# High-Order and Cortex-M4 First-Order Implementations of Masked FrodoKEM

François Gérard[1], Morgane Guerreau[2]

[1]University of the Bundeswehr Munich

[2]PQShield


# FrodoKEM

- Lattice-based Key Encapsulation Mechanism
- Not a NIST standard but...
- ANSSI and BSI's favorite
- Based on plain LWE
- Needs the FO transform to achieve CCA2 security

# FrodoKEM PKE

Public parameter $\mathbf{A} \in \mathbb{Z}_q^{n \times n}$ with $n \in \{640, 976, 1344\}$ and $q$ a power of two

PKE.Keygen()

1. $\mathbf{S}, \mathbf{E} \leftarrow \psi(\cdot)$
2. $\mathbf{B} \leftarrow \mathbf{AS} + \mathbf{E}$
3. Return $(pk = \mathbf{B}, sk = \mathbf{S})$

# FrodoKEM PKE

Public parameter $\mathbf{A} \in \mathbb{Z}_q^{n \times n}$ with $n \in \{640, 976, 1344\}$ and $q$ a power of two

PKE.Keygen()

1. $\mathbf{S}, \mathbf{E} \leftarrow \psi(\cdot)$
2. $\mathbf{B} \leftarrow \mathbf{AS} + \mathbf{E}$
3. Return $(pk = \mathbf{B}, sk = \mathbf{S})$

PKE.Encrypt()

1. $\mathbf{S}', \mathbf{E}', \mathbf{E}'' \leftarrow \psi(\cdot)$
2. $\mathbf{B}' \leftarrow \mathbf{S}'\mathbf{A} + \mathbf{E}'$
3. $\mathbf{V} \leftarrow \mathbf{S}'\mathbf{B} + \mathbf{E}'' + \mathsf{Encode}(u')$
4. Return $(\mathbf{B}', \mathbf{V})$

# FrodoKEM PKE

Public parameter $\mathbf{A} \in \mathbb{Z}_q^{n \times n}$ with $n \in \{640, 976, 1344\}$ and $q$ a power of two

PKE.Keygen()

1. $\mathbf{S}, \mathbf{E} \leftarrow \psi(\cdot)$
2. $\mathbf{B} \leftarrow \mathbf{AS} + \mathbf{E}$
3. Return $(pk = \mathbf{B}, sk = \mathbf{S})$

PKE.Encrypt()

1. $\mathbf{S}', \mathbf{E}', \mathbf{E}'' \leftarrow \psi(\cdot)$
2. $\mathbf{B}' \leftarrow \mathbf{S}'\mathbf{A} + \mathbf{E}'$
3. $\mathbf{V} \leftarrow \mathbf{S}'\mathbf{B} + \mathbf{E}'' + \mathsf{Encode}(u')$
4. Return $(\mathbf{B}', \mathbf{V})$

PKE.Decrypt()

1. $\mathbf{M} \leftarrow \mathbf{V} - \mathbf{B}'\mathbf{S}$
2. Return $\mathsf{Decode}(\mathbf{M})$

# FrodoKEM PKE

Public parameter $\mathbf{A} \in \mathbb{Z}_q^{n \times n}$ with $n \in \{640, 976, 1344\}$ and $q$ a power of two

PKE.Keygen()

1. $\mathbf{S}, \mathbf{E} \leftarrow \psi(\cdot)$
2. $\mathbf{B} \leftarrow \mathbf{AS} + \mathbf{E}$
3. Return $(pk = \mathbf{B}, sk = \mathbf{S})$

PKE.Encrypt()

1. $\mathbf{S}', \mathbf{E}', \mathbf{E}'' \leftarrow \psi(\cdot)$
2. $\mathbf{B}' \leftarrow \mathbf{S}'\mathbf{A} + \mathbf{E}'$
3. $\mathbf{V} \leftarrow \mathbf{S}'\mathbf{B} + \mathbf{E}'' + \mathsf{Encode}(u')$
4. Return $(\mathbf{B}', \mathbf{V})$

PKE.Decrypt()

