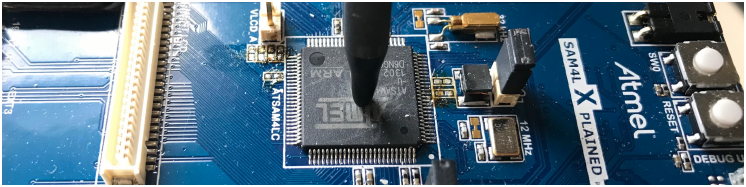


Bitslicing Arithmetic/Boolean Masking Conversions for Fun and Profit

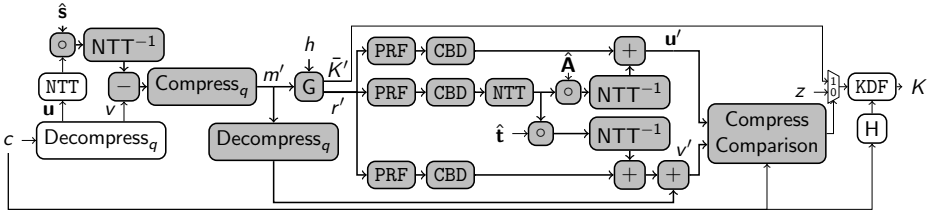
with Application to Lattice-Based KEMs

Olivier Bronchain Gaëtan Cassiers

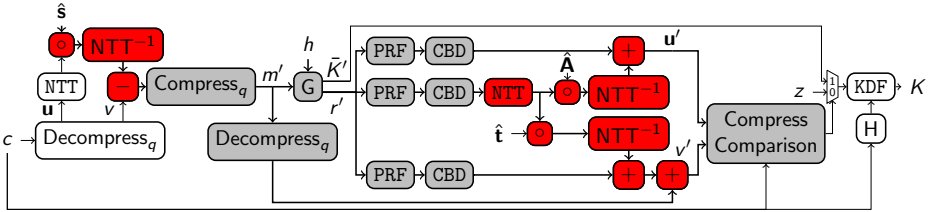
CHES 2022, Leuven, Belgium

Contents

Masked Kyber computation (Inspired by Bos et al., TCHES 2021.)



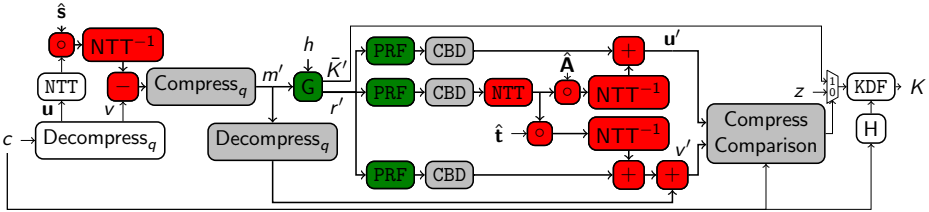
Masked Kyber computation (Inspired by Bos et al., TCHES 2021.)



Poly. arithmetic (■):

- Arith. masking.
- Linear overheads.

Masked Kyber computation (Inspired by Bos et al., TCHES 2021.)



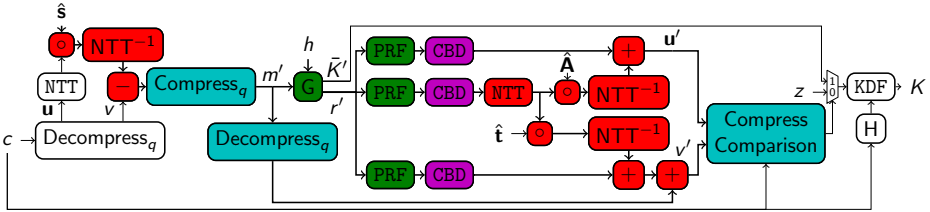
Poly. arithmetic (■):

- Arith. masking.
- Linear overheads.

Hash functions (■):

- Boolean masking.
- Quadratic overheads.

Masked Kyber computation (Inspired by Bos et al., TCHES 2021.)



Poly. arithmetic (■):

- Arith. masking.
- Linear overheads.

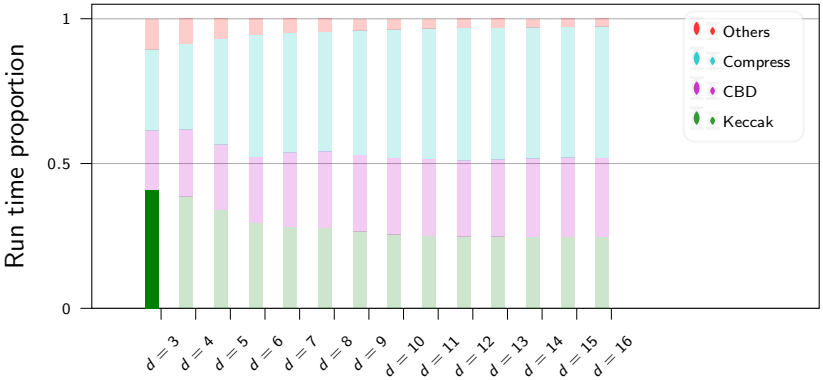
Hash functions (■):

- Boolean masking.
- Quadratic overheads.

