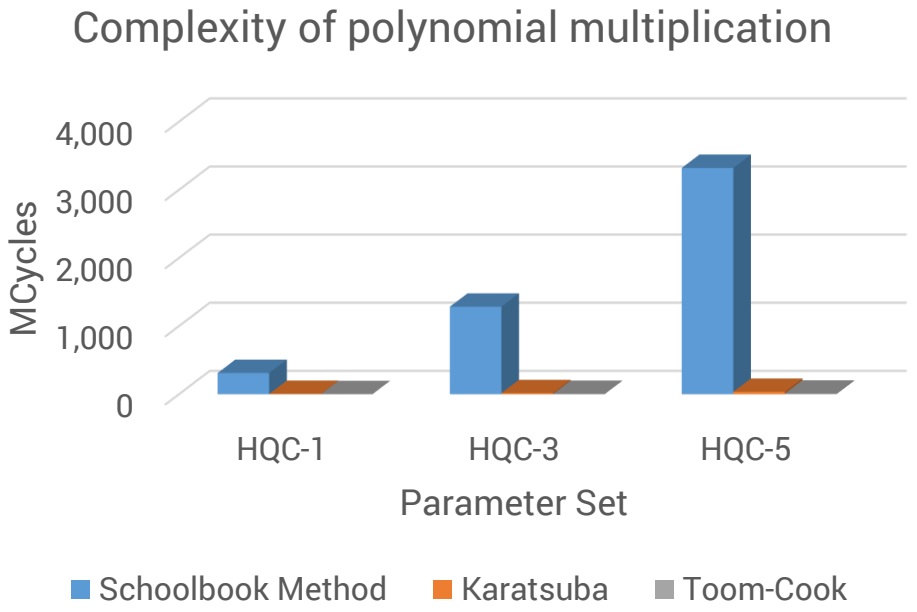
Karatsuba = $O(n^{1.585})$

Polynomial Multiplication – Toom-Cook Algorithm

A & B are polynomials in $\mathbb{F}_2[X]/(X^n-1)$

$$A(x)=a_0+a_1x+a_2x^2+\cdots+a_nx^n \qquad B(x)=b_0+b_1x+b_2x^2+\cdots+b_nx^n$$

Parameter Set	*n*	*w*
HQC-1	17,669	66
HQC-3	35,851	100
HQC-5	57,637	131



Schoolbook = $O(n^2)$
Karatsuba = $O(n^{1.585})$
Toom-Cook = $O(n^{1.465})$

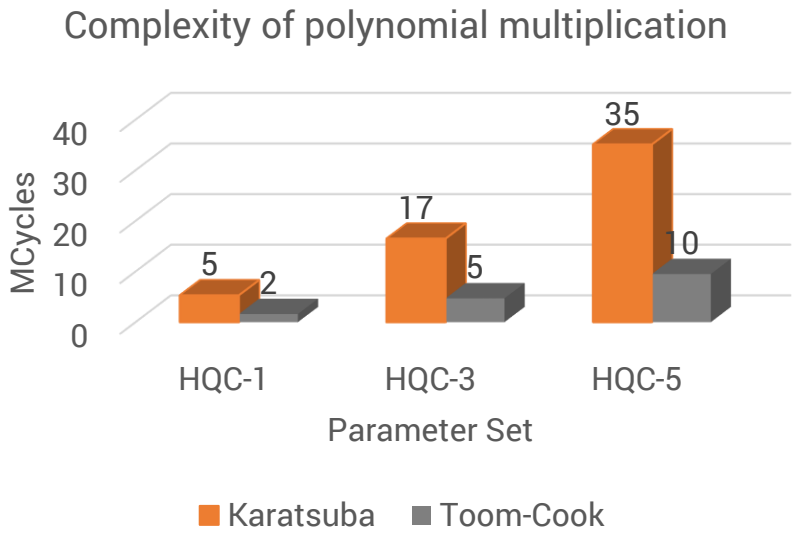
Polynomial Multiplication - Can we do better?

A & B are polynomials in $\mathbb{F}_2[X]/(X^n-1)$

$$A(x)=a_0+a_1x+a_2x^2+\cdots+a_nx^n \qquad B(x)=0+b_1x+0+\cdots+b_mx^m$$

One of the inputs to the polynomial multiplier is a sparse fixed weight vector.

Parameter Set	*n*	*w*
HQC-1	17,669	66
HQC-3	35,851	100
HQC-5	57,637	131



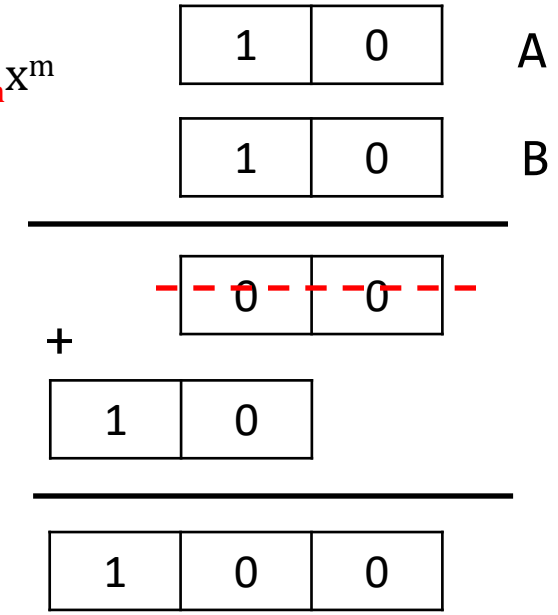
Karatsuba = $O(n^{1.585})$
Toom-Cook = $O(n^{1.465})$

Polynomial Multiplication involving Sparse Polynomials

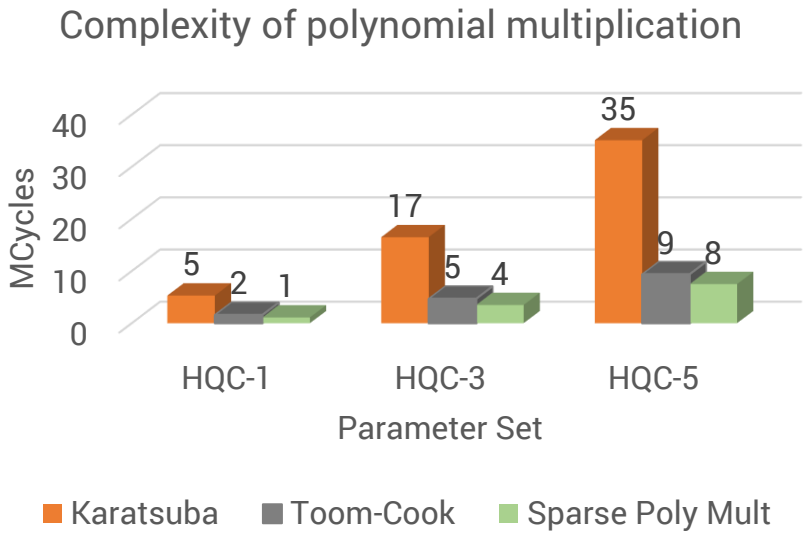
A & B are polynomials in $\mathbb{F}_2[X]/(X^n-1)$

$$A(x)=a_0+a_1x+a_2x^2+\cdots+a_nx^n \qquad B(x)=0+b_1x+0+\cdots+b_mx^m$$

$$c(x) = \sum_{i=0}^{w-1} A(x) << bindex[i]$$



Parameter Set	*n*	*w*
HQC-1	17,669	66
HQC-3	35,851	100
HQC-5	57,637	131

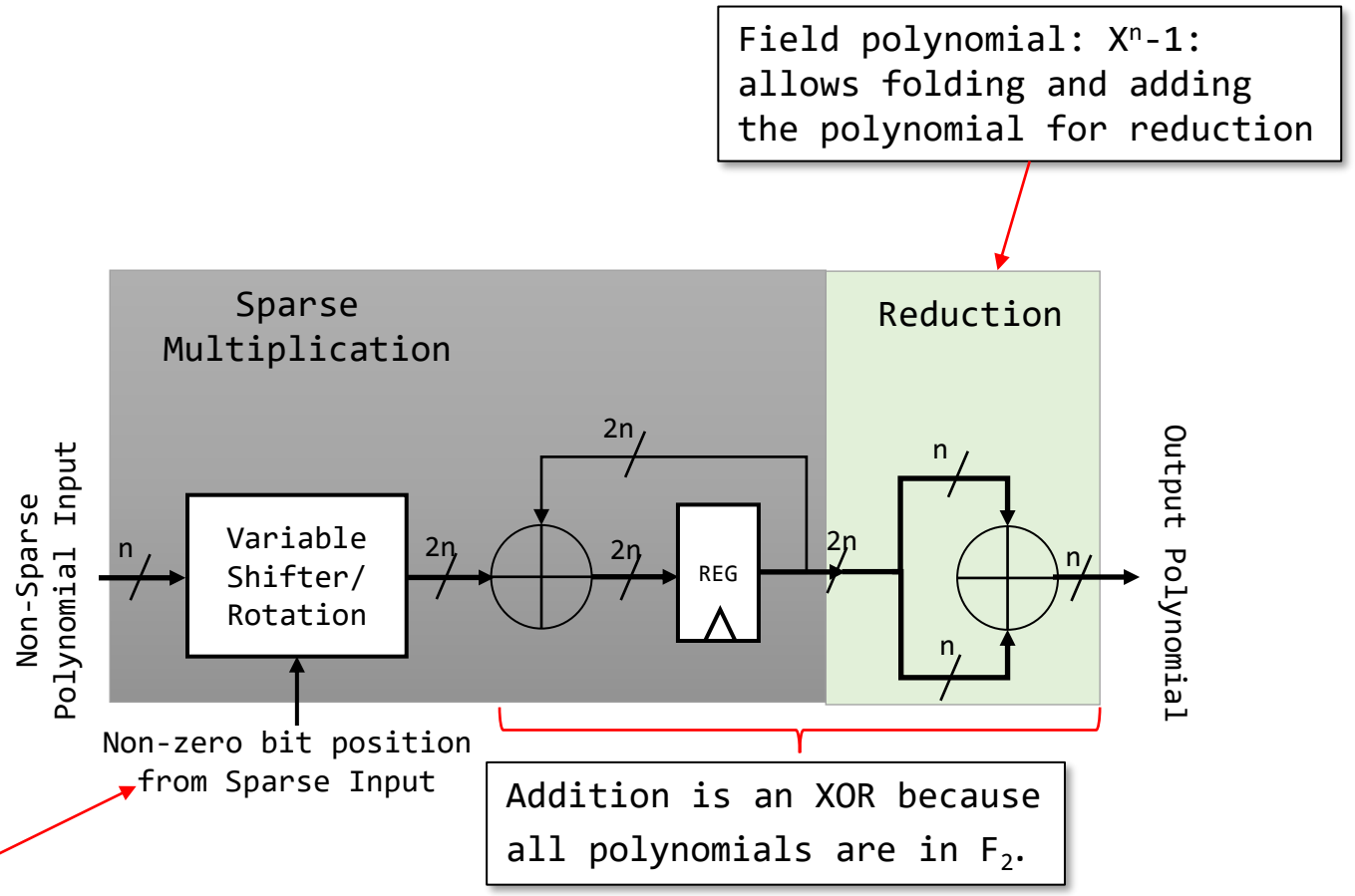


Karatsuba = $O(n^{1.585})$

Toom-Cook = $O(n^{1.465})$

Sparse poly mult = O(n.w)

