Fast and Efficient Hardware Implementation of HQC

Sanjay Deshpande[*], Chuanqi Xu[*], Mamuri Nawan[+], Kashif Nawaz[+], and Jakub Szefer[*]

[*]Computer Architecture and Security Lab, Yale University

[+]Cryptography Research Centre, Technology Innovation Institute

Selected Areas in Cryptography
2023

caslab.io

Outline

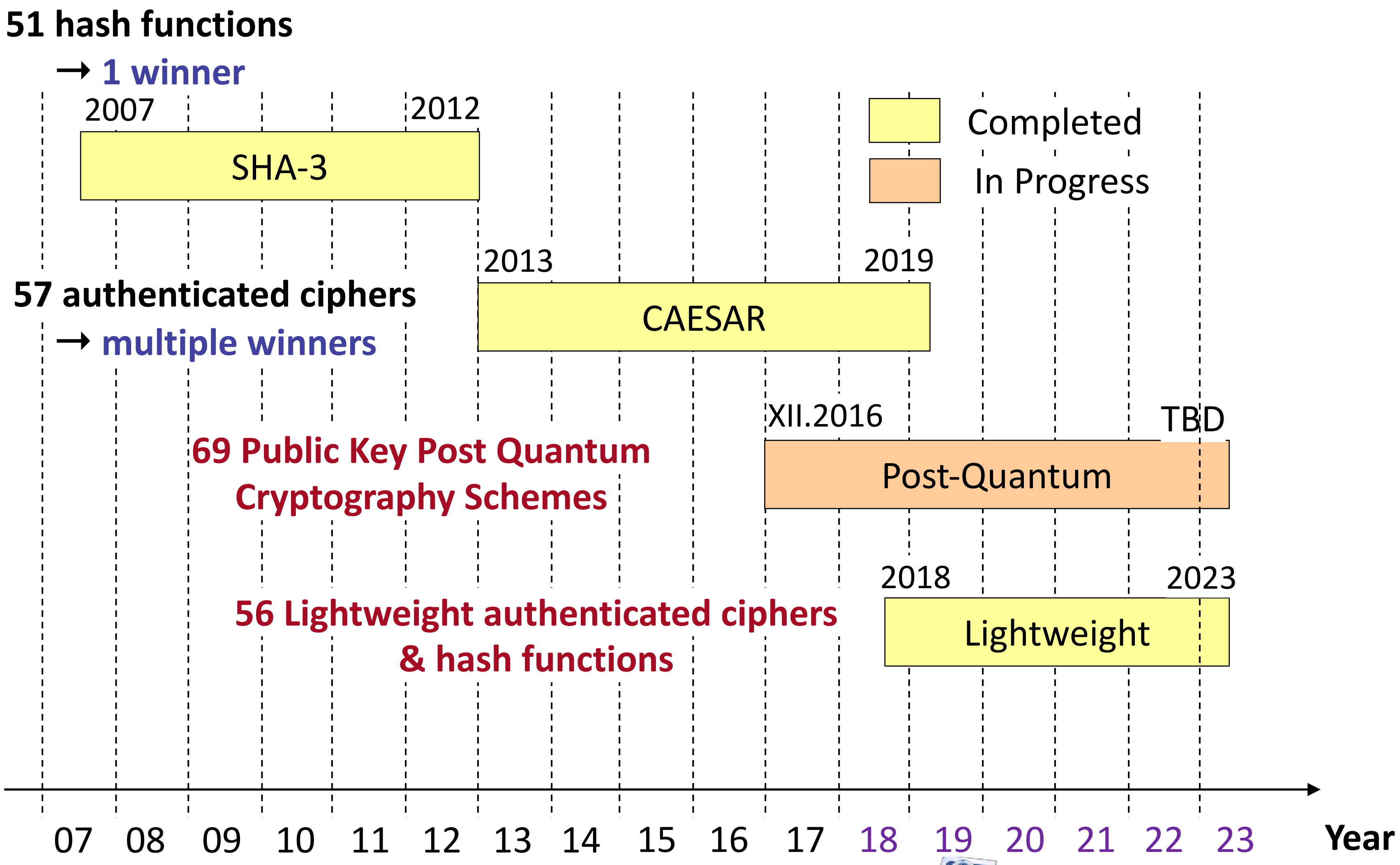
- Motivation
- Introduction
- Goal and Existing work
- Components in HQC and their Hardware Implementation
- Comparison with Related State of the Art

Motivation

- Quantum Computers with sufficient Qubits will be able to solve practical problems.
- Two algorithms are of special interest for cryptography community:
  - Shor’s Algorithm
  - Grover’s Search Algorithm
- For example, to break a 2048-bit RSA, a perfect quantum computer with 4099 ideal qubits can do it in 10 seconds.
- Most recently, IBM announced 433-qubits quantum computer “Osprey”.
- There is a need for Quantum-safe Cryptography!

Image Source: MIT Technology Review
Source: https://www.quintessencelabs.com/blog/breaking-rsa-encryption-update-state-art/
Source: https://newsroom.ibm.com/2022-11-09-IBM-Unveils-400-Qubit-Plus-Quantum-Processor-and-Next-Generation-IBM-Quantum-System-Two

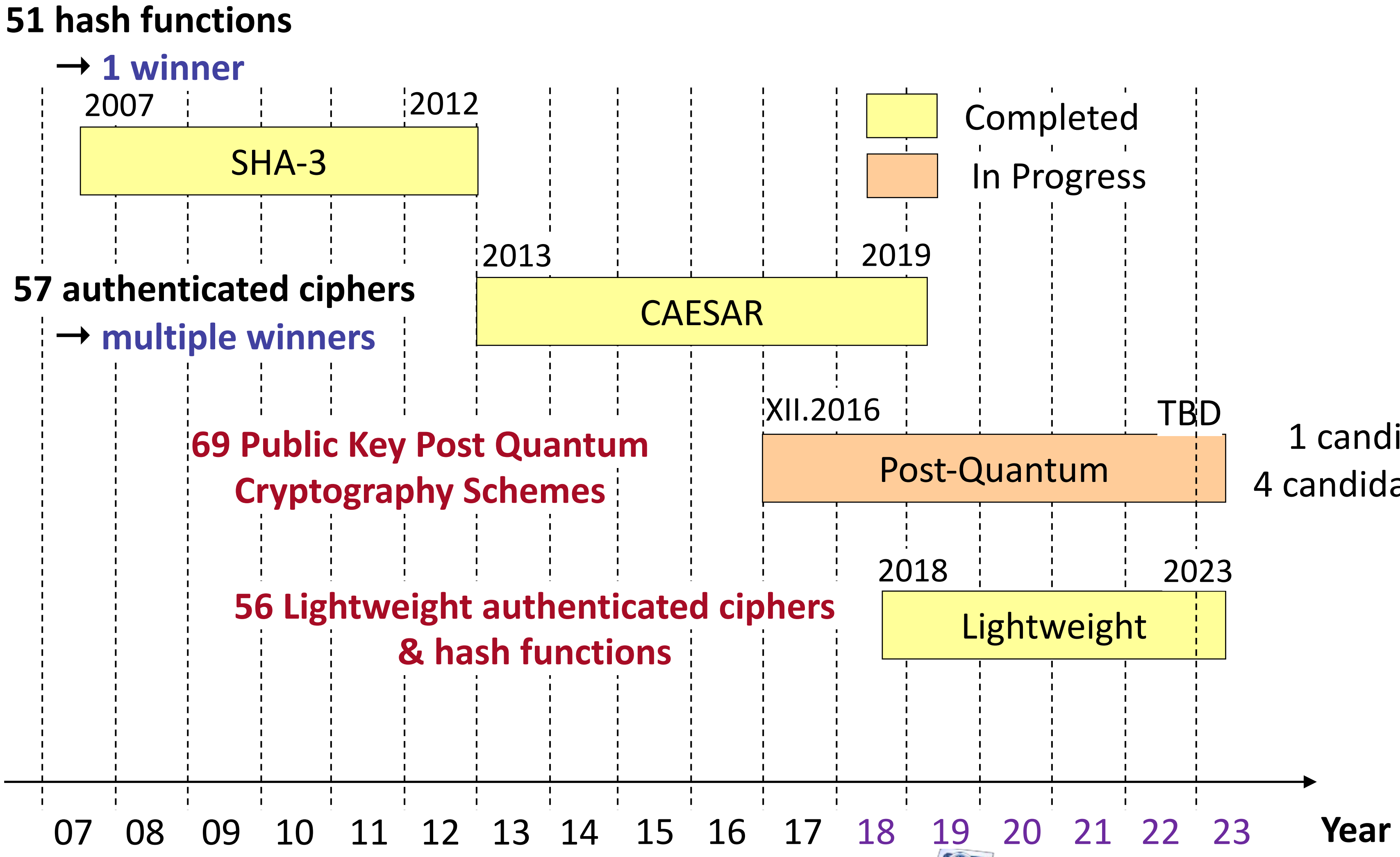
Post Quantum Cryptography Competition



Kris Gaj, Implementation and Benchmarking of Round 2 Candidates in the NIST Post-Quantum Cryptography Standardization Process Using FPGAs, NIST Seminars, Oct 2020.

caslab.io

Post Quantum Cryptography Competition

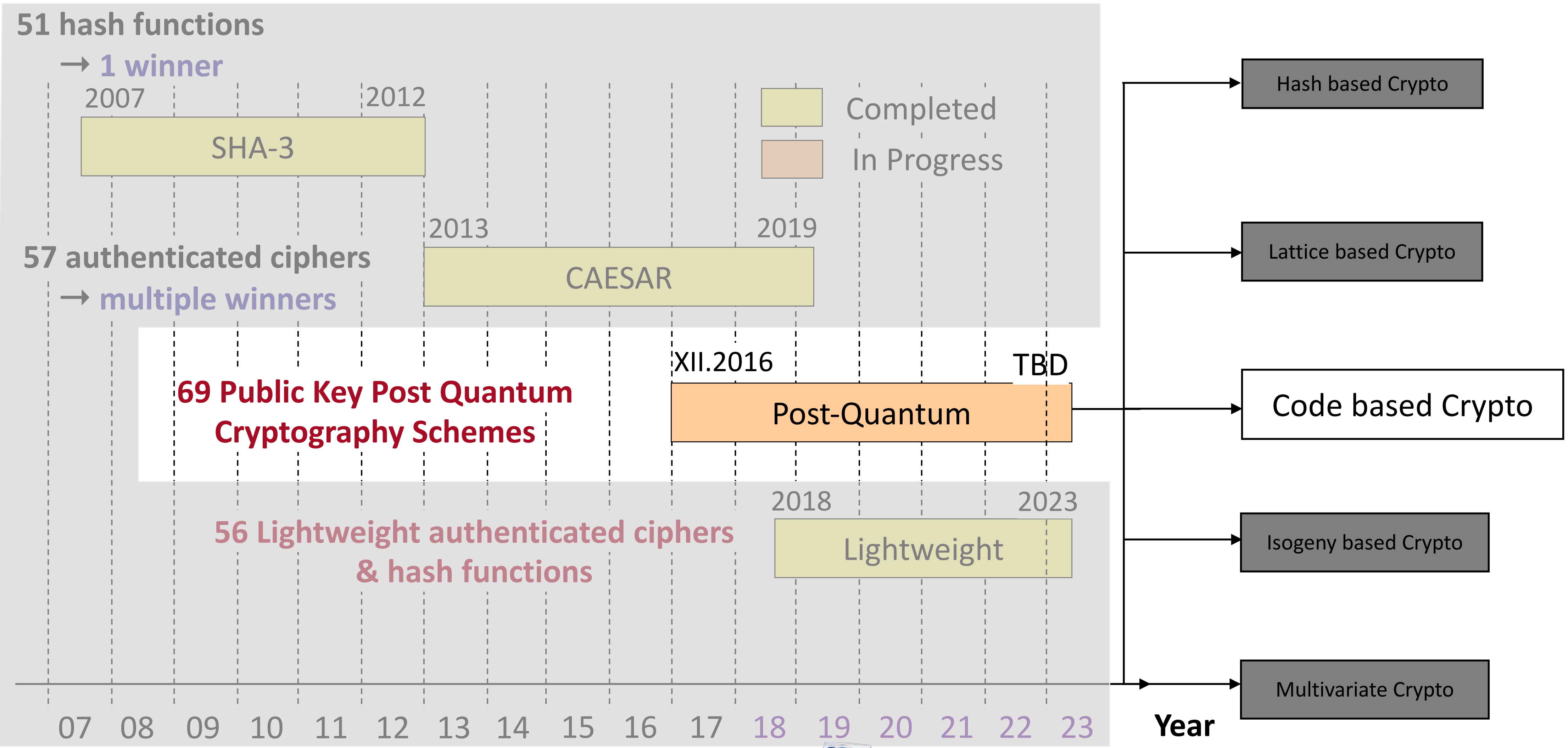


Round 4 announced
1 candidate selected for standardization
4 candidates will undergo further evaluation

caslab.io

Kris Gaj, Implementation and Benchmarking of Round 2 Candidates in the NIST Post-Quantum Cryptography Standardization Process Using FPGAs, NIST Seminars, Oct 2020.

