

Engineering Post-Quantum Cryptography for Embedded and Resource-Constrained Systems

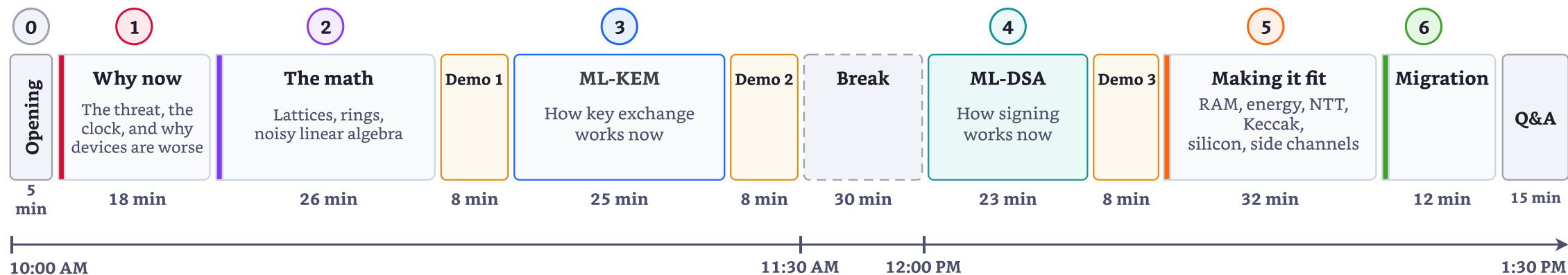
A hands-on tutorial on ML-KEM and ML-DSA, from lattices to silicon, for parts measured in kilobytes

Bardia Taghavi, Cryptographic Engineer · PQSecure Technologies
ewNA26 · Embedded World North America 2026 · 3-hour tutorial

The plan, to the minute

Three hours, one break, three demos

180 minutes exactly. About ninety seconds a slide, so there is room to stop and argue. The break falls at 1:30, when ML-KEM is finished.



What you get, and who this is for

What you walk out with

- ▶ What ML-KEM and ML-DSA actually compute, and why.
- ▶ The numbers: cycles, bytes, kilobytes of RAM, millijoules, gates.
- ▶ A checklist for your next design review.

What I am assuming

- ▶ You know what a hash and a block cipher do.
- ▶ You can read a block diagram and a RAM budget.
- ▶ You need no lattice theory and no quantum mechanics.
- ▶ Every term is defined before it is used.

What this is not

- ▶ Not a cloud talk. The constraint here is kilobytes, milliwatts and a ten-year field life.
- ▶ No proofs.
- ▶ No claim that this is easy, or that you can skip it.

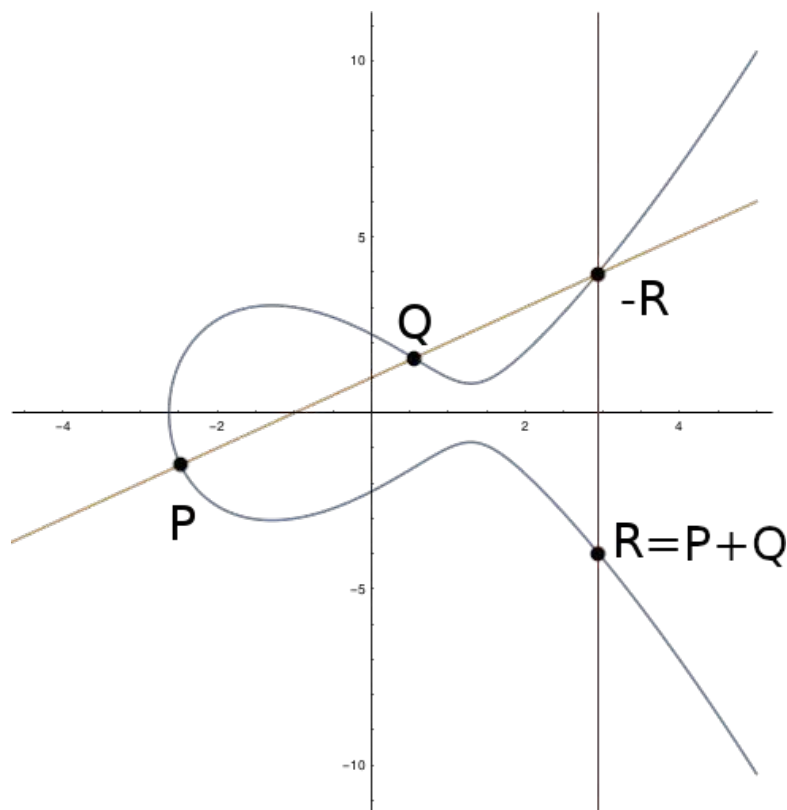
Stop me the moment a slide loses you. A room of confused engineers is a wasted three hours.

PART 01

Why this matters now

The threat, the standards, and why a ten-year product is the hard case

Everything public-key rests on two problems



Factoring

Multiplying two 1024-bit primes is instant. Recovering them from the product is not. That is RSA, the best way to recover m from $m^e \bmod n$.

Discrete logarithms

Computing $g^a \bmod q$ is instant. Recovering a is not. That is Diffie-Hellman, ECDH, ECDSA and EdDSA.

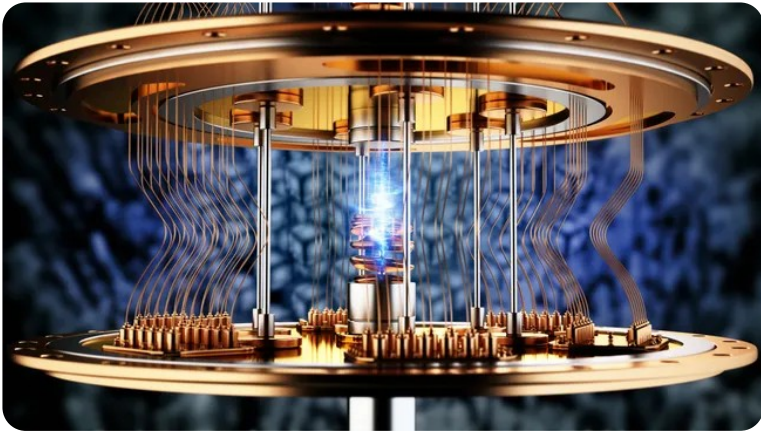
The uncomfortable part

Almost the whole public-key layer of the internet, and of your product, rests on **two** problems. They are close cousins: both are hidden-period problems in a finite group.

1994

Peter Shor showed that a quantum computer solves **both** in **polynomial time $O(n^3)$** .

The machine, and the gap that decides everything



Most of the picture is a refrigerator

The processor is the small piece at the bottom, at about ten millikelvin. A qubit is a two-state system you can put in a combination of both states at once; that is as much physics as this session needs.

Today's machines

Roughly a thousand physical qubits, with error rates far too high to run Shor's algorithm on a real key.

The gap is the whole question

Between a thousand noisy qubits and a million with error correction. Nobody credible will give you a date.

Which is why the date is not the point

The lifetime of the thing you are designing is.

What breaks, and what only bends

Shor's algorithm (1994)

Factoring and discrete logs go from “centuries” to “an afternoon”. Exponential speed-up, so bigger keys do not save you.

RSA

factoring
broken

Diffie-Hellman

discrete log
broken

ECDH / ECDSA

elliptic-curve log
broken

Grover's algorithm (1996)

Brute-force search gets a square-root speed-up: 2^n tries become $2^{n/2}$. Only a quadratic gain, so doubling the key fixes it.

AES-128

$2^{128} \rightarrow 2^{64}$

AES-256

$2^{256} \rightarrow 2^{128}$

SHA-2 / SHA-3

use longer digests

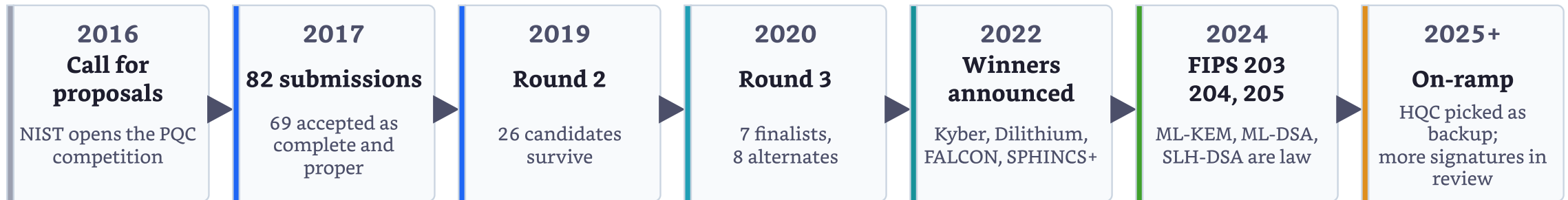
Read the second column before you panic

Algorithm	What it does for you	Quantum impact	What to do
AES-256	Symmetric encryption	Grover only	Keep it
SHA-384 / SHA-512	Hashing, HMAC, KDF	Grover only	Keep it
SHA-256	Hashing	Grover only	Fine for most uses; CNSA 2.0 wants 384+
AES-128	Symmetric encryption	Grover weakens it	Move to AES-256
RSA-2048 / 4096	Key transport, signatures	Shor breaks it	Replace
ECDH / X25519	Key agreement	Shor breaks it	Replace
ECDSA / EdDSA	Signatures	Shor breaks it	Replace

Grover's square-root speed-up is fixed by using a longer symmetric key. Shor's exponential speed-up is not fixed by any key length: RSA-4096 is only about eight times harder than RSA-2048 for a quantum attacker.

Eight years of public cryptanalysis

Eight years of public cryptanalysis, in one line



Why this matters to you: these are not new ideas rushed into a standard. Lattice cryptography has been studied since the 1990s, and these exact parameter sets were attacked in public for eight years before FIPS 203 and 204 were signed.

And two schemes did fall during the process. Rainbow and SIKE were broken publicly, which is exactly what an open competition is for. Neither was lattice-based.

Five families, and the four standards that came out of them

Lattice-based	Hash-based	Code-based	Isogeny-based	Multivariate
ML-KEM, ML-DSA FN-DSA	SLH-DSA LMS, XMSS	HQC Classic McEliece	SQIsign	MAYO, UOV SNOVA
Balanced: keys and signatures of a few kilobytes, fast in software and small in hardware.	Security rests only on a hash function. Big signatures, or state you must never reuse.	Very old, very studied. Huge public keys (up to a megabyte) but tiny ciphertexts.	Smallest keys of all. SIKE was broken in 2022; the survivors are slow and still young.	Fast verification, small signatures. Rainbow was broken in 2022, which cost trust.
The default choice	Best for firmware signing	Backup and niche use	Watch, do not ship	Under review

Different maths, same purpose: a problem Shor cannot touch

What they all share

None of them reduce to factoring or discrete logs, so Shor's algorithm has no handle. Each rests on a problem that stays hard even with a quantum computer, as far as anyone knows.

Why we hedge

"As far as anyone knows" is the whole caveat. That is why NIST kept a code-based backup, why hybrid modes exist, and why your design should let the algorithm be swapped.

For an embedded engineer the practical shortlist is short: ML-KEM for key exchange, ML-DSA for general signing, and LMS or XMSS when the verifier lives in ROM and must stay tiny.

Eighty-two submissions, eight years of public cryptanalysis, everything in the open

Five families, and the four standards that came out of them



Parameter sets and what they are worth

Parameter set	NIST category	Comparable to	Public key	Private key	Ciphertext / signature
ML-KEM-512	Category 1	AES-128	800 B	1632 B	768 B
ML-KEM-768	Category 3	AES-192	1184 B	2400 B	1088 B
ML-KEM-1024	Category 5	AES-256	1568 B	3168 B	1568 B
ML-DSA-44	Category 2	SHA-256 collision	1312 B	2560 B	2420 B
ML-DSA-65	Category 3	AES-192	1952 B	4032 B	3309 B
ML-DSA-87	Category 5	AES-256	2592 B	4896 B	4627 B

If you only remember one line: ML-KEM-768 and ML-DSA-65 are the sensible defaults for general use. NSA's CNSA 2.0 asks for ML-KEM-1024 and ML-DSA-87 in national security systems.

Final standards, not drafts. Lattices won the general case because they are small enough and fast enough at the same time

Four jobs, four deadlines

1

Agree on a key

TLS, DTLS, MQTT, VPN.
Two parties end up with
the same secret without
ever sending it.

Today
ECDH, RSA-KEM

After migration
ML-KEM

2

Prove who signed

Firmware images, OTA
bundles, certificates,
attestation reports.

Today
ECDSA, RSA-PSS

After migration
ML-DSA, LMS/XMSS

3

Prove who you are

Device identity, mutual
authentication, secure
onboarding.

Today
ECDSA certs

After migration
ML-DSA certs

4

Anchor the boot

ROM checks stage 1,
stage 1 checks stage 2.
One key, burned once.

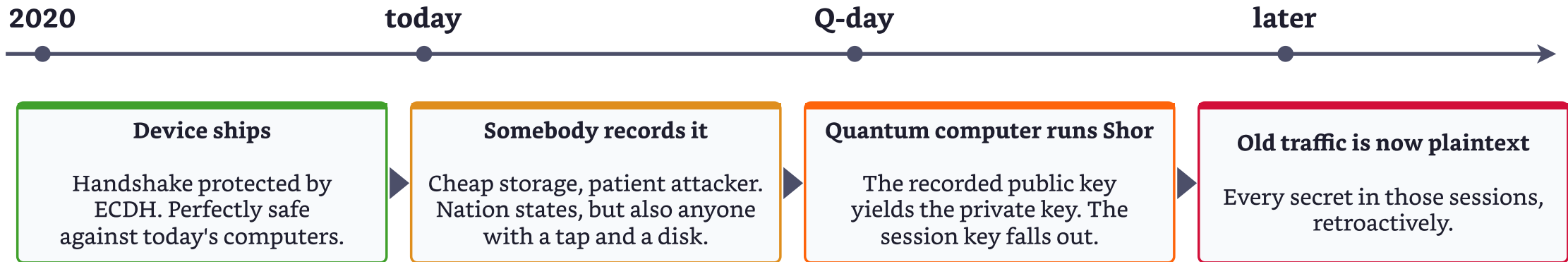
Today
RSA-2048/ECDSA

After migration
LMS or ML-DSA

Only the public-key layer breaks. AES-256, SHA-2/SHA-3 and HMAC stay exactly as they are. Every one of the four jobs above has to change, and each one has a different deadline.

Harvest now, decrypt later

Traffic recorded today can be opened the day a quantum computer arrives. Nothing about your device changes; the recording does the work.



The uncomfortable part: the clock starts when the data is captured, not when the quantum computer is built. Anything you send today that must stay secret for ten years is already exposed.

Signatures are different

A forged signature is only useful in the future, so signing can migrate a little later unless the key is in ROM.

Key exchange is urgent

Confidentiality is retroactive. This is the one to fix first, and hybrid mode lets you fix it without betting on new maths.

Three numbers that set your deadline

2030

CNSA 2.0: new firmware
signing must already be post-
quantum

2033

CNSA 2.0: every listed system
post-quantum by default

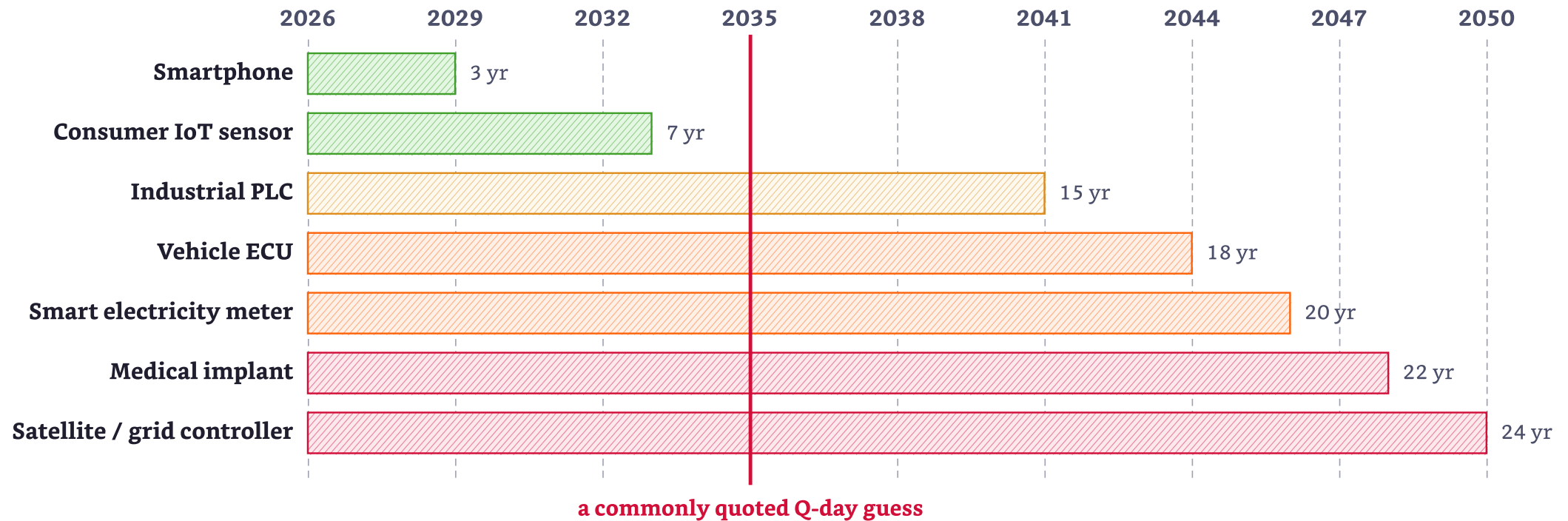
10 to 20

Years that a typical embedded
product stays in the field after it
ships

Subtract the third number from the first two. That is how much design time you actually have.

A server is replaced. Your product is not.

A phone is replaced in three years. The things at this conference are not.



The point: anything designed in 2026 with a 15-year life will still be running when the assumption behind its cryptography fails. If the root key or the verify algorithm is fixed in ROM, that device cannot be rescued by a software update.

What a resource-constrained part actually has

Peak stack for one operation, from published Cortex-M figures. Part 5 has the detail

Class	Typical part	RAM	ML-KEM-768	ML-DSA-65 verify	ML-DSA-65 sign
Deeply embedded	Cortex-M0+, M23, small RV32	8 to 32 kB	~3 kB, usually fits	~8 to 16 kB (needs 16 kB+ RAM)	~10 kB at best, often impossible
Secure element	smartcard core with an accelerator	16 to 64 kB	fits	fits	fits, but die area decides
Mainstream MCU	Cortex-M4, M33, RV32IMC	64 to 512 kB	comfortable	comfortable	comfortable
Application MPU	Cortex-A, RV64	256 MB and up	free	free	free

The one line to take away: **ML-KEM fits almost everywhere. ML-DSA signing is the operation that will not fit** - and for firmware you do not need it on the device, only verification.

The difference between this session and a cloud talk

A server

- ▶ Replaced every three to five years
- ▶ Gigabytes of RAM, gigahertz of clock
- ▶ Library update, restart, done
- ▶ Mains power and a fat link
- ▶ Physically inside a locked building

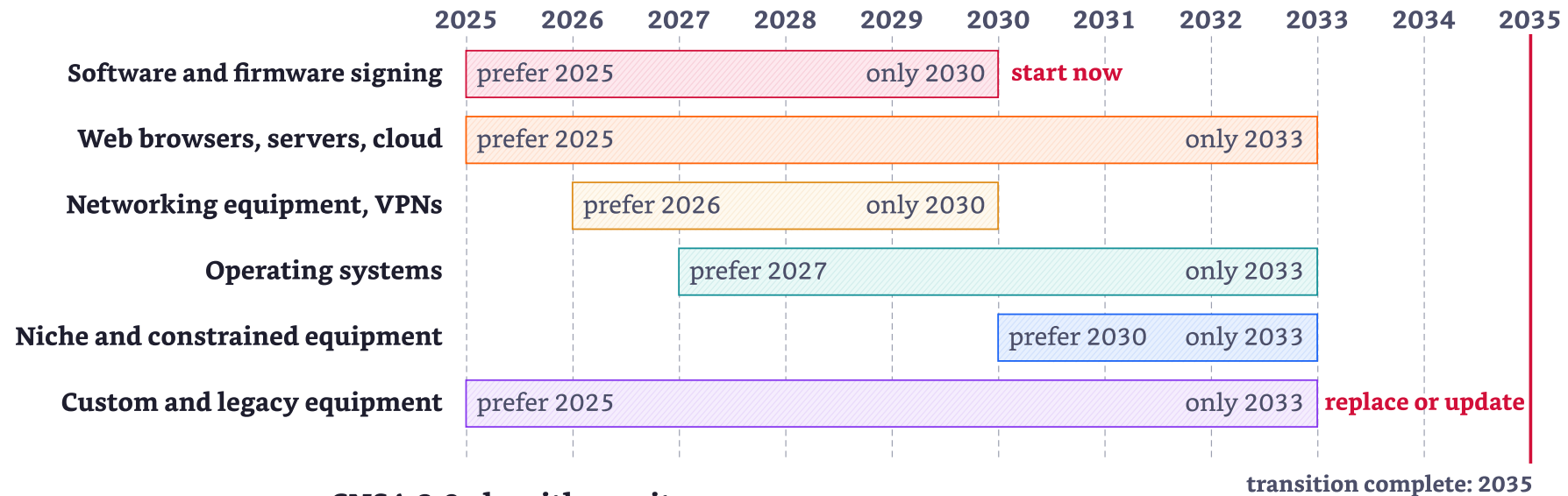
The things at this conference

- ▶ In the field for ten to twenty years
- ▶ Kilobytes of RAM, tens of megahertz
- ▶ Root of trust burned into ROM or fuses
- ▶ A coin cell and a 64-byte frame
- ▶ In the attacker's hand, on a bench, with a probe

For a server this is a software project. For a constrained product it is a silicon project, and silicon has a longer lead time than any standard.

