SDitH in Hardware

Sanjay Deshpande, James Howe, Jakub Szefer, and Dongze (Steven) Yue

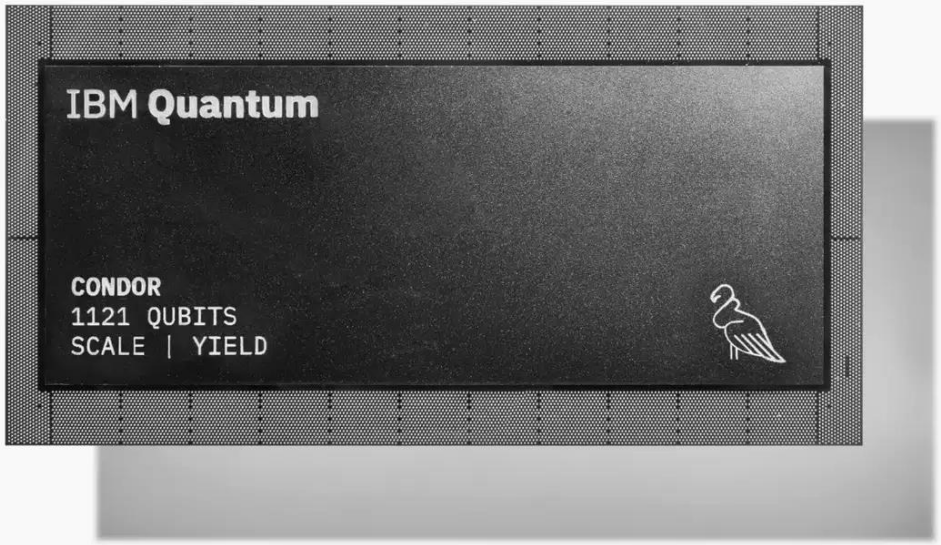
5th NIST PQC Standardization Conference
April 12, 2024

Computer Architecture and Security Lab (CASLAB)
https://caslab.io

SANDBOXAQ

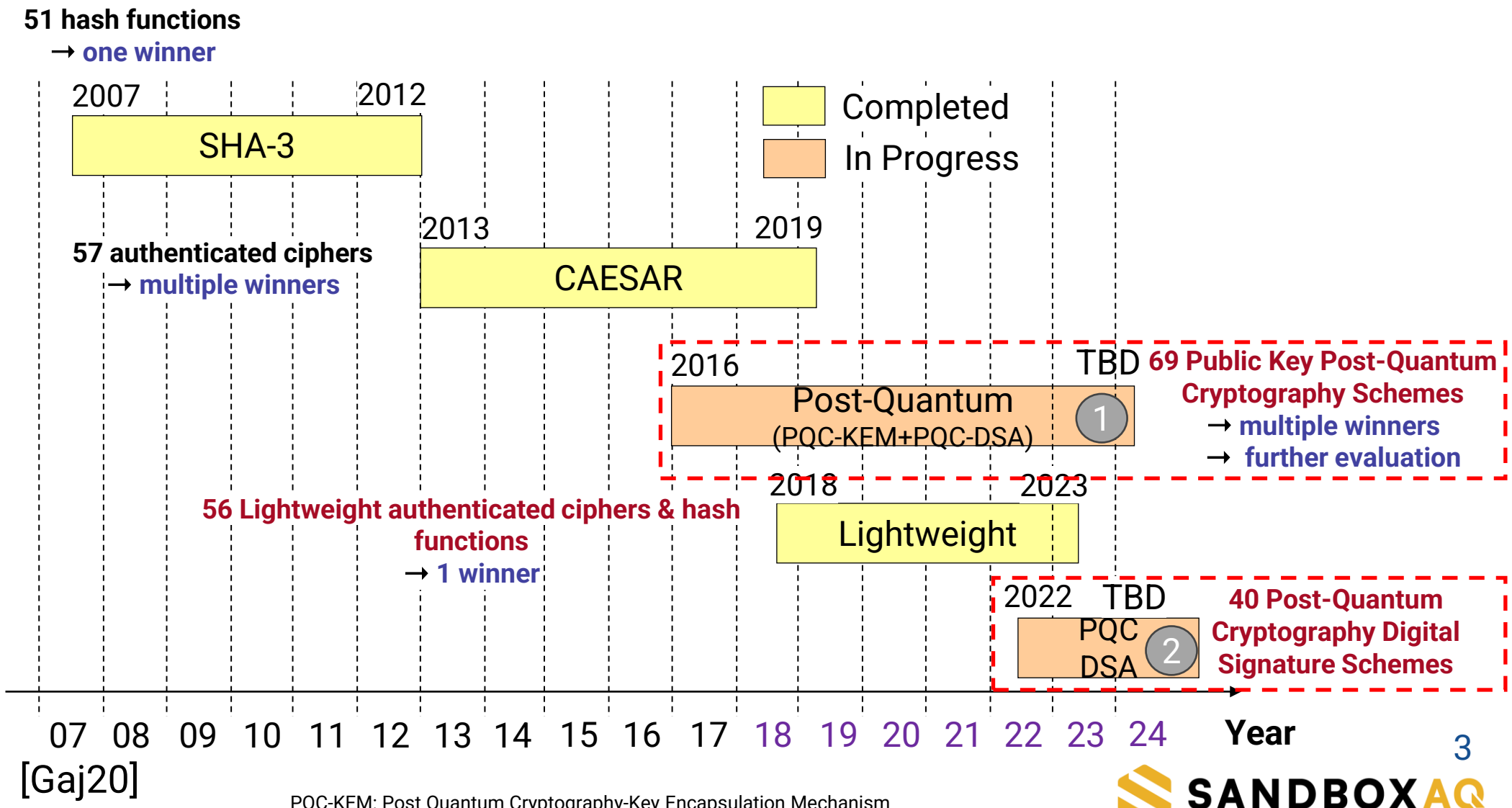
Motivation

- Quantum Computing holds tremendous potential that could solve complex problems that are out of reach for current high-performance computers
  - Life-saving pharmaceuticals
  - Green-battery technology
- However, they also pose significant cybersecurity risks
  - Can easily break existing standards of public key cryptography
  - Can jeopardize payment systems, encrypted chat, emails, etc.
- The Quantum Insider's report from 2022 forecasts the quantum security market worth $10 billion by 2030
- Currently, we do not have large-scale quantum computers
  - In 2023, IBM announced the 1,121-qubit quantum processor "Condor"
- Hence, there is a need for Quantum-safe Cryptography!
  - Post quantum cryptography emerges as a beacon of hope



Condor Image Source: IBM

NIST Post Quantum Cryptography Standardization Effort



[Gaj20]

PQC-KEM: Post Quantum Cryptography-Key Encapsulation Mechanism
PQC-DSA: Post Quantum Cryptography-Digital Signature Algorithm

Outline

- Introduction
  - SDitH Signature Scheme
- Hardware Design and Challenges
- Comparison with Related and Relevant Work
- Conclusion and Future Work