1. $\mathbf{M} \leftarrow \mathbf{V} - \mathbf{B}'\mathbf{S}$
2. Return $\mathsf{Decode}(\mathbf{M})$

$$\begin{aligned}\mathbf{V} - \mathbf{B}'\mathbf{S} &= \mathsf{Encode}(u') + \mathbf{S}'\mathbf{B} + \mathbf{E}'' - \mathbf{S}'\mathbf{AS} - \mathbf{E}'\mathbf{S} \\ &= \mathsf{Encode}(u') + \mathbf{S}'\mathbf{E} + \mathbf{E}'' - \mathbf{E}'\mathbf{S} \\ &\approx \mathsf{Encode}(u')\end{aligned}$$

# FrodoKEM vs ML-KEM

From a high-level perspective, both ML-KEM and FrodoKEM are constructed similarly using the Learning With Errors problem. However, here are some notable differences:

- Plain LWE instead of MLWE (No polynomial ring/NTT)
- Gaussian distribution instead of the binomial
- Power of two modulus instead of a prime
- No ciphertext compression

# Masking

- Common generic countermeasure against side-channel attacks
- Boolean masking

$$x = x_1 \oplus \cdots \oplus x_t \in \{0,1\}^k$$

- Arithmetic masking

$$y = y_1 + \cdots + y_t \pmod{2^k}$$

- Both types of masking are required to mask lattice-based schemes

# What should we mask?

**Input:** $c = c_1 \| c_2 \| salt,\ sk = s \| seed_A \| b \| \mathbf{S}^T$
**Output:** $ss$

$\mathbf{B}' \leftarrow \mathsf{Unpack}(c_1, \bar{n}, n)$
$\mathbf{C} \leftarrow \mathsf{Unpack}(c_2, \bar{n}, \bar{n})$
$\mathbf{M} \leftarrow \mathbf{C} - \mathbf{B}'\mathbf{S}$
$u' \leftarrow \mathsf{Decode}(\mathbf{M})$
$seed_{SE'} \| k' \leftarrow \mathsf{SHAKE}(u' \| salt)$
$\mathbf{S}', \mathbf{E}', \mathbf{E}'' \leftarrow \mathsf{SampleMatrix}(seed_{SE'})$
$\mathbf{A} \leftarrow \mathsf{Gen}(seed_A)$
$\mathbf{B}'' \leftarrow \mathbf{S}'\mathbf{A} + \mathbf{E}'$
$\mathbf{B} \leftarrow \mathsf{Unpack}(b, n, \bar{n})$
$\mathbf{V} \leftarrow \mathbf{S}'\mathbf{B} + \mathbf{E}''$
$\mathbf{C}' \leftarrow \mathbf{V} + \mathsf{Encode}(u')$
$\bar{k} \leftarrow k'$ **if** $\mathbf{B}' \| \mathbf{C} = \mathbf{B}'' \| \mathbf{C}'$ **else** $\bar{k} \leftarrow s$
$ss \leftarrow \mathsf{SHAKE}(c_1 \| c_2 \| salt \| \bar{k})$
**return** $ss$

- Matrix operations

Nothing fancy here.

$$\mathbf{A}(\mathbf{S}_1 + \mathbf{S}_2 + \cdots + \mathbf{S}_t) = \mathbf{A}\mathbf{S}_1 + \mathbf{A}\mathbf{S}_2 + \cdots + \mathbf{A}\mathbf{S}_t$$