CNSA 2.0: the only schedule with real dates

NSA's timeline for US national security systems. Not binding on your product, but it is what your customers' procurement teams will quote at you.



CNSA 2.0 algorithm suite

Role	Algorithm	Standard
Key establishment	ML-KEM-1024	FIPS 203
Signatures, general	ML-DSA-87	FIPS 204
Signatures, firmware	LMS or XMSS (SHA-256/192)	SP 800-208
Symmetric	AES-256	FIPS 197
Hashing	SHA-384 or SHA-512	FIPS 180-4 / 202

Read the first row again

Firmware signing has the earliest deadline and the longest lead time, because the verifying key often sits in mask ROM or one-time-programmable fuses.

That is a silicon decision, not a software one.

The full reasoning is in the CNSA 2.0 FAQ, published May 2025. Other governments have published roadmaps with the same shape and later dates.

Part 1 in five lines

Only the public-key layer breaks

AES-256, SHA-2 and SHA-3 are fine.

Key exchange is the urgent one

Recorded traffic can be opened later. Signatures can wait a little, unless the verifying key lives in ROM, and it usually does.

The standards are final, studied, and shipping

Real libraries have them today.

ML-KEM fits your part. ML-DSA signing may not

That single asymmetry drives most of the design decisions in parts 5 and 6.

You cannot predict Q-day

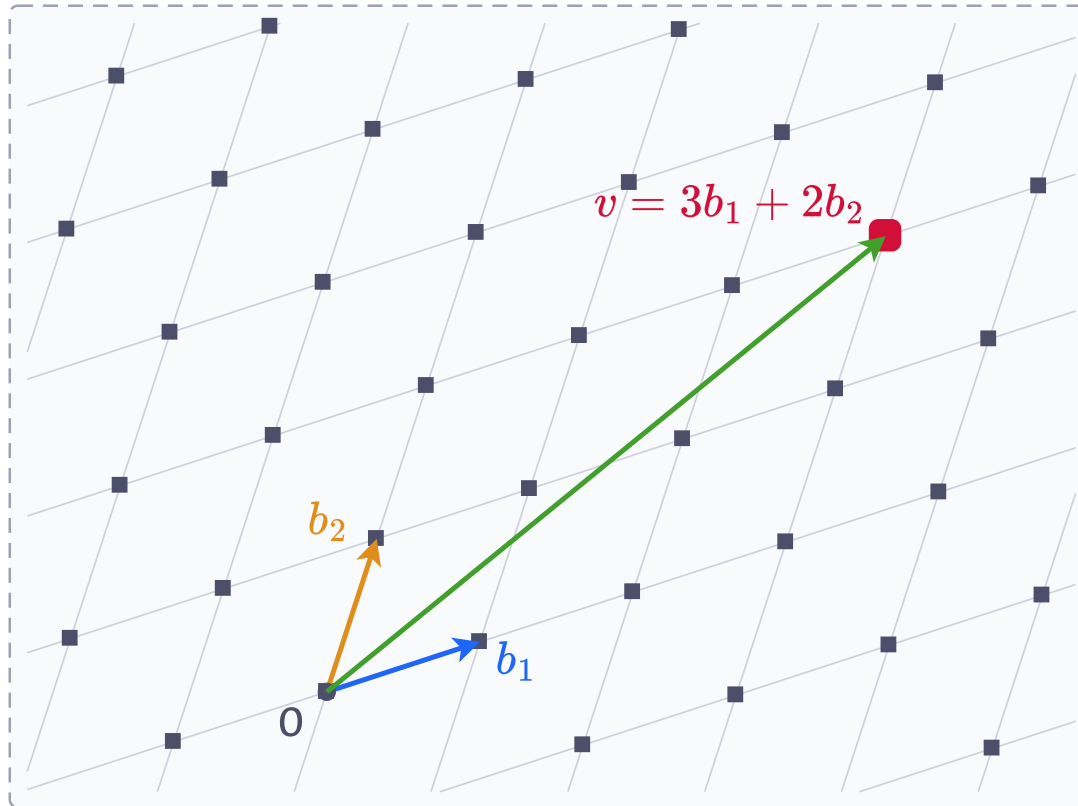
You can shrink how long your migration takes. Do that instead.

PART 02

The mathematics, minus the mystique

From a grid of points to the arithmetic your silicon has to do

A lattice is a grid you build from vectors



The rule

Pick two vectors. A lattice point is any **whole-number** combination of them.

$3b_1 + 2b_2$ is a lattice point.
 $2.5b_1 + 2b_2$ is not.

That word “whole-number” is where all the difficulty comes from.

Why cryptographers like it

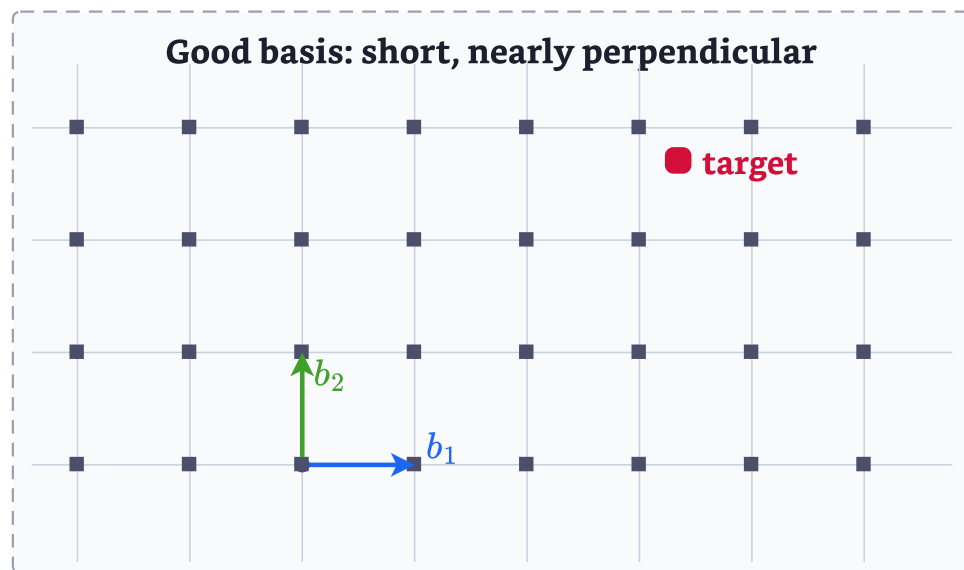
Given the basis, walking to a lattice point is trivial: multiply and add.

However, adding a little noise to an arbitrary point, and finding which grid point it came from becomes impossible once the dimension is high enough.

ML-KEM works in 512 to 1024 dimensions.
ML-DSA works in 1024 to 2048.

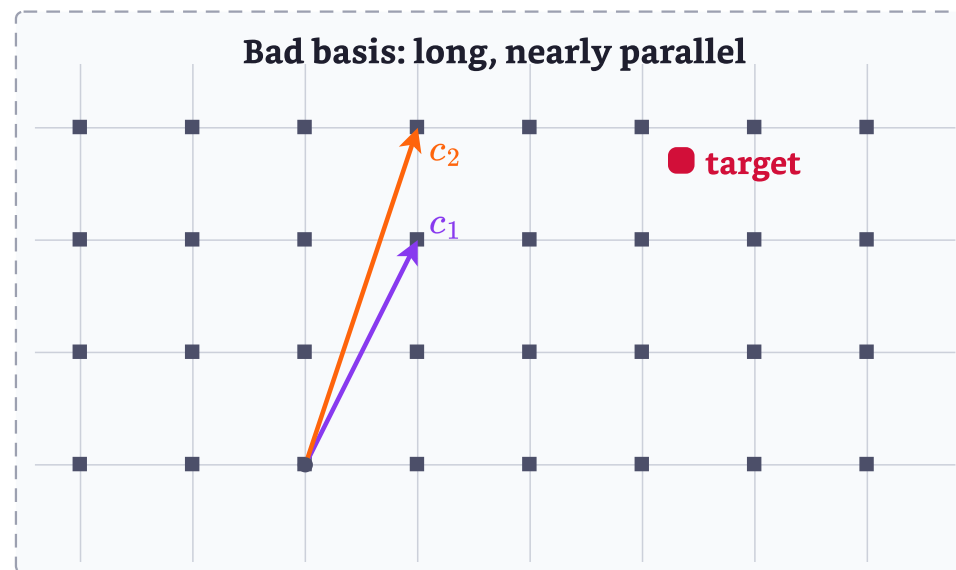
The one-way street: easy forwards, believed hard in reverse. Unlike factoring, no quantum algorithm is known that changes that. Finding the shortest or the closest vector is NP-hard in the worst case.

The same lattice, two bases



Coordinates in b_1, b_2 rounded: gives $3b_1 + 3b_2$.
Exact closest point found immediately.

Good basis: Short vectors $\rightarrow$ CVP solved by rounding



Coordinates in c_1, c_2 rounded: gives $7c_1 - 4c_2$.
Wrong point; true search takes exponential time.

Bad basis: Long/skewed vectors $\rightarrow$ CVP is hard

Falcon uses a short basis as its trapdoor. ML-KEM and ML-DSA use short secret vectors via Module-LWE.

Finding the nearest lattice point to an arbitrary point is the Closest Vector Problem. It is easy with a good basis, believed hard with a bad one in five hundred dimensions, and no quantum algorithm is known that helps.

Why plain CVP fails in silicon

Plain CVP on general lattices (Unstructured vector space)

High dimension needed ($n \geq 1000$)

Low dimensions are broken by BKZ/LLL.
Quantum security requires $n \geq 800-1000$.

10^6 matrix entries

- Key size: **2 to 4 MB**
- Multiplication: **1,000,000 operations** ($O(n^2)$)

Fails on embedded silicon

- 2 MB exhausts 32 kB MCU SRAM
- Needs 30,000+ CAN FD frames

Replace
matrix with
256-entry
poly



The polynomial fix (NIST standards) (ML-KEM and ML-DSA)

One polynomial replaces the matrix

Store **one array of 256 numbers** instead of 10^6 entries.
Rotation generates all rows algebraically.

Down to 384 bytes

- $256 \times 12 \text{ bits} = \mathbf{384 \text{ bytes}}$
- Keys fit in **1 network packet** (800 B–1.5 kB)
- Multiply in $O(n \log n)$ via NTT

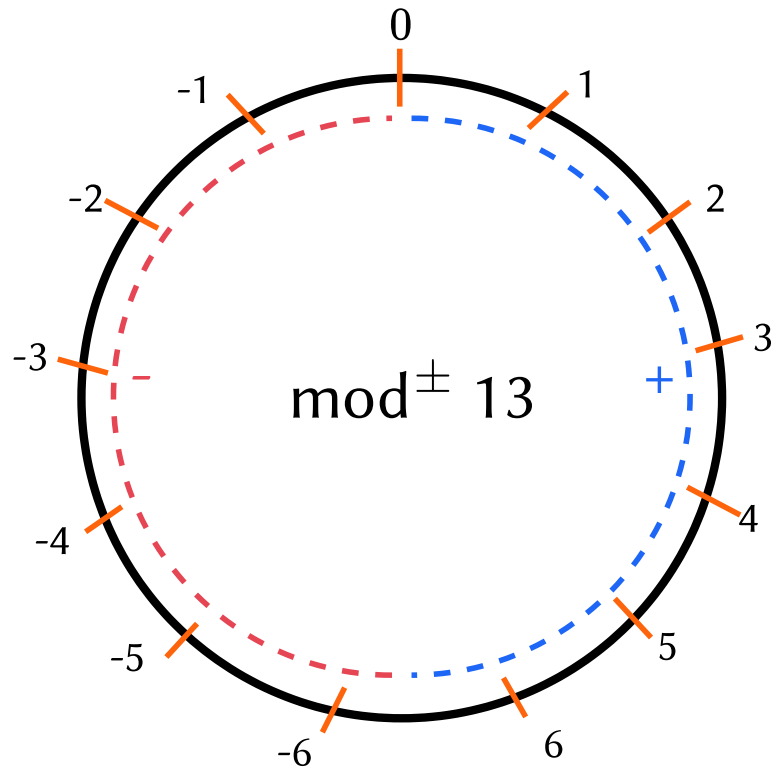
Silicon-ready

- Fits in small on-chip SRAM
- No packet fragmentation over serial/radio

Bridge to next slide: Instead of huge grids, polynomials hide secrets with **small noise modulo q** (on a clock face).

General lattices require megabytes of keys and quadratic runtime. Polynomials compress the matrix to 384 bytes.

Small means close to zero, in either direction



Every number lives on a clock face

Arithmetic modulo q , which everyone here already knows from a counter that wraps.

Same bits, different reading

Write a mod-plus-minus q for the representative between minus $q/2$ and $q/2$. Both standards use this whenever they say a value is small.

The size of a polynomial

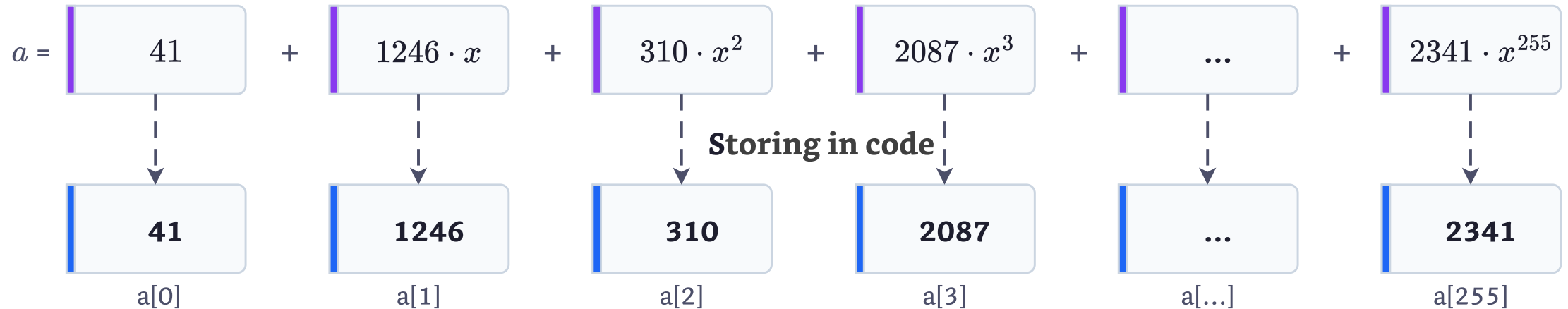
The infinity norm is the largest centred coefficient. With $q = 13$, the norm of $10x^2 + 4x + 1$ is 4, because 10 reads as minus 3.

In hardware

Testing $a > q/2$ is one comparison. Nothing needs to be re-encoded.

A polynomial is an array of 256 numbers

How a mathematician writes it



Addition is boring, and that is good

Add the arrays elementwise, reduce each one mod q .
256 independent adds. Perfectly parallel, one cycle per lane, no carries between coefficients.

Multiplication is the whole problem

Every coefficient of the answer touches every input coefficient: $256 \times 256 = 65\,536$ multiplies, done the obvious way. This is what the NTT exists to fix.

One polynomial in ML-KEM is 256×12 bits = 384 bytes. In ML-DSA it is 256×23 bits, packed into = 768 bytes.

Two rules that keep everything a fixed size

$$\mathcal{R}_q = \mathbb{Z}_q[x] / (x^n + 1)$$

Rule 1: $\mathbb{Z}_q$, coefficients wrap at q

Every coefficient is reduced mod q , so it always fits the same word width.

ML-KEM: $q = 3329 \rightarrow 12$ bits per coefficient
ML-DSA: $q = 8380417 \rightarrow 23$ bits per coefficient

Rule 2: $x^n + 1$, degrees wrap at n

Anything of degree n or higher folds back down, with a sign flip: $x^n = -1$.

Both standards use $n = 256$, always. Nothing ever grows.

Why an engineer should care: the ring is a promise that every intermediate value fits in a fixed-width array. No dynamic allocation, no growing buffers, no reallocation, so you can size every RAM at design time.

Reduce coefficients mod q , reduce exponents mod $x^{256} + 1$. Fixed size forever, which is what hardware needs.

Two rules that keep everything a fixed size

$$a = 1 + 2x + 3x^2 + 4x^3$$

$$b = 5 + 6x + 7x^2 + 8x^3$$

1. Multiply the ordinary way, and the degree grows to 6

$$5x^0$$

$$16x^1$$

$$34x^2$$

$$60x^3$$

$$61x^4$$

$$52x^5$$

$$32x^6$$

2. Fold the tail back with $x^4 = -1$

$$x^4 \rightarrow -1$$

$$-61 \text{ lands on } x^0$$

$$x^5 \rightarrow -x$$

$$-52 \text{ lands on } x^1$$

$$x^6 \rightarrow -x^2$$

$$-32 \text{ lands on } x^2$$

3. Collect, then reduce mod 17

$$x^0 : 5 - 61 = -56$$

$$12$$

$$x^1 : 16 - 52 = -36$$

$$15$$

$$x^2 : 34 - 32 = 2$$

$$2$$

$$x^3 : 60$$

$$9$$

$$a \cdot b = 12 + 15x + 2x^2 + 9x^3 \text{ in } \mathbb{Z}_{17}[x]/(x^4 + 1)$$

Same shape at $n = 256$. Sixteen multiplies here; 65 536 there. *Which is exactly why nobody does it this way.*

The wrap costs a sign flip. Walk one coefficient around it: this is the one piece of the ring that surprises people.

One polynomial multiply, two ways to pay

Schoolbook: the obvious way

Every output coefficient is a sum over every input pair, with a sign flip on the wrap-around. $n^2 = 65\,536$ multiplies for $n = 256$.

n^2 multiplies, n^2 adds, no extra memory

Still the right answer for very small n , or when you have no memory to spare.

NTT: change coordinates first



Two forward transforms, n multiplies in the middle, one inverse transform.

$3 \cdot (n/2) \log_2 n + n \approx 3\,328$ multiplies

And the forward transforms of a re-used polynomials can be cached, which cuts it further.

The gap, at the sizes the standards actually use

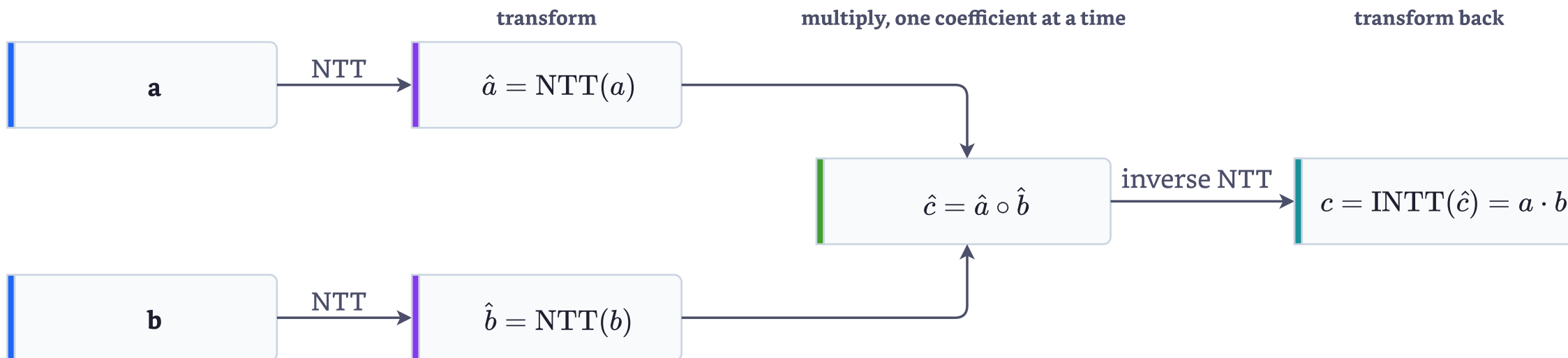
Ring size	Schoolbook multiplies	NTT multiplies	Ratio
$n = 8$	64	60	$\sim 1\times$
$n = 64$	4 096	640	$6\times$
$n = 256$	65 536	3 328	$20\times$
$n = 1024$	1 048 576	16 384	$64\times$

At $n = 256$ the NTT is worth about a factor of twenty. That factor is why every published PQC accelerator has an NTT unit in it.

Schoolbook convolution needs 65,536 modular multiplies and adds. The NTT changes coordinates, multiplies pointwise in 256 ops, and transforms back: $\sim 3,000$ operations total.

The NTT is a change of coordinates

Same trick as the FFT, but over integers modulo q , so everything stays exact. No rounding, no floats.



Why this is legal

Convolution in one domain is elementwise multiply in the other. Picking ζ with $\zeta^n = -1 \bmod q$ makes the transform match the ring's own wrap-around, so the sign flip comes for free.

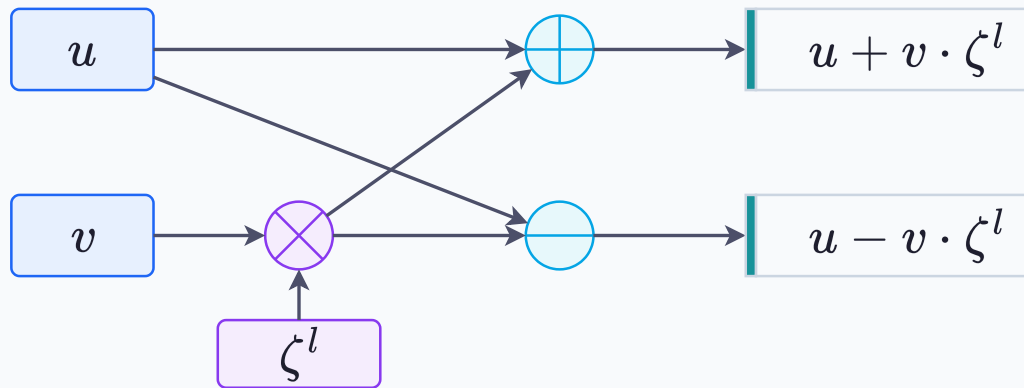
Why hardware likes it

The transform is $\log_2 n$ stages of one identical operation, a butterfly. One small datapath, run $n/2$ times per stage, does the whole job.