Introduction

SDitH Parameter Sets

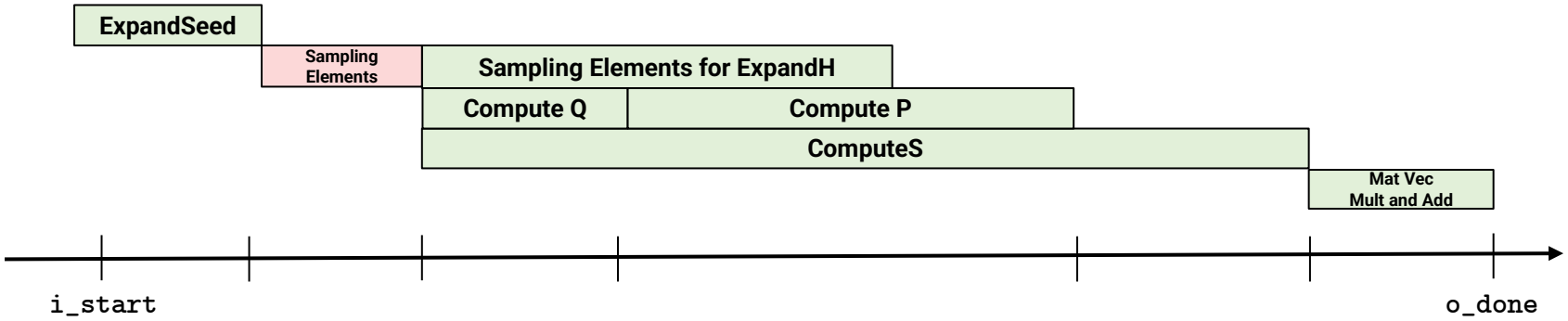
- Two Variants of the Algorithm
  - **Hypercube**
  - Threshold
- Three Security Levels
- Two Syndrome Decoding Fields
  - **GF256 and GF251**
- **d=2** splits for L3 and L5 Parameter Sets

	SDitH Parameters	NIST Security Categories		
		L1	L3	L5
Signature Parameters	NIST Security Level	143	207	272
	λ (Security Target)	128	192	256
	N^D (# Secret Shares)	2^8	2^8	2^8
	τ (Repetition Rate)	17	26	34
Syndrome Decoding	q (SD Base Field Size)	251/256	251/256	251/256
	m (Code Length)	230	352	480
	k (Vector Dimension)	126	193	278
	w (Hamming Weight Bound)	79	120	150
	d (d Splitting Size)	1	2	2
Multi-Party Computation	t (# Random evaluation points)	3	3	4
	$\mathbb{F}_q$ (SD base field)	$\mathbb{F}_q$	$\mathbb{F}_q$	$\mathbb{F}_q$
	η (Field extension size)	4	4	4
	$\mathbb{F}_{\text{points}}$ (Field extension of $\mathbb{F}_q$)	$\mathbb{F}_{q^\eta}$	$\mathbb{F}_{q^\eta}$	$\mathbb{F}_{q^\eta}$
	p (False positive probability)	$2^{-71.2}$	$2^{-72.4}$	$2^{-94.8}$
Output Sizes	pk Size (in Bytes)	120	183	234
	sk Size (in Bytes)	404	616	812
	Max Signature Size (in Bytes)	8 260	19 206	33 448

SDitH Key Generation

Algorithm 1 SDitH – Key Generation

1: $\mathsf{seed}_{\mathsf{root}} \leftarrow \{0,1\}^{\lambda}$
2: $(\mathsf{seed}_{\mathsf{wit}}, \mathsf{seed}_H) \leftarrow \mathrm{ExpandSeed}(\mathsf{salt} := 0, \mathsf{seed}_{\mathsf{root}}, 2)$
3: $(\boldsymbol{Q}, \boldsymbol{S}, \boldsymbol{P}) \leftarrow \mathrm{SampleWitness}(\mathsf{seed}_{\mathsf{wit}})$
4: $s = \mathrm{Serialize}(\boldsymbol{S})$
5: $(s_A, s_B) = \mathrm{Parse}(s, \mathbb{F}_q^k, \mathbb{F}_q^{m-k})$
6: $H' \leftarrow \mathrm{ExpandH}(\mathsf{seed}_H)$
7: $y = s_B + H' s_A$
8: $\boldsymbol{Q'} = \mathrm{TruncateQ}(\boldsymbol{Q})$
9: $\mathsf{wit\_plain} = \mathrm{Serialize}(s_A, \boldsymbol{Q'}, \boldsymbol{P})$
10: **return** $(pk = (\mathsf{seed}_H, y), sk = (\mathsf{seed}_H, y, \mathsf{wit\_plain}))$



Variable time due to rejection sampling

Constant time

SDitH Sign

- Only a little scope for parallelism at the algorithm level
- Processing Message (*m*) input happens much later in the algorithm
  - Hence, could be divided in to *Offline* and *Online* Parts

Algorithm 10 SD-in-the-Head – Hypercube Variant – Signature Algorithm

Input: a secret key $sk = (\mathsf{seed}_H, y, \mathsf{wit\_plain})$ and a message $m \in \{0,1\}^*$

$\mathsf{salt} \leftarrow \{0,1\}^{2\lambda}$
$\mathsf{mseed} \leftarrow \{0,1\}^{\lambda}$
$H' \leftarrow \mathrm{ExpandH}(\mathsf{seed}_H)$ ▷ $H' \in \mathbb{F}_q^{(n-k)\times k}$
$\{\mathsf{rseed}[e]\}_{e\in[1:\tau]} \leftarrow \mathrm{ExpandSeed}(\mathsf{salt}, \mathsf{mseed}, \tau)$ ▷ $\mathsf{rseed}[e] \in \{0,1\}^{\lambda}$
for $e \in [1:\tau]$ **do**
  $(\mathsf{seed}[e][i])_{i\in[1:2^D]} \leftarrow \mathrm{TreePRG}(\mathsf{salt}, \mathsf{rseed}[e])$
  $\mathsf{acc} = 0$ ▷ $\mathsf{acc} \in \mathbb{F}_q^{k+2w+t(2d+1)\eta}$
  $\mathsf{input\_mshare}[e][p] = 0$ for all $(e,p) \in [1:\tau] \times [1:D]$
  ▷ $\mathsf{input\_mshare}[e][i] \in \mathbb{F}_q^{k+2w+t(2d+1)\eta}$
  for $i \in [1:2^D]$ **do**
    if $i \neq 2^D$ **then**
      $\mathsf{input\_share}[e][i] \leftarrow \mathrm{SampleFieldElements}(\mathsf{salt}, \mathsf{seed}[e][i], k+2w+t(2d+1)\eta)$
      ▷ $\mathsf{input\_share}[e][i] \in \mathbb{F}_q^{k+2w+t(2d+1)\eta}$
      acc += $\mathsf{input\_share}[e][i]$
      $\mathsf{state}[e][i] = \mathsf{seed}[e][i]$
      for $p \in [1:D]$: the pth bit of $i-1$ is zero, **do**
        $\mathsf{input\_mshare}[e][p]$ += $\mathsf{input\_share}[e][i]$
    else
      $\mathsf{acc\_wit}, \mathsf{acc\_beav\_ab}, \mathsf{acc\_beav\_c} = \mathsf{acc}$
      $\mathsf{beav\_ab\_plain}[e] = \mathsf{acc\_beav\_ab} + \mathrm{SampleFieldElements}(\mathsf{salt}, \mathsf{seed}[e][i], 2dt\eta)$
      $\mathsf{beav\_c\_plain}[e] = \mathsf{beav\_c\_plain} \leftarrow \mathrm{InnerProducts}(\mathsf{beav\_ab\_plain})$ ▷ $a \cdot b = c$
      $\mathsf{aux}[e] = (\mathsf{wit\_plain} - \mathsf{acc\_wit}, \mathsf{beav\_c\_plain}[e] - \mathsf{acc\_beav\_c})$ ▷ $\mathsf{aux}[e] \in \mathbb{F}_q^{k+2w+t\eta}$
      $\mathsf{state}[e][i] = (\mathsf{seed}[e][i], \mathsf{aux}[e])$
  $\mathsf{com}[e][i] = \mathrm{Commit}(\mathsf{salt}, e, i, \mathsf{state}[e][i])$
$h_1 = \mathrm{Hash}_1(\mathsf{seed}_H, y, \mathsf{salt}, \mathsf{com}[1][1], \ldots, \mathsf{com}[\tau][2^D])$
$(\mathsf{chal}[e])_{e\in[1:\tau]} \leftarrow \mathrm{ExpandMPCChallenge}(h_1, \tau)$
for $e \in [1:\tau]$ **do**
  $\mathsf{input\_plain}[e] = (\mathsf{wit\_plain}, \mathsf{beav\_ab\_plain}[e], \mathsf{beav\_c\_plain}[e])$
  $\mathsf{broad\_plain}[e] \leftarrow \mathrm{ComputePlainBroadcast}(\mathsf{input\_plain}[e], \mathsf{chal}[e], (H', y))$
  for $p \in [1:D]$ **do**
    $\mathsf{broad\_share}[e][p] = \mathrm{PartyComputation}(\mathsf{input\_mshare}[e][p], \mathsf{chal}[e],$
    $(H', y), \mathsf{broad\_plain}[e], \mathtt{False})$
    ▷ $\mathsf{broad\_share}[e][p] \in \mathbb{F}_q^{(2d+1)t\eta}$
$h_2 = \mathrm{Hash}_2(m, \mathsf{salt}, h_1, \{\mathsf{broad\_plain}[e], \{\mathsf{broad\_share}[e][p]\}_{p\in[1:D]}\}_{e\in[1:\tau]})$.
$\{i^*[e]\}_{e\in[1:\tau]} \leftarrow \mathrm{ExpandViewChallenge}(h_2, 1)$.
for $e \in [1:\tau]$ **do**
  $\mathsf{path}[e] \leftarrow \mathrm{GetSeedSiblingPath}(\mathsf{rseed}[e], i^*[e])$.
  if $i^*[e] = 2^D$ **then**
    $\mathsf{view}[e] = \mathsf{path}[e]$
  else
    $\mathsf{view}[e] = (\mathsf{path}[e], \mathsf{aux}[e])$
$\sigma = \left(\mathsf{salt} \mid h_2 \mid (\mathsf{view}[e], \mathsf{broad\_plain}[e], \mathsf{com}[e][i^*[e]])_{e\in[1:\tau]}\right)$
return σ

