Bitslicing Arithmetic/Boolean Masking Conversions for Fun and Profit
with Application to Lattice-Based KEMs

Olivier Bronchain    Gaëtan Cassiers

CHES 2022, Leuven, Belgium
Contents

Masking Kyber

Bitslicing

New gadgets

Embedded software implementation
Masked Kyber computation

(Impressed 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.

O. Bronchain, G. Cassiers
Masked Kyber performance

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

How have we improved the performances of masked Kyber?

Run time proportion
Masked Kyber performance

![Graph showing performance comparison](image)

How have we improved the performances of masked Kyber?

- Optimize CBD and Compressions.
- Optimize masking conversion algorithms.
How have we improved the performances of masked Kyber?

▶ Optimize CBD and Compressions.
▶ Optimize masking conversion algorithms.
Masked Kyber performance

How have we improved the performances of masked Kyber?

▶ Optimize CBD and Compressions.
▶ Optimize masking conversion algorithms.
Masked Kyber performance

How have we improved the performances of masked Kyber?
▶ Optimize CBD and Compressions.
▶ Optimize masking conversion algorithms.
How have we improved the performances of masked Kyber?

- Optimize CBD and Compressions.
How have we improved the performances of masked Kyber?

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

Masking Kyber

Bitslicing

New gadgets

Embedded software implementation

Conclusion
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

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

![Diagram showing state representation](image-url)
Canonical vs bitslice representations

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

32-bit regs.

13-bit coeffs.

reg. 0
reg. 1
reg. 2
reg. 3
reg. 4
reg. 5
reg. 6
reg. 7
reg. 8
reg. 9
reg. 10
reg. 11
reg. 12
Canonical vs bitslice representations: performance

+ Arithmetic operations (+, ×).
  - Single-bit processing.
  - Memory/registers usage.
Canonical vs bitslice representations: performance

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

- Bitwise operations: throughput.
  + Security: registers fully used.
  - Representation change cost.
Canonical vs bitslice representations: performance

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

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

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

Masking Kyber

Bitslicing

New gadgets

Embedded software implementation
**SecAdd\(_{k+1}^d\): arithmetic addition of Boolean shares**

What is SoTA?

- Masked Kogge-Stone adders.
- \(\log k\) times SecAnd\(_k^d\).
SecAdd\(_{k+1}^d\): arithmetic addition of Boolean shares

What is SoTA?

- Masked Kogge-Stone adders.
- \(\log k\) times SecAnd\(_k^d\).

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 SecAnd\(_1^d\) per SecFullAdd\(_d^d\).

- \(k\) times SecAnd\(_1^d\).

\[
\begin{array}{cccccc}
  x^{B,1}[0] & y^{B,1}[0] & x^{B,1}[1] & y^{B,1}[1] & x^{B,1}[k-1] & y^{B,1}[k-1] \\
  \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
  \text{SecFullAdd}\_d^d & \text{SecFullAdd}\_d^d & \cdots & \text{SecFullAdd}\_d^d & \\
  z^{B,1}[0] & z^{B,1}[1] & & z^{B,1}[k-1] & z^{B,1}[k]
\end{array}
\]
SecAdd\(_{k+1}^d\): arithmetic addition of Boolean shares

What is SoTA?

- Masked Kogge-Stone adders.
- \(\log k\) times SecAnd\(_k^d\).

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 SecAnd\(_1^d\) per SecFullAdd\(_d^d\).

- \(k\) times SecAnd\(_1^d\).

\[
\begin{align*}
x^{B,1}[0] & \quad y^{B,1}[0] & \quad x^{B,1}[1] & \quad y^{B,1}[1] & \quad x^{B,1}[k−1] & \quad y^{B,1}[k−1] \\
\downarrow & & \downarrow & & \downarrow & & \downarrow \\
\text{SecFullAdd}^d & \quad \text{SecFullAdd}^d & \quad \cdots & \quad \text{SecFullAdd}^d \\
\downarrow & & \downarrow & & \downarrow & & \downarrow \\
z^{B,1}[0] & \quad z^{B,1}[1] & \quad z^{B,1}[k−1] & \quad z^{B,1}[k] \\
\end{align*}
\]

→ Security: trivial composition with PINI.
→ Reduced randomness and operations.
**SecAddModp^d_p**: addition with arbitrary modulus.