Transform, multiply pointwise, transform back: exactly the same trick as an FFT, in a finite field. Exact integer arithmetic with no rounding and no floats.

The butterfly: the only arithmetic block you need

Cooley-Tukey: forward NTT

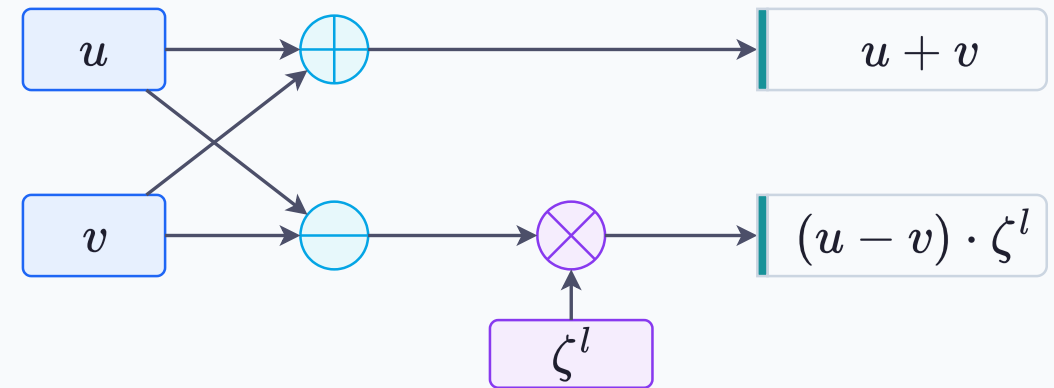


What it costs in gates

One modular multiplier, one modular adder, one modular subtracter. For ML-DSA that is a 23×23 multiply; for ML-KEM a 12×12 .

Everything else in an NTT core is memory and control.

Gentleman-Sande: inverse NTT



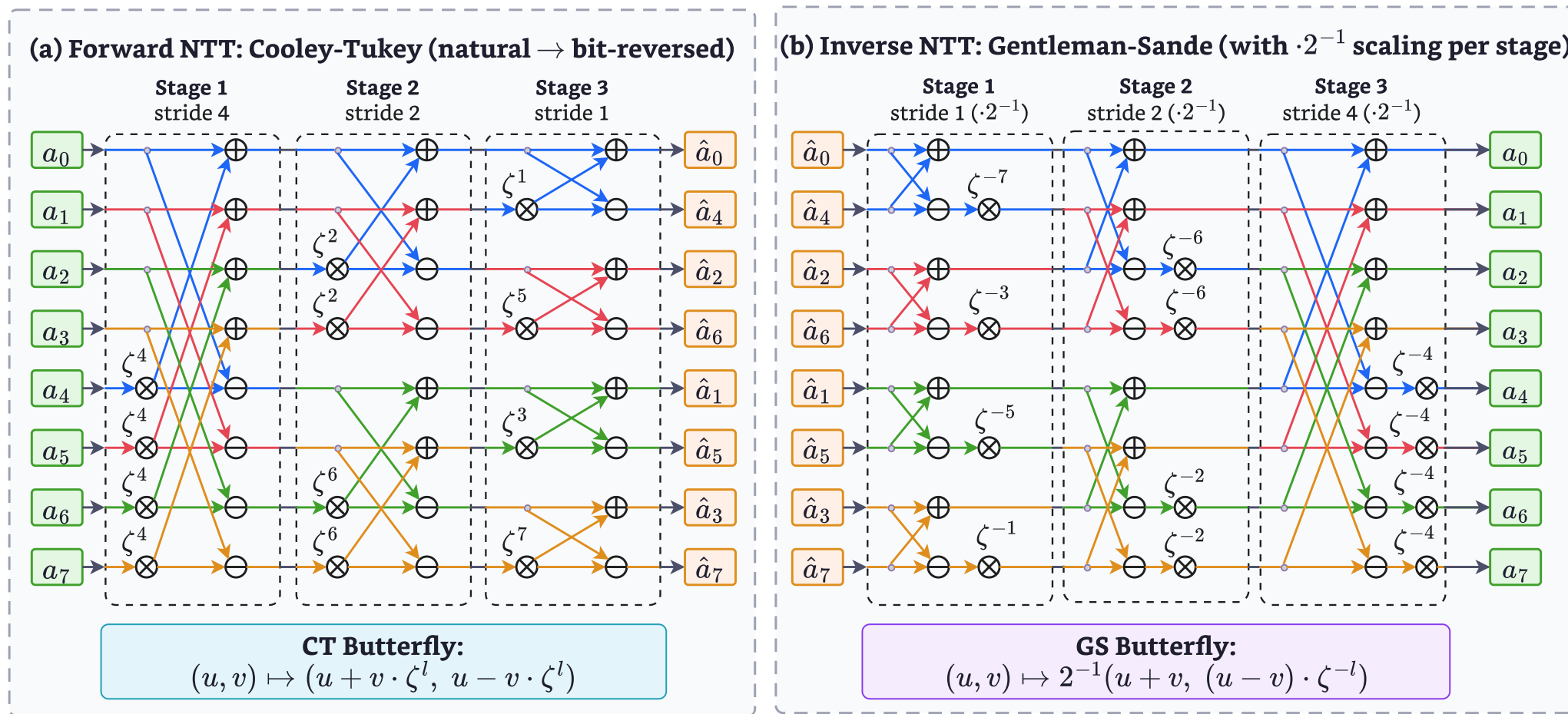
Why there are two flavours

Cooley-Tukey multiplies before the add and subtract; Gentleman-Sande multiplies after. Using CT forward and GS inverse means neither direction needs a bit-reversal pass, because the orders cancel out.

One datapath plus two multiplexers serves both.

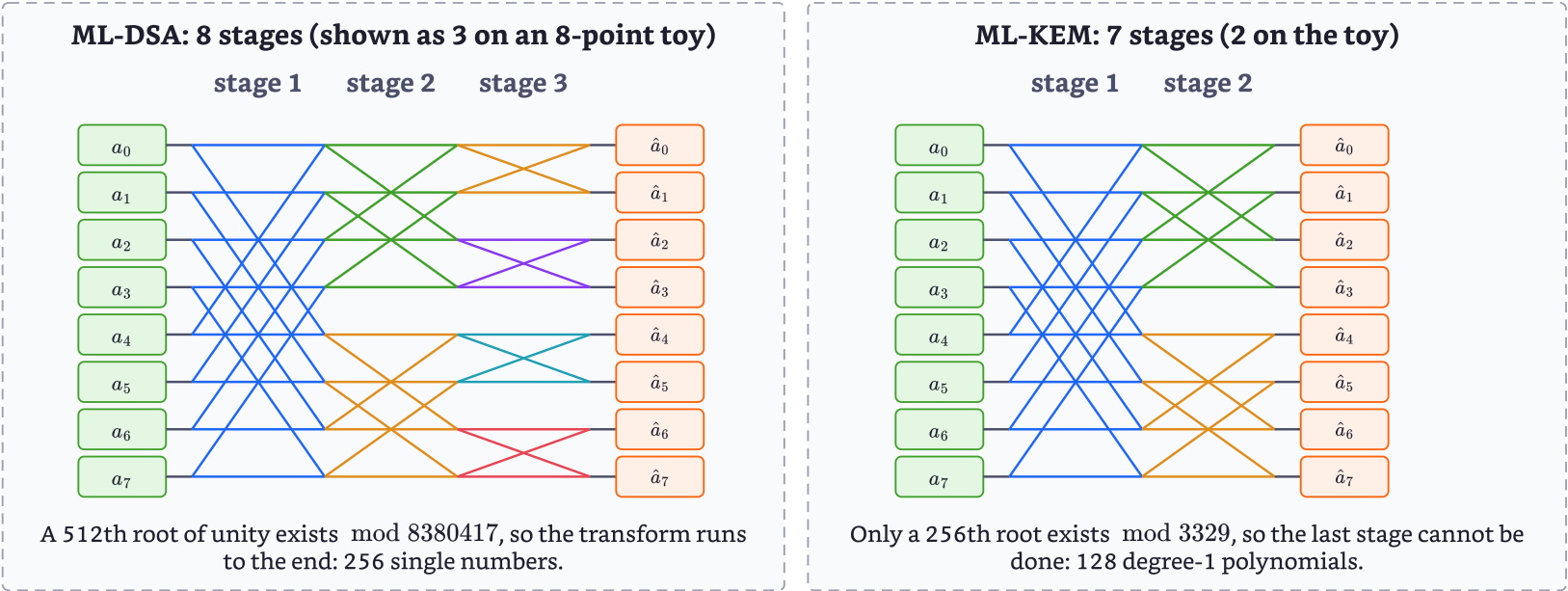
Cooley-Tukey for forward transform, Gentleman-Sande for inverse. One multiply, one add, one subtract. That is the entire datapath.

A complete 8-point transform



Three stages, twelve butterflies, and twiddle powers: (a) Cooley-Tukey forward transform (natural to bit-reversed); (b) Gentleman-Sande inverse transform (bit-reversed to natural) with $1/2$ scaling after each addition/subtraction.

Why ML-KEM's NTT stops one stage early



The consequence lands in the pointwise multiply

	ML-DSA (complete NTT)	ML-KEM (incomplete NTT)
What one NTT output is	a single number mod q	a pair $(c_0 + c_1x)$
Pointwise multiply	1 modular multiply per coefficient	5 multiplies per 2 coefficients
Extra constants needed	none	$\zeta^{2\text{brv}(i)+1}$, 128 of them
Stages	8 forward, 8 inverse	7 forward, 7 inverse
Effect on a shared core	n/a	control logic differs, arithmetic does not

ML-DSA runs all 8 stages down to degree-0 scalars. ML-KEM stops at 7 stages, producing 128 degree-1 polynomials, because $q=3329$ has no 512th root.

Pointwise multiply: where the two schemes really differ

ML-DSA: one multiply, done

$$\hat{c}_i = \hat{a}_i \cdot \hat{b}_i \bmod q$$

256 independent modular multiplies. Perfectly parallel: give it eight multipliers and it finishes in 32 cycles.

Nothing to store, nothing to look up.

ML-KEM: multiply degree-1 polynomials

$$\hat{c}_{2i} = \hat{a}_{2i}\hat{b}_{2i} + \hat{a}_{2i+1}\hat{b}_{2i+1} \cdot \zeta^{2\text{brv}(i)+1}$$

$$\hat{c}_{2i+1} = \hat{a}_{2i}\hat{b}_{2i+1} + \hat{a}_{2i+1}\hat{b}_{2i}$$

Five multiplies for every two output coefficients
Four products plus the twiddle. Karatsuba gets it to four, at the cost of an extra add.

And 128 extra constants to keep somewhere.

Two ways to deal with the extra constants

Store them

128 twelve-bit values: 192 bytes of ROM. Trivial on its own, but it is a second table next to the NTT twiddle table, with its own address logic.

Derive them

The PWM twiddles are $\pm$ entries that already sit in the NTT table, at a shifted index. Recognising that removes the second ROM and reuses the same adder and multiplier, with no extra arithmetic at all.

ML-DSA multiplies 256 numbers pointwise. ML-KEM multiplies 128 degree-1 polynomials modulo $(x^2 - \zeta)$.

Neither modulus was picked for looks

ML-KEM: $q = 3329$

$$3329 = 2^8 \cdot 13 + 1$$

Prime, so every non-zero value has an inverse.

$q - 1 = 2^8 \cdot 13$, so a 256th root of unity exists, but a 512th does not. That single fact is why the ML-KEM NTT stops at 7 stages and why its pointwise multiply works on pairs.

12 bits, so two coefficients fit in a 24-bit word and a 12×12 multiplier is enough.

ML-DSA: $q = 8380417$

$$8380417 = 2^{23} - 2^{13} + 1$$

Prime, and only three bits are set. Multiplying by q is three shifts and two adds: $x \cdot q = (x \ll 23) - (x \ll 13) + x$. No multiplier needed in the reduction step at all.

$q - 1$ is divisible by 2^{13} , so a 512th root of unity exists and the NTT runs all 8 stages to single constants.

This is a cryptographic standard designed by people who have built hardware. The shape of q is a gift: it turns the expensive part of modular reduction into wiring.

Watch out: 3329 has no such shape, so ML-KEM reduction really does need a small multiply. Designs that assume both moduli reduce the same way end up slower than either dedicated core.

Add noise, and the puzzle stops being solvable

Without noise: a first-year exercise

$$t = A \cdot s$$

$q = 13$, and you are told A and t :

$$5s_0 + 1s_1 + 9s_2 = 12$$

$$3s_0 + 7s_1 + 2s_2 = 6$$

$$8s_0 + 4s_1 + 6s_2 = 12$$

Gaussian elimination $\rightarrow s = (2, 1, 3)$. **Seconds.**

With noise: nobody has an answer

$$t = A \cdot s + e \quad \text{with } e \text{ small}$$

Same A , but each equation is off by a little:

$$5s_0 + 1s_1 + 9s_2 + e_0 = 0$$

$$3s_0 + 7s_1 + 2s_2 + e_1 = 5$$

$$8s_0 + 4s_1 + 6s_2 + e_2 = 12$$

3 equations, 6 unknowns. No unique solution to find.

The trick is that the noise is small, but not that small

Too small

$e \in \{-1, 0, 1\}$ in low dimension:
an attacker can search or round it
away. Broken.

Just right

Small enough that the legitimate
holder of s can subtract it and still
recover the message. Large enough
that guessing is hopeless.

Too big

The noise swamps the signal and
even the right key cannot decrypt.
This is where decryption failures
come from.

Without noise this is Gaussian elimination, a first-year exercise. With small noise added, recovering s is Learning With Errors, and its average case is provably at least as hard as the worst case of a lattice problem.

Why module earns its place in the name

A module is a vector space whose scalars come from a ring instead of a field.

	Plain LWE	Module LWE
What A is	700×700 integers	3×3 polynomials of 256 coefficients
Public key size	hundreds of kilobytes	1184 bytes
Multiply cost	n^2 integer multiplies	k^2 NTTs, each $n \log n$
Structure to exploit	none	ring structure $\rightarrow$ NTT works
Security argument	reduces to worst-case lattice problems	same, with a mild extra assumption
Used by	textbooks	ML-KEM and ML-DSA

The trade

Adding ring structure makes everything smaller and faster. It also gives an attacker more structure to work with, which is why the module version uses several small polynomials instead of one big one.

Where that leaves you

Fifteen years of cryptanalysis says the trade is safe at these parameters. And it is the only reason a post-quantum handshake fits in a Cortex-M4 with a few kilobytes of stack.

Stack polynomials into a matrix and you have a module. Structure buys you a key that fits in a packet: that is the trade the whole field made, and the reason any of this fits a constrained part.

Steps three and four, in the standards' notation

This is the “Module-Lattice” in ML-KEM and ML-DSA. Each cell below is one whole 256-coefficient polynomial in R_q .

$A (k \times k)$				$s (k)$		$e (k)$		$t (k)$
A_{00}	A_{01}	A_{02}		s_0		e_0		t_0
A_{10}	A_{11}	A_{12}	$\times$	s_1	$+$	e_1	$=$	t_1
A_{20}	A_{21}	A_{22}		s_2		e_2		t_2

$$t_0 = A_{00} \cdot s_0 + A_{01} \cdot s_1 + A_{02} \cdot s_2 + e_0: \text{three polynomial multiplies and three adds, per row}$$

The work, counter

ML-KEM-768 has $k = 3$, so $A \cdot s$ is 9 polynomial multiplies.

ML-DSA-65 has $k = 6, \ell = 5$, so 30.

Each one is a 256-point NTT problem.

Why not one giant lattice instead?

Plain LWE needs a 700×700 matrix of integers. Modules give the same security from a 3×3 matrix of polynomials, and the polynomial structure is what makes the NTT possible.

Stack polynomials into a matrix and you get MLWE: vectors of polynomials instead of scalars.

The three assumptions

MLWE

Module Learning With Errors

Given A and $t = A \cdot s + e$,
find the small s .

What it buys you

Hides the private key in
both ML-KEM and ML-DSA.

MSIS

Module Short Integer Solution

Given A , find a small
non-zero y with $A \cdot y = 0$.

What it buys you

Stops signature forgery
in ML-DSA.

SelfTargetMSIS

the inhomogeneous cousin

Given A and a target t ,
find small y with $A \cdot y = t$,
where t comes from a hash.

What it buys you

The precise assumption
ML-DSA's proof needs.

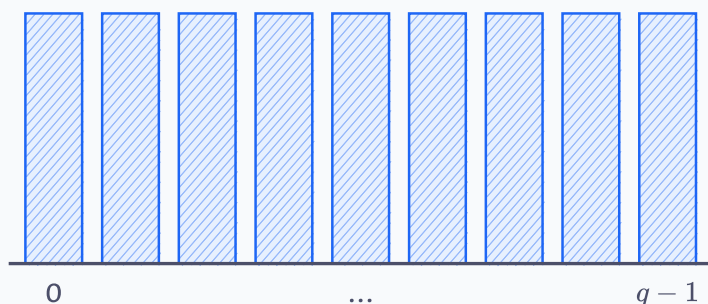
All three reduce to worst-case lattice problems. That is the strongest kind of guarantee this field offers: solving a random instance would imply solving the hardest instance of a problem studied since 1996.

The caveat: “no known quantum algorithm helps” is not the same as “no quantum algorithm exists”. Nobody can prove the latter for any public-key scheme, including RSA.

MLWE hides the key. MSIS and SelfTargetMSIS stop forgery.

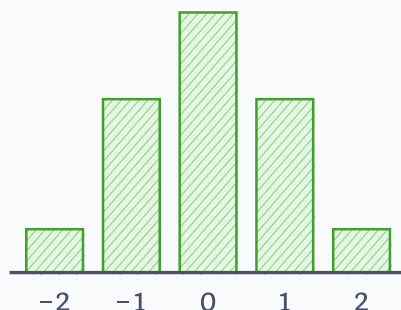
Three random shapes, three different jobs

Uniform over $\mathbb{Z}_q$



Used for: the public matrix A .
Made by: SHAKE128, then reject any 12-bit sample $\geq q$.

Centred binomial $\text{CBD}(\eta)$: small and symmetric



Used for: the secret s and the noise e .
Made by: count bits. For $\eta = 2$, sum two bits and subtract two bits: values $-2 \dots 2$.

Fixed-weight ± 1 (SampleInBall)

256 coefficients, all zero except exactly τ of them, which are ± 1 .



$\tau = 39, 49$ or 60 depending on the parameter set.

Used for: the ML-DSA challenge c .
Why: a tiny, sparse multiplier keeps $c \cdot s_2$ small enough to hide.

The engineering consequence you will feel

All three come out of Keccak

SHAKE128 for A , SHAKE256 for the secrets, SHA3 for the hashes. There is no separate RNG algorithm to implement, but there is a lot of Keccak, and that is where most of your cycles go.

Rejection sampling means variable time

Sampling A throws away about one 12-bit value in five, so the loop count depends on the seed. That is fine, because A is public. Never let a rejection loop touch secret data without care.

Uniform for the public matrix, centred binomial for secrets, sparse plus and minus one for the challenge.

The numbers to tape to your monitor

	ML-KEM (FIPS 203)	ML-DSA (FIPS 204)	What it means for the hardware
Ring degree n	256	256	same, so one NTT engine fits both
Modulus q	3329	8380417	12-bit vs 23-bit datapath
Coefficient width	12 bits	23 bits	two 12-bit values pack into one 24-bit word
Root of unity ζ	17 (256th)	1753 (512th)	ML-KEM has no 512th root → incomplete NTT
NTT stages	7	8	ML-KEM stops one stage early
After NTT, a poly is	128 linear pieces	256 constants	pointwise multiply differs
Pointwise multiply	5 mults per 2 coeffs	1 mult per coeff	ML-KEM's PWM needs extra twiddles
Matrix shape	$k \times k, k = 2/3/4$	$k \times \ell$, up to 8×7	ML-DSA does more polynomial work
Hash / XOF	SHAKE128, SHAKE256, SHA3-256/512	SHAKE128, SHAKE256, SHA3-256/512	identical Keccak core
Rejection loop?	only when sampling A	yes, in signing; 4 to 5 tries typical	signing time is variable
The headline for a hardware architect: same n , same ring shape, same Keccak, different modulus width. That is exactly the situation where one unified datapath beats two separate cores.			

Part 2 in five lines

A lattice is a grid of whole-number combinations

Walking to a point is easy. Finding the nearest point is not.

A polynomial is an array of 256 numbers modulo q

The ring guarantees it never grows, which is what makes a fixed datapath possible.

The NTT changes coordinates so polynomial multiply is pointwise

From 65,536 multiplications to 256, via eight stages of butterflies.

Both moduli were picked specifically so the NTT exists and reduction is free

ML-KEM stops at 7 stages (incomplete NTT); ML-DSA runs all 8 stages.

Add small error e to $A \cdot s$ and the system becomes underdetermined

That is the hard problem (MLWE), and it is the key generation of both schemes.

LIVE DEMO 1 · 8 MINUTES

Build a lattice cipher in twelve lines

Switch to the notebook. Section 2, cells 3 to 12.

What runs

The same lattice with a good basis and a bad basis, then a toy LWE encryption over the integers mod 97. Encrypt a bit, decrypt it, and brute-force the secret at a small dimension.

What to watch for

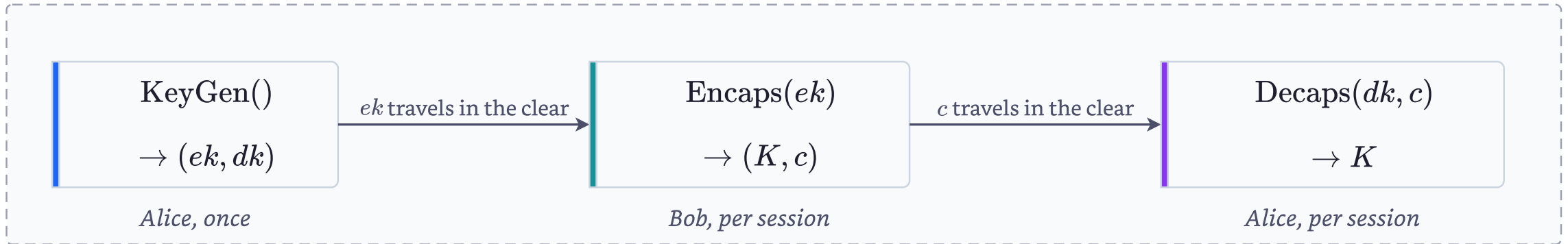
Turn the noise up until decryption starts failing. That boundary is exactly the design constraint behind every ML-KEM parameter set. Then turn the dimension down and watch a brute-force attack succeed.