Computer Architecture and Security Lab (CASLAB)
https://caslab.io

SDitH Sign - Offline

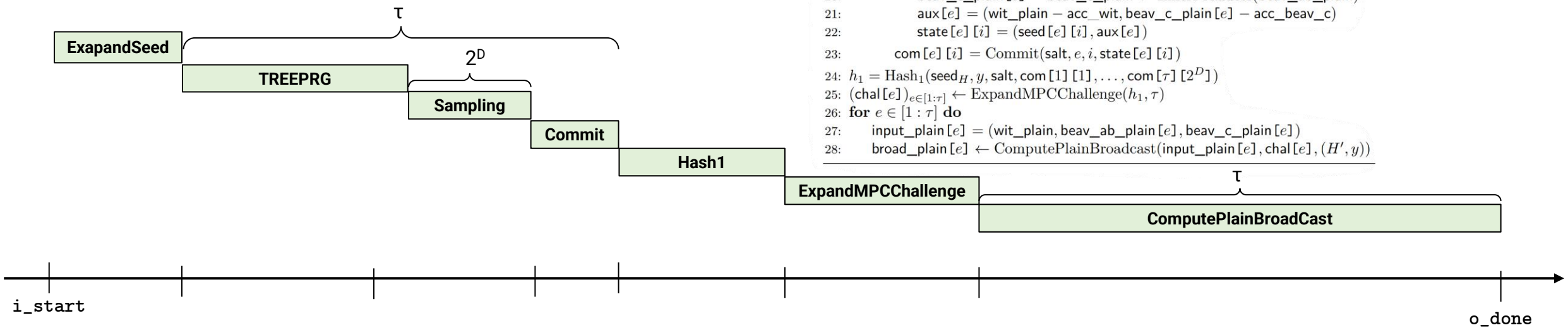
Algorithm 2a SDitH – Hypercube Variant – Signature Generation (Offline Part)

Input: a secret key $sk = (\mathsf{seed}_H, y, \mathsf{wit\_plain})$ and a message $m \in \{0,1\}^*$

```
salt ← {0,1}^{2λ}, mseed ← {0,1}^λ
H' ← ExpandH(seed_H)
{rseed[e]}_{e∈[1:τ]} ← ExpandSeed(salt, mseed, τ)
for e ∈ [1 : τ] do
    (seed[e][i])_{i∈[1:2^D]} ← TreePRG(salt, rseed[e])
    acc = 0
    input_mshare[e][p] = 0 for all (e,p) ∈ [1 : τ] × [1 : D]
    for i ∈ [1 : 2^D] do
        if i ≠ 2^D then
            input_share[e][i] ← SampleFieldElements(salt, seed[e][i], k + 2w + t(2d + 1)η)

            acc += input_share[e][i]
            state[e][i] = seed[e][i]
            for p ∈ [1 : D] : the p^th bit of i − 1 is zero, do
                input_mshare[e][p] += input_share[e][i]
        else
            acc_wit, acc_beav_ab, acc_beav_c = acc
            beav_ab_plain[e] = acc_beav_ab + SampleFieldElements(salt, seed[e][i], 2dtη)

            beav_c_plain[e] = beav_c_plain ← InnerProducts(beav_ab_plain)
            aux[e] = (wit_plain − acc_wit, beav_c_plain[e] − acc_beav_c)
            state[e][i] = (seed[e][i], aux[e])
        com[e][i] = Commit(salt, e, i, state[e][i])
h_1 = Hash_1(seed_H, y, salt, com[1][1], ..., com[τ][2^D])
(chal[e])_{e∈[1:τ]} ← ExpandMPCChallenge(h_1, τ)
for e ∈ [1 : τ] do
    input_plain[e] = (wit_plain, beav_ab_plain[e], beav_c_plain[e])
    broad_plain[e] ← ComputePlainBroadcast(input_plain[e], chal[e], (H', y))
```



Constant time

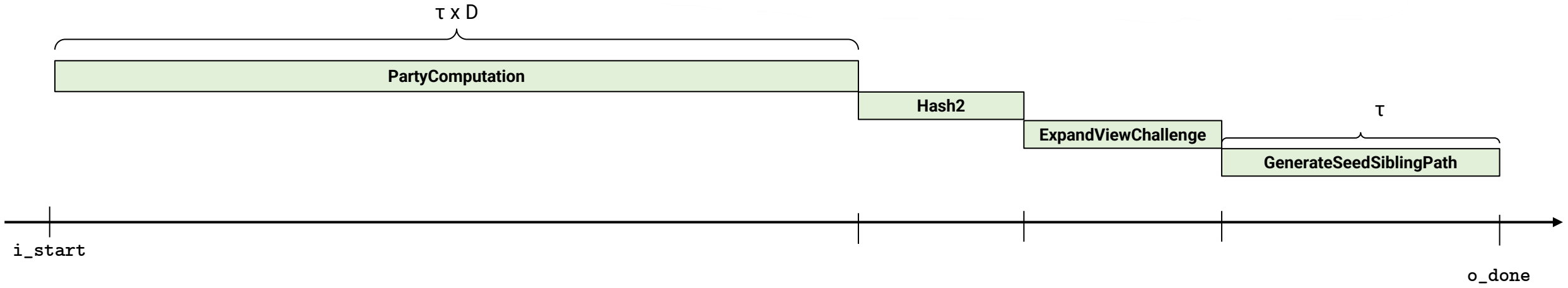
Computer Architecture and Security Lab (CASLAB)
https://caslab.io