Sparse F_2 Polynomial Multiplication with Reduction



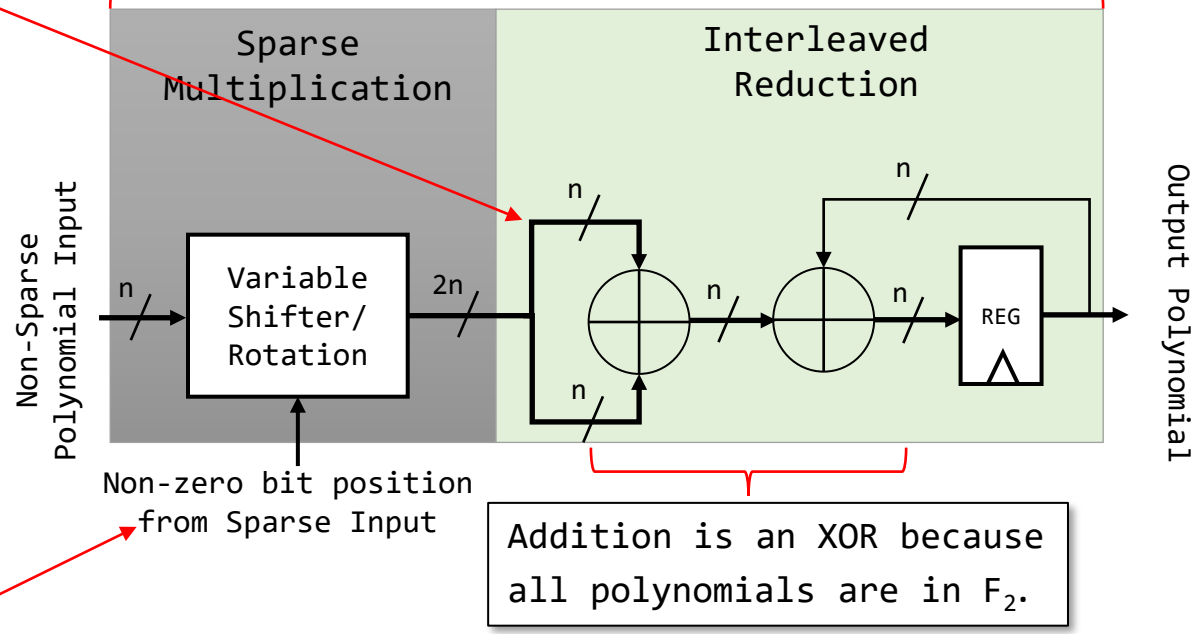
Conceptual Block Diagram

Sparse F_2 Polynomial Multiplication with Interleaved Reduction

Field polynomial: X^n-1: allows folding and adding the polynomial for reduction

The multiplication and the modular reduction is interleaved.

- Reduction in area
- Reduction in memory utilization

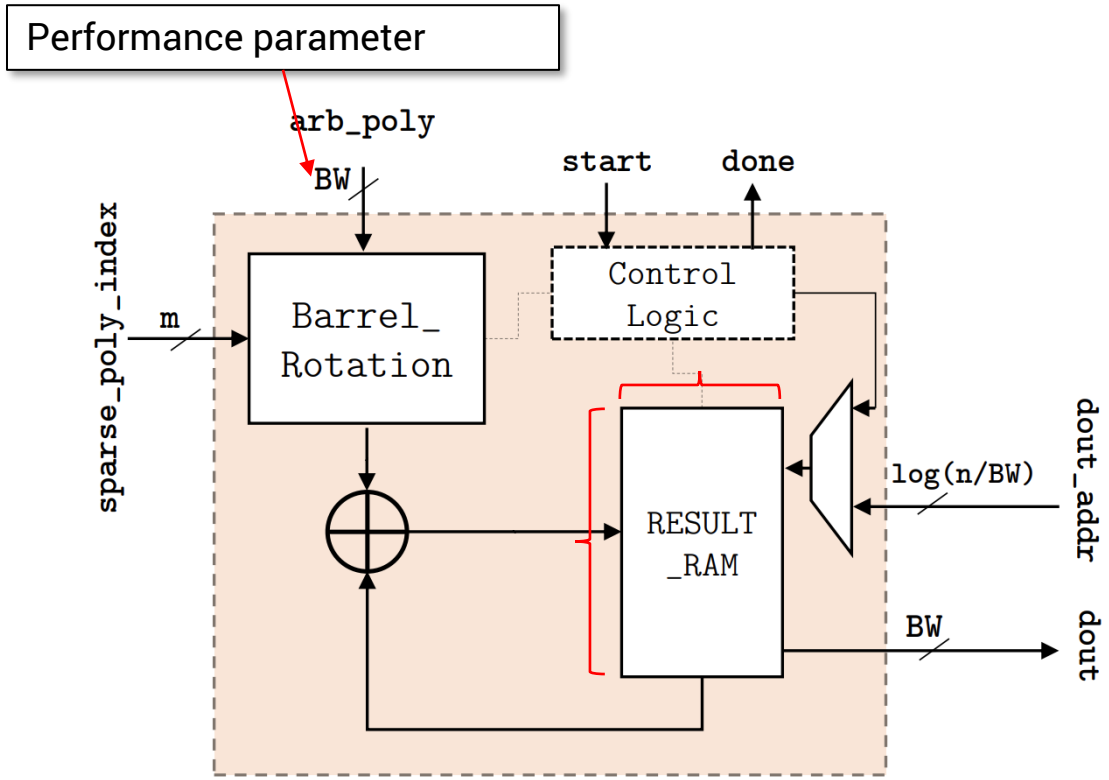


Addition is an XOR because all polynomials are in F_2.

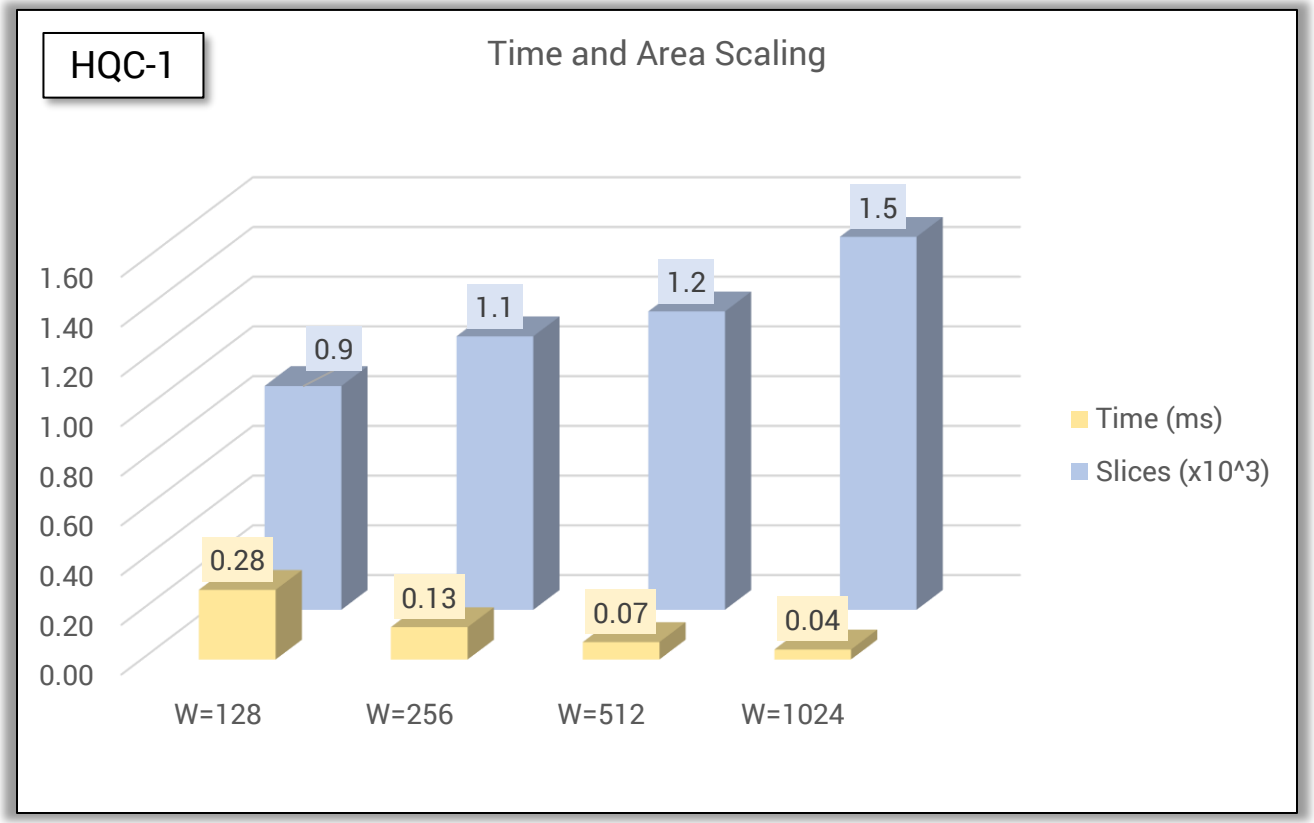
Indices of non-zero elements are used to shift non-sparse polynomial to imitate the multiplication.

Conceptual Block Diagram

Sparse F_2 Polynomial Multiplication Hardware Design



Hardware design of sparse polynomial multiplication module



Parameter Set	*n*	*w*
HQC-1	17,669	66
HQC-3	35,851	100
HQC-5	57,637	131

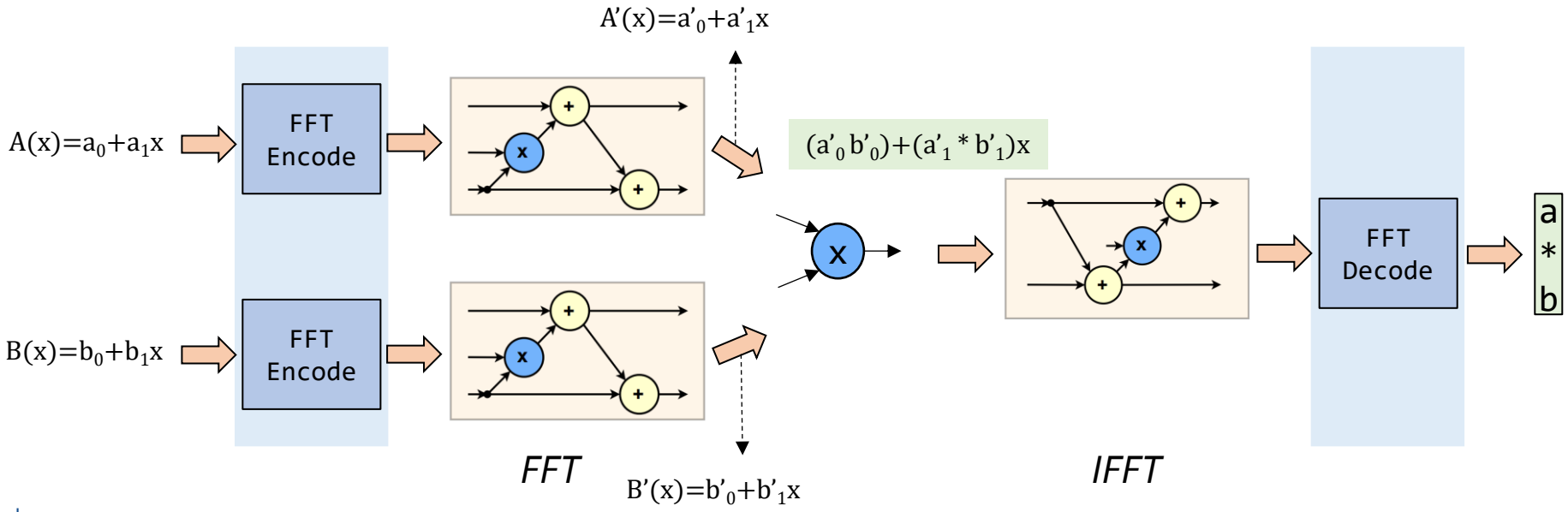
Fast Fourier Transform (FFT)/ Number Theoretic Transform (NTT)

