

MPSpeed: Implementing and Optimizing MPC-in-the-Head Digital Signatures in Hardware

Accelerating Mirath on FPGA

CBCrypto Workshop
May 9, 2026

Stelios Manasidis, Quinten Norga, Suparna Kundu, and Ingrid Verbauwhede



COSIC

Outline

- Introduction
- Background
- Optimizing GGM Tree Expansion for Hardware
- Hardware Architecture and Scheduling of Operations
- Results & Conclusions

Post-Quantum Crypto (PQC) Standardization



NIST Additional Digital Signatures call (in progress):

- many different families
- limited hardware research

MPC in-the-Head (MPCitH):

- Promising DSA^[1] family
- **Relatively slow, but improving**
- **Very small public keys**
- Relatively large signature sizes

[1] DSA = Digital Signature Algorithm

The MPCitH Construction

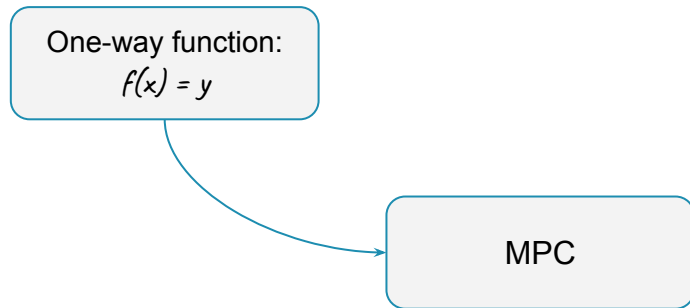
One-way function:

$$f(x) = y$$

Zero-knowledge proof (ZKP):

The **prover** wants to prove knowledge of a **secret x** s.t. $f(x) = y$, without revealing any info on x.

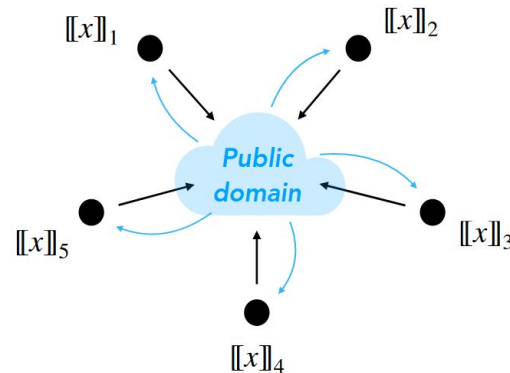
The MPCitH Construction



Multi-Party Computation (MPC):
Split **secret** x into N private shares

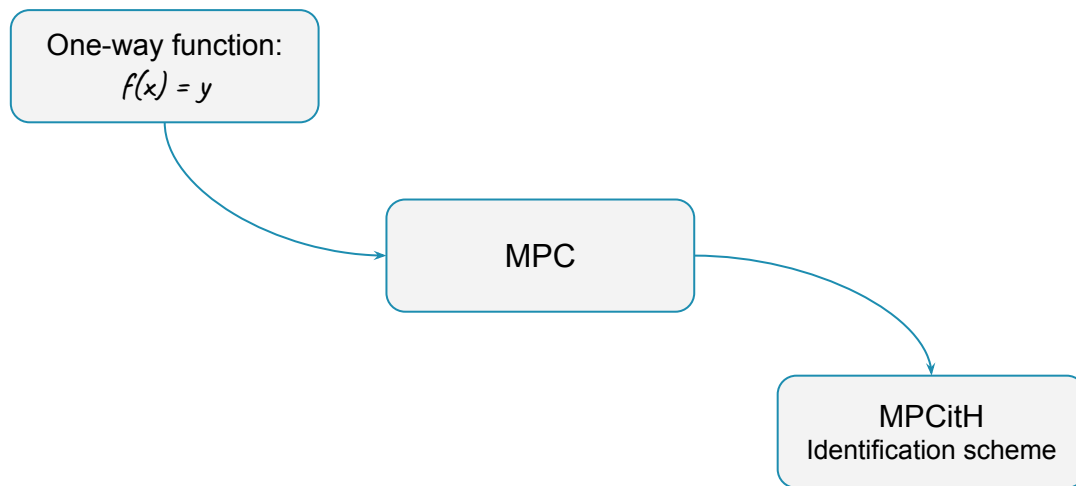
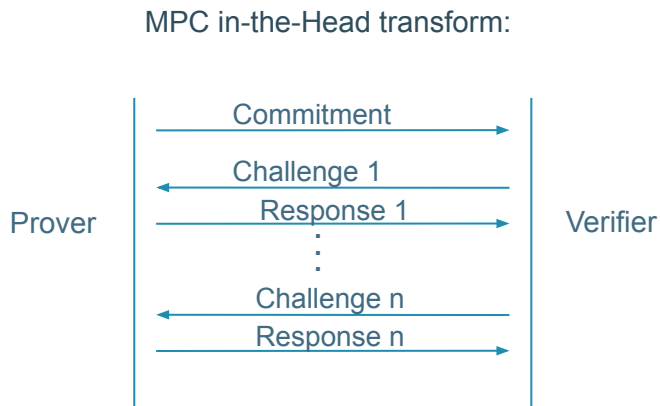
$$x = \llbracket x \rrbracket_1 + \llbracket x \rrbracket_2 + \dots + \llbracket x \rrbracket_N$$

Parties **jointly** compute $f(x)$:

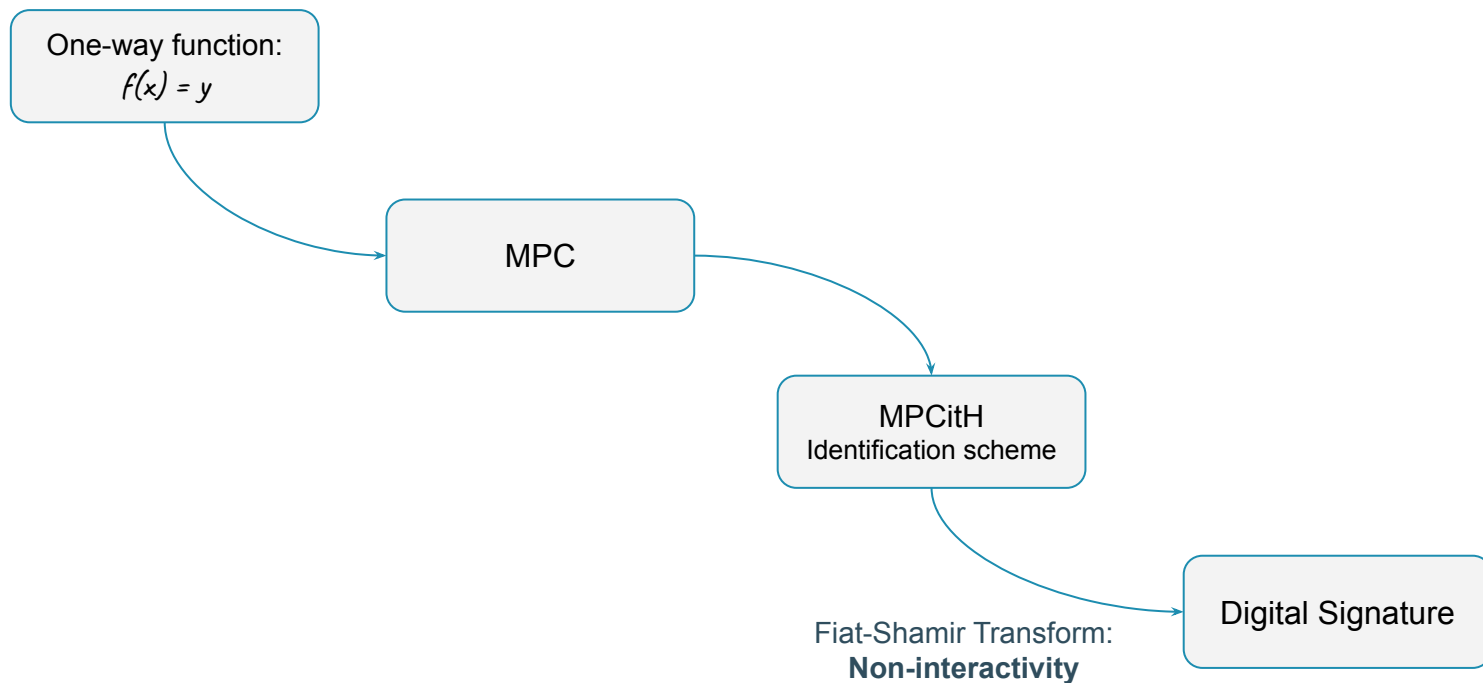


N-1 privacy: Revealing $N-1$ private shares
leaks zero information on x

The MPCitH Construction



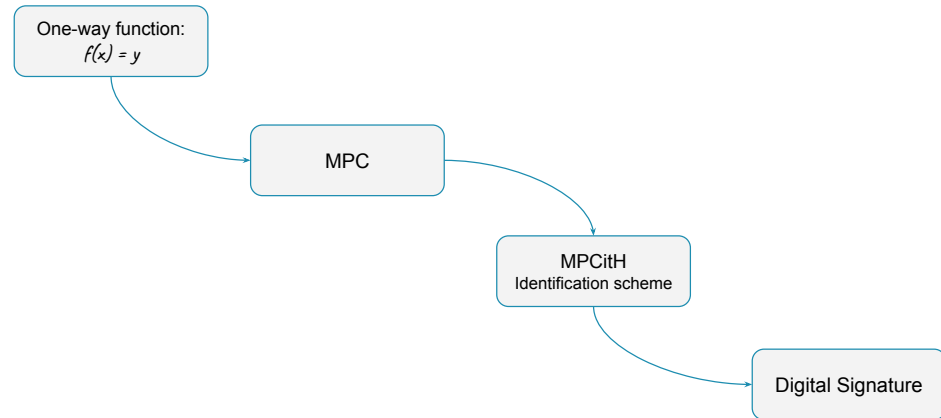
The MPCitH Construction



Soundness

$P(\text{cheating prover convinces verifier}) = P(\text{corrupted party stays hidden}) = 1/N$

Protocol repeated τ times in parallel $\longrightarrow$ soundness error $(1/N)^\tau$



Generating Random Shares

The GGM Tree

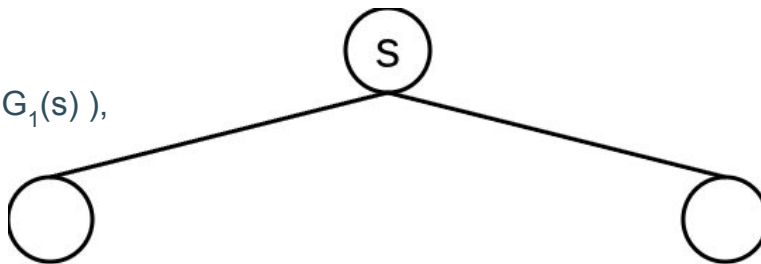
GGM (Goldreich–Goldwasser–Micali) Tree

- Starting from a random *root seed* s



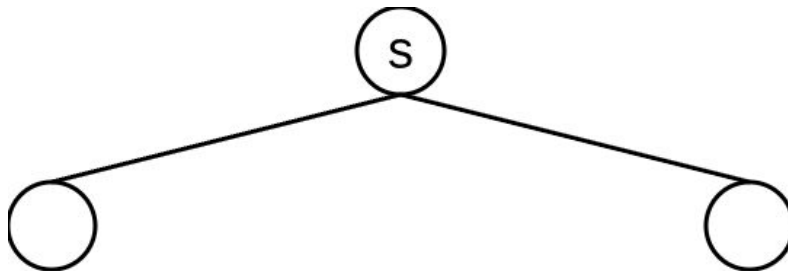
GGM (Goldreich–Goldwasser–Micali) Tree

- Starting from a random *root seed* s
- Using a length-doubling PRG: $G(s) = (G_0(s), G_1(s))$, start building a **binary tree**
- In Mirath: **PRG = AES / Rijndael**



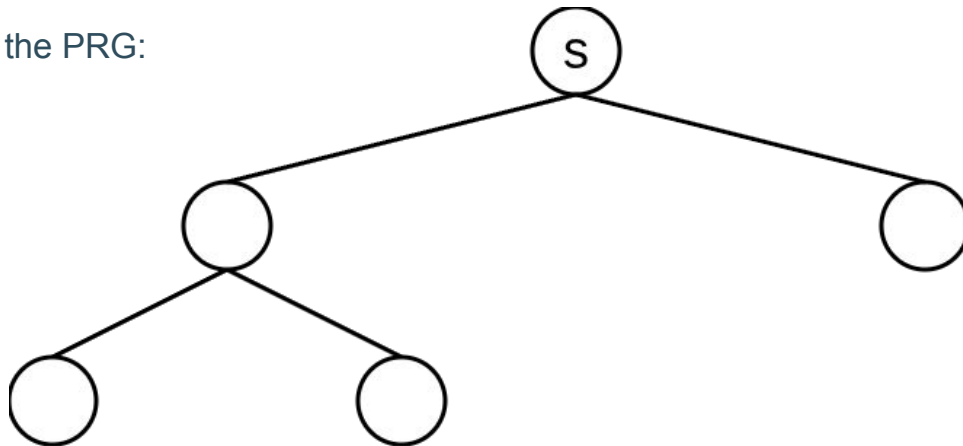
GGM (Goldreich–Goldwasser–Micali) Tree

- Repeatedly apply the PRG:



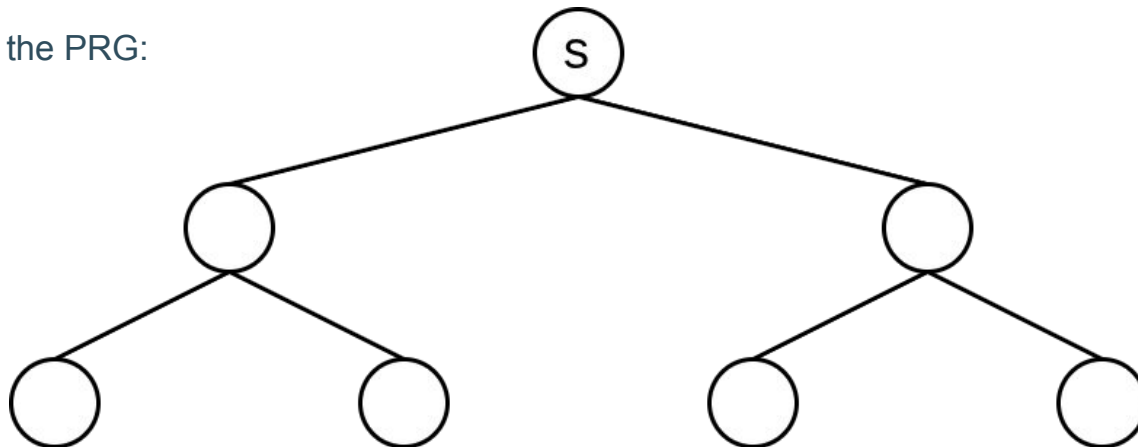
GGM (Goldreich–Goldwasser–Micali) Tree

- Repeatedly apply the PRG:



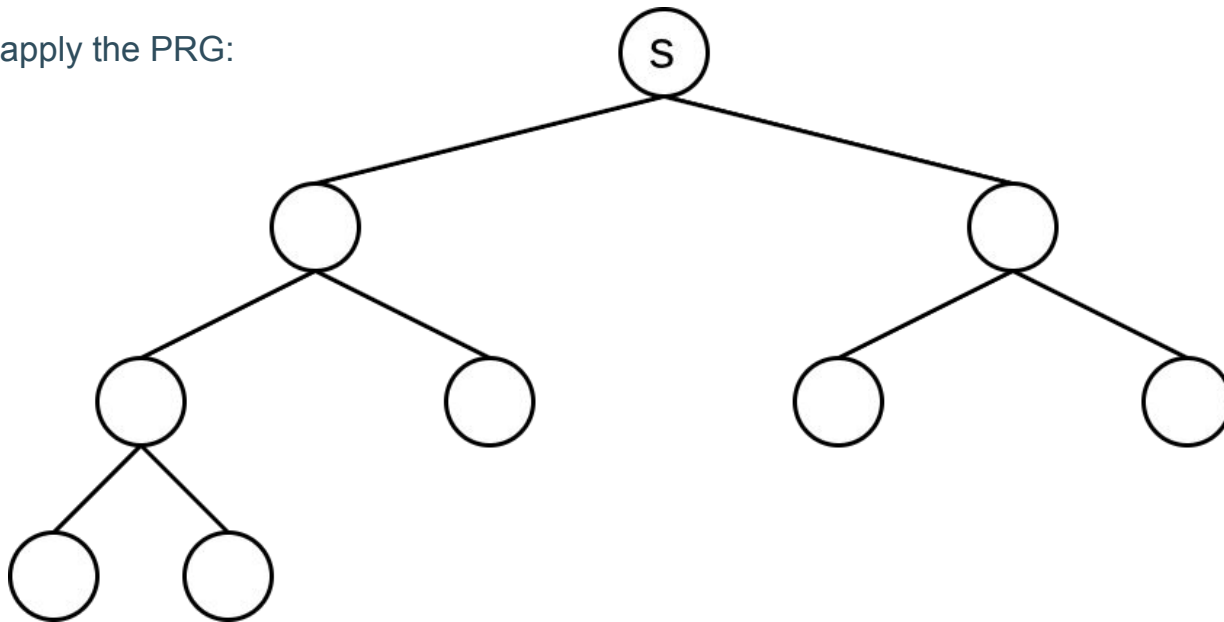
GGM (Goldreich–Goldwasser–Micali) Tree

- Repeatedly apply the PRG:



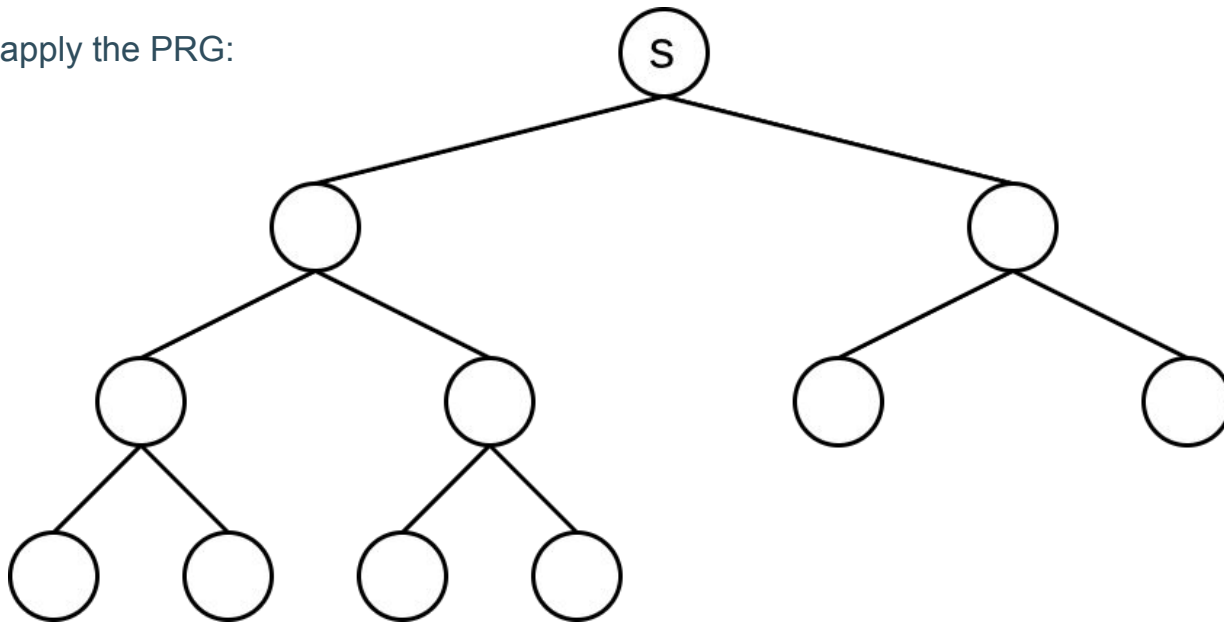
GGM (Goldreich–Goldwasser–Micali) Tree

- Repeatedly apply the PRG:



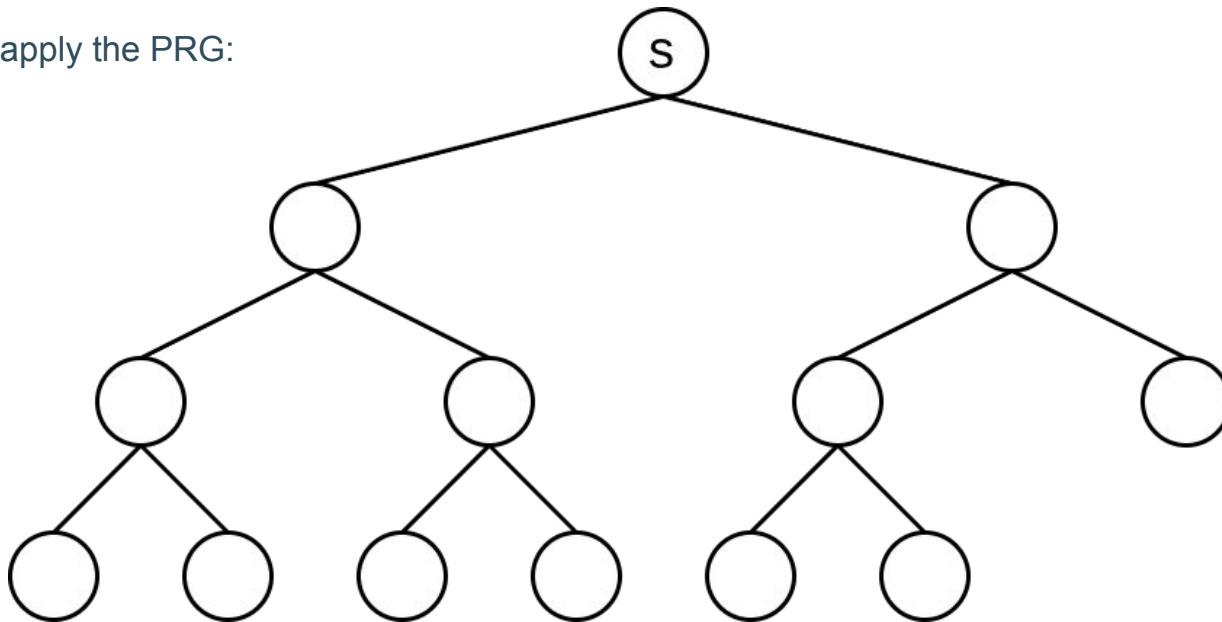
GGM (Goldreich–Goldwasser–Micali) Tree

- Repeatedly apply the PRG:



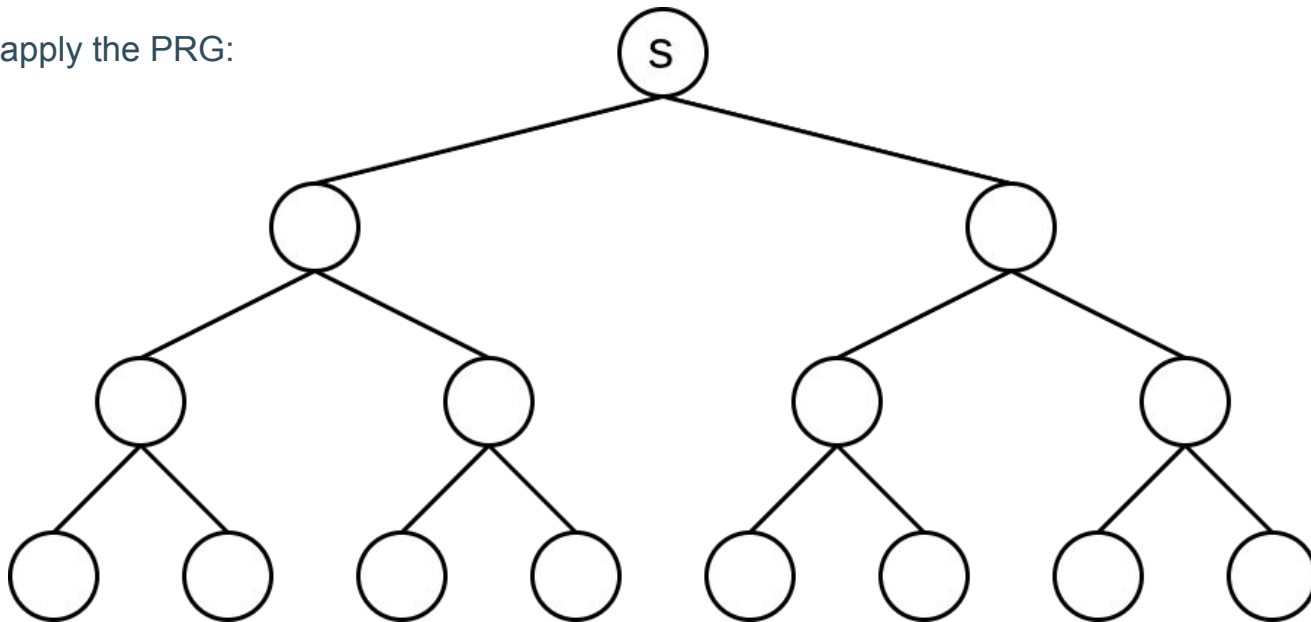
GGM (Goldreich–Goldwasser–Micali) Tree

- Repeatedly apply the PRG:



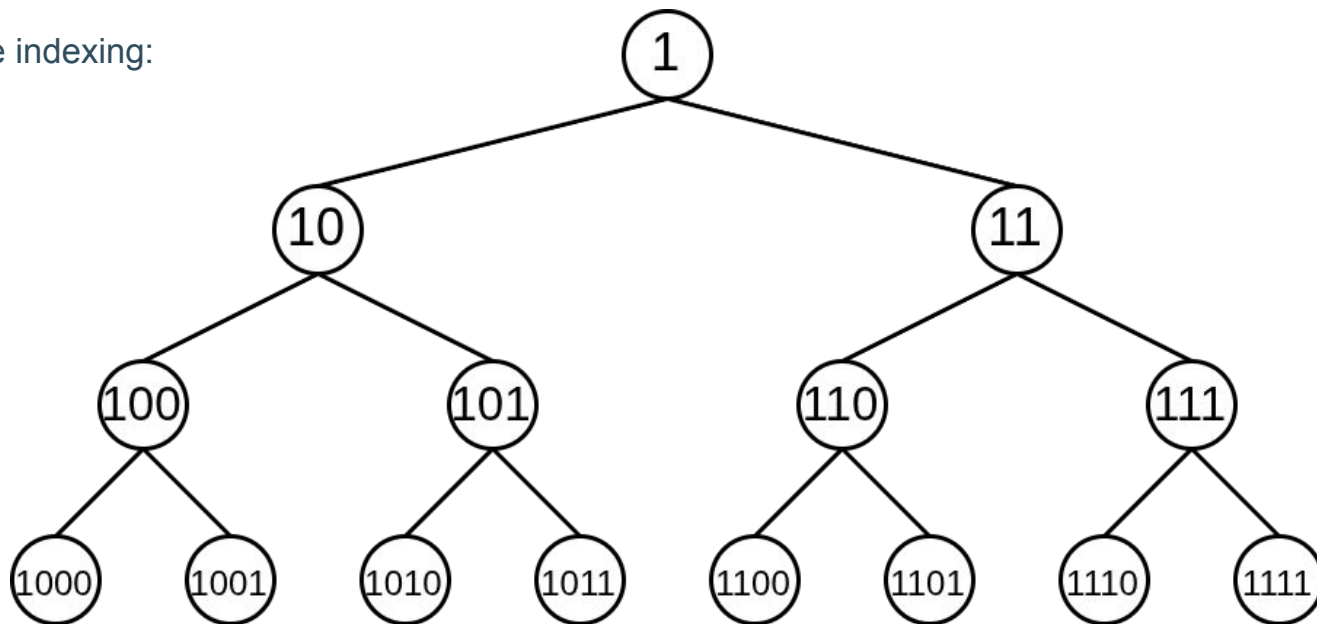
GGM (Goldreich–Goldwasser–Micali) Tree

- Repeatedly apply the PRG:



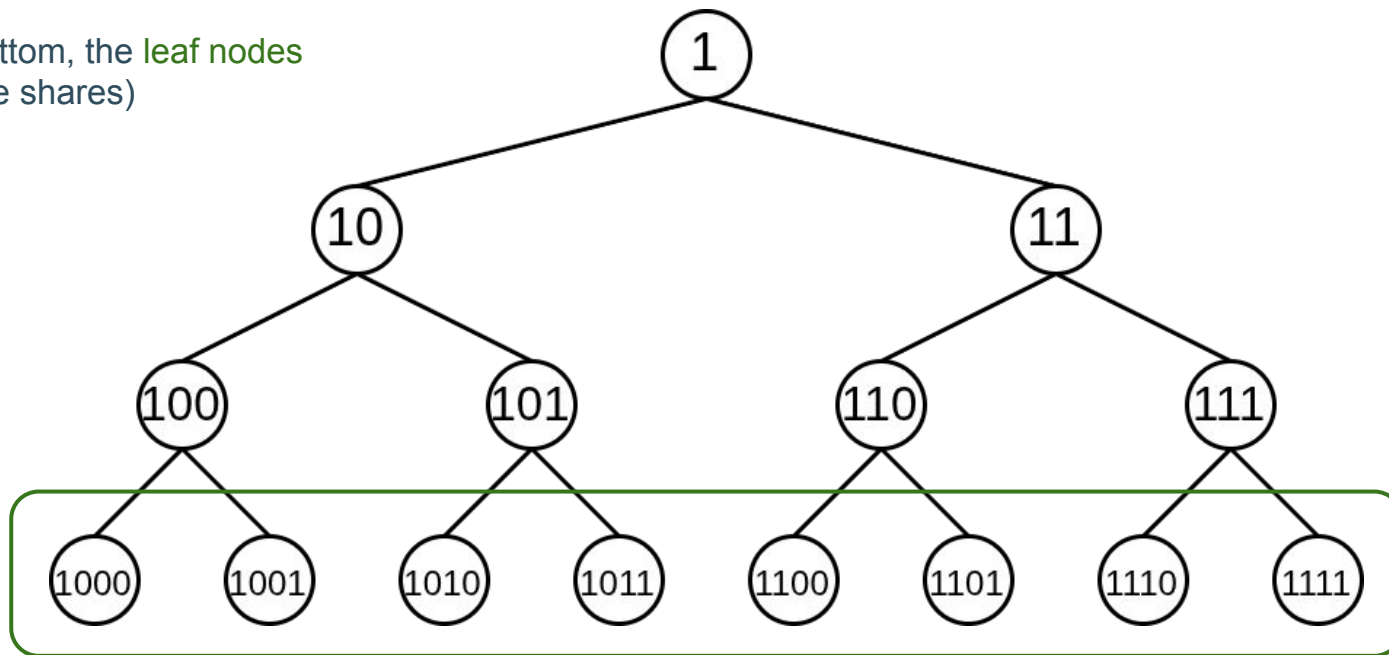
GGM (Goldreich–Goldwasser–Micali) Tree

- Binary node indexing:



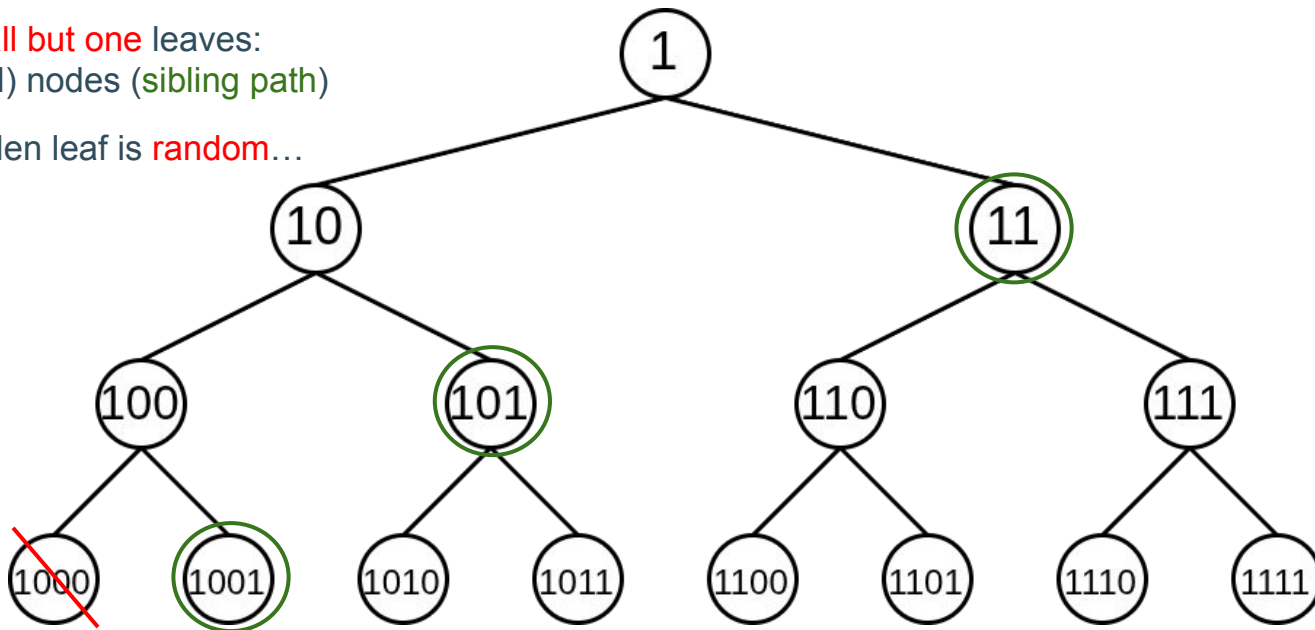
GGM (Goldreich–Goldwasser–Micali) Tree

- At the bottom, the **leaf nodes** (N private shares)



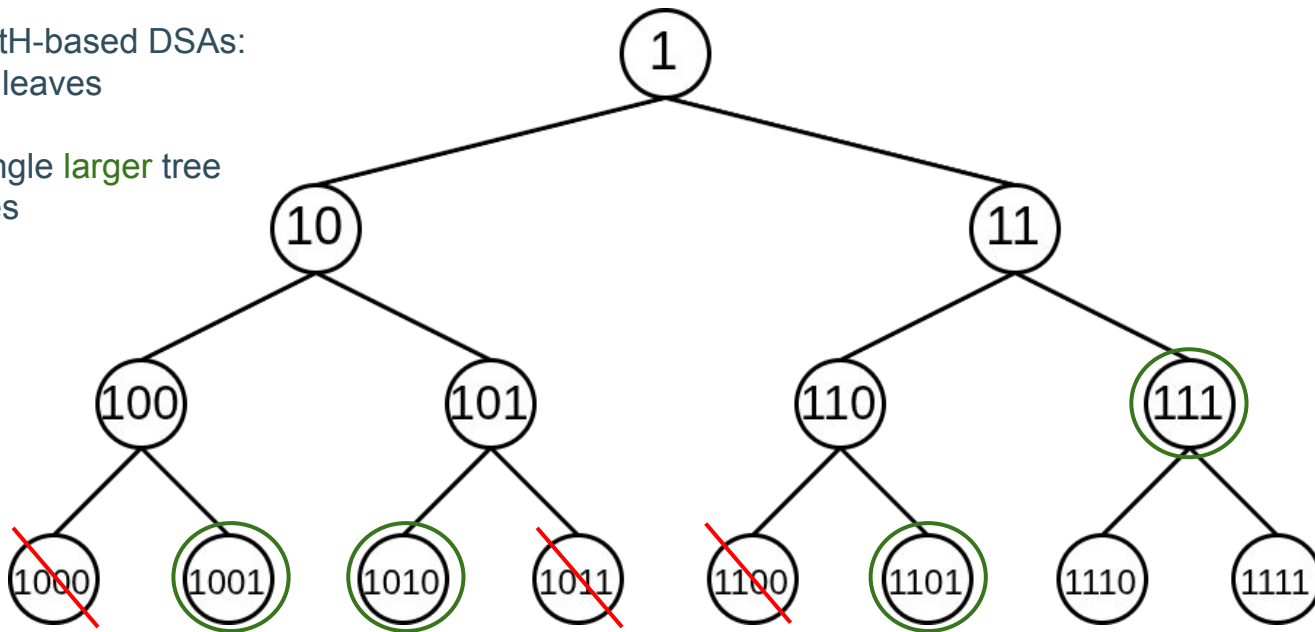
GGM (Goldreich–Goldwasser–Micali) Tree

- Revealing **all but one** leaves:
Send $\log_2(N)$ nodes (**sibling path**)
- But the hidden leaf is **random**...



