

How small can you (reasonably) get an ML-DSA and ML-KEM implementation?

Mike Ounsworth
OpenSSL Conf 2026

OpenSSL CORPORATION



What is “high-performance” cryptography?

… is it

- Speed? How do you measure speed?
  - Wall time?
  - CPU cycles?
  - Throughput?
- Memory footprint?
  - Size of the compiled binary?
  - Stack usage?
  - Heap usage?

What is “high-performance” cryptography?

Let’s say it’s throughput.

- Throughput on a single-core, single-process machine is easy to measure.
- Concept is fuzzier on a multi-core personal device or server.



What is “high-performance” cryptography?

Let’s say it’s throughput.

- On a hyper-scaler such as a cloud TLS front-end handling billions of connections per second, throughput might actually be bottlenecked on the number of parallel instances and ability to suspend efficiently to cache while waiting for network IO.
- I.e. throughput is Memory-bound, not CPU-bound.
- Suddenly, throughput is about memory footprint, not speed.



What is “high-performance” cryptography?

This talk will walk through the ML-DSA and ML-KEM algorithms with an eye for where you can make speed-for-memory trade-offs.

Based on implementation work conducted earlier this year in the new open-source Bouncy Castle Rust cryptographic library (currently in Alpha).



Rust

https://github.com/bcgit/bc-rust/

Speaker Bio: Mike Ounsworth



```
Terminal
~

→  ~ cat whois_mike_ounsworth.txt
Mike Ounsworth wears many hats:

Lead maintainer of Bouncy Castle Rust crypto library
Adjunct Professor, Université de Sherbrooke, Canada

IETF:
  * Author of 4 RFCs and 12 active Internet Drafts
  * Chair of the ACME Working Group
  * Member of the Security Directorate review team



Past:
  * Software Security Architect and Appsec Tester, Entrust
  * #14 all-time ranked user on security.stackexchange.com
  * M.Sc Robotics, AI, McGill, Canada
  * B.Sc, CompSci, Math, Phys, Queen's University, Canada
→  ~
```

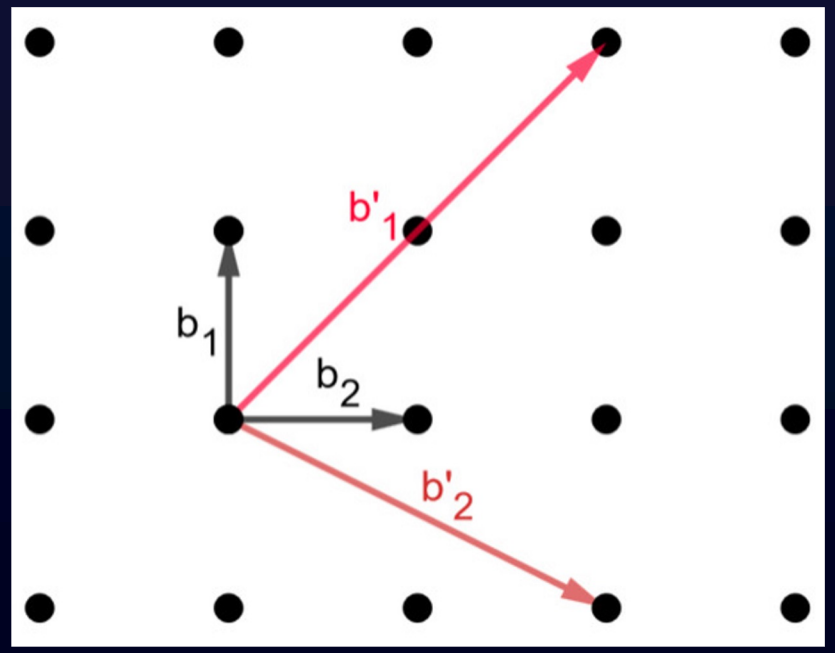
ML-KEM and ML-DSA

Refresher

ML-KEM and ML-DSA
Refresher

The **Module Lattice Key Encapsulation Mechanism (ML-KEM)** is a PQC encryption algorithm.