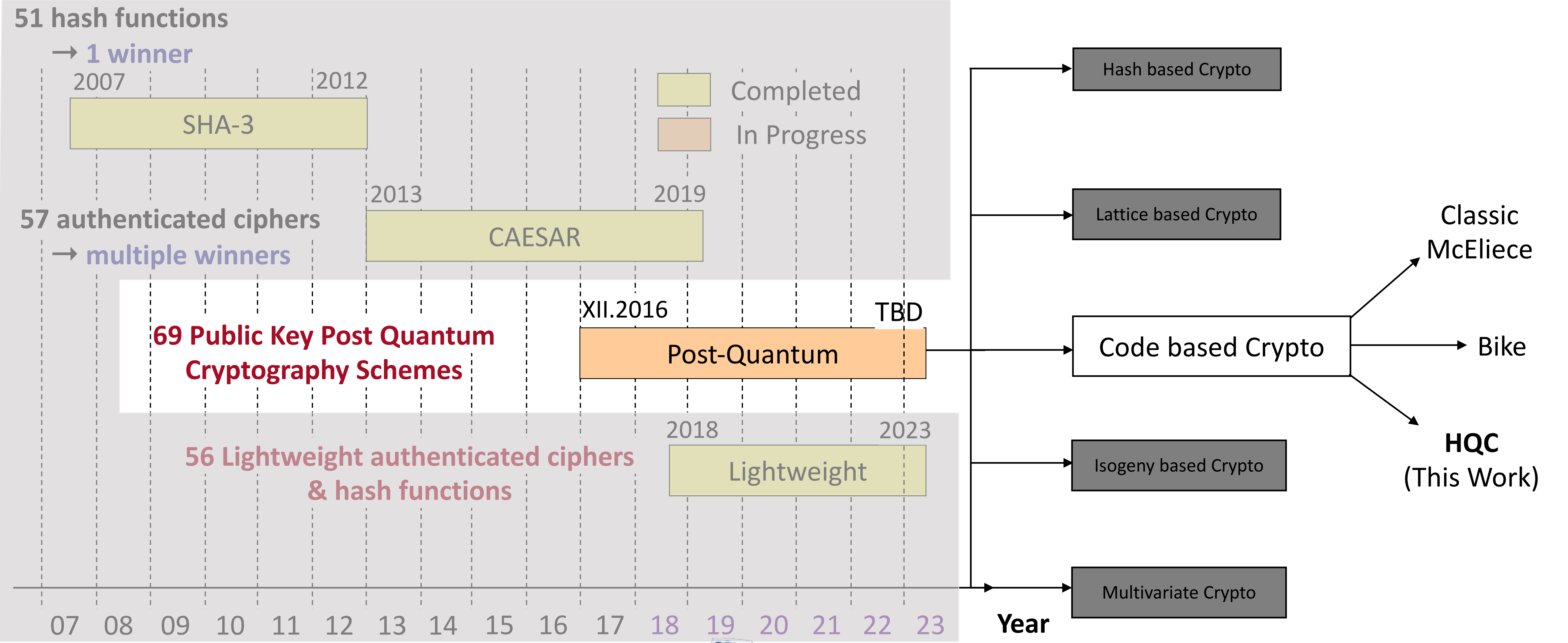
Post Quantum Cryptography Competition



Kris Gaj, Implementation and Benchmarking of Round 2 Candidates in the NIST Post-Quantum Cryptography Standardization Process Using FPGAs, NIST Seminars, Oct 2020.

caslab.io

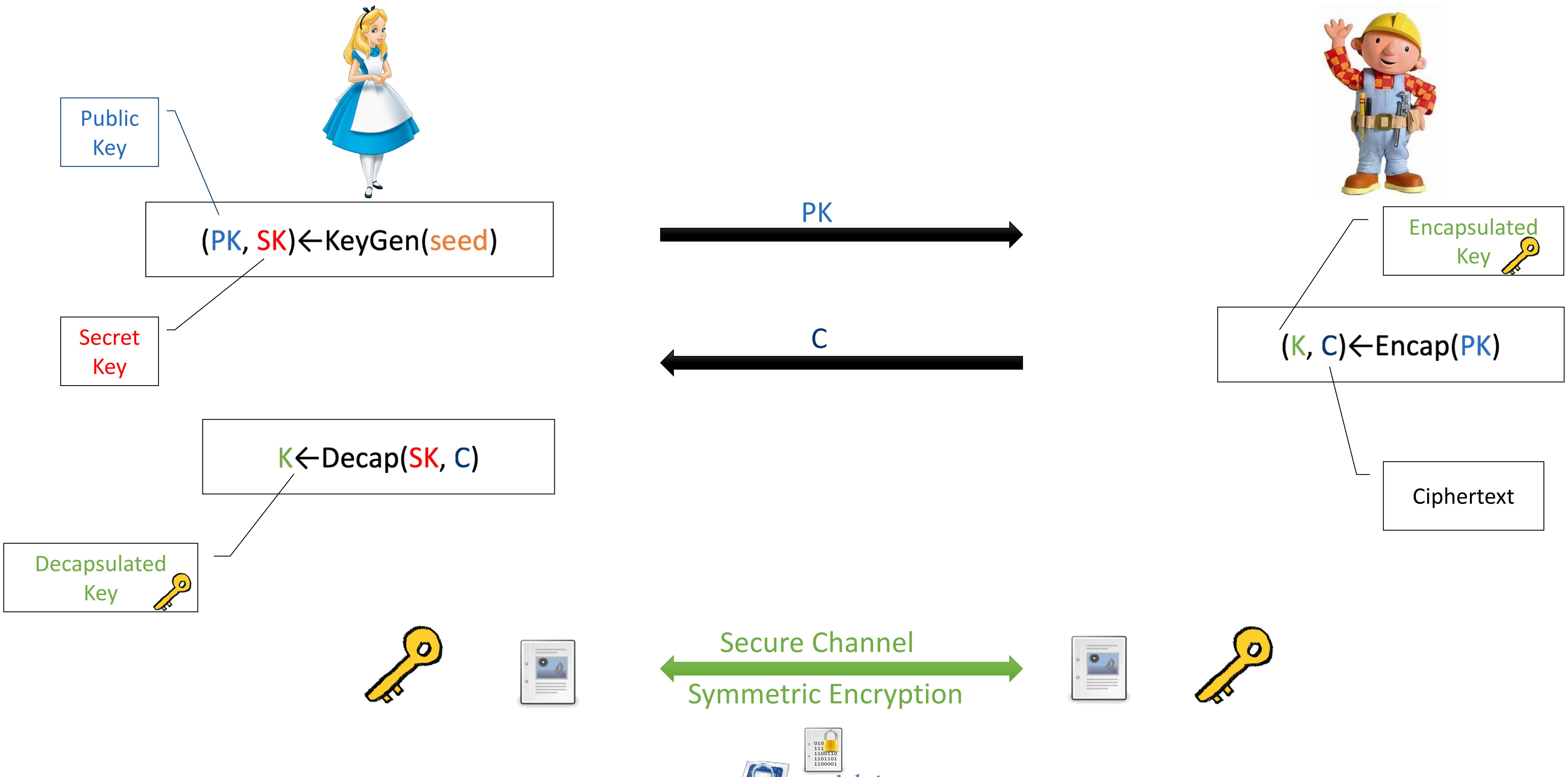
Post Quantum Cryptography Competition



Kris Gaj, Implementation and Benchmarking of Round 2 Candidates in the NIST Post-Quantum Cryptography Standardization Process Using FPGAs, NIST Seminars, Oct 2020.

caslab.io

Key Encapsulation Mechanism



caslab.io

Source: https://iacr.org/cryptodb/data/paper.php?pubkey=31347

Design & Experimental Setup

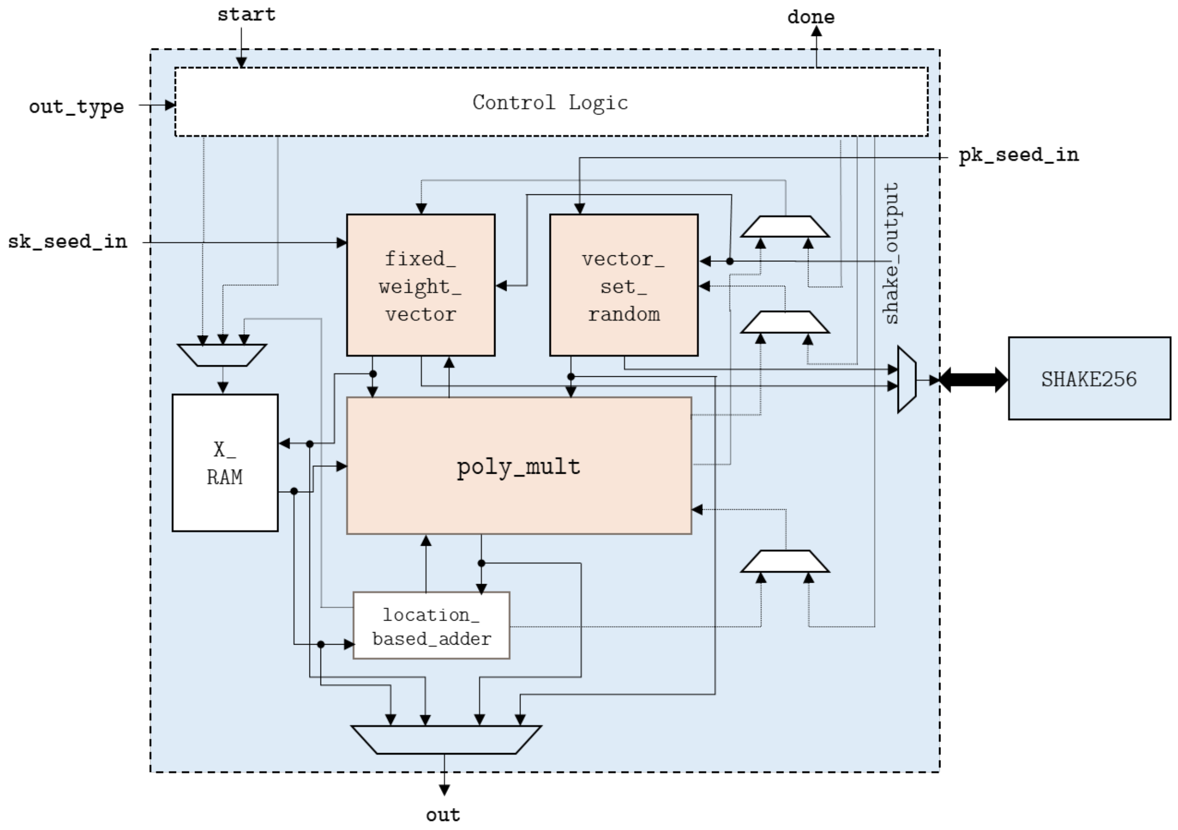
- HQC KEM parameter sets recommended as per the specification:
  - **hqc128**
  - **hqc192**
  - **hqc256**
- Target device: Xilinx - Artix 7 – xc7a200t-3 and xc7a100t-3
- Tool: Xilinx Vivado 2020.2.2
- Verification: Using the Software Reference implementation by HQC team [AAB+20]
- Goal: Implement a constant-time hardware design which is parameterizable across
  - Security level
  - Performance parameters

Image Source: Xilinx

Primitives

1. Key Generation

- Fixed weight error vector generation
  - SHAKE256
- Polynomial Multiplication
- Polynomial Addition

