



# PERMAFROST

Post-Quantum Threshold ML-DSA with Dealerless Key Generation

---

Post-quantum Encrypted Ring-based Multi-party Aggregated FROST A complete threshold signing protocol for the post-quantum era

# Table of Contents

---

## 1. Abstract

## 2. Introduction

### 2.1 The Quantum Threat

### 2.2 Why FROST and MuSig2 Die Under Quantum

### 2.3 What is ML-DSA?

### 2.4 The Fundamental Challenge

### 2.5 PERMAFROST: The Solution

## 3. Account-Level Invisible Multisig

### 3.1 Your Account IS the Multisig

### 3.2 On-Chain Indistinguishability

### 3.3 Comparison: Smart Contract vs Account-Level

### 3.4 OPNet Integration

## 4. Comparison with Existing Approaches

### 4.1 vs MuSig2

### 4.2 vs FROST

### 4.3 vs Smart Contract Multisig

### 4.4 Feature Matrix

## 5. Mathematical Foundations

### 5.1 The Polynomial Ring

### 5.2 Number Theoretic Transform

### 5.3 Modular Arithmetic

### 5.4 Decomposition Functions

### 5.5 Sampling Functions

## 6. ML-DSA (FIPS 204) Background

### 6.1 Parameters

### 6.2 Key Generation

### 6.3 Signing Algorithm

### 6.4 Verification

### 6.5 Why Threshold is Hard

## 7. Threshold Construction Overview



- 7.1 Hyperball Masking
- 7.2 Additive Secret Sharing
- 7.3 Combinatorial Approach
- 8. Key Generation Internals
  - 8.1 How Shares Are Constructed
  - 8.2 Gosper's Hack for Subset Enumeration
  - 8.3 deriveUniformLeqEta
- 9. Dealerless DKG: The PERMAFROST Key Ceremony
  - 9.1 Design Decisions
  - 9.2 Hard Invariants
  - 9.3 Prerequisites
  - 9.4 The Four Phases
  - 9.7 Complaint and Abort Protocol
  - 9.8 Security Proofs
- 10. Hyperball Sampling
  - 10.1 The sampleHyperball Algorithm
  - 10.2 Box-Muller Transform
  - 10.3 BigInt Precision
  - 10.4 The C1 Fix
- 11. Share Recovery
- 12. The 3-Round Distributed Signing Protocol
  - 12.1 Round 1: Commitment
  - 12.2 Round 2: Reveal
  - 12.3 Round 3: Partial Response
  - 12.4 The H1 Timing Fix
- 13. Combine: Signature Finalization
- 14. Polynomial Packing: 23-Bit Encoding
- 15. Parameter Tables
- 16. Security Analysis and Hardening
- 17. Frontend Integration: Communication Layer
- 18. Float64 Precision Analysis
- 19. Architecture and Implementation



## 20. API Reference

## 21. Known Limitations

### A. Notation Reference

### B. Test Coverage

# 1. Abstract

PERMAFROST (Post-quantum Encrypted Ring-based Multi-party Aggregated FROST) is a complete threshold signing protocol for the post-quantum era. Built on ML-DSA (FIPS 204), the NIST-standardized lattice-based digital signature scheme, PERMAFROST enables T-of-N parties to collaboratively produce a standard FIPS 204 signature without any single party holding the full signing key.

The protocol consists of two major components: (1) a dealerless distributed key generation (DKG) ceremony where N parties jointly create an ML-DSA key pair with no trusted dealer, and (2) a 3-round distributed signing protocol where any T parties independently generate their own randomness, exchange only commitments and responses, and produce a single standard FIPS 204 signature that is byte-for-byte indistinguishable from a single-signer signature.

Key properties of PERMAFROST include: post-quantum security based on Module-LWE and Module-SIS hardness assumptions; account-level invisible multisig where the threshold key is indistinguishable from any standard ML-DSA public key; structural secrecy where no single party ever possesses the full secret key, even during key generation; standard signature output of 2,420 bytes (ML-DSA-44), 3,309 bytes (ML-DSA-65), or 4,627 bytes (ML-DSA-87) regardless of the threshold configuration; and support for all three NIST security levels (128-bit, 192-bit, 256-bit).