The **Module Lattice Digital Signature Algorithm (ML-DSA)** is a PQC signature algorithm.



$$b_A = \begin{bmatrix} 4 & 1 & 11 & 10 \\ 5 & 5 & 9 & 5 \\ 3 & 9 & 0 & 10 \\ 1 & 3 & 3 & 2 \\ 12 & 7 & 3 & 4 \\ 6 & 5 & 11 & 4 \\ 3 & 3 & 5 & 0 \end{bmatrix} \times \begin{bmatrix} 6 \\ 9 \\ 11 \\ 11 \end{bmatrix} + \begin{bmatrix} 0 \\ -1 \\ 1 \\ 1 \\ 1 \\ 0 \\ -1 \end{bmatrix} \pmod{13} = \begin{bmatrix} 4 \\ 7 \\ 2 \\ 11 \\ 5 \\ 12 \\ 8 \end{bmatrix}$$

SHA-3

Both ML-DSA and ML-KEM make extensive use of SHA3

SHA-3:

Both ML-DSA and ML-KEM make extensive use of SHA3

Any discussion of performance of ML-DSA / ML-KEM needs to start with performance of SHA3.

Object	Size (bytes)
SHAKE128/256	440
Suspended state (Suspendable trait)	415

SHA-3:
Both ML-KEM and ML-DSA make extensive use of SHA3

MLKEM.SampleNTT(ρ) / MLDSA.RejNTTPoly(ρ) :

Squeeze 3 bytes at a time from SHAKE128/256 inside a rejection loop:

Algorithm 7 SampleNTT(B)

Takes a 32-byte seed and two indices as input and

Input: byte array $B \in \mathbb{B}^{34}$.
Output: array $\hat{a} \in \mathbb{Z}_q^{256}$.

1: ctx $\leftarrow$ XOF.Init()
2: ctx $\leftarrow$ XOF.Absorb(ctx, B)
3: $j \leftarrow 0$
4: **while** $j < 256$ **do**
5: (ctx, C) $\leftarrow$ XOF.Squeeze(ctx, 3)
6: $d_1 \leftarrow C[0] + 256 \cdot (C[1] \bmod 16)$
7: $d_2 \leftarrow \lfloor C[1]/16 \rfloor + 16 \cdot C[2]$
8: **if** $d_1 < q$ **then**
9: $\hat{a}[j] \leftarrow d_1$
10: $j \leftarrow j + 1$
11: **end if**

Algorithm 30 RejNTTPoly(ρ)

Samples a polynomial $\in T_q$.

Input: A seed $\rho \in \mathbb{B}^{34}$.
Output: An element $\hat{a} \in T_q$.

1: $j \leftarrow 0$
2: ctx $\leftarrow$ G.Init()
3: ctx $\leftarrow$ G.Absorb(ctx, ρ)
4: **while** $j < 256$ **do**
5: (ctx, s) $\leftarrow$ G.Squeeze(ctx, 3)
6: $\hat{a}[j] \leftarrow$ CoeffFromThreeBytes($s[0], s[1], s[2]$)
7: **if** $\hat{a}[j] \neq \perp$ **then**
8: $j \leftarrow j + 1$
9: **end if**
10: **end while**
11: **return** $\hat{a}$

SHA-3:
Both ML-DSA and ML-KEM make extensive use of SHA3

MLDSA.RejNTTPoly(ρ) / MLKEM.SampleNTT(ρ) :

But SHAKE is more efficient squeezing one large block than many small blocks, so let's squeeze in blocks.

- Sweet spot is
  504 (MLKEM) / 272 (MLDSA)
  to align with SHAKE block size boundary.
- Gives **20% speedup** for 504 bytes to ML-KEM keygen, encaps, decaps.
  (less for ML-DSA)

```
// 504 is exactly three SHAKE128 blocks (3 × 168).
let mut C :[u8;504] = [0u8; 504];
(&mut xof).squeeze_out(&mut C);
let mut idx: usize = 0;

// 4: while j < 256 do
while j < N {
    // 5: (ctx, C) ← XOF.Squeeze(ctx, 3)
    //   ▷ get a fresh 3-byte array C from XOF
    if idx == C.len() {
        (&mut xof).squeeze_out(&mut C);
        idx = 0;
    }
```