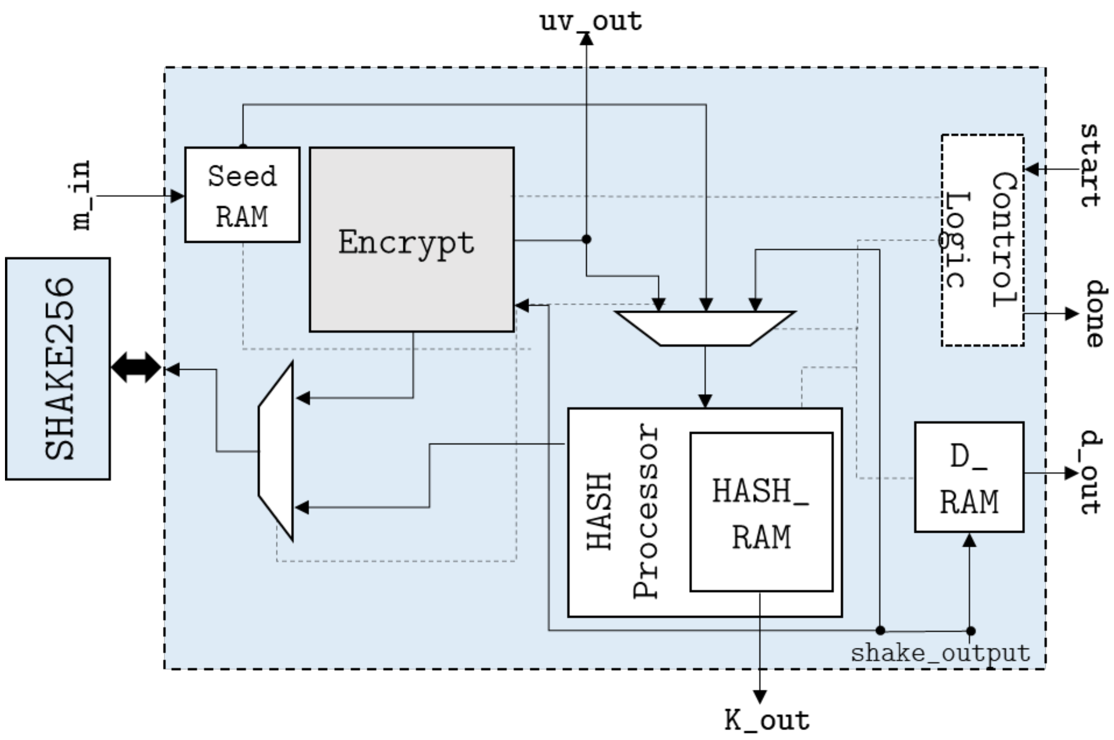


caslab.io

Primitives

2. **Encapsulation**

- Encrypt
  - Encode
  - Polynomial Multiplication
  - Polynomial Addition
  - Fixed weight error vector generation
    - SHAKE256
- Hash Computation
  - SHAKE256

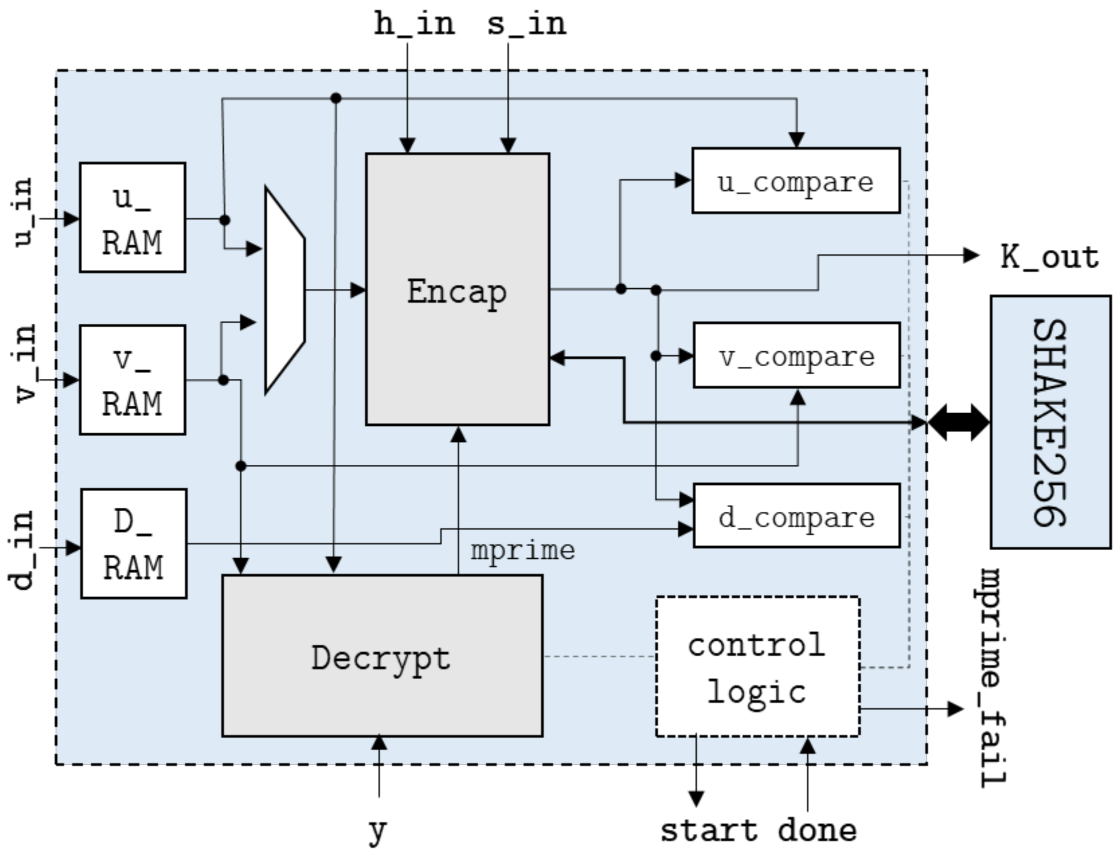


Primitives

3. Decapsulation

- Decrypt
  - Decode
  - Polynomial Subtraction
  - Polynomial Multiplication
- Encrypt
- Hash Computation
  - SHAKE256

} Encapsulation



Primitives – Common Modules

1. **Key Generation**
   - Fixed weight error vector generation
     - SHAKE256
   - Polynomial Multiplication
   - Polynomial Addition

2. **Encapsulation**
   - Encrypt
     - Encode
     - Polynomial Multiplication
     - Polynomial Addition
     - Fixed weight error vector generation
       - SHAKE256
   - Hash Computation
     - SHAKE256

3. Decapsulation
   - Decrypt
     - Decode
     - Polynomial Subtraction
     - Polynomial Multiplication
   - Encrypt
   - Hash Computation
     - SHAKE256

Primitives - Existing Work

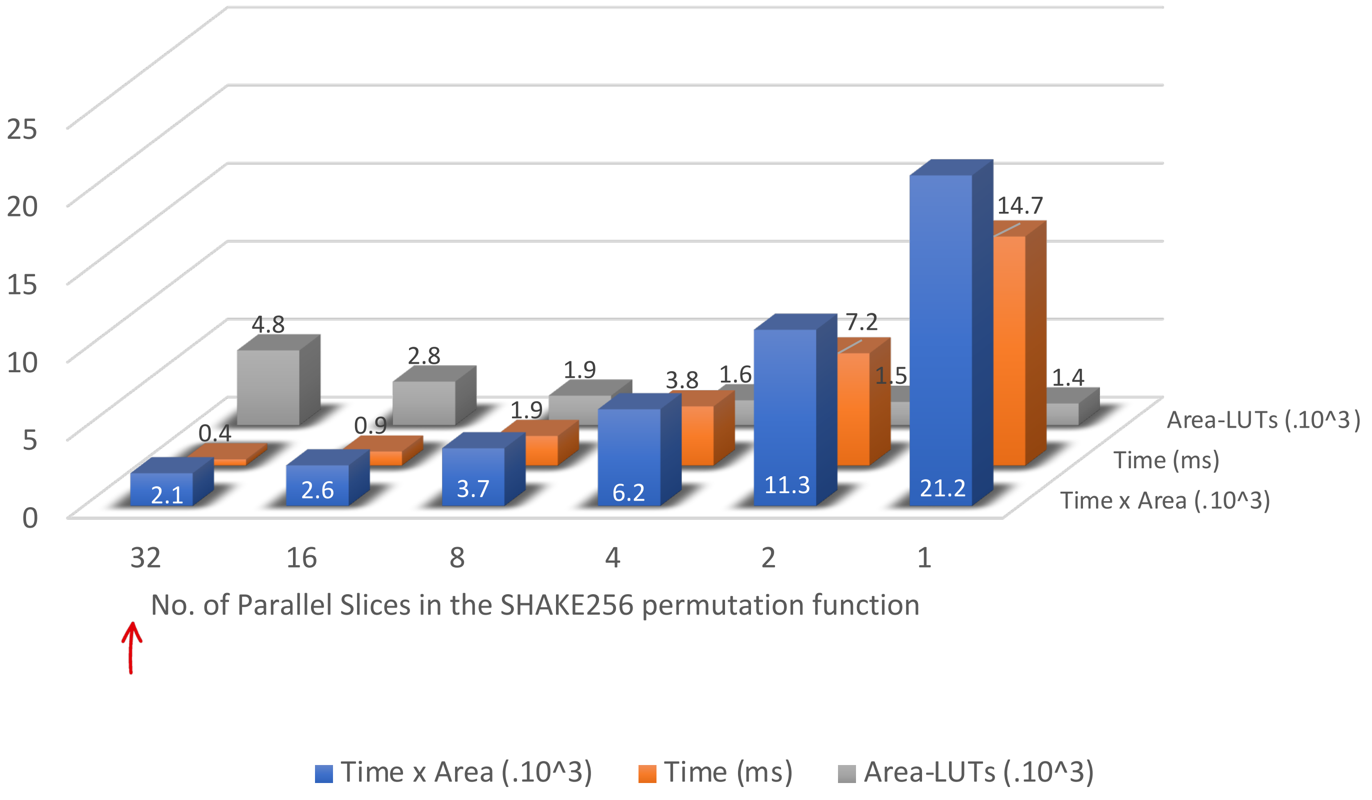
1. **Key Generation**
   - Fixed weight error vector generation
     - SHAKE256 [CCD+22]
   - Polynomial Multiplication
   - Polynomial Addition

2. **Encapsulation**
   - Encrypt
     - Encode
     - Polynomial Multiplication
     - Polynomial Addition
     - Fixed weight error vector generation
       - SHAKE256 [CCD+22]
   - Hash Computation
     - SHAKE256 [CCD+22]

3. **Decapsulation**
   - Decrypt
     - Decode
     - Polynomial Subtraction
     - Polynomial Multiplication
   - Encapsulation

SHAKE256

- Improved existing design [CCD+22].
  - Added state preserving capability.
  - Added an extra mode in the performance parameter (Parallel Slices).



caslab.io

Hardware Implementation of HQC KEM

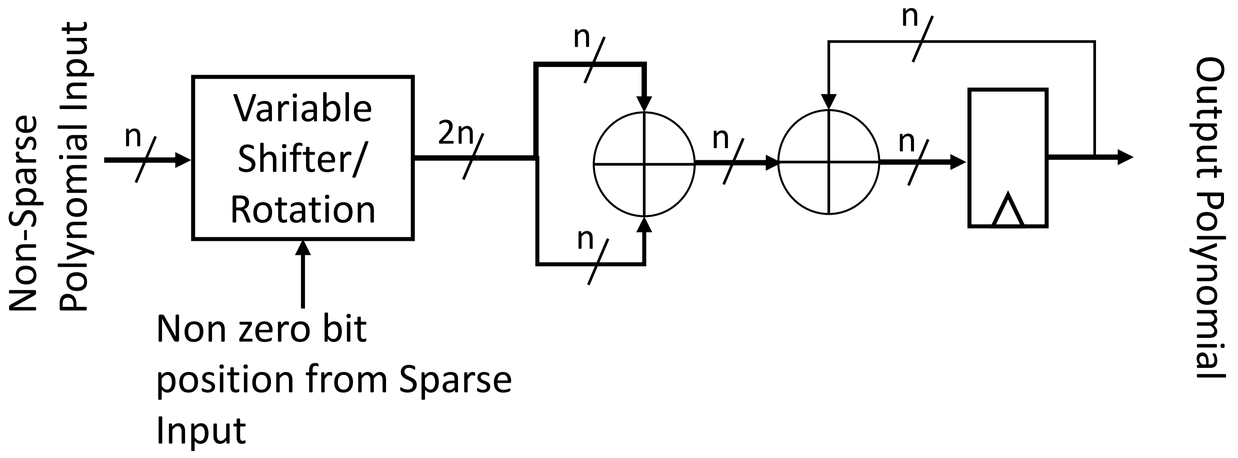
1. **Key Generation**
   - Fixed weight error vector generation
     - SHAKE256
   - Polynomial Multiplication
   - Polynomial Addition