<table>
<thead>
<tr>
<th><strong>SoTA:</strong></th>
<th><strong>New:</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Add x and y.</td>
<td>1. Add x and y.</td>
</tr>
<tr>
<td>3. Select output with underflow bit.</td>
<td>3. Add p if underflow occurred.</td>
</tr>
</tbody>
</table>

Taking advantage of bitslice:
- ▶ SecAdd^d_k ≈ SecAnd^d_k.
- ▶ Replace 2 SecAnd^d_k with 1 SecAdd^d_k.
- Secure composition: PINI.
SecAddModp^d_p: addition with arbitrary modulus.

**SoTA:**

1. Add \( x \) and \( y \).

**New:**

\[
\begin{align*}
\rightarrow & \quad x^{B,k} \quad y^{B,k} \\
\downarrow & \\
SecAdd^d_{k+1} & \\
\end{align*}
\]
SecAddModp^d_p: addition with arbitrary modulus.

**SoTA:**
1. Add \( x \) and \( y \).
2. Subtract \( p \).

**New:**

\[
\begin{align*}
    x^{B,k} & \quad \downarrow \quad y^{B,k} \\
    \text{SecAdd}^d_{k+1} & \downarrow -p \\
    -p & \downarrow \\
    \text{SecAdd}^d_k & \\
\end{align*}
\]
**SecAddModp^d_p**: addition with arbitrary modulus.

**SoTA:**

1. Add $x$ and $y$.
3. Select output with underflow bit.

**New:**
SecAddModp^d_p: addition with arbitrary modulus.

**SoTA:**
1. Add \( x \) and \( y \).
2. Subtract \( p \).
3. Select output with underflow bit.

\[
\begin{align*}
&x^{B,k} \quad y^{B,k} \\
\downarrow & \quad \downarrow \\
\text{SecAdd}^d_{k+1} & \quad -p \\
\downarrow & \quad \downarrow \\
\text{SecAdd}^d_k & \\
\downarrow & \\
\text{R-SNI}^d_1 & \quad \text{R-SNI}^d_1 \\
\downarrow & \quad \downarrow \\
\text{SecAnd}^d_k & \quad \text{SecAnd}^d_k \\
\downarrow & \quad \downarrow \\
+ & \\
\downarrow \\
&
\end{align*}
\]

**New:**
1. Add \( x \) and \( y \).
2. Subtract \( p \).

\[
\begin{align*}
&x^{B,k} \quad y^{B,k} \\
\downarrow & \quad \downarrow \\
\text{SecAdd}^d_{k+1} & \quad -p \\
\downarrow & \quad \downarrow \\
\text{SecAdd}^d_k & \\
\downarrow & \\
&
\end{align*}
\]
**SecAddModp^d_p**: addition with arbitrary modulus.

**SoTA:**
1. Add x and y.
3. Select output with underflow bit.

**New:**
1. Add x and y.
3. Add p if underflow occurred.

O. Bronchain, G. Cassiers

Bitslicing Arithmetic/Boolean Masking Conversions for Fun and Profit

10 / 19
**SecAddModp^d_p:** addition with arbitrary modulus.