PART 03

ML-KEM, FIPS 203

Key exchange after the transition, drawn out one step at a time

The whole interface, and where it sits



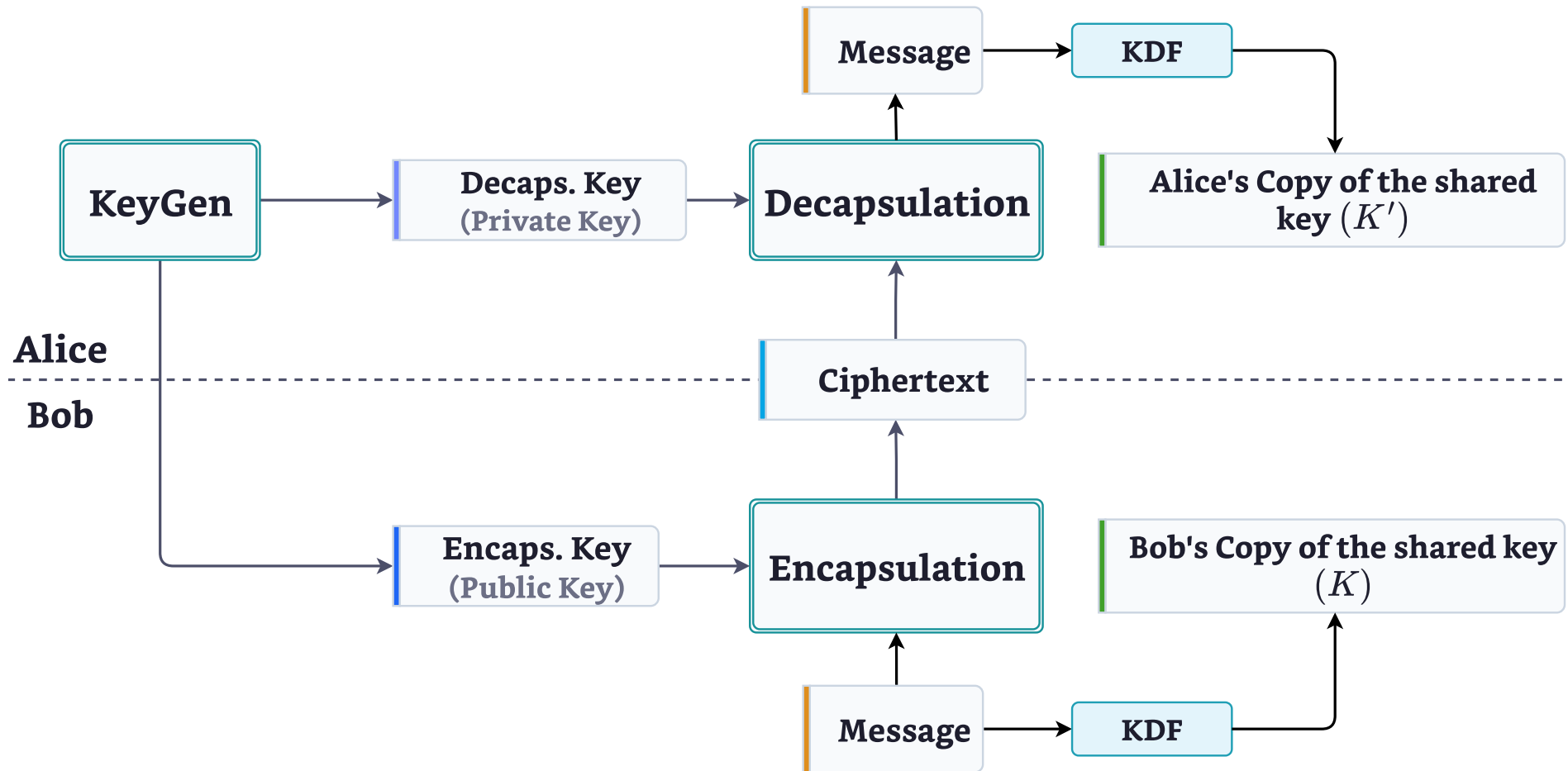
The sizes your protocol has to carry (ML-KEM-768)

Object	Size	Where it lives	For comparison
ek , the encapsulation key	1 184 bytes	public, sent once	32 B for X25519
dk , the decapsulation key	2400 bytes	never leaves the device	32 B for X25519
c , the ciphertext	1088 bytes	public, sent per session	32 B for X25519
K , the shared secret	32 bytes	feeds your KDF, as before	32 B for X25519

Note the last row: the shared secret is still 32 bytes. Everything downstream of the handshake is unchanged.

KeyGen, Encaps, Decaps. There is nothing else to choose, and that is deliberate: no message means no padding to get wrong.

The whole interface, and where it sits



The same three calls, mapped onto a session you already know: Encaps needs a public key, Decaps needs a secret key.

Compress and decompress, in modular arithmetic

Lossy Compression

$$\text{Compress}_d(x)$$

- Maps $\mathbb{Z}_q$ elements into 2^d integers:
$$\text{Compress}_d(x) = \left\lfloor \frac{2^d}{q} x \right\rfloor \bmod 2^d$$
- Discards noisy low-order bits, slashing ciphertext and public key bandwidth.
- Applied to public vectors and ciphertexts ($d = 10, 11$ for $\mathbf{u}$, $d = 4, 5$ for $\mathbf{v}$).

Decompression

$$\text{Decompress}_d(y)$$

- Reconstructs coordinates back into $\mathbb{Z}_q$:
$$\text{Decompress}_d(y) = \left\lfloor \frac{q}{2^d} y \right\rfloor$$
- Bounded deterministic error:
$$|x' - x| \leq \left\lfloor \frac{q}{2^{d+1}} \right\rfloor$$
- Purely public arithmetic; no secret key needed to lift values back into $\mathbb{Z}_q$.

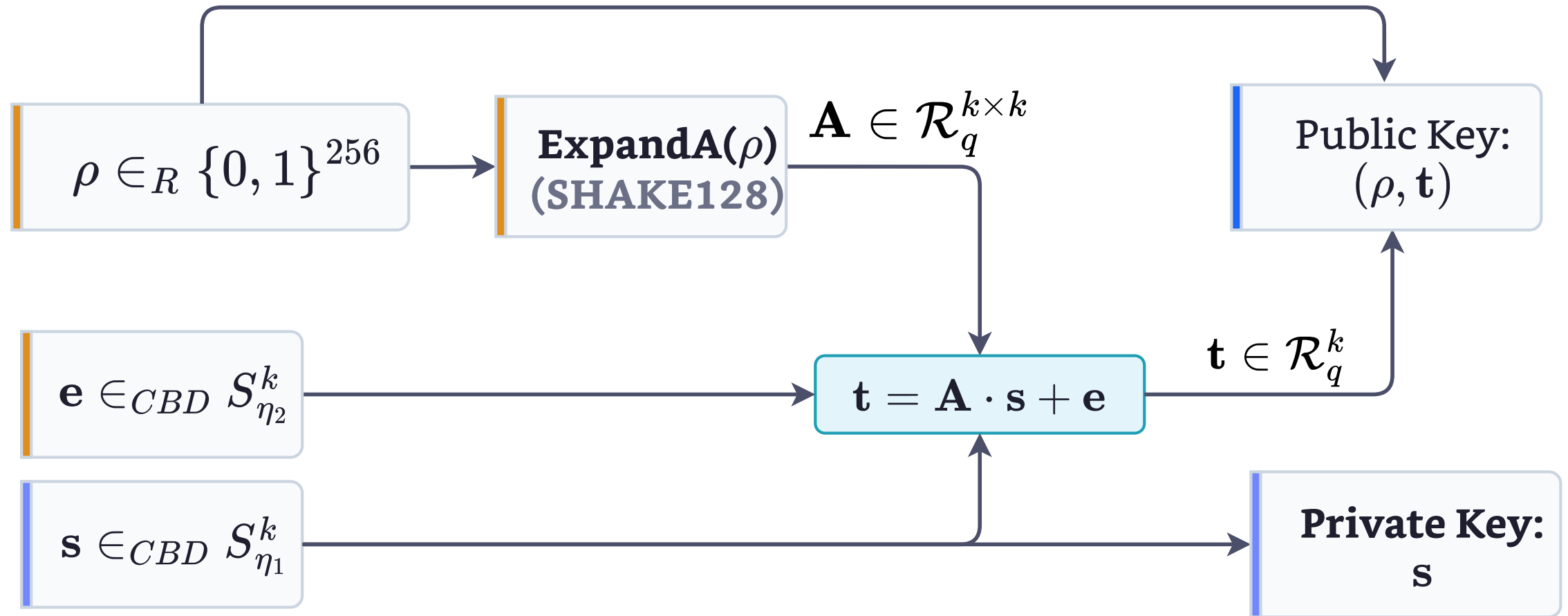
Message Encoding

$$d = 1 \text{ (Mod } q \text{ Rounding)}$$

- A 1-bit message is scaled across the circle:
$$\text{Decompress}_1(m) = m \cdot \left\lfloor \frac{q}{2} \right\rfloor \rightarrow (0 \text{ or } \approx 1664)$$
- Maximizes separation between 0 and 1 on the modular circle.
- Decryption recovers m via $\text{Compress}_1(w)$ if noise stays strictly inside $\pm \lfloor q/4 \rfloor$ (± 832).

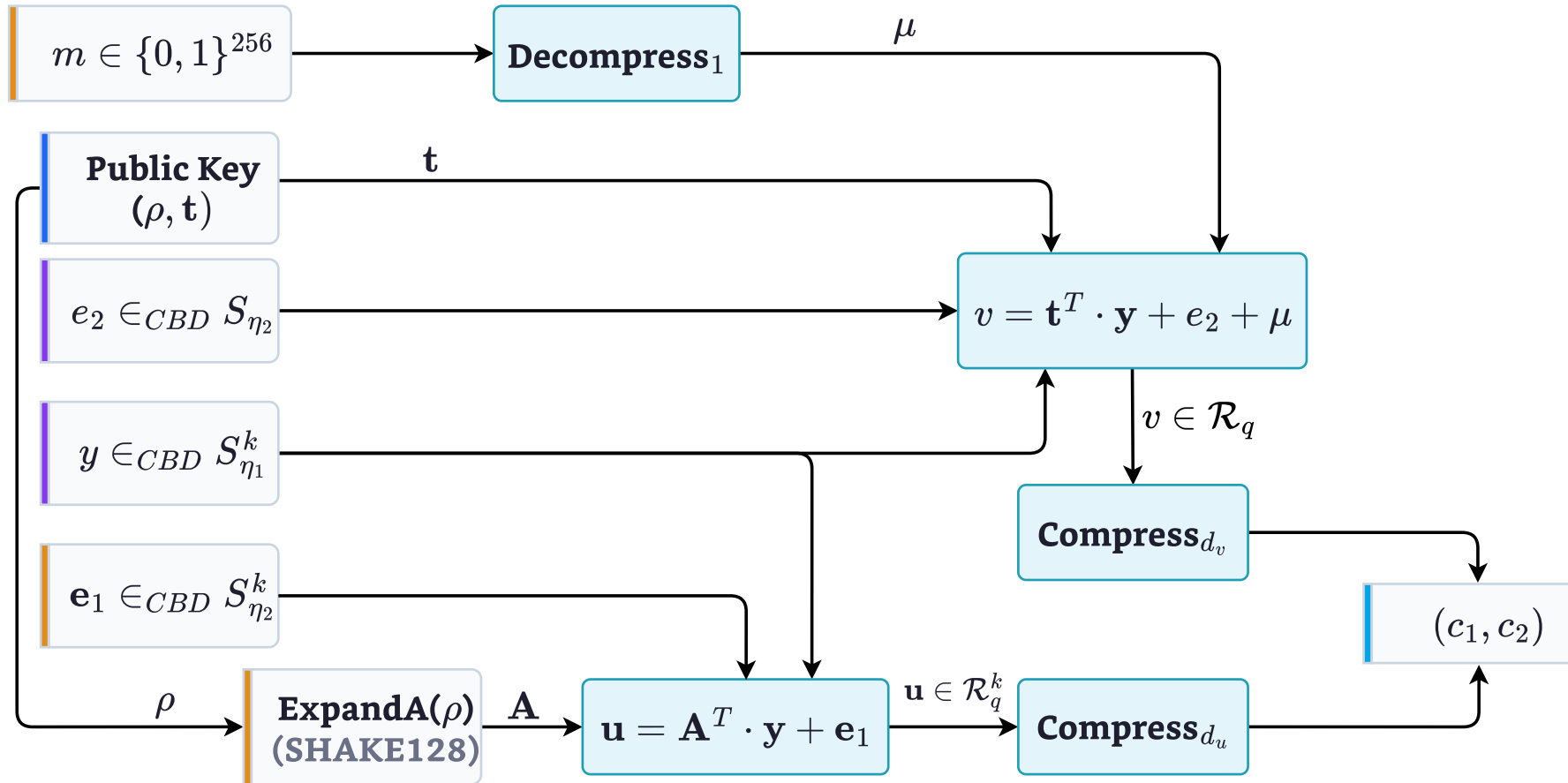
Compression saves network bandwidth; decompression restores coordinates for polynomial arithmetic.

K-PKE: Key generation



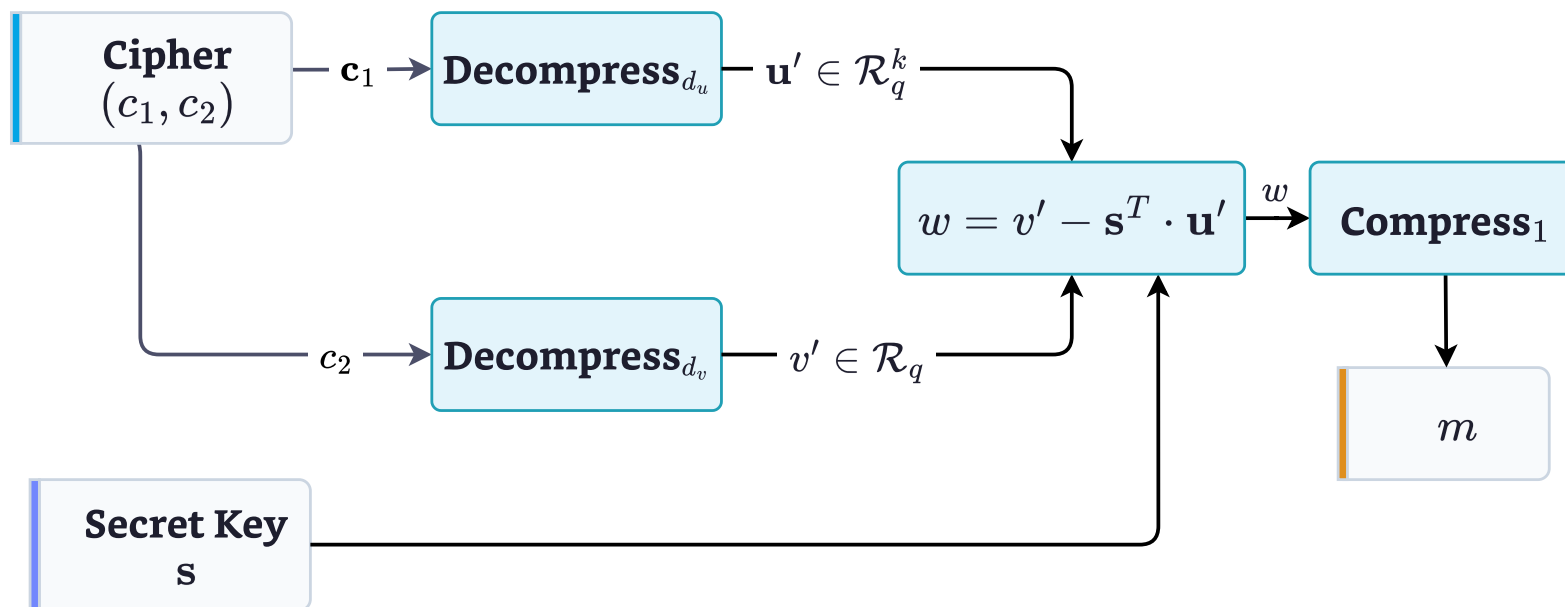
Sample seed ρ , expand public matrix $\mathbf{A}$ via SHAKE128, sample small secret $\mathbf{s}$ and error $\mathbf{e}$ from CBD, compute $\mathbf{t} = \mathbf{A} \cdot \mathbf{s} + \mathbf{e}$.

K-PKE: Encryption



Sample ephemeral vector y and errors e_1, e_2 . Compute $u = A^T \cdot y + e_1$ and $v = t^T \cdot y + e_2 + \text{Decompress}(m)$, then compress both to form ciphertext (c_1, c_2) .

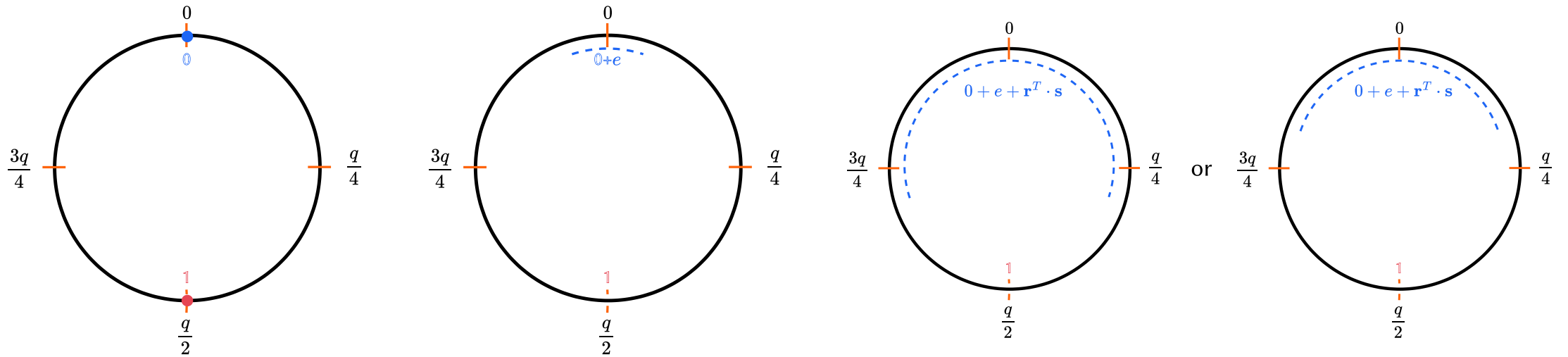
K-PKE: Decryption



$$\begin{aligned}
 w &= v - \mathbf{s}^T \mathbf{u} \\
 &= \mathbf{t}^T \mathbf{y} + e_2 + \mu - \mathbf{s}^T \mathbf{A}^T \mathbf{y} - \mathbf{s}^T \mathbf{e}_1 \\
 &= \cancel{\mathbf{s}^T \mathbf{A}^T \mathbf{y}} + \mathbf{e}^T \mathbf{y} + e_2 + \lceil \frac{q}{2} \cdot m \rceil - \cancel{\mathbf{s}^T \mathbf{A}^T \mathbf{y}} - \mathbf{s}^T \mathbf{e}_1 \\
 &\approx \lceil \frac{q}{2} \cdot m \rceil + (\mathbf{e}^T \mathbf{y} + e_2 - \mathbf{s}^T \mathbf{e}_1) \\
 &\Rightarrow \text{Compress}_1(w) \approx m
 \end{aligned}$$

Decompress ciphertext components (c_1, c_2) , compute $w = v - \mathbf{s}^T \cdot \mathbf{u}$, and apply $\text{Compress}_1(w)$ to recover the original message bits.

Hiding one bit, in three moves



1. Stretch the bit to $q/2$, as far as two points can be

2. Add small noise, so it drifts but stays near

3. Add a full-range secret value, so it could be anywhere

Decryption failures: the actual numbers

Well inside $q/4$ is a probabilistic statement, not a guarantee

Parameter set	Failure probability	Roughly
ML-KEM-512	$2^{-138.8}$	$10^{-41.8}$
ML-KEM-768	$2^{-164.8}$	$10^{-49.6}$
ML-KEM-1024	$2^{-174.8}$	$10^{-52.6}$

Your protocol must survive a key mismatch, and the failure must never be observable. One bit of that ciphertext failed, queried enough times, recovers the private key.

Where the matrix comes from, and why it matters to you

A is big

$k \times k$ polynomials of 256 twelve-bit coefficients. For ML-KEM-768 that is 4.5 kB; for ML-DSA-65 it is 30 kB.

Nobody sends it

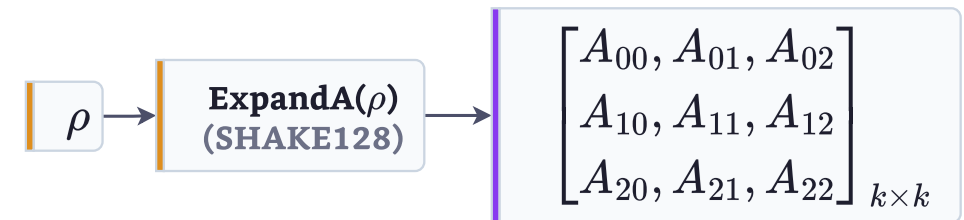
Both sides regenerate it from a 32-byte seed ρ , using SHAKE128.

Which is why a PQC accelerator needs a Keccak block

Not just an NTT block. Remember this slide in part 5.