2. **Encapsulation**
   - Encrypt
     - Encode
     - Polynomial Multiplication
     - Polynomial Addition
     - Fixed weight error vector generation
       - SHAKE256
   - Hash Computation
     - SHAKE256

3. **Decapsulation**
   - Decrypt
     - Decode
     - Polynomial Subtraction
     - Polynomial Multiplication
   - Encapsulation

Polynomial Multiplication - Sparse Multiplication with Interleaved Reduction

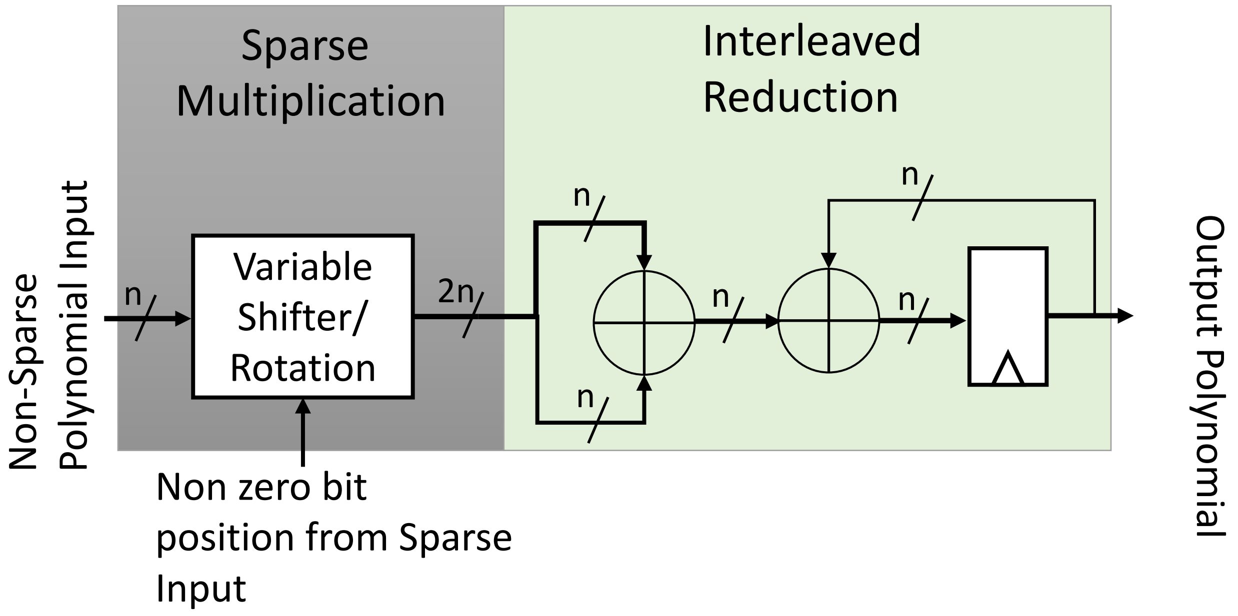
- One of the inputs to polynomial multiplier is a sparse fixed weight vector.
- Indices of non-zero elements are used to shift non-sparse polynomial to imitate the multiplication.
- The multiplication and the modular reduction is interleaved.



*Conceptual Block Diagram

Polynomial Multiplication - Sparse Multiplication with Interleaved Reduction

- One of the inputs to polynomial multiplier is a sparse fixed weight vector.
- Indices of non-zero elements are used to shift non-sparse polynomial to imitate the multiplication.
- The multiplication and the modular reduction is interleaved.



*Conceptual Block Diagram

Polynomial Multiplication - Results

BW (bits)	Logic (SLICES)	(LUT)	(DSP)	Memory (FF)	(BR)	F (MHz)	Cycles (cyc.)	Time (us)	T x A
Our poly_mult module, Polynomial Length⁺ = 12,323, $W^*_{SPARSE} = 71$									
32	134	396	0	181	1	270	27,621	0.10	14
64	202	599	0	205	2	277	13,918	0.05	10
128	486	1,438	0	456	4	238	7,102	0.03	14
General Sparse Multiplier, Polynomial Length⁺ = 12,323, $W^*_{SPARSE} = 71$ [19]									
32	132	319	0	127	2	234	27,691	0.12	16
64	197	549	0	190	4	222	13,988	0.06	12
128	378	1,136	0	381	8	185	7,172	0.04	15
Sparse Multiplier, Polynomial Length⁺ = 10,163, $W^*_{SPARSE} = 71$ [15]									
32	100		—	—	2	240	158,614	0.66	66
64	157		—	—	3	220	90,880	0.41	64
128	292		—	—	5	210	51,688	0.24	70

† = Slices (no info on LUTs), + Length of the non-sparse arbitrary polynomial, * = Weight of the sparse polynomial input

Target Device: Artix 7 – xc7a200t

caslab.io

Fixed Weight Vector Generation

- Generates a uniform random n-bit fixed-weight (w) vector.
- Algorithm given in [AAB+20].

Guo et. al [GHJ+22] demonstrated a timing attack on the software reference implementation

```
Input: N, w, seed
Output: w distinct elements in range 0 to N − 1
pos = []*w
i,j,k,l = 0
prng_init(seed)
while (i < w) do
        rand_bits ← prng_draw(outsize = 24)
        if (rand_bits < THRESHOLD_VALUE)
                pos [i] = (ran      N)
                i++
        end
end whil
```

Threshold Check

```
while (j < w) do
        while (k < w) d
                      s[j] == pos[k] then
                        while (l < 1) do
                          rand_bits ← prng_draw(outsize=24)
                          if (rand_bits < THRESHOLD) then
                        pos [k] = (rand_bits % N)
                        l++
                          end if
                        end while
                end
                else then
                        k++
                end
        end while
        j++
end while
return pos
```

Duplicate Detection

//Value for THRESHOLD is a constant based on the parameter set

caslab.io

Fixed Weight Vector – Constant Weight Word (CWW) Method

- Constant time method proposed by Sendrier [SEN21].
- Recommended by HQC Team as a 4th round update.
- Barrett Reduction used in Index Generation.
- However, this method has small bias.

```
Input: N, w, seed
Output: w distinct elements in range 0 to N − 1
rand bits ← prng(input = seed, output size = 32 × w)
```
RNG

```
for i ← w − 1 to 0 do
    pos[i] = i + (rand bits[32 + 32 ∗ i − 1 : 32 ∗ i])%(N − i)
end for
```
Index Generation

```
for j ← w − 1 to 0 do
    duplicate found ← 0
    for k ← j + 1 to w − 1 do
        if pos[j] == pos[k] then
            duplicate found ← 1
        end if
    end for
    if duplicate found == 1 then
        pos[j] = j
    end
end for
```
Duplicate Detection and Index Replacement

```
return pos
```

caslab.io

Fixed Weight Vector Generation – Fast and Non-Biased(FNB) Method

- Improvement on the original algorithm proposed given in [AAB+20]

```
Input: N, w, seed, ACC_REJ
Output: w distinct elements in range 0 to N − 1
1: pos = []*(w + ACC_REJ)
2. i,j,k,m = 0
3. prng_init(seed)
4: while (i < (w+ ACC_REJ)) do
5:        rand_bits ← prng_draw(outsize = 24)          RNG
6:        if (rand_bits < THRESHOLD_VALUE) then
7:                pos [i] = (rand bits % N)
8:                i++
9:        end                                   Threshold Check
10: end while

//Value for THRESHOLD is a constant based on the parameter set
// ACC_REJ  = ACCEPTABLE_REJECTIONS
// e.g., ACC_REJ = 75
```

Small prob non-constant time behavior

```
11: while (j < w + ACC_REJ) do
12:        while (k < (w+ ACC_REJ)) do
13:          if (j < w) then
14:                if pos[j] == pos[k] then
15:                    if (m<ACC_REJ) then
16:                            pos[k] = pos[m+w]
17:                            m++
18:                    end
19:                    else
20:                        USE PRNG TO DRAW MORE RAND BITS AND
                           REDO THRESHOLD CHECK and ASSIGN TO pos[k]
21:                    end
22:                end
23                 else
24:                        DUMMY OPERATIONS
25                 end
26:          else
27:                DUMMY OPERATIONS
28:          end
29:          k++
30:        end while
31:        j++                                  Duplicate Detection
32: end while
33: return pos
```

Fixed Weight Vector Generation - Evaluation

Design	Weight (w_r)	Resources: Logic (LUT)	Resources: Logic (DSP)	Resources: Memory (FF)	Resources: Memory (BR)	F (MHz)	Cycles (cyc.)	Time (us)	T x A	Failure[+] Prob.
Fast and Non-Biased Design (ACCEPTABLE_REJECTIONS $= w_r$)										
hqc128	75	316	0	124	2.0	223	1,479	6.63	2.10	2.8×2^{-199}
hqc192	114	295	0	125	2.0	236	2,226	9.43	2.78	1.1×2^{-280}
hqc256	149	314	0	192	2.5	242	3,248	13.42	4.21	4.9×2^{-355}
Constant Weight Word (CWW)										
hqc128	75	201	4	229	1.0	201	3,062	15.23	3.06	0
hqc192	114	211	5	245	1.0	200	6,817	34.09	7.19	0
hqc256	149	216	5	248	1.0	204	11,487	56.31	1216	0

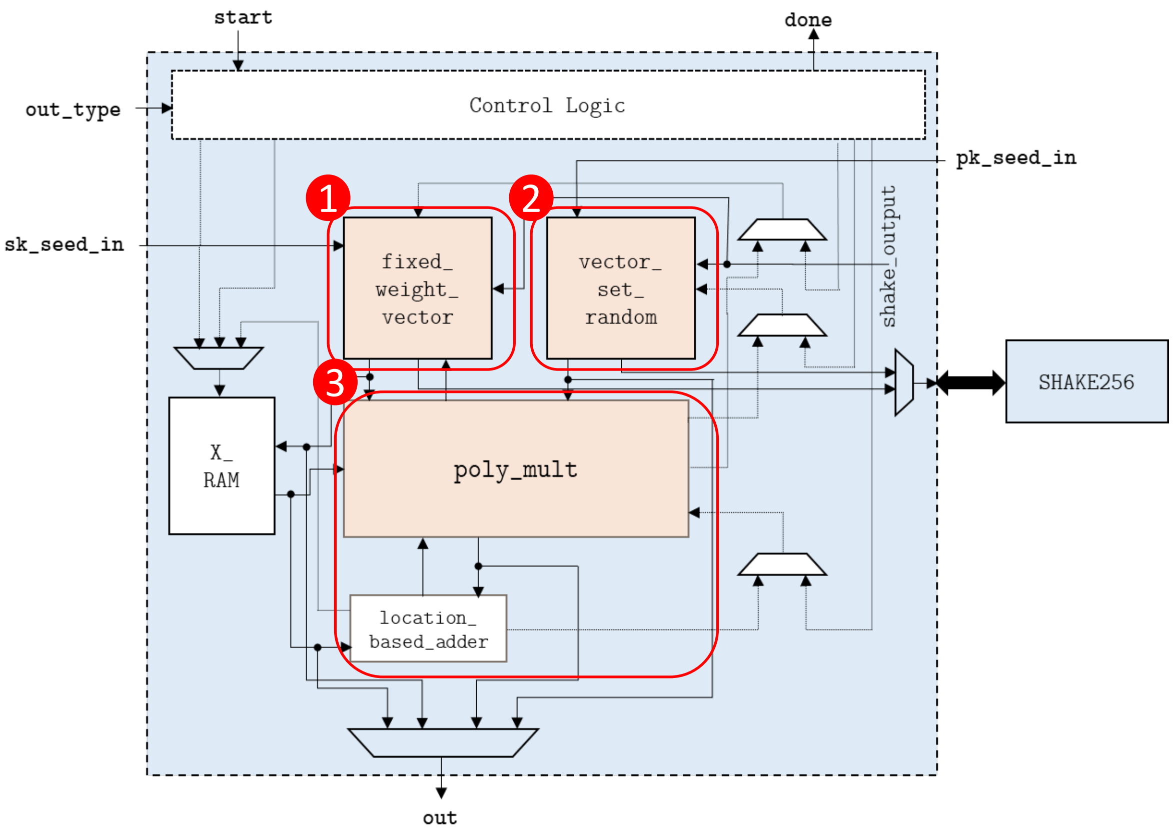
[+] = Probability of our design failing to behave constant-time.