**SoTA:**
1. Add \(x\) and \(y\).
2. Subtract \(p\).
3. Select output with underflow bit.

\[
\begin{align*}
x^{B,k} & \quad y^{B,k} \\
& \quad \downarrow \quad \downarrow \\
\text{SecAdd}^d_{k+1} & \quad -p \\
& \quad \downarrow \\
\text{SecAdd}^d_k & \\
\text{R-SNI}_1^d & \quad \text{R-SNI}_1^d \\
& \quad \downarrow \quad \downarrow \\
\text{SecAnd}^d_k & \quad \text{SecAnd}^d_k \\
\oplus_k & \quad \downarrow \\
& \quad \downarrow \\
\end{align*}
\]

**New:**
1. Add \(x\) and \(y\).
2. Subtract \(p\).
3. Add \(p\) if underflow occurred.

\[
\begin{align*}
x^{B,k} & \quad y^{B,k} \\
& \quad \downarrow \quad \downarrow \\
\text{SecAdd}^d_{k+1} & \quad -p \\
& \quad \downarrow \\
\text{SecAdd}^d_k & \\
\text{SecAdd}^d_k & \quad \text{SecAdd}^d_{k+1} \\
\text{SecAdd}^d_k & \quad \text{SecAdd}^d_k \\
\times & \quad \downarrow \quad \downarrow \\
& \quad \downarrow \quad \downarrow \\
\text{SecAdd}^d_k & \quad \text{SecAdd}^d_k \\
\end{align*}
\]

Taking advantage of bitslice:
- \(\text{SecAdd}^d_k \approx \text{SecAnd}^d_k\).
- Replace 2 \(\text{SecAnd}^d_k\) with 1 \(\text{SecAdd}^d_k\).
- Secure composition: PINI.
SecAdd and SecAddModp performances

\[\text{SecAdd}_{13}^{d}\]

\[\text{SecAddModp}_{3329}^{d}\]

- **Cycles**
  - \[\text{SoTA}\] and \[\text{New}\] lines show the cycle counts for different numbers of shares.

- **Speedup**
  - \[\text{New}\] line indicates the speedup compared to the baseline.

- **Number of shares**
  - Graphs display performance metrics for 2 to 16 shares.

---

O. Bronchain, G. Cassiers

Bitslicing Arithmetic/Boolean Masking Conversions for Fun and Profit
SecA2B^d_\text{B_k}: arithmetic to Boolean masking conversions

How to convert \(c^{A_{2k}}\) to \(c^{B_k}\) with:
\[
c = \sum_i c_i^{A_{2k}} \mod 2^k = \bigoplus_i c_i^{B_k}
\]

How to perform the conversion?

- \((c_i^{A_{2k}}, 0)\) is a Boolean sharing of \(c_i^{A_{2k}}\).
- Add all the \(c_i^{A_{2k}}\)'s with SecAdd_k's.
SecA2B_k: arithmetic to Boolean masking conversions

How to convert \( c^{A_{2^k}} \) to \( c^{B_{i,k}} \) with:

\[
c = \sum_i c_i^{A_{2^k}} \mod 2^k = \bigoplus_i c_i^{B_{i,k}}
\]

How to perform the conversion?

- \((c_i^{A_{2^k}}, 0)\) is a Boolean sharing of \(c_i^{A_{2^k}}\).
- Add all the \(c_i^{A_{2^k}}\)'s with SecAdd_k's.

SoTA construction:

\[
\begin{align*}
(c_0^{A_{2^k}}, 0) & \quad (c_1^{A_{2^k}}, 0) \\
R{-}\text{SNI}^2_k & \downarrow \quad R{-}\text{SNI}^2_k \\
\quad \quad \quad \quad \quad \quad \quad \quad SecAdd^2_k
\end{align*}
\]
**SecA2B\(_d\)_k**: arithmetic to Boolean masking conversions

How to convert \(c^{A_{2k}}\) to \(c^{B_{1k}}\) with:

\[
c = \sum_{i} c_{i}^{A_{2k}} \mod 2^k = \bigoplus_{i} c_{i}^{B_{1k}}
\]

How to perform the conversion?

- \((c_{i}^{A_{2k}}, 0)\) is a Boolean sharing of \(c_{i}^{A_{2k}}\).
- Add all the \(c_{i}^{A_{2k}}\)'s with SecAdd\(_k\)'s.