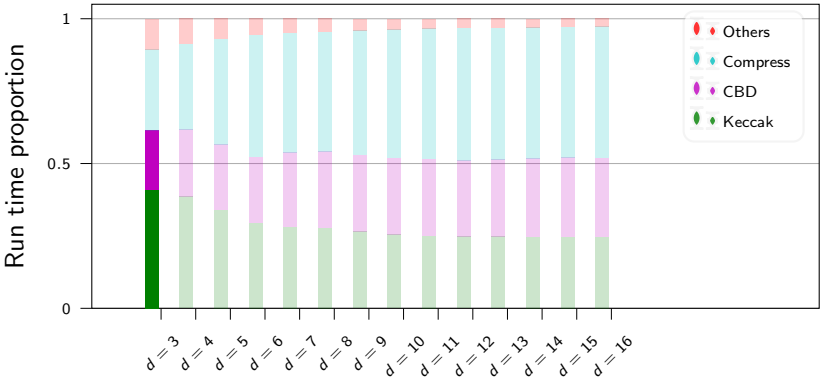
Poly. sampl. (■) & compress. (■):

- Boolean & arith. masking.
- Quadratic overheads.

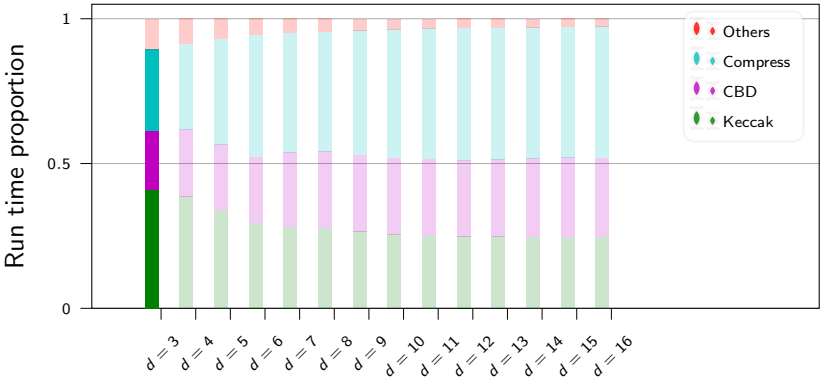
Masked Kyber performance



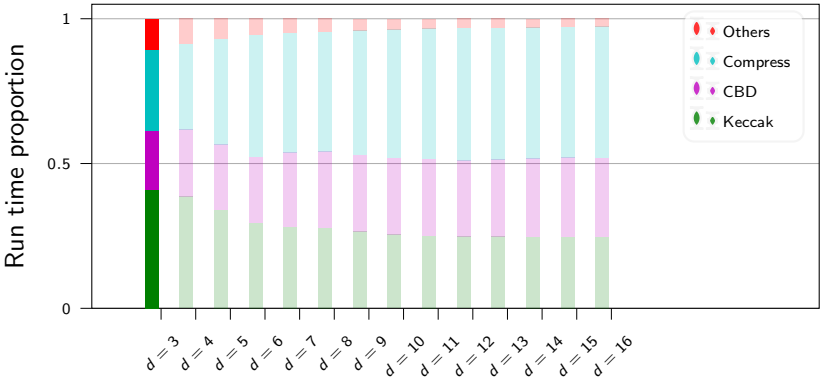
Masked Kyber performance



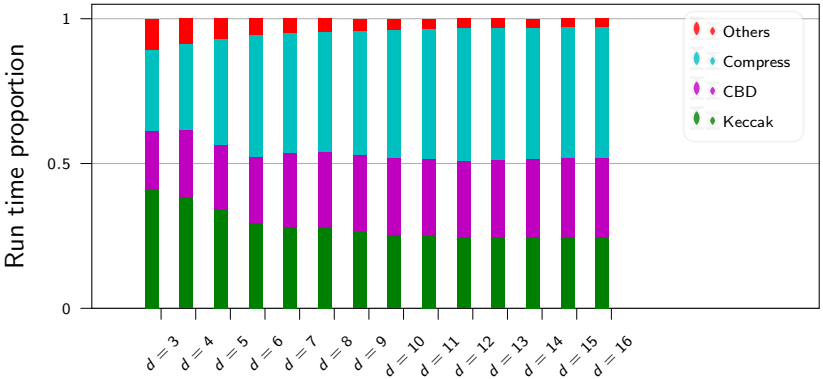
Masked Kyber performance



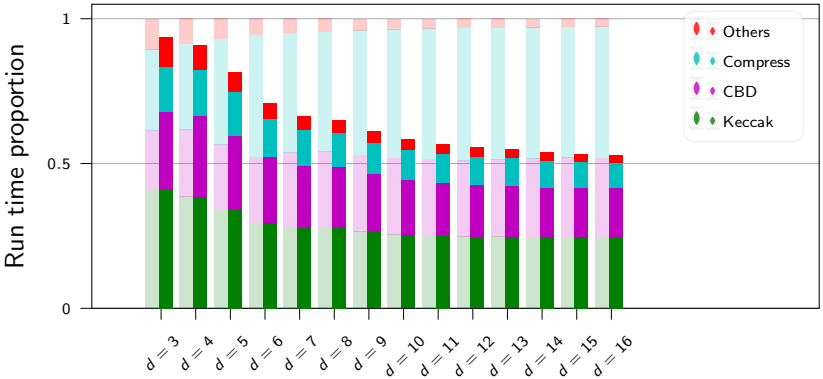
Masked Kyber performance



Masked Kyber performance



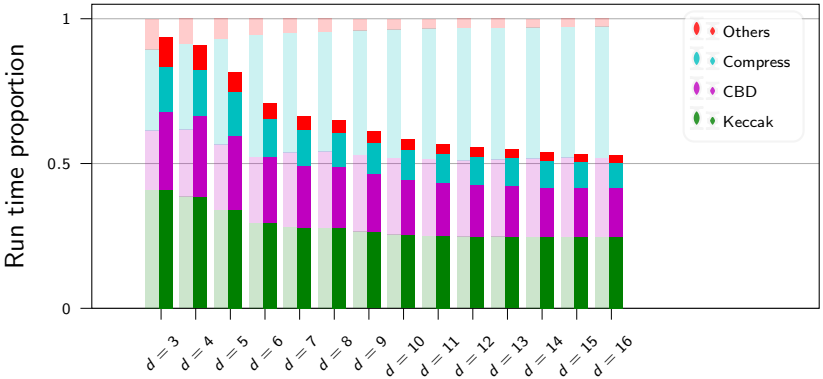
Masked Kyber performance



How have we improved the performances of masked Kyber ?

- Optimize CBD and Compressions.

Masked Kyber performance