Key Generation - Algorithm and Hardware Design

Inputs: pk_seed, sk_seed

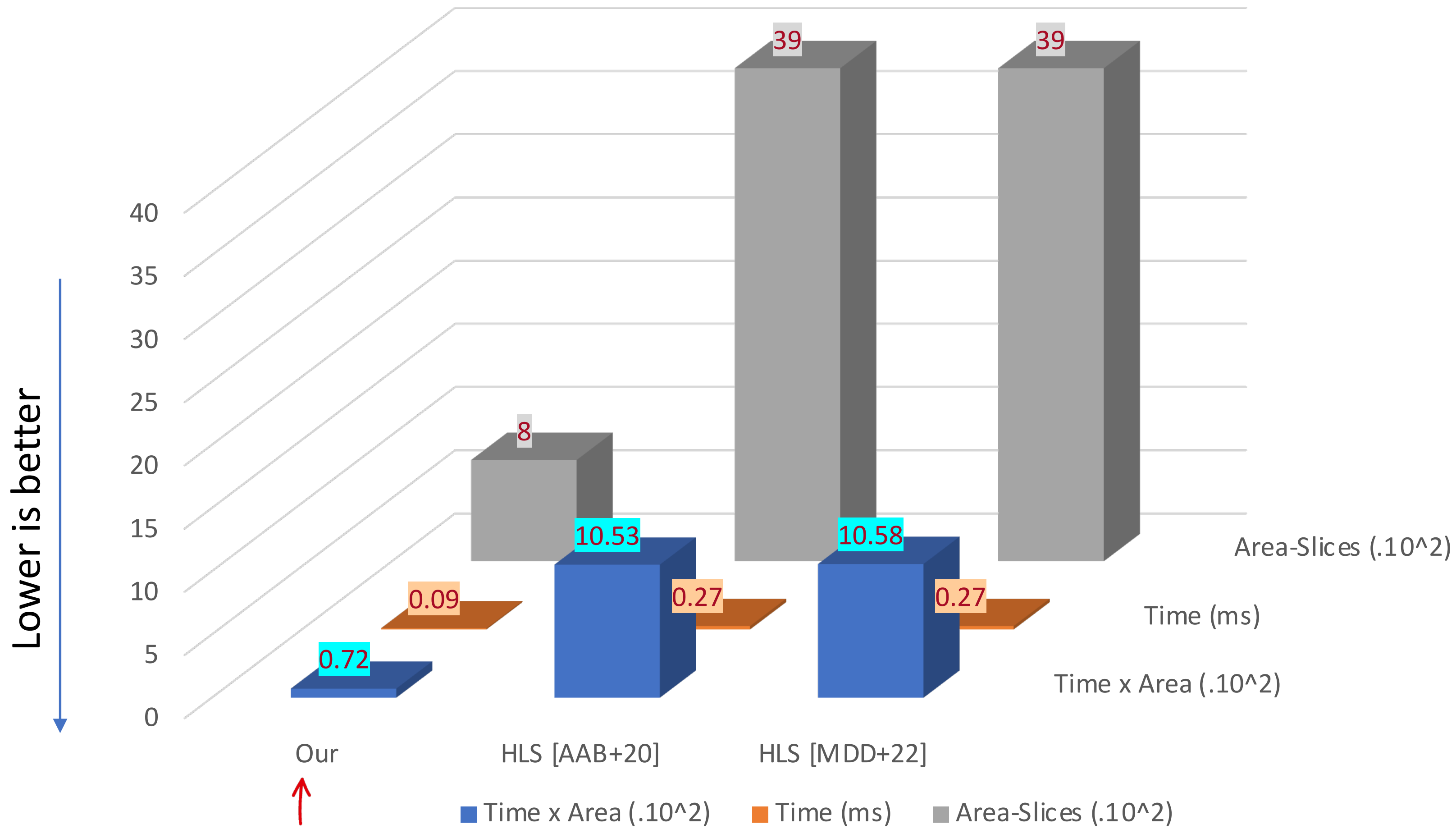
Outputs: public key(h, s), secret key (x, y)

1. (x,y) = FixedWeight(sk_seed)
2. h = VectorRandom(pk_seed)
3. s = x + h.y



Key Generation – Hardware Design Performance Comparison

HQC128



caslab.io

Hardware Implementation of HQC KEM

1. **Key Generation**
    - Fixed weight error vector generation
        - SHAKE256
    - Polynomial Multiplication
    - Polynomial Addition

2. **Encapsulation**
    - Encrypt
        - Encode
        - Polynomial Multiplication
        - Polynomial Addition
        - Fixed weight error vector generation
            - SHAKE256
    - Hash Computation
        - SHAKE256

3. **Decapsulation**
    - Decrypt
        - Decode
        - Polynomial Subtraction
        - Polynomial Multiplication
    - Encapsulation

Encrypt - Algorithm and Hardware Design

Inputs: theta, Public Key (h_in, s_in), message (m_in)

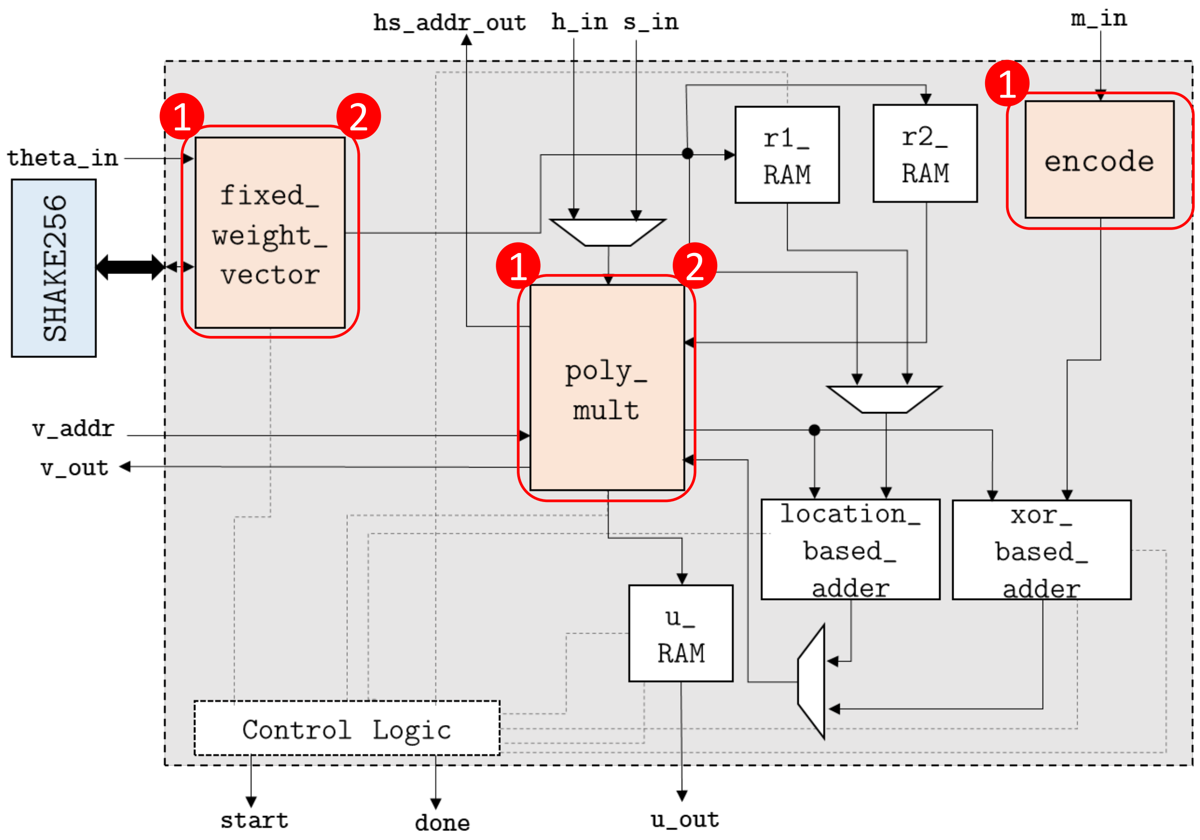
Outputs: Ciphertext (u_out, v_out)

1. $u\_out = r_1 + h.r_2$
2. $v\_out = m\_in.G + S.r_2 + e$

Where m_inG = Encoded message

r_1, r_2, e are fixed weight vectors

- Encode consists of Reed-Solomon and Reed-Muller Encoding



Encrypt - Algorithm and Hardware Design

Inputs: theta, Public Key (h_in, s_in),

message (m_in)

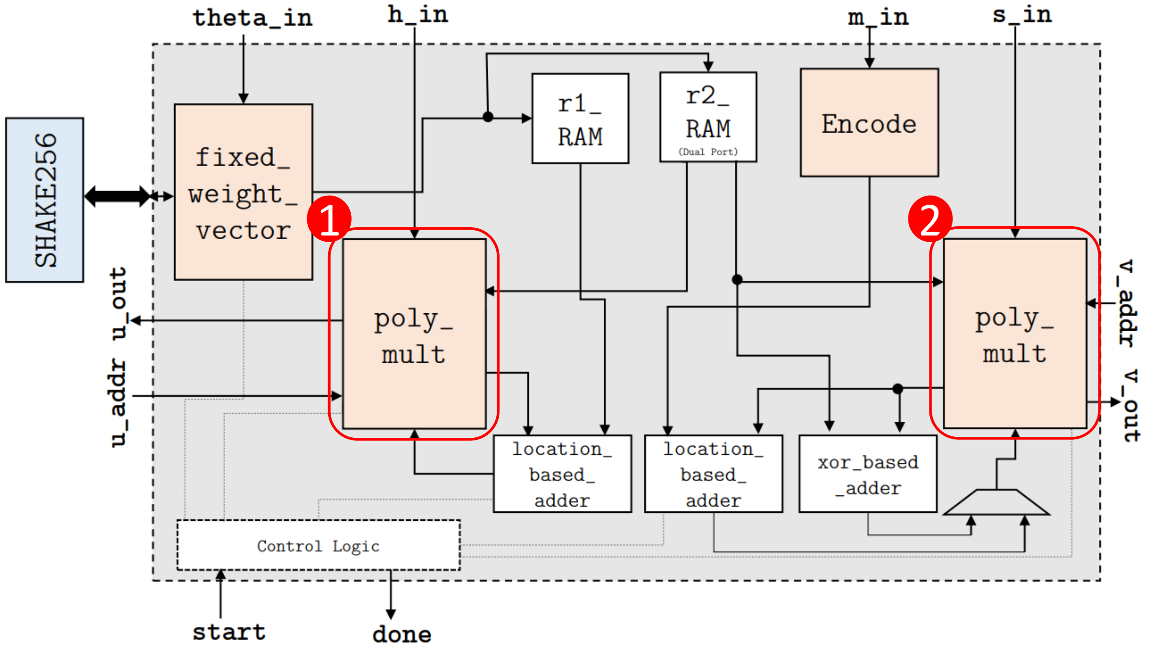
Outputs: Ciphertext (u_out, v_out)

1. $u\_out = r_1 + h.r_2$
2. $v\_out = m\_in.G + S.r_2 + e$

Where m_inG = Encoded message

r_1, r_2, e are fixed weight vectors

- Two polynomial multiplications can be run in parallel.