Recall: **Schoolbook Technique**

$A(x)=a_0+a_1x \quad B(x)=b_0+b_1x$

1) Multiplication: $C(x)=a_1b_1x^2 + (a_1b_0+ a_0b_1)x + a_0b_0$
2) Reduction: $C(x)$ % X^n-1

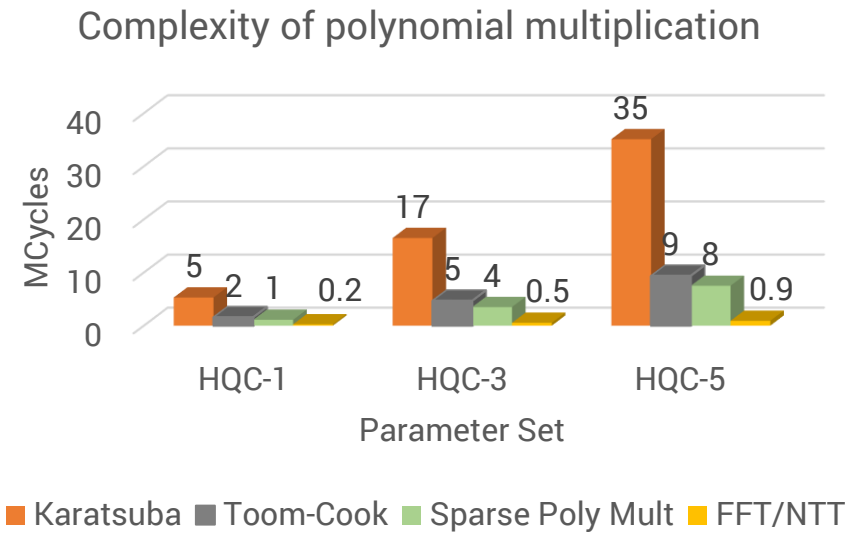
Complexity = $O(n^2)$



Complexity = O(n.log(n))

Is it possible to apply NTT/FFT type multiplication to HQC?

Parameter Set	*n*	*w*
HQC-1	17,669	66
HQC-3	35,851	100
HQC-5	57,637	131

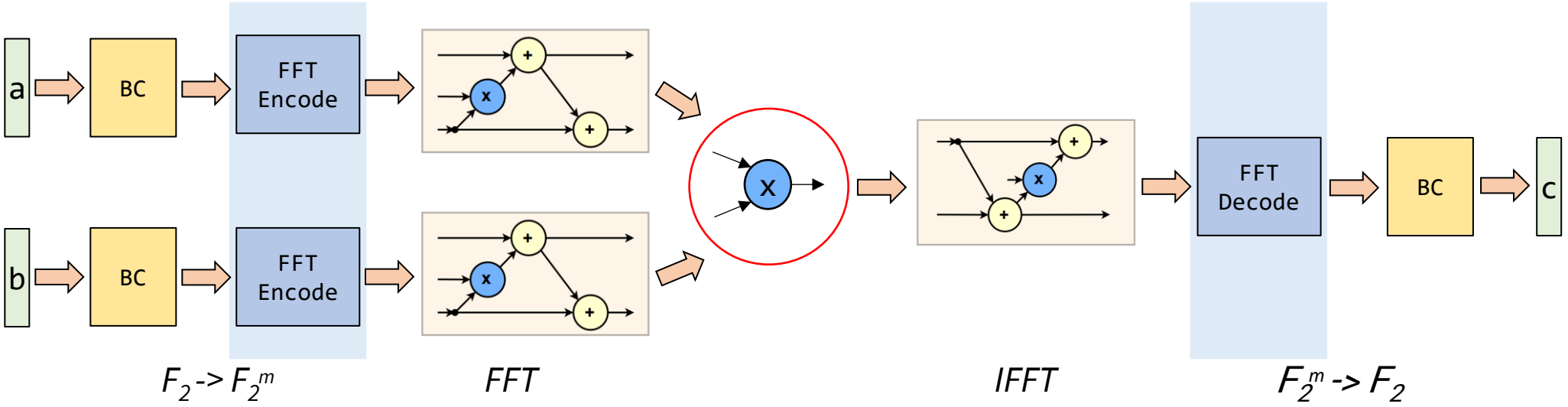


Karatsuba = $O(n^{1.585})$
Toom-Cook = $O(n^{1.465})$
Sparse poly mult = O(n.w)
FFT/NTT = *O(n.logn)*

- Cannot directly apply NTT (Number Theoretic Transform) to binary field polynomial multiplication
  - Requires a finite field with special roots of unity
- Chen et. al [CCK21] proposed using **Frobenius Additive FFTs (FAFFT).**

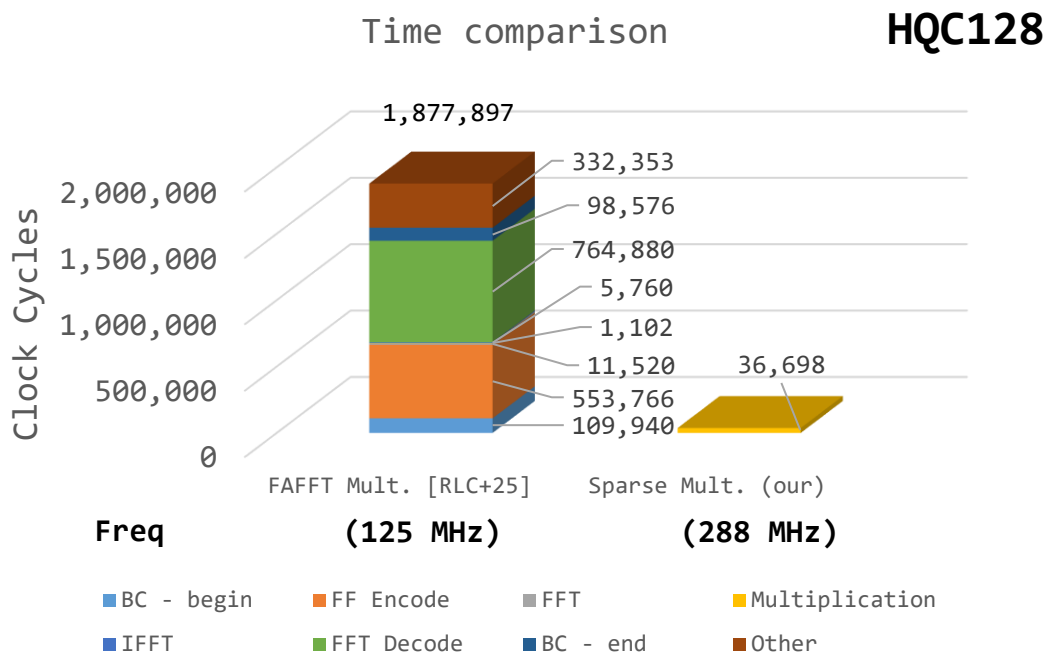
Frobenius Additive FFT (FAFFT)

- Convert to new polynomial basis – **Cantor Basis.**
- **FFT Encode** and **FFT Decode** are used to reduce the number of evaluations.

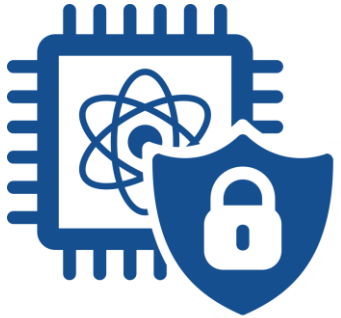


Performance FAFFT vs Sparse Multiplication

- Implemented in hardware by Ras et. al [RLC+25]
- Taking advantage of sparsity doesn't lead to non-constant time behavior on hardware.
  - Weight of the sparse polynomial is fixed.