**SoTA construction:**

\[
\begin{align*}
&d = 2 \\
&\begin{cases}
(c_0^{A_{1k}}, 0) \quad &\quad (c_1^{A_{1k}}, 0) \\
\text{SecAdd}^2_k & \quad \text{R-SNI}^2_k \\
\text{R-SNI}^2_k & \quad \text{R-SNI}^2_k
\end{cases}
\end{align*}
\]

\[
\begin{align*}
&d = 4 \\
&\begin{cases}
(c_0^{B_{1k}}, c_1^{B_{1k}}, 0, 0) \quad &\quad (c_2^{A_{1k}}, 0, 0) \\
\text{R-SNI}^4_k & \quad \text{R-SNI}^4_k \\
\text{R-SNI}^4_k & \quad \text{R-SNI}^4_k
\end{cases}
\end{align*}
\]

**Differences:**

- PINI removes refreshes.
- Input shares in different positions.
- Proofs based on "Gadget Embeddings".
**SecA2B\(_d\)_:** arithmetic to Boolean masking conversions

How to convert \(c^{A_{2k}}\) to \(c^{B_{1k}}\) with:

\[
c = \sum_{i} c_{i}^{A_{2k}} \mod 2^k = \bigoplus_{i} c_{i}^{B_{1k}}
\]

How to perform the conversion?

- \((c_{i}^{A_{2k}}, 0)\) is a Boolean sharing of \(c_{i}^{A_{2k}}\).
- Add all the \(c_{i}^{A_{2k}}\)'s with SecAdd\(_k\)'s.

New PINI construction:

<table>
<thead>
<tr>
<th>(d=2)</th>
<th>(d=4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>((c^{A_{2k}}<em>{0}, 0), (0, c^{A</em>{2k}}_{1}))</td>
<td>((c^{A_{2k}}<em>{0}, 0), (0, c^{A</em>{2k}}_{3}))</td>
</tr>
<tr>
<td>(\text{SecAdd}_{k}^{2})</td>
<td>(\text{SecAdd}_{k}^{2})</td>
</tr>
<tr>
<td>(\downarrow)</td>
<td>(\downarrow)</td>
</tr>
<tr>
<td>((c'^{B_{1k}}<em>{0}, c'^{B</em>{1k}}_{1}, 0, 0))</td>
<td>((0, 0, c'^{B_{1k}}<em>{0}, c'^{B</em>{1k}}_{1}))</td>
</tr>
<tr>
<td>(\text{SecAdd}_{k}^{4})</td>
<td>(\text{SecAdd}_{k}^{4})</td>
</tr>
</tbody>
</table>

**SoTA construction:**

\[
\begin{align*}
\text{d} = 2 & : \quad \begin{cases}
(c^{A_{2k}}_{0}, 0) & \rightarrow \text{R-SNI}_{k}^{2} \rightarrow \text{R-SNI}_{k}^{2} \rightarrow \text{SecAdd}_{k}^{2} \\
(c^{A_{2k}}_{1}, 0) & \rightarrow \text{R-SNI}_{k}^{2} \rightarrow \text{R-SNI}_{k}^{2} \rightarrow \text{SecAdd}_{k}^{2}
\end{cases} \\
\text{d} = 4 & : \quad \begin{cases}
(c^{A_{2k}}_{0}, 0) & \rightarrow \text{R-SNI}_{k}^{4} \rightarrow \text{R-SNI}_{k}^{4} \rightarrow \text{SecAdd}_{k}^{4} \\
(c^{A_{2k}}_{1}, 0) & \rightarrow \text{R-SNI}_{k}^{4} \rightarrow \text{R-SNI}_{k}^{4} \rightarrow \text{SecAdd}_{k}^{4}
\end{cases}
\end{align*}
\]

**Differences:**

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

---

O. Bronchain, G. Cassiers

Bitslicing Arithmetic/Boolean Masking Conversions for Fun and Profit
SecA2B and SecA2BModp performances

![Graphs showing the comparison between SecA2B and SecA2BModp performances.](image)

- **SecA2B**
  - **SecA2B\(_d^{13}\)**
  - **SecA2BModp\(_d^{3329}\)**

- **Cycles**
  - **SoTA**
  - **New**

- **Speedup**
  - **New**

**Number of shares**: 2, 4, 6, 8, 10, 12, 14, 16
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 SecB2A$_d^k$:

- Converts any $x$
- Leverage bitsliced gadgets.