# What should we mask?

**Input:** $c = c_1 \| c_2 \| salt,\ sk = s \| seed_A \| b \| \mathbf{S}^T$
**Output:** $ss$

$\mathbf{B}' \leftarrow \mathsf{Unpack}(c_1, \bar{n}, n)$
$\mathbf{C} \leftarrow \mathsf{Unpack}(c_2, \bar{n}, \bar{n})$
$\mathbf{M} \leftarrow \mathbf{C} - \mathbf{B}'\mathbf{S}$
$u' \leftarrow \mathsf{Decode}(\mathbf{M})$
$seed_{SE'} \| k' \leftarrow \mathsf{SHAKE}(u' \| salt)$
$\mathbf{S}', \mathbf{E}', \mathbf{E}'' \leftarrow \mathsf{SampleMatrix}(seed_{SE'})$
$\mathbf{A} \leftarrow \mathsf{Gen}(seed_A)$
$\mathbf{B}'' \leftarrow \mathbf{S}'\mathbf{A} + \mathbf{E}'$
$\mathbf{B} \leftarrow \mathsf{Unpack}(b, n, \bar{n})$
$\mathbf{V} \leftarrow \mathbf{S}'\mathbf{B} + \mathbf{E}''$
$\mathbf{C}' \leftarrow \mathbf{V} + \mathsf{Encode}(u')$
$\bar{k} \leftarrow k'$ **if** $\mathbf{B}' \| \mathbf{C} = \mathbf{B}'' \| \mathbf{C}'$ **else** $\bar{k} \leftarrow s$
$ss \leftarrow \mathsf{SHAKE}(c_1 \| c_2 \| salt \| \bar{k})$
**return** $ss$

- Matrix operations
- Hashing

Nothing fancy here neither.
We can reuse state-of-the-art.

# What should we mask?

**Input:** $c = c_1 \| c_2 \| salt,\ sk = s \| seed_A \| b \| \mathbf{S}^T$
**Output:** $ss$

$\mathbf{B}' \leftarrow \mathsf{Unpack}(c_1, \bar{n}, n)$
$\mathbf{C} \leftarrow \mathsf{Unpack}(c_2, \bar{n}, \bar{n})$
$\mathbf{M} \leftarrow \mathbf{C} - \mathbf{B}'\mathbf{S}$
$u' \leftarrow \mathsf{Decode}(\mathbf{M})$
$seed_{SE'} \| k' \leftarrow \mathsf{SHAKE}(u' \| salt)$
$\mathbf{S}', \mathbf{E}', \mathbf{E}'' \leftarrow \mathsf{SampleMatrix}(seed_{SE'})$
$\mathbf{A} \leftarrow \mathsf{Gen}(seed_A)$
$\mathbf{B}'' \leftarrow \mathbf{S}'\mathbf{A} + \mathbf{E}'$
$\mathbf{B} \leftarrow \mathsf{Unpack}(b, n, \bar{n})$
$\mathbf{V} \leftarrow \mathbf{S}'\mathbf{B} + \mathbf{E}''$
$\mathbf{C}' \leftarrow \mathbf{V} + \mathsf{Encode}(u')$
$\bar{k} \leftarrow k'$ **if** $\mathbf{B}' \| \mathbf{C} = \mathbf{B}'' \| \mathbf{C}'$ **else** $\bar{k} \leftarrow s$
$ss \leftarrow \mathsf{SHAKE}(c_1 \| c_2 \| salt \| \bar{k})$
**return** $ss$

- Matrix operations
- Hashing
- Encoding and decoding

This is starting to get interesting!
This was hard to do with ML-KEM, what about FrodoKEM?

# What should we mask?

**Input:** $c = c_1 \| c_2 \| salt,\ sk = s \| seed_A \| b \| \mathbf{S}^T$
**Output:** $ss$

$\mathbf{B}' \leftarrow \mathsf{Unpack}(c_1, \bar{n}, n)$
$\mathbf{C} \leftarrow \mathsf{Unpack}(c_2, \bar{n}, \bar{n})$
$\mathbf{M} \leftarrow \mathbf{C} - \mathbf{B}'\mathbf{S}$
$u' \leftarrow \mathsf{Decode}(\mathbf{M})$
$seed_{SE'} \| k' \leftarrow \mathsf{SHAKE}(u' \| salt)$
$\mathbf{S}', \mathbf{E}', \mathbf{E}'' \leftarrow \mathsf{SampleMatrix}(seed_{SE'})$
$\mathbf{A} \leftarrow \mathsf{Gen}(seed_A)$
$\mathbf{B}'' \leftarrow \mathbf{S}'\mathbf{A} + \mathbf{E}'$
$\mathbf{B} \leftarrow \mathsf{Unpack}(b, n, \bar{n})$
$\mathbf{V} \leftarrow \mathbf{S}'\mathbf{B} + \mathbf{E}''$
$\mathbf{C}' \leftarrow \mathbf{V} + \mathsf{Encode}(u')$
$\bar{k} \leftarrow k'$ **if** $\mathbf{B}' \| \mathbf{C} = \mathbf{B}'' \| \mathbf{C}'$ **else** $\bar{k} \leftarrow s$
$ss \leftarrow \mathsf{SHAKE}(c_1 \| c_2 \| salt \| \bar{k})$
**return** $ss$

- Matrix operations
- Hashing
- Encoding and decoding
- Comparison

Should be similar to ML-KEM...
Unless?