PERMAFROST is the post-quantum successor to FROST (Flexible Round-Optimized Schnorr Threshold Signatures). Where FROST relies on the discrete logarithm problem and Schnorr signatures — both of which fall to quantum computers running Shor's algorithm — PERMAFROST rebuilds the same threshold functionality on lattice-based mathematics that quantum computers cannot crack. The protocol supports  $2 \leq T \leq N \leq 6$  with empirically optimized parameters for all 45 valid (T, N, security level) configurations.

While developed for OPNet's account-model blockchain, PERMAFROST is chain-agnostic. Any blockchain that adopts ML-DSA for post-quantum signatures — including Bitcoin itself when it inevitably migrates beyond ECDSA/Schnorr — can use PERMAFROST for invisible threshold multisig. The output is always a standard FIPS 204 signature. No protocol changes, no new opcodes, no consensus modifications required on the verifying chain.

## LIVE DEMO

An interactive browser demo of the full PERMAFROST protocol (DKG + signing + verification) is available at:

<https://ipfs.opnet.org/ipfs/bafybeiejz5fc44scvp5qbgz4zzeg3jieccd5t7dwr6o3dqqhbk54czlhia/>

## 2. Introduction

---

### 2.1 The Quantum Threat to Threshold Signatures

The development of large-scale quantum computers poses an existential threat to the cryptographic foundations underlying all modern digital signature schemes. Shor's algorithm, when executed on a sufficiently powerful quantum computer, can solve the discrete logarithm problem and the integer factorization problem in polynomial time. These two mathematical problems form the security basis for ECDSA, EdDSA, Schnorr signatures, RSA, and every threshold signature scheme built on top of them.

The implications for multi-party custody are severe. Every blockchain multisig system deployed today — Bitcoin's MuSig2, Ethereum's Gnosis Safe, Cosmos's on-chain multisig, OPNet's opMultisig — relies on elliptic curve cryptography (specifically secp256k1 or Ed25519) that a quantum computer can break. When quantum computers of sufficient scale arrive, the security of billions of dollars in multi-party custody arrangements will collapse simultaneously.

NIST recognized this threat and spent eight years (2016–2024) evaluating post-quantum cryptographic algorithms. The result was FIPS 204, standardizing ML-DSA (Module Lattice-Based Digital Signature Algorithm, formerly CRYSTALS-Dilithium) as the primary post-quantum signature scheme. ML-DSA's security is based on the Module Learning With Errors (MLWE) and Module Short Integer Solution (MSIS) problems — lattice problems for which no efficient quantum algorithm is known.

### 2.2 Why FROST and MuSig2 Die Under Quantum

FROST (Flexible Round-Optimized Schnorr Threshold, RFC 9591) and MuSig2 (BIP-327) are the state-of-the-art threshold and multi-signature schemes for pre-quantum cryptography. Both are elegant protocols that exploit a fundamental property of Schnorr signatures: linearity.

In Schnorr signatures, the signature equation is  $s = r + c \cdot x$ , where  $r$  is the nonce,  $c$  is the challenge, and  $x$  is the secret key. This equation is perfectly linear: if multiple parties each hold a share  $x_i$  such that  $\sum x_i = x$ , they can each compute  $s_i = r_i + c \cdot x_i$  and sum the partial signatures to get  $s = \sum s_i = \sum r_i + c \cdot \sum x_i = r + c \cdot x$ . Threshold signing over Schnorr is almost trivial because of this linearity.

Both FROST and MuSig2 depend entirely on the hardness of the discrete logarithm problem over elliptic curves. Shor's algorithm solves this problem in  $O(n^3)$  time on a quantum computer, where  $n$  is the bit length of the group order. For secp256k1 (256-bit), this means a quantum computer with approximately 2,500 logical qubits could break these schemes in hours. While such quantum computers do not exist today, the trajectory of quantum computing research makes their eventual arrival a matter of "when," not "if."

### ANALOGY

If FROST is a house of cards built on mathematical assumptions that quantum computers can blow down, PERMAFROST is the same house rebuilt in reinforced concrete — same shape, same function, but the foundation is lattice-based mathematics that quantum computers cannot crack.