Key Generation, Encapsulation, and Decapsulation

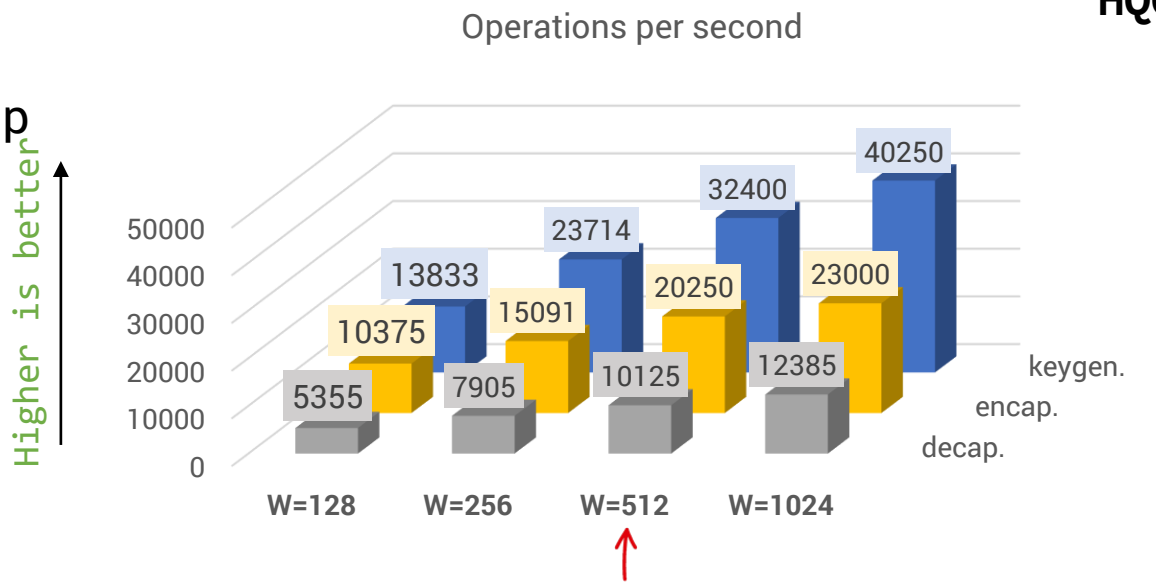


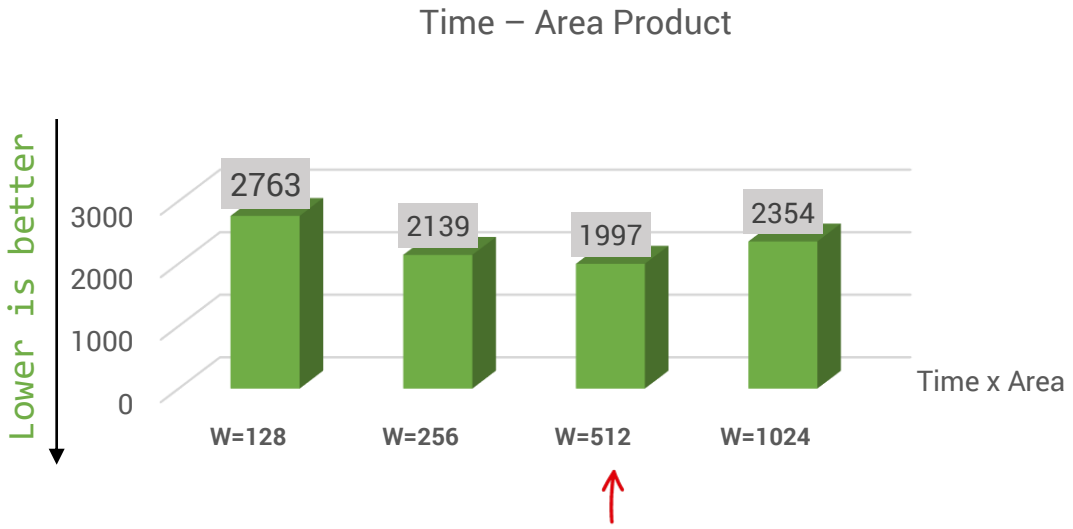
Joint Design – Sweep

FPGA: AMD Artix 7

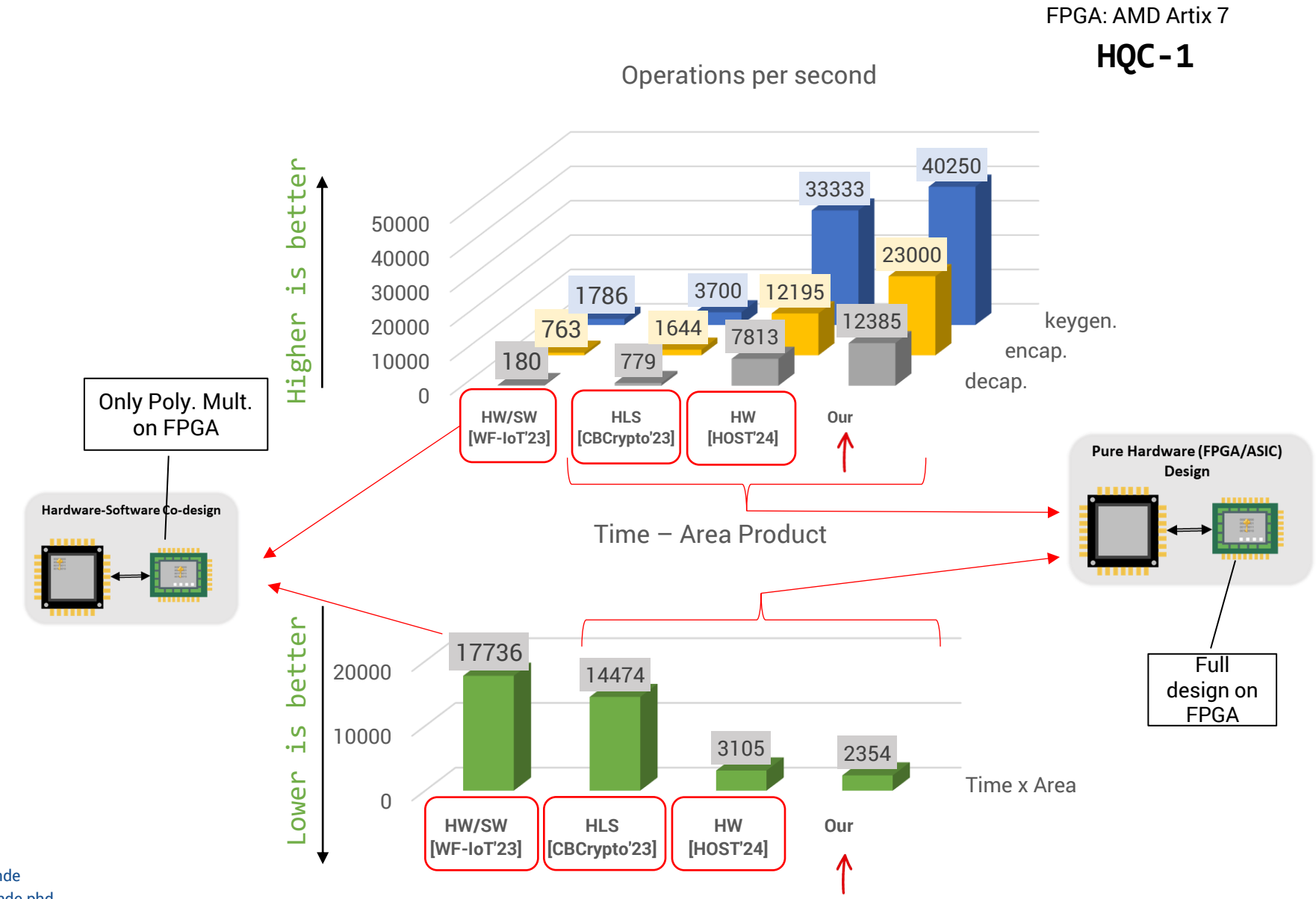
HQC-1

- Joint Design – combines Keygen, Encap, and Decap into one.
- Shared module among different primitives.
  - SHAKE256
  - Polynomial Multiplication
  - Encapsulation
  - Polynomial Addition
- W – performance parameter
- Time-Area product
  - Time * max{LUT/4 + 128 ∗ BRAM + 100 ∗ DSP, FF/8}

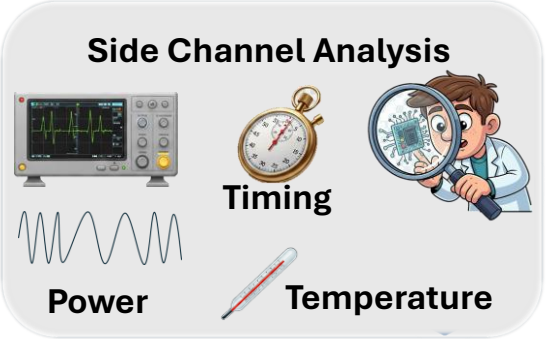




Joint Design – Comparison with other HQC Hardware Designs

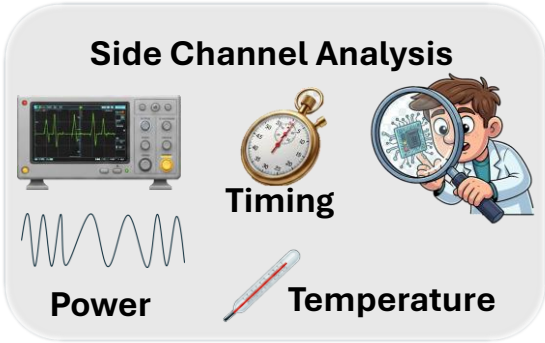


Side-Channel Evaluation

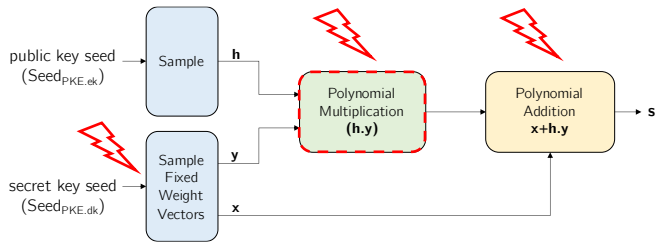


Side Channel Evaluation

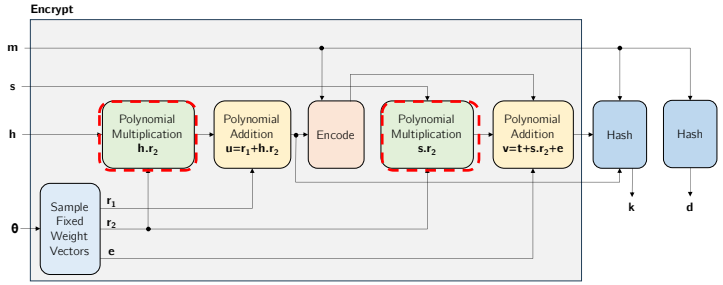
- **Side-channel** attacks are based on the **implementation** of a system
- It is not a brute-force attack or a theoretical weakness based attack
- Observe **unintentional features** such as timing, power, etc