# What should we mask?

**Input:** $c = c_1 \| c_2 \| salt,\ sk = s \| seed_A \| b \| \mathbf{S}^T$
**Output:** $ss$

$\mathbf{B}' \leftarrow \mathsf{Unpack}(c_1, \bar{n}, n)$
$\mathbf{C} \leftarrow \mathsf{Unpack}(c_2, \bar{n}, \bar{n})$
$\mathbf{M} \leftarrow \mathbf{C} - \mathbf{B}'\mathbf{S}$
$u' \leftarrow \mathsf{Decode}(\mathbf{M})$
$seed_{SE'} \| k' \leftarrow \mathsf{SHAKE}(u' \| salt)$
$\mathbf{S}', \mathbf{E}', \mathbf{E}'' \leftarrow \mathsf{SampleMatrix}(seed_{SE'})$
$\mathbf{A} \leftarrow \mathsf{Gen}(seed_A)$
$\mathbf{B}'' \leftarrow \mathbf{S}'\mathbf{A} + \mathbf{E}'$
$\mathbf{B} \leftarrow \mathsf{Unpack}(b, n, \bar{n})$
$\mathbf{V} \leftarrow \mathbf{S}'\mathbf{B} + \mathbf{E}''$
$\mathbf{C}' \leftarrow \mathbf{V} + \mathsf{Encode}(u')$
$\bar{k} \leftarrow k'$ **if** $\mathbf{B}' \| \mathbf{C} = \mathbf{B}'' \| \mathbf{C}'$ **else** $\bar{k} \leftarrow s$
$ss \leftarrow \mathsf{SHAKE}(c_1 \| c_2 \| salt \| \bar{k})$
**return** $ss$

- Matrix operations
- Hashing
- Encoding and decoding
- Comparison
- Gaussian sampling

This will be a nightmare :D

# FrodoKEM vs ML-KEM (regarding masking)

Both schemes stem from the same line of research. Masking techniques could be reused. However, there are two major differences:

- Power of two modulus ☺:
  - Easier Encode/Decode (Compress/Uncompress in ML-KEM)
  - Faster A2B and B2A conversions
- Gaussian distribution ☹:
  - Not as easy as sampling a binomial

# Gaussian sampler

Table-based technique:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 4643 | 13363 | 20579 | 25843 | 29227 | 31145 | 32103 | 32525 | 32689 | 32745 | 32762 | 32766 | 32767 |

1. Pick a number $r$ between $0$ and $2^{15} - 1 = 32767$
2. Find the smallest index $i$ such that $T[i] \geq r$
3. Flip a coin $s$
4. Output $(-1)^s \cdot i$

Technique improved by Eid et al. (TCHES2026/2)

# Encode/Decode

The message $\mu \in \{0,1\}^{k=\bar{n}^2 \cdot B}$ in the underlying PKE is mapped to $\mathbb{Z}_q^{\bar{n} \times \bar{n}}$during encryption and added to the matrix $\mathbf{V}$.

$$\mathbf{V} \leftarrow \mathbf{S}'\mathbf{B} + \mathbf{E}''$$
$$\mathbf{C}' \leftarrow \mathbf{V} + \mathsf{Encode}(u')$$

This encoding procedure maps the $\bar{n}^2$ $B$-bit chunks $\mu_i$ to $\mathbb{Z}_q$ by computing $\mu_i \cdot q/2^B$. Since $q$ is a power of two, this can be simply implemented by a shift, which is straightfoward on masked bitstrings!

A final B2A conversion is needed to perform the addition in $\mathbb{Z}_q^{\bar{n} \times \bar{n}}$.