How have we improved the performances of masked Kyber ?

- Optimize CBD and Compressions.
- Optimize masking conversion algorithms.

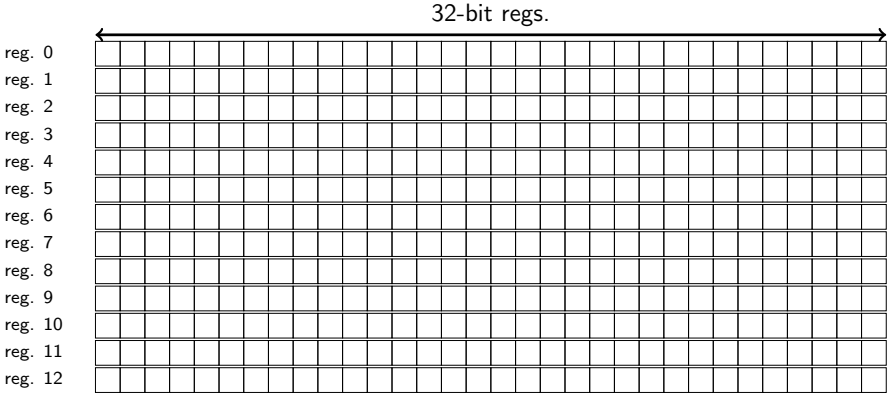
Contents

Canonical vs bitslice representations

State $x \in \mathbb{Z}_{3329}[X]/(X^{256}+1)$ represented with 256 13-bit integers.

Canonical vs bitslice representations

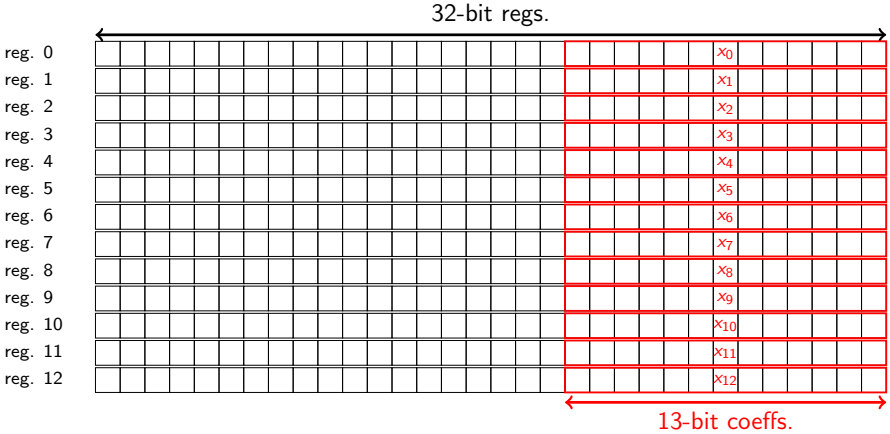
State $x \in \mathbb{Z}_{3329}[X]/(X^{256}+1)$ represented with 256 13-bit integers.



Canonical vs bitslice representations

Canonical

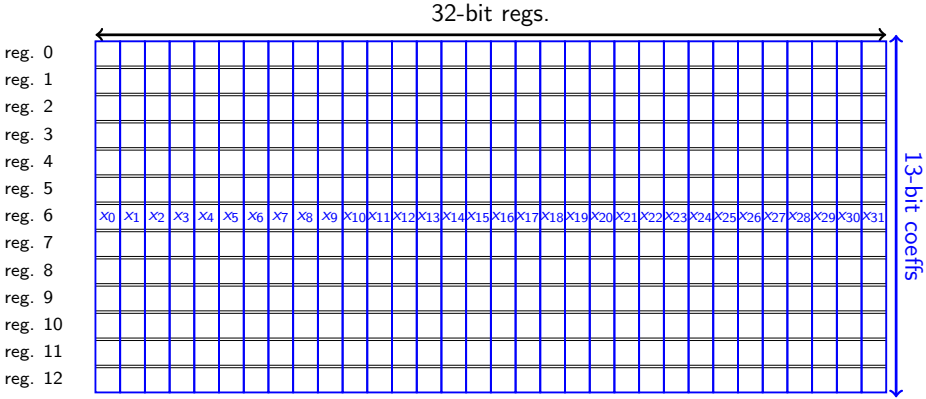
State $x \in \mathbb{Z}_{3329}[X]/(X^{256}+1)$ represented with 256 13-bit integers.