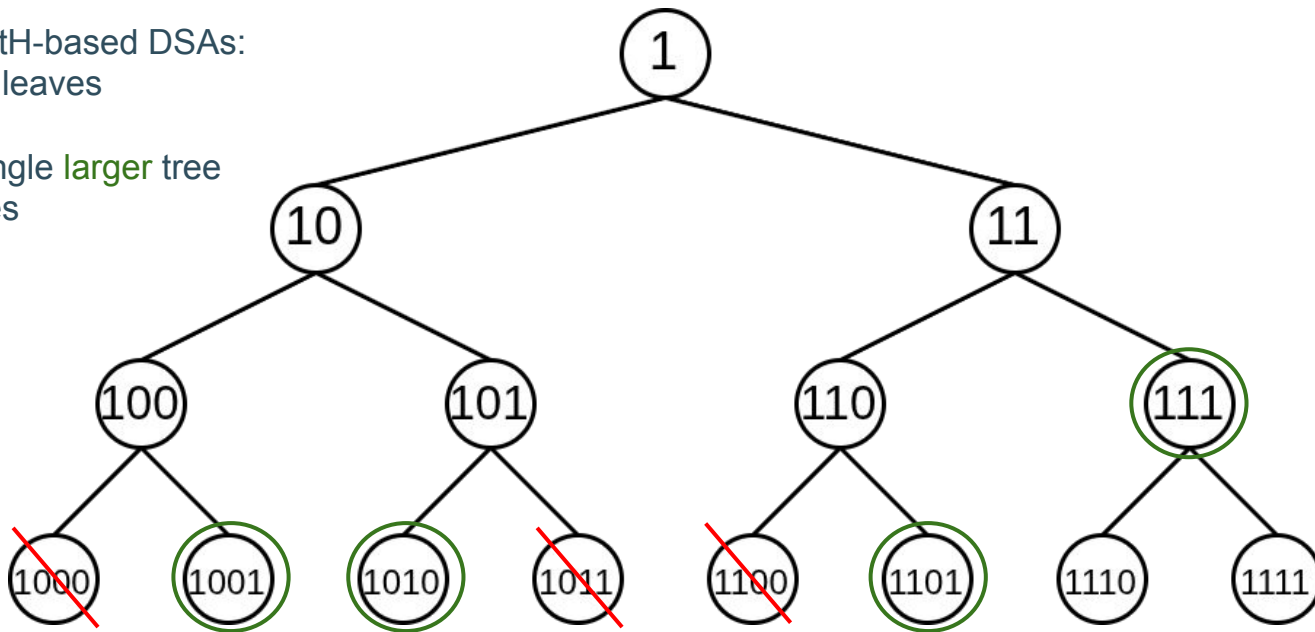
The One Tree Optimization

- Older MPCitH-based DSAs:
T trees of N leaves
- Mirath: A single larger tree
of T*N leaves



The One Tree Optimization

- Older MPCitH-based DSAs:
T trees of N leaves
- Mirath: A single larger tree
of T*N leaves



Sibling paths merge: Signature size significantly improved, but required flexibility also increases

The Mirath DSA

Mirath Variants & Parameter Sets

Two variants of Mirath

- **Threshold Computation in-the-Head (TCitH)**
- VOLE-in-the-Head

Short and **fast** parameter sets

Two finite fields supported

- $q = 16$ (Mirath-a)
- $q = 2$ (**Mirath-b**)

Mirath-TCitH-b-fast

Mirath Routines



- Mirath: ***Keygen, Sign, Verify***

Mirath Routines



- Mirath: ***Keygen, Sign, Verify***
- **Sign:** Most complex & computationally expensive routine

Mirath Routines



- Mirath: ***Keygen, Sign, Verify***
- **Sign:** Most complex & computationally expensive routine
- **Keygen:** (computationally) identical to Sign Step 0

Mirath Routines

- Mirath: **Keygen**, **Sign**, **Verify**
- **Sign**: Most complex & computationally expensive routine
- **Keygen**: (computationally) identical to Sign Step 0
- **Verify**: Steps 1-3 similar to Sign Steps 1-3

Mirath Routines



- Mirath: ***Keygen, Sign, Verify***
- **Sign:** Most complex & computationally expensive routine
- **Keygen:** (computationally) identical to Sign Step 0
- **Verify:** Steps 1-3 similar to Sign Steps 1-3
- Our design supports **all**

Mirath Routines

- Mirath: **Keygen, Sign, Verify**
- **Sign:** Most complex & computationally expensive routine
- **Keygen:** (computationally) identical to Sign Step 0
- **Verify:** Steps 1-3 similar to Sign Steps 1-3
- Our design supports **all**
- Here: we focus on the **Signing algorithm**

Algorithm 22 Signing algorithm.

Sign(sk, msg)

```
// Step 0: Initialization.
1 :  $(H', pk, S, C') \leftarrow \text{DecompressSK}(sk)$ 
2 :  $salt \leftarrow \text{TRG.Seed}(2\lambda)$ 
3 :  $rseed \leftarrow \text{TRG.Seed}(\lambda)$ 

// Step 1: Build & Commit to Witness Polynomials.
4 :  $(base, v, h_{sh}, key, aux) \leftarrow \text{CommitWitnessPolynomials}(salt, rseed, S, C')$ 

// Step 2: Compute the polynomial proof  $P_\alpha(X)$ .
5 :  $\Gamma \leftarrow \text{ExpandChallengeMatrix}(h_{sh}) \triangleright \Gamma \in \mathbb{F}_{q^\mu}^{D \times (mn-k)}$ 
6 : for  $e$  from 0 to  $(\tau - 1)$  do
7 :    $(\alpha_{mid}[e], \alpha_{base}[e]) \leftarrow \text{ComputePolynomialProof}(base[e], v[e], S, C', \Gamma, H')$ 
8 :  $h_{piop} \leftarrow \text{Hash}_2(pk, salt, msg, h_{sh},$ 
    $\alpha_{mid}[0], \alpha_{base}[0], \dots, \alpha_{mid}[\tau - 1], \alpha_{base}[\tau - 1])$ 

// Step 3: Open random evaluations of the polynomials  $P_S(X)$ ,  $P_C(X)$  and  $P_v(X)$ .
9 :  $(ctr, \pi_{BAVC}) \leftarrow \text{OpenRandomEvaluations}(key, h_{piop})$ 
10 :  $\sigma \leftarrow \text{UnparseSignature}(salt, ctr, h_{piop}, \pi_{BAVC}, aux, \alpha_{mid})$ 
11 : return  $\sigma$ 
```

Step 1: Commit to Witness Polynomials

- GGM Tree expansion + Committing to shares
= repeated AES / Rijndael PRG ops
- Hashing all commits: binds prover
- GGM Tree: 139 to 590 KB = **large memory requirement**
- Need a memory-optimized approach = **This work**

Algorithm 22 Signing algorithm.

Sign(sk, msg)

// Step 0: Initialization.

1 : $(H', pk, S, C') \leftarrow \text{DecompressSK}(sk)$

2 : $\text{salt} \leftarrow \text{TRG.Seed}(2\lambda)$

3 : $\text{rseed} \leftarrow \text{TRG.Seed}(\lambda)$

// Step 1: Build & Commit to Witness Polynomials.

4 : $(\text{base}, v, h_{\text{sh}}, \text{key}, \text{aux}) \leftarrow \text{CommitWitnessPolynomials}(\text{salt}, \text{rseed}, S, C')$

// Step 2: Compute the polynomial proof $P_{\alpha}(X)$.

5 : $\Gamma \leftarrow \text{ExpandChallengeMatrix}(h_{\text{sh}}) \triangleright \Gamma \in \mathbb{F}_{q^{\mu}}^{\rho \times (mn-k)}$

6 : **for** e from 0 to $(\tau - 1)$ **do**

7 : $(\alpha_{\text{mid}}[e], \alpha_{\text{base}}[e]) \leftarrow \text{ComputePolynomialProof}(\text{base}[e], v[e]S, C', \Gamma, H')$

8 : $h_{\text{piop}} \leftarrow \text{Hash}_2(pk, \text{salt}, \text{msg}, h_{\text{sh}},$
 $\alpha_{\text{mid}}[0], \alpha_{\text{base}}[0], \dots, \alpha_{\text{mid}}[\tau - 1], \alpha_{\text{base}}[\tau - 1])$

// Step 3: Open random evaluations of the polynomials $P_S(X)$, $P_C(X)$ and $P_v(X)$.

9 : $(\text{ctr}, \pi_{\text{BAVC}}) \leftarrow \text{OpenRandomEvaluations}(\text{key}, h_{\text{piop}})$

10 : $\sigma \leftarrow \text{UnparseSignature}(\text{salt}, \text{ctr}, h_{\text{piop}}, \pi_{\text{BAVC}}, \text{aux}, \alpha_{\text{mid}})$

11 : **return** σ

Step 2: Compute polynomial proof

- Linear algebra-heavy
- Computation performed τ times

Algorithm 22 Signing algorithm.

Sign(sk, msg)

// Step 0: Initialization.

1 : $(H', pk, S, C') \leftarrow \text{DecompressSK}(sk)$

2 : $\text{salt} \leftarrow \text{TRG.Seed}(2\lambda)$

3 : $\text{rseed} \leftarrow \text{TRG.Seed}(\lambda)$

// Step 1: Build & Commit to Witness Polynomials.

4 : $(\text{base}, v, h_{\text{sh}}, \text{key}, \text{aux}) \leftarrow \text{CommitWitnessPolynomials}(\text{salt}, \text{rseed}, S, C')$

// Step 2: Compute the polynomial proof $P_\alpha(X)$.

5 : $\Gamma \leftarrow \text{ExpandChallengeMatrix}(h_{\text{sh}}) \triangleright \Gamma \in \mathbb{F}_{q^\mu}^{\rho \times (mn-k)}$

6 : **for** e from 0 to $(\tau - 1)$ **do**

7 : $(\alpha_{\text{mid}}[e], \alpha_{\text{base}}[e]) \leftarrow \text{ComputePolynomialProof}(\text{base}[e], v[e]S, C', \Gamma, H')$

8 : $h_{\text{piop}} \leftarrow \text{Hash}_2(pk, \text{salt}, \text{msg}, h_{\text{sh}},$

$\alpha_{\text{mid}}[0], \alpha_{\text{base}}[0], \dots, \alpha_{\text{mid}}[\tau - 1], \alpha_{\text{base}}[\tau - 1])$

// Step 3: Open random evaluations of the polynomials $P_S(X)$, $P_C(X)$ and $P_v(X)$.

9 : $(\text{ctr}, \pi_{\text{BAVC}}) \leftarrow \text{OpenRandomEvaluations}(\text{key}, h_{\text{piop}})$

10 : $\sigma \leftarrow \text{UnparseSignature}(\text{salt}, \text{ctr}, h_{\text{piop}}, \pi_{\text{BAVC}}, \text{aux}, \alpha_{\text{mid}})$

11 : **return** σ

Step 3: Open Random Evaluations

- **Repeatedly** sample hidden leaf indices until a valid one is found

Algorithm 22 Signing algorithm.

Sign(sk, msg)

// Step 0: Initialization.

1 : $(H', pk, S, C') \leftarrow \text{DecompressSK}(sk)$

2 : $\text{salt} \leftarrow \text{TRG.Seed}(2\lambda)$

3 : $\text{rseed} \leftarrow \text{TRG.Seed}(\lambda)$

// Step 1: Build & Commit to Witness Polynomials.

4 : $(\text{base}, v, h_{\text{sh}}, \text{key}, \text{aux}) \leftarrow \text{CommitWitnessPolynomials}(\text{salt}, \text{rseed}, S, C')$

// Step 2: Compute the polynomial proof $P_\alpha(X)$.

5 : $\Gamma \leftarrow \text{ExpandChallengeMatrix}(h_{\text{sh}}) \triangleright \Gamma \in \mathbb{F}_{q^\mu}^{\rho \times (mn-k)}$

6 : **for** e from 0 to $(\tau - 1)$ **do**

7 : $(\alpha_{\text{mid}}[e], \alpha_{\text{base}}[e]) \leftarrow \text{ComputePolynomialProof}(\text{base}[e], v[e]S, C', \Gamma, H')$

8 : $h_{\text{piop}} \leftarrow \text{Hash}_2(pk, \text{salt}, \text{msg}, h_{\text{sh}},$
 $\alpha_{\text{mid}}[0], \alpha_{\text{base}}[0], \dots, \alpha_{\text{mid}}[\tau - 1], \alpha_{\text{base}}[\tau - 1])$

// Step 3: Open random evaluations of the polynomials $P_S(X)$, $P_C(X)$ and $P_v(X)$.

9 : $(\text{ctr}, \pi_{\text{BAVC}}) \leftarrow \text{OpenRandomEvaluations}(\text{key}, h_{\text{piop}})$

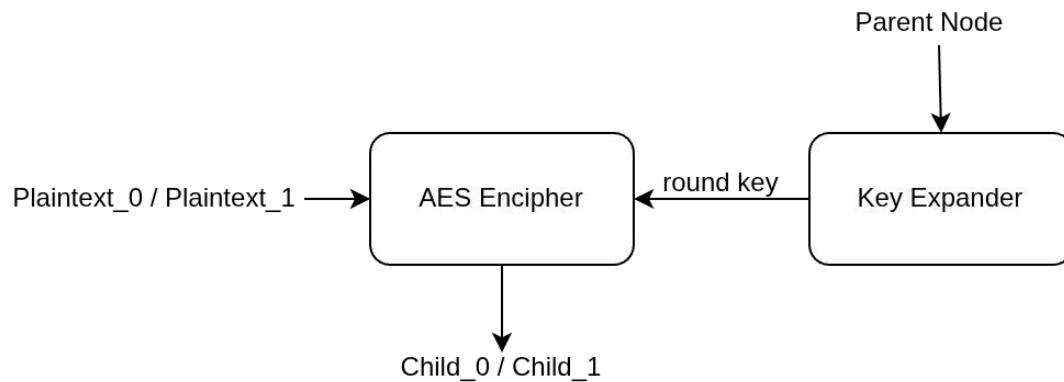
10 : $\sigma \leftarrow \text{UnparseSignature}(\text{salt}, \text{ctr}, h_{\text{piop}}, \pi_{\text{BAVC}}, \text{aux}, \alpha_{\text{mid}})$

11 : **return** σ

Optimizing GGM Tree Expansion For Hardware Platforms

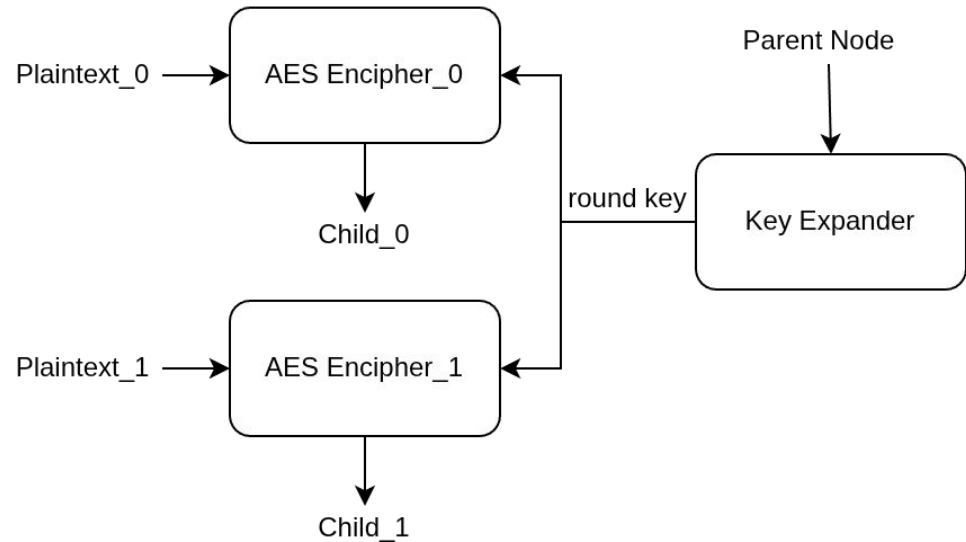
The AES Engine

- Start from a high-throughput (1 round/cycle) AES engine



The AES Engine

- Start from a high-throughput (1 round/cycle) AES engine
- Duplicate the encipher datapath = **2x throughput**



Our Proposed Expansion Algorithm



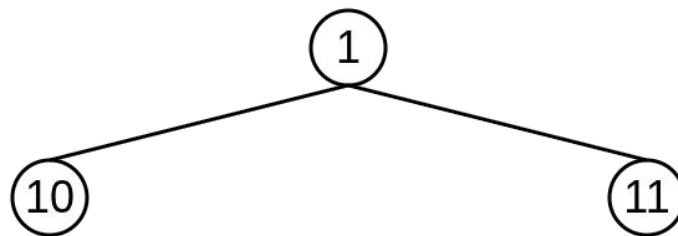
- Full GGM Tree = $2 \cdot N \cdot \tau - 1$ nodes = 139 to 590 KB: **Large memory** requirement for resource-constrained platforms
- Expanding the full tree before starting subsequent routines: **High latency** overhead
- Can we do better?

$$N = 256$$

$$\tau = \{17, 26, 36\}$$

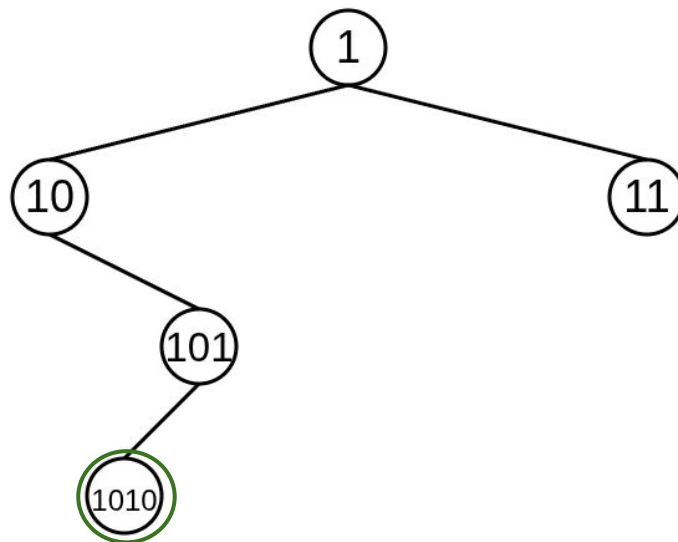
Our Proposed Expansion Algorithm

- Simple idea: Expand the full GGM tree **except the last K levels**
- Storage requirement reduces by a factor $\sim 2^K$



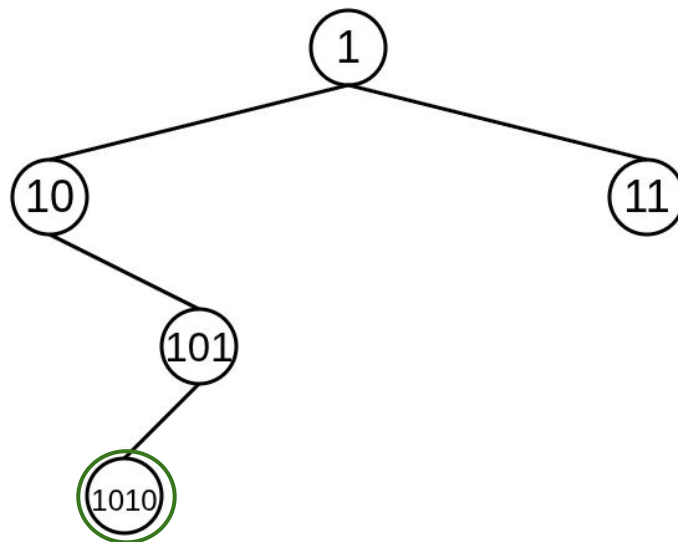
Our Proposed Expansion Algorithm

- Simple idea: Expand the full GGM tree **except the last K levels**
- Storage requirement reduces by a factor $\sim 2^K$
- Obtain a **leaf**: Expand a **small** part of the remaining tree *on the fly*



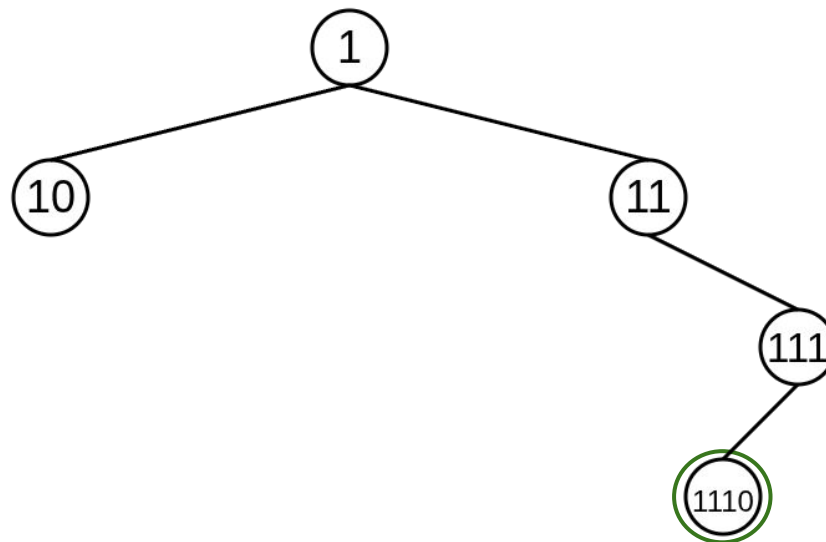
Our Proposed Expansion Algorithm

- Simple idea: Expand the full GGM tree **except the last K levels**
 - Storage requirement reduces by a factor $\sim 2^K$
 - Obtain a **leaf**: Expand a **small** part of the remaining tree *on the fly*
- Does not take advantage of our dual AES datapath



Our Proposed Expansion Algorithm

- Simple idea: Expand the full GGM tree **except the last K levels**
- Storage requirement reduces by a factor $\sim 2^K$
- Obtain a **leaf**: Expand a **small** part of the remaining tree *on the fly*
 - Does not take advantage of our dual AES datapath
 - Introduces **a lot** of recomputation
- Can we do better?



Our Proposed Expansion Algorithm

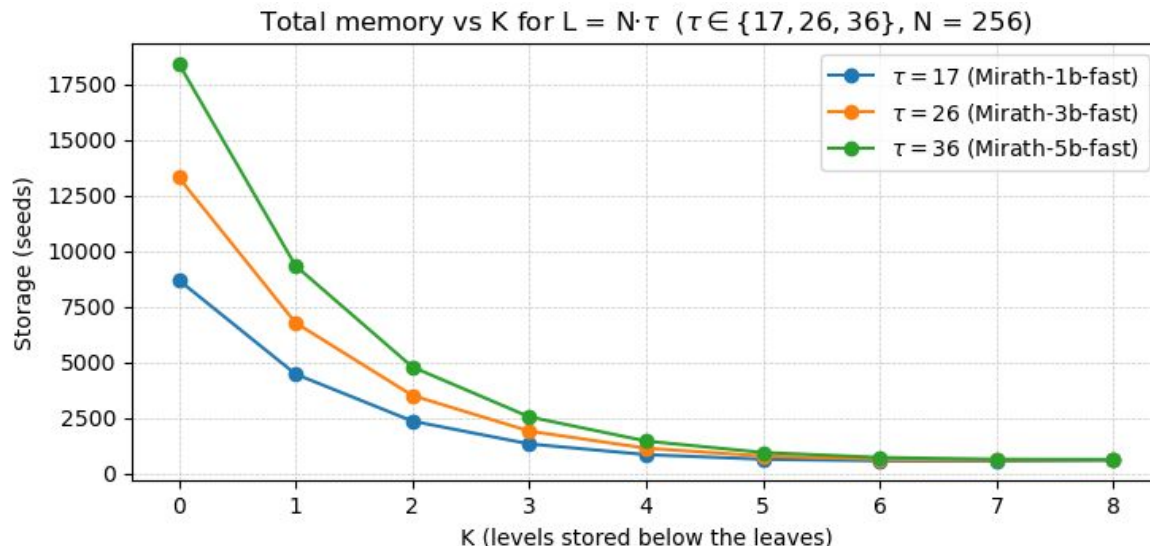
Instead: Use a hierarchy of **small runtime buffers**

- + takes full advantage of dual AES PRG
- + minimal memory overhead
- + near-zero recomputation

Our Proposed Expansion Algorithm

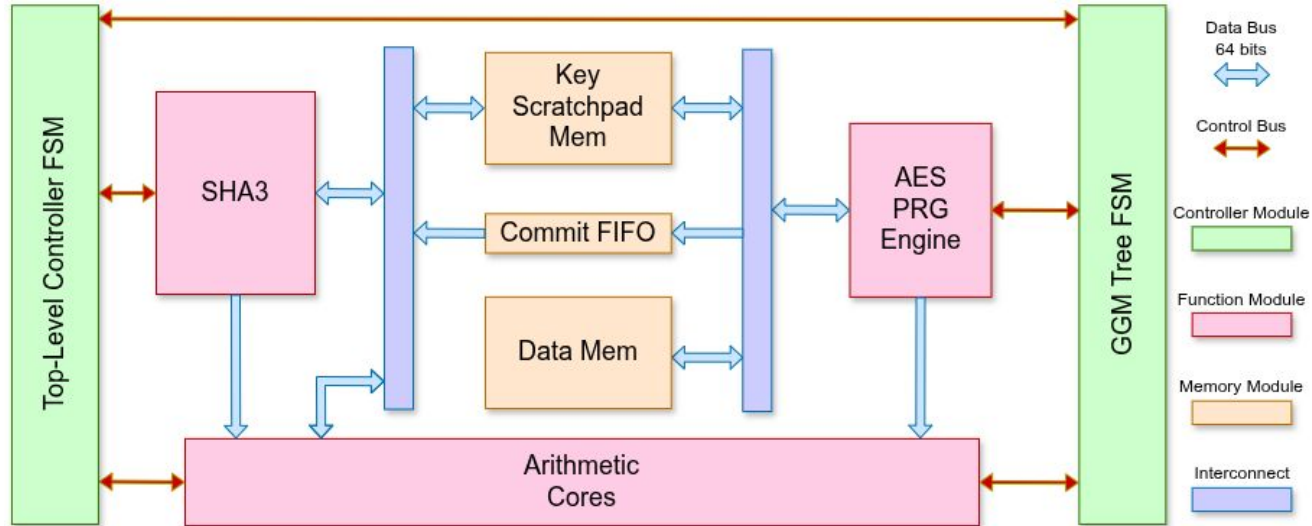
Instead: Use a hierarchy of **small runtime buffers**

- + takes full advantage of dual AES PRG
- + minimal memory overhead
- + near-zero recomputation



The Hardware Architecture

Top-Level Hardware Architecture



Scheduling of Operations

Mirath Sign Scheduling

- Most computationally expensive routine

Algorithm 22 Signing algorithm.

Sign(sk, msg)

// Step 0: Initialization.

1 : $(H', pk, S, C') \leftarrow \text{DecompressSK}(sk)$

2 : $salt \leftarrow \text{TRG.Seed}(2\lambda)$

3 : $rseed \leftarrow \text{TRG.Seed}(\lambda)$

// Step 1: Build $\mathcal{E}$ Commit to Witness Polynomials.

4 : $(base, v, h_{sh}, key, aux) \leftarrow \text{CommitWitnessPolynomials}(salt, rseed, S, C')$

// Step 2: Compute the polynomial proof $P_\alpha(X)$.

5 : $\Gamma \leftarrow \text{ExpandChallengeMatrix}(h_{sh}) \triangleright \Gamma \in \mathbb{F}_{q^\mu}^{\rho \times (mn-k)}$

6 : **for** e from 0 to $(\tau - 1)$ **do**

7 : $(\alpha_{mid}[e], \alpha_{base}[e]) \leftarrow \text{ComputePolynomialProof}(base[e], v[e], S, C', \Gamma, H')$

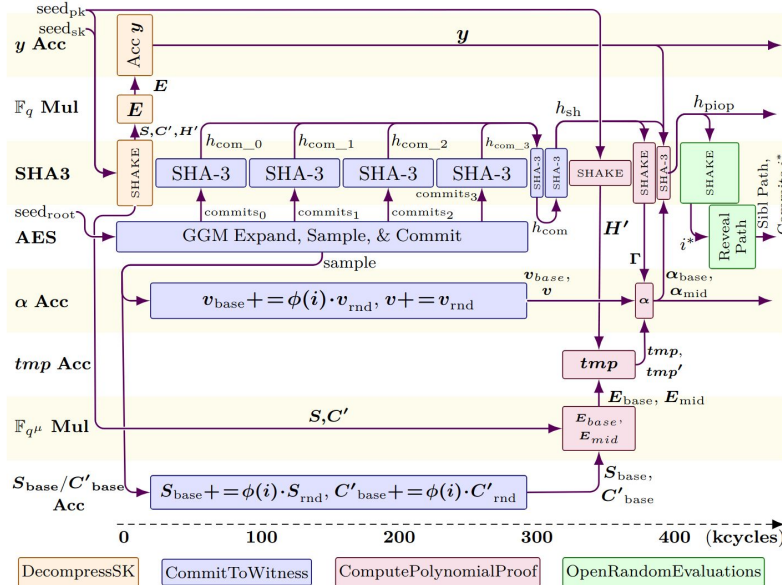
8 : $h_{piop} \leftarrow \text{Hash}_2(pk, salt, msg, h_{sh},$
 $\alpha_{mid}[0], \alpha_{base}[0], \dots, \alpha_{mid}[\tau - 1], \alpha_{base}[\tau - 1])$

// Step 3: Open random evaluations of the polynomials $P_S(X)$, $P_C(X)$ and $P_v(X)$.

9 : $(ctr, \pi_{BAVC}) \leftarrow \text{OpenRandomEvaluations}(key, h_{piop})$

10 : $\sigma \leftarrow \text{UnparseSignature}(salt, ctr, h_{piop}, \pi_{BAVC}, aux, \alpha_{mid})$

11 : **return** σ



Mirath Sign Scheduling

Step 0: Initialization

- Functionally equivalent to KeyGen (~2% total Sign latency)

Algorithm 22 Signing algorithm.

Sign(sk, msg)

// Step 0: Initialization.

```
1: ( $H'$ , pk,  $S$ ,  $C'$ )  $\leftarrow$  DecompressSK(sk)
2: salt  $\leftarrow$  TRG.Seed( $2\lambda$ )
3: rseed  $\leftarrow$  TRG.Seed( $\lambda$ )
```

// Step 1: Build $\mathcal{E}$ Commit to Witness Polynomials.