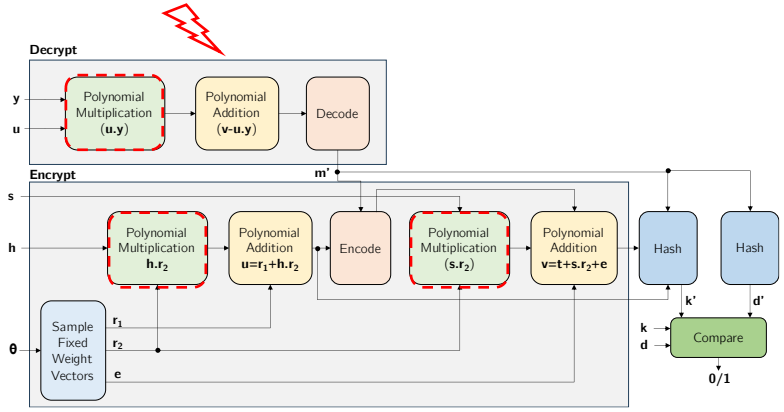
HQC Primitives



Key Generation



Encapsulation

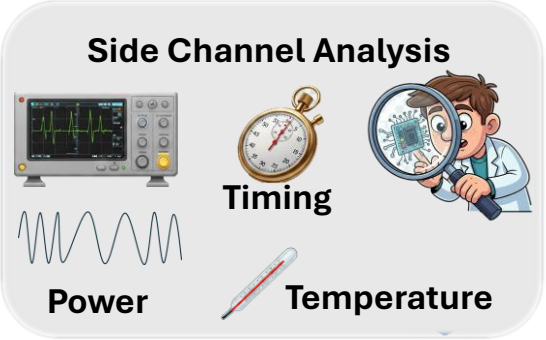


Decapsulation

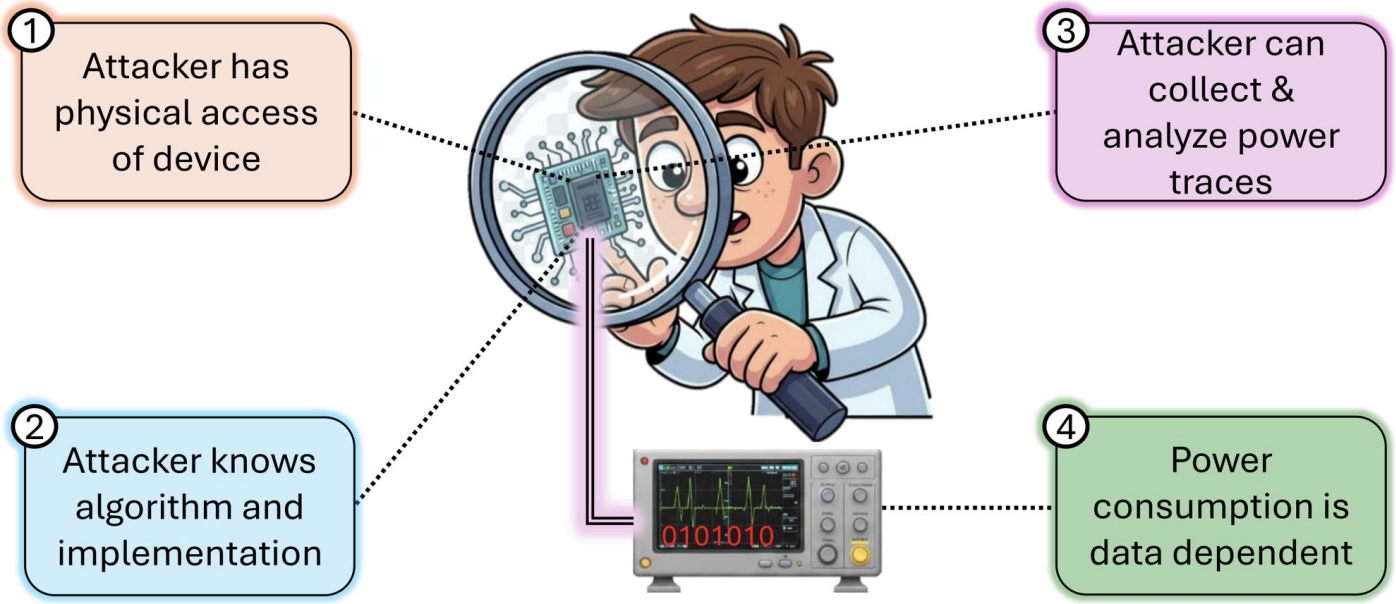
Performance Bottleneck

Side-channel attack target

Power Side-Channel Evaluation

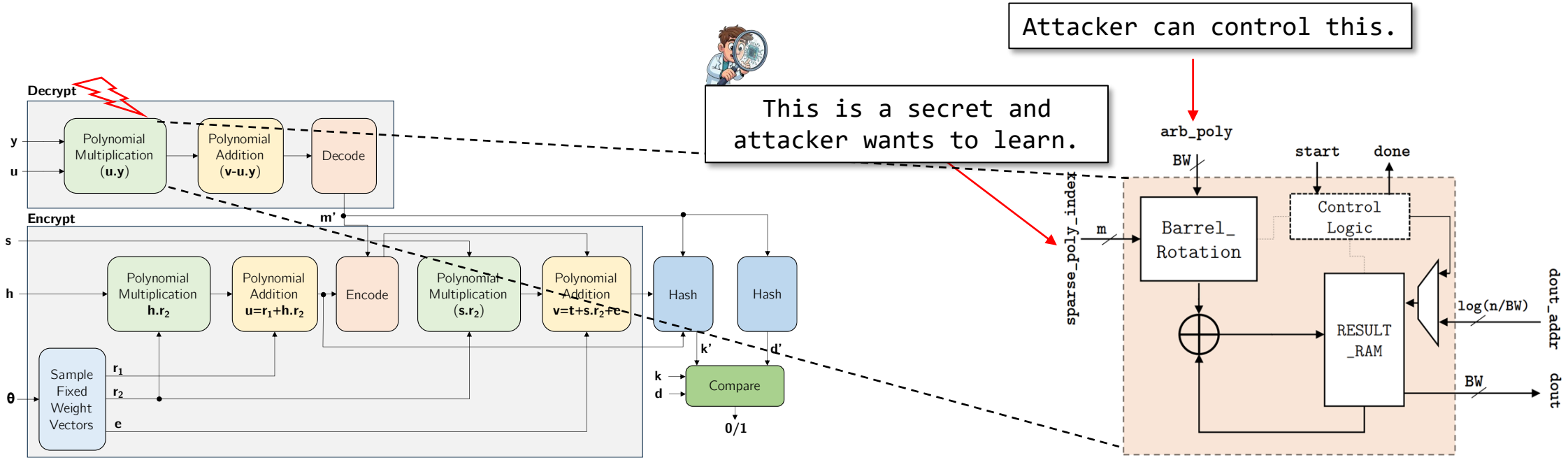


Threat Model



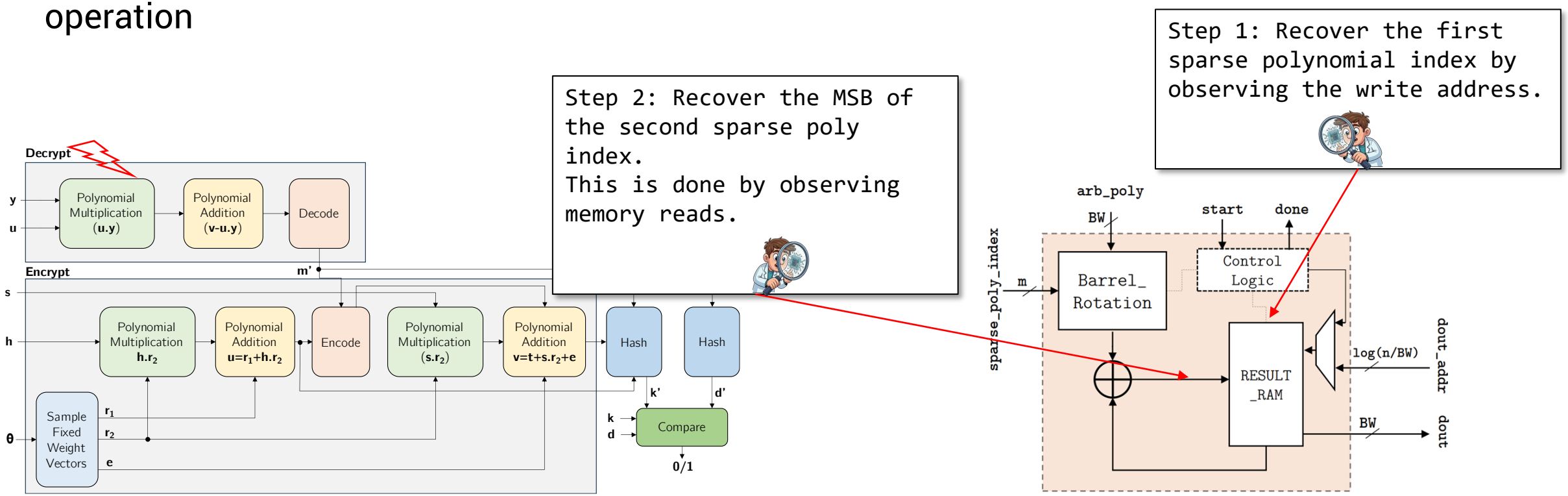
Interesting place to observe for side-channels

- Attacking the variable rotation in the sparse polynomial multiplication in the decapsulation operation