SANDBOXAQ

SDitH Sign - Online

Algorithm 2b SDitH – Hypercube Variant – Signature Generation (Online Part)

```
for e ∈ [1 : τ] do
    for p ∈ [1 : D] do
        broad_share[e][p] = PartyComputation(input_mshare[e][p], chal[e],
                                             (H′, y), broad_plain[e], False)
h2 = Hash2(m, salt, h1, {broad_plain[e], {broad_share[e][p]}_{p∈[1:D]}}_{e∈[1:τ]})
{i*[e]}_{e∈[1:τ]} ← ExpandViewChallenge(h2, 1)
for e ∈ [1 : τ] do
    path[e] ← GetSeedSiblingPath(rseed[e], i*[e])
    if i*[e] = 2^D then
        view[e] = path[e]
    else
        view[e] = (path[e], aux[e])
return σ = (salt | h2 | (view[e], broad_plain[e], com[e][i*[e]])_{e∈[1:τ]})
```



Constant time

SDitH Verify

- Similar to sign_offline and sign_online not so much scope for the parallelism at the algorithmic level
- Possibility of parallelism at the function/module level at cost of additional hardware
- Unrolling the for loops at the cost of duplicating modules

Algorithm 7 SDitH – Hypercube Variant – Verification Algorithm

Input: a public key $pk = (\mathsf{seed}_H, y)$, a signature σ and a message $m \in \{0,1\}^*$

Parse σ as $\left(\mathsf{salt} \mid h_2 \mid (\mathsf{view}[e], \mathsf{broad\_plain}[e], \mathsf{com}[e][i^*[e]])_{e\in[1:\tau]}\right)$
$H' \leftarrow \mathrm{ExpandH}(\mathsf{seed}_H)$
$\{i^*[e]\}_{e\in[1:\tau]} \leftarrow \mathrm{ExpandViewChallenge}(h_2, 1)$
for $e \in [1:\tau]$ **do**
  $(\mathsf{seed}[e][i])_{i\in[1:2^D\setminus i^*[e]]} \leftarrow \mathrm{GetLeavesFromSiblingPath}(i^*[e], \mathsf{salt}, \mathsf{path}[e])$
  for $i \in \{2^D \setminus i^*[e]\}$ **do**
    if $i \neq 2^D$ **then**
      $\mathsf{state}[e][i] = \mathsf{seed}[e][i]$
    else
      $\mathsf{state}[e][i] = (\mathsf{seed}[e][i], \mathsf{aux}[e])$
    $\mathsf{com}[e][i] = \mathrm{Commit}(\mathsf{salt}, e, i, \mathsf{state}[e][i])$
$h_1 = \mathrm{Hash}_1(\mathsf{seed}_H, y, \mathsf{salt}, \mathsf{com}[1][1], \ldots, \mathsf{com}[\tau][2^D])$
$\mathsf{chal} \leftarrow \mathrm{ExpandMPCChallenge}(h_1, \tau)$
for $e \in [1:\tau]$ **do**
  $\mathsf{input\_mshare}^*[e][p] = 0$ for all $(e,p) \in [1:\tau] \times [1:D]$
  for $i \in [1:2^D \setminus i^*[e]]$ **do**
    if $i \neq 2^D$ **then**
      $\mathsf{input\_share}[e][i] \leftarrow \mathrm{SampleFieldElements}(\mathsf{salt}, \mathsf{seed}[e][i], k + 2w + t(2d+1)\eta)$

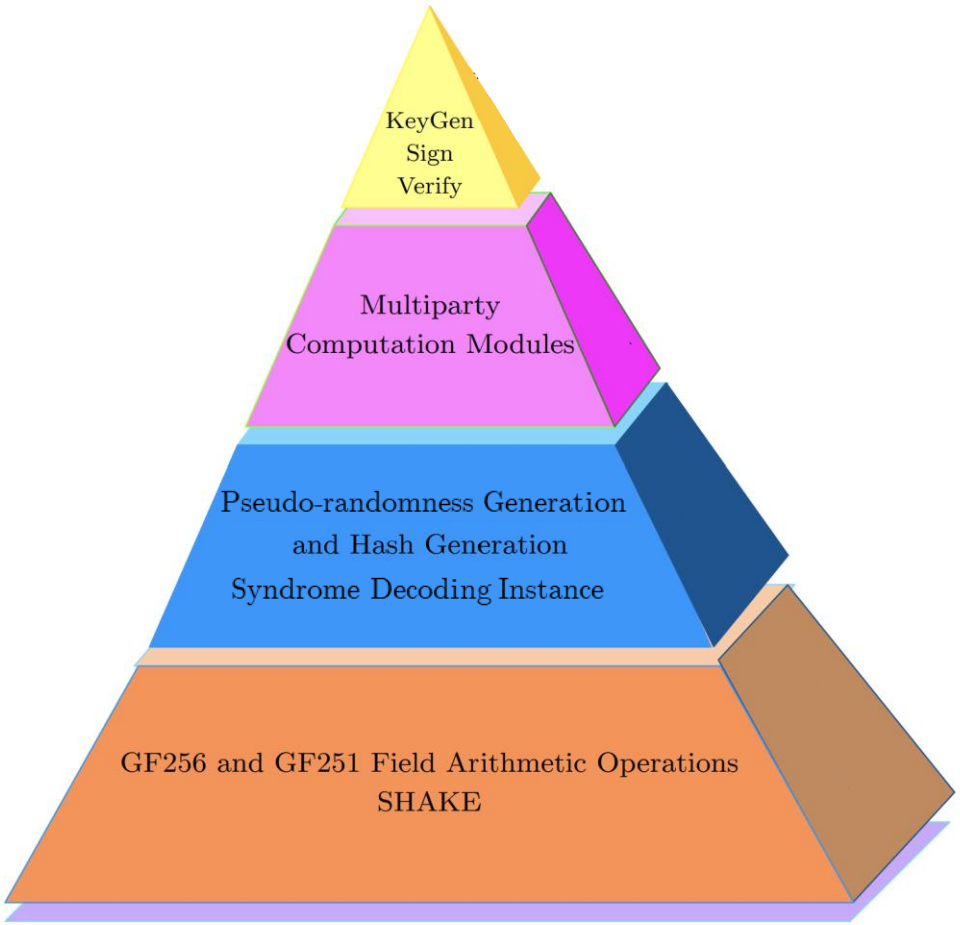
    else
      $\mathsf{beav\_ab\_plain}[e][2^D] = \mathrm{SampleFieldElements}(\mathsf{salt}, \mathsf{seed}[e][2^D], 2dt\eta)$

      $\mathsf{input\_share}[e][2^D] = (\mathsf{aux}[e] \mid \mathsf{beav\_ab\_plain}[e][2^D])$
    for $p \in [1:D]$: the pth bit of $i-1$ and $i^*[e]$ are different **do**
      $\mathsf{input\_mshare}'[e][p]$ += $\mathsf{input\_share}[e][i]$
  for $p \in [1:D]$ **do**
    if the pth bit of $i^*[e]$ is 1 **then**
      $\mathsf{broad\_share}[e][p] = \mathrm{PartyComputation}(\mathsf{input\_mshare}'[e][p], \mathsf{chal}, (H', y), \mathsf{broad\_plain}, \texttt{False})$
    else
      $\mathsf{broad\_share}[e][p] = \mathsf{broad\_plain}[e] - \mathrm{PartyComputation}(\mathsf{input\_mshare}'[e][p], \mathsf{chal}, (H', y), \mathsf{broad\_plain}, \texttt{True})$

$h'_2 = \mathrm{Hash}_2(m, \mathsf{salt}, h_1, \{\mathsf{broad\_plain}[e], \{\mathsf{broad\_share}[e][p]\}_{p\in[1:D]}\}_{e\in[1:\tau]})$.
return $h_2 \stackrel{?}{=} h'_2$