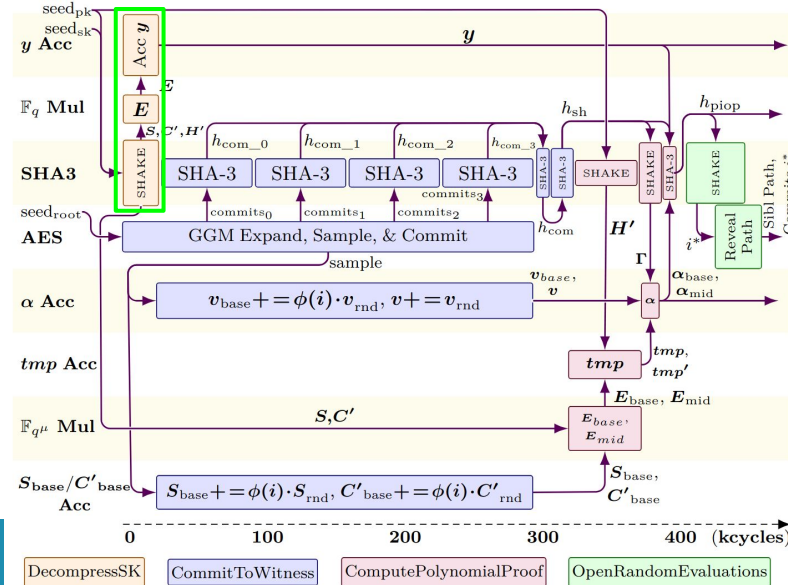
```
4: (base,  $v$ ,  $h_{sh}$ , key, aux)  $\leftarrow$  CommitWitnessPolynomials(salt, rseed,  $S$ ,  $C'$ )
```

// Step 2: Compute the polynomial proof $P_\alpha(X)$.

```
5:  $\Gamma \leftarrow$  ExpandChallengeMatrix( $h_{sh}$ )  $\triangleright \Gamma \in \mathbb{F}_{q^\mu}^{p \times (mn-k)}$ 
6: for  $e$  from 0 to  $(\tau - 1)$  do
7:   ( $\alpha_{mid}[e]$ ,  $\alpha_{base}[e]$ )  $\leftarrow$  ComputePolynomialProof(base[e],  $v[e]$ ,  $S$ ,  $C'$ ,  $\Gamma$ ,  $H'$ )
8:    $h_{piop} \leftarrow$  Hash2(pk, salt, msg,  $h_{sh}$ ,
       $\alpha_{mid}[0]$ ,  $\alpha_{base}[0]$ , ...,  $\alpha_{mid}[\tau - 1]$ ,  $\alpha_{base}[\tau - 1]$ )
```

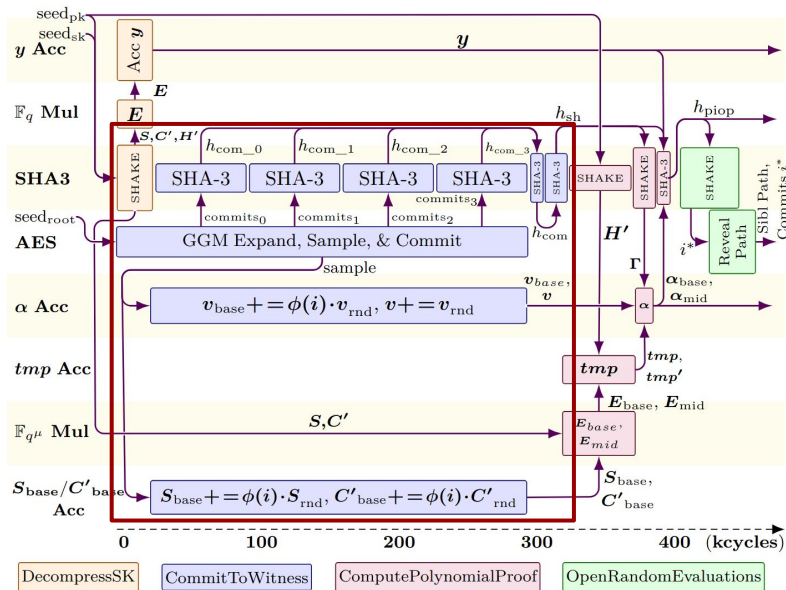
// Step 3: Open random evaluations of the polynomials $P_S(X)$, $P_C(X)$ and $P_v(X)$.

```
9: (ctr,  $\pi_{BAVC}$ )  $\leftarrow$  OpenRandomEvaluations(key,  $h_{piop}$ )
10:  $\sigma \leftarrow$  UnparseSignature(salt, ctr,  $h_{piop}$ ,  $\pi_{BAVC}$ , aux,  $\alpha_{mid}$ )
11: return  $\sigma$ 
```



Algorithm 22 Signing algorithm.

- GGM Tree expansion = repeated AES / Rijndael PRG ops (~60% of total Sign latency)
- Initial GGM expansion performed in parallel with *Step 0*
- Full GGM Tree = 139 to 590 KB: ***Expand on-the-fly***



// Step 0: Initialization.

$$1: (H', \text{pk}, S, C') \leftarrow \text{DecompressSK}(\text{sk})$$

```
2: salt  $\leftarrow$  TRG.Seed( $2\lambda$ )
```

```
3 : rseed  $\leftarrow$  TRG.Seed( $\lambda$ )
```

// Step 1: Build $\mathcal{E}$ Commit to Witness Polynomials.

4 : $(\text{base}, v, h_{\text{sh}}, \text{key}, \text{aux}) \leftarrow \text{CommitWitnessPolynomials}(\text{salt}, \text{rseed}, S, C')$

// Step 2: Compute the polynomial proof $P_\alpha(X)$.

$$5: \quad \mathbf{I} \leftarrow \text{ExpandChallengeMatrix}(h_{\text{sh}}) \quad \triangleright \mathbf{I} \in \mathbb{F}_{q^\mu}^{\rho \times (mn-k)}$$
6: **for** e from 0 to $(\tau - 1)$ **do**
$$7: \quad (\alpha_{\text{mid}}[e], \alpha_{\text{base}}[e]) \leftarrow \text{ComputePolynomialProof}(\text{base}[e], v[e]S, C', \Gamma, H')$$
$$8: h_{\text{piop}} \leftarrow \text{Hash}_2(\text{pk}, \text{salt}, \text{msg}, h_{\text{sh}}, \alpha_{\text{mid}}[0], \alpha_{\text{base}}[0], \dots, \alpha_{\text{mid}}[\tau-1], \alpha_{\text{base}}[\tau-1])$$

// Step 3: Open random evaluations of the polynomials $P_S(X)$, $P_C(X)$ and $P_v(X)$.

9 : $(\text{ctr}, \pi_{\text{BAVC}}) \leftarrow \text{OpenRandomEvaluations}(\text{key}, h_{\text{piop}})$

```

10 :  $\sigma \leftarrow \text{UnparseSignature}(\text{salt}, \text{ctr}, h_{\text{piop}}, \pi_{\text{BAVC}}, \text{aux}, \alpha_{\text{mid}})$ 

```

```

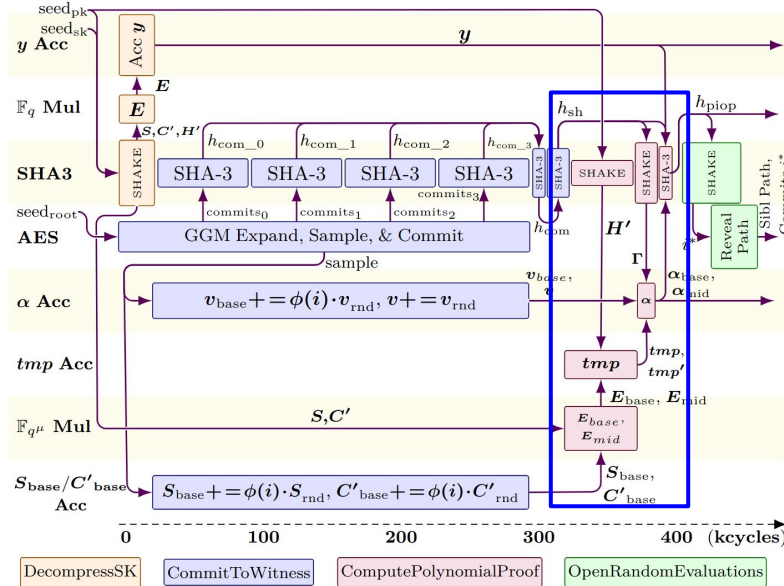
11 :   return  $\sigma$ 

```

Mirath Sign Scheduling

Step 2: Compute polynomial proof

- Linear algebra-heavy
- Small Field + Regular Ops = Efficient HW Acceleration (~13% of total Sign latency)
- All τ protocol iterations performed in parallel
- On-the-fly arithmetic: minimize on-chip memory & bandwidth req.



Algorithm 22 Signing algorithm.

Sign(sk, msg)

// Step 0: Initialization.

1: $(H', pk, S, C') \leftarrow \text{DecompressSK}(sk)$

2: $\text{salt} \leftarrow \text{TRG.Seed}(2\lambda)$

3: $\text{rseed} \leftarrow \text{TRG.Seed}(\lambda)$

// Step 1: Build & Commit to Witness Polynomials.

4: $(\text{base}, v, h_{\text{sh}}, \text{key}, \text{aux}) \leftarrow \text{CommitWitnessPolynomials}(\text{salt}, \text{rseed}, S, C')$

// Step 2: Compute the polynomial proof $P_\alpha(X)$.

5: $\Gamma \leftarrow \text{ExpandChallengeMatrix}(h_{\text{sh}}) \triangleright \Gamma \in \mathbb{F}_q^{\rho \times (mn-k)}$

6: **for** e from 0 to $(\tau - 1)$ **do**

7: $(\alpha_{\text{mid}}[e], \alpha_{\text{base}}[e]) \leftarrow \text{ComputePolynomialProof}(\text{base}[e], v[e], S, C', \Gamma, H')$

8: $h_{\text{piop}} \leftarrow \text{Hash}_2(pk, \text{salt}, \text{msg}, h_{\text{sh}},$
 $\alpha_{\text{mid}}[0], \alpha_{\text{base}}[0], \dots, \alpha_{\text{mid}}[\tau - 1], \alpha_{\text{base}}[\tau - 1])$

// Step 3: Open random evaluations of the polynomials $P_S(X)$, $P_C(X)$ and $P_v(X)$.

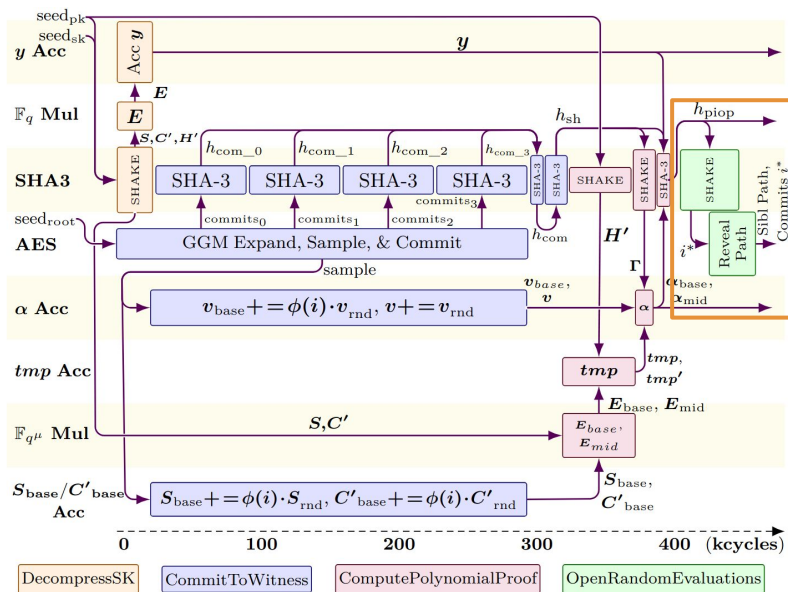
9: $(\text{ctr}, \pi_{\text{BAVC}}) \leftarrow \text{OpenRandomEvaluations}(\text{key}, h_{\text{piop}})$

10: $\sigma \leftarrow \text{UnparseSignature}(\text{salt}, \text{ctr}, h_{\text{piop}}, \pi_{\text{BAVC}}, \text{aux}, \alpha_{\text{mid}})$

11: **return** σ

Algorithm 22 Signing algorithm.

- *Repeatedly* sample hidden leaf indices
- *Sequential* in nature
- *Variable latency* (~25% of total Signing latency avg.)

 $\text{Sign}(\text{sk}, \text{msg})$

// Step 0: Initialization.

$$1: (H', \text{pk}, S, C') \leftarrow \text{DecompressSK}(\text{sk})$$

```
2: salt  $\leftarrow$  TRG.Seed( $2\lambda$ )
```

```
3 : rseed ← TRG.Seed( $\lambda$ )
```

// Step 1: Build & Commit to Witness Polynomials.

4 : $(\text{base}, v, h_{\text{sh}}, \text{key}, \text{aux}) \leftarrow \text{CommitWitnessPolynomials}(\text{salt}, \text{rseed}, S, C')$

// Step 2: Compute the polynomial proof $P_\alpha(X)$.

$$5: \quad \mathbf{I} \leftarrow \text{ExpandChallengeMatrix}(h_{\text{sh}}) \quad \triangleright \mathbf{I} \in \mathbb{F}_{a^\mu}^{\rho \times (mn-k)}$$
6 : **for** e from 0 to $(\tau - 1)$ **do**
$$7: \quad (\alpha_{\text{mid}}[e], \alpha_{\text{base}}[e]) \leftarrow \text{ComputePolynomialProof}(\text{base}[e], v[e]S, C', \Gamma, H')$$
$$8: h_{\text{piop}} \leftarrow \text{Hash}_2(\text{pk}, \text{salt}, \text{msg}, h_{\text{sh}}, \alpha_{\text{mid}}[0], \alpha_{\text{base}}[0], \dots, \alpha_{\text{mid}}[\tau-1], \alpha_{\text{base}}[\tau-1])$$

// Step 3: Open random evaluations of the polynomials $P_S(X)$, $P_C(X)$ and $P_v(X)$.

9 : $(\text{ctr}, \pi_{\text{BAVC}}) \leftarrow \text{OpenRandomEvaluations}(\text{key}, h_{\text{piop}})$

```

10 :  $\sigma \leftarrow \text{UnparseSignature}(\text{salt}, \text{ctr}, h_{\text{piop}}, \pi_{\text{BAVC}}, \text{aux}, \alpha_{\text{mid}})$ 

```

```
11: return  $\sigma$ 
```

Comparison with Related Work

Comparison with Software Implementations

Table 1: Comparison of **Mirath** hardware and software implementations.

Scheme	Parameter Set	Freq. (MHz)	KeyGen (Mcycles/ms)	Sign (Mcycles/ms)	Verify (Mcycles/ms)
Mirath (This work)	1b-fast	254	0.0161/0.0634	0.554/2.181	0.5/1.969
	5b-fast	218	0.0607/0.2784	1.3726/6.296	1.2116/5.558
Mirath (SW)^a [AAB⁺25a, AAB⁺25b]	1b-fast	3000	1.8/0.60	1600/533.3	2600/866.67
	5b-fast	3000	6.5/2.17	8400/2800.0	13000/4333.3
	1b-fast-AVX2	3000	0.5/0.17	9.8/3.27	5.5/1.83
	5b-fast-AVX2	3000	2.0/0.67	70.1/23.37	51.7/17.23

^a Intel i9-13900K (3GHz), 64GB RAM, GCC v11.4.0.

14.8–46.8× cycle reduction & **1.23–3.9×** speedup vs optimized AVX2

ATP Comparison (NIST Add. DSAs Round 2)

Scheme	Parameter Set	Resources		Freq. (MHz)	KeyGen (Mcycles/ms)	Sign (Mcycles/ms)	Verify (Mcycles/ms)	ATP ^a
		LUT/FF/DSP/BRAM	Area ^a					
Mirath (This work)	1b-fast	15,400/9,445/0/42.0	28,000	254	0.0161/0.0634	0.554/2.181	0.5/1.969	118.0
	5b-fast	30,415/18,223/0/96.0	59,215	218	0.0607/0.2784	1.3726/6.296	1.2116/5.558	718.4
CROSS [KAB ⁺ 25]	RSDP-1-f	26,741/11,678/0/45.0	40,241	118	−/0.035	−/0.478	−/0.409	37.1
	RSDP-5-f	34,626/15,331/0/91.5	62,076	113	−/0.028	−/1.20	−/1.14	147.0
LESS [BWMG23]	L1-s	54,800/39,900/0/59.5	72,650	200	0.174/0.87	4.17/20.83	4.137/20.69	3,709.6
	L5-s	104,300/76,700/0/167.5	154,550	143	0.247/1.73	87.16/610.13	87.013/609.09	188,697.8
MAYO [HSK ⁺ 24]	L1-f	106,000/38,000/2/45.5	119,850	75	0.0058/0.08	0.031/0.43	0.0046/0.05	67.1
UOV [BCD ⁺ 25]	Is-pkc+skc	28,947/24,444/2/66.0	48,947	90.8	11.1/122.25	0.84/9.251	0.284/3.128	6,589.7
	V-pkc+skc	77,352/38,217/4/359.0	185,452	92.5	39.1/422.70	3.31/35.784	1.92/20.757	88,876.2

^a Area-Time-Product (ATP), with area measured as $\#LUTs + 100 \times \#DSPs + 300 \times \#BRAMs$ [YCH22].

ATP Comparison (Previous MPCitH-based HW)

Scheme	Parameter Set	Resources		Freq. (MHz)	KeyGen (Mcycles/ms)	Sign (Mcycles/ms)	Verify (Mcycles/ms)	ATP ^a
		LUT/FF/DSP/BRAM	Area ^a					
Mirath (This work)	1b-fast	15,400/9,445/0/42.0	28,000	254	0.0161/0.0634	0.554/2.181	0.5/1.969	118.0
	5b-fast	30,415/18,223/0/96.0	59,215	218	0.0607/0.2784	1.3726/6.296	1.2116/5.558	718.4
SDitH [DHSY24]	L1-GF256	16,592/8,778/0/164.5	65,942	164	0.055/0.34	6.73/41.04	8.688/52.98	6,222.3
	L5-GF256	23,323/14,962/0/520.5	179,473	164	0.083/0.51	17.48/106.60	24.940/152.10	46,521.2
PICNIC [KRR ⁺ 20]	L1	90,337/23,105/0/52.5	106,087	125	—/—	0.03/0.25	0.030/0.24	52.0
	L5	167,530/33,164/0/98.5	197,080	125	—/—	0.15/1.24	0.147/1.17	475.0

^a Area-Time-Product (ATP), with area measured as $\#LUTs + 100 \times \#DSPs + 300 \times \#BRAMs$ [YCH22].

Conclusions

Summary



- First hardware implementation of a Round-2 NIST Additional Signatures MPCitH-based DSA
- An optimized method for memory-efficient on-the-fly GGM tree expansion
- High-throughput, low-footprint MPC arithmetic units
- On-the-fly arithmetic scheduling that significantly reduces on-chip memory requirements
- High-throughput SHA3 / AES engines with stall-minimized scheduling
- **14.8–46.8×** cycle reduction & **1.23–3.9×** speedup compared to optimized AVX2
- Design methodology & optimizations extendable to other MPCitH DSAs

Future Work

- State-of-the-art MPCitH-based signatures can be efficiently implemented in hardware while achieving competitive performance
- Required runtime flexibility (GGM Tree) still limits hardware efficiency and increases design complexity

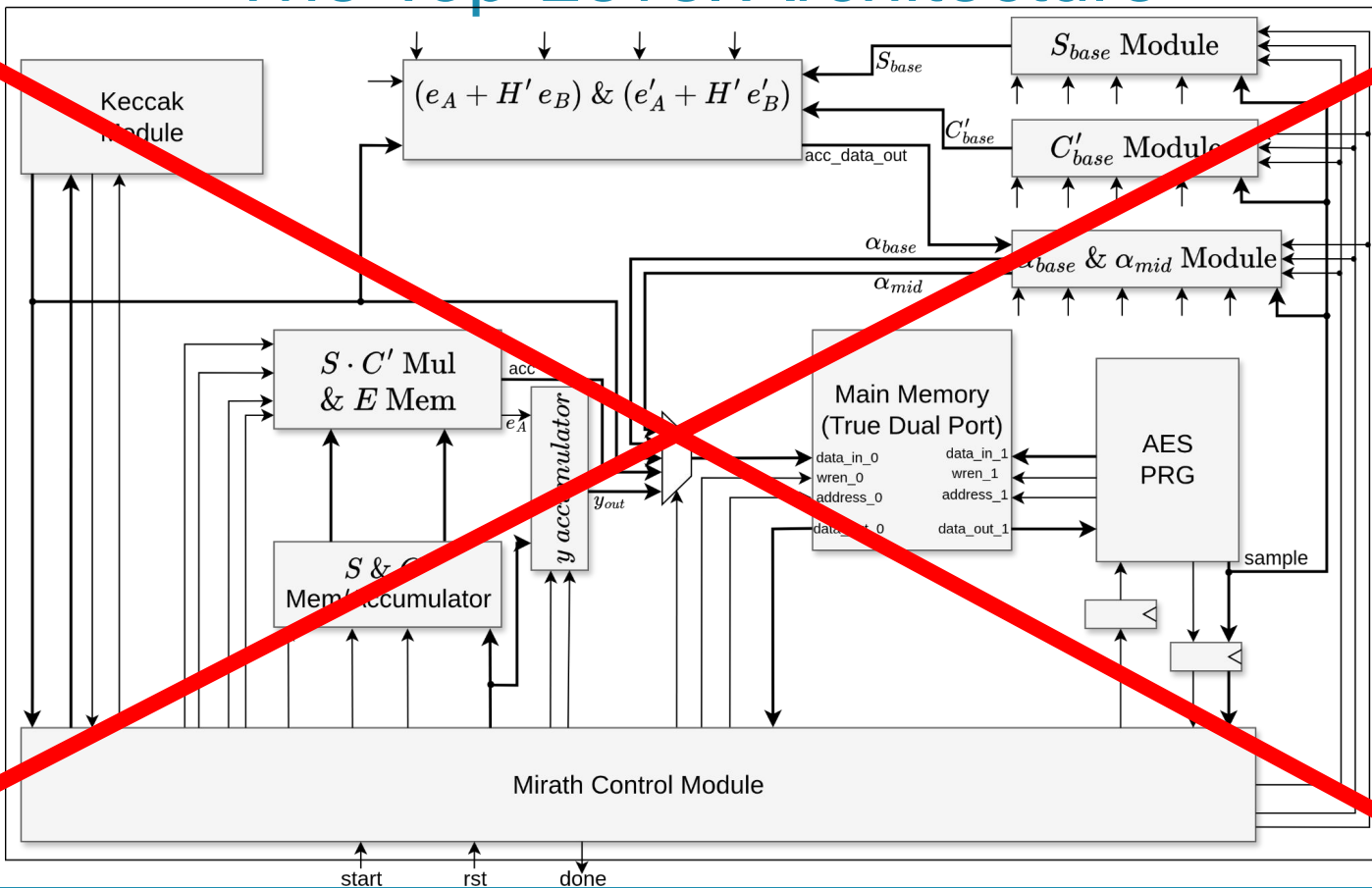
RTL open-source & available @ <https://github.com/KULeuven-COSIC/MIRATH-in-HW/>
Paper accepted at TCHES & available @ <https://eprint.iacr.org/2026/206>

Thank you for your attention!

Questions ?

Old Slides

The Top-Level Architecture



What about the MPC arithmetic?

Commit phase:

```

for  $e \leftarrow 0$  to  $\tau - 1$  do                                 $\tau \in [11, 36]$ 
     $(S_{\text{acc}}, C'_{\text{acc}}, v_{\text{acc}}) \leftarrow (0, 0, 0)$            $N \in \{256, 4096\}$ 
     $(S_{\text{base}}, C'_{\text{base}}, v_{\text{base}}) \leftarrow (0, 0, 0)$ 
    for  $i \leftarrow 0$  to  $N - 1$  do
         $(S_{\text{rnd}}, C'_{\text{rnd}}, v_{\text{rnd}}) \leftarrow \text{ExpandSeedShares}(\text{salt}, \text{seeds}[e][i])$ 
         $S_{\text{acc}} \leftarrow S_{\text{acc}} + S_{\text{rnd}}$ 
         $C'_{\text{acc}} \leftarrow C'_{\text{acc}} + C'_{\text{rnd}}$ 
         $v_{\text{acc}} \leftarrow v_{\text{acc}} + v_{\text{rnd}}$ 
         $S_{\text{base}} \leftarrow S_{\text{base}} - \phi(i) \cdot S_{\text{rnd}}$ 
         $C'_{\text{base}} \leftarrow C'_{\text{base}} - \phi(i) \cdot C'_{\text{rnd}}$ 
         $v_{\text{base}} \leftarrow v_{\text{base}} - \phi(i) \cdot v_{\text{rnd}}$ 
  
```

What about the MPC arithmetic?

MPC party emulation:

$\tau \in [11, 36]$

for $e \leftarrow 0$ **to** $(\tau - 1)$ **do**

$E_{\text{base}} \leftarrow [0_{m \times r} \mid S_{\text{base}} \cdot C'_{\text{base}}]$

$e \leftarrow \text{vec}(E_{\text{base}})$

$(e_A, e_B) \leftarrow \text{Split}(e)$

$\alpha_{\text{base}} \leftarrow \Gamma(e_A + H'e_B) + v_{\text{base}}$

$E_{\text{mid}} \leftarrow [S_{\text{base}} \mid S_{\text{base}}C' + SC'_{\text{base}}]$

$e' \leftarrow \text{vec}(E_{\text{mid}})$

$(e'_A, e'_B) \leftarrow \text{Split}(e')$

$\alpha_{\text{mid}} \leftarrow \Gamma(e'_A + H'e'_B) + v$

Implementing the Arithmetic of Mirath in Hardware

Arithmetic in Mirath



Need to perform intensive matrix operations with high-throughput

Aims of the arithmetic modules:

- Small area footprint per-module -> Compactness & ease of parallelizability
- Flexibility for reuse across many subroutines
- High performance

Proposed Arithmetic Modules

Multiple custom modules presented in this work.

Here, a detailed description of a single module is provided.

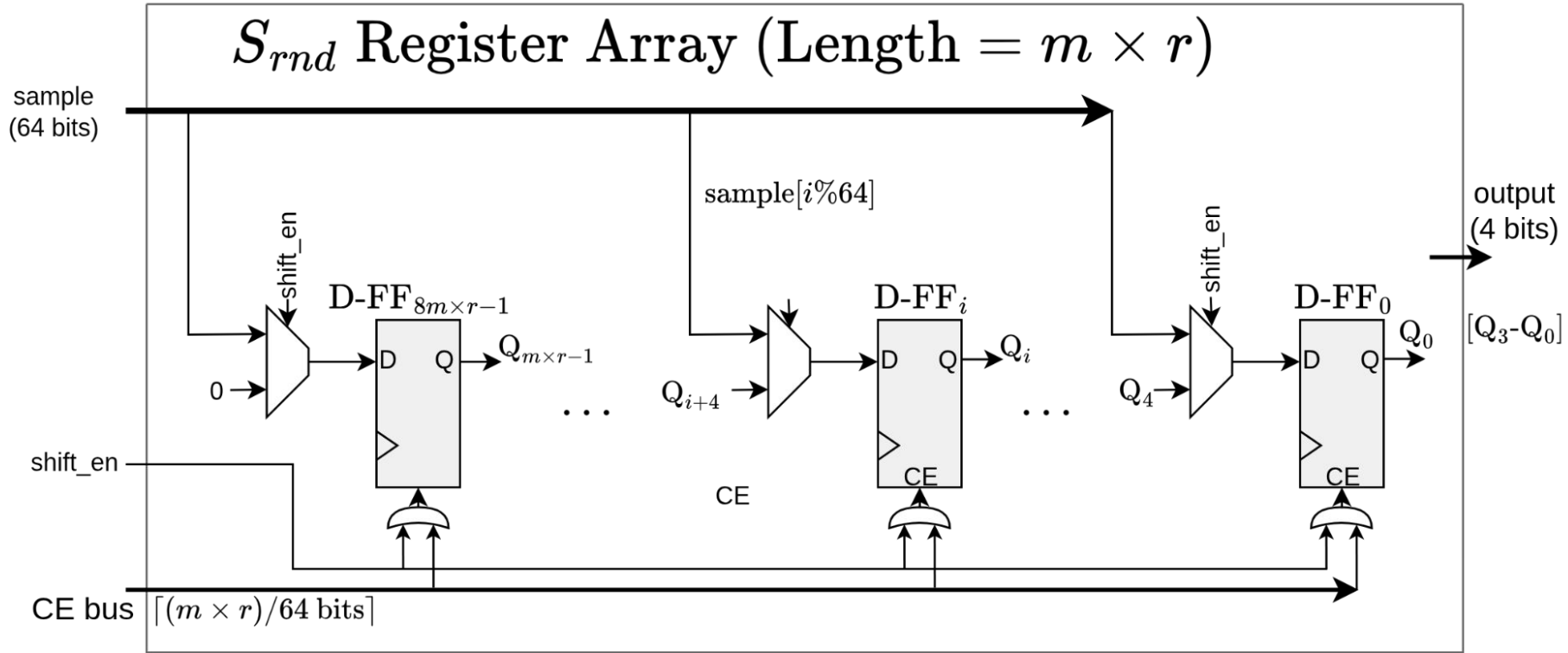
The S_{base} accumulator:

1. First stage in the arithmetic pipeline
2. Relatively simple, yet provides intuition into the implementation

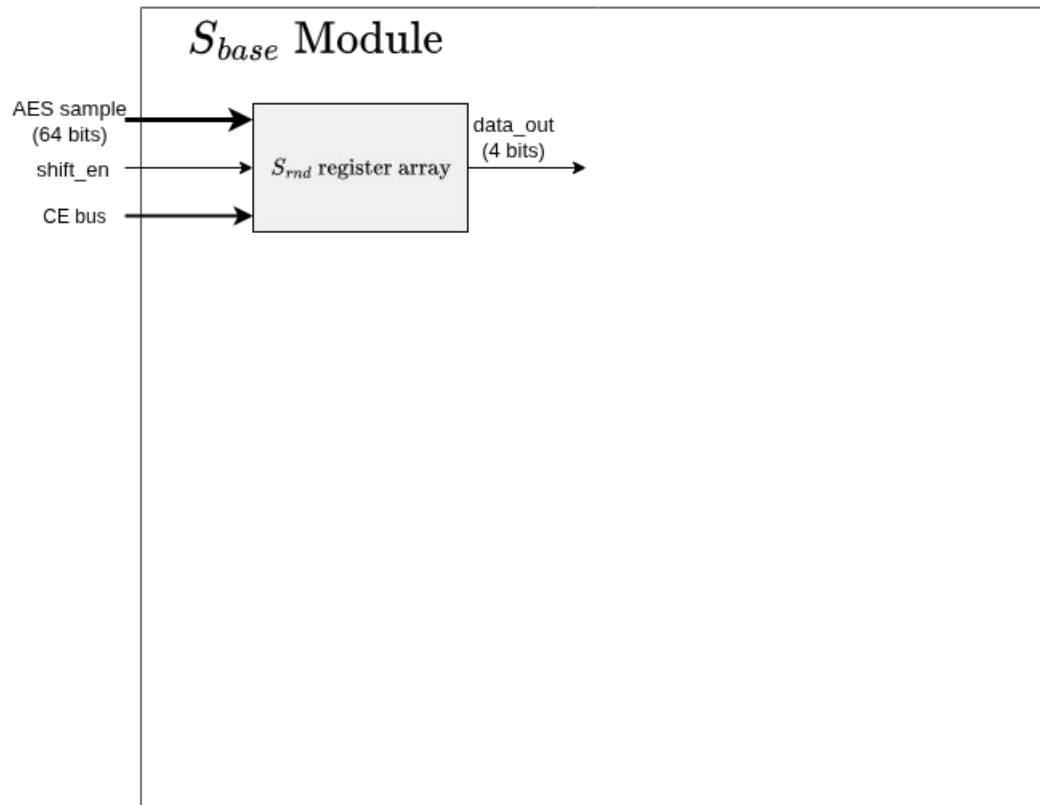
Recap: Commit Phase

```
for  $e \leftarrow 0$  to  $\tau - 1$  do
   $(S_{\text{acc}}, C'_{\text{acc}}, v_{\text{acc}}) \leftarrow (0, 0, 0)$ 
   $(S_{\text{base}}, C'_{\text{base}}, v_{\text{base}}) \leftarrow (0, 0, 0)$ 
  for  $i \leftarrow 0$  to  $N - 1$  do
     $(S_{\text{rnd}}, C'_{\text{rnd}}, v_{\text{rnd}}) \leftarrow \text{ExpandSeedShares}(\text{salt}, \text{seeds}[e][i])$ 
     $S_{\text{acc}} \leftarrow S_{\text{acc}} + S_{\text{rnd}}$ 
     $C'_{\text{acc}} \leftarrow C'_{\text{acc}} + C'_{\text{rnd}}$ 
     $v_{\text{acc}} \leftarrow v_{\text{acc}} + v_{\text{rnd}}$ 
     $S_{\text{base}} \leftarrow S_{\text{base}} - \phi(i) \cdot S_{\text{rnd}}$ 
     $C'_{\text{base}} \leftarrow C'_{\text{base}} - \phi(i) \cdot C'_{\text{rnd}}$ 
     $v_{\text{base}} \leftarrow v_{\text{base}} - \phi(i) \cdot v_{\text{rnd}}$ 
```