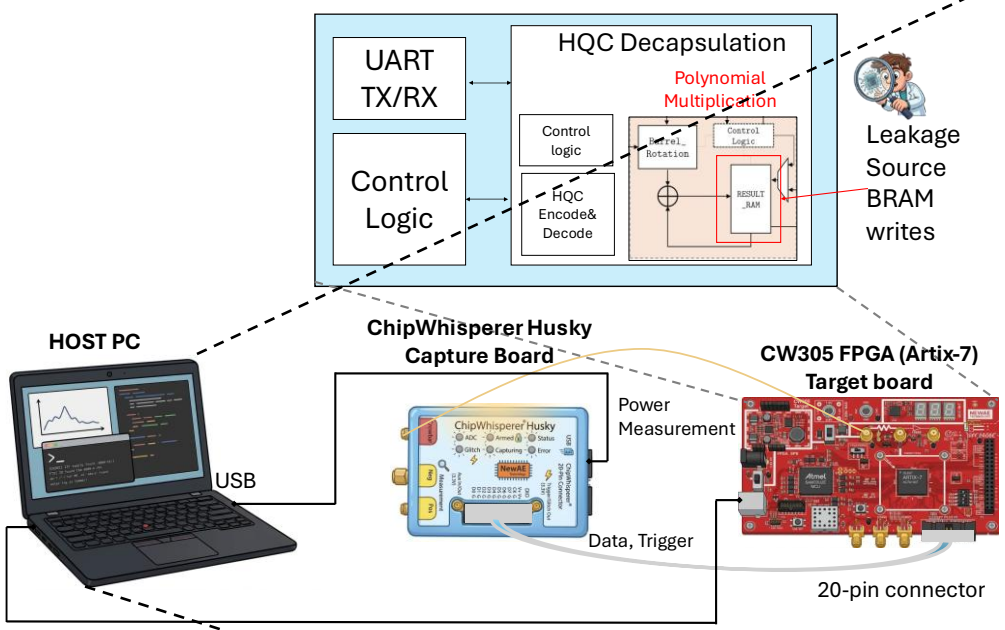


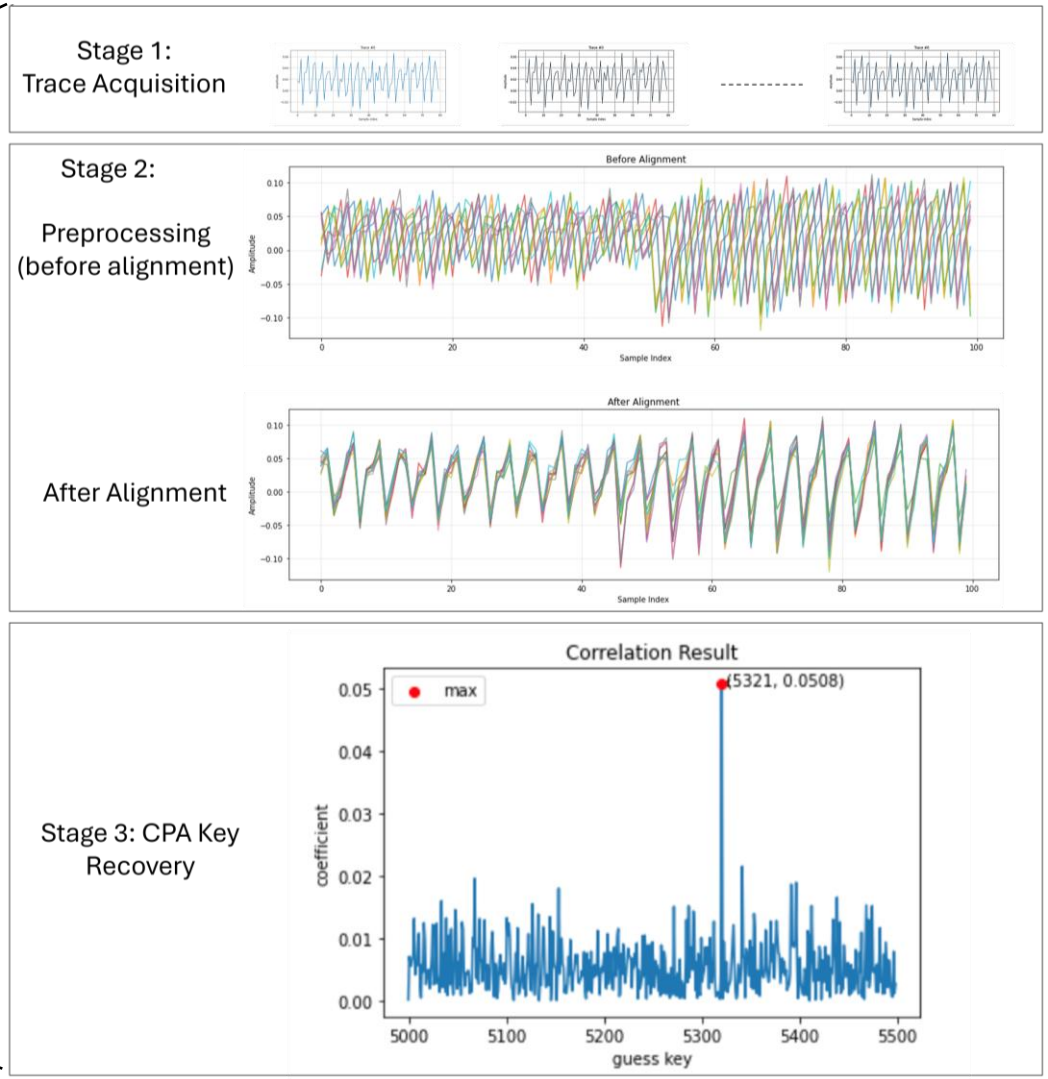
Power Side-Channel Attack on HQC Decapsulation

- Attacking the variable rotation in the sparse polynomial multiplication in the decapsulation operation



Attack Setup and Working

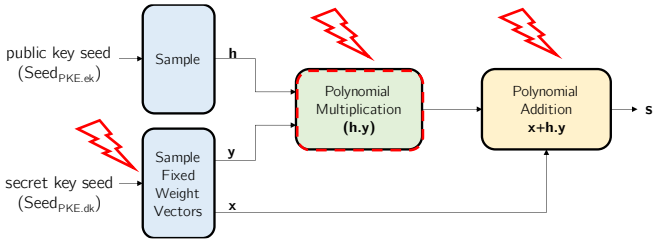




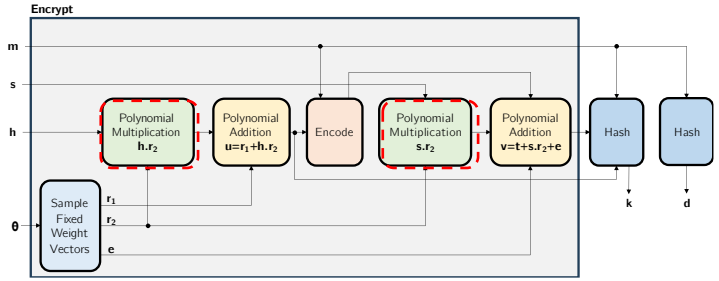
What can we do about it?

- Could be thwarted using robust implementation with **side-channel countermeasures**
- Countermeasures:
  - **Masking**
  - **Shuffling** sparse indices for each run of decapsulation
  - Random **shifting** for non-sparse polynomial before multiplication

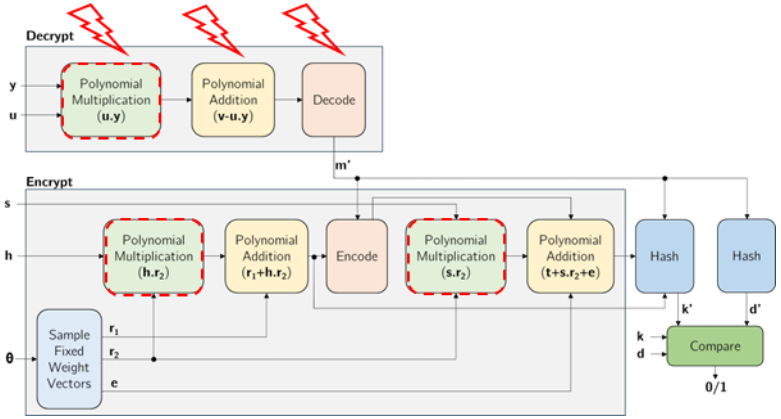
Summary



Key Generation



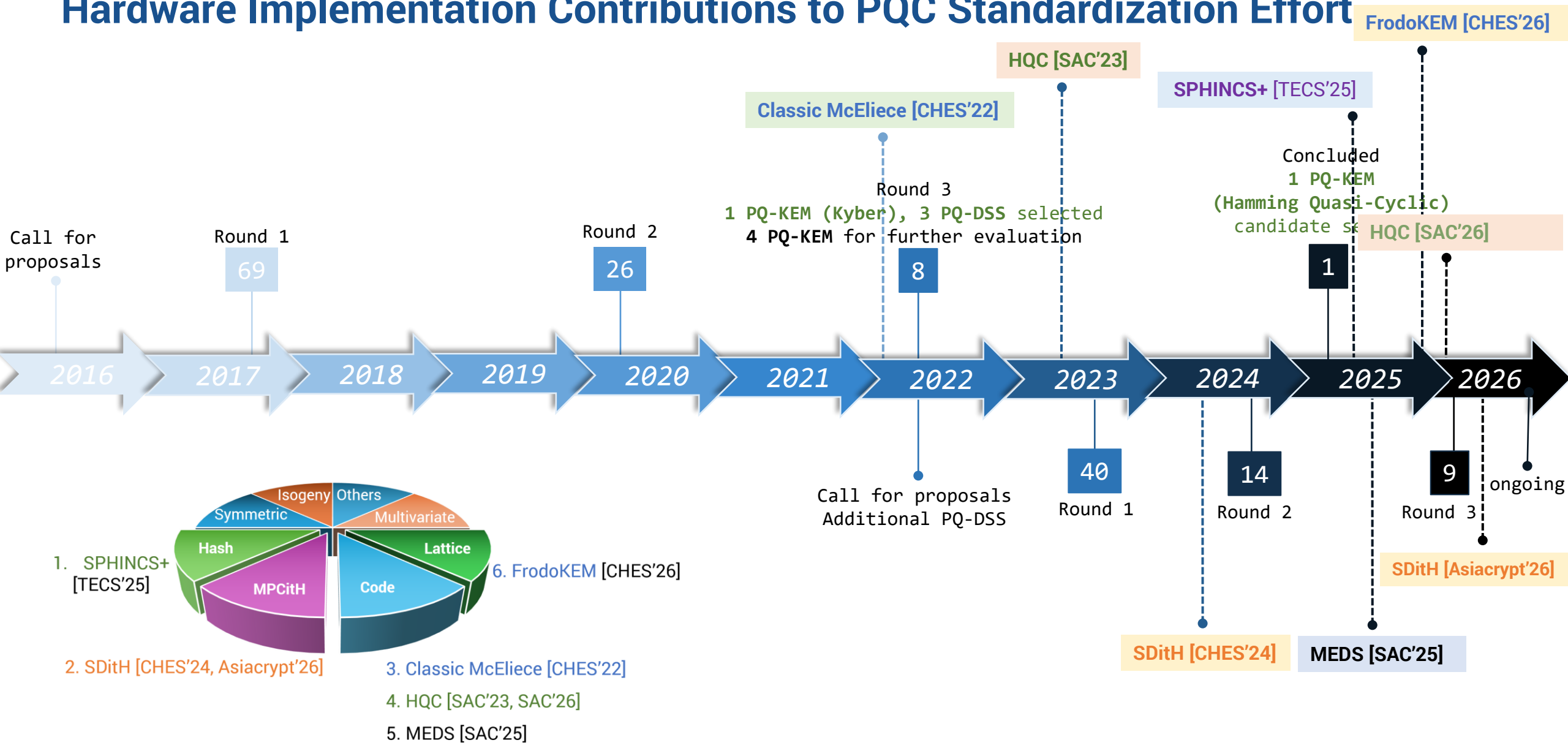
Encapsulation



Decapsulation

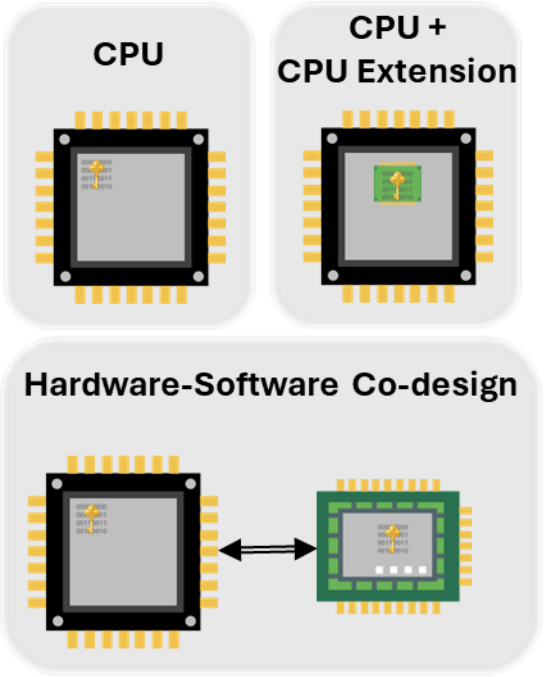
- Design of performance and security critical aspects of HQC implementation

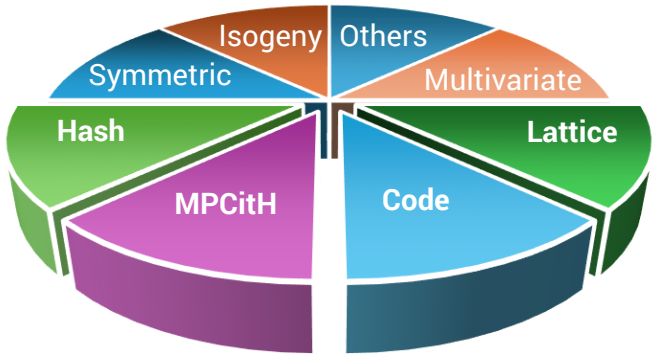
Hardware Implementation Contributions to PQC Standardization Effort