KeyGen

Idea: hold only intermediate seeds, derive vectors on-demand

KeyGen:
ML-KEM

Initial seed

Object	Size (bytes)
seed d	32

seed expand

Intermediate seeds

Object	Size (bytes)
seed ρ	32
seed σ	32

vector derive

Semi-expanded priv key

Object	Size (bytes)
Pub key	800 - 1567
VecK s	1024 - 2048

Algorithm 13 K-PKE.KeyGen(d)

1: $(\rho, \sigma) \leftarrow G(d\|k)$ ▷ expand 32+
2: $N \leftarrow 0$
3: **for** $(i \leftarrow 0; i < k; i$++$)$
4: **for** $(j \leftarrow 0; j < k; j$++$)$
5: $\hat{\mathbf{A}}[i, j] \leftarrow \text{SampleNTT}(\rho\|j\|i)$
6: **end for**
7: **end for**
8: **for** $(i \leftarrow 0; i < k; i$++$)$
9: $\mathbf{s}[i] \leftarrow \text{SamplePolyCBD}_{\eta_1}(\text{PRF}_{\eta_1}(\sigma, N))$
10: $N \leftarrow N + 1$
11: **end for**
12: **for** $(i \leftarrow 0; i < k; i$++$)$
13: $\mathbf{e}[i] \leftarrow \text{SamplePolyCBD}_{\eta_1}(\text{PRF}_{\eta_1}(\sigma, N))$
14: $N \leftarrow N + 1$
15: **end for**
16: $\hat{\mathbf{s}} \leftarrow \text{NTT}(\mathbf{s})$ ▷ ru
17: $\hat{\mathbf{e}} \leftarrow \text{NTT}(\mathbf{e})$
18: $\hat{\mathbf{t}} \leftarrow \hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}}$
19: $\text{ek}_{\text{PKE}} \leftarrow \text{ByteEncode}_{12}(\hat{\mathbf{t}})\|\rho$ ▷ ru
20: $\text{dk}_{\text{PKE}} \leftarrow \text{ByteEncode}_{12}(\hat{\mathbf{s}})$
21: **return** $(\text{ek}_{\text{PKE}}, \text{dk}_{\text{PKE}})$

KeyGen:
ML-DSA

Initial seed

Object	Size (bytes)
seed ξ	32

seed expand

Intermediate seeds

Object	Size (bytes)
seed ρ	32
seed ρ'	64
seek K	32

vector derive

Semi-expanded private key

Object	Size (bytes)
seeds ρ, K, tr	32 + 32 + 64
VecL s1	4096 - 8192
VecK s2	4096 - 7168
VecK t0	4096 - 7168

Algorithm 6 ML-DSA.KeyGen_internal(ξ)

1: $(\rho, \rho', K) \in \mathbb{B}^{32} \times \mathbb{B}^{64} \times \mathbb{B}^{32} \leftarrow \mathsf{H}(\xi||$
2:
3: $\hat{\mathbf{A}} \leftarrow \mathsf{ExpandA}(\rho)$ ▷ $\mathbf{A}$ i
4: $(\mathbf{s}_1, \mathbf{s}_2) \leftarrow \mathsf{ExpandS}(\rho')$
5: $\mathbf{t} \leftarrow \mathsf{NTT}^{-1}(\hat{\mathbf{A}} \circ \mathsf{NTT}(\mathbf{s}_1)) + \mathbf{s}_2$
6: $(\mathbf{t}_1, \mathbf{t}_0) \leftarrow \mathsf{Power2Round}(\mathbf{t})$
7: ▷ PowerTwoRound is applied c
8: $pk \leftarrow \mathsf{pkEncode}(\rho, \mathbf{t}_1)$
9: $tr \leftarrow \mathsf{H}(pk, 64)$
10: $sk \leftarrow \mathsf{skEncode}(\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$
11: **return** (pk, sk)

KeyGen_lowmemory:

Idea: hold only intermediate seeds, derive vectors on-demand

Key Object	PK size on disk	PK size in memory	SK Size on disk	SK size in memory
ML-KEM-512	800 (800)	800 (1056)	32 (1632)	161 (2178)
ML-KEM-768	1184 (1184)	1184 (1568)	32 (2400)	161 (3202)
ML-KEM-1024	1568 (1568)	1568 (2080)	32 (3168)	161 (4226)