\[
\begin{align*}
(x^{B,k} &\downarrow) \\
(S_0^{A_{2^k}}, \ldots, S_{d-2}^{A_{2^k}}, 0) &\downarrow \\
\text{SecA2B}_k^d &\rightarrow \text{SecAdd}_k^d \\
R-\text{IOS}_k^d + \text{umsk}_k^d &\downarrow \\
(-S_0^{A_{2^k}}, \ldots, -S_{d-2}^{A_{2^k}}, x') &\rightarrow
\end{align*}
\]
SecB2A and SecB2AModp: Boolean to arithmetic masking.

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

**Generic SecB2A\(_d\)\(^k\):**
- Converts any \( x \)
- Leverage bitsliced gadgets.

\[
\begin{align*}
(x^{B,k}) & \xrightarrow{\text{SecA2B}_d^k} \text{SecAdd}_d^k \\
(A_0^{2^k}, \ldots, A_{d-2}^{2^k}, 0) & \xrightarrow{\text{SecB2A}_d^k} (-A_0^{2^k}, \ldots, -A_{d-2}^{2^k}, x')
\end{align*}
\]

**Based on SecB2A\(_1\)\(^d\):**
- Converts \( x < 2^{k'} \)
- How?
  - Convert all \( k' \) bits with SecB2A\(_1\)\(^d\).
  - Recombine with arithmetic operations.
- SecB2A\(_1\)\(^d\) does not benefit from bitslice.
SecB2A and SecB2AModp performances

Observation:
- Better option depends on $k'$. 

For Kyber:
- SecB2AModp used for binomial sampling.
- $k' = 3$ hence generic is better.
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 $\text{HW}(a) - \text{HW}(b)$ ($a$ 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

\[
\text{HW}(a) - \text{HW}(b) = \text{HW}(a) + (\text{HW}(\neg b) - k') = \text{HW}(a\parallel \neg b) - k'
\]

(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

\[
HW(a) - HW(b) = HW(a) + (HW(\neg b) - k')
\]
\[
= HW(a || \neg b) - k'
\]

(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 halfadders

- Kyber: 8 AND gates per coefficient

**New:** Chains of fulladders

$$HW(a) - HW(b) = HW(a) + (HW(\neg b) - k')$$
$$= HW(a \parallel \neg b) - k'$$

(subtract $k'$ after SecB2AModp)

- Kyber: 3 AND gates per coefficient
Contents

Masking Kyber

Bitslicing

New gadgets

Embedded software implementation
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”.

O. Bronchain, G. Cassiers

Bitslicing Arithmetic/Boolean Masking Conversions for Fun and Profit
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 $\approx 1.6$. 

O. Bronchain, G. Cassiers

Bitslicing Arithmetic/Boolean Masking Conversions for Fun and Profit
Conclusion

**Performance** $\approx 5x$ improvement over [BGRSV21] for Kyber768 ($d = 4$)
Conclusion

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

- $\approx 20x$ for arithmetic $\leftrightarrow$ Boolean conversion gadgets.
Conclusion

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

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

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

- $\approx 20x$ 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 $\approx 5x$ improvement over [BGRSV21] for Kyber768 ($d = 4$)

- $\approx 20x$ for arithmetic $\leftrightarrow$ Boolean conversion gadgets.
- Kyber-specific algorithmic optimizations:
  - Compression using $\text{SecA2B}$ [CGMZ21b, DHP+22].
  - Hamming Weight computation.
- Keccak is now the bottleneck.

Security

- Algorithmic noise (1 target bit out of 32 bits).
Performance $\approx 5\times$ improvement over [BGRSV21] for Kyber768 ($d = 4$)

- $\approx 20\times$ 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

Performance \(\approx 5x\) improvement over [BGRSV21] for Kyber768 (\(d = 4\))

- \(\approx 20x\) 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.

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