And why it is also your RAM budget

You can expand A once into memory, or generate it one polynomial at a time and consume it as it comes out of SHAKE. On a constrained part, always the second.



From CPA to CCA: the Fujisaki-Okamoto transform

K-PKE alone: safe against a passive listener

Someone who only reads traffic learns nothing. That is CPA security, and the lattice problem gives it to you directly.

Not enough. An attacker who can send you ciphertexts and watch what happens can recover the key.

ML-KEM: the Fujisaki-Okamoto transform

Decapsulation re-runs the encryption itself and checks that it would have produced the ciphertext it was given. Anything tampered with is rejected.

Now it is CCA2-secure. You can safely decapsulate anything that arrives, including deliberate garbage.

What Decaps actually does, in order



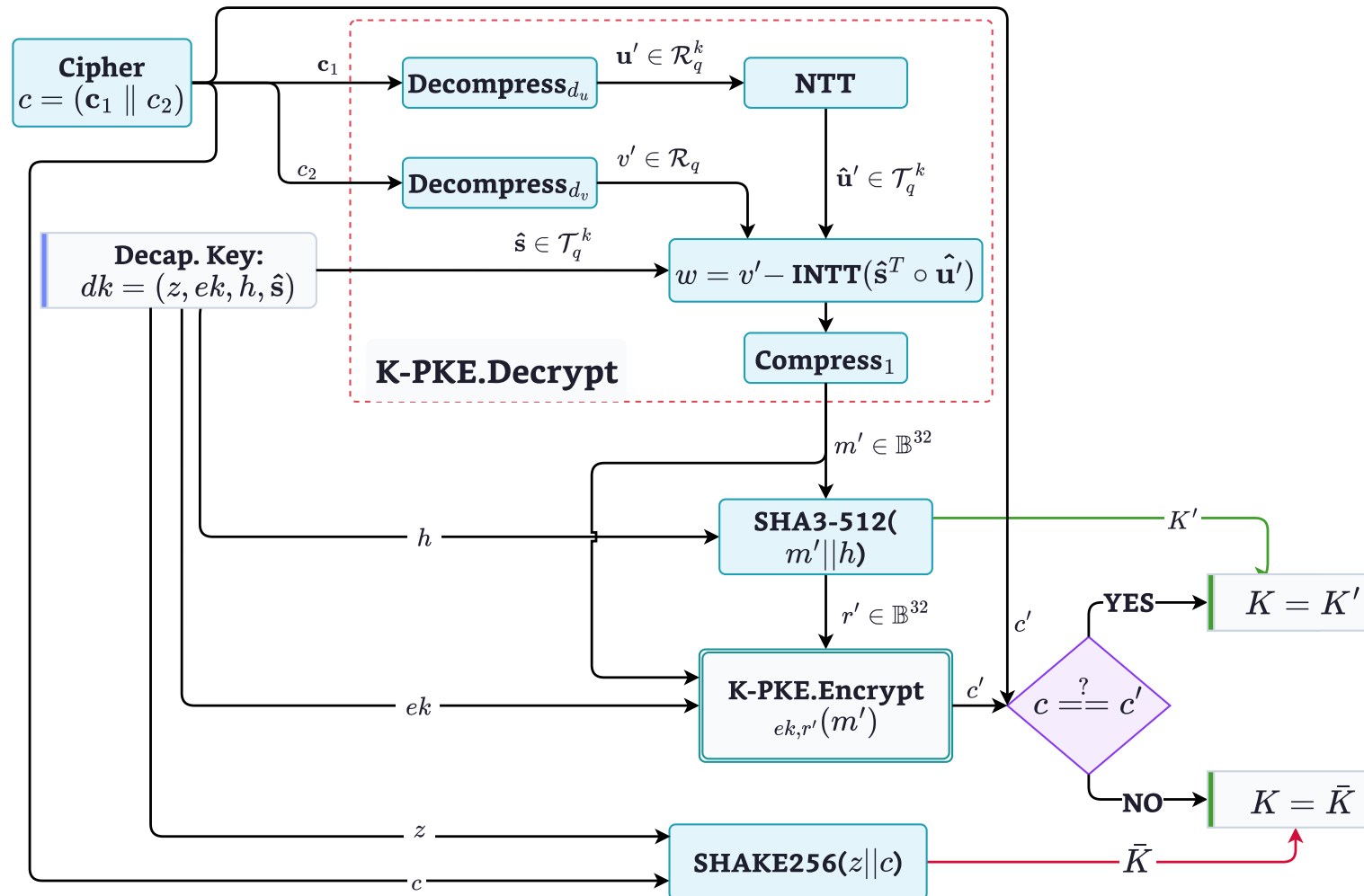
Implicit rejection: the part you must get right

On failure, ML-KEM does not return an error. It returns a key derived from a secret value z stored in the private key. The caller cannot tell success from failure. The handshake then fails later, when the two sides find their keys do not match, with nothing leaked.

Two ways implementations get this wrong

1. Returning an error code on mismatch. That single bit, queried enough times, recovers the key.
2. Comparing c and c' with `memcmp`, which stops at the first difference. Fold the whole buffer into one accumulator and test once.

ML-KEM: decapsulation, in full



Read that diagram again, and count the hashes

Primitive	What it is doing	Where	Permutations
SHA3-256	hash the encapsulation key	ML-KEM Encaps, Decaps	1
SHA3-512	derive (shared secret, randomness) from the message	ML-KEM Encaps, Decaps	1
SHAKE128	expand the seed into the public matrix A	both, key generation and use	27 for ML-KEM-768, 150 for ML-DSA-65
SHAKE256	sample the secret and the noise	both	7 for ML-KEM-768, 55 for ML-DSA-65
SHAKE256	implicit-rejection key on failure	ML-KEM Decaps	1
SHAKE256	hash the message into μ	ML-DSA Sign, Verify	1–2
SHAKE256	derive the challenge $\tilde{c}$ from μ and w_1	ML-DSA Sign, Verify	1 per rejection round
SHAKE256	expand the mask y from a seed and a counter	ML-DSA Sign	several per rejection round

One core covers all of it

SHA3-256, SHA3-512, SHAKE128 and SHAKE256 are the same permutation with different rates and padding. One block.

And look at where they go

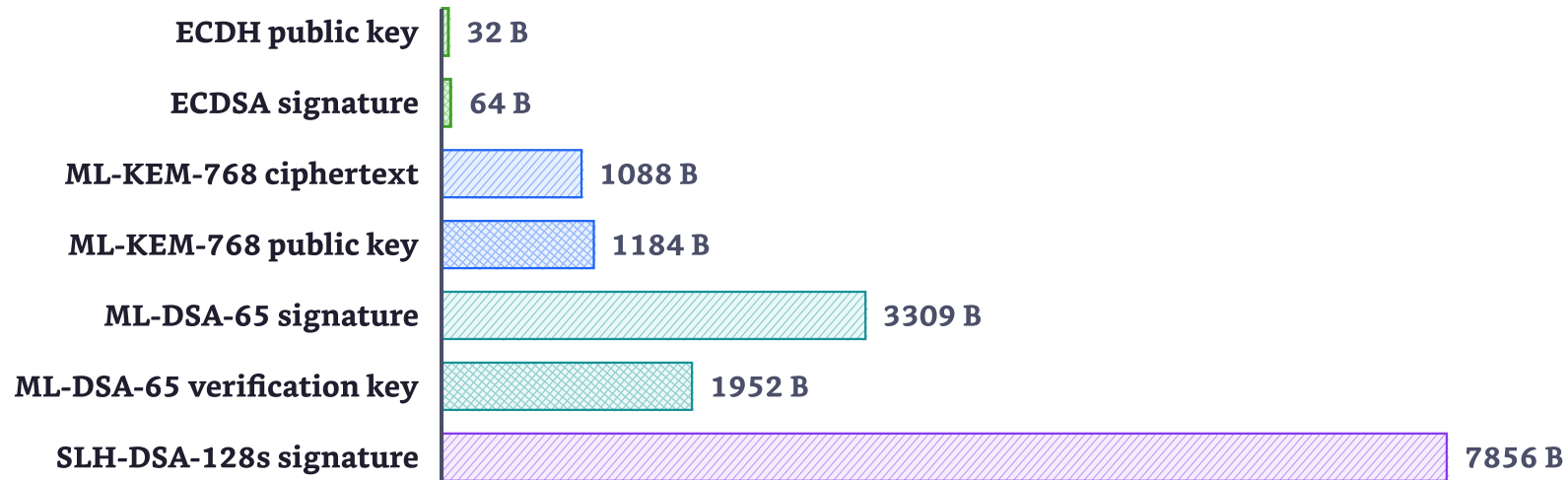
44 permutations for one ML-KEM-768 encapsulation;
27 of those are expanding A .
For ML-DSA-65 it is 337 and 150.

Sizes, next to what you have today

Scheme	Public key	Private key	Ciphertext	Shared secret
ML-KEM-512	800 B	1632 B	768 B	32 B
ML-KEM-768	1184 B	2400 B	1088 B	32 B
ML-KEM-1024	1568 B	3168 B	1568 B	32 B
RSA-2048 key transport	256 B	256 B	256 B	32 B
Diffie-Hellman 2048	256 B	256 B	256 B	32 B
ECDH / X25519	32 B	32 B	32 B	32 B

The last column never changes. Everything after the handshake, your AES, your HMAC, your KDF, stays exactly as it is.

The number that will actually break your product



Now compare that with what your link can carry in one frame

Link	Payload per frame	What that means
Ethernet / IP MTU	1500 B	fits one ML-KEM key, splits a ML-DSA signature
IPv6 minimum MTU	1280 B	an ML-KEM public key barely fits
BLE 5 extended data length	251 B	everything fragments
CAN FD frame	64 B	an ML-DSA signature is 52 frames
LoRaWAN payload (EU868, DR5)	~242 B	signatures need many uplinks
NFC / ISO 14443 APDU	~256 B	chained transfers required
802.15.4 / Thread frame	127 B	fragmentation at 6LoWPAN

ML-KEM: the three parameter sets

Scheme	n	q	k	eta1	eta2	du	dv	Category
ML-KEM-512	256	3329	2	3	2	10	4	1 (AES-128)
ML-KEM-768	256	3329	3	2	2	10	4	3 (AES-192)
ML-KEM-1024	256	3329	4	2	2	11	5	5 (AES-256)

Only k , η_1 and the compression widths change. One datapath covers every security level, and the parameter set is a register value.

Part 3 in five lines

A KEM has no message and no padding

KeyGen, Encaps, Decaps, and nothing to choose.

The mechanism is one bit stretched to $q/2$

Buried under a large secret-dependent value, and recovered by rounding.

Decapsulation re-encrypts and compares

On mismatch it returns a different key, never an error.

The shared secret is still 32 bytes

The keys and ciphertexts are not, and on a constrained link that is the part that hurts.

The matrix A comes out of SHAKE128 every single time

Budget for Keccak, and generate A on the fly.

LIVE DEMO 2 · 8 MINUTES

Real ML-KEM, in Python you can read

Switch to the notebook. Section 4, cells 23 to 29.

What runs

A complete FIPS 203 implementation, a few hundred lines, with a self-check suite that would catch almost any real bug. Run KeyGen, Encaps and Decaps, and confirm both sides derive the same 32 bytes.

What to watch for

Flip a single bit of the ciphertext and watch decapsulation return a perfectly valid-looking key that simply does not match. That is implicit rejection doing its job. Then count the SHAKE calls.

Break

30 minutes. We start again at **12:00** with ML-DNA, signatures, and the rejection loop.

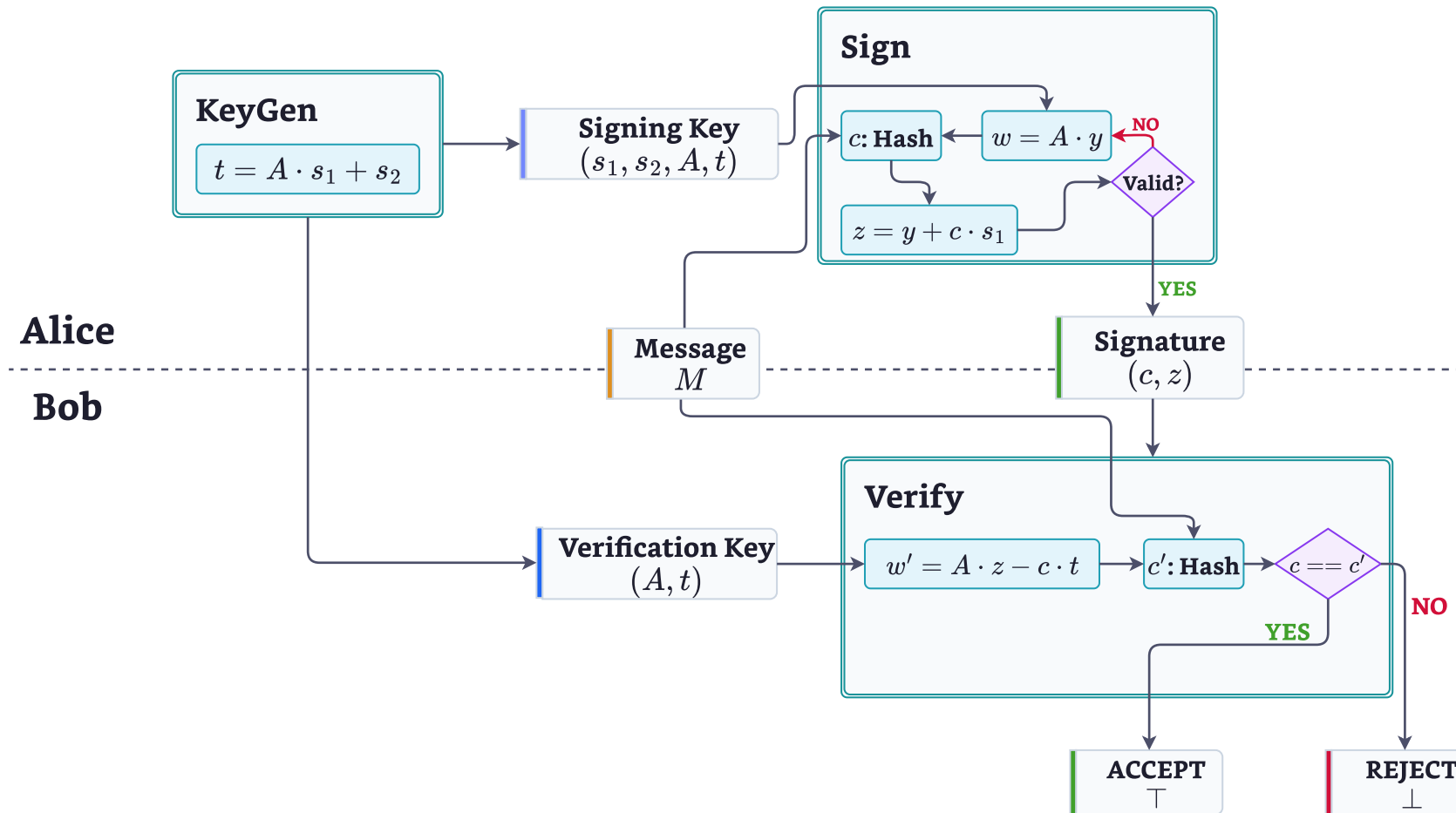
Sections 1 to 4 of the notebook are done. If you have not run it yet, the whole thing takes ten seconds: `pip install notebook numpy matplotlib`, then open `PQC_Workshop.ipynb`.

PART 04

ML-DSA, FIPS 204

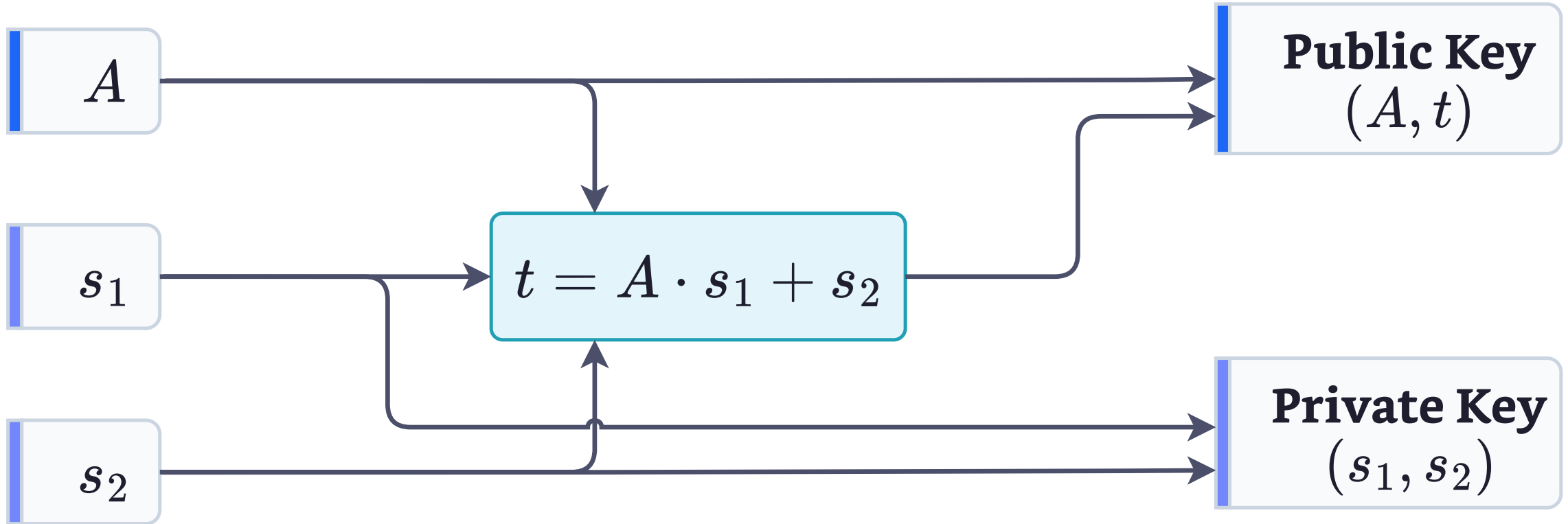
Signatures, the rejection loop, and why they are so much bigger

The same three moves as Schnorr, with polynomials



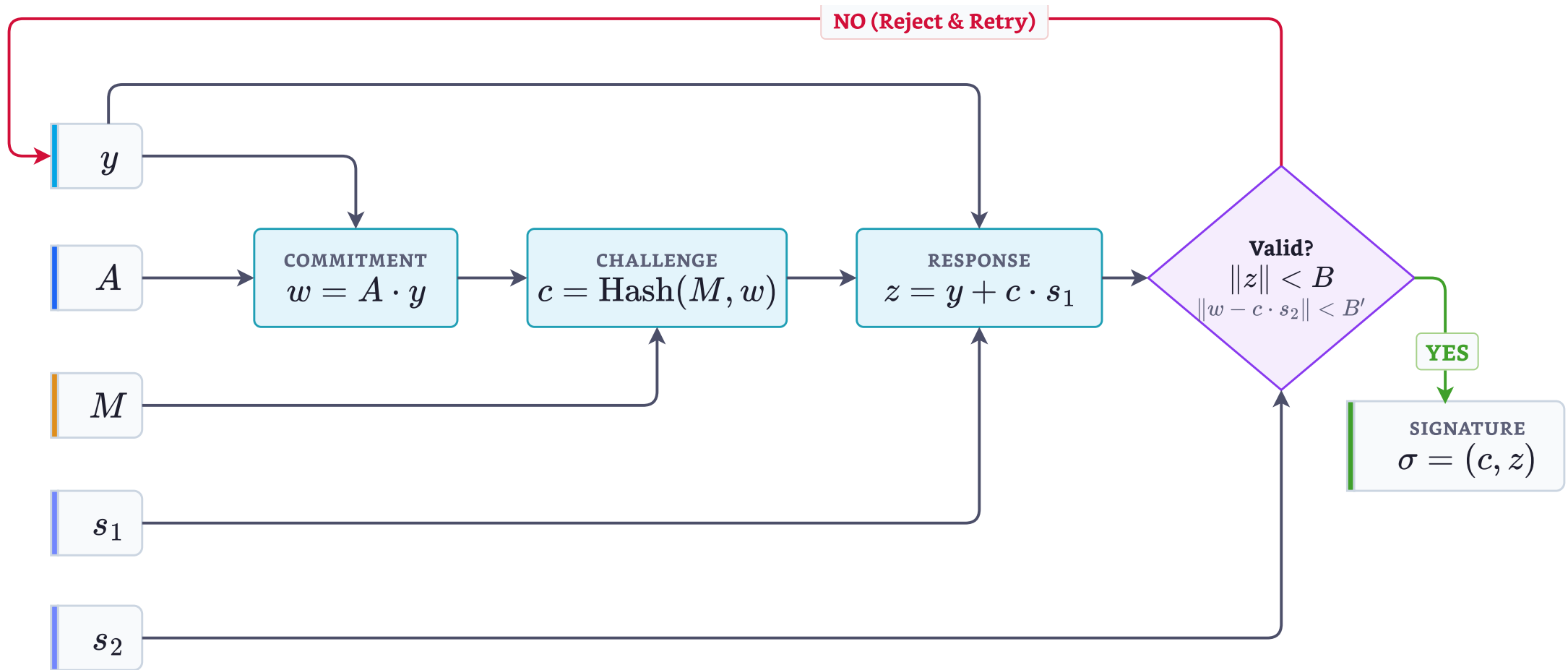
Commit, challenge, respond: the shape is thirty years old and unchanged. Key generation is the same MLWE equation as ML-KEM, with the two secrets named s_1 and s_2 . The one new thing is a diamond in the middle.