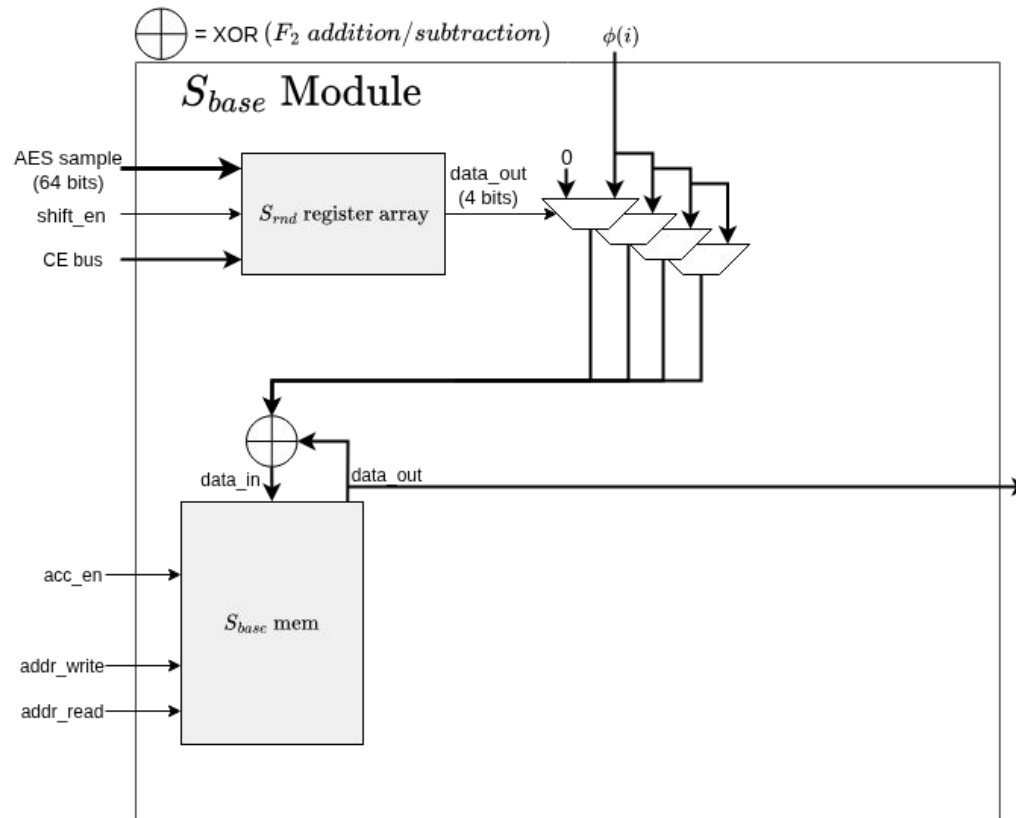
Capturing the Incoming Sample



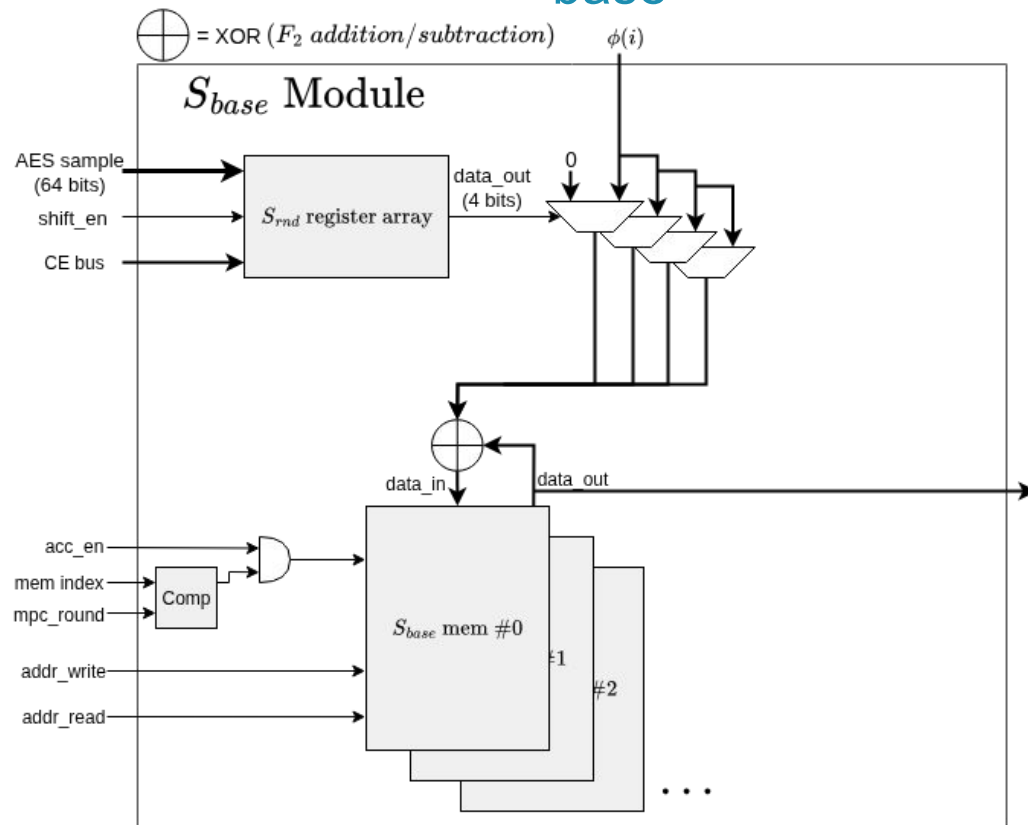
Building the S_{base} Module



Building the S_{base} Module



Building the S_{base} Module



Recap: MPC Emulation

During the *MPC party emulation*:

for $e \leftarrow 0$ **to** $(\tau - 1)$ **do**

$\tau \in [11, 36]$

$E_{\text{base}} \leftarrow [0_{m \times r} \mid S_{\text{base}} \cdot C'_{\text{base}}]$

$e \leftarrow \text{vec}(E_{\text{base}})$

$(e_A, e_B) \leftarrow \text{Split}(e)$

$\alpha_{\text{base}} \leftarrow \Gamma(e_A + H'e_B) + v_{\text{base}}$

$E_{\text{mid}} \leftarrow [S_{\text{base}} \mid S_{\text{base}}C' + SC'_{\text{base}}]$

$e' \leftarrow \text{vec}(E_{\text{mid}})$

$(e'_A, e'_B) \leftarrow \text{Split}(e')$

$\alpha_{\text{mid}} \leftarrow \Gamma(e'_A + H'e'_B) + v$

$H', \Gamma, E_{\text{base}}, E_{\text{mid}}$:

largest dimensions amongst all matrices in Mirath

Computing E_{mid}

During the *MPC party emulation*:

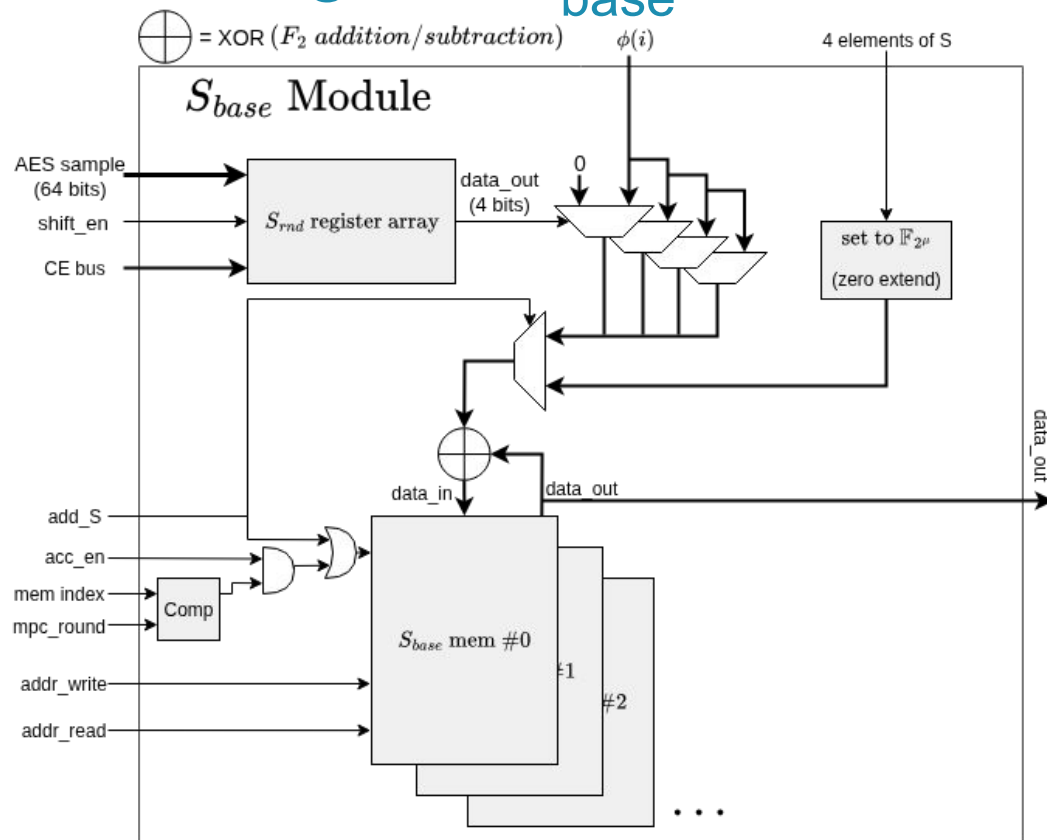
$$E_{\text{mid}} \leftarrow [S_{\text{base}} \mid S_{\text{base}}C' + SC'_{\text{base}}] \quad (2 \text{ Multiplications})$$

Instead:

$$S_{\text{base}}C' + SC'_{\text{base}} \leftarrow (S_{\text{base}} + S)(C'_{\text{base}} + C') - S_{\text{base}}C'_{\text{base}} - SC'$$

We already have $S_{\text{base}}C'_{\text{base}}$ and SC' from previous routines!

Building the S_{base} Module



Future Work

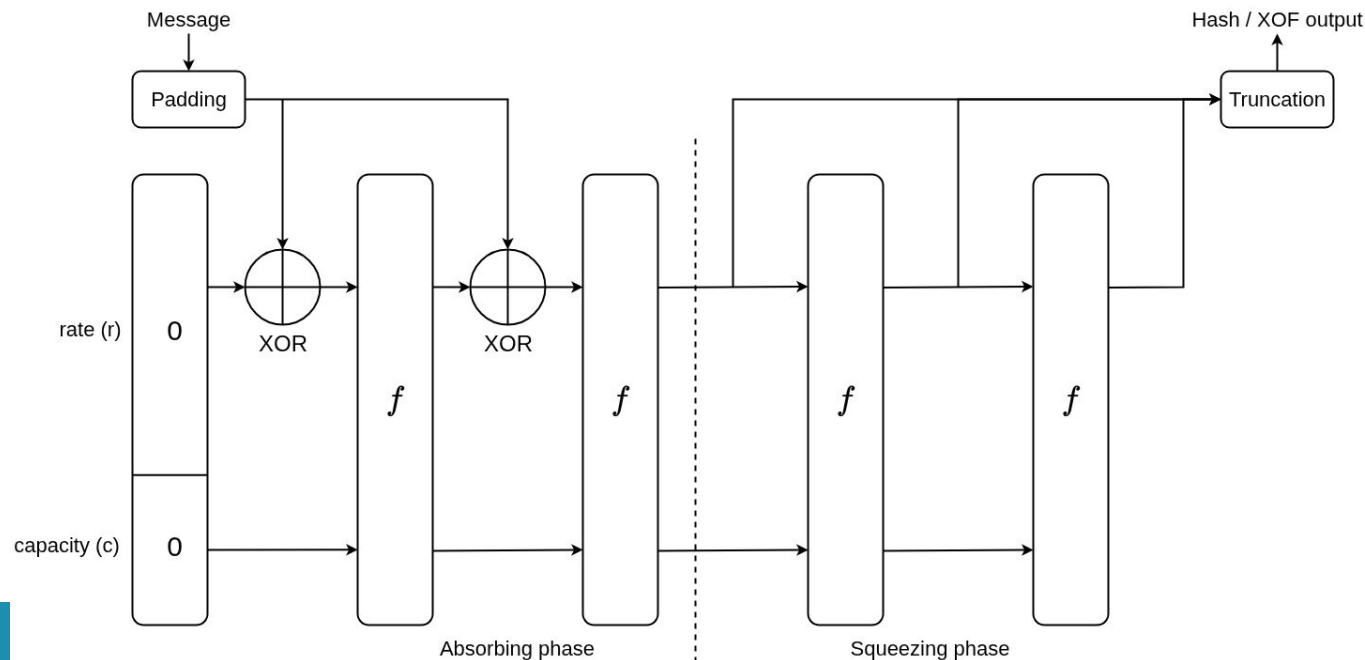
Possible areas for future improvement:

- Separating clock domains
- Adapting the top-level FSM for further subroutine parallelization

Keccak-1600 functions in Mirath

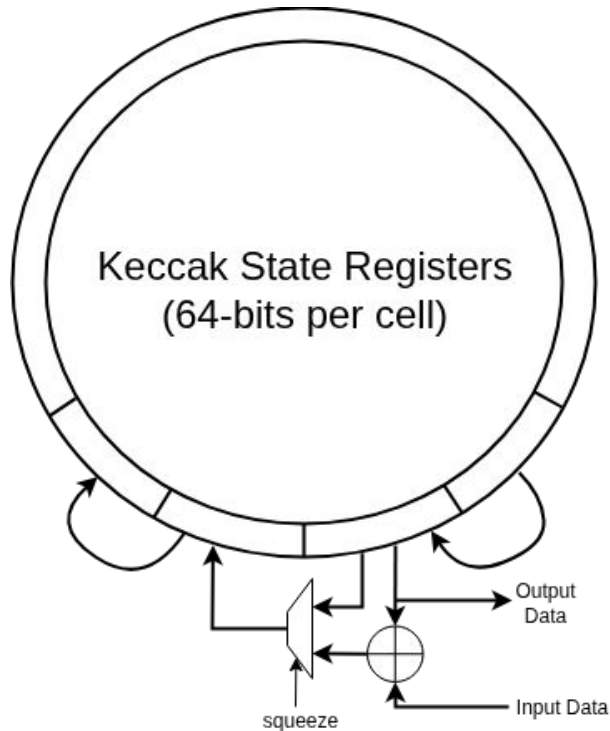
Mirath relies on the SHA-3 and SHAKE functions for hashing and pseudorandomness generation.

Both of which belong to the Keccak-1600 family.



The Main Datapath

The operation of the module relies on a circular shift register array:



- A single read/write location
- No memory addressing or clock enable overhead

Enabling Parallel Hashing Contexts

In the *commit* phase:

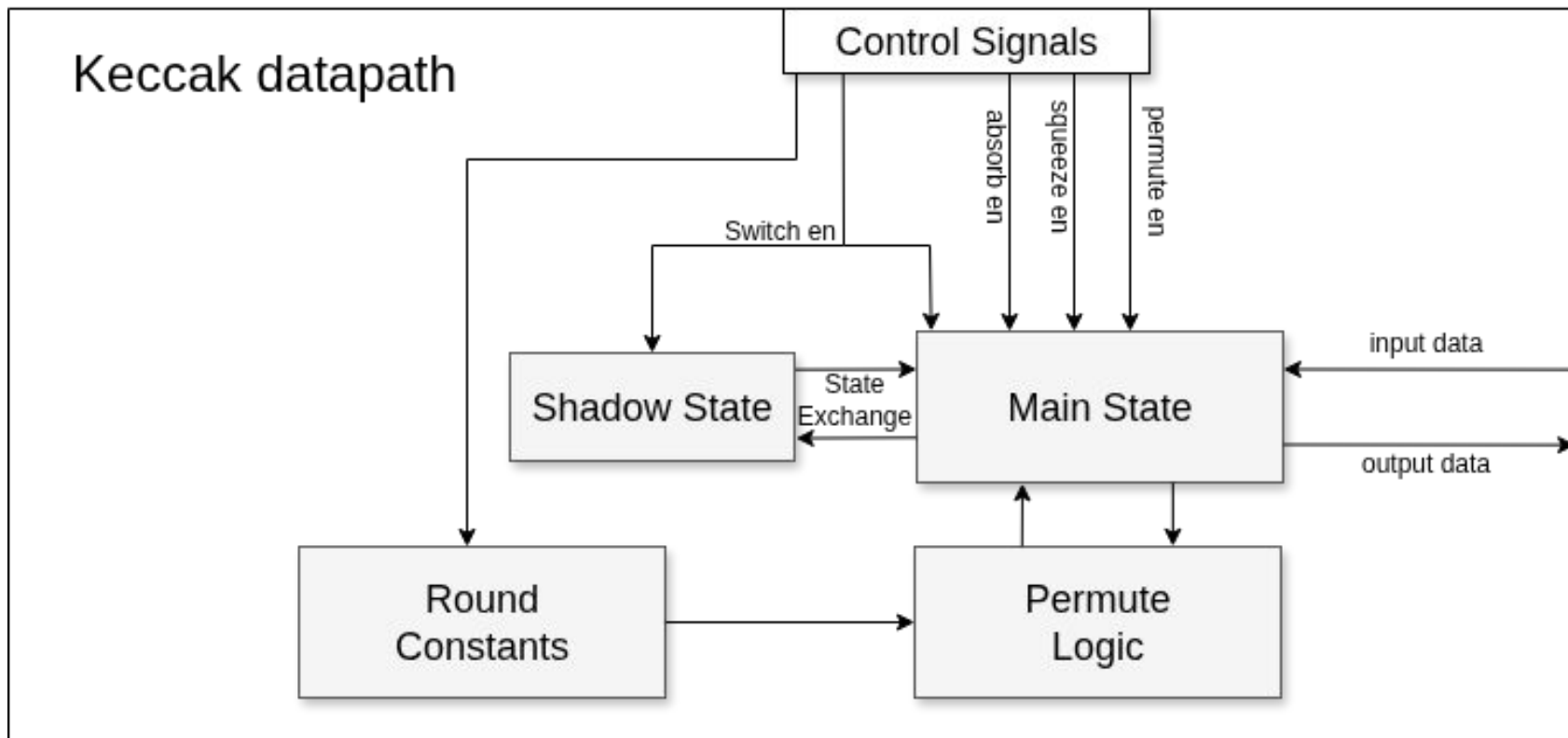
- Each party's share is hashed (commit to share).
- All commits hashed together to obtain h_{com} .

Total size of commits:

- ~140 KB for the smallest parameter set.
- ~6 MB for the largest!

Compute h_{com} on the fly: enter the “**Shadow state**”.

The Complete Keccak Datapath



MPC in-the-Head

Given: MPC Protocol

Goal: zero-knowledge proof knowledge

(Prover want to prove knowledge of x s.t. $f(x) = z$ to Verifier)

MPC in-the-Head

Given: MPC Protocol

Goal: zero-knowledge proof knowledge
(Prover want to prove knowledge of x s.t. $f(x) = z$ to Verifier)

Prover

Prepare MPC inputs and
commit

Com1



Verifier

MPC in-the-Head

Given: MPC Protocol

Goal: zero-knowledge proof knowledge

(Prover want to prove knowledge of x s.t. $f(x) = z$ to Verifier)

Prover

Prepare MPC inputs and
commit

Com1

R

Verifier

Sample first challenge R

MPC in-the-Head

Given: MPC Protocol

Goal: zero-knowledge proof knowledge
(Prover want to prove knowledge of x s.t. $f(x) = z$ to Verifier)

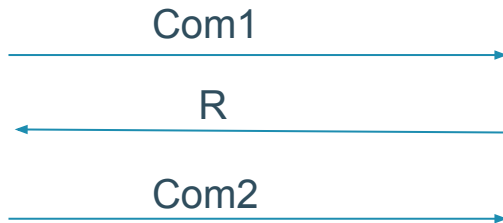
Prover

Prepare MPC inputs and
commit

Use R to simulate MPC
protocol, then commit

Verifier

Sample first challenge R



MPC in-the-Head

Given: MPC Protocol

Goal: zero-knowledge proof knowledge
(Prover want to prove knowledge of x s.t. $f(x) = z$ to Verifier)

Prover

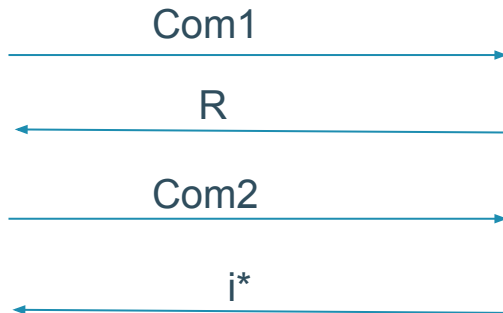
Prepare MPC inputs and
commit

Use R to simulate MPC
protocol, then commit

Verifier

Sample first challenge R

Sample second challenge i^*



MPC in-the-Head

Given: MPC Protocol

Goal: zero-knowledge proof knowledge
 (Prover want to prove knowledge of x s.t. $f(x) = z$ to Verifier)

Prover

Prepare MPC inputs and
commit

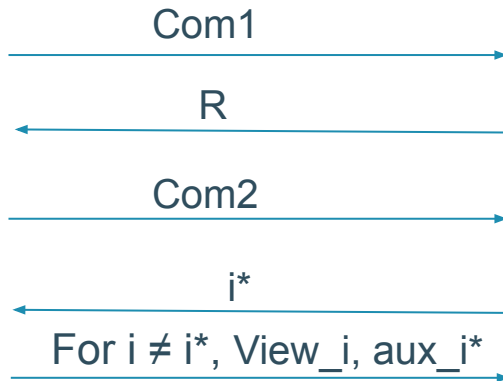
Use R to simulate MPC
protocol, then commit

Reveal all views of parties
 $P_i, i \neq i^*$

Verifier

Sample first challenge R

Sample second challenge i^*



MPC in-the-Head

Given: MPC Protocol

Goal: zero-knowledge proof knowledge
 (Prover want to prove knowledge of x s.t. $f(x) = z$ to Verifier)

Prover

Prepare MPC inputs and
commit

Use R to simulate MPC
protocol, then commit

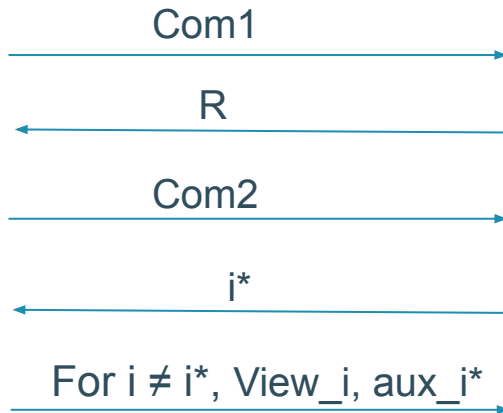
Reveal all views of parties
 $P_i, i \neq i^*$

Verifier

Sample first challenge R

Sample second challenge i^*

Check validity of all commitments



MPC in-the-Head

Given: MPC Protocol

Goal: zero-knowledge proof knowledge

(Prover want to prove knowledge of x s.t. $f(x) = z$ to Verifier)

Prover

Prepare MPC input
commit

Use R to simulate
protocol, then commit

Reveal all views of parties

$P_i, i \neq i^*$

Convert to Signature scheme via
Fiat-Shamir

Verifier

Sample first challenge R

Sample second challenge i^*

For $i \neq i^*$, View _{i} , aux _{i^*}

Check validity of all commitments

The Mirath Signature Scheme



In practice:

Mirath's arithmetic works with a large number of matrices and vectors.

Total size of matrices and vectors:

- ~420 KB for the smallest parameter set
- ~20 MB for the largest

What about the MPC arithmetic?

To achieve high performance, *efficient* and *fast* arithmetic modules are required.

Additional Challenges?

Matrix dimensions are not hardware-friendly.

$$y = H \cdot \text{vec}(E)$$
$$H \rightarrow 321 \times 1443$$
$$\text{vec}(E) \rightarrow 1443 \times 1$$

Next Few Slides Overview

These I consider good slides in general, especially in terms of how they convey the hardware intuition for the thesis design. The problem is that the design itself is not really that relevant for the accelerated version I am working on now. What do you think? **Keep?** **Change?** **Remove altogether?**

Extra point

Hard to compare directly for inherently faster schemes like Dilithium, that are much more regular and allow more hardware optimizations.

Still, the thesis design achieves almost the same (to provide an exact ratio) HW/SW speed-up ratio that the (much more mature) Dilithium design achieves compared to its own reference algo. Current work aims to *greatly* improve the ratio (2x or more).

The AES PRG Module: To discuss

A couple of slides: Compare tree expansion algorithm for the software and hardware versions; On-the-fly is much more complicated to work with (as compared with simple binary heap expansion in software).

During the signature: Can we avoid having to build a full sorted list but still provide the revealed nodes in a sorted manner (for compatibility with the SW version) using a hardware-efficient implementation? YES! Show how we do it (a couple of slides).

During the verification: Partial GGM tree expansion is a bit more complicated than the full expansion, but this can be solved quite easily for software. For the on-the-fly version for hardware, this does become a challenge. Explain design directions in 3-4 slides.

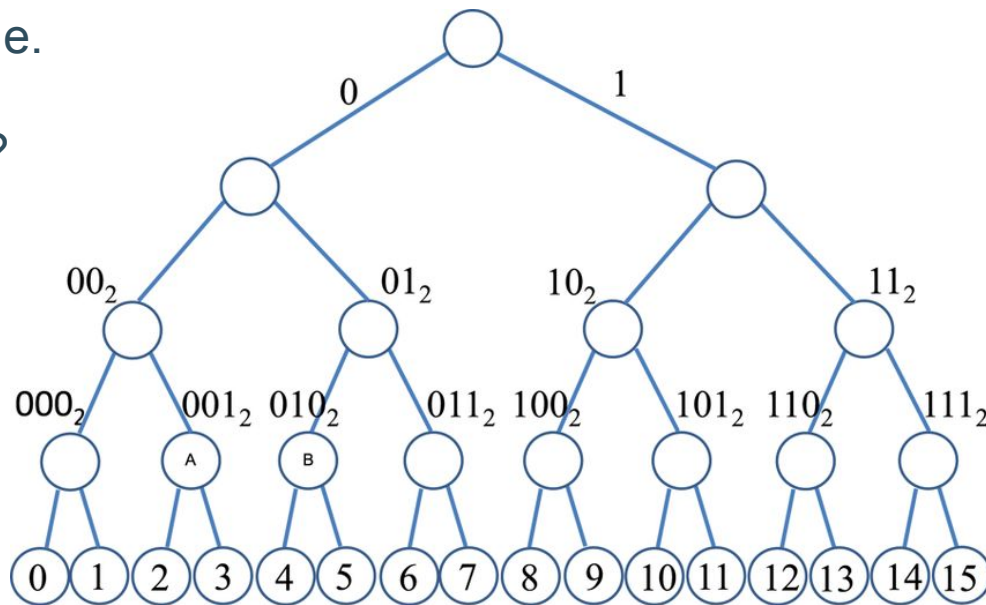
Commit Phase: Hashing each share

Each individual party's share has to be hashed to obtain a value called a **commit**.

1 commit: 2x the size of a tree node.

Should we store all $\tau * N$ commits?

Store first, process later?
Huge latency!

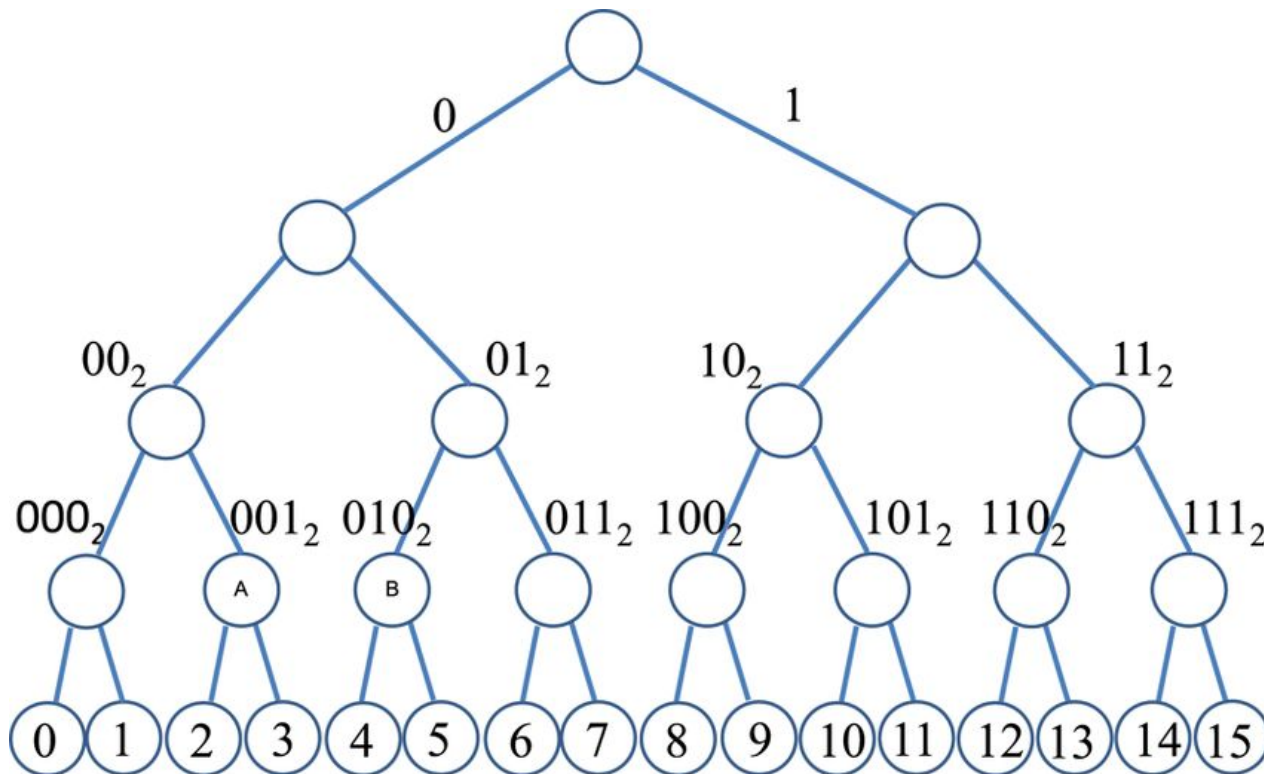


All-but-one share opening

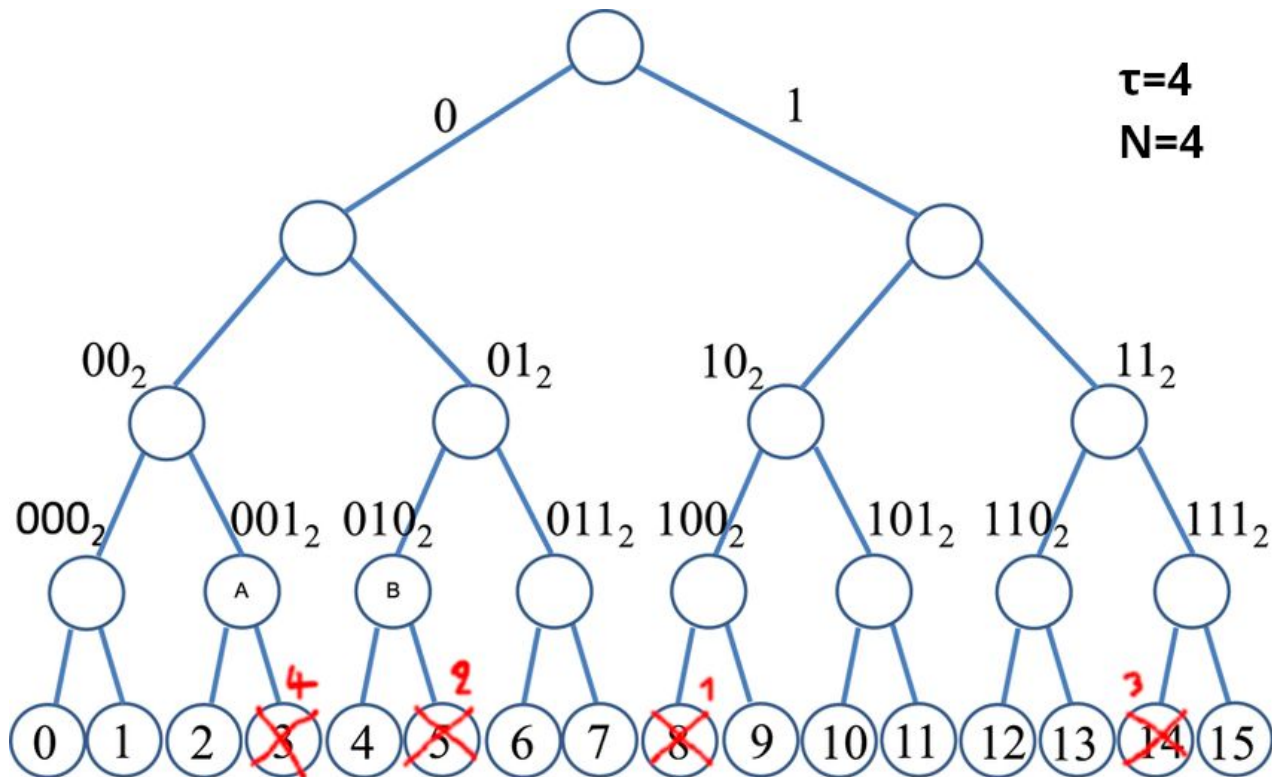
Revealing any $N-1$ party shares from each MPC round leaks no secret information.

The verifier can use these shares to evaluate the validity of the signature.