Canonical vs bitslice representations

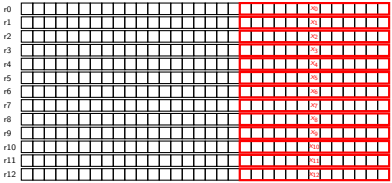
Bitslice

State $x \in \mathbb{Z}_{3329}[X]/(X^{256}+1)$ represented with 256 13-bit integers.



Canonical vs bitslice representations: performance

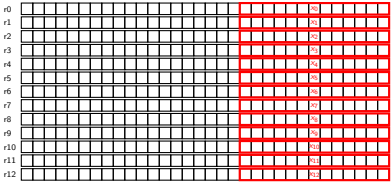
Canonical



+ Arithmetic operations ($+$, $\times$).
– Single-bit processing.
– Memory/registers usage.

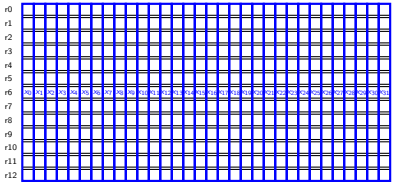
Canonical vs bitslice representations: performance

Canonical



+ Arithmetic operations (+, ×).
− Single-bit processing.
− Memory/registers usage.

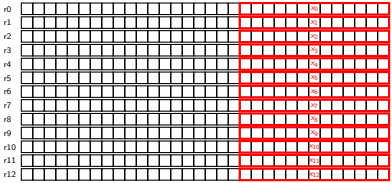
Bitslice



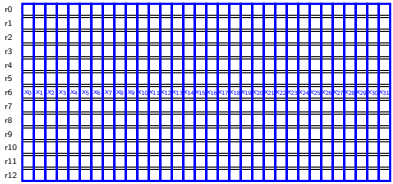
+ Bitwise operations: throughput.
+ Security: registers fully used.
− Representation change cost.

Canonical vs bitslice representations: performance

Canonical



Bitslice



+ Arithmetic operations ($+$, $\times$).
− Single-bit processing.
− Memory/registers usage.

+ Bitwise operations: throughput.
+ Security: registers fully used.
− Representation change cost.

Representation change: $\mathcal{O}(w \log w)$ instructions for $w \lfloor w / \log p \rfloor$ coefficients.

Contents

$\texttt{SecAdd}^d_{k+1}$: arithmetic addition of Boolean shares

What is SoTA ?

- Masked Kogge-Stone adders.
- $\log k$ times $\texttt{SecAnd}^d_k$.

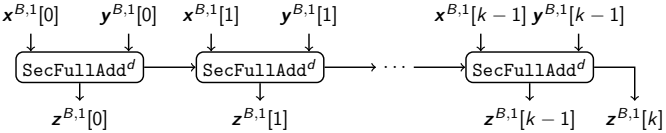
$\texttt{SecAdd}^d_{k+1}$: arithmetic addition of Boolean shares

What is SoTA ?

- Masked Kogge-Stone adders.
- $\log k$ times $\texttt{SecAnd}^d_k$.

Taking advantage of bitslice:

- Masked ripple-carry adder:
  - Carry bit: $ab + ac + bc = (a + b)(a + c) + a$
  - Output bit: $a + b + c$
  - One single $\texttt{SecAnd}^d_1$ per $\texttt{SecFullAdd}^d$.
- k times $\texttt{SecAnd}^d_1$.



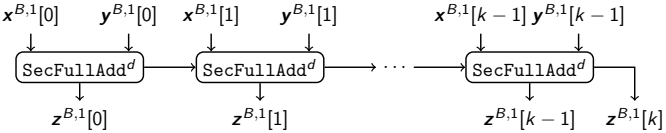
$\texttt{SecAdd}^d_{k+1}$: arithmetic addition of Boolean shares

What is SoTA ?

- Masked Kogge-Stone adders.
- $\log k$ times $\texttt{SecAnd}^d_k$.

Taking advantage of bitslice:

- Masked ripple-carry adder:
  - Carry bit: $ab + ac + bc = (a + b)(a + c) + a$
  - Output bit: $a + b + c$
  - One single $\texttt{SecAnd}^d_1$ per $\texttt{SecFullAdd}^d$.
- k times $\texttt{SecAnd}^d_1$.



$\rightarrow$ Security: trivial composition with PINI.

$\rightarrow$ **Reduced randomness and operations.**

$\texttt{SecAddModp}_p^d$: addition with arbitrary modulus.

SoTA:

New:

$\texttt{SecAddModp}_p^d$: addition with arbitrary modulus.