Hardware Design and Challenges

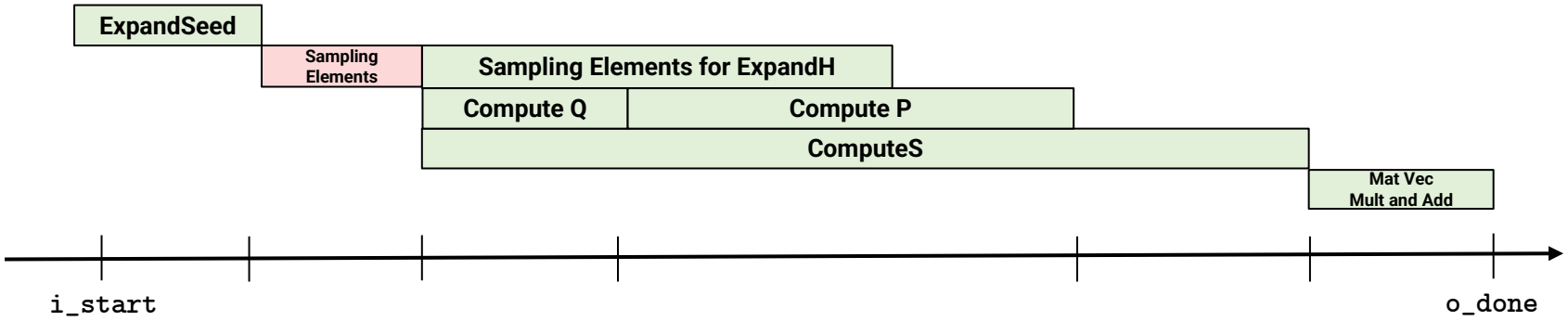
Hardware Design Architecture



Our Contributions

- First parameterizable hardware realization of Hypercube Variant of SDitH Signature Scheme
- Two Variants of Syndrome Decoding Modules
  - Sample first, then multiply
  - Sample and multiply on the fly
- Split Hardware Implementation of Sign into *Offline* and *Online* phases
- Drastic Reduction in terms of Clock Cycles when compared to
  - Key Generation – Up to **250x**
  - Signature Generation – Up to **3.4x**
  - Signature Verification – Up to **2.2x**

Syndrome Computation Module – Key Generation



- $Y = s_B + H's_A$
- Syndrome Computation needs to be done after S (Sa, Sb) is computed by ComputeS module
- Hence, Sample First then Multiply approach (STFM)

Syndrome Computation Module – Sign and Verify

Algorithm 2a SDitH – Hypercube Variant – Signature Generation (Offline Part)

Input: a secret key $sk = (\mathsf{seed}_H, y, \mathsf{wit\_plain})$ and a message $m \in \{0,1\}^*$

1: $\mathsf{salt} \leftarrow \{0,1\}^{2\lambda}$, $\mathsf{mseed} \leftarrow \{0,1\}^{\lambda}$

2: $H' \leftarrow \mathrm{ExpandH}(\mathsf{seed}_H)$

Algorithm 4 ComputePlainBroadcast

Input: $\mathsf{input\_plain} := (\mathsf{wit\_plain}, \mathsf{beav\_ab\_plain}, \mathsf{beav\_c\_plain})$, chal, (H', y)

Output: $\mathsf{broad\_plain}$

1: $(s_A, \boldsymbol{Q}', \boldsymbol{P}) \leftarrow \mathrm{Parse}(\mathsf{wit\_plain}, \mathbb{F}_q^k, (\mathbb{F}_q^{w/d})^d, (\mathbb{F}_q^{w/d})^d)$

2: $(\boldsymbol{a}, \boldsymbol{b}) \leftarrow \mathrm{Parse}(\mathsf{beav\_ab\_plain}, (\mathbb{F}_{q^n}^d)^t)$

3: $c \leftarrow \mathrm{Parse}(\mathsf{beav\_c\_plain}, \mathbb{F}_{q^n}^t)$

4: $(r, \boldsymbol{\varepsilon}) \leftarrow \mathrm{Parse}(\mathsf{chal}, \mathbb{F}_{q^n}^t, (\mathbb{F}_{q^n}^d)^t)$

5: $s = (s_A \mid y + H' s_A)$

6: $\boldsymbol{Q} = \mathrm{CompleteQ}(\boldsymbol{Q}', 1)$

7: $\boldsymbol{S} \leftarrow \mathrm{Parse}(s, (\mathbb{F}_q^{m/d})^d)$

8: **for** $j \in [1:t]$ **do**

9: **for** $\nu \in [1:d]$ **do**

10: $\boldsymbol{\alpha}[j][\nu] = \boldsymbol{\varepsilon}[j][\nu] \otimes \mathrm{Evaluate}(\boldsymbol{Q}[\nu], r[j]) + \boldsymbol{a}[j][\nu]$

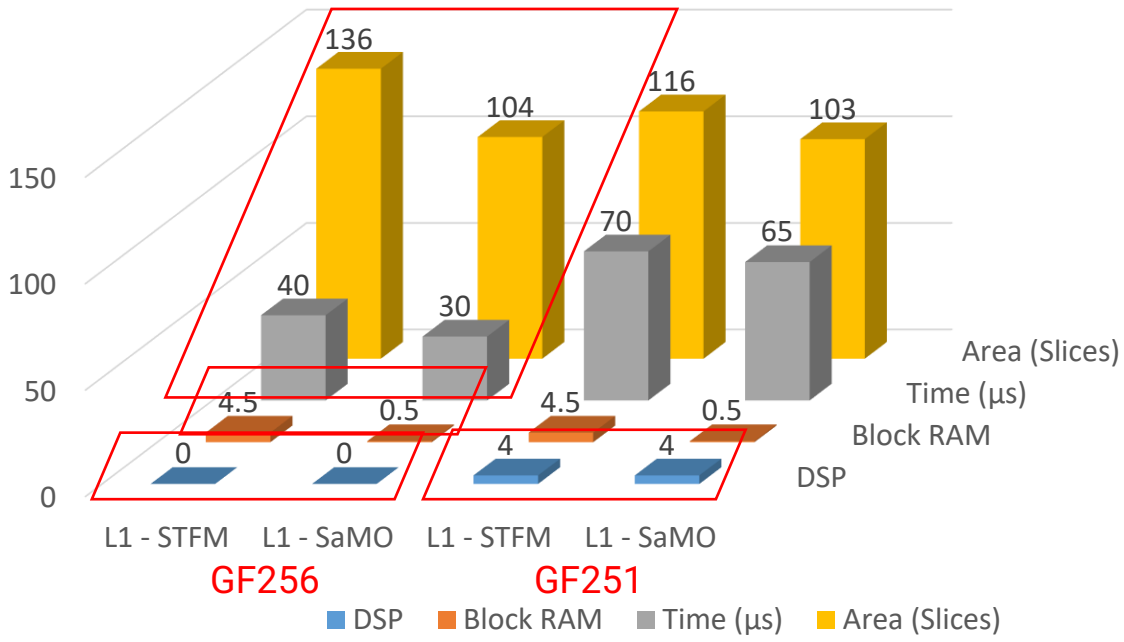
11: $\boldsymbol{\beta}[j][\nu] = \mathrm{Evaluate}(\boldsymbol{S}[\nu], r[j]) + \boldsymbol{b}[j][\nu]$

12: $\mathsf{broad\_plain} = \mathrm{Serialize}(\boldsymbol{\alpha}, \boldsymbol{\beta})$

13: **return** $\mathsf{broad\_plain}$

- y and S_a are inputs.
- Only need *H'* to compute the syndrome.
- "Sample and Multiply On the Fly" (SaMO) approach