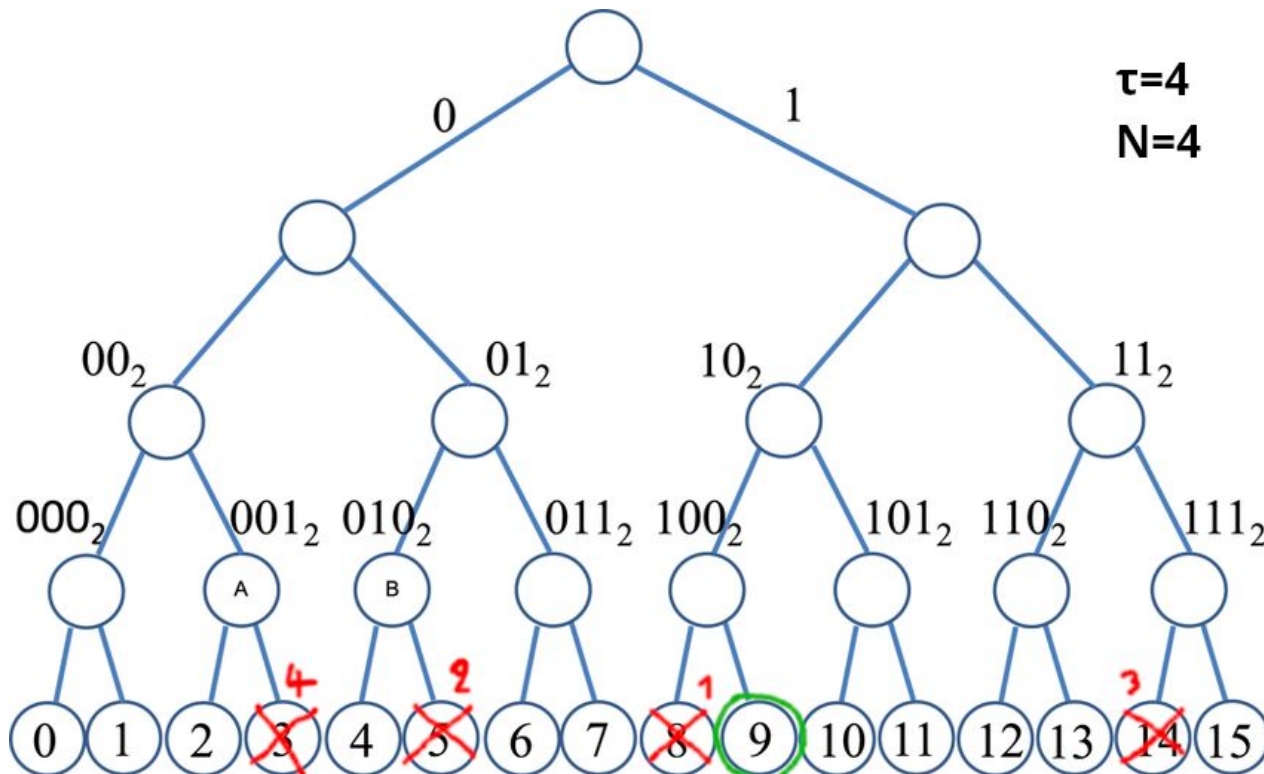
All-but-one share opening



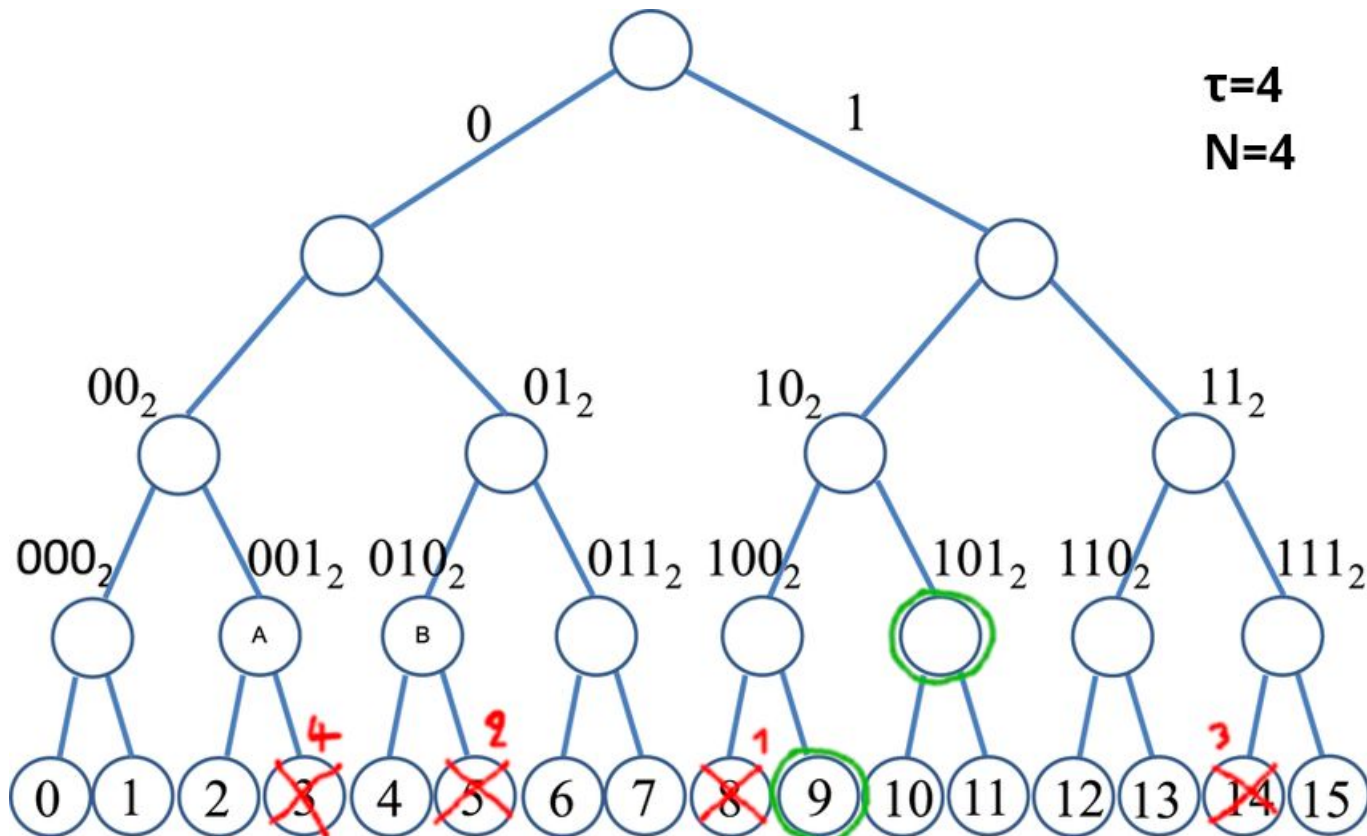
All-but-one share opening



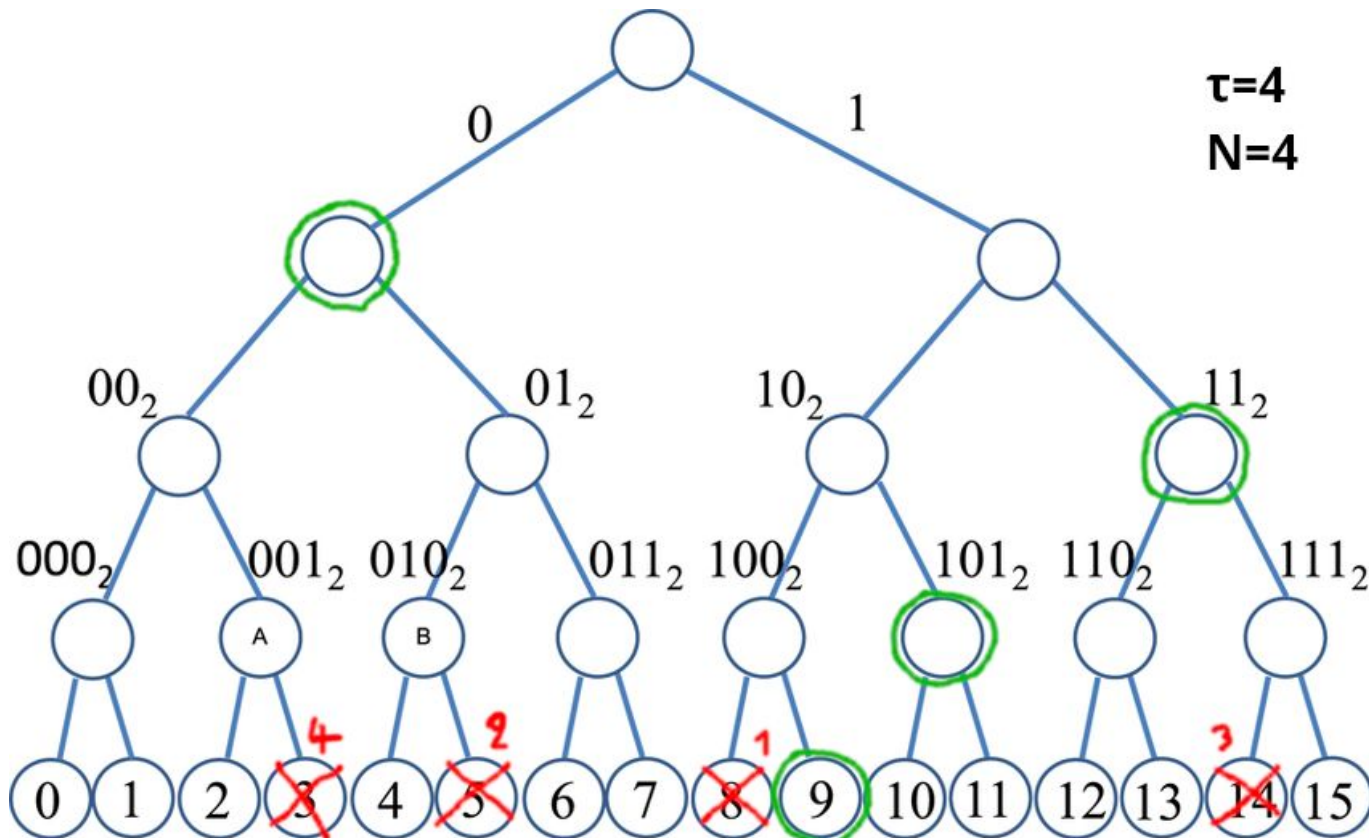
All-but-one share opening



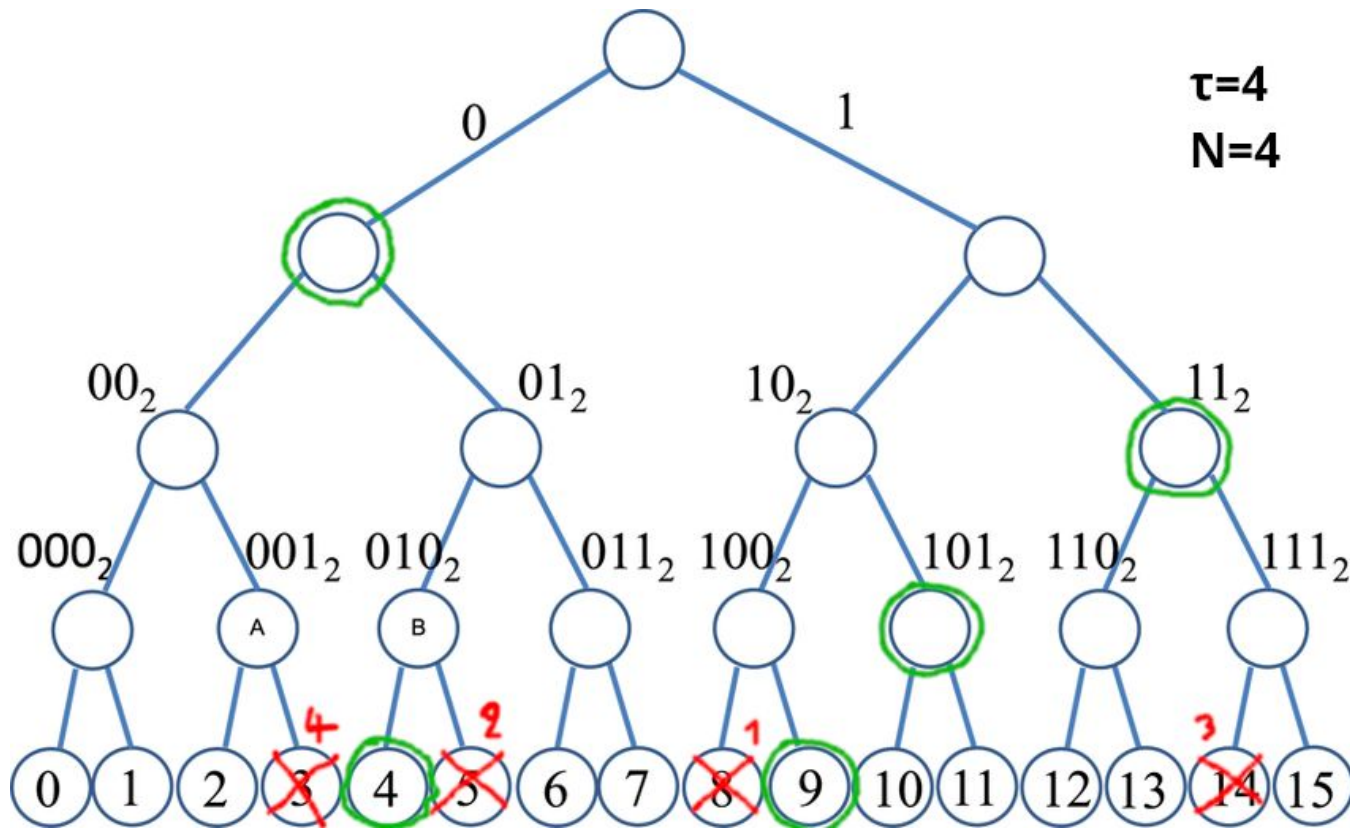
All-but-one share opening



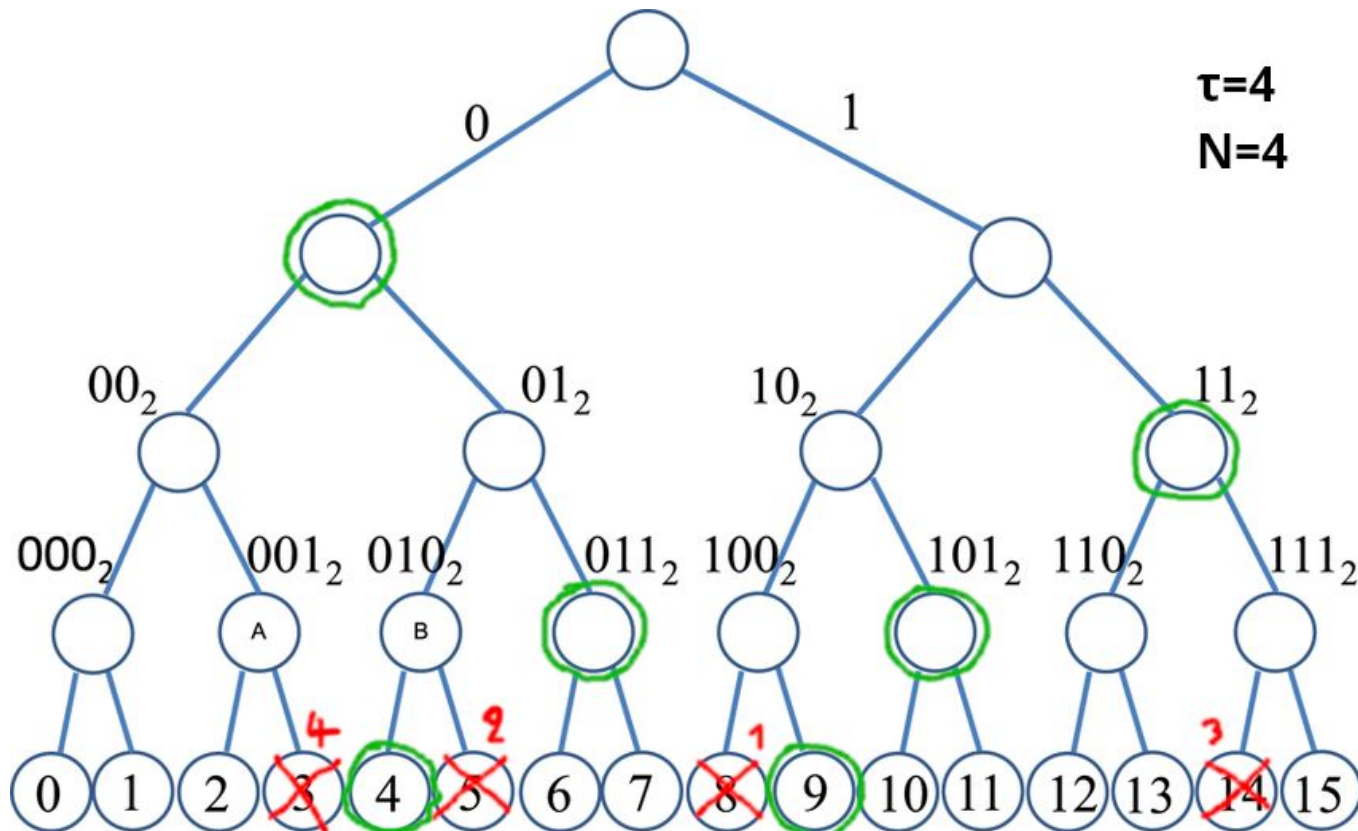
All-but-one share opening



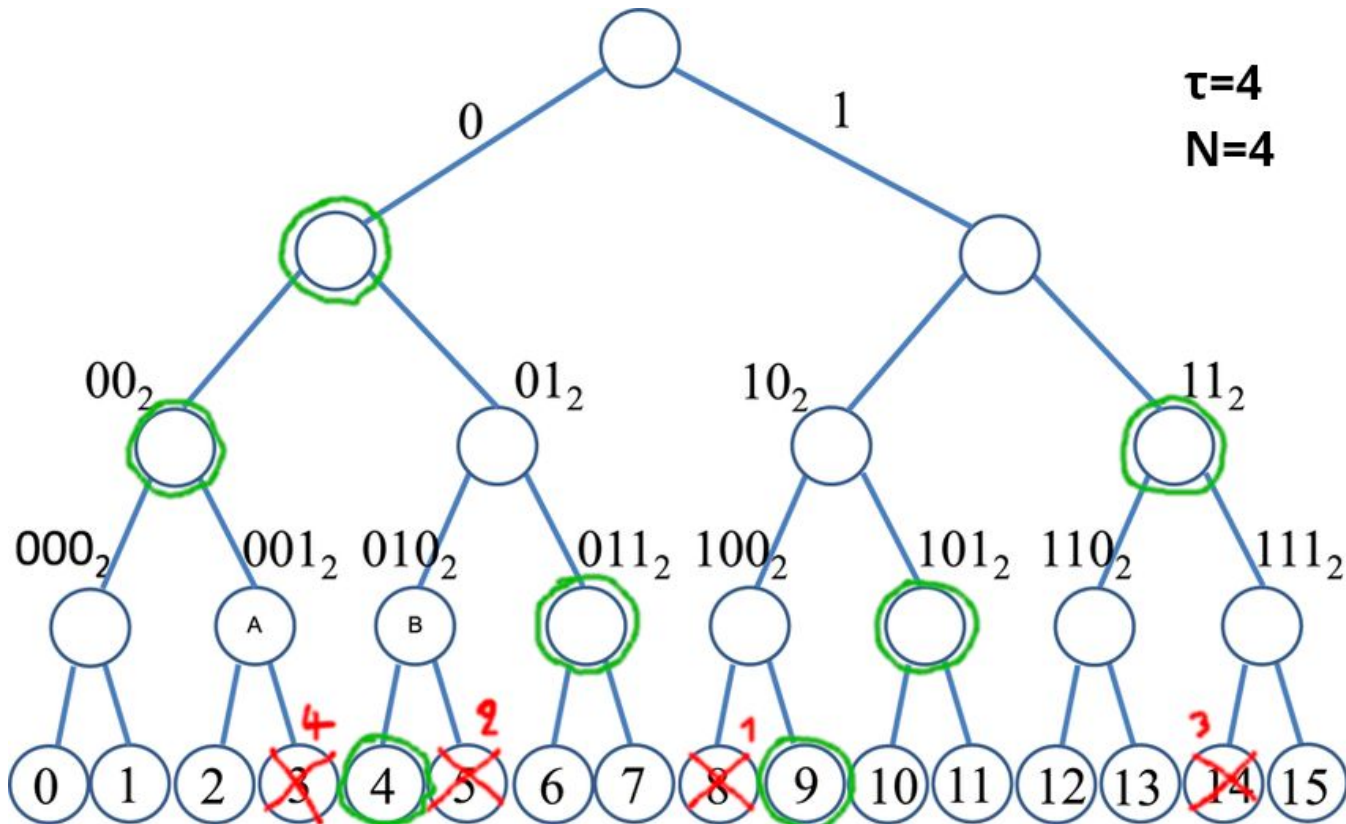
All-but-one share opening



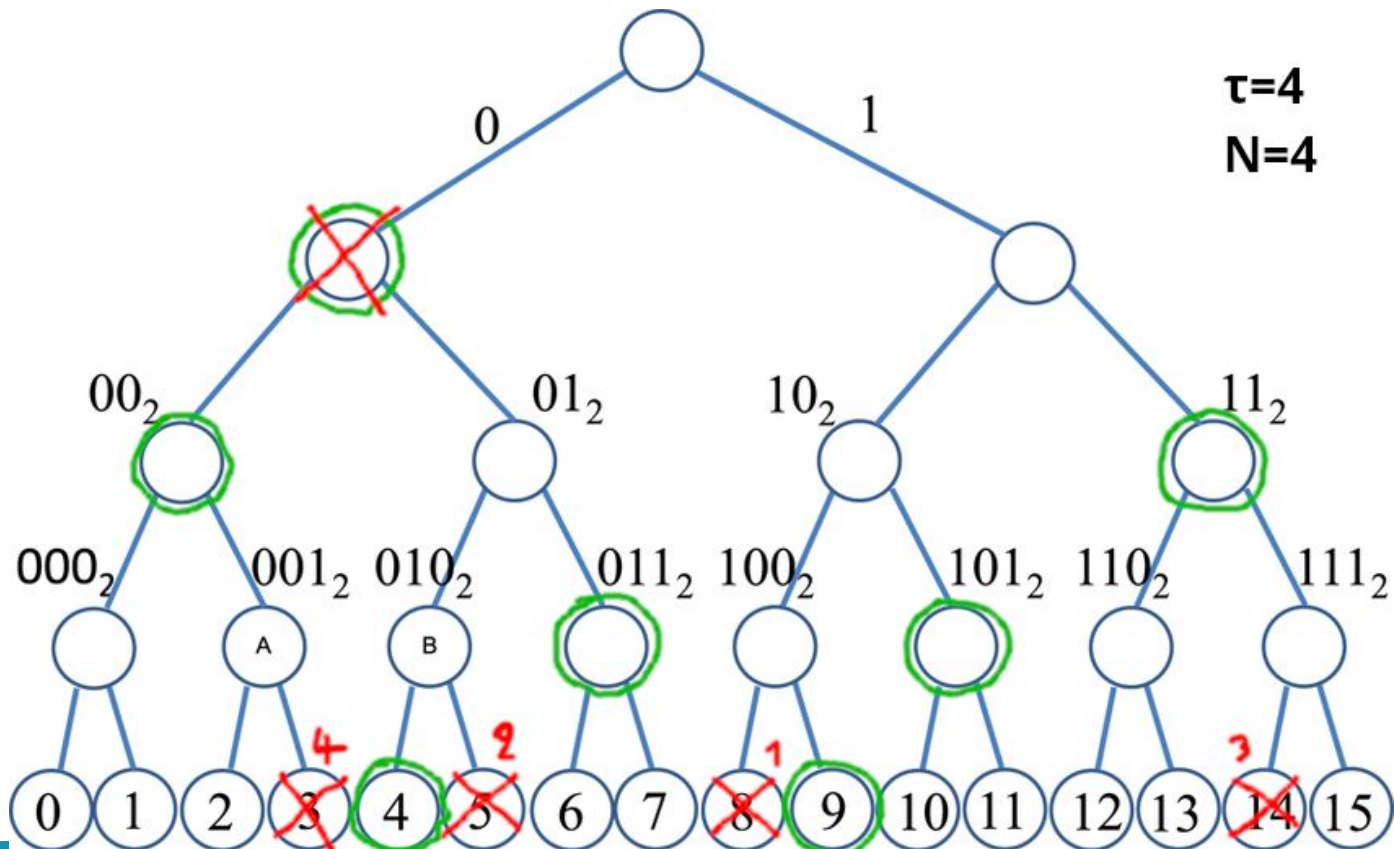
All-but-one share opening



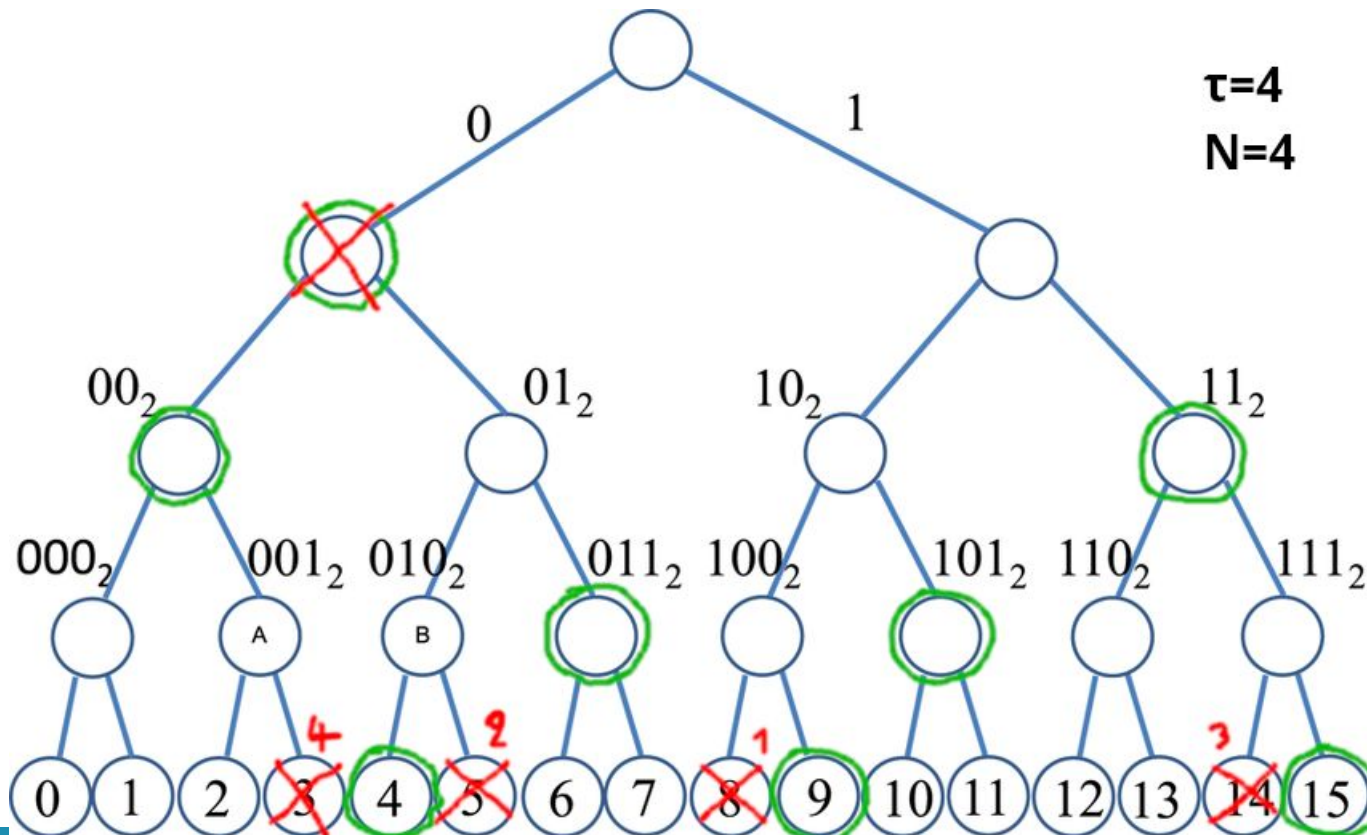
All-but-one share opening



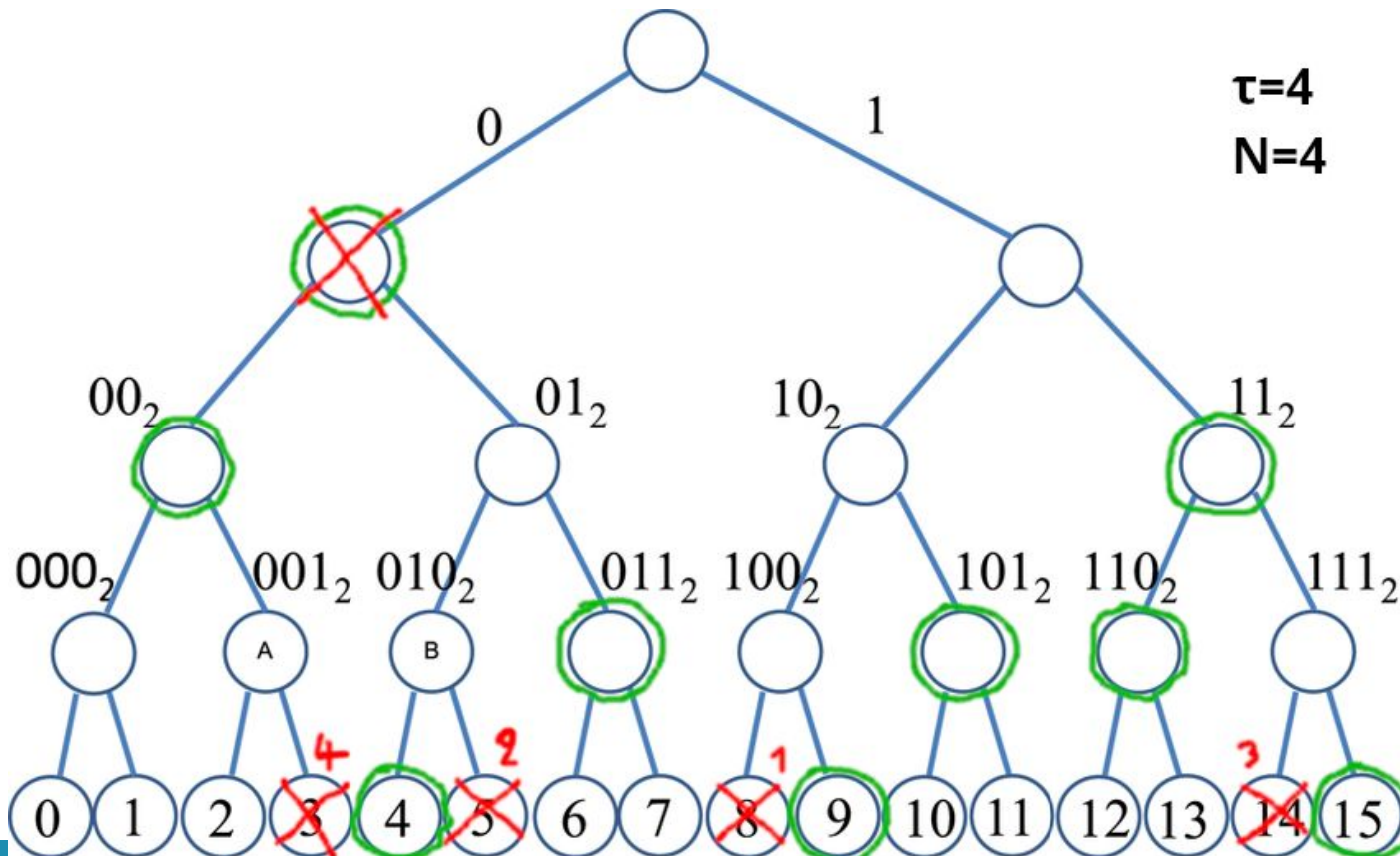
All-but-one share opening



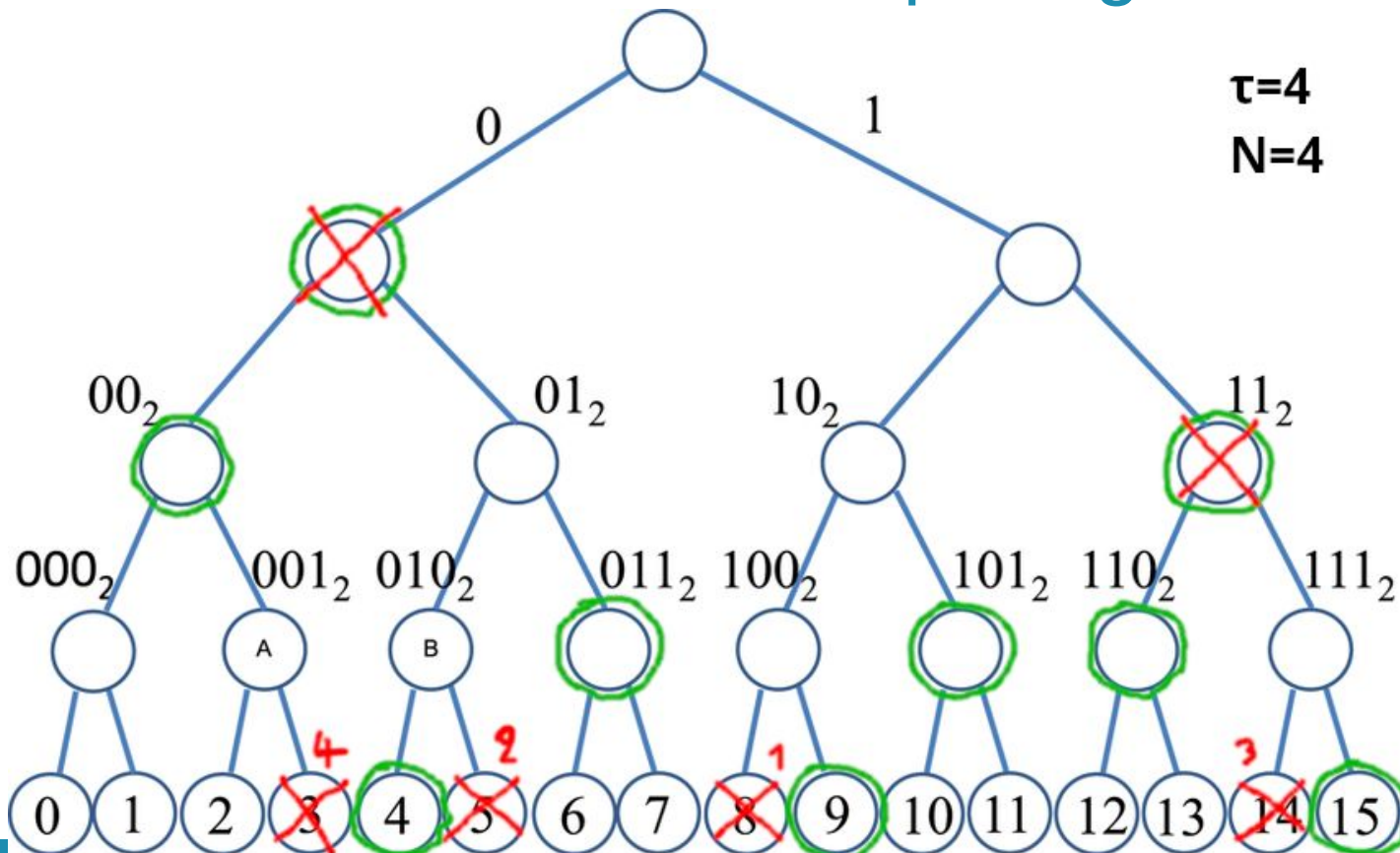
All-but-one share opening



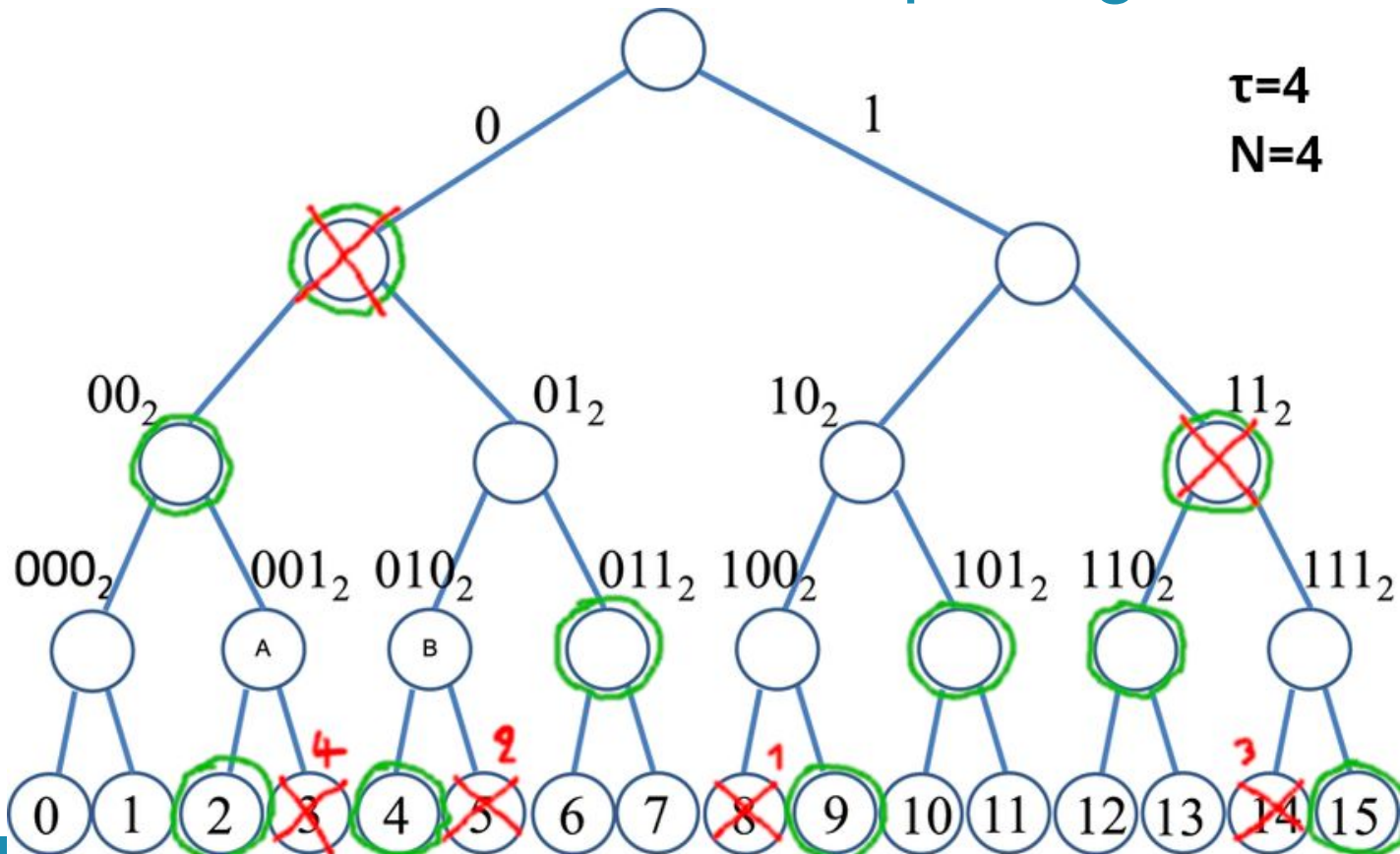
All-but-one share opening



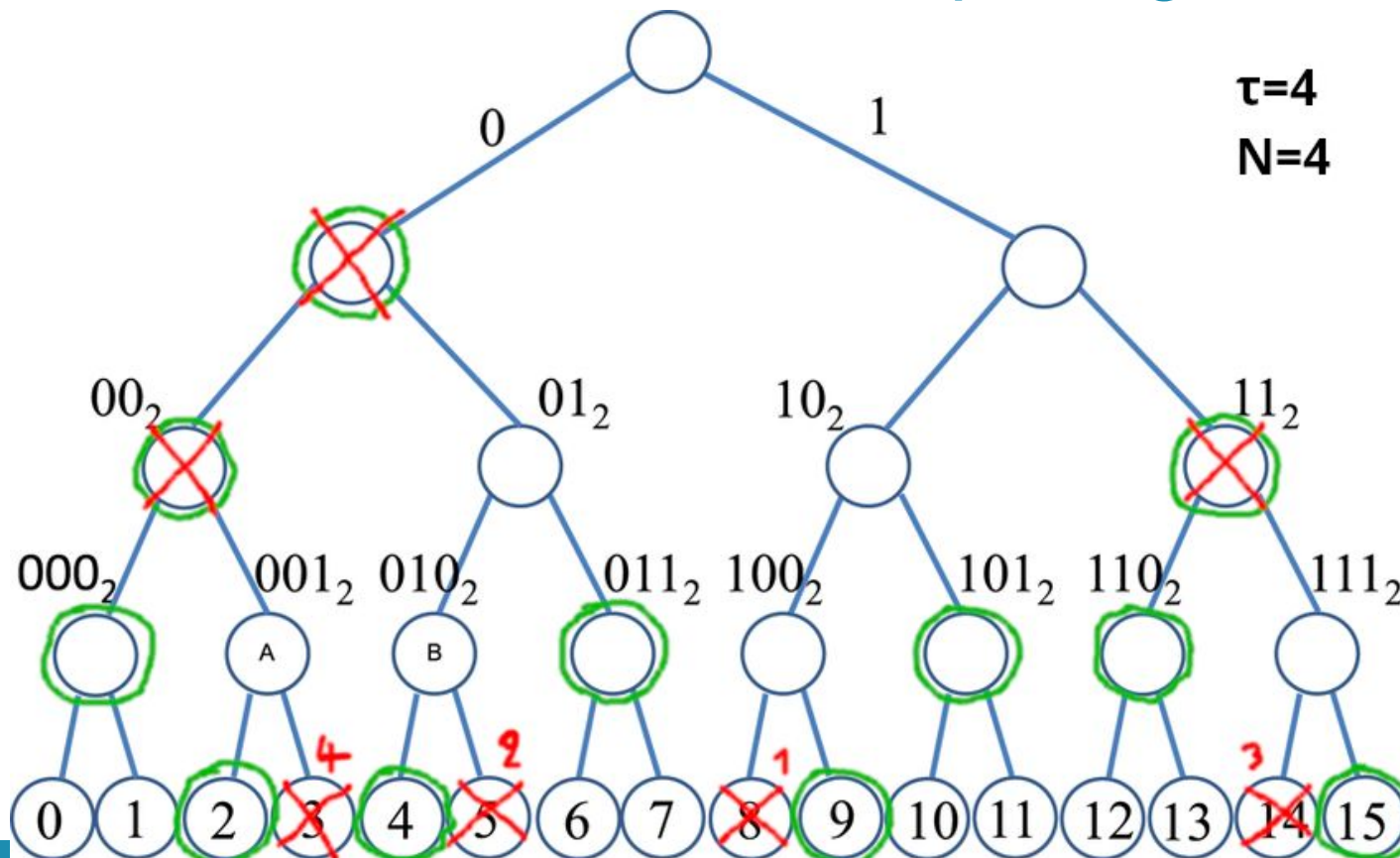
All-but-one share opening



All-but-one share opening



All-but-one share opening



All-but-one share opening

The result?

The whole GGM tree (partially) revealed. Possible by only exposing a (relatively) small number of nodes.

The downside?

The revealed tree's construction is *very runtime-dependent*.
A lot of flexibility needed to accommodate both signature and verification functionality in hardware...

Commit Phase: The GGM Tree



of virtual parties in a round: **N**

of MPC rounds: **τ**

Goal: pseudo-randomly generate **$L = \tau * N$** party views, given a **root_seed** and a **salt**

Mirath relies on the ***AES / Rijndael*** block cipher for all expansions related to the GGM tree.

Commit Phase: The GGM Tree

During the signature packing: reveal *all but one* (N-1) party views for all τ MPCitH rounds

The problem?

Packing all $\tau^*(N-1)$ revealed party views¹ would lead to a **huge** signature

The solution?

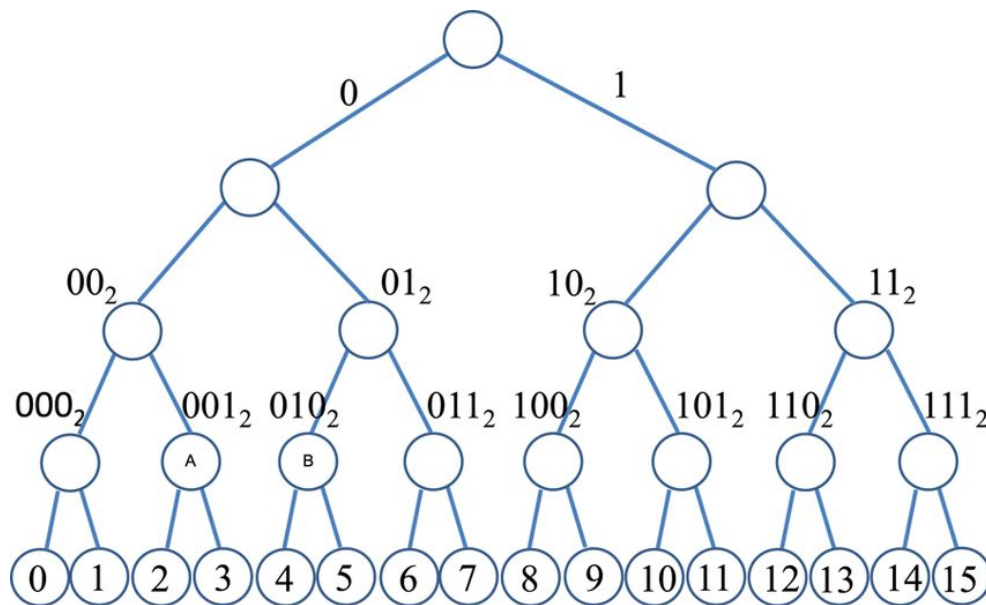
Using a GGM tree allows us to reveal a (potentially) large number of revealed nodes (shares) by sending only a common ancestor

In Mirath: *A single* GGM tree for *all* MPC rounds

[1] Party view: the secret information held only by the given party. The complete secret can only be obtained by combining every party's view.

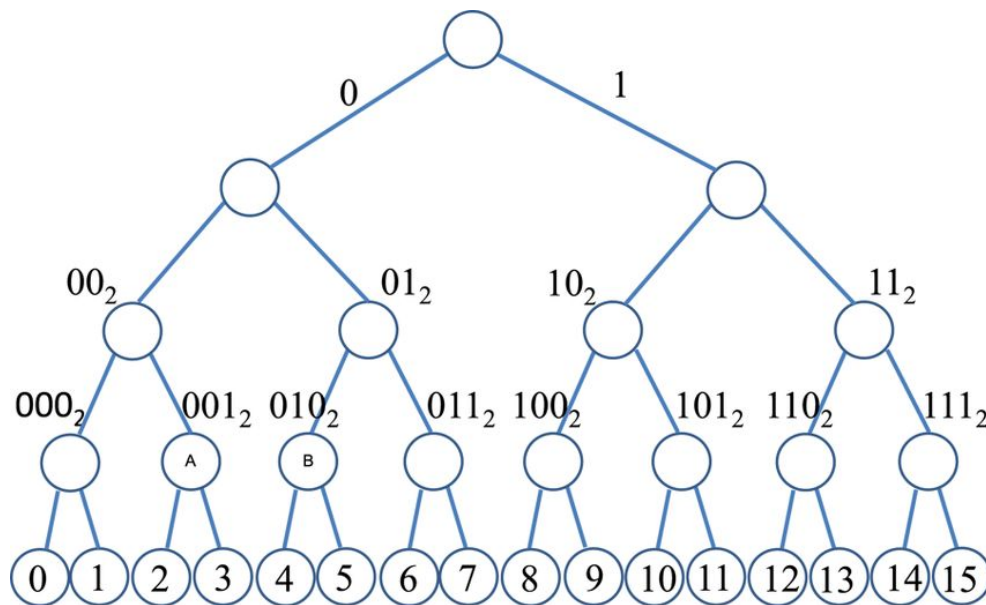
Commit Phase: The GGM Tree

Total size of the GGM tree: $2 * \tau * N - 1$ nodes



Commit Phase: The GGM Tree

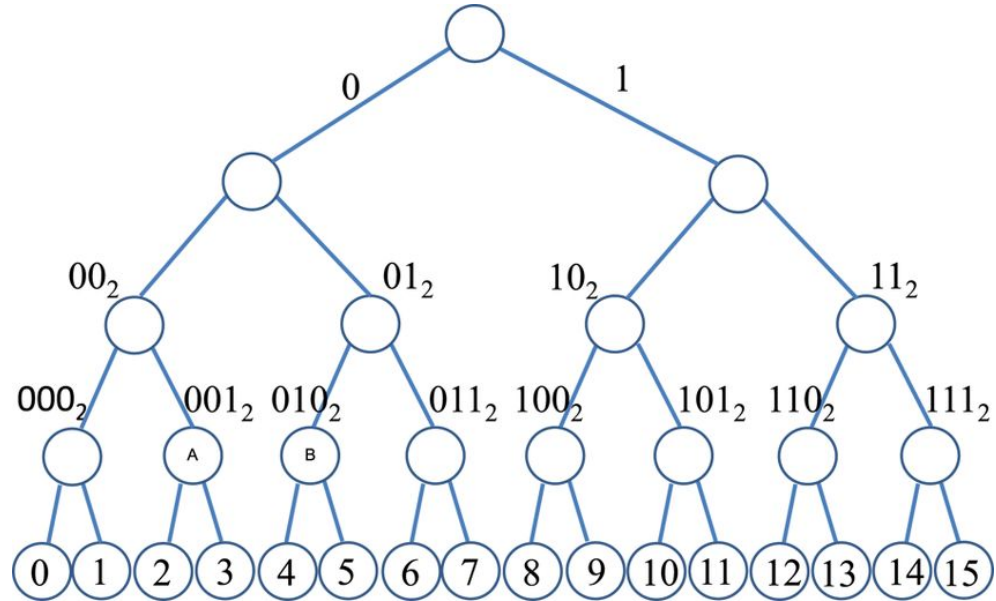
Each **leaf node** needs to be expanded to obtain the matrices used in the arithmetic operations (AES)



Commit Phase: The GGM Tree

Total size of the GGM tree and the sampled randomness:

- ~490 KB for the smallest parameter set.
- For the largest parameter set: ~21 MB!



The Mirath Signature Scheme

NIST PQC Additional DS Round 2 Candidate

The MinRank Problem

Given:

1. a large *public* matrix H and
2. a *public* vector y

The *secret key* encodes a *secret* matrix E that satisfies

$$y = H \cdot \text{vec}(E)$$

Mirath

Mirath's routines rely on the Keccak functions (SHA-3 and SHAKE), the AES block cipher (as a PRG)

The randomness obtained is used for a variety of arithmetic operations

Arithmetic Intensity of Mirath

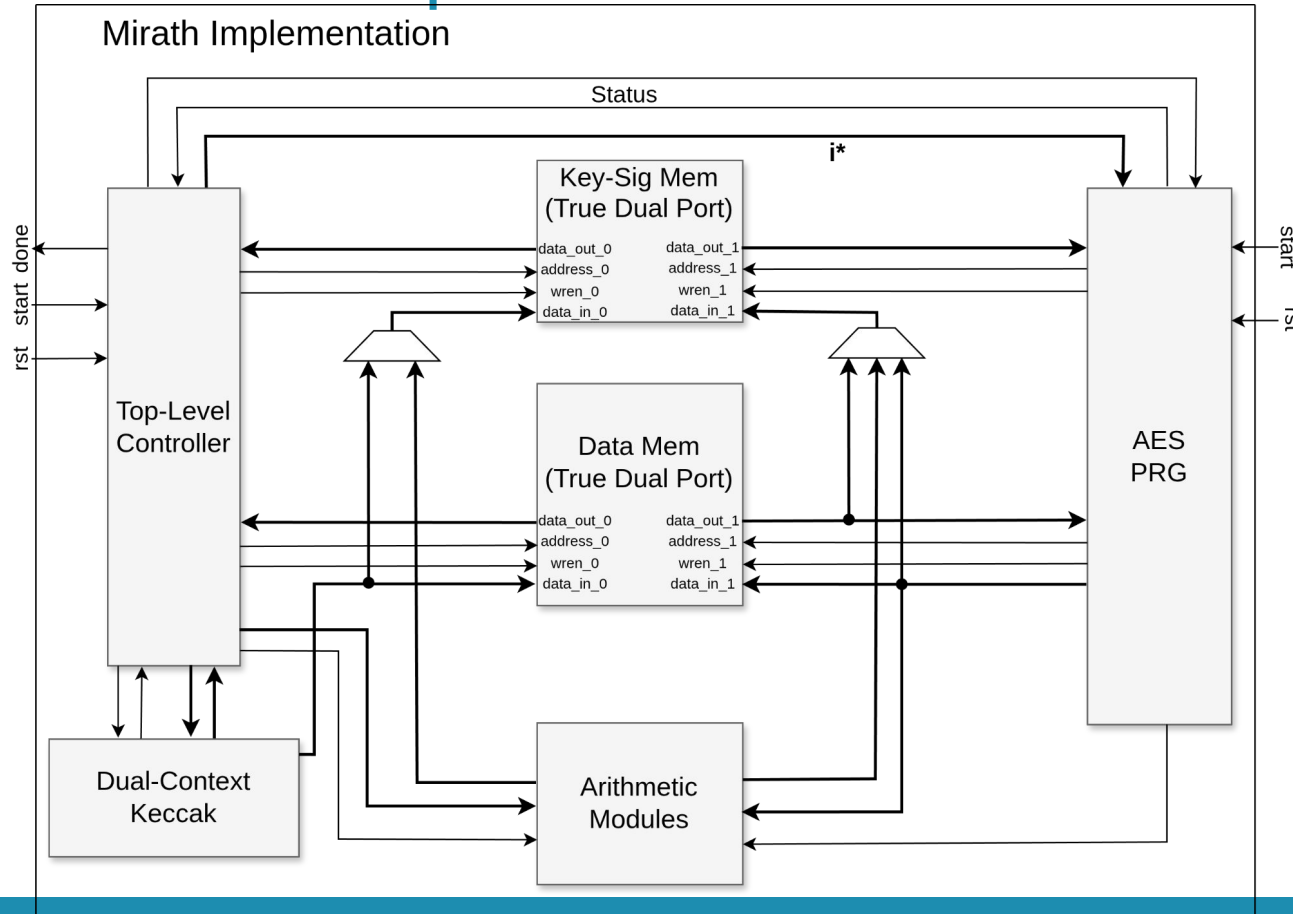
3 main subroutines stand out as the most arithmetically intensive:

- *Commit phase*: shares are distributed (split) amongst the N virtual parties
- *MPC Emulation*: function evaluations are performed here
- *Share opening*: Reveal all but one shares ($N-1$) for *each* MPC round

Outline

- **Multi-Party Computation in the Head (MPCitH) & Mirath**
- **High-Level Architecture & AES Core**
- **Results & Evaluation**

The Top-Level Architecture



The AES PRG Module

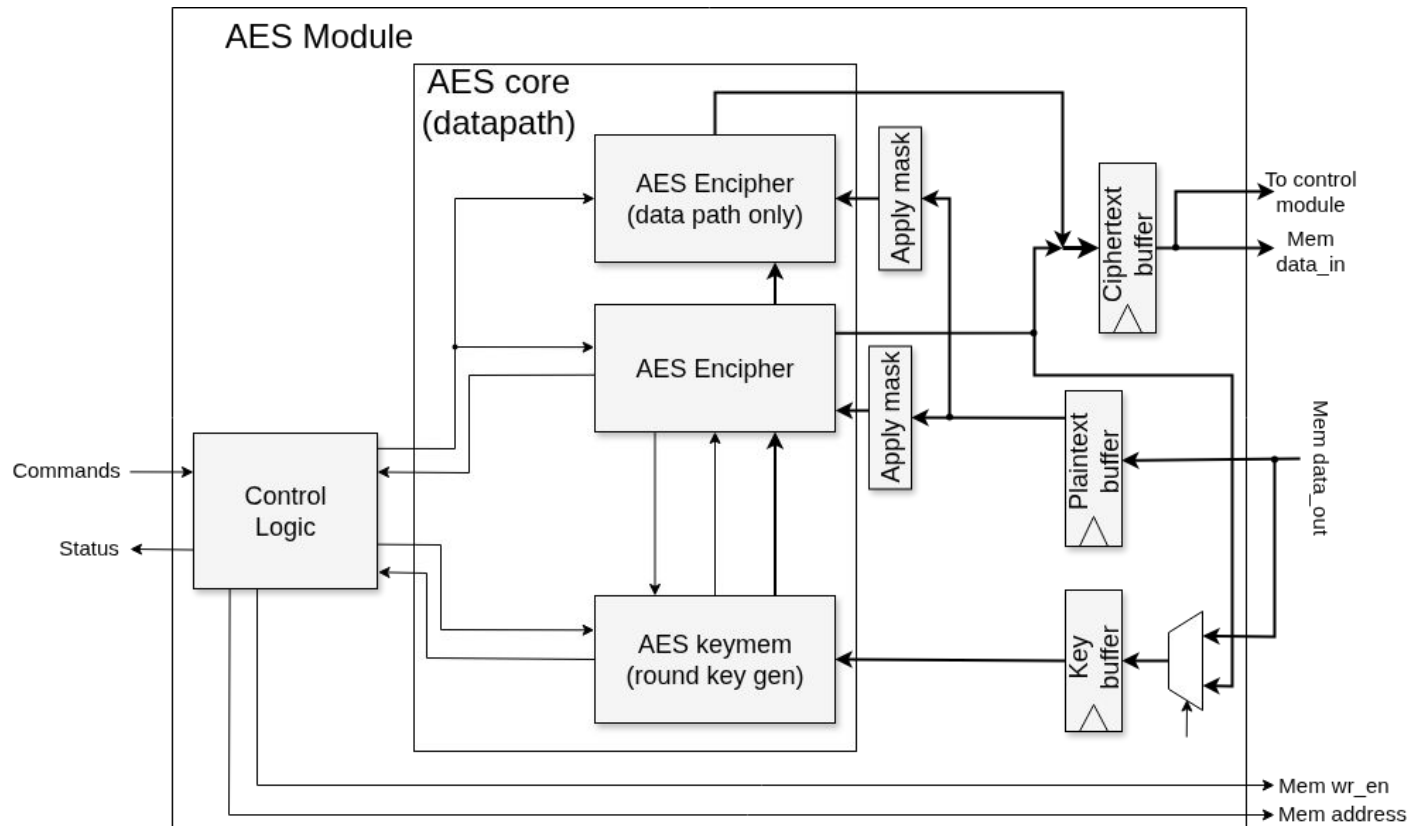
Recap: GGM Tree

- We start from a *root_seed* and expand:
Each node (seed) is expanded twice
- The final nodes (leaves of the tree) are expanded further to obtain the randomness to set each party's matrices and vectors
- In Mirath, the AES block cipher is used as the primitive for the GGM tree

Recap: GGM Tree

- In the signature, the whole GGM tree is compressed in a way that minimizes the signature size, while keeping a number of leaf nodes *hidden*
- **Any single leaf node can end up as a *hidden* node: this implies a huge runtime flexibility requirement for a hardware implementation**

The AES PRG Module



Expanding the Binary Tree

Our AES module uses an open-source AES encryption core [1].

By adding the second Encipher block, we obtain:

- 89-100% throughput gain
- Less than 20% area overhead (1 S-box + output registers)

GGM Tree Expansion in Software:



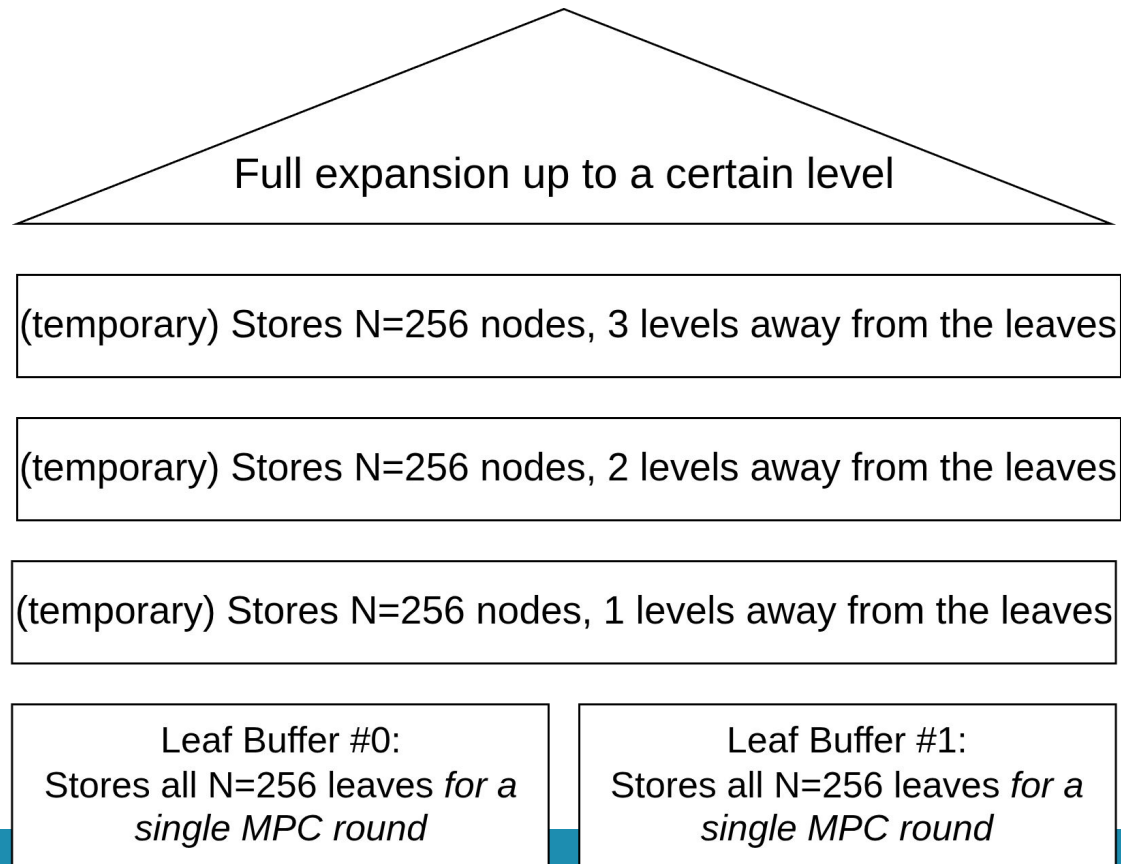
Algorithm 1 Binary GGM Tree Expansion

Require: $root_{seed}$, $salt$, number of leaves L

- 1: let $tree$ be an array large enough to hold all nodes
 - 2: $node[0] \leftarrow root_{seed}$ $\triangleright root_{seed}$ at position 0
 - 3: **for** $i \leftarrow 0$ to $L - 2$ **do**
 - 4: $(node[2i + 1], node[2i + 2]) \leftarrow \text{RIJNDAELEXPANDSEED}(node[i], salt, i)$ $\triangleright$ expand node i into its two children
 - 5: **end for**
 - 6: **return** $tree$
-

Very easy to do with simple array indexing. If you have the memory to spare...

GGM Tree Expansion in Hardware:

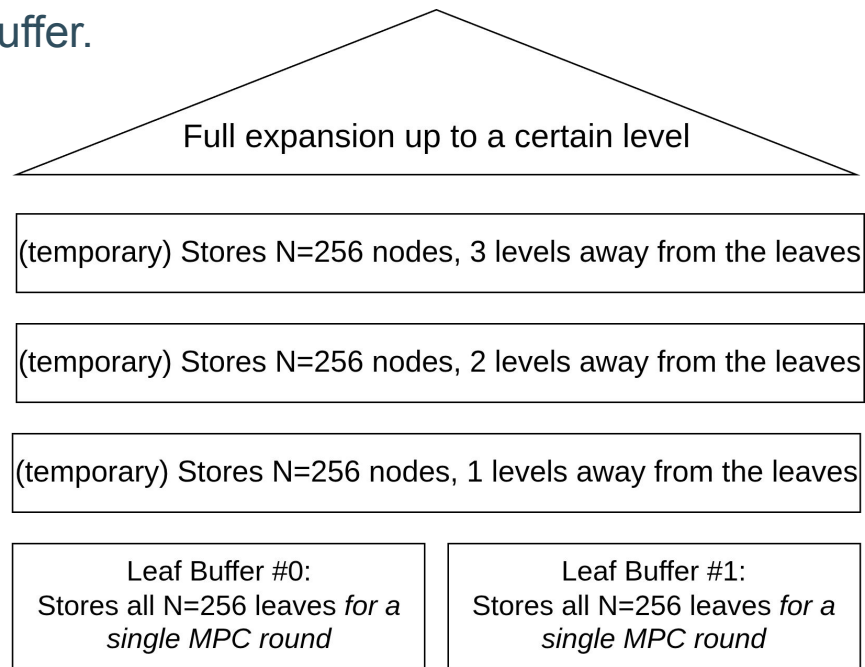


GGM Tree Expansion in Hardware:

MPC round #0: We have nothing, apart from the permanent storage

All other rounds: Half the leaves already in the buffer.
The rest are (mostly) quick to obtain

Recomputation kept to a minimum!



GGM Tree Expansion in Hardware:

Core idea: move to 1-based indexing

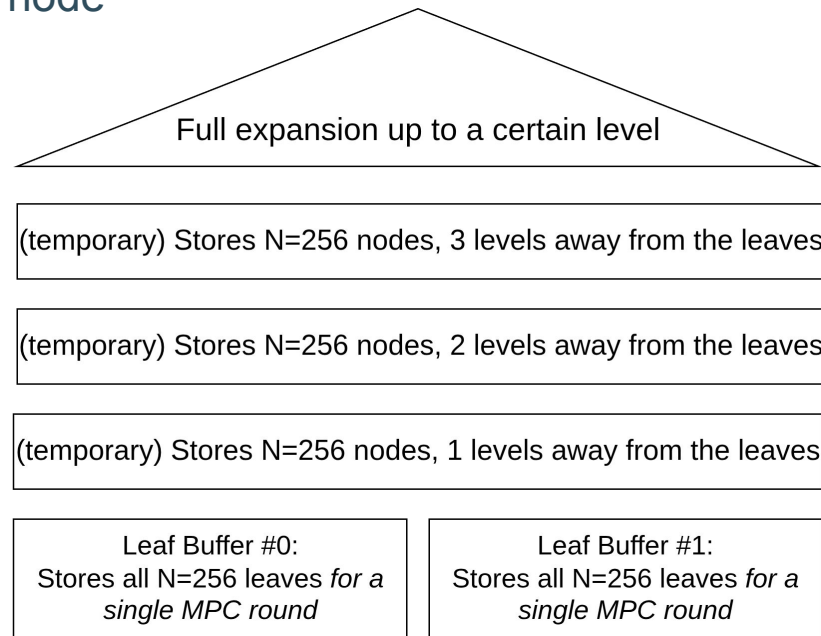
Right-shifting a *node_index* directly returns its parent node

Looking at a single bit of a *node_index* can provide valuable information:

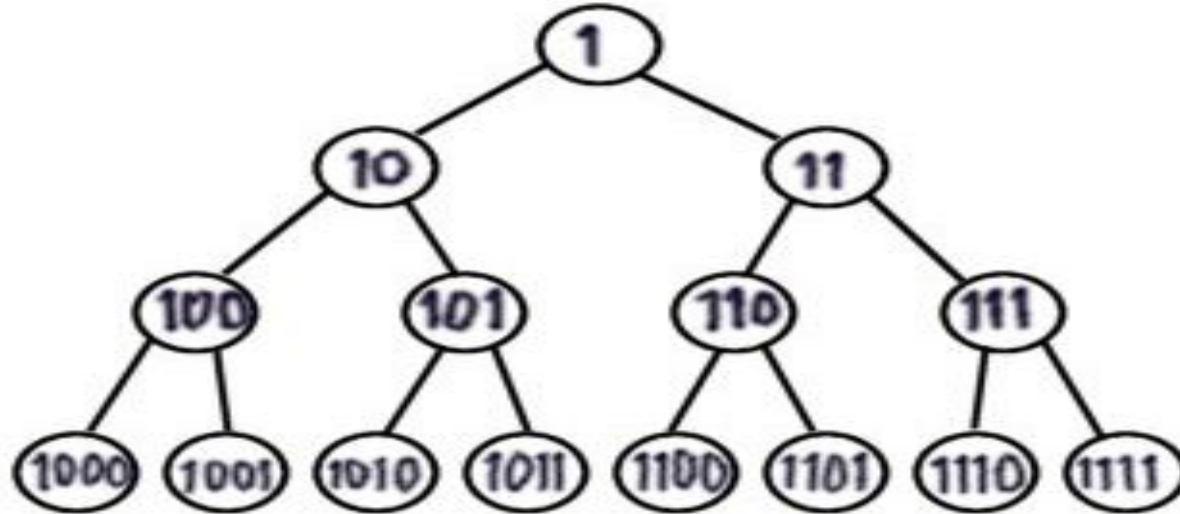
`node_index[0] ? (node_is_right_child) : (node_is_left_child)`

`node_index[1] ? (parent_is_right_child) : (parent_is_left_child)`

•
•
•



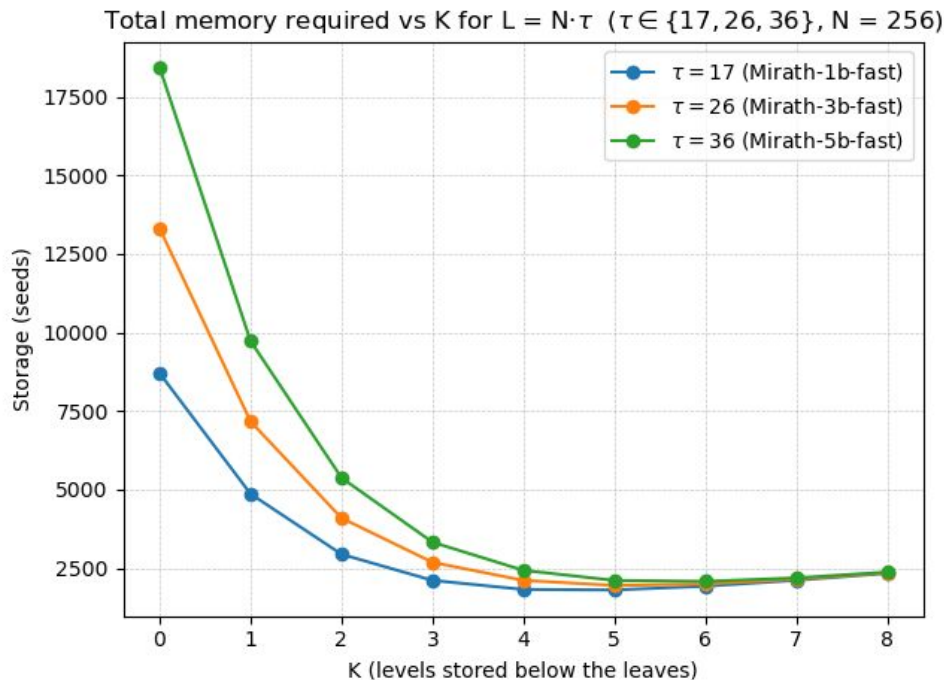
GGM Tree Expansion in Hardware:



Suppose we want to get the leaf with *node_index* **1001101...**

Proposed Binary Tree Expansion Algorithm

Memory requirements *significantly* improved



- Proposed algorithm: obtains the leaves for the first round *much* faster than simple array expansion
- Flat minimum allows for design flexibility

The Keccak Module

Our Contributions



This work expands on the SHAKE module presented in [1].