SoTA:

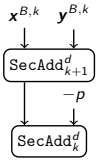
1. Add x and y.

New:

$\texttt{SecAddModp}_p^d$: addition with arbitrary modulus.

SoTA:

1. Add x and y.
2. Subtract p.

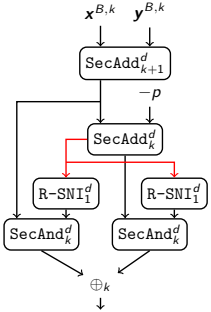


New:

$\texttt{SecAddModp}_p^d$: addition with arbitrary modulus.

SoTA:

1. Add x and y.
2. Subtract p.
3. Select output with underflow bit.

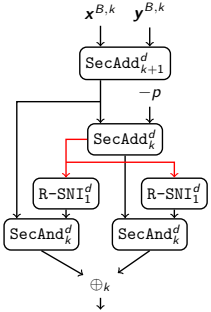


New:

$\mathtt{SecAddModp}_p^d$: addition with arbitrary modulus.

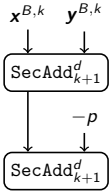
SoTA:

1. Add x and y.
2. Subtract p.
3. Select output with underflow bit.



New:

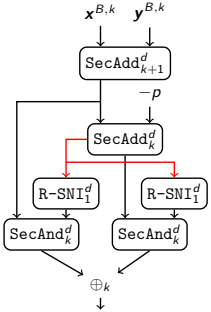
1. Add x and y.
2. Subtract p.



$\texttt{SecAddModp}_p^d$: addition with arbitrary modulus.

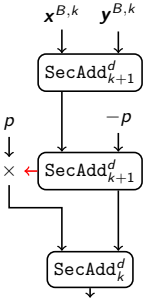
SoTA:

1. Add x and y.
2. Subtract p.
3. Select output with underflow bit.



New:

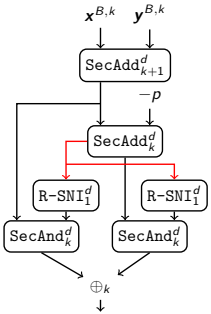
1. Add x and y.
2. Subtract p.
3. Add p if underflow occurred.



$\texttt{SecAddModp}_p^d$: addition with arbitrary modulus.

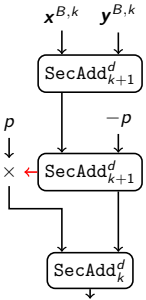
SoTA:

1. Add x and y.
2. Subtract p.
3. Select output with underflow bit.



New:

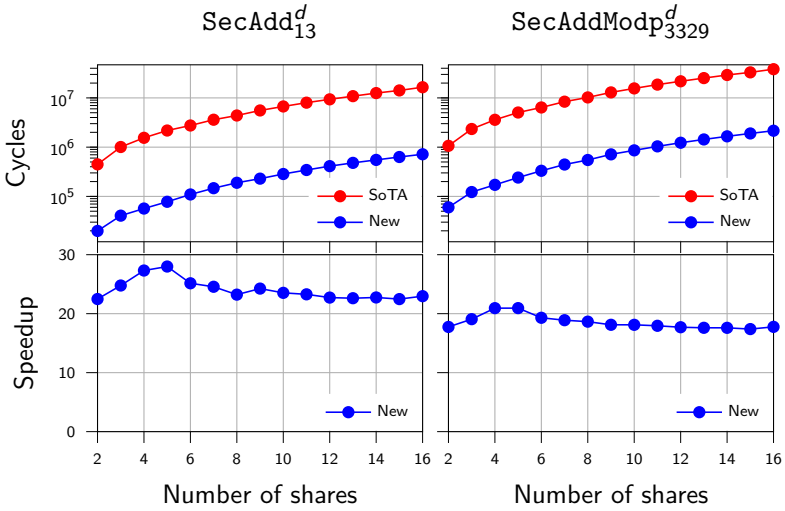
1. Add x and y.
2. Subtract p.
3. Add p if underflow occurred.



Taking advantage of bitslice:

- $\texttt{SecAdd}_k^d \approx \texttt{SecAnd}_k^d$.
- Replace 2 $\texttt{SecAnd}_k^d$ with 1 $\texttt{SecAdd}_k^d$.
- Secure composition: PINI.

SecAdd and SecAddModp performances



$\texttt{SecA2B}_k^d$: arithmetic to Boolean masking conversions

How to convert $\boldsymbol{c}^{A_{2^k}}$ to $\boldsymbol{c}^{B,k}$ with:

$$c = \sum_i \boldsymbol{c}_i^{A_{2^k}} \mod 2^k = \bigoplus_i \boldsymbol{c}_i^{B,k}$$

How to perform the conversion ?

- $(\boldsymbol{c}_i^{A_{2^k}}, 0)$ is a Boolean sharing of $\boldsymbol{c}_i^{A_{2^k}}$.
- Add all the $\boldsymbol{c}_i^{A_{2^k}}$'s with $\texttt{SecAdd}_k$'s.

$\texttt{SecA2B}_k^d$: arithmetic to Boolean masking conversions

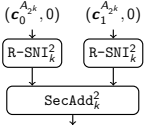
How to convert $\boldsymbol{c}^{A_{2^k}}$ to $\boldsymbol{c}^{B,k}$ with:

$$c = \sum_i \boldsymbol{c}_i^{A_{2^k}} \mod 2^k = \bigoplus_i \boldsymbol{c}_i^{B,k}$$

How to perform the conversion ?