Key Object	PK size on disk	PK size in memory	SK Size on disk	SK size in memory
ML-DSA-44	1312 (1312)	1312 (4128)	32 (2560)	176 (12464)
ML-DSA-65	1952 (1952)	1952 (6176)	32 (4032)	176 (17584)
ML-DSA-87	2592 (2592)	2592 (8224)	32 (4896)	176 (23728)

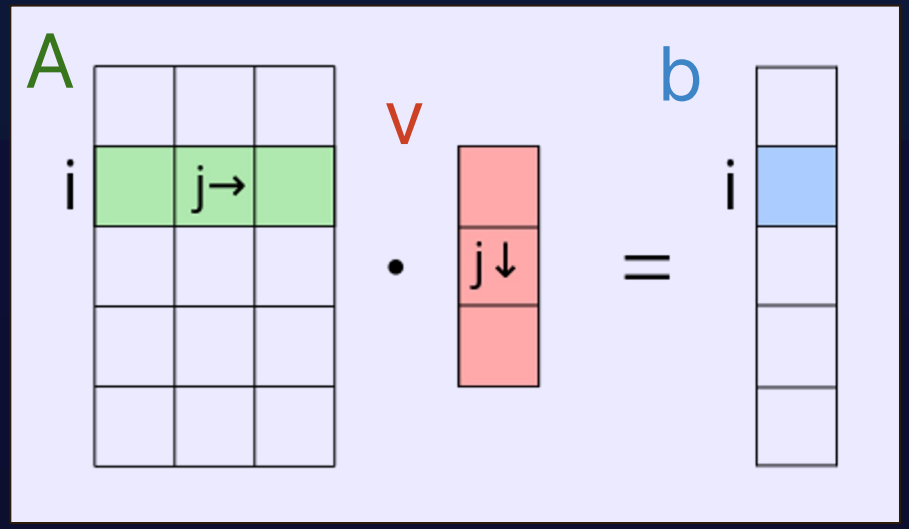
* values in (parentheses) are the non-optimized bc-rust implementation, for comparison.

On-demand matrix & vector expansion

Idea: you actually only need one entry at a time

On-demand matrix & vector expansion

Refresher: matrix multiplication



$$b_i = (A_{i,0} * v_0) + (A_{i,1} * v_1) + .. + (A_{i,n} * v_n)$$

Observation:

If we work one (parentheses term) at a time, then we only ever need to hold a single entry of $A_{i,j}$ and v_j at a time.

But what about b?
Don't you need to hold the entire b vec in memory?

Nope!

Turns out that ML-KEM and ML-DSA feed b into SHAKE one entry at a time!

On-demand matrix & vector expansion

Refresher: matrix multiplication

$$b_i = (A_{i,0} * v_0) + (A_{i,1} * v_1) + .. + (A_{i,n} * v_n)$$

With some clever re-arranging of the order of the algorithm steps, you can do the entire core operation in **3 registers**.

```
let shake = SHAKE::new();
for i in 0..l {
    let b_i[1] = Polynomial::new();
    for j in 0..k {
        let b_i += A.derive_from_seed(rho, i, j)[2] * v.derive_from_seed(rho_p, j)[3];
    }
    shake.absorb(b_i);
}
```

(simplified)

Actual size? Each register is a polynomial, so fundamental lowest memory footprint: 3072 bytes

ML-KEM: 3x [i16; 256] = 1536 bytes
ML-DSA: 3x [i32; 256] =

On-demand matrix & vector expansion

Other memory hygiene tricks (aka “good engineering”)

- Be careful to scope your variables into functions or un-named scopes so they get popped off the stack as soon as no longer needed.
  - Needs some careful re-arranging of the order of the algorithm steps so that all uses of a variable are clustered for easy scoping.
- Copies – the Rust compiler likes to make silent Copy’s of things.
  - Be careful impl’ing the Copy trait on core objects.
  - Pass-by-value vs pass-by-reference
- Working in-place, rather than in a newly allocated variable.
- Heap vs stack.