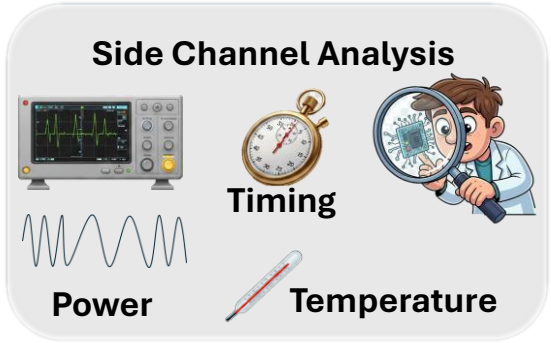


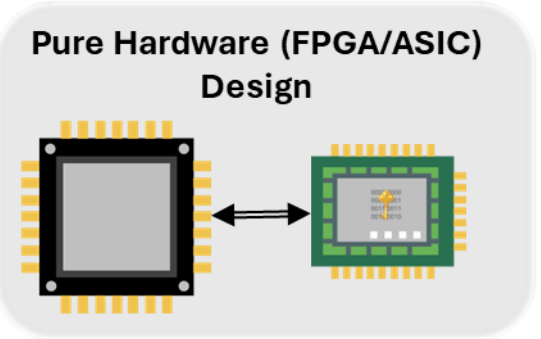
Future Research Directions – PQC

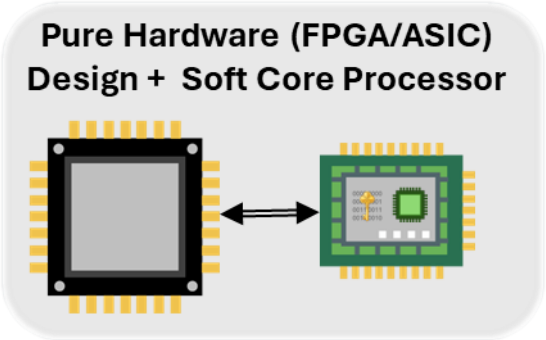
- Implementation and side-channel evaluation of PQC algorithms, proposing countermeasures, and developing robust implementations



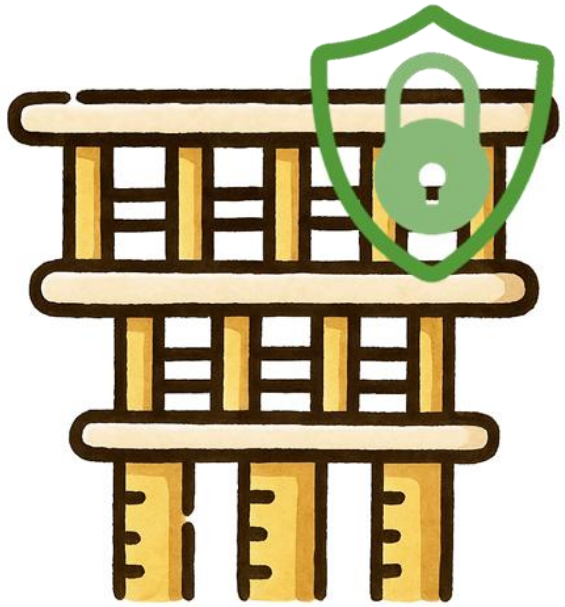






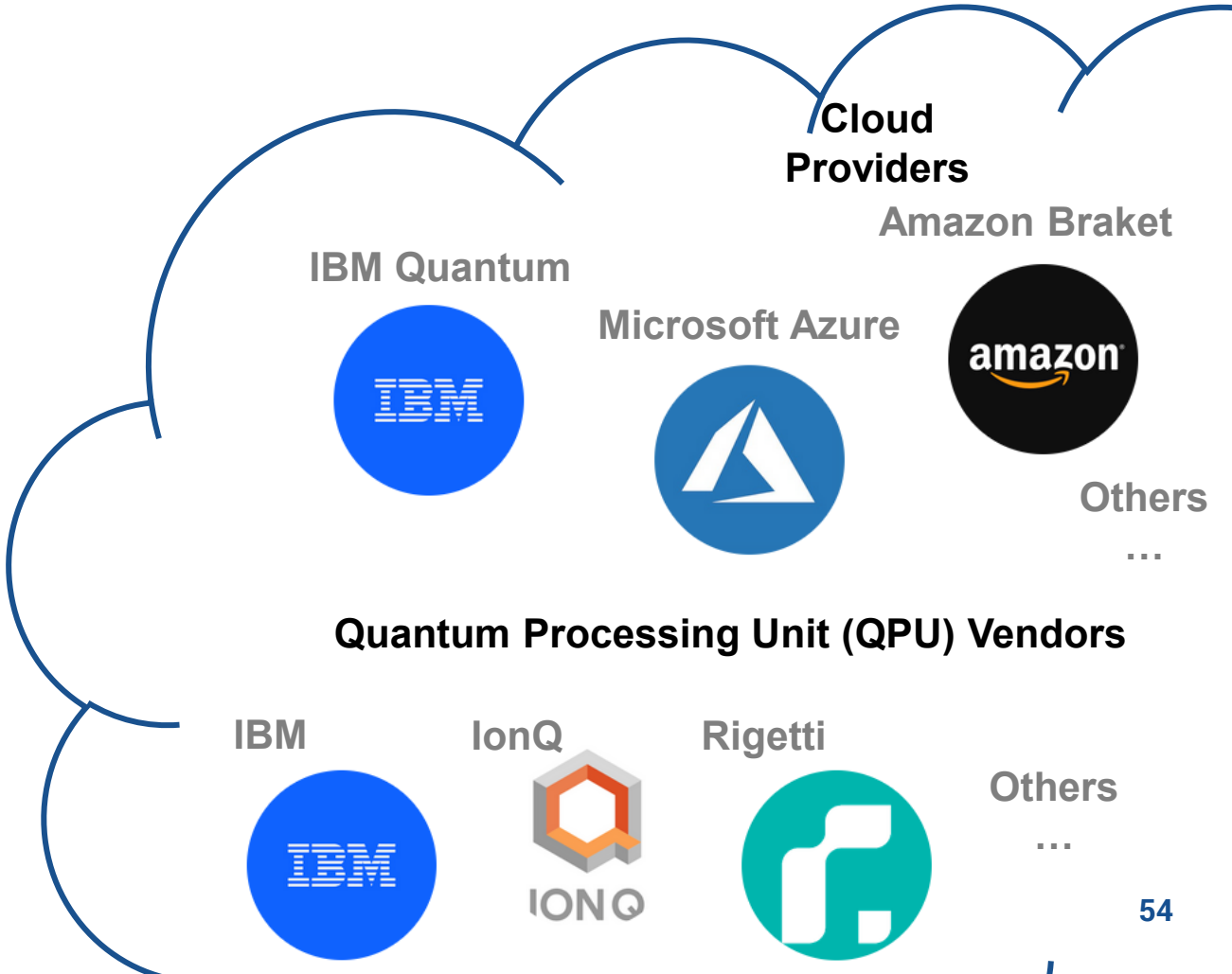


Quantum Computer Cybersecurity



Cloud-based Access to Quantum Computers

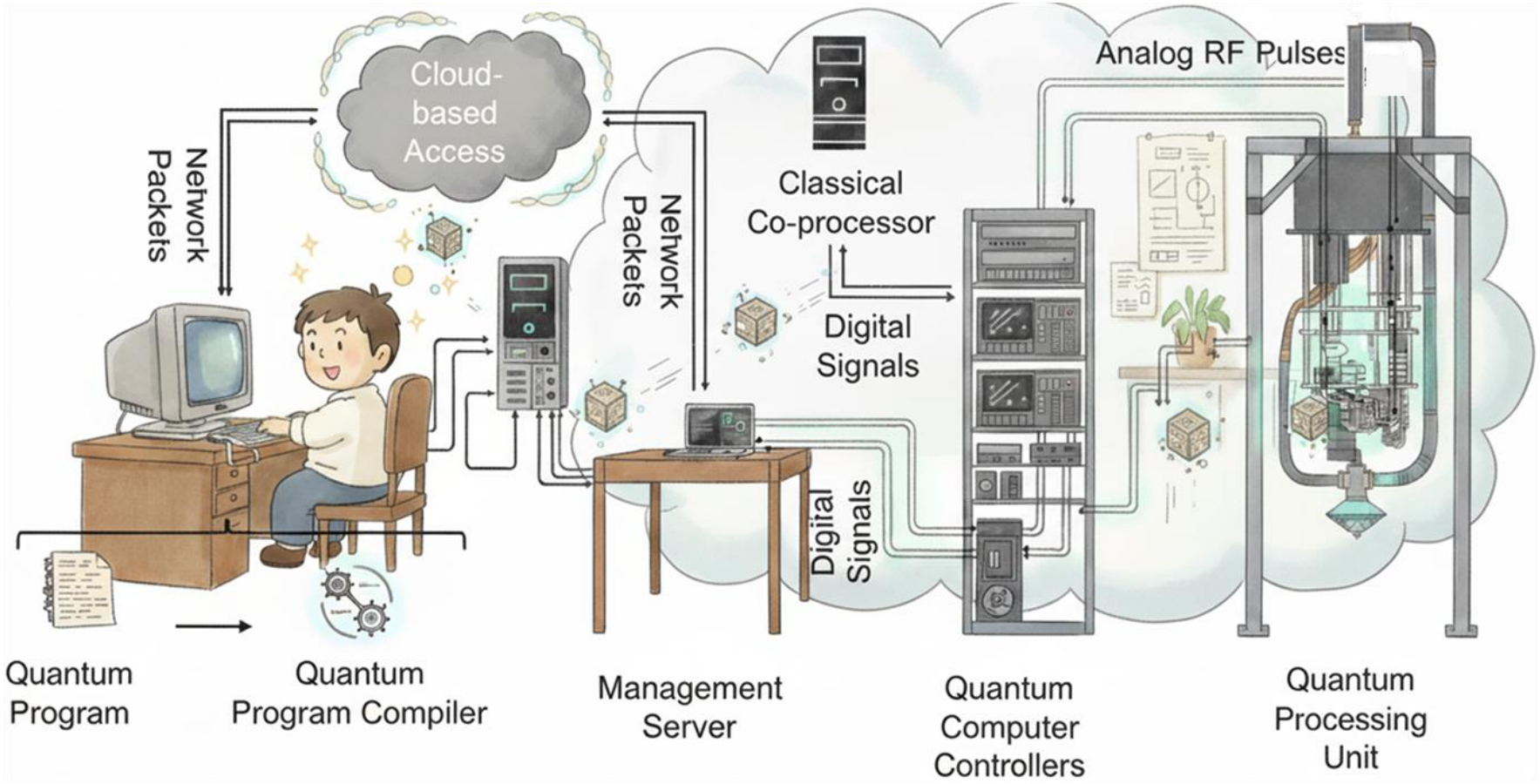
- Quantum Computers are rapidly being deployed as cloud-based accelerators today
  - Many companies are pushing for Quantum Computer as a Service (QCaaS) deployments
- But there are no security considerations in QCaaS today
  - Possibly malicious users can now access quantum computing hardware remotely
  - Can attack other users, or try to reverse engineer the infrastructure