# Encode/Decode

The decode operation basically reverts the encode operation by shifting in the other direction. However, since the sequence of operations is

$$u' \leftarrow \text{Decode}(\mathbf{M})$$
$$seed_{SE'} \| k' \leftarrow \text{SHAKE}(u' \| salt)$$

and since the (masked) hash function requires a value in Boolean masked form, the decoding comes ”for free” since the A2B conversion is performed anyway.

# Masked comparison

The last step before computing the shared secret is to verify that the re-encryption is equal to the received ciphertext. In general, this boils down to a masked zero-test

$$\bar{k} \leftarrow k' \textbf{ if } \mathbf{B}'\|\mathbf{C} = \mathbf{B}''\|\mathbf{C}' \textbf{ else } \bar{k} \leftarrow s$$

It is tempting to re-use the techniques from [CGMZ23]. However, most of the tricks in this paper rely on $q$ being a prime and do not apply here. We thus perform a zero-test by performing a masked OR between the individual bits of the value.

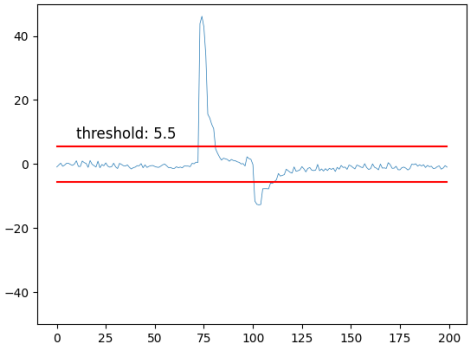
[CGMZ23] High-order Polynomial Comparison and Masking Lattice-based Encryption. TCHES23

# Boolean to Arithmetic conversion in C (5000 traces)

Let's assess how secure our code is in practice!

# Boolean to Arithmetic conversion in C (5000 traces)

Let's assess how secure our code is in practice!



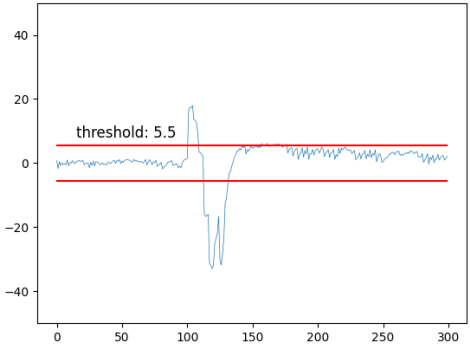

Yeah...

# Boolean to Arithmetic conversion in ASM (5000 traces)

No problem, let's rewrite this in assembly!

# Boolean to Arithmetic conversion in ASM (5000 traces)

No problem, let's rewrite this in assembly!




We need a miracle...

# $\mu$-arch leakage

Leakage due to phenomenon out of control in software implementations:

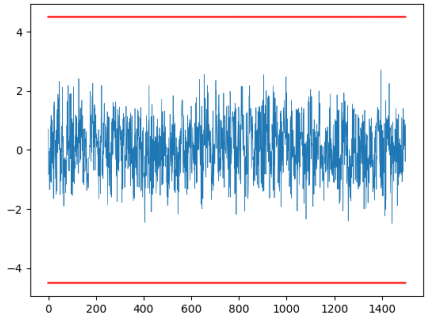
- Glitches
- Registers leaking together
- Pipeline

[MPW22] studied the case of the Cortex-M4 and gave some guidance (e.g. do not load/store multiples shares in a row).

[MPW22]. MIRACLE: MIcRo-ArChitectural Leakage Evaluation: A study of micro-architectural power leakage across many devices.

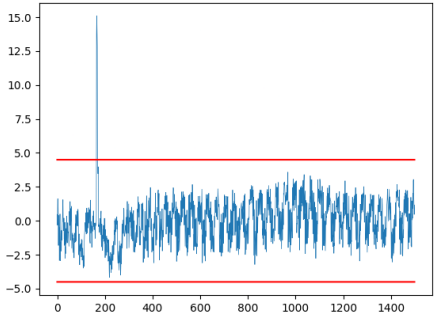
# $\mu$-arch leakage

```
ldrh rx0, [px], #2
ldrh r, [pool]
ldrh rx1, [px], #2
```