- $(\boldsymbol{c}_i^{A_{2^k}}, 0)$ is a Boolean sharing of $\boldsymbol{c}_i^{A_{2^k}}$.
- Add all the $\boldsymbol{c}_i^{A_{2^k}}$'s with $\texttt{SecAdd}_k$'s.

SoTA construction:



$\texttt{SecA2B}_k^d$: arithmetic to Boolean masking conversions

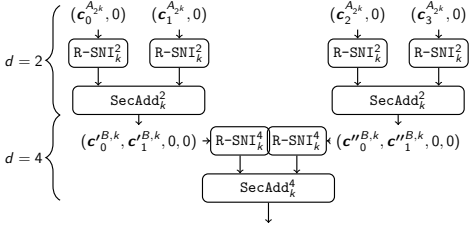
How to convert $\boldsymbol{c}^{A_{2^k}}$ to $\boldsymbol{c}^{B,k}$ with:

$$c = \sum_i \boldsymbol{c}_i^{A_{2^k}} \mod 2^k = \bigoplus_i \boldsymbol{c}_i^{B,k}$$

How to perform the conversion ?

- $(\boldsymbol{c}_i^{A_{2^k}}, 0)$ is a Boolean sharing of $\boldsymbol{c}_i^{A_{2^k}}$.
- Add all the $\boldsymbol{c}_i^{A_{2^k}}$'s with $\texttt{SecAdd}_k$'s.

SoTA construction:



$\texttt{SecA2B}_k^d$: arithmetic to Boolean masking conversions

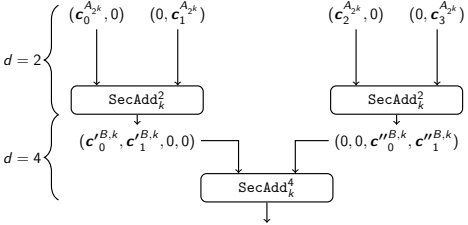
How to convert $\boldsymbol{c}^{A_{2^k}}$ to $\boldsymbol{c}^{B,k}$ with:

$$c = \sum_i \boldsymbol{c}_i^{A_{2^k}} \mod 2^k = \bigoplus_i \boldsymbol{c}_i^{B,k}$$

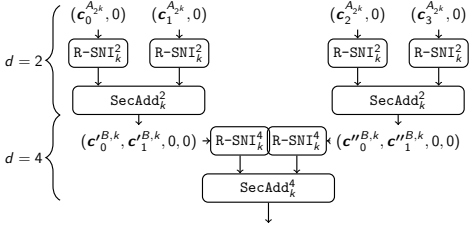
How to perform the conversion ?

- $(\boldsymbol{c}_i^{A_{2^k}}, 0)$ is a Boolean sharing of $\boldsymbol{c}_i^{A_{2^k}}$.
- Add all the $\boldsymbol{c}_i^{A_{2^k}}$'s with $\texttt{SecAdd}_k$'s.

New PINI construction:



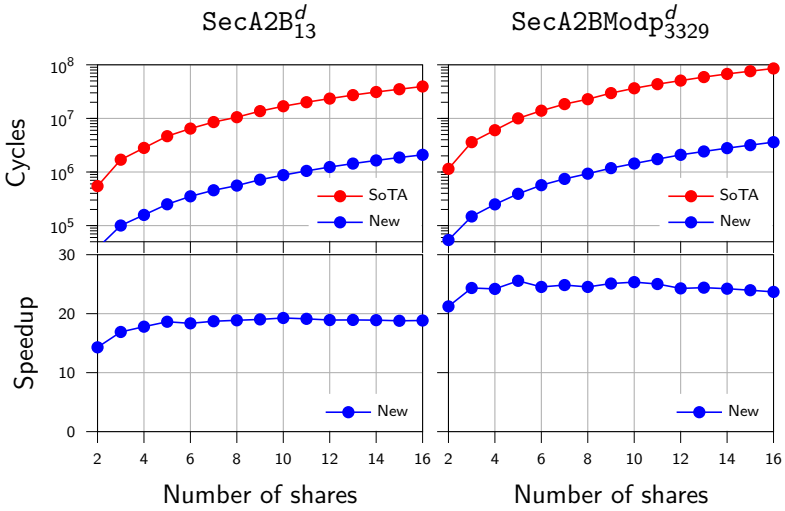
SoTA construction:



Differences:

- PINI removes refreshes.
- Input shares in different positions.
- Proofs based on *"Gadget Embeddings"*.

SecA2B and SecA2BModp performances



`SecB2A` and `SecB2AModp`: Boolean to arithmetic masking.

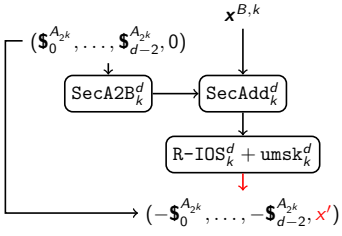
Convert a Boolean k'-bit sharing to arithmetic masking with 2^k modulus ?