Our improvements:

- The Keccak module can perform both SHA-3 and SHAKE functions.
- ~2.1x the throughput, minimal area overhead.
- Supports 2 concurrent hashing contexts.

Implementing the Arithmetic of Mirath in Hardware

MPC Arithmetic

Additions & multiplications of *large* matrices

Matrix dimensions not hardware friendly (e.g. 1443×321)

Arithmetic modules at least as fast as the AES to avoid stalling

During MPC emulation: Keccak and AES functions unused. Arithmetic limits performance

Arithmetic in Mirath



Need to perform intensive matrix operations with high-throughput

Aims of the arithmetic modules:

- Small area footprint per-module -> Compactness & ease of parallelizability
- Flexibility for reuse across many subroutines
- High performance

Arithmetic not covered further here. Questions are welcome ;)

Detailed Area Metrics

TABLE 5.1: Resource utilization of all modules and the complete design.

Module	LUT	FF	DSP	BRAM
Top-level Control Module	1493	374	0	0
Keccak (Dual Context)	4570	1773	0	0
AES PRG (Perf-Optimized)	2111	1016	0	0
Main Memory	0	0	0	7.5
$S, C', acc[e]$ accumulator	130	512	0	0.5
$E = [S \mid SC']$ mul/mem	261	20	0	0
y accumulator	154	333	0	0
S_{base} mem/acc	1202	722	0	0
C'_{base} mem/acc	1195	700	0	0
v_{base} & v module	1865	714	0	0
$(e_A + H'e_B) / (e'_A + H'e'_B)$ module	1108	857	0	2.5
Complete Implementation	14085	7022	0	10.5

Performance Comparison

TABLE 5.4: Performance comparison (KeyGen and Sign) of signature schemes in terms of latency (Mcycles) and time (ms).

Parameter Sets	KeyGen (Mcycles)	KeyGen (ms)	Sign (Mcycles)	Sign (ms)
Mirath [This work]				
Mirath-1b-fast	0.016	0.11	2.30	15.86
Dilithium [20]				
Dilithium-L2	0.004	0.04	0.03	0.29
SPHINCS⁺-simple [21]				
SPHINCS ⁺ -128s	–	–	12.40	–
SPHINCS ⁺ -128f	–	–	1.01	–
SDitH [22]				
SDitH-L1-GF256	0.055	0.34	6.73	41.04
SDitH-L1-GF251	0.057	0.35	21.45	130.77
PICNIC [23]				
PICNIC-L1	–	–	0.03	0.25
LESS [24]				
LESS-L1-b	0.029	0.14	5.20	26.02
LESS-L1-i	0.077	0.38	5.13	25.63
LESS-L1-s	0.174	0.87	4.17	20.83
UOV (4 AES rounds, pipelined AES) [18]				
uov-Ip-pkc	4.05	42.71	0.01	0.08
uov-Ip-pkc+skc	3.76	39.76	0.30	3.21

Mirath vs SDitH:
3.4x [keygen]
2.6x [sign]

Performance Comparison: Relative speed-up



Scheme	Impl./CPU	KeyGen (Kcycles)	Sign (Kcycles)
Mirath-1b-fast	AVX2, i9-13900K	500	9,800
Dilithium-L2	AVX2, i7-11700F	106	251

Table 1: Software clock cycles for KeyGen and Sign for Mirath-1b-fast and Dilithium (L2)

Scheme	KeyGen (speed-up)	Sign (speed-up)
Mirath-1b-fast	1.515x	0.206x
Dilithium-L2	0.883x	0.289x

Table 2: Relative speed-up (HW vs SW) for Mirath-1b-fast and Dilithium-L2. SW latency computed at 3.0 GHz.

Area Utilization Comparison

TABLE 5.3: FPGA resource utilization and clock frequency for recent PQC schemes.

Parameter sets	LUT	FF	DSP	BRAM	Frequency (MHz)
Mirath [This work]					
Mirath-1b-fast	14,085	7,022	0	10.5 (+4) ²	145
Dilithium [20]					
Dilithium-L2	29,998	10,336	10	11.0	97
SPHINCS⁺-simple[†] [21]					
SPHINCS ⁺ -128s	48,231	72,514	0	11.5	250 & 500
SPHINCS ⁺ -128f	47,991	72,505	1	11.5	250 & 500
SDitH [22]					
SDitH-L1-GF256	16,592	8,778	0	164.5	164
SDitH-L1-GF251	17,423	16,336	196	164.5	164
PICNIC[†] [23]					
PICNIC-L1	90,337	23,105	0	52.5	125
LESS [24]					
LESS-L1-b	54,800	39,900	0	59.5	200
LESS-L1-i	76,700	57,900	0	102.5	167
LESS-L1-s	104,300	76,700	0	167.5	143
UOV (4 AES rounds, pipelined AES) [18]					
uov-Ip-pkc	33,220	23,913	2	81	94.8
uov-Ip-pkc+skc	33,363	24,233	2	48	94.5

Current Work



- Further parallelizing subroutines that depend on different modules for decreased latency with zero area overhead:

Share splitting: 1.29 Mcycles -> 0.98 Mcycles!

Current Work



- Optimizing the MPC arithmetic modules to half the total MPC computation latency:

MPC emulation: 988 Kcycles -> 494 Kcycles!

Current Work



- Design with high frequency in mind. Current design runs at about 20% higher $f_{\max}$. (Only takes a couple of hundreds of extra FFs.)

Current Work



- Further parallelizing subroutines that depend on different modules for decreased latency with zero area overhead.
- Optimizing the MPC arithmetic modules to half the total MPC computation latency (area overhead of parallelization much less than 100% for the arithmetic modules).
- Design with high frequency in mind. Current design runs at about 20% higher $f_{\max}$. (Only using a couple of hundreds of extra FFs.)

Total speed-up: **~45.7%!**

Discussion

This work shows that:

- State-of-the-art MPCitH-based signatures can be efficiently implemented, yet provide competitive performance
- Still, the required runtime flexibility significantly bounds the potential for hardware efficiency and increases design complexity
- Parameter-level optimizations to the algorithm would enable significant hardware-relevant improvements

ATP Comparison with Add. Sig. Round 2 DSAs



COSIC

Scheme	Parameter Set	Resources LUT/FF/DSP/BRAM	Area ^a	Freq. (MHz)	KeyGen (Mcycles/ms)	Sign (Mcycles/ms)	Verify (Mcycles/ms)	ATP ^a
Mirath (This work)	1b-fast	15,400/9,445/0/42.0	28,000	254	0.0161/0.0634	0.554/2.181	0.5/1.969	118.0
	5b-fast	30,415/18,223/0/96.0	59,215	218	0.0607/0.2784	1.3726/6.296	1.2116/5.558	718.4
Mirath^b [AAB ⁺ 25a, AAB ⁺ 25b]	1b-fast	-/-/-/-	-	3000	1.8/0.60	1600/533.3	2600/866.67	-
	5b-fast	-/-/-/-	-	3000	6.5/2.17	8400/2800.0	13000/4333.3	-
	1b-fast-AVX2	-/-/-/-	-	3000	0.5/0.17	9.8/3.27	5.5/1.83	-
	5b-fast-AVX2	-/-/-/-	-	3000	2.0/0.67	70.1/23.37	51.7/17.23	-
ML-DSA [NKV25]	L2	54,942/29,309/16/29.0	65,242	230	0.0049/0.0213	0.024/0.1043	0.0066/0.0287	10.07
	L5	54,942/29,309/16/29.0	65,242	230	0.014/0.0609	0.038/0.1652	0.0146/0.0635	18.89
Falcon [OZZ⁺25]	L1	79,065/46,389/226/45.0	115,165	65	-/-	0.16/0.64	-/-	-
SPHINCS⁺ [ALCZ20]	128f	47,991/72,505/1/11.5	51,541	250/500	-/-	-/1.01	-/0.16	60.3
	256f	51,009/74,539/1/22.5	57,859	250/500	-/-	-/2.52	-/0.21	158.0
CROSS [KAB ⁺ 25]	RS DP-1-f	26,741/11,678/0/45.0	40,241	118	-/0.035	-/0.478	-/0.409	37.1
	RS DP-5-f	34,626/15,331/0/91.5	62,076	113	-/0.028	-/1.20	-/1.14	147.0
LESS [BWMG23]	L1-s	54,800/39,900/0/59.5	72,650	200	0.174/0.87	4.17/20.83	4.137/20.69	3,709.6
	L5-s	104,300/76,700/0/167.5	154,550	143	0.247/1.73	87.16/610.13	87.013/609.09	188,697.8
MAYO [HSK⁺24]	L1-f	106,000/38,000/2/45.5	119,850	75	0.0058/0.08	0.031/0.43	0.0046/0.05	67.1
UOV [BCD ⁺ 25]	Is-pkc+skc	28,947/24,444/2/66.0	48,947	90.8	11.1/122.25	0.84/9.251	0.284/3.128	6,589.7
	V-pkc+skc	77,352/38,217/4/359.0	185,452	92.5	39.1/422.70	3.31/35.784	1.92/20.757	88,876.2
SDitH [DHSY24]	L1-GF256	16,592/8,778/0/164.5	65,942	164	0.055/0.34	6.73/41.04	8.688/52.98	6,222.3
	L5-GF256	23,323/14,962/0/520.5	179,473	164	0.083/0.51	17.48/106.60	24.940/152.10	46,521.2
PICNIC [KRR ⁺ 20]	L1	90,337/23,105/0/52.5	106,087	125	-/-	0.03/0.25	0.030/0.24	52.0
	L5	167,530/33,164/0/98.5	197,080	125	-/-	0.15/1.24	0.147/1.17	475.0

Post-Quantum Crypto (PQC) Standardization



NIST Additional Digital Signatures call (in progress):

- many different families
- limited hardware research.

MPC in-the-Head (MPCitH):

- promising DSA family
- **Quite slow, but improving.**
- **Very small public keys.**

	Mirath-b-fast Parameter	NIST Security Level		
		L1	L3	5
Security (Bits)	Classical Bit Security	158	224	255
	Target Security (λ)	128	192	255
Output Sizes (Bytes)	sk (secret key)	32	48	64
	pk (public key)	57	84	128
	σ (signature)	3 456	7 924	14 336

[1] DSA = Digital Signature Algorithm

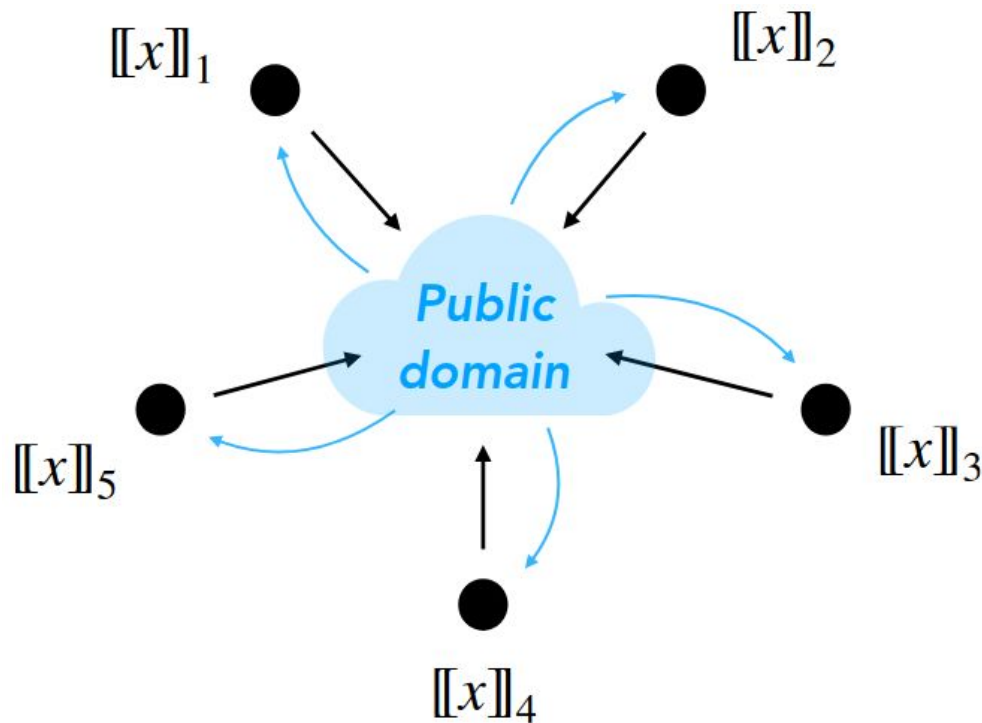
Motivation

We need cryptographic algorithms that are secure against attacks by both classical and quantum computers.



Image credit: Google

The MPC-in-the-Head (MPCitH) Framework



Standard MPC: N parties interact to jointly compute a function without revealing their private data.

MPCitH: simulates the entire process inside a single machine.

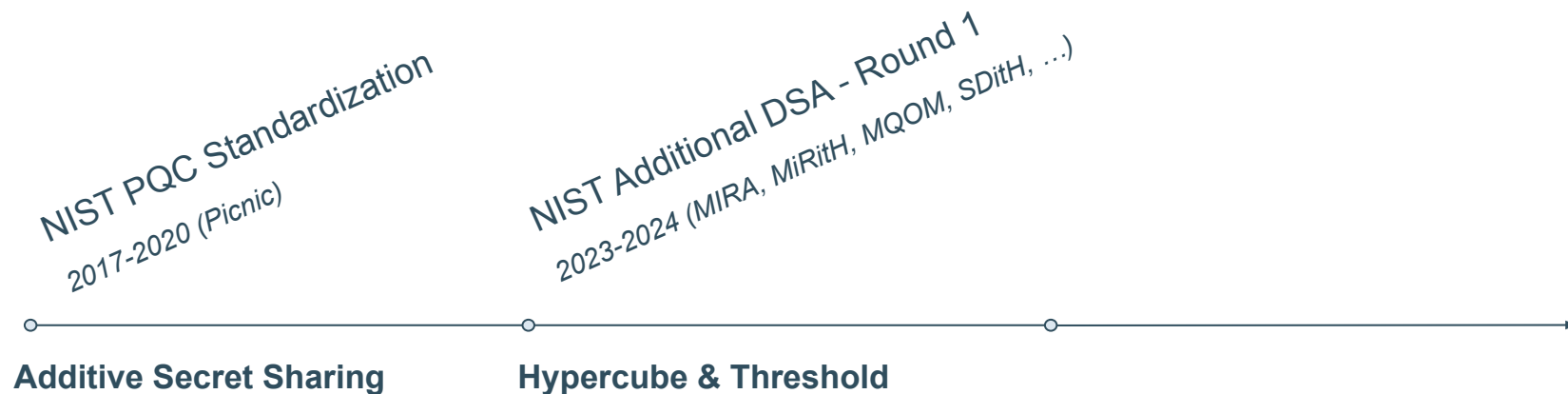
MPCitH Framework Evolution

NIST PQC Standardization
2017-2020 (Picnic)



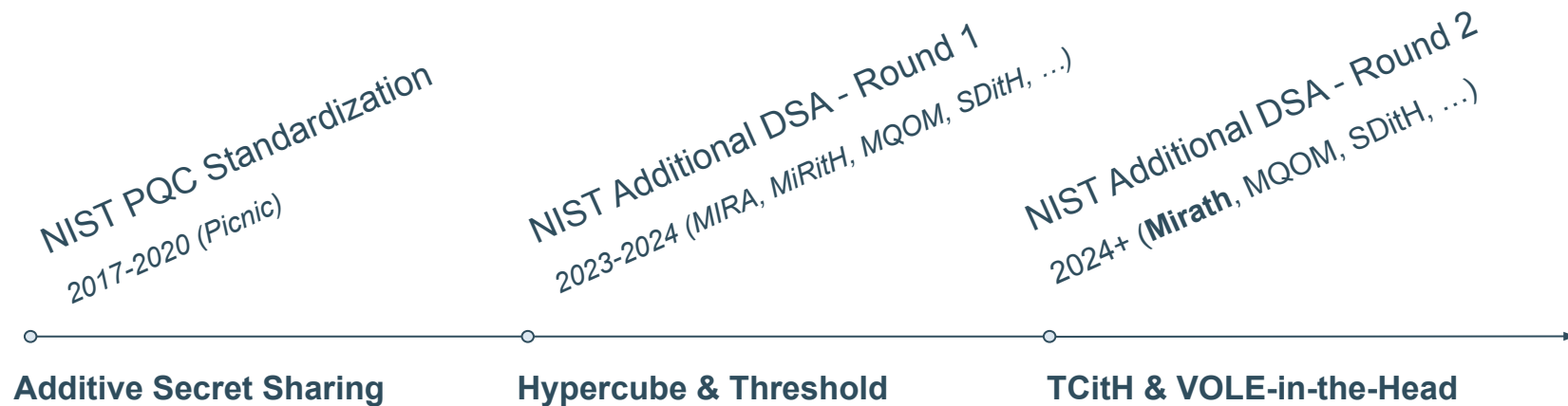
NIST (on Picnic): “[...] *future cryptosystems that evolve out of the MPCitH paradigm may eventually prove significantly superior to the third-round Picnic design*”^[1]

MPCitH Framework Evolution



Deshpande et al. (on SDiTH): Signing & Verification latency dominated by polynomial evaluations. **Large on-chip memory requirements.**^[1]

MPCitH Framework Evolution



Step 0: Initialization

- Functionally equivalent to KeyGen
- Computationally trivial compared to Steps 1-3

Algorithm 22 Signing algorithm.

Sign(sk, msg)

// Step 0: Initialization.

1 : $(H', pk, S, C') \leftarrow \text{DecompressSK}(sk)$

2 : $\text{salt} \leftarrow \text{TRG.Seed}(2\lambda)$

3 : $\text{rseed} \leftarrow \text{TRG.Seed}(\lambda)$

// Step 1: Build & Commit to Witness Polynomials.

4 : $(\text{base}, v, h_{\text{sh}}, \text{key}, \text{aux}) \leftarrow \text{CommitWitnessPolynomials}(\text{salt}, \text{rseed}, S, C')$

// Step 2: Compute the polynomial proof $P_{\alpha}(X)$.

5 : $\Gamma \leftarrow \text{ExpandChallengeMatrix}(h_{\text{sh}}) \triangleright \Gamma \in \mathbb{F}_{q^{\mu}}^{p \times (mn-k)}$

6 : **for** e from 0 to $(\tau - 1)$ **do**

7 : $(\alpha_{\text{mid}}[e], \alpha_{\text{base}}[e]) \leftarrow \text{ComputePolynomialProof}(\text{base}[e], v[e], S, C', \Gamma, H')$

8 : $h_{\text{piop}} \leftarrow \text{Hash}_2(pk, \text{salt}, \text{msg}, h_{\text{sh}},$
 $\alpha_{\text{mid}}[0], \alpha_{\text{base}}[0], \dots, \alpha_{\text{mid}}[\tau - 1], \alpha_{\text{base}}[\tau - 1])$

// Step 3: Open random evaluations of the polynomials $P_S(X)$, $P_C(X)$ and $P_v(X)$.

9 : $(\text{ctr}, \pi_{\text{BAVC}}) \leftarrow \text{OpenRandomEvaluations}(\text{key}, h_{\text{piop}})$

10 : $\sigma \leftarrow \text{UnparseSignature}(\text{salt}, \text{ctr}, h_{\text{piop}}, \pi_{\text{BAVC}}, \text{aux}, \alpha_{\text{mid}})$

11 : **return** σ

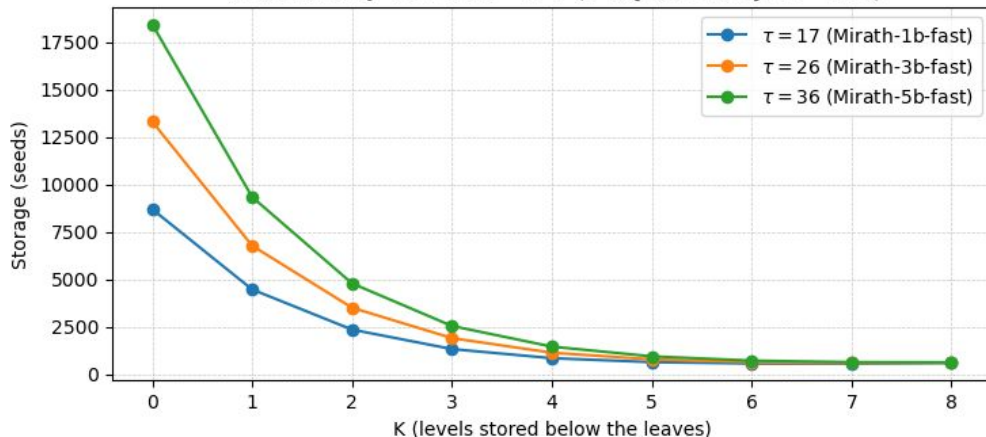
Our Proposed Expansion Algorithm

Instead: Use a hierarchy of **small runtime buffers**

- + minimal memory overhead
- + near-zero recomputation

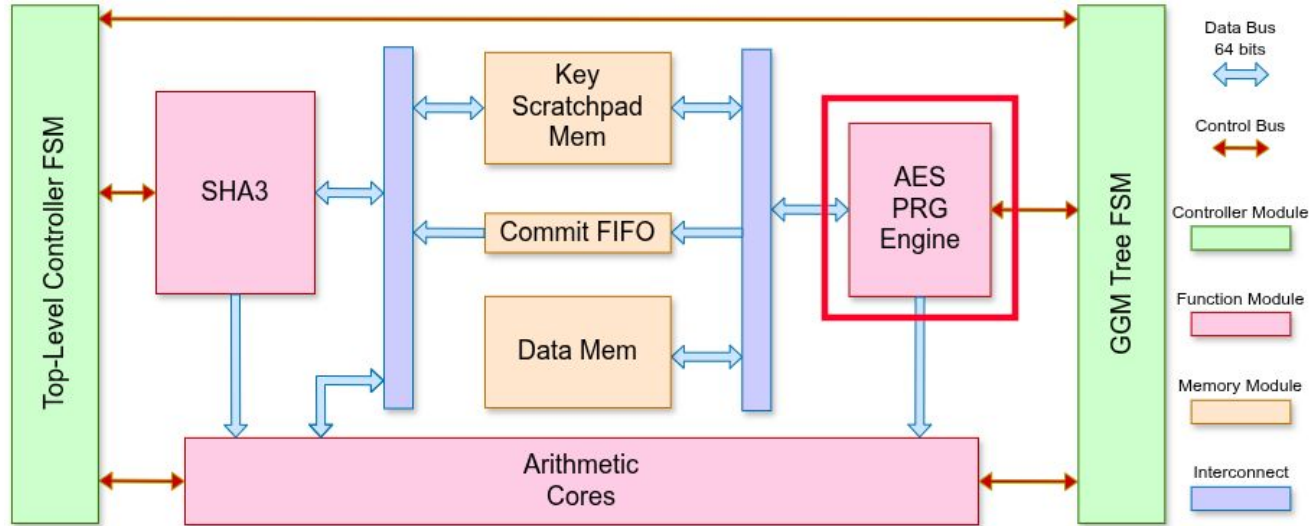
$$\text{mem}_{\text{req}}(N, \tau, K) = \begin{cases} 2 \cdot N \cdot \tau - 1, & \text{if } K = 0, \\ \boxed{\frac{N \cdot \tau}{2^{K-1}}} + \boxed{\frac{N}{4} \cdot (K+1)} - 1, & \text{if } K \neq 0. \end{cases}$$

Total memory vs K for $L = N \cdot \tau$ ($\tau \in \{17, 26, 36\}$, $N = 256$)

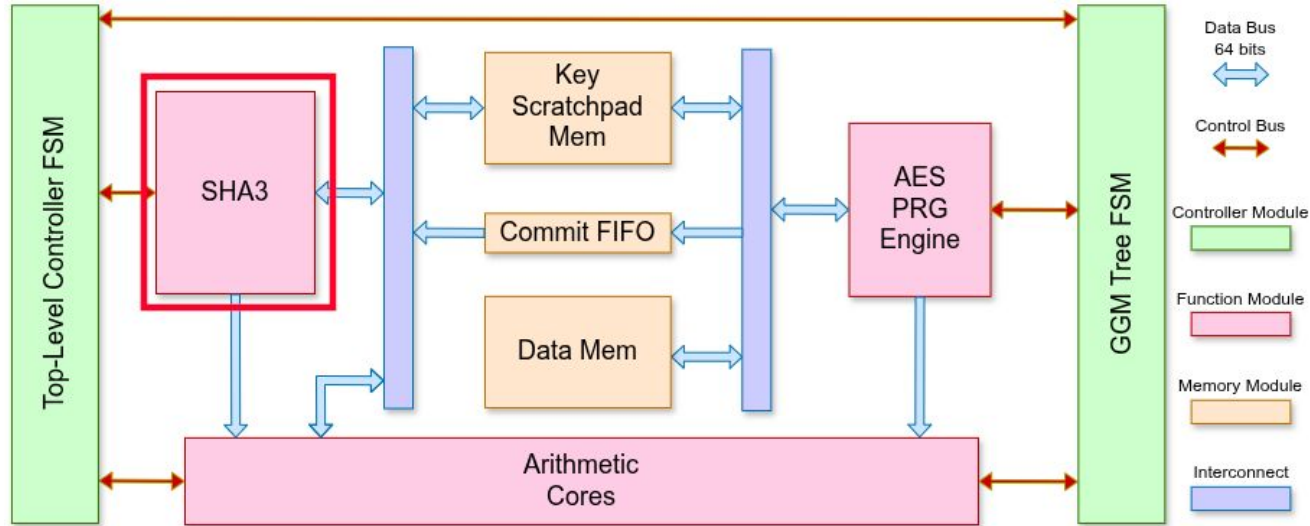


$$N = 256$$
$$\tau = \{17, 26, 36\}$$

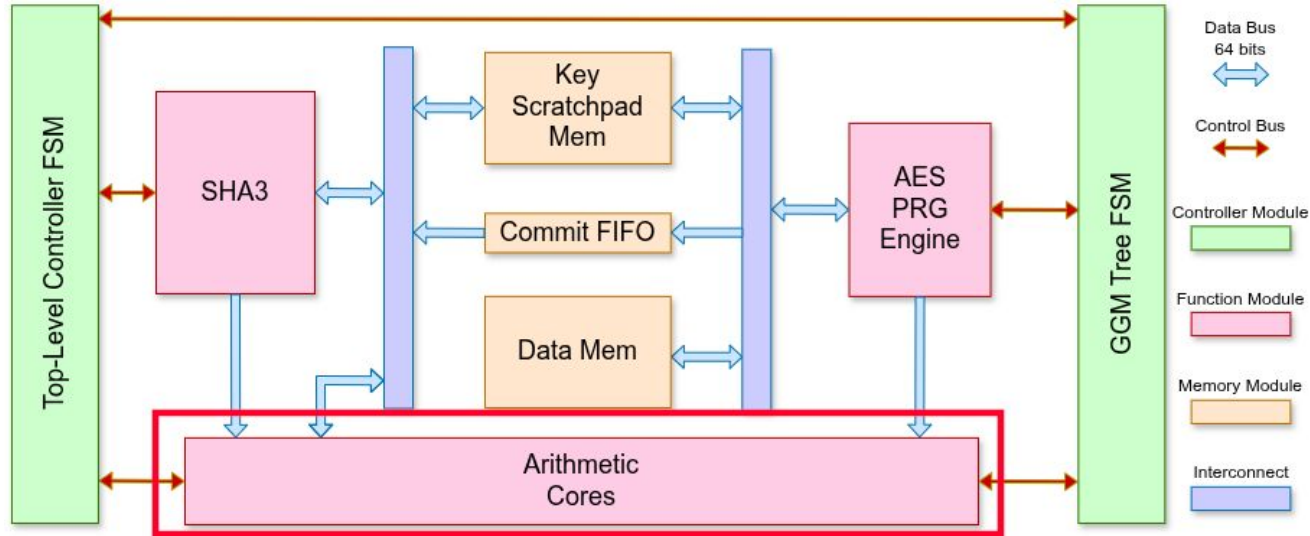
Top-Level Hardware Architecture



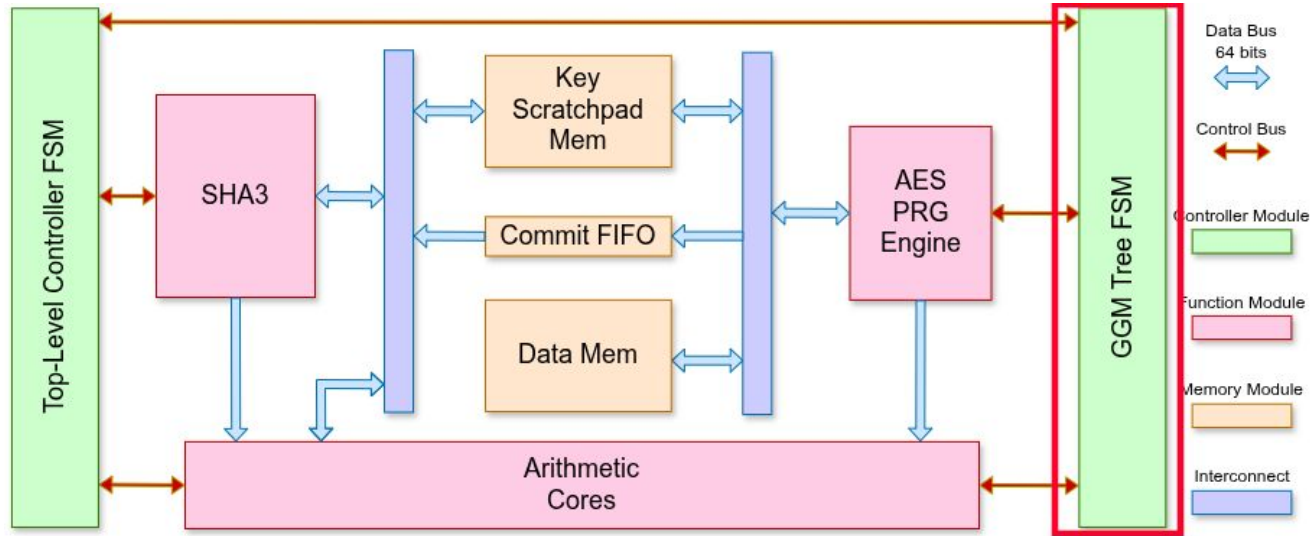
Top-Level Hardware Architecture



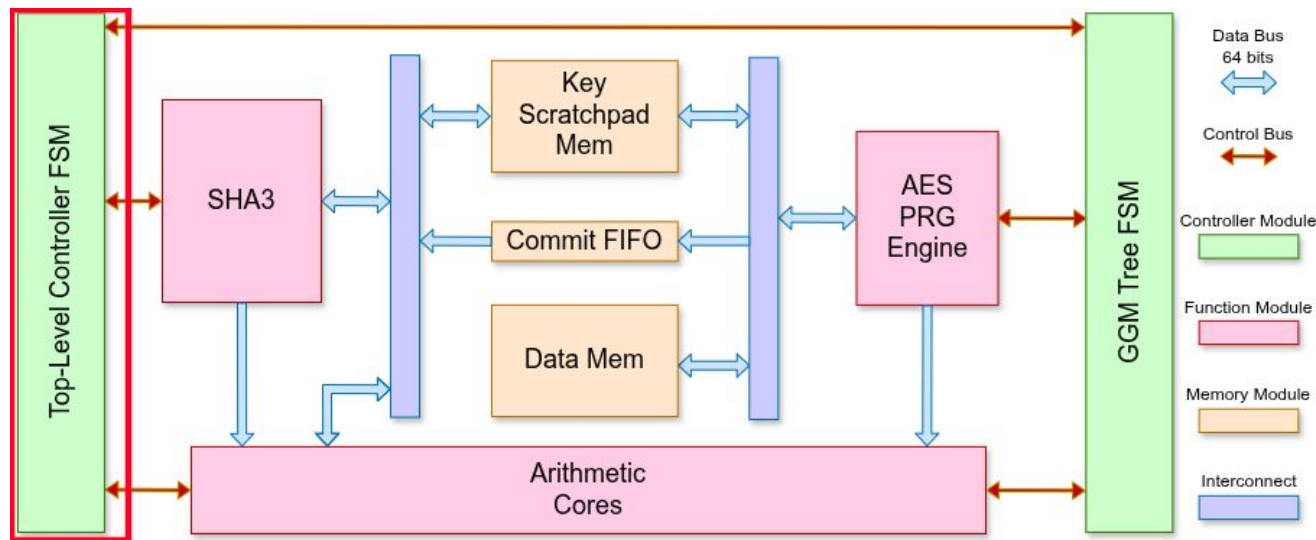
Top-Level Hardware Architecture



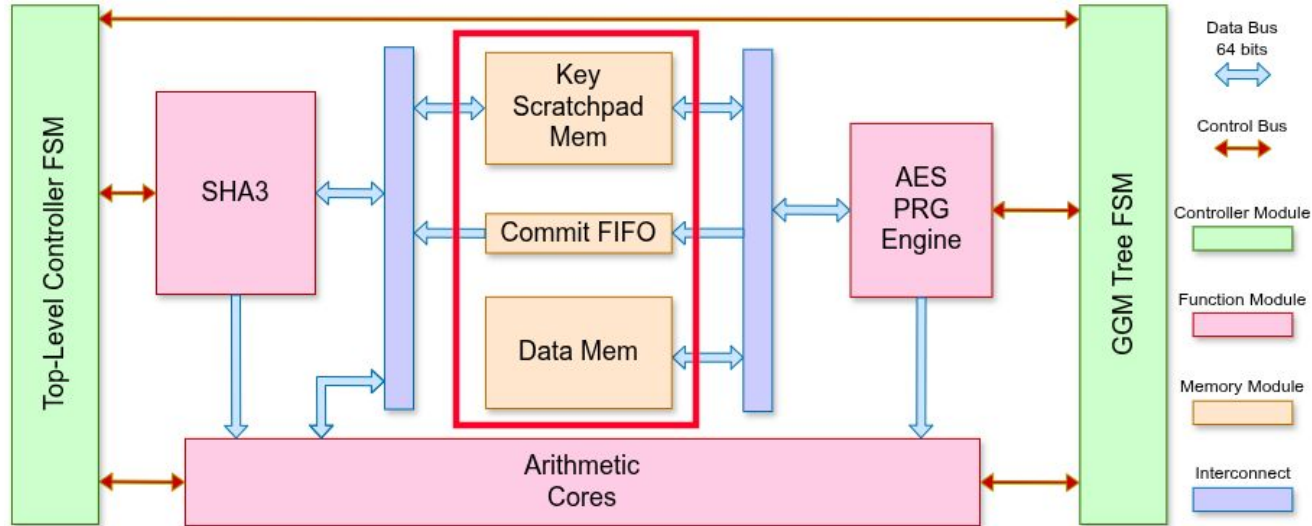
Top-Level Hardware Architecture



Top-Level Hardware Architecture



Top-Level Hardware Architecture



Mirath Variants & Parameter Sets

Two variants of Mirath

- **Threshold Computation in-the-Head (TCitH)**
- Vole-in-the-Head

Short and **fast** parameter sets

Two finite fields supported

- $q = 16$ (Mirath-a)
- **$q = 2$ (Mirath-b)**

Mirath-TCitH-b-fast

Mirath KeyGen

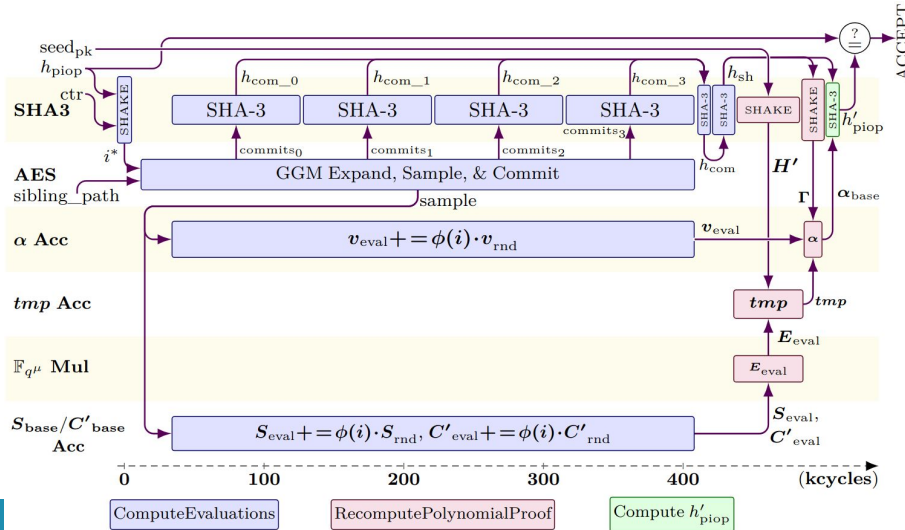


- Computationally inexpensive relative to Sign / Verify
- ~98% of clock cycles: generating $\mathbf{H}'$ (via SHAKE XOF)
- Constant time
- $\mathbf{H}'$ (59 to 200 KB) consumed *on-the-fly* to limit on-chip RAM consumption

Algorithm 19 Key generation algorithm.