Simplified key generation



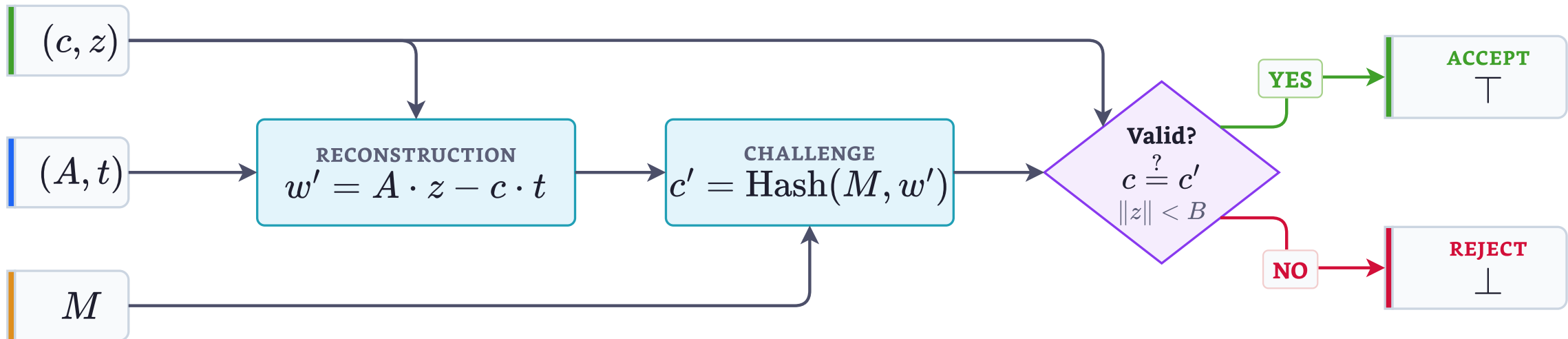
Exactly the same MLWE instance as ML-KEM, with the two secrets named s_1 and s_2 instead of s and e .

Simplified signing



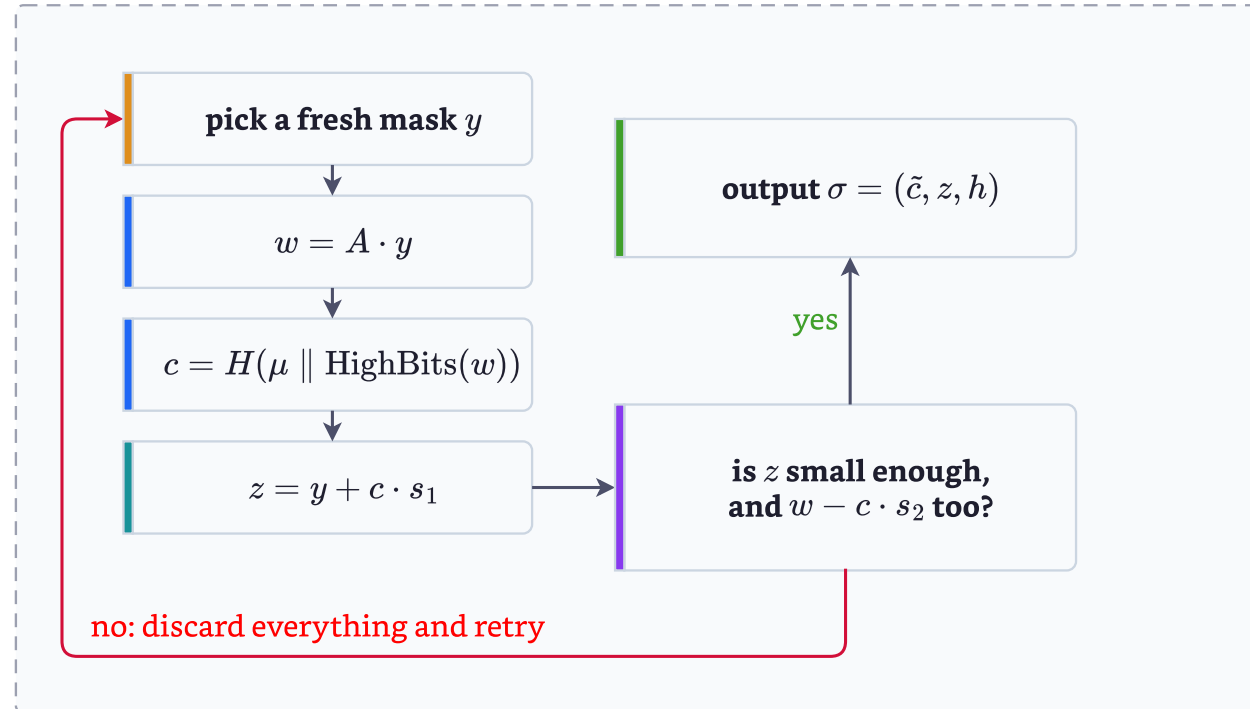
Commit with a fresh mask, hash to get a challenge, respond with $z = y + c \cdot s_1$, and throw it away if it is too big.

Simplified verification



Recompute the commitment from public data alone, hash it, and check you get the same challenge. The verifier is off by exactly $c \cdot s_2$, and if that is small enough the high bits do not change.

Fiat-Shamir with aborts



Why not just publish z ?

$z = y + c \cdot s_1$. If y were small, z would carry a visible imprint of $c \cdot s_1$, and a few hundred signatures would reveal s_1 .

Rejecting the large ones makes the published z look independent of s_1 , because the distribution is the same whatever the key is.

What it costs

Parameter set	Expected attempts
ML-DSA-44	~4.3
ML-DSA-65	~5.1
ML-DSA-87	~3.9

Worst case is unbounded, though astronomically unlikely.

Two consequences you have to design around

1. **Signing time is non-deterministic:** one message might take 1 loop iteration, while the next needs 5 or more, taking several times longer. A hard real-time budget must use the worst realistic case, not the average
2. **Every retry re-runs Keccak** to expand a new y and to hash a new challenge. The loop is why signing costs several times what verifying does.

And one thing that is fine

The number of attempts depends on y , which is fresh and secret, and on the message. It is safe to let that vary.

What is **not** safe is revealing *which* of the three checks failed, or how close a rejected z was to the bound. Compute all the checks, then combine them.

The leak the rejection loop prevents

$z = y + c \cdot s_1$, and y is uniform over its range

If the norm of z comes out at its maximum, then $c \cdot s_1$ must have been exactly at the bound. That is a fact about the private key, published.

Values just under the maximum leak a little less

They still leak.

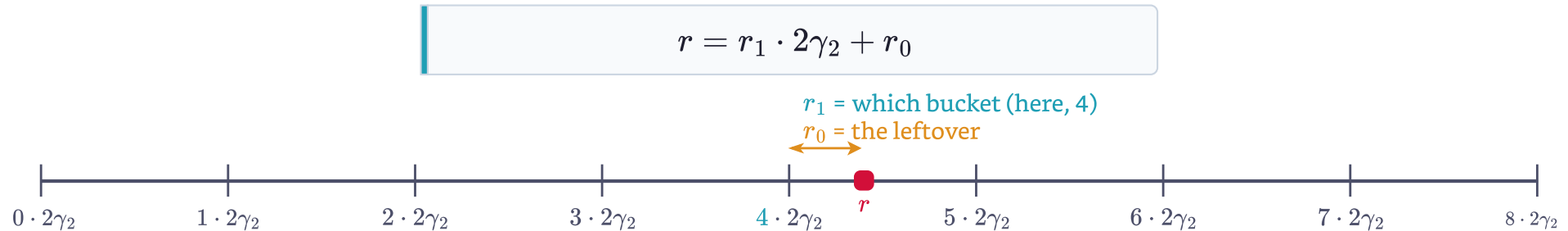
The fix

Only release z when its norm is below $\gamma_1 - \beta$. Inside that bound every possible z is equally likely whatever the private key is, so the signature carries no information about s_1 at all.

Same failure mode as a reused ECDSA nonce

And the same lesson: the randomness in a signature is not decoration.

Splitting numbers in two, and why the key needs hints



Three variants, three jobs

Operation	What it does	Used for	In practice
HighBits / LowBits	split around a chosen γ_2	the commitment w , so the verifier only needs the coarse part	$w_1 = \text{HighBits}(w)$
Power2Round	split at a bit position, 2^d	the public key t , so only the top bits are published	$t_1 = t \gg 13$
MakeHint / UseHint	one bit saying “the bucket is off by one”	letting the verifier repair its coarse value	at most ω ones

Why this saves so much space

The public key t is 256×23 bits per polynomial. Publishing only the top 10 bits cuts the verification key by more than half, from 4448 to 1952 bytes for ML-DSA-65.

And why hints are needed at all

Throwing away the low bits of t means the verifier's arithmetic can land in the neighbouring bucket. One hint bit per coefficient tells it when to step across. At most ω of them are set, which bounds the signature size.

Dropping 13 lower bits shrinks the public key from 4448 to 1952 bytes; 1-bit hints resolve boundary crossings.

Splitting numbers in two, and why the key needs hints

t is k polynomials of 23-bit coefficients

For ML-DSA-65 that is 4416 bytes, so only the top bits are published.

The verifier is then working with an approximation

That last dropped term can push a coefficient into the next bucket.

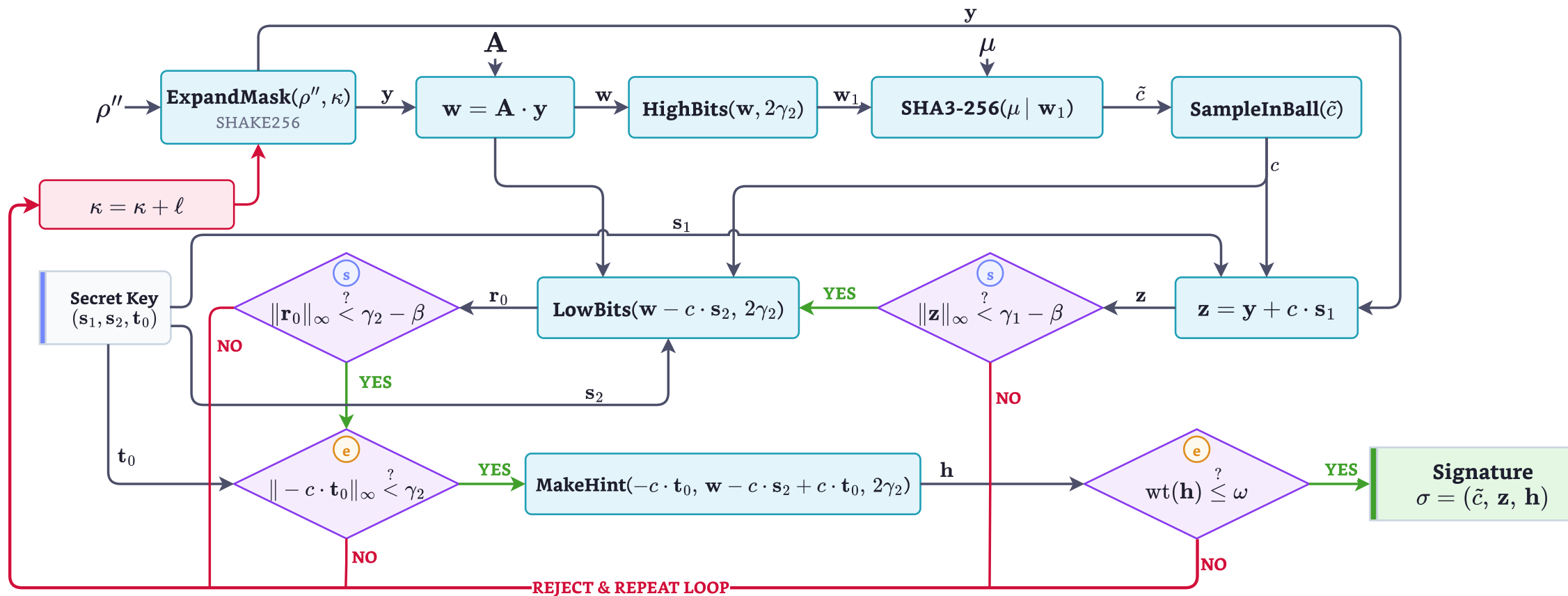
The hint is one bit per coefficient saying step across

At most ω of them are set, which is what bounds the signature size.

Dropping 13 bits shrinks the verification key from 4448 bytes to 1952

The hints cost far less than the bits they save. This is the fiddliest part of ML-DSA to implement correctly.

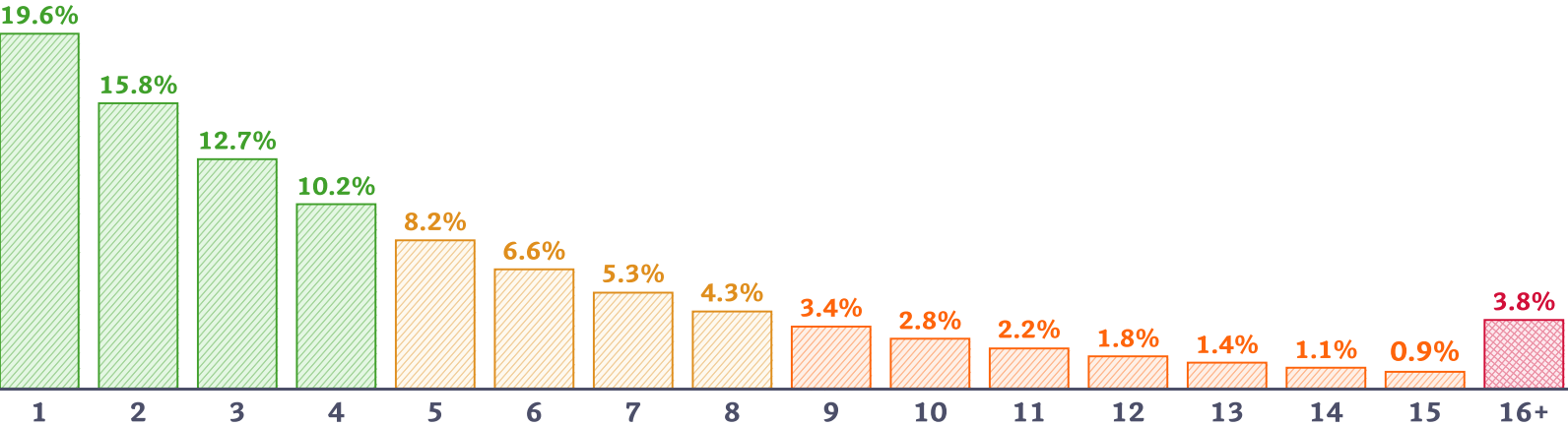
ML-DSA signing: inside the loop



Signing has no worst case, and your watchdog needs to know

ML-DSA-65 succeeds on any one attempt with probability about 1 in 5.1, independently each time.

Probability (% of signatures)



Attempts through the rejection loop before one signature comes out

Budget you might pick	Attempts	Cycles	At 80 MHz
Typical (median)	4	3.4 M	43 ms
1 signature in 20 (95%)	14	9.4 M	118 ms
1 signature in 100 (99%)	22	14.2 M	178 ms
1 in 1000 (99.9%)	32	20.2 M	253 ms
Absolute worst case	none	unbounded	unbounded

Verification has none of this: one pass, fixed work, no randomness. If your part only verifies, skip this slide.

What to do

Budget the tail, not the mean.

Feed the watchdog inside the loop, or run signing from a task and not an interrupt.

Never cap the attempts and fall back to something weaker.

Deterministic, hedged, and where the randomness comes from

Deterministic

$$\rho'' = H(K \parallel \mu)$$

Same message, same signature, every time.

- ✓ No random number generator needed.
- ✓ Reproducible, which auditors like.
- ✗ A fault attack can be repeated on the identical computation until it succeeds.
- ✗ Differential power analysis gets as many traces of the same operation as it wants.

Hedged: the default

$$\rho'' = H(K \parallel rnd \parallel \mu)$$

rnd fresh 32 bytes

Same message, different signature each time.

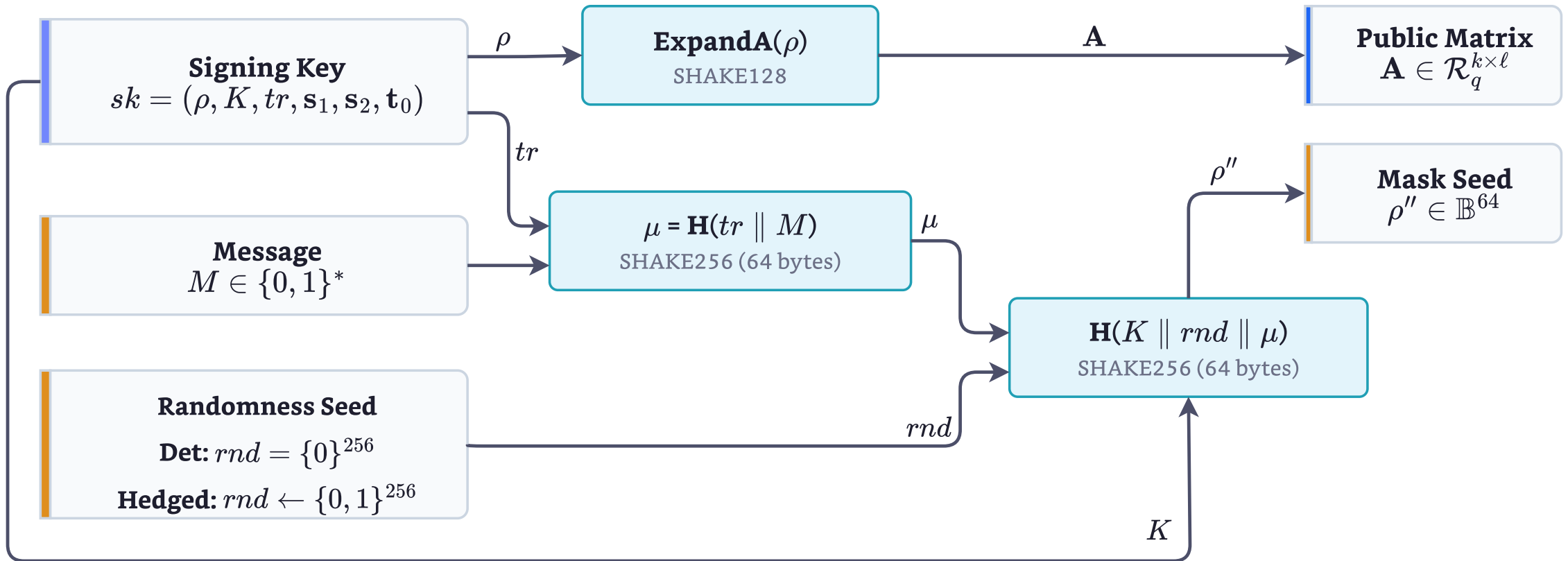
- ✓ Every trace is of a different computation.
- ✓ A fault has to succeed first time.
- ✓ Still safe if the RNG fails: it degrades to deterministic, not to broken.
- ✗ Needs 32 bytes of entropy per signature.

When each one is the right answer

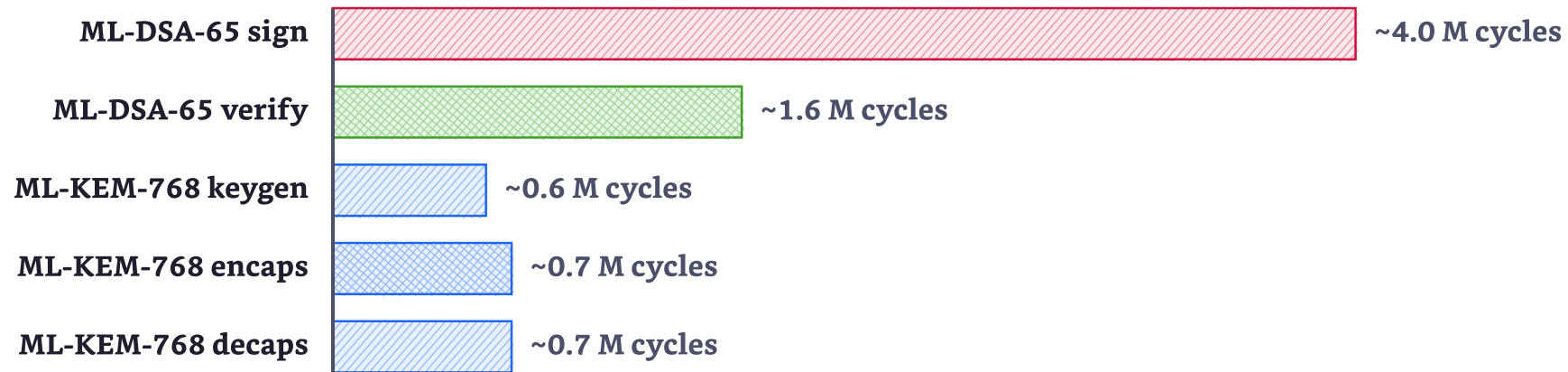
Situation	Choose	Because
A signing key in a device an attacker can hold	Hedged	power and fault attacks are the realistic threat
A build server signing firmware in a locked room	Either	physical access is already the bigger problem
Audit logs that must be byte-identical on replay	Deterministic	and accept the side-channel exposure
A device with no trustworthy entropy source	Deterministic	a biased RNG is worse than none here
Anything you are unsure about	Hedged	it is the safer default, and the standard's default

Hedged mixes 32 bytes of fresh entropy with secret seed; degrades gracefully to deterministic if RNG fails.

ML-DSA signing: before the loop



Signing and verifying are not the same job



Which side of the asymmetry is your device on?

Devices that only verify

Bootloaders, OTA clients, sensors checking a command's authenticity.

- ✓ Verify is the cheap direction.
- ✓ No private key to protect.
- ✓ No entropy needed.
- ✓ No rejection loop, so constant time.

This is the easy case. Most embedded devices are here.

Devices that sign

Attestation, mutual authentication, secure logging, device identity.

- ✗ Four times the work of verifying.
- ✗ A private key that must never leak.
- ✗ Needs entropy for hedged mode.
- ✗ Variable-time rejection loop.

Plan for the side-channel work here.