## 2.3 What is ML-DSA (FIPS 204)?

ML-DSA (Module Lattice-Based Digital Signature Algorithm) is the NIST-standardized post-quantum signature scheme, published as FIPS 204 in August 2024. It is based on the CRYSTALS-Dilithium scheme, which was selected after eight years of rigorous evaluation by the global cryptographic community. ML-DSA operates in the polynomial ring  $R_q = \mathbb{Z}_q[X]/(X^{256} + 1)$  where  $q = 8,380,417$ , using the "Fiat-Shamir with Aborts" paradigm.

ML-DSA provides three security levels: ML-DSA-44 (NIST Level 2, ~128-bit security), ML-DSA-65 (NIST Level 3, ~192-bit security), and ML-DSA-87 (NIST Level 5, ~256-bit security). The signature sizes are 2,420 bytes, 3,309 bytes, and 4,627 bytes respectively — significantly larger than Schnorr (64 bytes) but still practical for blockchain applications.

## 2.4 The Fundamental Challenge: Non-Linearity

The fundamental challenge in constructing threshold ML-DSA is that ML-DSA is non-linear. Unlike Schnorr, where the signature equation  $z = y + c \cdot s$  is perfectly linear, ML-DSA's signing process involves multiple non-linear operations that resist naive secret sharing:

1. **Rejection sampling:** Standard ML-DSA rejects approximately 75% of signing attempts. The masking vector  $y$  must hide the secret  $s_1$  via  $z = y + c \cdot s_1$ , but the norm check  $\|z\|_\infty < \gamma_1 - \beta$  depends on both  $y$  and the secret. In a threshold setting, parties must coordinate their rejection — if one party rejects, all must restart.
2. **Decomposition functions:** HighBits, LowBits, MakeHint, and UseHint are all non-linear operations that cannot be computed independently on shares and then combined. These functions involve division, modular centering, and conditional logic that break under additive sharing.
3. **Lattice structure:** The masking vector  $y$  must be jointly generated without any party learning the full  $y$ . Naive approaches (each party samples uniformly, then sum) do not produce the correct distribution for the sum to pass ML-DSA's norm checks.

The "Threshold Signatures Reloaded" paper by Borin, Celi, del Pino, Espitau, Niot, and Prest (2025) solves these problems through a breakthrough construction: hyperball sampling. Instead of sampling  $y$  uniformly, each party independently samples a point on a high-dimensional hypersphere, and the sum of these samples has the correct statistical properties to pass all norm checks after rounding.

## 2.5 PERMAFROST: The Complete Solution

PERMAFROST combines three components into a complete post-quantum threshold signing system:

1. Dealerless Distributed Key Generation (DKG): A 4-phase protocol where N parties jointly create an ML-DSA key pair. No trusted dealer is required. Each party's share is derived from jointly-committed entropy via per-bitmask seed derivation, ensuring structural secrecy, forced randomness, and equivocation-immunity.
2. 3-Round Distributed Signing Protocol: Any T parties independently generate their own randomness (on their own machine, disconnected), exchange commitments and responses over any communication channel, and produce a single standard FIPS 204 signature. No party's secret ever leaves their device.
3. Security Hardening: All five audit findings from the Go reference implementation (C1, C8, H1, H6, M7) have been addressed, plus additional hardening including duplicate ID detection, constant-time commitment verification, state destruction classes, and failure-path cleanup.

### KEY PROPERTY

PERMAFROST produces standard FIPS 204 signatures that any ML-DSA verifier can validate. The signature size is ALWAYS the standard size (2,420 / 3,309 / 4,627 bytes) regardless of the threshold configuration (T, N). A verifier cannot distinguish a PERMAFROST threshold signature from a single-signer ML-DSA signature.

## 3. Account-Level Invisible Multisig

### 3.1 Your Account IS the Multisig

In account-model blockchains like OPNet, every account is identified by its ML-DSA public key. With threshold ML-DSA, the public key can be a threshold key — one that requires  $T$ -of- $N$  parties to sign — while looking identical to any other ML-DSA public key on-chain. This is not a smart contract multisig. This is the account itself being a multisig.

The distinction is fundamental. Smart contract multisig operates at the application layer: a contract receives  $N$  individual signatures, verifies each one on-chain, counts that at least  $T$  are valid, then executes the transaction. Account-level threshold operates below the contract layer:  $T$  parties coordinate off-chain to produce a single signature, and the blockchain sees one address, one signature, one transaction — identical to any other account.