KeyGen()

```
1 : seedsk ← TRG.Seed( $\lambda$ )
2 : seedpk ← TRG.Seed( $\lambda$ )
3 :  $\mathbf{S}, \mathbf{C}' \leftarrow \text{ExpandSeedSecretMatrices}(\text{seed}_{\text{sk}}) \triangleright \mathbf{S} \in \mathbb{F}_q^{m \times r}, \mathbf{C}' \in \mathbb{F}_q^{r \times (n-r)}$ 
4 :  $\mathbf{H}' \leftarrow \text{ExpandSeedPublicMatrix}(\text{seed}_{\text{pk}}) \triangleright \mathbf{H}' \in \mathbb{F}_q^{(m \cdot n - k) \times k}$ 
5 :  $\mathbf{y} \leftarrow \text{ComputeY}(\mathbf{S}, \mathbf{C}', \mathbf{H}') \triangleright \mathbf{y} \in \mathbb{F}_q^{(mn-k) \times 1}$ 
6 : pk ← (seedpk,  $\mathbf{y}$ )
7 : sk ← (seedsk, seedpk)
8 : return (pk, sk)
```



Algorithm 23 Verification algorithm.

Verify(pk, σ , msg)

// Step 0: Initialization (parsing and expansion).

1 : (salt, ctr, h_{piop} , π_{BAVC} , aux, α_{mid}) $\leftarrow$ ParseSignature(σ)

2 : (H' , y) $\leftarrow$ DecompressPK(pk)

// Step 1: Computing Opened Evaluations.

3 : (evals, point, h_{sh} , v_{grinding}) $\leftarrow$ ComputeEvaluations(salt, ctr, h_{piop} , π_{BAVC} , aux)

// Step 2: Recomputation of the Polynomial Proof $P_{\alpha}(X)$.

4 : $\Gamma \leftarrow$ ExpandChallengeMatrix(h_{piop}) $\triangleright \Gamma \in \mathbb{F}_{q^\mu}^{\rho \times (mn-k)}$

5 : for e from 0 to $(\tau - 1)$ do

6 : $\alpha_{\text{base}}[e] \leftarrow$ RecomputePolynomialProof(point[e], evals[e],
 $\Gamma, (H', y), \alpha_{\text{mid}}[e])$

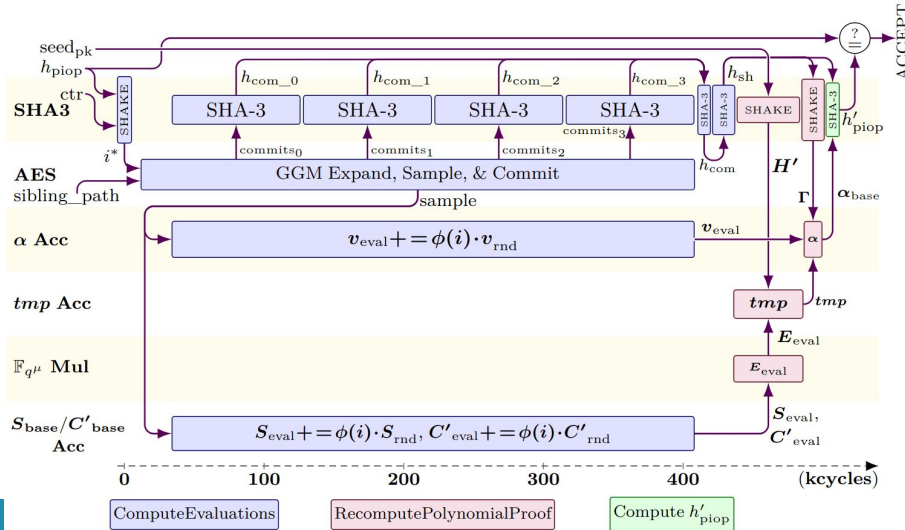
// Step 3: Verification.

7 : $h'_{\text{piop}} \leftarrow$ Hash₂(pk, salt, msg, h_{sh} , $\alpha_{\text{mid}}[0]$, $\alpha_{\text{base}}[0]$,
 $\dots$, $\alpha_{\text{mid}}[\tau - 1]$, $\alpha_{\text{base}}[\tau - 1]$)

8 : return ($h_{\text{piop}} \stackrel{?}{=} h'_{\text{piop}}$) and ($v_{\text{grinding}} \stackrel{?}{=} 0$)

Step 0: Initialization

- Essentially free in Hardware



Algorithm 23 Verification algorithm.

Verify(pk, σ , msg)

// Step 0: Initialization (parsing and expansion).

- 1 : (salt, ctr, h_{piop} , π_{BAVC} , aux, α_{mid}) $\leftarrow$ ParseSignature(σ)
- 2 : (H' , y) $\leftarrow$ DecompressPK(pk)

// Step 1: Computing Opened Evaluations.

- 3 : (evals, point, h_{sh} , v_{grinding}) $\leftarrow$ ComputeEvaluations(salt, ctr, h_{piop} , π_{BAVC} , aux)

// Step 2: Recomputation of the Polynomial Proof $P_{\alpha}(X)$.

- 4 : $\Gamma \leftarrow$ ExpandChallengeMatrix(h_{piop}) $\triangleright \Gamma \in \mathbb{F}_{q^{\mu}}^{\rho \times (mn-k)}$
- 5 : for e from 0 to $(\tau - 1)$ do
- 6 : $\alpha_{\text{base}}[e] \leftarrow$ RecomputePolynomialProof(point[e], evals[e], Γ , (H' , y), $\alpha_{\text{mid}}[e]$)

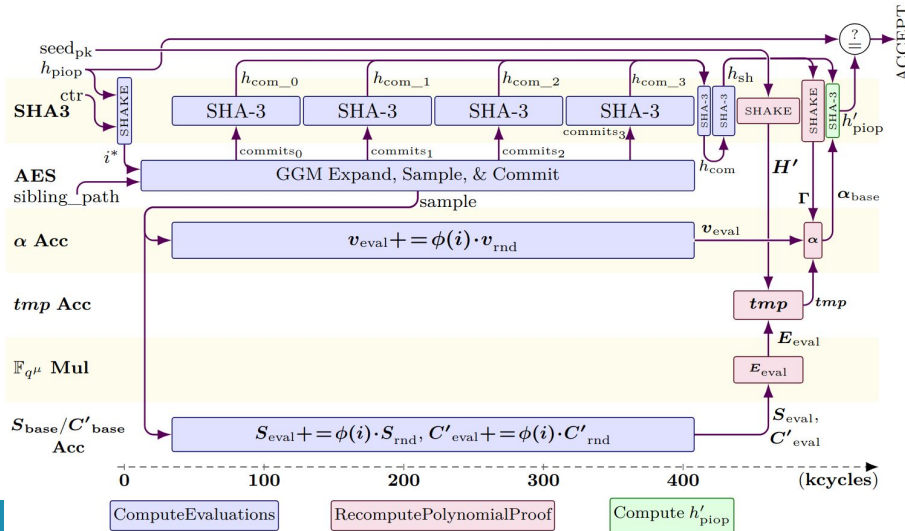
// Step 3: Verification.

- 7 : $h'_{\text{piop}} \leftarrow$ Hash₂(pk, salt, msg, h_{sh} , $\alpha_{\text{mid}}[0]$, $\alpha_{\text{base}}[0]$, $\dots$, $\alpha_{\text{mid}}[\tau - 1]$, $\alpha_{\text{base}}[\tau - 1]$)
- 8 : return ($h_{\text{piop}} \stackrel{?}{=} h'_{\text{piop}}$) and ($v_{\text{grinding}} \stackrel{?}{=} 0$)

Mirath Verify

Step 1: Compute Evaluations

- ~25% slower than Sign Step 1 (partial GGM expand)
- ~85% of total Verify latency



Algorithm 23 Verification algorithm.

Verify(pk, σ , msg)

// Step 0: Initialization (parsing and expansion).

1 : (salt, ctr, h_{piop} , π_{BAVC} , aux, α_{mid}) $\leftarrow$ ParseSignature(σ)

2 : (H' , y) $\leftarrow$ DecompressPK(pk)

// Step 1: Computing Opened Evaluations.

3 : (evals, point, h_{sh} , $v_{grinding}$) $\leftarrow$ ComputeEvaluations(salt, ctr, h_{piop} , π_{BAVC} , aux)

// Step 2: Recomputation of the Polynomial Proof $P_\alpha(X)$.

4 : $\Gamma \leftarrow$ ExpandChallengeMatrix(h_{piop}) $\triangleright \Gamma \in \mathbb{F}_{q^\mu}^{\rho \times (mn-k)}$

5 : for e from 0 to $(\tau - 1)$ do

6 : $\alpha_{base}[e] \leftarrow$ RecomputePolynomialProof(point[e], evals[e],
 $\Gamma, (H', y), \alpha_{mid}[e]$)

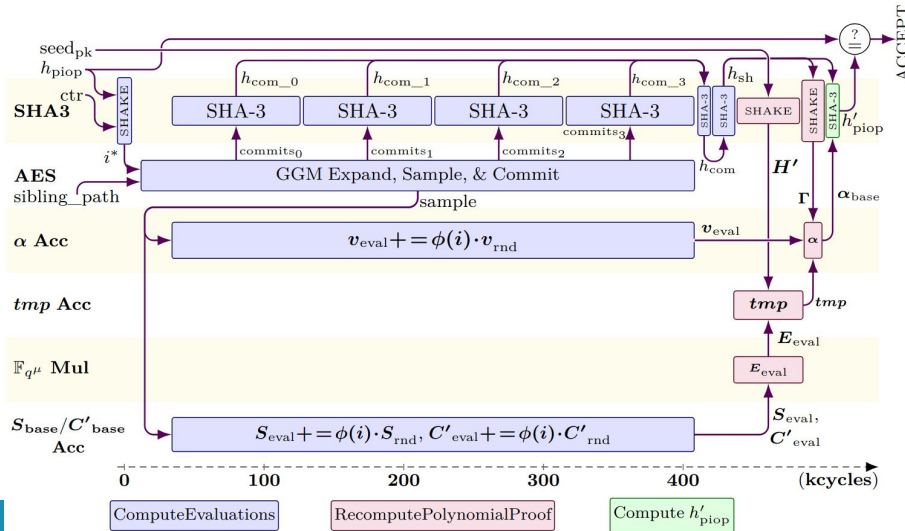
// Step 3: Verification.

7 : $h'_{piop} \leftarrow$ Hash₂(pk, salt, msg, h_{sh} , $\alpha_{mid}[0]$, $\alpha_{base}[0]$,
 $\dots$, $\alpha_{mid}[\tau - 1]$, $\alpha_{base}[\tau - 1]$)

8 : return ($h_{piop} \stackrel{?}{=} h'_{piop}$ and ($v_{grinding} \stackrel{?}{=} 0$))

Step 2: Recompute Polynomial Proof

- ~14.5% of total Verify latency
- MPC Arithmetic datapath replicated τ times



Algorithm 23 Verification algorithm.

Verify(pk, σ , msg)

// Step 0: Initialization (parsing and expansion).

1 : (salt, ctr, h_{piop} , π_{BAVC} , aux, α_{mid}) $\leftarrow$ ParseSignature(σ)

2 : (H' , y) $\leftarrow$ DecompressPK(pk)

// Step 1: Computing Opened Evaluations.

3 : (evals, point, h_{sh} , $v_{grinding}$) $\leftarrow$ ComputeEvaluations(salt, ctr, h_{piop} , π_{BAVC} , aux)

// Step 2: Recomputation of the Polynomial Proof $P_\alpha(X)$.

4 : $\Gamma \leftarrow$ ExpandChallengeMatrix(h_{piop}) $\triangleright \Gamma \in \mathbb{F}_{q^\mu}^{\rho \times (mn-k)}$

5 : for e from 0 to $(\tau - 1)$ do

6 : $\alpha_{base}[e] \leftarrow$ RecomputePolynomialProof(point[e], evals[e], Γ , (H' , y), $\alpha_{mid}[e]$)

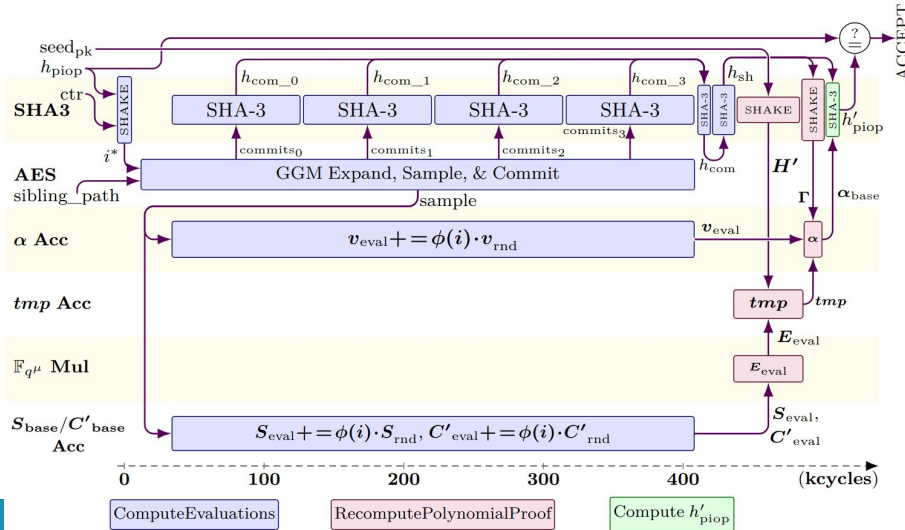
// Step 3: Verification.

7 : $h'_{piop} \leftarrow$ Hash2(pk, salt, msg, h_{sh} , $\alpha_{mid}[0]$, $\alpha_{base}[0]$,
 $\dots$, $\alpha_{mid}[\tau - 1]$, $\alpha_{base}[\tau - 1]$)

8 : return ($h_{piop} \stackrel{?}{=} h'_{piop}$) and ($v_{grinding} \stackrel{?}{=} 0$)

Step 3: Verification

- ~0.5% of total Verify latency



Algorithm 23 Verification algorithm.

Verify(pk, σ , msg)

// Step 0: Initialization (parsing and expansion).

1 : (salt, ctr, h_{piop} , π_{BAVC} , aux, α_{mid}) $\leftarrow$ ParseSignature(σ)

2 : (H' , y) $\leftarrow$ DecompressPK(pk)

// Step 1: Computing Opened Evaluations.

3 : (evals, point, h_{sh} , $v_{grinding}$) $\leftarrow$ ComputeEvaluations(salt, ctr, h_{piop} , π_{BAVC} , aux)

// Step 2: Recomputation of the Polynomial Proof $P_\alpha(X)$.

4 : $\Gamma \leftarrow$ ExpandChallengeMatrix(h_{piop}) $\triangleright \Gamma \in \mathbb{F}_{q^\mu}^{\rho \times (mn-k)}$

5 : for e from 0 to $(\tau - 1)$ do

6 : $\alpha_{base}[e] \leftarrow$ RecomputePolynomialProof(point[e], evals[e], Γ , (H' , y), $\alpha_{mid}[e]$)

// Step 3: Verification.

7 : $h'_{piop} \leftarrow$ Hash₂(pk, salt, msg, h_{sh} , $\alpha_{mid}[0]$, $\alpha_{base}[0]$, $\dots$, $\alpha_{mid}[\tau - 1]$, $\alpha_{base}[\tau - 1]$)

8 : return ($h_{piop} \stackrel{?}{=} h'_{piop}$) and ($v_{grinding} \stackrel{?}{=} 0$)

Comparison with Software Implementations



COSIC

Scheme	Parameter Set	Resources		Freq. (MHz)	KeyGen (Mcycles/ms)	Sign (Mcycles/ms)	Verify (Mcycles/ms)	ATP ^a
		LUT/FF/DSP/BRAM	Area ^a					
Mirath (This work)	1b-fast	15,400/9,445/0/42.0	28,000	254	0.0161/0.0634	0.554/2.181	0.5/1.969	118.0
	5b-fast	30,415/18,223/0/96.0	59,215	218	0.0607/0.2784	1.3726/6.296	1.2116/5.558	718.4
Mirath^b [AAB ⁺ 25a, AAB ⁺ 25b]	1b-fast	-/-/-/-	-	3000	1.8/0.60	1600/533.3	2600/866.67	-
	5b-fast	-/-/-/-	-	3000	6.5/2.17	8400/2800.0	13000/4333.3	-
	1b-fast-AVX2	-/-/-/-	-	3000	0.5/0.17	9.8/3.27	5.5/1.83	-
	5b-fast-AVX2	-/-/-/-	-	3000	2.0/0.67	70.1/23.37	51.7/17.23	-
ML-DSA [NKV25]	L2	54,942/29,309/16/29.0	65,242	230	0.0049/0.0213	0.024/0.1043	0.0066/0.0287	10.07
	L5	54,942/29,309/16/29.0	65,242	230	0.014/0.0609	0.038/0.1652	0.0146/0.0635	18.89
Falcon [OZZ ⁺ 25]	L1	79,065/46,389/226/45.0	115,165	65	-/-	0.16/0.64	-/-	-
SPHINCS⁺ [ALCZ20]	128f	47,991/72,505/1/11.5	51,541	250/500	-/-	-/1.01	-/0.16	60.3
	256f	51,009/74,539/1/22.5	57,859	250/500	-/-	-/2.52	-/0.21	158.0
CROSS [KAB ⁺ 25]	RS DP-1-f	26,741/11,678/0/45.0	40,241	118	-/0.035	-/0.478	-/0.409	37.1
	RS DP-5-f	34,626/15,331/0/91.5	62,076	113	-/0.028	-/1.20	-/1.14	147.0
LESS [BWMG23]	L1-s	54,800/39,900/0/59.5	72,650	200	0.174/0.87	4.17/20.83	4.137/20.69	3,709.6
	L5-s	104,300/76,700/0/167.5	154,550	143	0.247/1.73	87.16/610.13	87.013/609.09	188,697.8
MAYO [HSK ⁺ 24]	L1-f	106,000/38,000/2/45.5	119,850	75	0.0058/0.08	0.031/0.43	0.0046/0.05	67.1
UOV [BCD ⁺ 25]	Is-pkc+skc	28,947/24,444/2/66.0	48,947	90.8	11.1/122.25	0.84/9.251	0.284/3.128	6,589.7
	V-pkc+skc	77,352/38,217/4/359.0	185,452	92.5	39.1/422.70	3.31/35.784	1.92/20.757	88,876.2
SDitH [DHSY24]	L1-GF256	16,592/8,778/0/164.5	65,942	164	0.055/0.34	6.73/41.04	8.688/52.98	6,222.3
	L5-GF256	23,323/14,962/0/520.5	179,473	164	0.083/0.51	17.48/106.60	24.940/152.10	46,521.2
PICNIC [KRR ⁺ 20]	L1	90,337/23,105/0/52.5	106,087	125	-/-	0.03/0.25	0.030/0.24	52.0
	L5	167,530/33,164/0/98.5	197,080	125	-/-	0.15/1.24	0.147/1.17	475.0

ATP Comparison (Add. DSAs Round 2)



COSIC

Scheme	Parameter Set	Resources		Freq. (MHz)	KeyGen (Mcycles/ms)	Sign (Mcycles/ms)	Verify (Mcycles/ms)	ATP ^a
		LUT/FF/DSP/BRAM	Area ^a					
Mirath (This work)	1b-fast	15,400/9,445/0/42.0	28,000	254	0.0161/0.0634	0.554/2.181	0.5/1.969	118.0
	5b-fast	30,415/18,223/0/96.0	59,215	218	0.0607/0.2784	1.3726/6.296	1.2116/5.558	718.4
Mirath^b [AAB ⁺ 25a, AAB ⁺ 25b]	1b-fast	-/-/-/-	-	3000	1.8/0.60	1600/533.3	2600/866.67	-
	5b-fast	-/-/-/-	-	3000	6.5/2.17	8400/2800.0	13000/4333.3	-
	1b-fast-AVX2	-/-/-/-	-	3000	0.5/0.17	9.8/3.27	5.5/1.83	-
	5b-fast-AVX2	-/-/-/-	-	3000	2.0/0.67	70.1/23.37	51.7/17.23	-
ML-DSA [NKV25]	L2	54,942/29,309/16/29.0	65,242	230	0.0049/0.0213	0.024/0.1043	0.0066/0.0287	10.07
	L5	54,942/29,309/16/29.0	65,242	230	0.014/0.0609	0.038/0.1652	0.0146/0.0635	18.89
Falcon [OZZ ⁺ 25]	L1	79,065/46,389/226/45.0	115,165	65	-/-	0.16/0.64	-/-	-
SPHINCS⁺ [ALCZ20]	128f	47,991/72,505/1/11.5	51,541	250/500	-/-	-/1.01	-/0.16	60.3
	256f	51,009/74,539/1/22.5	57,859	250/500	-/-	-/2.52	-/0.21	158.0
CROSS [KAB ⁺ 25]	RS DP-1-f	26,741/11,678/0/45.0	40,241	118	-/0.035	-/0.478	-/0.409	37.1
	RS DP-5-f	34,626/15,331/0/91.5	62,076	113	-/0.028	-/1.20	-/1.14	147.0
LESS [BWMG23]	L1-s	54,800/39,900/0/59.5	72,650	200	0.174/0.87	4.17/20.83	4.137/20.69	3,709.6
	L5-s	104,300/76,700/0/167.5	154,550	143	0.247/1.73	87.16/610.13	87.013/609.09	188,697.8
MAYO [HSK ⁺ 24]	L1-f	106,000/38,000/2/45.5	119,850	75	0.0058/0.08	0.031/0.43	0.0046/0.05	67.1
UOV [BCD ⁺ 25]	Is-pkc+skc	28,947/24,444/2/66.0	48,947	90.8	11.1/122.25	0.84/9.251	0.284/3.128	6,589.7
	V-pkc+skc	77,352/38,217/4/359.0	185,452	92.5	39.1/422.70	3.31/35.784	1.92/20.757	88,876.2
SDitH [DHSY24]	L1-GF256	16,592/8,778/0/164.5	65,942	164	0.055/0.34	6.73/41.04	8.688/52.98	6,222.3
	L5-GF256	23,323/14,962/0/520.5	179,473	164	0.083/0.51	17.48/106.60	24.940/152.10	46,521.2
PICNIC [KRR ⁺ 20]	L1	90,337/23,105/0/52.5	106,087	125	-/-	0.03/0.25	0.030/0.24	52.0
	L5	167,530/33,164/0/98.5	197,080	125	-/-	0.15/1.24	0.147/1.17	475.0

FPGA Area Comparison



COSIC

Scheme	Parameter Set	Resources		Freq. (MHz)	KeyGen (Mcycles/ms)	Sign (Mcycles/ms)	Verify (Mcycles/ms)	ATP ^a
		LUT/FF/DSP/BRAM	Area ^a					
Mirath (This work)	1b-fast	15,400/9,445/0/42.0	28,000	254	0.0161/0.0634	0.554/2.181	0.5/1.969	118.0
	5b-fast	30,415/18,223/0/96.0	59,215	218	0.0607/0.2784	1.3726/6.296	1.2116/5.558	718.4
Mirath^b [AAB ⁺ 25a, AAB ⁺ 25b]	1b-fast	-/-/-/-	-	3000	1.8/0.60	1600/533.3	2600/866.67	-
	5b-fast	-/-/-/-	-	3000	6.5/2.17	8400/2800.0	13000/4333.3	-
	1b-fast-AVX2	-/-/-/-	-	3000	0.5/0.17	9.8/3.27	5.5/1.83	-
	5b-fast-AVX2	-/-/-/-	-	3000	2.0/0.67	70.1/23.37	51.7/17.23	-
ML-DSA [NKV25]	L2	54,942/29,309/16/29.0	65,242	230	0.0049/0.0213	0.024/0.1043	0.0066/0.0287	10.07
	L5	54,942/29,309/16/29.0	65,242	230	0.014/0.0609	0.038/0.1652	0.0146/0.0635	18.89
Falcon [OZZ ⁺ 25]	L1	79,065/46,389/226/45.0	115,165	65	-/-	0.16/0.64	-/-	-
SPHINCS⁺ [ALCZ20]	128f	47,991/72,505/1/11.5	51,541	250/500	-/-	-/1.01	-/0.16	60.3
	256f	51,009/74,539/1/22.5	57,859	250/500	-/-	-/2.52	-/0.21	158.0
CROSS [KAB ⁺ 25]	RS DP-1-f	26,741/11,678/0/45.0	40,241	118	-/0.035	-/0.478	-/0.409	37.1
	RS DP-5-f	34,626/15,331/0/91.5	62,076	113	-/0.028	-/1.20	-/1.14	147.0
LESS [BWMG23]	L1-s	54,800/39,900/0/59.5	72,650	200	0.174/0.87	4.17/20.83	4.137/20.69	3,709.6
	L5-s	104,300/76,700/0/167.5	154,550	143	0.247/1.73	87.16/610.13	87.013/609.09	188,697.8
MAYO [HSK ⁺ 24]	L1-f	106,000/38,000/2/45.5	119,850	75	0.0058/0.08	0.031/0.43	0.0046/0.05	67.1
UOV [BCD ⁺ 25]	Is-pkc+skc	28,947/24,444/2/66.0	48,947	90.8	11.1/122.25	0.84/9.251	0.284/3.128	6,589.7
	V-pkc+skc	77,352/38,217/4/359.0	185,452	92.5	39.1/422.70	3.31/35.784	1.92/20.757	88,876.2
SDitH [DHSY24]	L1-GF256	16,592/8,778/0/164.5	65,942	164	0.055/0.34	6.73/41.04	8.688/52.98	6,222.3
	L5-GF256	23,323/14,962/0/520.5	179,473	164	0.083/0.51	17.48/106.60	24.940/152.10	46,521.2
PICNIC [KRR ⁺ 20]	L1	90,337/23,105/0/52.5	106,087	125	-/-	0.03/0.25	0.030/0.24	52.0
	L5	167,530/33,164/0/98.5	197,080	125	-/-	0.15/1.24	0.147/1.17	475.0

MPCitH Evolution



1st NIST PQC Process (additive secret sharing): Picnic^[1]

- NIST: “[...] future cryptosystems that evolve out of the MPCitH paradigm may eventually prove significantly superior to the third-round Picnic design”^[2]

Additional Signatures Round 1 (hypercube & threshold variants prominent): MIRA, MiRitH, MQOM, SDitH, ...

- SDitH in Hardware^[3]: Signing & Verification latency dominated by polynomial evaluations. Large on-chip memory requirements.

Additional Signatures Round 2 (TCitH & Vole-in-the-Head variants prominent): Mirath, MQOM, SDitH, ...

- **MPSpeed: This work**

^[1] Final Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process

^[3] Sanjay Deshpande, James Howe, Jakub Szefer, and Dongze Yue. Sdith in

hardware. IACR Cryptol. ePrint Arch., page 69, 2024.

- Most computationally expensive routine
- Step 0 (computationally) identical to KeyGen
- Steps 1 - 3 largely similar to Verification

Algorithm 22 Signing algorithm.

```

Sign(sk, msg)

    // Step 0: Initialization.
    1 :  $(H', pk, S, C') \leftarrow \text{DecompressSK}(sk)$ 
    2 :  $\text{salt} \leftarrow \text{TRG.Seed}(2\lambda)$ 
    3 :  $\text{rseed} \leftarrow \text{TRG.Seed}(\lambda)$ 

    // Step 1: Build & Commit to Witness Polynomials.
    4 :  $(\text{base}, v, h_{\text{sh}}, \text{key}, \text{aux}) \leftarrow \text{CommitWitnessPolynomials}(\text{salt}, \text{rseed}, S, C')$ 

    // Step 2: Compute the polynomial proof  $P_{\alpha}(X)$ .
    5 :  $\Gamma \leftarrow \text{ExpandChallengeMatrix}(h_{\text{sh}}) \triangleright \Gamma \in \mathbb{R}_{q^{\mu}}^{p \times (mn-k)}$ 
    6 : for  $e$  from 0 to  $(\tau - 1)$  do
    7 :    $(\alpha_{\text{mid}}[e], \alpha_{\text{base}}[e]) \leftarrow \text{ComputePolynomialProof}(\text{base}[e], v[e]S, C', \Gamma, H')$ 
    8 :  $h_{\text{piop}} \leftarrow \text{Hash}_2(pk, \text{salt}, \text{msg}, h_{\text{sh}},$ 
        $\alpha_{\text{mid}}[0], \alpha_{\text{base}}[0], \dots, \alpha_{\text{mid}}[\tau - 1], \alpha_{\text{base}}[\tau - 1])$ 

    // Step 3: Open random evaluations of the polynomials  $P_S(X)$ ,  $P_C(X)$  and  $P_v(X)$ .
    9 :  $(\text{ctr}, \pi_{\text{BAVC}}) \leftarrow \text{OpenRandomEvaluations}(\text{key}, h_{\text{piop}})$ 
    10 :  $\sigma \leftarrow \text{UnparseSignature}(\text{salt}, \text{ctr}, h_{\text{piop}}, \pi_{\text{BAVC}}, \text{aux}, \alpha_{\text{mid}})$ 
    11 : return  $\sigma$ 
    
```