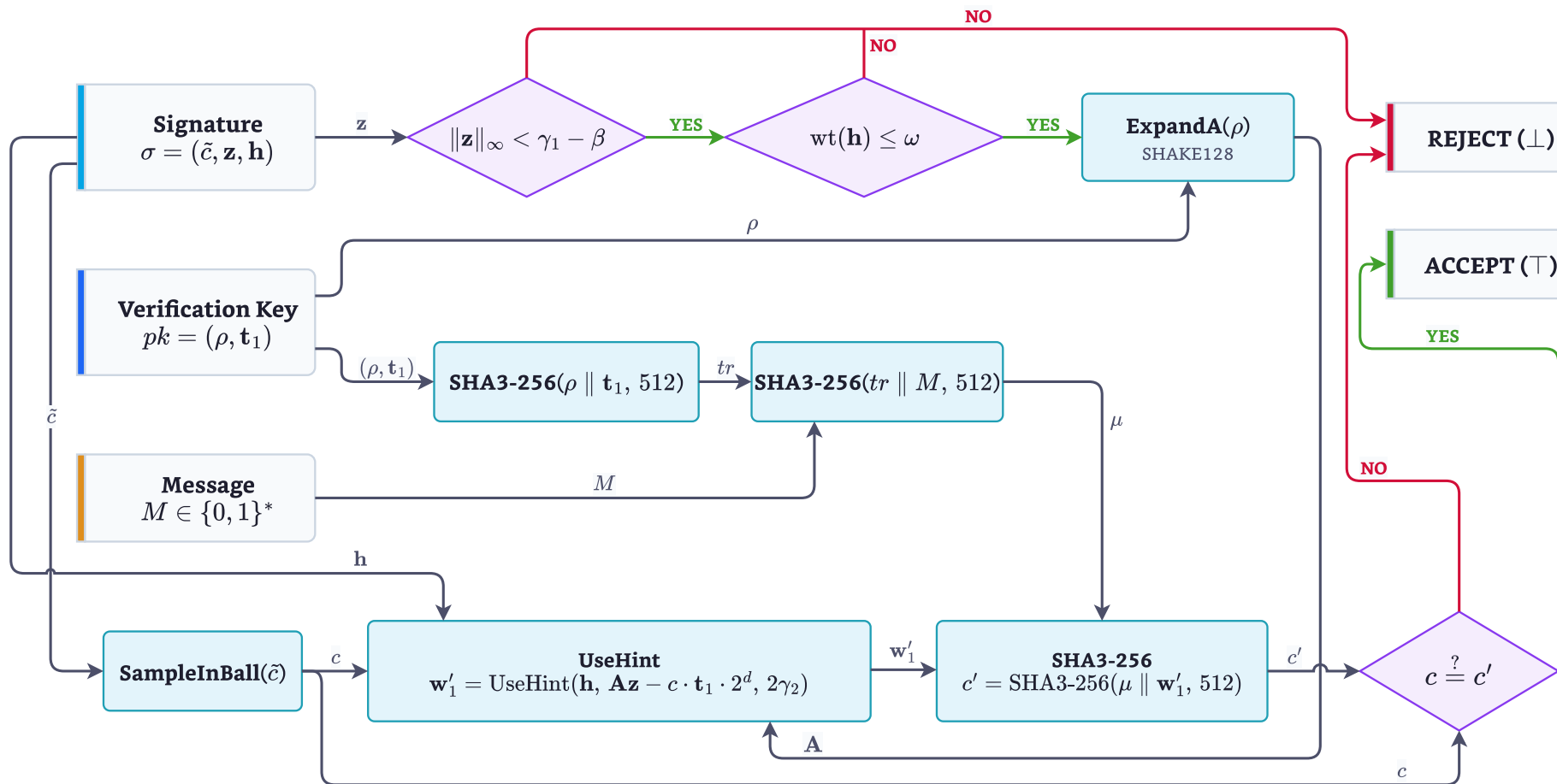
Devices that do key exchange

Anything that opens a TLS, DTLS or MQTT session.

- ✓ Cheapest of the three.
- ✓ Comparable to X25519.
- ✗ Ciphertext and key sizes hurt the protocol, not the CPU.

Fix this first: harvest-now-decrypt-later makes it the urgent one.

ML-DSA: verification



One pass, no loop, no randomness.

Signature sizes, across every option you have

Scheme	Signing key	Verification key	Signature
ML-DSA-44	2560 B	1312 B	2420 B
ML-DSA-65	4032 B	1952 B	3309 B
ML-DSA-87	4896 B	2592 B	4627 B
FN-DSA-512 (draft)	13953 B	1793 B	1280 B
SLH-DSA-SHA2-192s	n/a	48 B	16224 B
XMSS-SHA2_20_256	2573 B	64 B	2820 B
RSA-3072	384 B	384 B	384 B
ECDSA-P256	32 B	64 B	64 B

FN-DSA has the smallest signatures and needs floating-point arithmetic, a poor fit for most microcontrollers.
SLH-DSA has the smallest verification key, which is why it appeals for a root of trust in fuses.

ML-DSA: the three parameter sets

$q = 2^{23} - 2^{13} + 1 = 8380417$, $\zeta = 1753$ and $d = 13$ for all three

Scheme	lambda	gamma1	k	l	eta	tau	beta	omega	Category
ML-DSA-44	128	2^{17}	4	4	2	39	78	80	2
ML-DSA-65	192	2^{19}	6	5	4	49	196	55	3
ML-DSA-87	256	2^{19}	8	7	2	60	120	75	5

Expected signing attempts: 4.25, 5.1 and 3.85. ML-DSA-65 needs more attempts than ML-DSA-87, so higher security does not always mean slower signing.

Part 4 in five lines

ML-DSA is Schnorr with polynomials

A sparse plus and minus one challenge, and a rejection loop.

The loop exists so the published response carries no information about the private key

Same lesson as an ECDSA nonce.

Signing is variable-time by construction, with no worst case

Budget a percentile and design the watchdog around it. Verification is single-pass and fixed.

Rounding and hints exist to shrink the public key

They are the fiddliest part to implement correctly.

Use hedged signing

Unless you have a specific, written reason not to.

LIVE DEMO 3 · 8 MINUTES

Real ML-DSA, and a look inside the loop

Switch to the notebook. Section 5, cells 31 to 37.

What runs

A complete FIPS 204 implementation. Generate a key, sign a message, verify it, then check that the average number of rejection rounds matches the figure FIPS 204 predicts.

What to watch for

The rejection counter. Sign the same message twenty times in hedged mode and watch the attempt count vary from one to a dozen. That spread is the timing tail from the last part, measured. Then flip one byte of the message and watch verification reject it.

PART 05

Making it fit

RAM, energy, silicon: where the time goes, what runs out first, and what to put on the die

What it costs with no accelerator at all

Operation	Cycles	At 100 MHz	Peak stack	Code size
X25519 (for scale)	~0.6 M	~6 ms	~1 kB	~3 kB
ECDSA-P256 sign	~1.2 M	~12 ms	~2 kB	~6 kB
ML-KEM-768 KeyGen	~0.6 M	~6 ms	~3 kB	~12 kB
ML-KEM-768 Encaps	~0.7 M	~7 ms	~3 kB	n/a
ML-KEM-768 Decaps	~0.7 M	~7 ms	~3 kB	n/a
ML-DSA-65 KeyGen	~1.6 M	~16 ms	~40 kB	~18 kB
ML-DSA-65 Sign	~4 M	~40 ms	~50 kB	n/a
ML-DSA-65 Verify	~1.6 M	~16 ms	~35 kB	n/a

The good news nobody expects

ML-KEM-768 encapsulation costs about the same as one X25519 scalar multiplication. Key exchange is not the problem. It never was.

The bad news you will hit first

50 kB of stack for one signature will not fit a part with 64 kB of RAM. Stack-reduced ML-DSA variants get this to roughly 10 kB by streaming, at a cost in speed.

Order of magnitude (from pqm4, mbed-tls), not a datasheet. Numbers vary by two or three times across compilers, implementations and cache behaviour. Measure your own build before you size a part.

The one result that surprises everybody

ML-KEM-768 encapsulation costs roughly one X25519 scalar multiplication

About 0.7 million cycles either way, on a Cortex-M4. Post-quantum key exchange is not slow. It never was.

ML-DSA signing is genuinely more expensive

Around four million cycles, against about 1.2 million for an ECDSA-P256 signature. Still tens of milliseconds, not seconds.

So cycles are not what runs out first

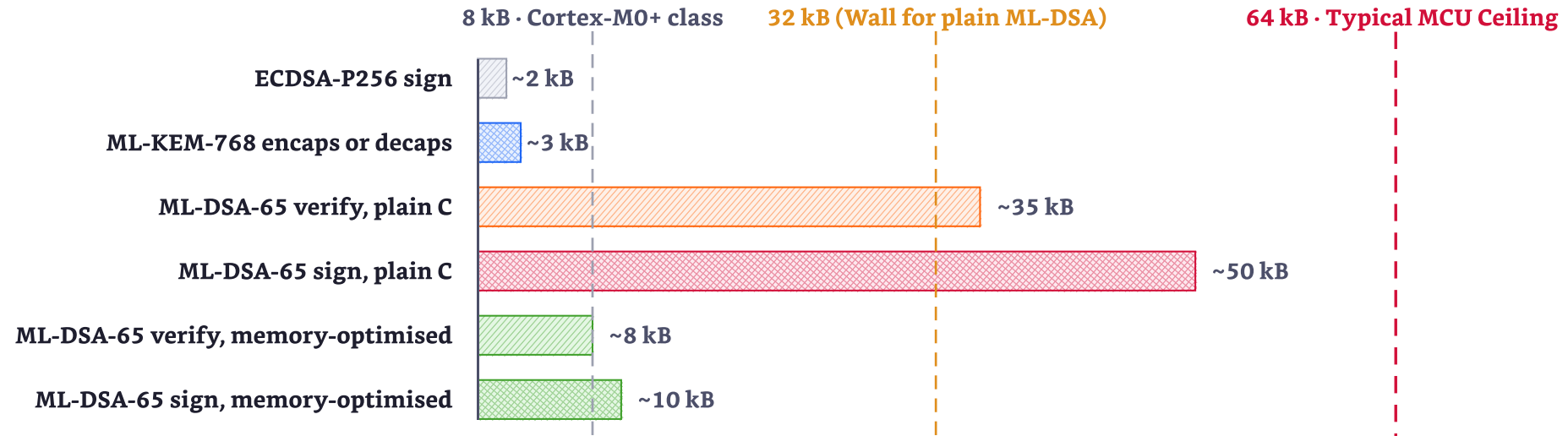
RAM is. Then flash. Then the link. Then, on a battery, energy. Cycles come fourth or fifth.

Which is why the next three slides are not about speed

They are the three budgets that actually decide whether this fits your part.

The RAM wall

Published Cortex-M figures. Cycles were never the problem on a constrained part. This is.



Where the kilobytes go

Expanding the matrix A costs 30 kB for ML-DSA-65: 6×5 polynomials, 256 coefficients each, four bytes a coefficient. For ML-KEM-768 it is 3×3 polynomials of two-byte coefficients, so 4.5 kB.

Everything else is small beside it.

How you get them back

Generate A one polynomial at a time and use it as it comes out of SHAKE: 30 kB becomes **1 kB**, and 4.5 kB becomes 512 B. You pay by re-running SHAKE, so it costs cycles, not RAM. On a constrained part that is the right way round.

The rule this figure exists to give you: on a constrained part, trade cycles for RAM every time. Recompute rather than cache, stream rather than buffer, and never hold a whole expanded matrix in memory.

Fitting it in: what you can actually trade

RAM you can get back

- ▶ Expand A on the fly: avoids buffering all 30 kB of A in RAM.
- ▶ Stream message into SHAKE: pure ML-DSA without 2 MB in RAM.
- ▶ Work in the NTT domain and in place, reusing one buffer.
- ▶ Read public keys straight out of flash (they are read-only).

Flash you can get back

- ▶ Ship one parameter set, not a library that supports six.
- ▶ A verify-only build drops the sampler, the rejection loop and the RNG plumbing.
- ▶ Share SHAKE with whatever else on the part already hashes.

What you pay with

- ▶ Cycles: two to three times slower for the memory-optimised path.
- ▶ Design time: streaming code is harder to get right and harder to keep constant time.
- ▶ Nothing else. There is no security cost to any of this.

None of these is a research result. All of them are in published implementations you can read today.

On a battery device, the bytes cost more than the arithmetic

80 MHz core at 24 mW, BLE at 30 mW and 100 kbps, LoRaWAN SF7 at 132 mW and 5.5 kbps.

What you are paying for	Work	Energy	Times from one CR2032 ¹
ML-KEM-768 encapsulation	0.7 M cycles	0.2 mJ	11 million
ML-DSA-65 verification	1.6 M cycles	0.5 mJ	5 million
ML-DSA-65 signing	4 M cycles	1.2 mJ	2 million
Send a 64 B ECDSA signature over BLE	512 bits	0.2 mJ	16 million
Send a 3309 B ML-DSA signature over BLE	26 kbit	7.9 mJ	300 thousand
Send the same signature over LoRaWAN SF7	26 kbit	640 mJ	3.8 thousand

¹ ~220 mAh/2400 J

The ratio that matters

Signing costs 1.2 mJ of arithmetic.
Moving the signature it produced
costs 7.9 mJ over BLE and **640 mJ**
over LoRaWAN.

Five hundred times the compute.

And airtime is worse than energy

3309 bytes at SF7 is 4.8 seconds on
air, split across 14 uplinks at the
242-byte payload limit. A 1% duty
cycle then forces **eight minutes** of
silence after every signature.

So on a radio-bound device

Pick the scheme by signature size,
not by cycle count. Sign rarely and
verify often. Send a signature once
and cache the result. Consider
whether the device needs to send
one at all.

Order of magnitude, and every term is yours to change. Swap in your own core current, radio and data rate before you quote any of it. The coin-cell column also ignores pulse-current limits, which a 40 mA transmit burst will hit long before the energy runs out.

Adding butterflies: what you get

Butterflies	Cycles for a 256-point NTT	Speed-up	Memory needed	Comment
1	1024 cycles	1×	1 dual-port RAM	smallest core
2	512 cycles	2×	2 banks	sweet spot for MCU-class
4	256 cycles	4×	4 banks	needs careful banking
8	128 cycles	8×	8 banks or wide words	address logic starts to dominate
16	~80 cycles	~12×	wide words + shuffle network	returns fall off
128	~16 cycles	~64×	all coefficients in registers	only for a throughput engine

Why the speed-up stops being linear

Two butterflies need two independent word **pairs** in the same cycle. Four need four. Past a point you are not buying multipliers any more, you are buying a crossbar.

The classic fix is to widen the word instead: pack several coefficients per address and read them together.

And remember what you are speeding up

Even at one butterfly, a 256-point NTT is about 1024 cycles, a few microseconds. A full ML-KEM operation needs a few dozen of those.

If Keccak is still in software and dominates your profile, an eight-way NTT buys you almost nothing. Profile first.

One butterfly core takes 1024 cycles for a 256-point NTT. Two butterflies halve it to 512; four butterflies hit 256 cycles. Beyond four, memory bandwidth dominates.

Where do the twiddle factors come from?

Store them all in ROM

Precompute $\zeta[k] = \zeta^{brv(k)} \bmod q$

ML-DSA: 255×23 bits ≈ 734 bytes
ML-KEM: 127×12 bits ≈ 191 bytes,
plus 128 more for the pointwise step

One read per butterfly. Simple address counter.

Cost: a small ROM or one BRAM.

Generate on the fly

$$\zeta^{k+1} = \zeta^k \cdot \zeta$$

One extra modular multiply per twiddle, chained from the previous one.

Cost: no ROM, but one more multiply in the critical path, and the chain must restart cleanly at every stage boundary.

Errors accumulate silently if it does not.

Hybrid: store the stage roots

$\log_2 n$ roots, derive the rest

Keep the 8 stage-level roots in registers and walk within a stage.

Cost: tiny storage, one multiply, and much simpler restart logic than the fully chained version.

A good default for area-constrained cores.

Do not forget the order

Twiddles are indexed in bit-reversed order. Storing them pre-permuted turns the address generator into a plain counter and removes a bit-reversal unit from the design.

And the sign

The inverse NTT needs ζ^{-k} and a final multiply by n^{-1} . Storing the inverse table costs another ROM; deriving ζ^{-k} from ζ^{n-k} costs nothing but an address flip.

Store all 255 twiddles in ROM, generate them on the fly with chained multipliers, or use a hybrid stage-root register table.

Modular reduction: four choices

Method	Idea	Cost	Applies to	Verdict
Naive divide	quotient and remainder	hardware divider, 20+ cycles	never in a datapath	Correct and useless.
Barrett	multiply by a precomputed $1/q$	2 multipliers, 1 subtract	any q , no scaling	Result is a true residue, so it can be stored and compared.
Montgomery	keep everything in a scaled domain	1 multiply-add, 1 shift	odd q , needs enter / exit	Cheapest per multiply, but the scaling factor must be tracked everywhere.
Plantard / K-RED	signed variant tuned to a specific q	1 multiply, shifts	specific q , small operands	Very cheap for 12-bit ML-KEM; needs care with bounds.
Shift-and-add on a special q	exploit $q = 2^{23} - 2^{13} + 1$	shifts and adds only	ML-DSA's q only	No multiplier in the reduction step at all.

What most designs actually do

Barrett for ML-DSA, because the special shape of q makes the multiply-by- μ step nearly free and the output is a plain residue.

Montgomery inside tight NTT loops where the scaling can be absorbed into the twiddle table.

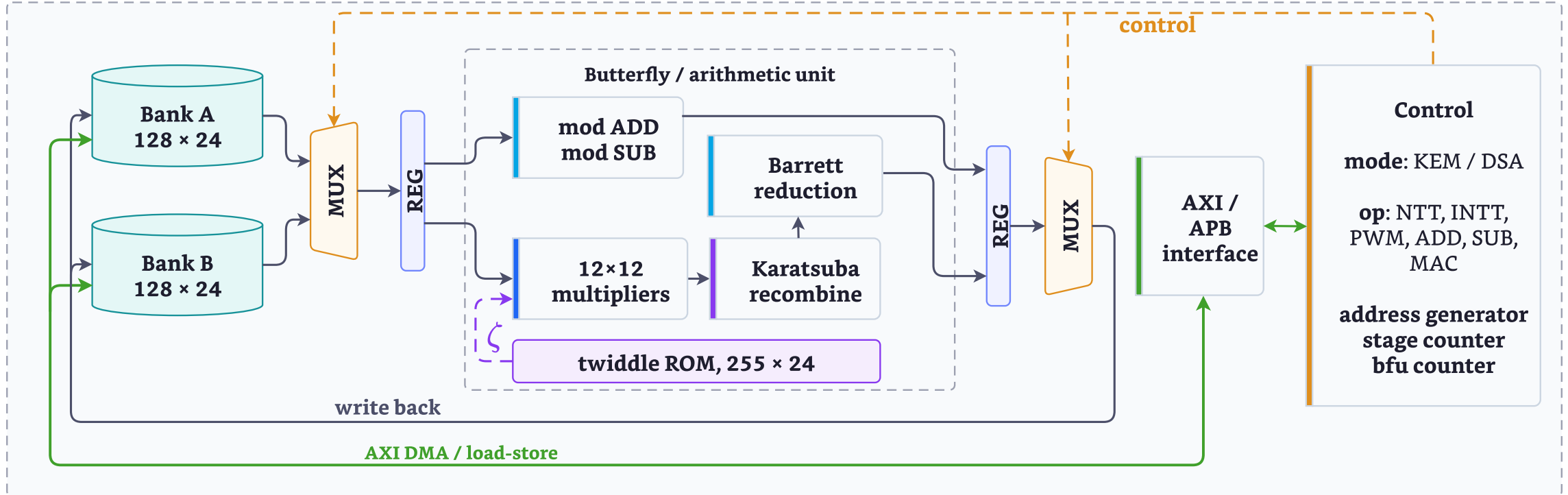
The trap to avoid

Mixing domains. If half your twiddles are Montgomery-scaled and half are not, you get an answer that is wrong by a constant factor and passes small self-tests.

Write the domain into the variable names.

Barrett, Montgomery, shift-and-add, and table-based reduction. The modulus shape decides which one wins.

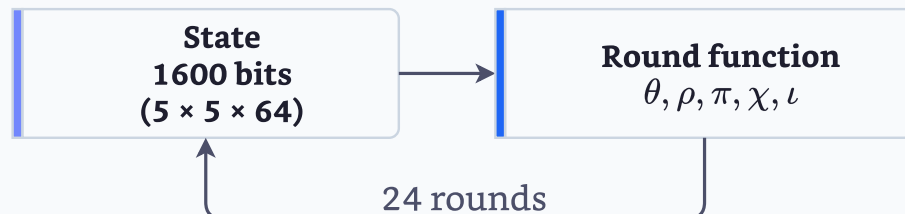
A unified polynomial core



An ML-DSA coefficient is 23 bits; two ML-KEM coefficients are 24. So one 24-bit lane carries either, and one core runs two ML-KEM polynomials at once.

Keccak: the block that decides your throughput

The core



θ parity mixing, ρ bit rotations,
 π lane permutation, χ the only non-linear step

Pick a point on the curve

Unrolling	Cycles per permutation	LUTs	When
1 round / cycle	24	~5–8 k	the usual choice
2 rounds / cycle	12	~10–15 k	if latency matters
1/2 round / cycle	48	~3–4 k	area-first designs
Fully unrolled	1	~50 k+	throughput engines only

Rate is 168 bytes per permutation for SHAKE128 and 136 for SHAKE256. One ML-KEM matrix polynomial needs 473 bytes, so three permutations; one ML-DSA polynomial needs 771, so five.

Where Keccak goes, per ML-KEM-768 encapsulation

SHA3-256 over the 1184-byte encapsulation key: 9
SHA3-512 to derive (K, r) from 64 bytes: 1
SHAKE128 to expand A, 9 polynomials × 3: 27
SHAKE256 for the noise, 7 calls × 1: 7

44 permutations. At 24 cycles each that is about 1 050 cycles, against 6 272 butterflies for the transforms.