SecB2A and SecB2AModp: Boolean to arithmetic masking.

Convert a Boolean k'-bit sharing to arithmetic masking with 2^k modulus ?

Generic $\texttt{SecB2A}_k^d$:

- Converts any x
- Leverage bitsliced gadgets.

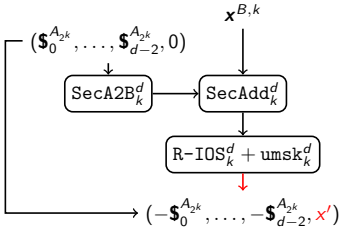


SecB2A and SecB2AModp: Boolean to arithmetic masking.

Convert a Boolean k'-bit sharing to arithmetic masking with 2^k modulus ?

Generic $\texttt{SecB2A}_k^d$:

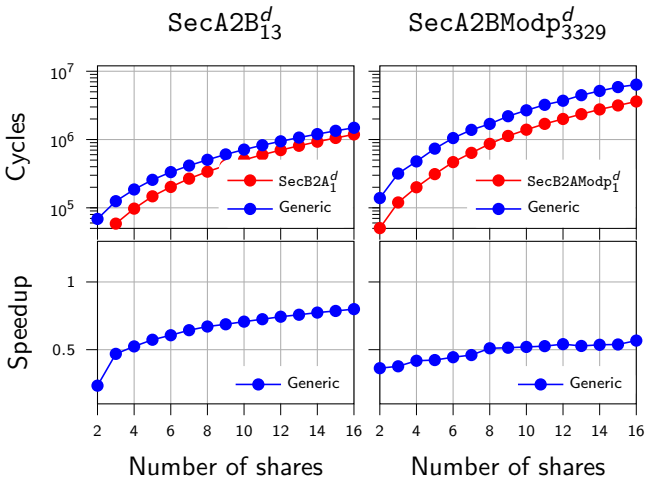
- Converts any x
- Leverage bitsliced gadgets.



Based on $\texttt{SecB2A}_1^d$:

- Converts $x < 2^{k'}$
- How ?
  - Convert all k' bits with $\texttt{SecB2A}_1^d$.
  - Recombine with arithmetic operations.
- $\texttt{SecB2A}_1^d$ does not benefit from bitslice.

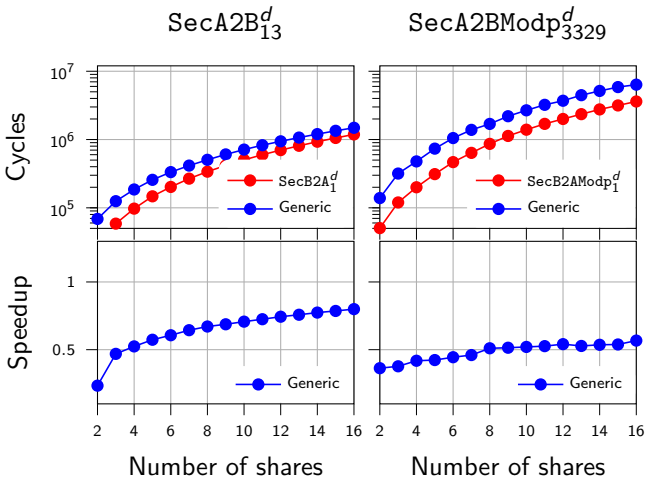
SecB2A and SecB2AModp performances



Observation:

- Better option depends on k'.

SecB2A and SecB2AModp performances



Observation:

- Better option depends on k'.

For Kyber:

- SecB2AModp used for binomial sampling.
- $k' = 3$ hence generic is better.

Binomial sampling

Compute $HW(a) - HW(b)$ *(a and b are* $k' = 2$*-bit values for Kyber768).*

Binomial sampling

Compute $HW(a) - HW(b)$ (a and b are $k' = 2$-bit values for Kyber768).

SOTA: *Chains of half-adders*

- Kyber: 8 AND gates per coefficient

Binomial sampling

Compute $HW(a) - HW(b)$ (a and b are $k' = 2$-bit values for Kyber768).

SOTA: *Chains of half-adders*

- Kyber: 8 AND gates per coefficient

New: *Chains of full-adders*

$$\begin{aligned} HW(a) - HW(b) &= HW(a) + (HW(\neg b) - k') \\ &= HW(a \| \neg b) - k' \end{aligned}$$

a_0

a_1

$\neg b_0$

$\neg b_1$

(subtract k' after `SecB2AModp`)

- Kyber: 3 AND gates per coefficient

Binomial sampling

Compute $HW(a) - HW(b)$ (a and b are $k' = 2$-bit values for Kyber768).

SOTA: *Chains of half-adders*

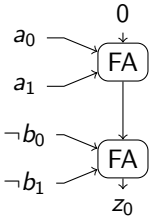
- Kyber: 8 AND gates per coefficient

New: *Chains of full-adders*

$$\begin{aligned} HW(a) - HW(b) &= HW(a) + (HW(\neg b) - k') \\ &= HW(a \| \neg b) - k' \end{aligned}$$

(subtract k' after `SecB2AModp`)

- Kyber: 3 AND gates per coefficient



Binomial sampling

Compute $HW(a) - HW(b)$ (a and b are $k' = 2$-bit values for Kyber768).