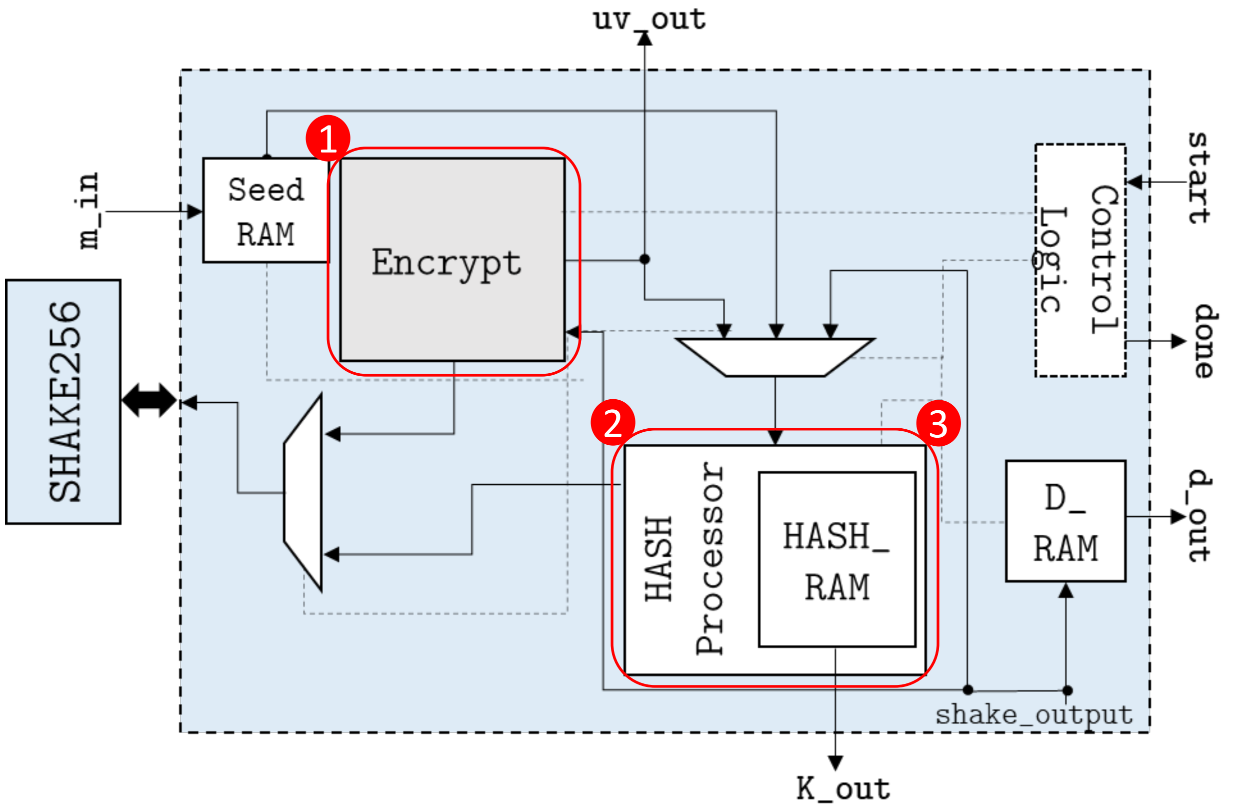
caslab.io

Encapsulation - Algorithm and Hardware Design

Inputs: Public Key (h_in, s_in), message (m_in)

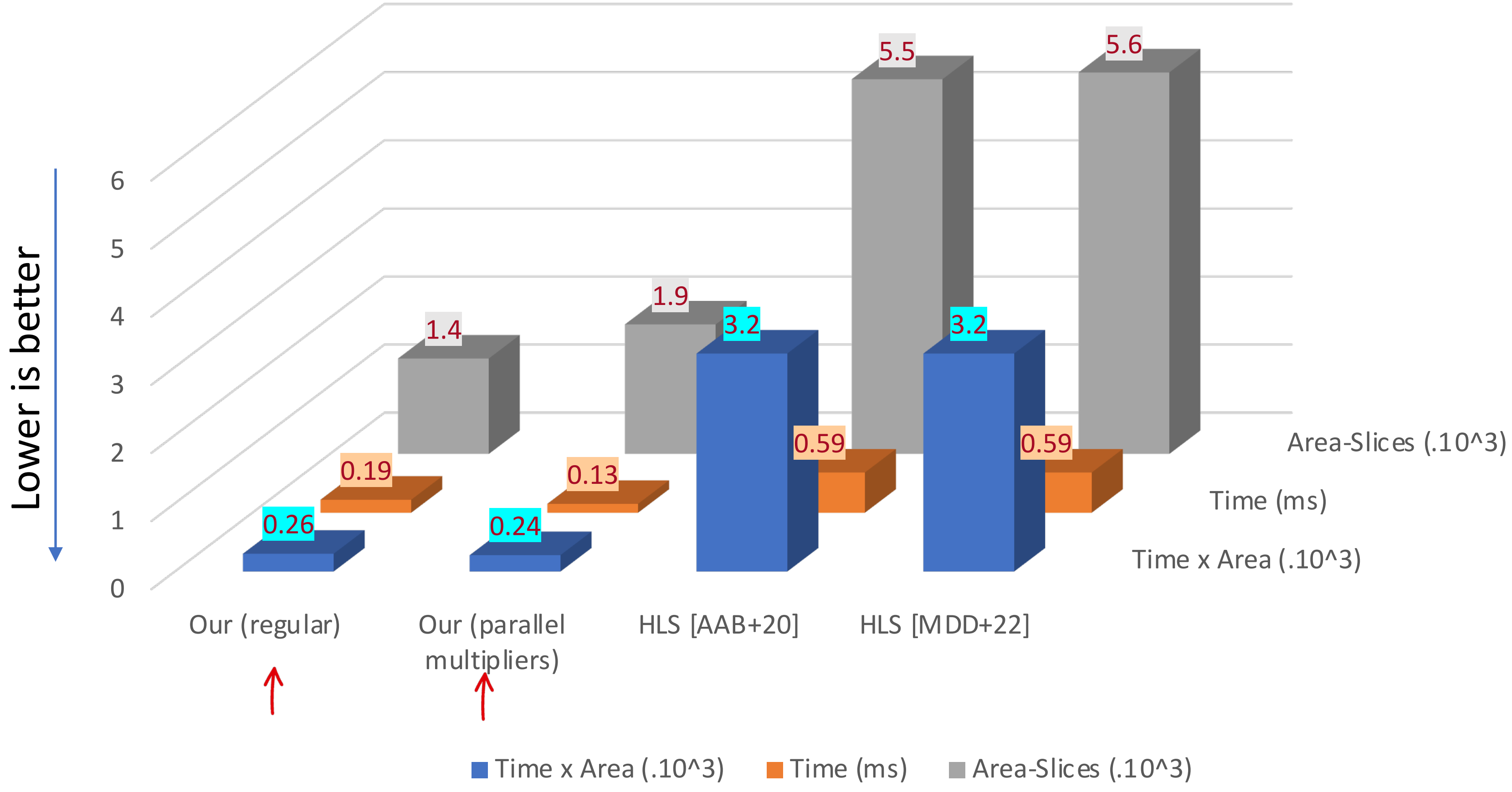
Outputs: Ciphertext (u_out, v_out), Shared Secret (K_out), Hashed message(d_out)

1. (u_out, v_out) = Encrypt(m_in)
2. d = Hash(m_in)
3. SS = Hash(m_in, u_out, v_out)



Encapsulation – Hardware Design Performance Comparison

HQC128



caslab.io

Hardware Implementation of HQC KEM

1. **Key Generation**
   - Fixed weight error vector generation
     - SHAKE256
   - Polynomial Multiplication
   - Polynomial Addition

2. **Encapsulation**
   - Encrypt
     - Encode
     - Polynomial Multiplication
     - Polynomial Addition
     - Fixed weight error vector generation
       - SHAKE256
   - Hash Processing
     - SHAKE256

3. Decapsulation
   - Decrypt
     - Decode
     - Polynomial Subtraction
     - Polynomial Multiplication
   - Encapsulation

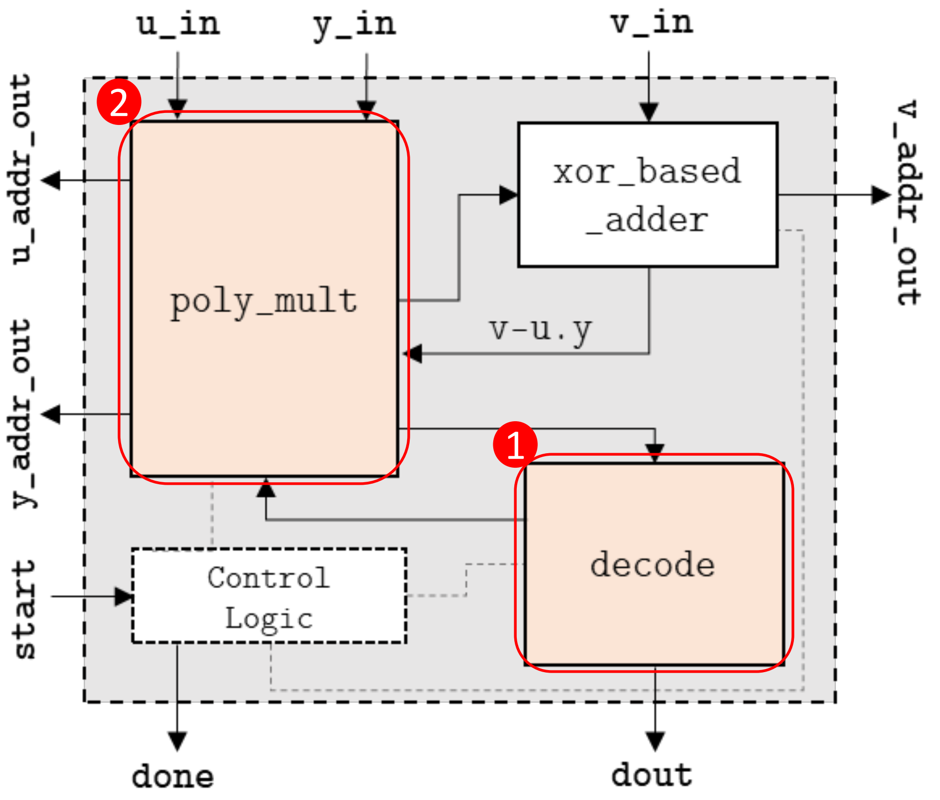
Decrypt - Algorithm and Hardware Design

Inputs: Secret Key (y_in), Ciphertext (u_in,v_in)

Outputs: Decoded Message (dout)

1. v_minus_uy = v_in - u_in.y_in
2. dout = Decode(v_minus_uy)

- Decode consists of Reed-Muller and Reed-Solomon decoding



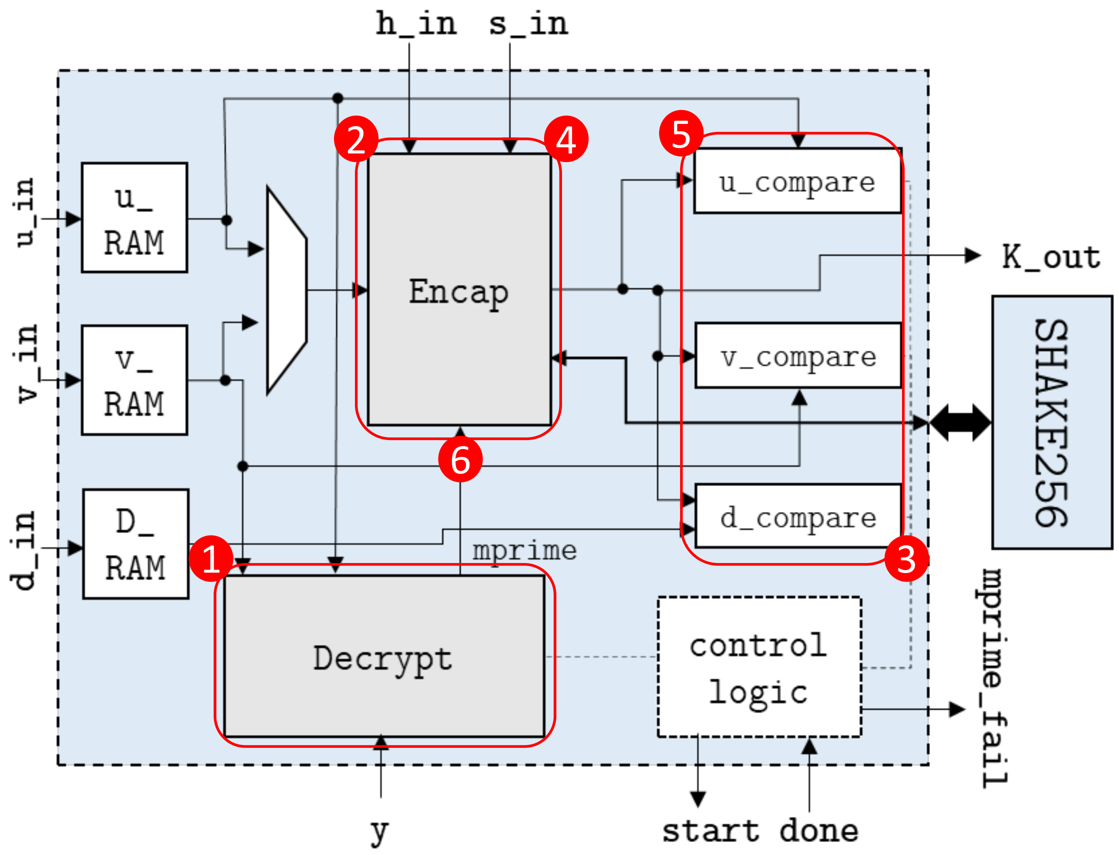
Decapsulation

Inputs: Public Key(h_in, s_in), Secret Key (y_in), Ciphertext (u_in, v_in), Hashed message (d_in)