The design decision this forces

Accelerate only the NTT and Keccak stays in software, so you keep most of the runtime. Accelerate only Keccak and you remove most of it. So Keccak goes first.

And you may already have one. Many SoCs ship SHA-3 for other reasons. Check before you add a second: a shared Keccak with a good DMA path is often enough.

The result that decides how you size the two blocks

One ML-KEM-768 encapsulation, priced in software and then in hardware. Same work, opposite conclusion.

Block	Work to do	On a Cortex-M4	One butterfly, one Keccak round per cycle
Keccak	44 permutations	thousands of cycles each (25 lanes of 64 bits on a 32-bit core)	24 cycles each ≈ 1 000 cycles total
NTT / INTT	7 transforms = 6 272 butterflies	a handful of cycles each ≈ 50 000 cycles total	1 cycle each ≈ 6 300 cycles total

So the bottleneck moves

In software: hashing dominates

Every published profile of Kyber and Dilithium on a microcontroller puts SHAKE at or near the top.

Therefore: assembly Keccak, or a small SHA-3 block on the bus, is the single biggest win available.

In hardware: polynomial arithmetic dominates

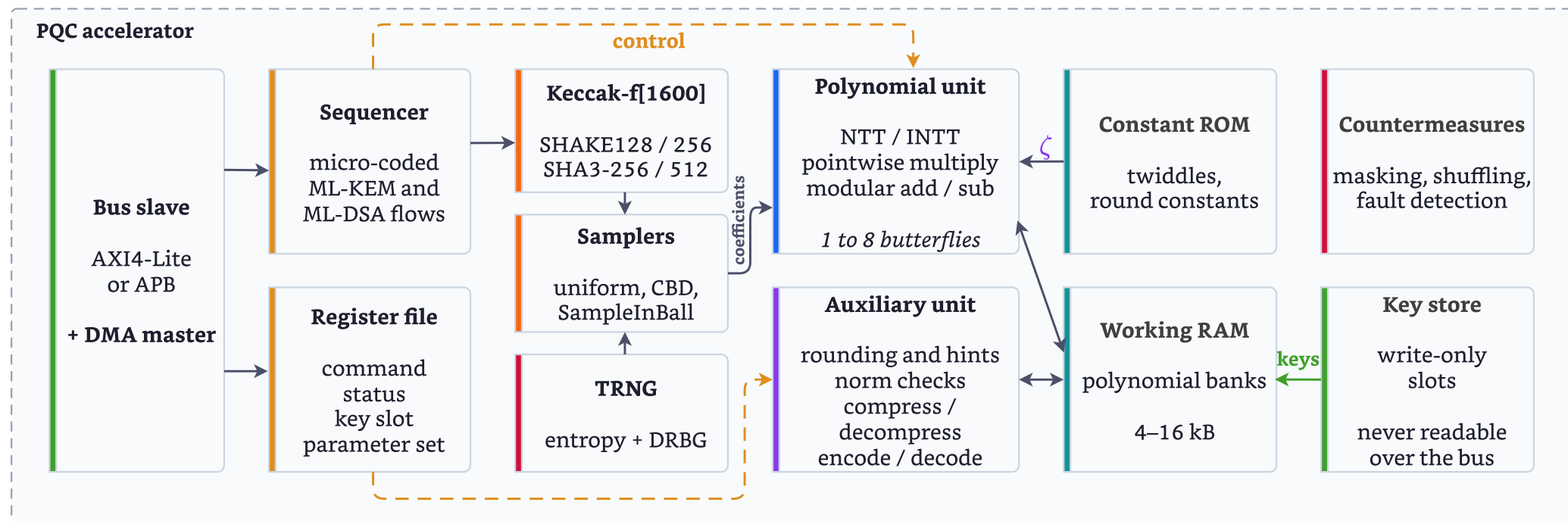
With one butterfly and one Keccak round per cycle, the NTT is 65 % to 95 % of the cycles, and for ML-DSA signing it is over 90 %.

Therefore: once Keccak is a block, spend on butterflies and memory banks, not on unrolling Keccak further.

Why: Keccak is pure XOR and rotate, which a 32-bit CPU does badly and a wire does perfectly. A butterfly needs a real multiplier and two memory ports, and those do not come free.

The practical consequence: size the two blocks against each other, and re-profile after every change.

A complete PQC engine



Three details that separate a demo from a product

1. The key never leaves

Key slots are write-only from the bus. The sequencer can use a key; software can only name it. This is the whole point of putting crypto in hardware.

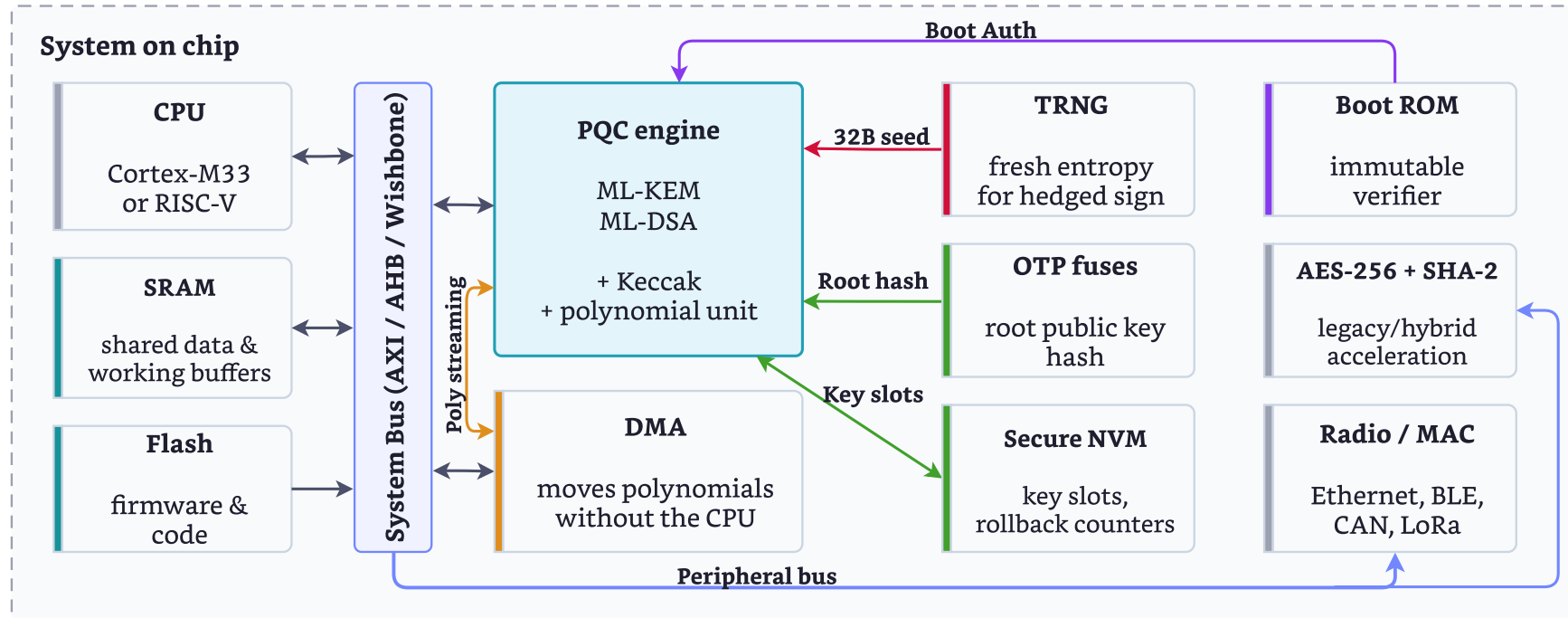
2. The sequencer owns the flow

If software drives every NTT, the bus becomes the bottleneck and the timing becomes data-dependent. Give the block a whole operation and let it finish.

3. Countermeasures are not a late add-on

Masking changes the datapath width, the RAM size and the randomness budget. Retrofitting it means redesigning it. Decide at the start whether this part needs it.

Where it goes in the SoC



Four integration questions worth settling early

How does the key get in?

Provisioned into OTP or secure NVM at manufacture, or derived from a PUF. Not over the bus at runtime.

Who moves the data?

A DMA path, or the CPU will spend more cycles copying polynomials than the engine spends transforming them.

Where does entropy come from?

ML-DSA hedged signing needs 32 fresh bytes per signature. A weak TRNG is a key-recovery bug, not a quality issue.

What does the boot ROM contain?

A verifier you can never change. Choose that algorithm as if the product will still be shipping in 2040.

Three design points

Tiny	Balanced	Fast
<i>A meter, a sensor, a fuse-level root of trust.</i>	<i>Smart cards, secure elements, automotive ECUs, smart locks.</i>	<i>Cloud TLS accelerator, PCIe HSM, high-throughput appliance.</i>
1 butterfly	1–4 butterflies	8–16 butterflies
1/2-round Keccak	1-round Keccak	2-round or unrolled Keccak
single RAM bank	ping-pong banks	multi-bank + crossbar
no masking (public key only)	shuffling + 1st-order masking	masked (HSM) · unmasked (cloud TLS)
verify only (secure boot / RoT)	full ML-KEM and ML-DSA	pipelined, multi-context
Area ~5–10 k LUTs ~30–60 kGE	Area ~15–30 k LUTs ~80–200 kGE	Area ~40–80 k LUTs ~250–500 kGE
Speed ML-DSA verify in ~200 k cycles	Speed ML-KEM encaps in ~10–20 k cycles	Speed ML-KEM encaps in ~2–5 k cycles
Threat model dictates masking: Physical access (smart cards, HSM) requires masking. Cloud servers omit masking for throughput. RoT/boot uses public keys only.		

Ask four questions before you draw a block diagram: how often does the operation happen, do you need signing or only verification, is the part in an attacker's hands, and do you already have a Keccak block?

What leaks, where

Where	What leaks	Through	Consequence	Counter
ML-KEM decaps	The secret polynomial s during pointwise multiply	power / EM	full private key	mask s , shuffle coefficient order
ML-KEM decaps	The re-encryption comparison ($c == c'$)	timing / power	chosen-ciphertext key recovery	constant-time compare, no early exit
ML-KEM decaps	Decoding the message bit by bit	power	message and then key	masked decoder
ML-DSA sign	The nonce y , or the rejection decision	timing / power	signing key from a few signatures	hedged signing, masked rounding
ML-DSA sign	The number of rejection loops	timing	leaks a little about s_1	loop count depends on data by design, so do not leak <i>which</i> check failed
Both	Keccak absorbing secret input	power / EM	the seed, hence everything	masked Keccak, or keep secrets out of the XOF where possible
Both	Rejection sampling on secret data	timing	direct	never do it; the standards avoid it
Both	A fault injected during signing or verify	clock / voltage glitch	signature forgery, or a bypassed boot check	redundant verify, signature check after signing

The rule that covers most of the table: no branch, no memory index and no loop bound may depend on a secret.

The constant-time checklist

No secret-dependent branch
if (secret) ... becomes a masked select.

No secret-dependent memory index
table[secret] leaks through the cache or the address bus.

No secret-dependent loop bound
including “early exit when we already know the answer”.

Compare with a fold, not a loop
 $\text{diff} \mid= a[i] \wedge b[i]$ for all i , then test diff once.

Zero your intermediates
and make sure the compiler cannot remove the memset.

Division and modulo by secrets
some cores have variable-time dividers. Check yours.

Verify with a tool
valgrind / ctgrind, dudect, or formal timing analysis on the RTL.

Then measure on real silicon
a leakage assessment (TVLA) on the actual part, not the model.

The compiler is not on your side. It will turn your careful masked select back into a branch if it thinks that is faster. Check the disassembly of the release build, not the debug build, and re-check after every toolchain update.

Part 5 in five lines

RAM runs out before cycles do

Generate A on the fly, stream the message, and reuse one buffer. Trade cycles for RAM every time.

On a battery, the bytes cost more than the arithmetic

Pick a signature scheme by size, not by speed.

Accelerate Keccak before the NTT

Then measure again, because the bottleneck moves.

Both moduli were chosen to make reduction cheap

Use that. Do not write a generic modular multiplier.

One 24-bit datapath serves both schemes

The difference lives in the control unit, and control bugs are the hard ones.

PART 06

The migration playbook

What to change in your product, and in what order

Four steps, in this order

1

Inventory

Find every place you use public-key cryptography, and every key that is burned in

2

Hybrid the key exchange

Classical and post-quantum together, so you fix the retroactive risk first

3

Design for agility

Algorithm identifiers, dispatch tables, and buffers sized for kilobytes

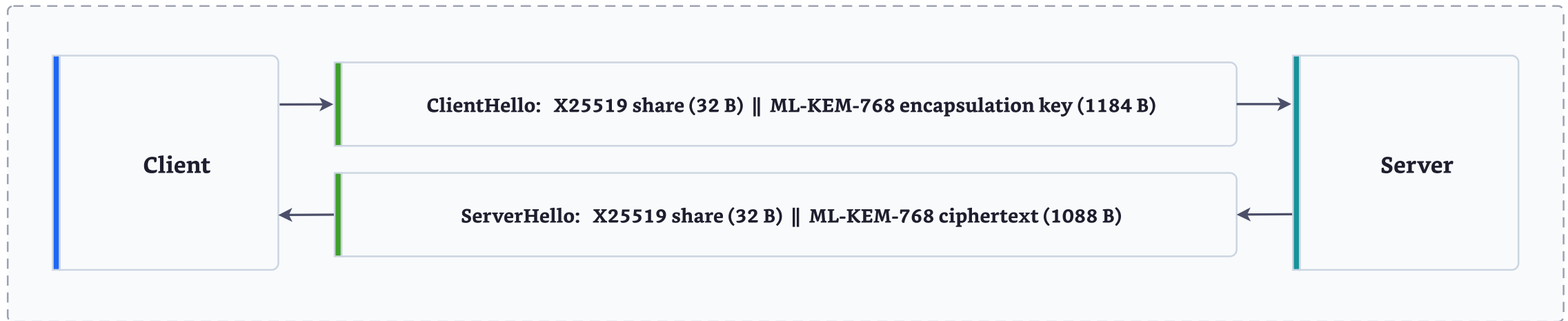
4

Fix secure boot

The earliest deadline and the longest lead time, because it is decided in silicon

The order matters. Step 1 costs nothing and makes the problem finite.

Step 2: hybrid the key exchange



shared secret = KDF(X25519 output || ML-KEM output)

Why both

- ✓ If lattices turn out to be weak, X25519 still protects you against today's attackers.
- ✓ If a quantum computer arrives, ML-KEM still protects you.
- ✓ An attacker has to break both.
- ✓ It is what the deployed internet is doing right now, so interoperability is easy.

What it costs

- About 1.2 kB extra in the first flight and 1.1 kB back. That is the real cost.
- The ClientHello no longer fits one packet, so watch for middleboxes that dislike that.
- Two key exchanges of CPU time, but ML-KEM is about as cheap as X25519, so it roughly doubles a small number.

When to skip the hybrid

- When a policy such as CNSA 2.0 requires pure ML-KEM.
- When the link genuinely cannot carry the extra bytes, but then reconsider whether it can carry pure ML-KEM either.
- Never to save CPU. That is not where the cost is.

Where the verification key lives, and why it may not fit

What you have room for

- ▶ An ECDSA P-256 public key is 64 B, or 512 bits.
- ▶ A user OTP or fuse area is often a few hundred to a few thousand bits, and it is shared with serial numbers, lifecycle state and debug locks.

What you are being asked to store

- ▶ An ML-DSA verification key is 1312, 1952 or 2592 bytes.
- ▶ That is 10 to 20 thousand bits. It does not fit an OTP area, and in mask ROM it cannot ever be changed.
- ▶ LMS and SLH-DSA keys are 32 to 64 B, which is why they suit a fuse-held root.

The pattern that works

- ▶ Put a 32-byte hash of the verification key in OTP.
- ▶ Keep the key itself in flash beside the image, hash-check it, then verify with it.
- ▶ 32 bytes of OTP for any scheme, any size, and you can rotate the key without new silicon.

This one decision, made before tape-out, is the difference between a firmware update and a product recall.

Firmware signing: the three candidates side by side

	LMS / XMSS	ML-DSA	SLH-DSA
Standard	SP 800-208	FIPS 204	FIPS 205
Verification key	32 to 64 B	1312 to 2592 B	32 to 64 B
Signature	1 to 5 kB	2.4 to 4.6 kB	8 to 30 kB
Verifier code	very small	moderate	small
Verifier needs	a hash only	polynomial arithmetic	a hash only
Verifier RAM	~2 kB	~8 to 35 kB	~2 kB
Signer state	must never repeat	stateless	stateless
Signatures per key	finite	unlimited	unlimited
In CNSA 2.0	yes, for firmware	yes, general	not listed

The pattern most designs land on: a tiny hash-based verifier in ROM, which authenticates a loader containing the full ML-DSA implementation for everything above it.

Pick your profile, then stop arguing about it

	Deeply embedded, 8 to 32 kB	Mainstream MCU, 64 to 512 kB	Gateway or MPU
Key exchange	ML-KEM-512 or 768 if you need a session at all; often a pre-shared symmetric key is the right answer	ML-KEM-768, hybrid with X25519	ML-KEM-768 hybrid, straight from the TLS library
Verify on device	LMS or SLH-DSA out of fuses	hash-based root in ROM, ML-DSA-65 above it	ML-DSA-65 with certificates
Sign on device	do not; attest with a symmetric MAC or a secure element	ML-DSA-65, hedged, from a task and not an interrupt	yes, and worry about handshake bytes instead
Implementation	memory-optimised C, one parameter set, A generated on the fly	pqm4-class assembly, or reuse an on-chip SHA-3 block	liboqs, or OpenSSL 3.5 and later
Where to spend	flash and RAM	Keccak throughput	certificate and handshake size

Where the ecosystem actually is

Where	State of play	Verdict
TLS 1.3	Hybrid X25519 + ML-KEM-768 widely deployed	Ready
OpenSSL / BoringSSL	ML-KEM in recent releases; ML-DSA landing	Ready
mbedTLS / wolfSSL	ML-KEM support arriving; check your version	Check
SSH	Hybrid key exchange shipping in OpenSSH	Ready
X.509 certificates	Object identifiers assigned; tooling maturing	Check
IKEv2 / IPsec	Multiple key exchanges defined for PQC	Check
DTLS / CoAP	Follows TLS, but the fragmentation hurts more	Care needed
Firmware signing formats	LMS and XMSS standardised; vendor tooling varies	Check
Hardware security modules	PQC support depends entirely on the vendor and firmware	Ask early
Your own protocol	Nobody has done this for you	Your job

The libraries are ahead of the devices. The blocker is rarely the algorithm.

Ask your HSM and silicon vendors now. Their roadmap sets yours.

A checklist for your next design review

Sizing	Agility	Silicon	Assurance
☐ Key store slots at least 3 kB, not 32 B	☐ An algorithm identifier in every signed object	☐ Keccak / SHA-3 block present or planned	☐ ACVP test vectors run in CI
☐ Signature buffer at least 5 kB	☐ Verify behind a function pointer or table	☐ TRNG good enough for hedged signing	☐ Negative tests: tampered ciphertext, bad lengths
☐ Manifest and header length fields 32-bit	☐ Two algorithms can coexist in one image	☐ Key slots write-only from the bus	☐ Constant-time review of the release build
☐ Flash budget for a 16 kB certificate chain	☐ Nothing hard-codes a curve or a digest length	☐ DMA path for polynomial-sized transfers	☐ Leakage assessment on real silicon
☐ Stack headroom for a 35 kB verify	☐ OTA can replace the crypto layer	☐ ROM verifier chosen for a 15-year life	☐ Fault-injection review of the boot path

If you take one thing from this session, take this slide. None of it requires knowing what a lattice is, and every item on it is cheap today and expensive after tape-out.

Part 6 in five lines

Start with the inventory

It costs nothing and makes the problem finite.

Hybrid the key exchange first

It fixes the retroactive risk without betting on new mathematics.

Design for agility now

It is the difference between an update and a recall.

Put a hash of the verification key in fuses, not the key

32 bytes buys you every scheme and the right to change your mind.

The libraries and protocols are ahead of the devices

You are the blocker, not the algorithm.

Ten things worth remembering

- 1 Only the public-key layer breaks. AES-256 and SHA-2/3 are fine.
- 2 Key exchange is the urgent one: recorded traffic is already at risk.
- 3 ML-KEM-768 encapsulation costs about as much as one X25519.
- 4 The sizes, not the cycles, are what will break your protocol.
- 5 Hashing is over half the runtime. Accelerate Keccak first.
- 6 The NTT is one small datapath and a lot of memory addressing.
- 7 One core can serve both schemes: same n , same ring, different q .
- 8 Signing is variable-time by design; verifying is not.
- 9 Never return an error from Decaps, and never memcmp the check.
- 10 Size your key slots and signature buffers now, not at tape-out.

And the one-line version: the algorithms are ready, the libraries are ready, and the thing most likely to stop you is a buffer size somebody chose in 2015.

Monday morning: start the inventory, and grep your bootloader for the number 64.

One last thought

The mathematics in this session is about thirty years old

The standards are two years old. The libraries work today.

The thing most likely to stop your product from migrating is not a lattice

It is a buffer size, a fuse area, a mask ROM, or a schedule.

All four of those are engineering problems

Which is to say, they are your problems, and they are solvable.

Where to read more

The standards themselves

- ▶ FIPS 203, ML-KEM
- ▶ FIPS 204, ML-DSA
- ▶ FIPS 205, SLH-DSA
- ▶ SP 800-208, LMS and XMSS
- ▶ They are readable, and the pseudocode is complete.

Practical material

- ▶ CNSA 2.0 FAQ, for the dates
- ▶ NIST CMVP and ACVP, for test vectors
- ▶ pqm4, Cortex-M4 cycle and stack numbers you can reproduce
- ▶ liboqs and OpenSSL 3.5, to compare against
- ▶ Start with pqm4 if you want numbers for your own part.

Thank you

Questions, arguments and war stories all welcome.

Bardia Taghavi · PQSecure Technologies

Come and find me at the PQSecure booth if you want to talk about a specific part.