On-demand matrix & vector expansion

Results!

Total speed-vs-memory tradeoff: up to ~ 6x speed ⬇ for ~ 8x memory ⬇

Algorithm	Peak stack memory usage (kB)	Throughput (Kops/s)
MLKEM512_lowmemory/Encaps	9.3 (14.4)	39.4 (47.2)
MLKEM768_lowmemory/Encaps	11.0 (24.3)	23.2 (28.6)
MLKEM1024_lowmemory/Encaps	13.3 (35.0)	15.2 (19.6)
MLKEM512_lowmemory/Decaps	11.9 (21.6)	13.8 (36.8)
MLKEM768_lowmemory/Decaps	13.6 (33.4)	8.1 (22.8)
MLKEM1024_lowmemory/Decaps	15.8 (49.2)	5.2 (16.1)

Algorithm	Peak stack memory usage (kB)	Throughput (ops/s)
MLDSA44_lowmemory/Sign	22.3 (86.5)	990 (3,400)
MLDSA65_lowmemory/Sign	27.9 (128.0)	490 (2,400)
MLDSA87_lowmemory/Sign	31.6 (187.3)	330 (1,700)
MLDSA44_lowmemory/Verify	15.2 (56.4)	11,200 (13,200)
MLDSA65_lowmemory/Verify	19.1 (86.2)	6,400 (8,000)
MLDSA87_lowmemory/Verify	22.3 (130.0)	3,700 (4,800)

* values in (parentheses) are the non-optimized bc-rust implementation, for comparison.

Take with a grain of salt:
* stack measurements this small depend heavily on compiler optimization relative to the calling function.
* over-measurements by ~ 3 kb because it includes main() and print!().
* Throughput measured on my laptop, not scientific, meant for relative comparison only.

Going the other way

Pre-expanding the public matrix

Going the other way

Pre-expanding the public matrix

So far we've dealt with the public key in its compressed (NIST) encoding where the public matrix A is represented by its 32-byte seed ρ.

What if we pre-expand the public matrix A on key load; ie outside the sign() / verify() / encaps() / decaps()?

It's not a free lunch – you still have to expand it – but:

- Do it in another thread while loading content off disk / network.
- Store the fully expanded key in a secure boot verifier.
- Batch verification; ex.: multiple PGP-signed packages from the same code-signer.

* Note: no effect on *peak* memory usage, since the un-optimized implementation starts by expanding A anyway, but it does make the *key objects* significantly larger.

Going the other way

Pre-expanding the public matrix

Key Object	PK size on disk	PK size in memory	SK Size on disk	SK size in memory
ML-KEM-512_expanded	800	3104	1632	4226
ML-KEM-768_expanded	1184	6176	2400	7810
ML-KEM-1024_expanded	1568	10272	3168	12418

Key Object	encaps()	decaps()
ML-KEM-512_expanded	x1.3	x1.2
ML-KEM-768_expanded	x1.5	x1.4
ML-KEM-1024_expanded	x1.6	x1.5

Key Object	PK size on disk	PK size in memory	SK Size on disk	SK size in memory
ML-DSA-44_expanded	1312	20512	2560	28848
ML-DSA-65_expanded	1952	36896	4032	48304
ML-DSA-87_expanded	2592	65568	4896	81072

Key Object	sign()	verify()
ML-DSA-44_expanded	x1.13	x1.9
ML-DSA-65_expanded	x1.2	x2.2
ML-DSA-87_expanded	x1.28	x2.5

Conclusion

You can have either low memory or speed, but not both!

Conclusion

You can have either low memory or speed, but not both!



Seed-based private keys, on-demand expansion

ML-KEM: ~ 8 - 16 kb

ML-DSA: ~ 12 - 32 kb

Trade ~ 6x speed

for ~ 8x memory

Un-optimized implementation

ML-KEM: ~ 13 - 63 kb

ML-DSA: ~ 70 - 200 kb

Pre-expanded public keys

ML-KEM: ~ 13 - 63 kb

ML-DSA: ~ 70 - 200 kb

(same peak mem usage)

Keys ~ 8x larger

for ~ 1.4 - 2.5x speed