SOTA: *Chains of half-adders*

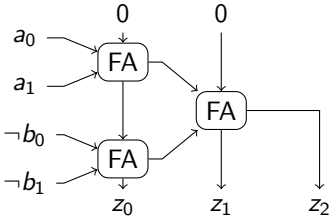
- Kyber: 8 AND gates per coefficient

New: *Chains of full-adders*

$$\begin{aligned} HW(a) - HW(b) &= HW(a) + (HW(\neg b) - k') \\ &= HW(a \| \neg b) - k' \end{aligned}$$

(subtract k' after `SecB2AModp`)

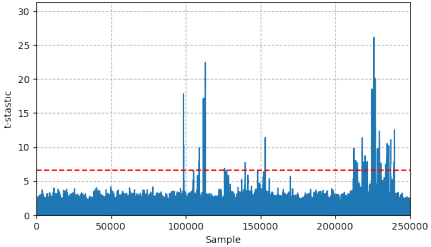
- Kyber: 3 AND gates per coefficient



Contents

Implementation on Cortex-M4

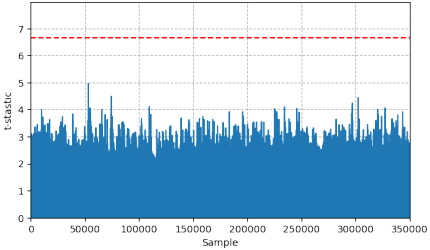
(example with $d = 2$)



C implementation:

- High-level C code.
- `gcc -O2 -flto`.
- Leads to first order leakage.

Implementation on Cortex-M4 (example with $d = 2$)



C implementation:

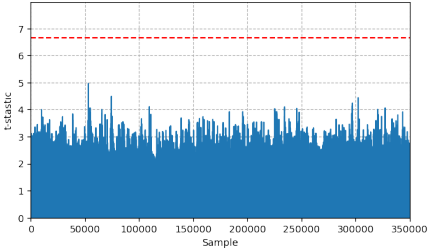
- High-level C code.
- `gcc -O2 -flto`.
- Leads to first order leakage.

Using assembly code:

- ASM code for secure gates (e.g., PINI1-AND, XOR).
- Defensive regarding leakage:
  - One single share in register file at the time.
  - (Micro-arch.) state "cleaning".

Implementation on Cortex-M4

(example with $d = 2$)



C implementation:

- High-level C code.
- `gcc -O2 -flto`.
- Leads to first order leakage.

Using assembly code:

- ASM code for secure gates (e.g., PINI1-AND, XOR).
- Defensive regarding leakage:
  - One single share in register file at the time.
  - (Micro-arch.) state "cleaning".
- Performance overheads ≈ 1.6.

Conclusion

Performance ≈5x improvement over [BGRSV21] for Kyber768 ($d = 4$)

Conclusion

Performance ≈5x improvement over [BGRSV21] for Kyber768 ($d = 4$)

- ≈20x for arithmetic ↔ Boolean conversion gadgets.

Conclusion

Performance ≈5x improvement over [BGRSV21] for Kyber768 ($d = 4$)

- ≈20x for arithmetic ↔ Boolean conversion gadgets.
- Kyber-specific algorithmic optimizations:
  - Compression using `SecA2B` [CGMZ21b,DHP+22].
  - Hamming Weight computation.

Conclusion

Performance ≈5x improvement over [BGRSV21] for Kyber768 ($d = 4$)

- ≈20x for arithmetic ↔ Boolean conversion gadgets.
- Kyber-specific algorithmic optimizations:
  - Compression using `SecA2B` [CGMZ21b,DHP+22].
  - Hamming Weight computation.
- Keccak is now the bottleneck.

Conclusion

Performance ≈5x improvement over [BGRSV21] for Kyber768 ($d = 4$)

- ≈20x for arithmetic ↔ Boolean conversion gadgets.
- Kyber-specific algorithmic optimizations:
  - Compression using `SecA2B` [CGMZ21b,DHP+22].
  - Hamming Weight computation.
- Keccak is now the bottleneck.

Security

- Algorithmic noise (1 target bit out of 32 bits).

Conclusion

Performance ≈5x improvement over [BGRSV21] for Kyber768 ($d = 4$)

- ≈20x for arithmetic ↔ Boolean conversion gadgets.
- Kyber-specific algorithmic optimizations:
  - Compression using `SecA2B` [CGMZ21b,DHP+22].
  - Hamming Weight computation.
- Keccak is now the bottleneck.

Security

- Algorithmic noise (1 target bit out of 32 bits).
- Easy to securely implement: `SecAnd` and `Refresh`.
- Trivial composition with PINI.

Conclusion

see: https://github.com/uclcrypto/pqm4_masked

Performance ≈5x improvement over [BGRSV21] for Kyber768 ($d = 4$)

- ≈20x for arithmetic ↔ Boolean conversion gadgets.
- Kyber-specific algorithmic optimizations:
  - Compression using `SecA2B` [CGMZ21b,DHP+22].
  - Hamming Weight computation.
- Keccak is now the bottleneck.

Security

- Algorithmic noise (1 target bit out of 32 bits).
- Easy to securely implement: `SecAnd` and `Refresh`.
- Trivial composition with PINI.