Comparison of Syndrome Computation – STFM vs SaMO

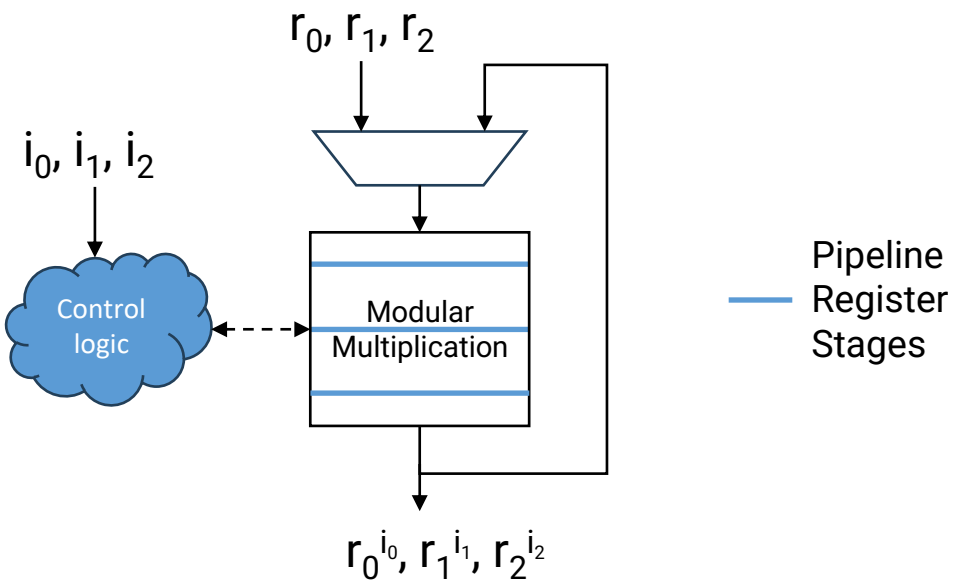


Evaluate Module

- Evaluate - takes as input an F_q-vector Q representing the coefficients of a polynomial $F_q[X]$ and a point $r \in F_{points}$ and computes evaluation as follows:

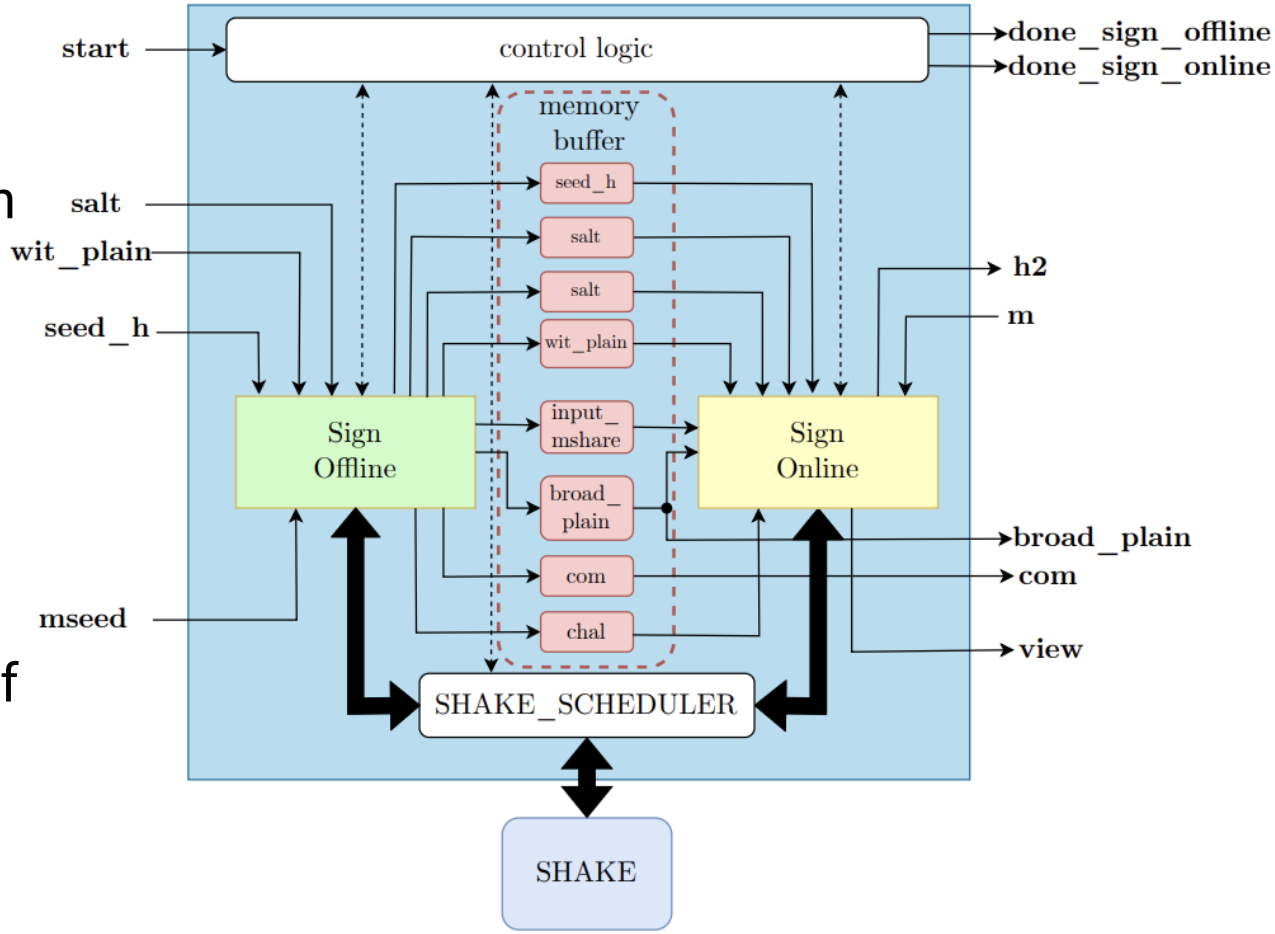
$$\text{Evaluate}: \begin{cases} \bigcup_{|Q|} (\mathbb{F}_q)^{|Q|} \times \mathbb{F}_{q^\eta} & \to \mathbb{F}_{q^\eta} \\ (Q, r) & \mapsto \sum_{i=1}^{|Q|} Q[i] \cdot r^{i-1} \end{cases} \quad \text{where } r^{i-1} = \underbrace{r \otimes r \otimes \cdots \otimes r}_{i-1 \text{ times}} .$$

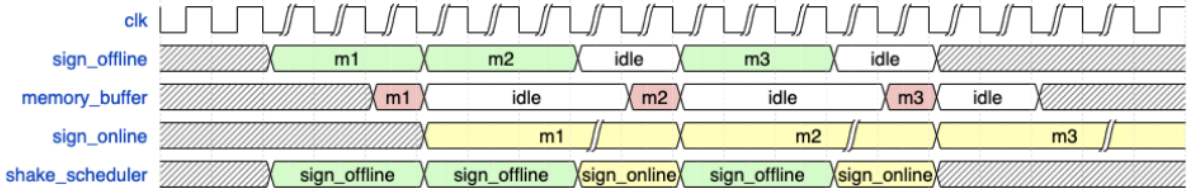
- Evaluate is used in both sign and verify operations. It contributes to:
  - 99% of cycles of the online part of the signing
  - 70%-90% of clock cycles in the verification based on the parameter set
- r^{i-1} is a 32-bit modular exponentiation; it is an expensive operation
  - Software implementation (target device Intel Xeon E-2378 CPU) accomplishes this by two large look-up-tables (370 KB to 1.5 MB for full design - based on parameter set)
  - Our lightweight target, Artix 7 FPGA, does not have these resources. Hence, we take an on-the-fly computation approach



Signature Generation Module

- The block shown is for our area optimized implementation of SDitH signature generation scheme
- Signature generation is divided into two phases offline and online – they can run in parallel
- SHAKE256 is a hash function that is used in both the offline and online phases
- However, SHAKE256 is area expensive 31% of overall hardware design
- Hence, we design an optimized SHAKE scheduler such that
  - the same SHAKE module is switched between Offline and Online phases without wasting cycles and additional area