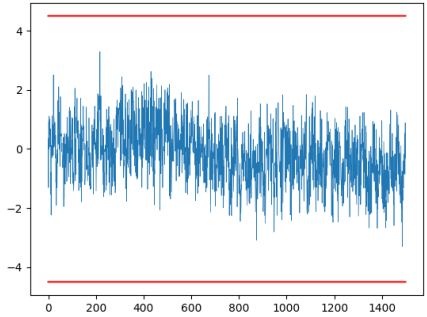
# $\mu$-arch leakage

```
ldrh rx0, [px], #2
ldrh r, [pool]
ldrh rx1, [px], #2

strh t1, [pool]
```



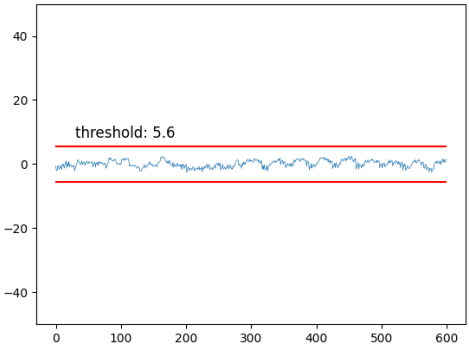
# $\mu$-arch leakage

```
ldrh rx0, [px], #2
ldrh r, [pool]
ldrh rx1, [px], #2

eor t2, t2, t2
strh t1, [pool]
```



# Hardened Boolean to Arithmetic conversion (100 000 traces)

After following good practices and a lot of trial and error...



# Performances

## High Order C code

Benchmarks on x64 of Decaps function (in kilocycles).

| Order | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| Frodo-640 | 57395 | 293703 | 498200 | 809345 | 1138708 | 1523870 | 2010644 |
| Frodo-976 | 93498 | 446945 | 781157 | 1258737 | 1844603 | 2490802 | 3143712 |
| Frodo-1344 | 117860 | 522497 | 903360 | 1550615 | 2215093 | 2962697 | 3874540 |

# Performances

## Hardened ASM

| | C | ASM | $\text{ASM}_h$ |
|---|---|---|---|
| SecAND | 49 | 51 | 68 (+39%) |
| SecADD | 206 | 149 | 248 (+20%) |
| BooleanToArithmetic | 33 | 47 | 54 (+64%) |
| ArithmeticToBoolean | 154 | 125 | 222 (+44%) |
| SecZeroTest | 144 | 126 | 223 (+55%) |

Overhead from hardening ranges from +20% to +65% compared to C code (compiled with -O3).

# Memory usage

Memory usage for the Decaps function (in bytes)

| Order | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| Frodo-640 | 196 368 | 268 232 | 340 112 | 412 008 | 483 880 | 555 760 | 627 632 |
| Frodo-976 | 300 016 | 410 320 | 520 560 | 630 856 | 741 064 | 851 344 | 961 616 |
| Frodo-1344 | 412 144 | 563 728 | 715 312 | 866 896 | 1 018 480 | 1 170 056 | 1 321 648 |

$\mathbf{A}$ amounts for most of the memory footprint and remains unmasked
$\longrightarrow$ Memory usage increases a bit slower than what you could expect

# Randomness usage

Number of calls to `rand_u16()` for the Decaps function

| Order | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| Frodo-640 | 855 481 | 9 388 635 | 18 632 934 | 31 628 890 | 47 673 783 | 66 720 765 | 88 816 684 |
| Frodo-976 | 1 113 553 | 12 524 699 | 24 861 150 | 42 336 282 | 63 884 495 | 89 434 749 | 119 058 084 |
| Frodo-1344 | 1 013 993 | 12 392 155 | 24 611 670 | 42 342 362 | 64 121 191 | 89 850 621 | 119 628 188 |

Most of the randomness is consumed by the masked sampler.
Length of the CDT is smaller for Frodo-1344.
$\longrightarrow$ Randomness does not increase much between Frodo-976 and Frodo-1344.

# Conclusion

- FrodoKEM requires different gadgets than ML-KEM because the modulus is a power of two and the distribution is Gaussian
- Costliest component to mask is the Gaussian sampler
- Memory footprint is **huge**
- As usual, micro-architectural leakage is a problem

Thank you for your attention !

Github