Outputs: Shared Secret (K_out)

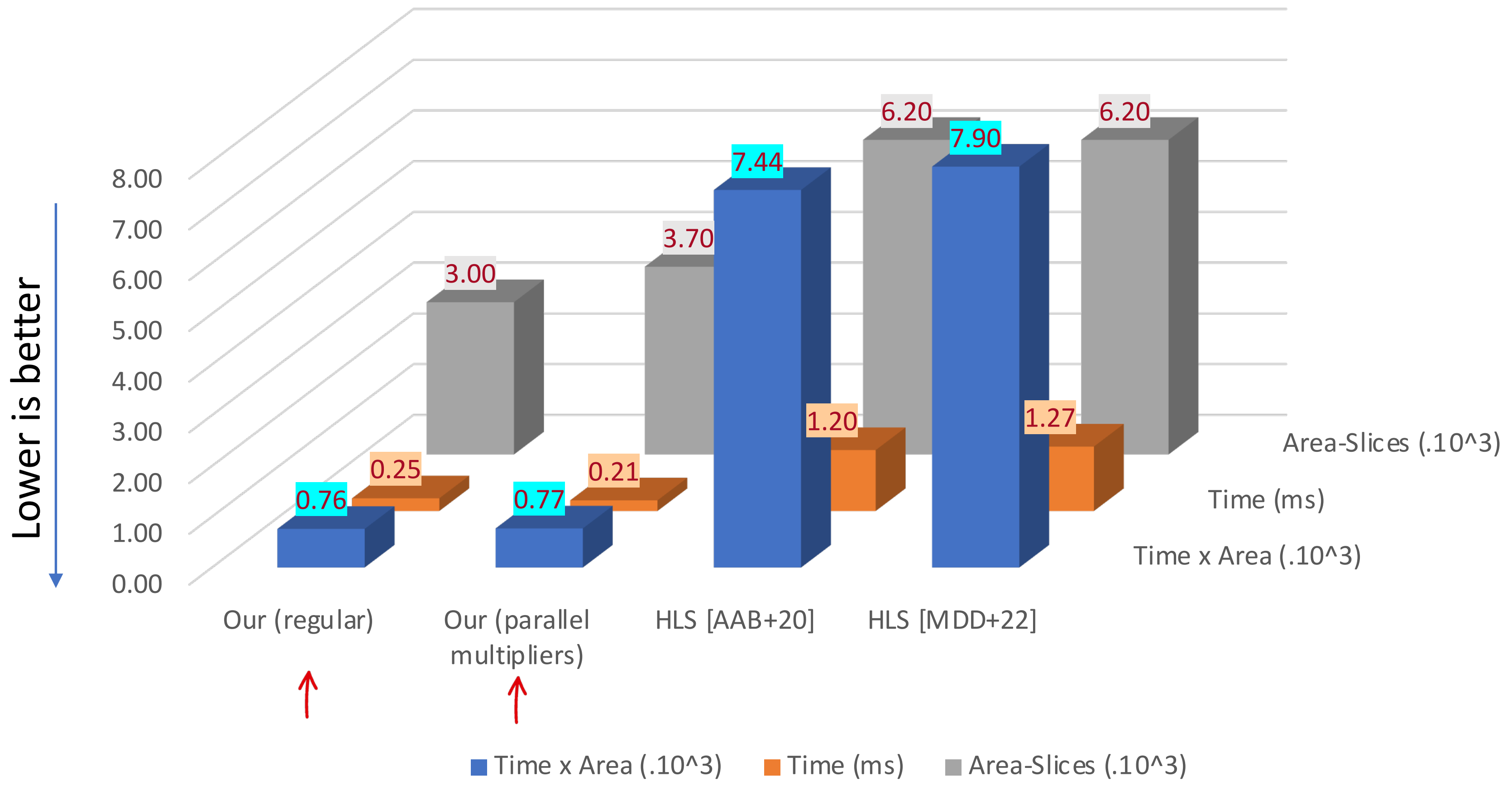
1. m' = Decode(v_in - u_in.y_in)
2. d'= Hash(m')
3. Verify d_in == d'?
4. (u', v') = Encrypt(m')
5. Verify (u', v') == (u_in, v_in)?
6. K_out = Hash(m', u_in, v_in)

- Decapsulation uses a tweaked version of Encapsulation module where uprime and vprime could be swapped with u_in and v_in.



Decapsulation – Hardware Design Performance Comparison

HQC128

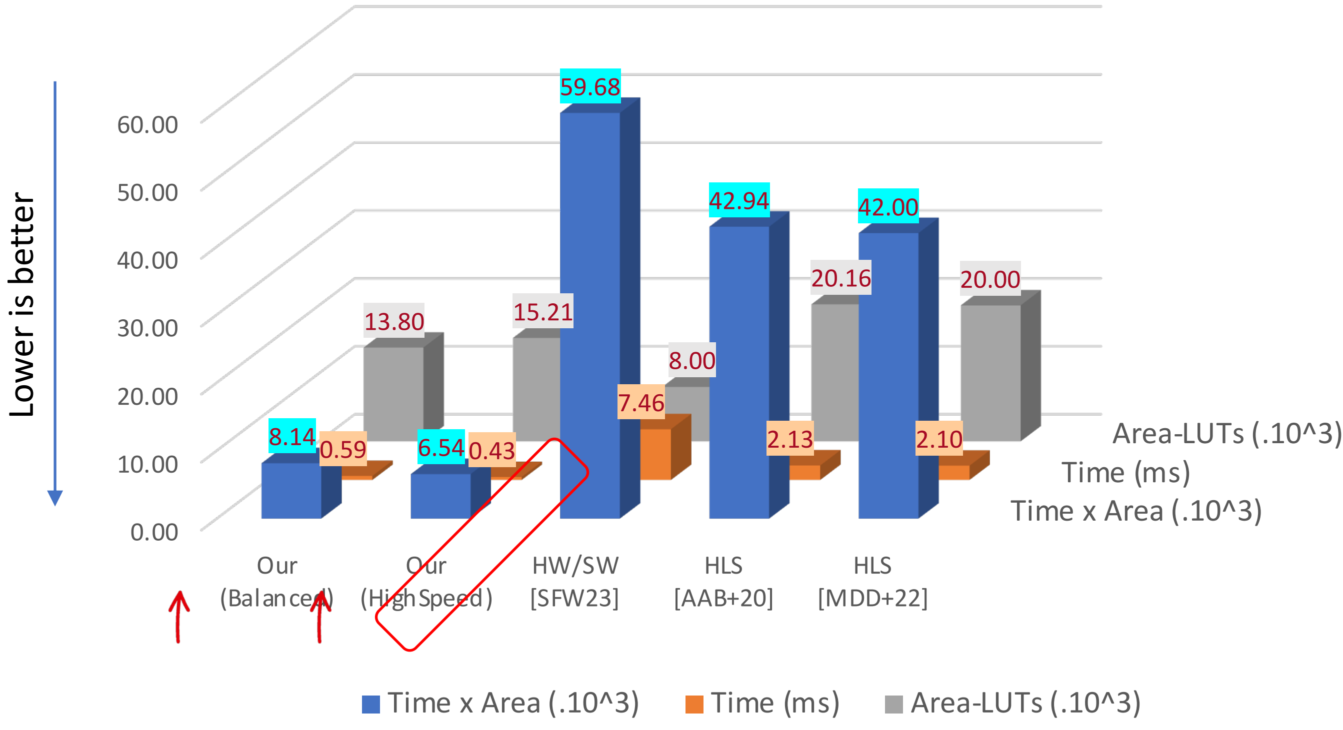


caslab.io

Joint Design – Comparison with other HQC Designs

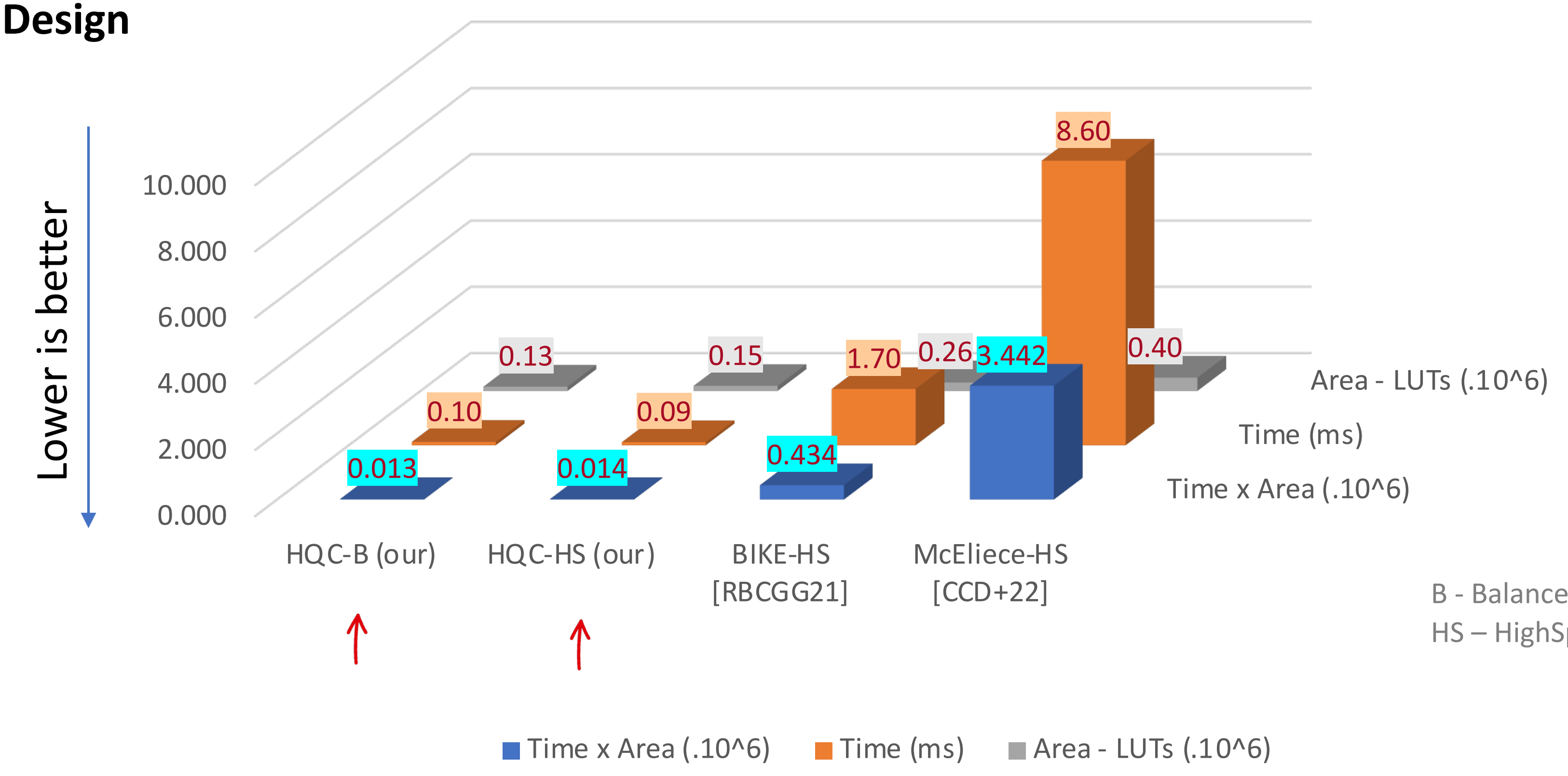
- Joint Design – combines KeyGen, Encap, and Decap in to one.
- Shared module among different primitives.
    - SHAKE256
    - Polynomial Multiplication
    - Encapsulation
    - Polynomial Addition
- Time = KeyGen + Encap + Decap.