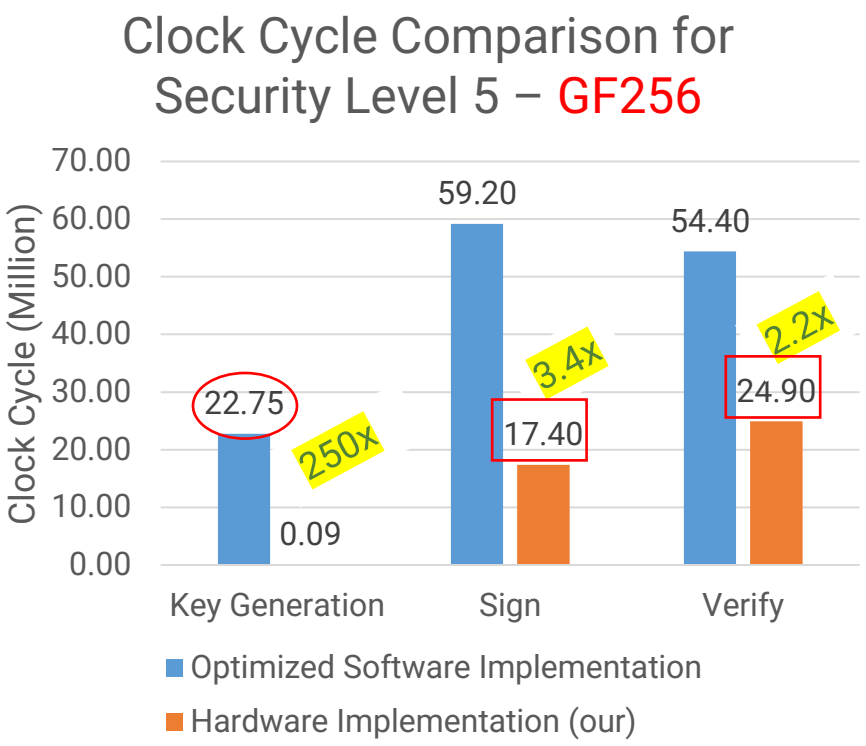


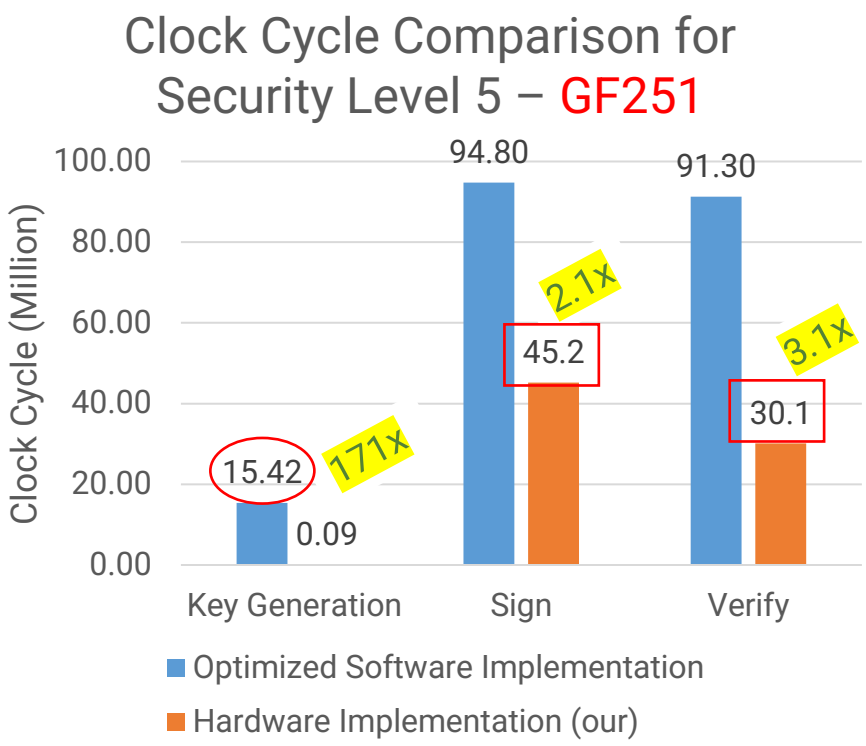
Computer Architecture and Security Lab (CASLAB)
https://caslab.io

SANDBOXAQ

Comparison with Related and Relevant Work

Clock Cycles Comparison – Optimized Software v/s Our Hardware Implementation – Hypercube Variant





Galois Field New Instructions from Intel are used

~70-99% of the clock cycles are taken in the 'sign_online' and 'verify' modules by the 'Evaluate' module

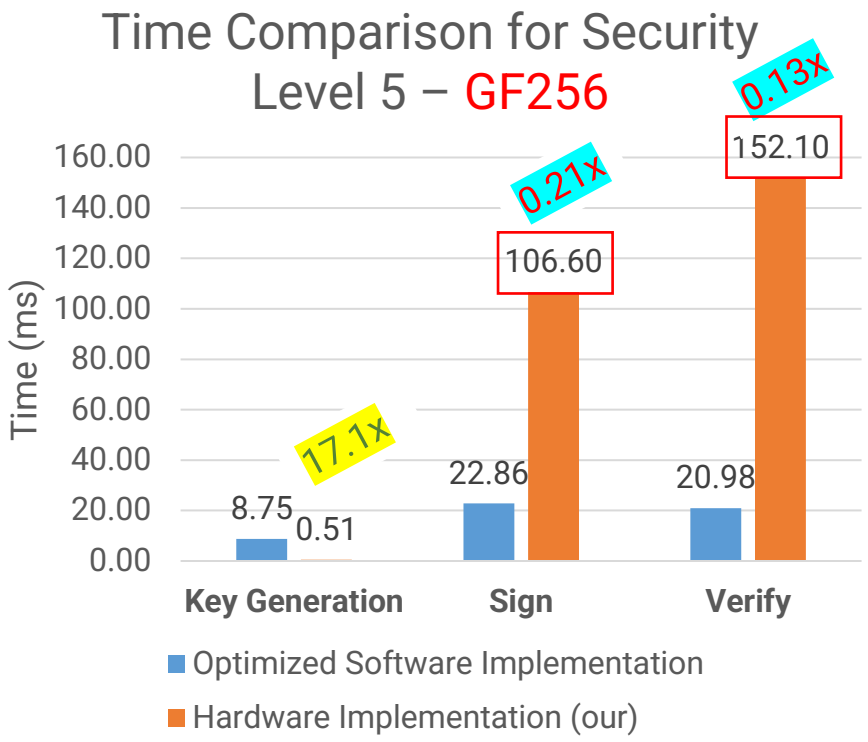
Improvement

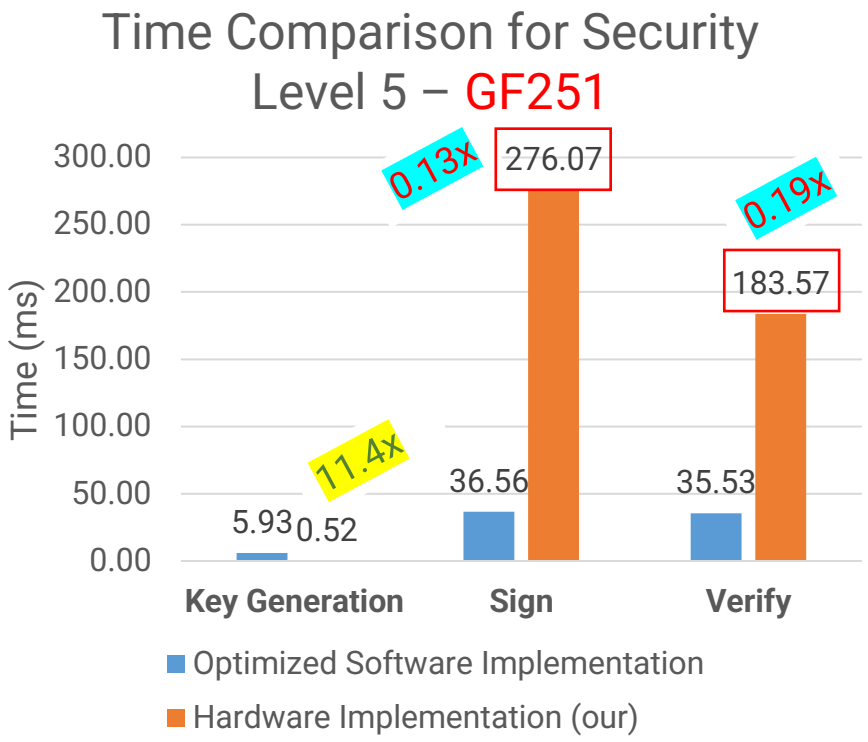
Computer Architecture and Security Lab (CASLAB)
https://caslab.io