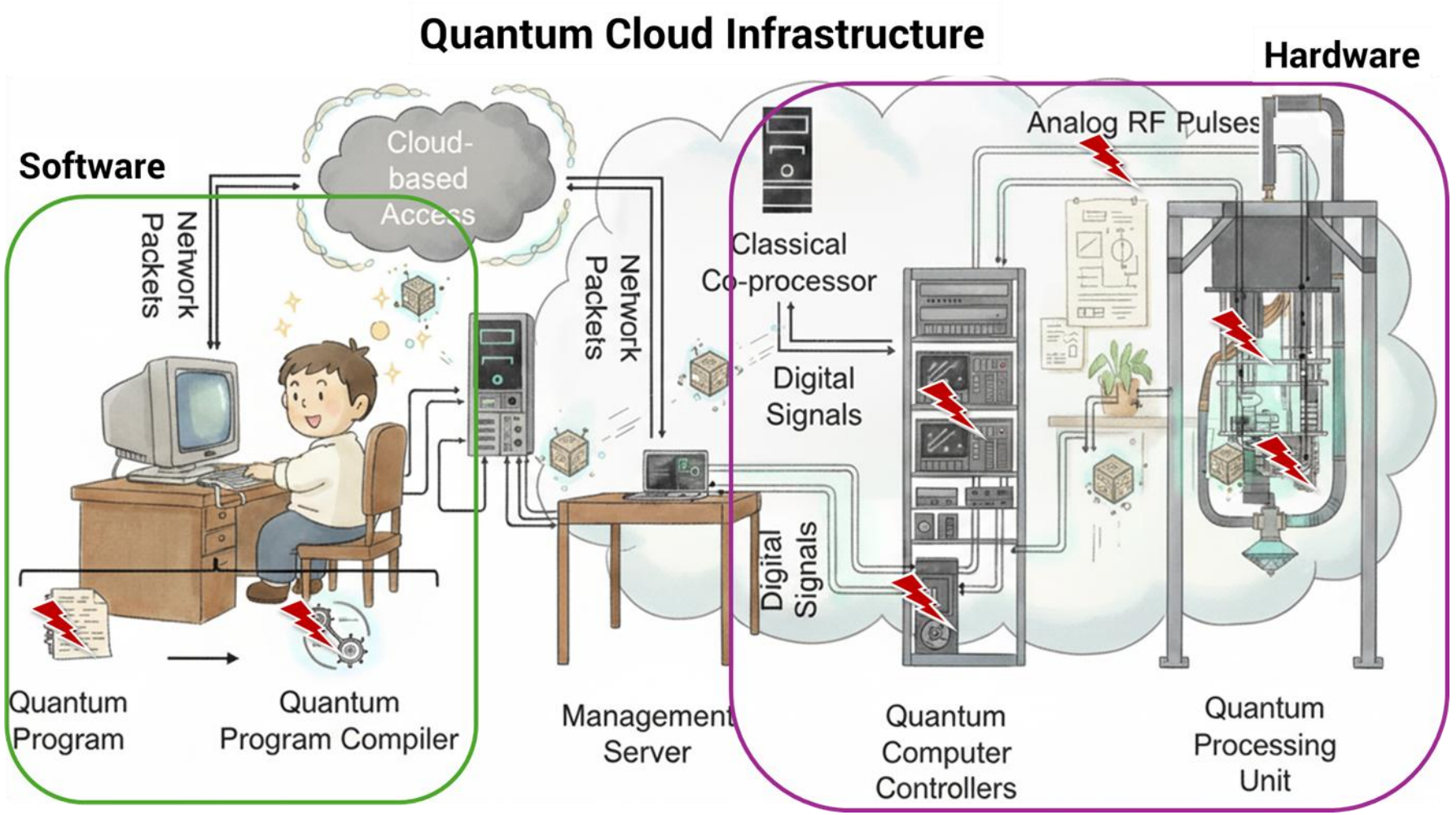
[Szefer'25]

Workflow of Cloud-Based Quantum Computers

Quantum Cloud Infrastructure

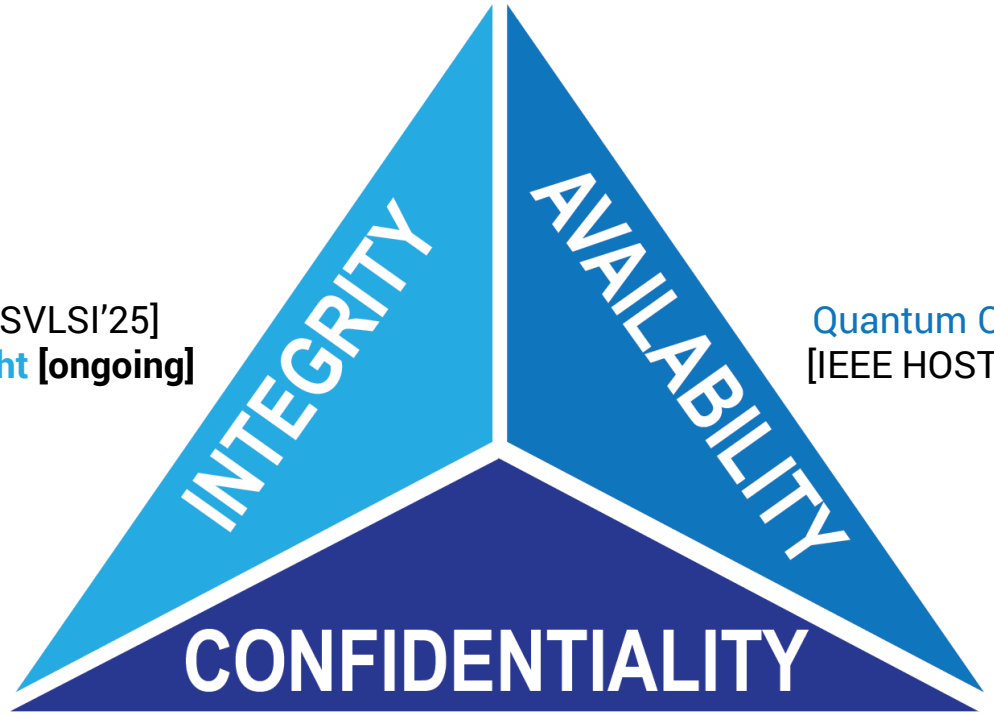


Potential Threat Landscape



The CIA Triad

- CHEQ [GLSVLSI'25]
- **DAC Knight [ongoing]**

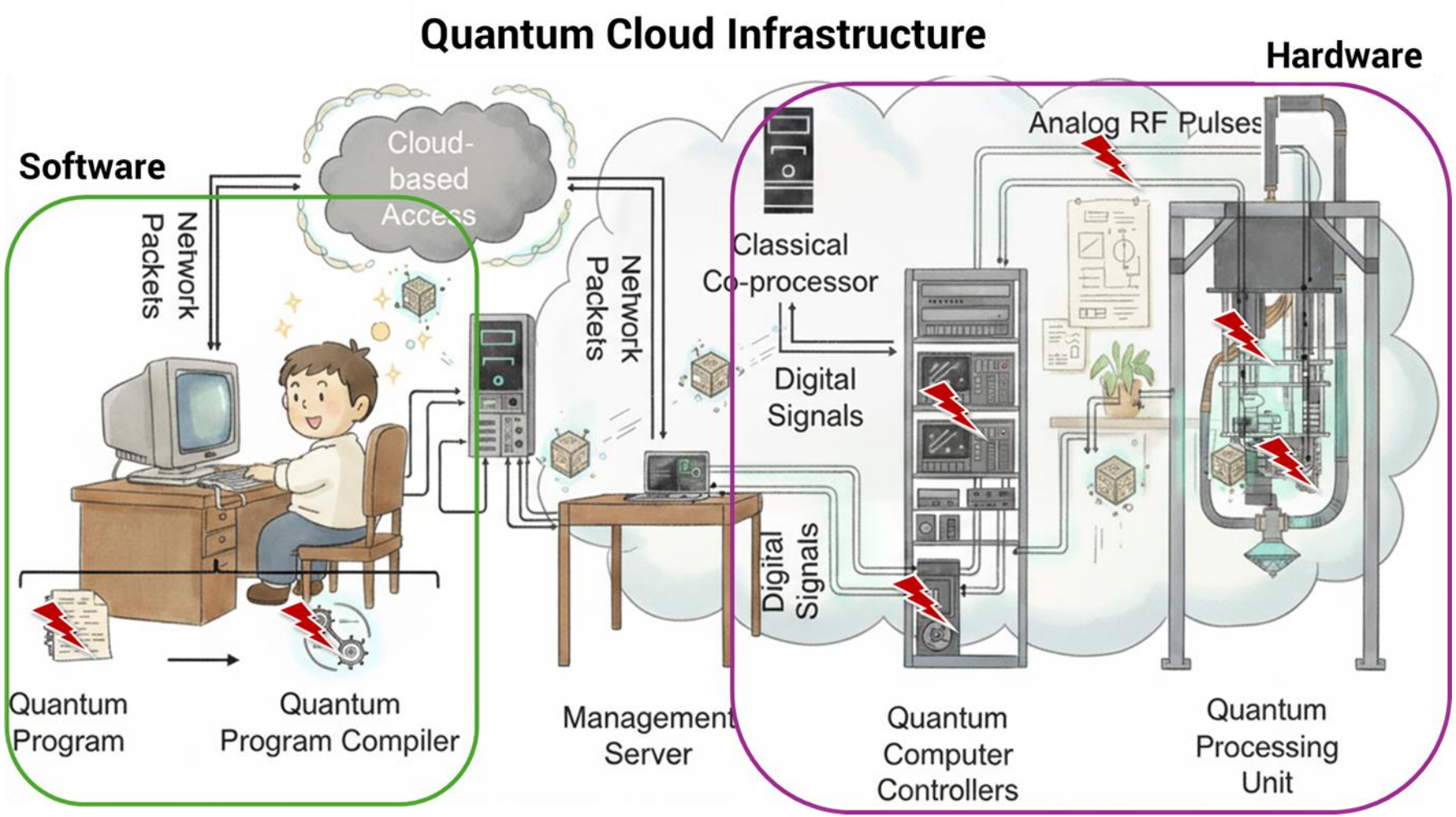


Quantum Computer Antivirus
[IEEE HOST'22, IEEE HOST'23]

[NIST'22]

- A Quantum Computer Trusted Execution environment [IEEE CAL'23] ← *Best of CAL*
- Dynamic Pulse Switching for Protection of Quantum Computation on Untrusted Clouds [IEEE HOST'24]
- Protecting quantum computers with a trusted controller [IEEE QCE'24]
- Trusted execution environments for quantum computers [Front. Comp. Sci'25]
- SoteriaQ [IEEE QCE'26]

Some of Our Work on Quantum Computer Cybersecurity (QCC)



Quantum Computer Antivirus
[IEEE HOST'22, IEEE HOST'23]

- **CHEQ** **[GLSVLSI'25]**
- **DAC Knight** **[ongoing]**

A Quantum Computer Trusted Execution environment **[IEEE CAL'23]**

Dynamic Pulse Switching for Protection of Quantum Computation on Untrusted Clouds **[IEEE HOST'24]**

Protecting quantum computers with a trusted controller **[IEEE QCE'24]**

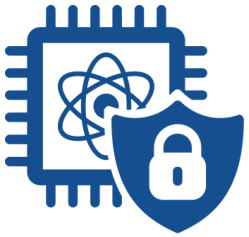
Trusted execution environments for quantum computers **[Front. Comp. Sci'25]**

QCCS 2026

4th Annual Quantum Computer Cybersecurity Symposium

To be held November 11 to 13, 2026 at The Guild Lounge in Scott Hall at Northwestern University

https://caslab.io/events/qccs/



Thank you!

https://sanjaydeshpande.phd/