HQC128



Comparison with other Code Based Schemes - Key Generation

128-bit Security Level
Hardware Design

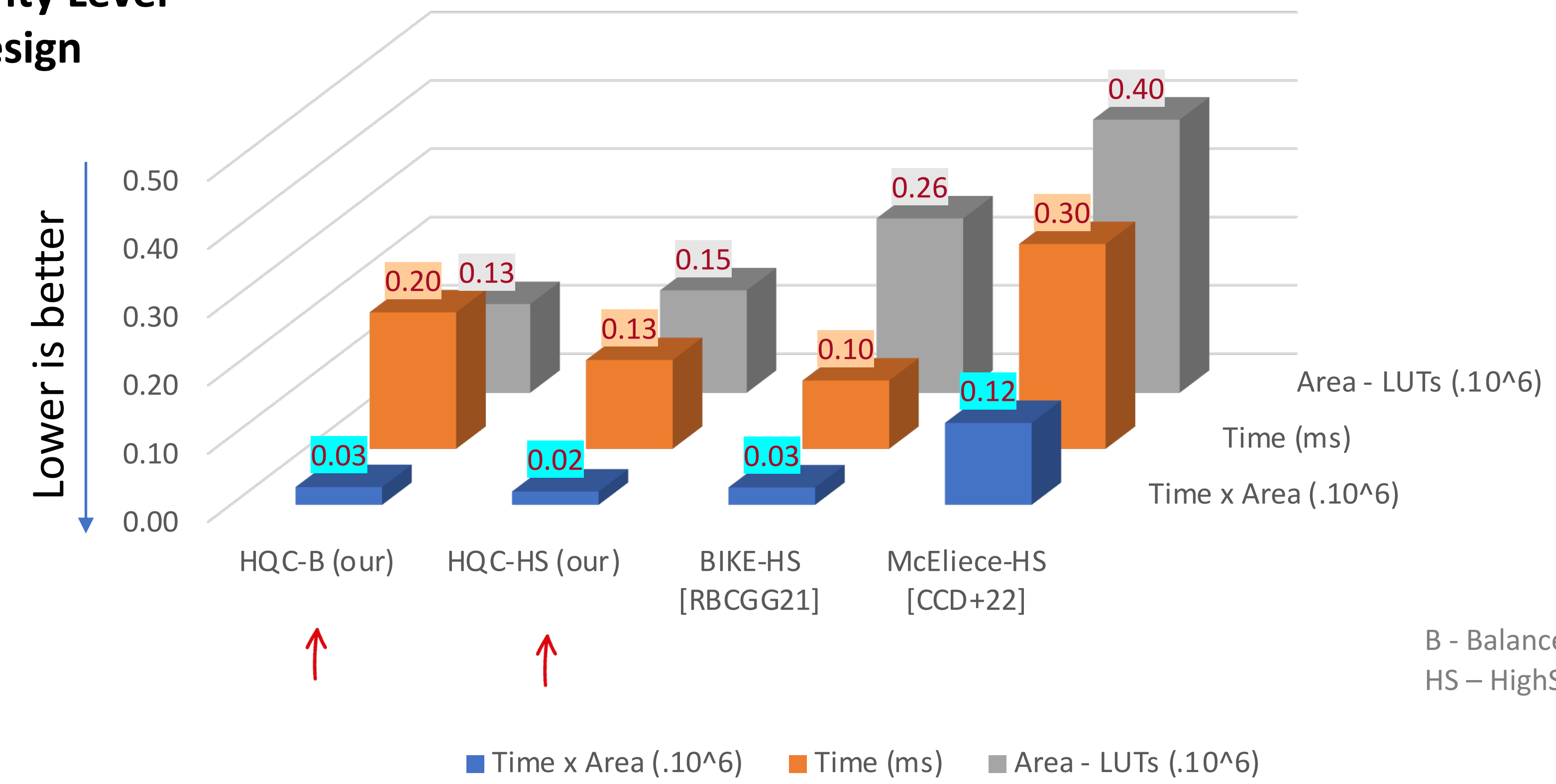


B - Balanced
HS – HighSpeed

caslab.io

Comparison with other Code Based Schemes - Encapsulation

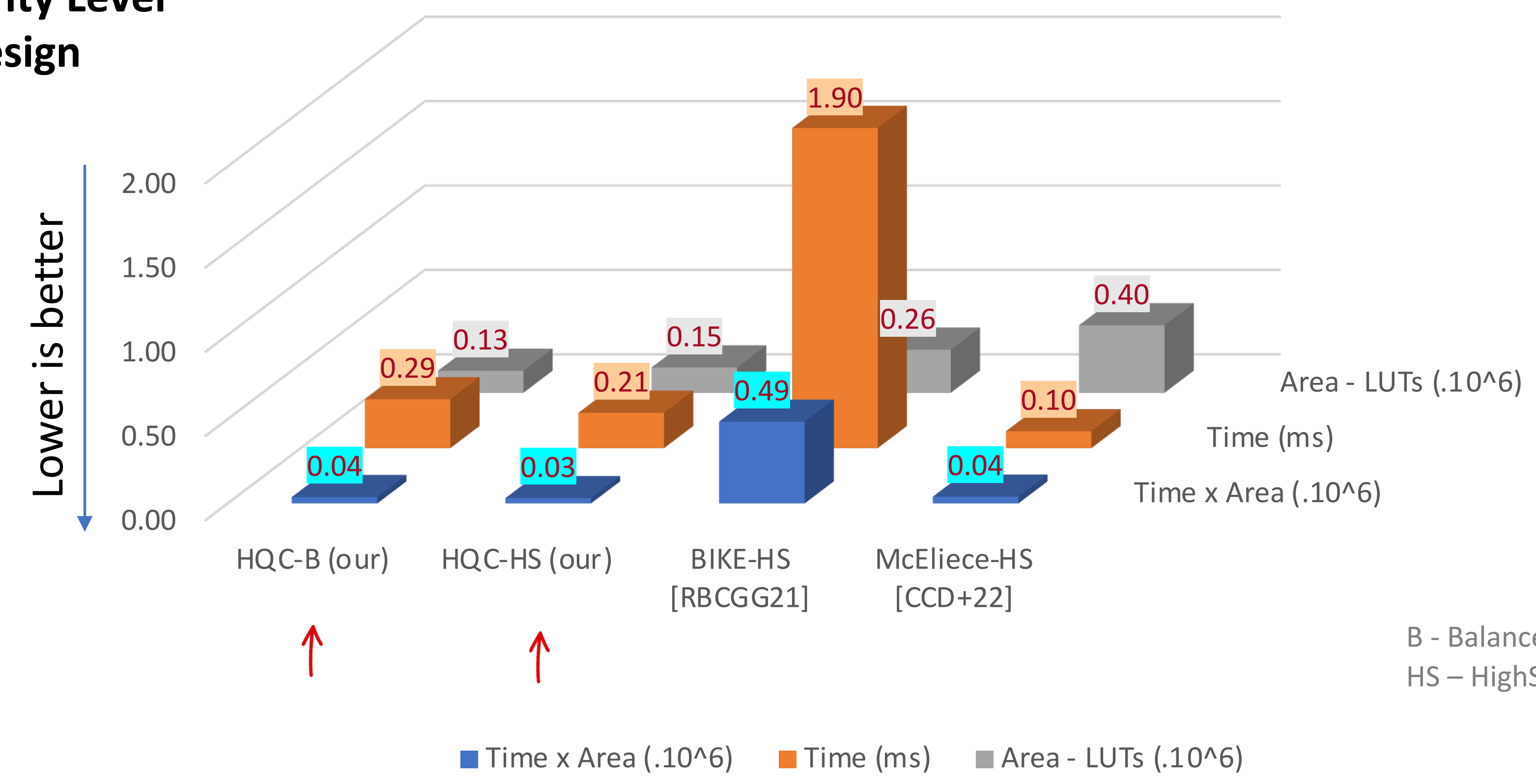
128-bit Security Level
Hardware Design



B - Balanced
HS – HighSpeed

Comparison with other Code Based Schemes - Decapsulation

128-bit Security Level
Hardware Design



B - Balanced
HS – HighSpeed

caslab.io

Summary

- First (hand-tailored) hardware implementations of HQC Key Encapsulation Mechanism parameterizable at compile-time across all parameter sets.
- HQC can be a competitive candidate when optimized hardware is developed.

References

[AAB+20] Carlos Aguilar Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Edoardo Persichetti, Gilles Zémor, and Jurjen Bos. HQC. Technical report, National Institute of Standards and Technology, 2020. available at https://pqc-hqc.org/doc/hqc-specification_2021-06-06.pdf

[MDD+22] Carlos Aguilar-Melchor, Jean-Christophe Deneuville, Arnaud Dion, James Howe, Romain Malmain, Vincent Migliore, Mamuri Nawan, and Kashif Nawaz. Towards automating cryptographic hardware implementations: a case study of hqc. Cryptology ePrint Archive, Paper 2022/1425, 2022. https://eprint.iacr.org/2022/1425.

[CCD+22] Po-Jen Chen, Tung Chou, Sanjay Deshpande, Norman Lahr, Ruben Niederhagen, Jakub Szefer, and Wen Wang. Complete and improved FPGA implementation of Classic Mceliece. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2022(3), 2022.

[RBCGG21] Jan Richter-Brockmann, Ming-Shing Chen, Santosh Ghosh, and Tim Güneysu. Racing bike: Improved polynomial multiplication and inversion in hardware. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2022(1):557–588, Nov. 2021.

[GHJ+22] Qian Guo, Clemens Hlauschek, Thomas Johansson, Norman Lahr, Alexander Nilsson, and Robin Leander Schröder. Don't reject this: Key-recovery timing attacks due to rejection-sampling in hqc and bike. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2022, Issue 3:223–263, 2022.

[HWCW19] Jingwei Hu, Wen Wang, Ray C.C. Cheung, and Huaxiong Wang. Optimized polynomial multiplier over commutative rings on fpgas: A case study on bike. In 2019 International Conference on Field-Programmable Technology (ICFPT), pages 231–234, 2019.

[**ZZY+21]** Zhong, Han-Sen & Deng, Yu-Hao & Qin, Jian & Wang, Hui & Chen, Ming-cheng & Peng, Li-Chao & Luo, Yi-Han & Wu, Dian & Gong, Si-Qiu & Su, Hao & Hu, Yi & Hu, Peng & Yang, Xiao-Yan & Zhang, Weijun & Li, Hao & Yuxuan, Li & Jiang, Xiao & Gan, Lin & Yang, Guangwen & Pan, Jian-Wei. (2021). Phase-Programmable Gaussian Boson Sampling Using Stimulated Squeezed Light

[SEN21] Sendrier, Nicolas**,** Secure sampling of constant-weight words – application to bike. Cryptology ePrint Archive, Paper 2021/1631 (2021), https://eprint.iacr.org/2021/1631.pdf

[SFW21] Schöffel, Maximilian, Feldmann, Johannes, Wehn, Norbert. (2023). Code-based Cryptography in IoT: A HW/SW Co-Design of HQC. 10.48550/arXiv.2301.04888.

.

Summary

- First (hand-tailored) hardware implementations of HQC Key Encapsulation Mechanism parameterizable at compile-time across all parameter sets.
- HQC can be a competitive candidate when optimized hardware is developed.

Thank you!
Questions?

Link to the code base

caslab.io