SANDBOXAQ

Time Comparison – Optimized Software v/s Our Hardware Implementation – Hypercube Variant

Operating Frequency:
Intel Xeon Processor = 2.6 GHz
Xilinx Artix 7 FPGA = 164 MHz



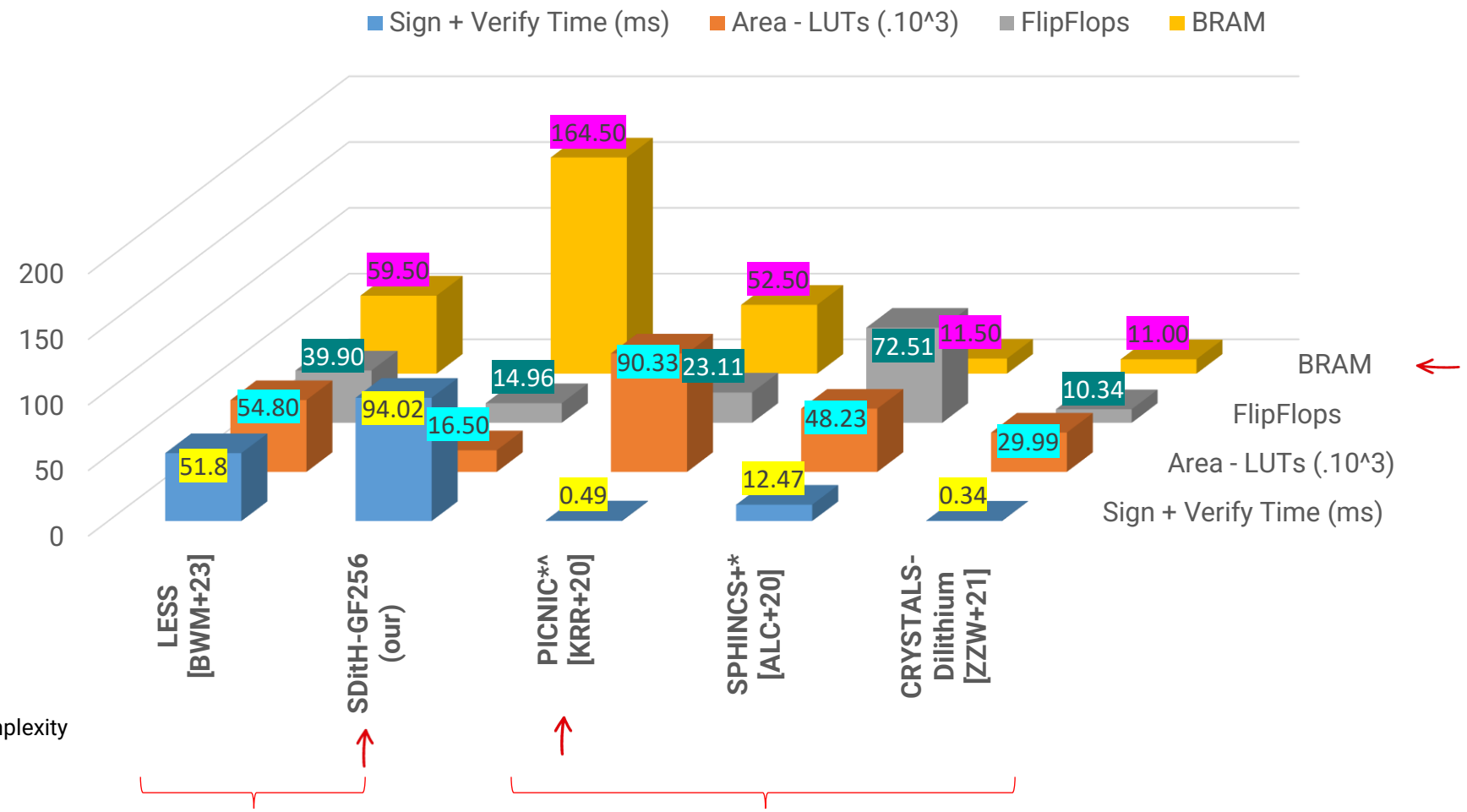


Improvement

Decline

~70-99% of the clock cycles are taken in the 'sign_online' and 'verify' modules by the 'Evaluate' module

Comparison with other PQC-DSA candidates – Security Level 1



*No KeyGen
^Low Multiplication Complexity

Latest NIST Competition Candidates

Old NIST Competition Candidates

Conclusion and Future Work

Conclusion

- This work presents first hardware realization of SDitH Signature Scheme.
  - Parameterizable across three Security Levels and Two Arithmetic Fields.
- SDitH could be realized as a light-weight implementation. However, the memory consumption is higher.

Future Work

- The lower-level modules implemented as part of this work could be used to construct the ‘threshold’ variant of SDitH easily.
- Module level parallelism could be exploited to build a high-performance design which could speed-up the sign and verify operations.
- The MPC hardware modules’ components could be reused outside SDitH.

References

[Gaj20] Kris Gaj, Implementation and Benchmarking of Round 2 Candidates in the NIST Post-Quantum Cryptography Standardization Process Using FPGAs, NIST Seminars, Oct 2020.

[BWM+23] Luke Beckwith, Robert Wallace, Kamyar Mohajerani, and Kris Gaj. A high-performance hardware implementation of the less digital signature scheme. In Thomas Johansson and Daniel Smith-Tone, editors, Post-Quantum Cryptography, pages 57–90, Cham, 2023. Springer Nature Switzerland.

[KRR+20] Daniel Kales, Sebastian Ramacher, Christian Rechberger, Roman Walch, and Mario Werner. Efficient FPGA implementations of lowmc and picnic. In Stanislaw Jarecki, editor, Topics in Cryptology – CT-RSA 2020, pages 417–441, Cham, 2020. Springer International Publishing.

[ZZW+23] Cankun Zhao, Neng Zhang, Hanning Wang, Bohan Yang, Wenping Zhu, Zhengdong Li, Min Zhu, Shouyi Yin, Shaojun Wei, and Leibo Liu. A compact and high-performance hardware architecture for CRYSTALS-Dilithium. IACR Transactions on Cryptographic Hardware and Embedded Systems,2022(1):270–295, Nov. 2021.

[ALC+20] Dorian Amiet, Lukas Leuenberger, Andreas Curiger, and Paul Zbinden. Fpga-based sphincs+ implementations: Mind the glitch. In 2020 23rd Euromicro Conference on Digital System Design (DSD), pages 229–237, 2020.

Sanjay Deshpande, James Howe, Jakub Szefer, and Dongze Yue, "SDitH in Hardware", in Transactions on Cryptographic Hardware and Embedded Systems (TCHES), September 2024.

https://ia.cr/2024/069

Thank you!

Sanjay Deshpande
email: sanjay.deshpande@yale.edu

Computer Architecture and Security Lab (CASLAB)
https://caslab.io

SANDBOXAQ