#### ANALOGY

Think of smart contract multisig as a lockbox where everyone puts their key in separate slots visibly on camera. Account-level threshold is more like a single lock that secretly requires multiple people to turn simultaneously — but from the outside, it looks and works like any other lock.

### 3.2 On-Chain Indistinguishability

This is the critical property: no on-chain observer can determine whether a signature was produced by a single signer or by a threshold group. The threshold configuration ( $T$ ,  $N$ ) is completely invisible on-chain.



Property	Single-Signer ML-DSA	PERMAFROST Threshold
Public key size	1,312 / 1,952 / 2,592 bytes	Identical
Signature size	2,420 / 3,309 / 4,627 bytes	Identical
Verification algorithm	Standard FIPS 204	Identical
On-chain format	Standard encoding	Identical
tx.origin	Signer's public key	Aggregated threshold key
Distinguishable?	N/A	NO — mathematically identical

### 3.3 Comparison: Smart Contract vs Account-Level

Capability	Smart Contract Multisig	Account-Level Threshold
Contract deployer	Single key (SPOF)	Threshold group (no single compromise)
Token admin keys	N-of-N via on-chain logic	N-of-N via aggregated key — invisible
Transaction signing	Multiple sigs verified on-chain	One signature, zero overhead
tx.origin	One signer's address	Aggregated identity
Visibility	On-chain — anyone sees multisig	Invisible — looks like normal account
Gas cost	Verifying N signatures + logic	Verifying 1 signature (standard)
Scope	Only supported operations	EVERY operation
Key custody	N independent keys	N shares; secret key never exists in one place

## 3.4 OPNet Integration

OPNet uses an account model where every account is identified by its ML-DSA key. With PERMAFROST, the account itself becomes the multisig — not at the contract level, not at the application level, but at the account level. Below the contract. Below everything.

This works for every operation on OPNet: deployments, token transfers, admin calls, parameter changes, contract upgrades — everything. Because the multisig lives at the account level, every transaction from that address inherits threshold protection automatically. No smart contract logic needed. No on-chain multisig overhead. Just one address, one signature.

### TYPESCRIPT

```
1 // Corporate treasury: 4-of-6 dealerless key ceremony
2 const th = ThresholdMLDSA.create(87, 4, 6); // NIST Level 5, maximum security
3 const sessionId = randomBytes(32);
4
5 // Phase 1: Each board member commits entropy on their own device
6 const p1 = th.dkgPhase1(myPartyId, sessionId);
7 // broadcast p1.broadcast to all parties...
8
9 // Phase 2: Reveal entropy, derive shared secrets per-bitmask
10 const p2 = th.dkgPhase2(myPartyId, sessionId, p1.state, allPhase1);
11 // broadcast p2.broadcast, send p2.privateToHolders via encrypted channel...
12
13 // Phase 2 Finalize + Phase 3: Verify, derive shares, distribute masks
14 const p2f = th.dkgPhase2Finalize(myPartyId, sessionId, p1.state,
15     allPhase1, allPhase2, receivedReveals);
16 // send mask pieces via encrypted channel...
17
18 // Phase 4: Aggregate masks, derive public key
19 const p4 = th.dkgPhase4(myPartyId, setup.bitmasks, p2f.generatorAssignment,
20     receivedMasks, p2f.ownMaskPieces);
21 // broadcast p4...
22
23 // Finalize: everyone computes the same public key
24 const { publicKey, share } = th.dkgFinalize(myPartyId, p2f.rho, allPhase4, p2f.shares);
25 // publicKey -> OPNet/Bitcoin address. Nobody ever saw the full secret.
26 // share -> stored on this device only. 4 of 6 needed to sign.
```

Why this matters for post-quantum — and for Bitcoin: Every chain's multisig today is built on pre-quantum cryptography. Bitcoin MuSig2: Schnorr-based — dead when quantum arrives. Ethereum Safe: ECDSA-based — dead. Cosmos multisig: secp256k1 — dead.



PERMAFROST is the first account-level threshold multisig that survives quantum computers, based on a NIST-standardized algorithm, producing signatures any ML-DSA verifier can validate. This is not limited to OPNet — any blockchain that adopts ML-DSA can use PERMAFROST for invisible threshold signing. Bitcoin itself, when it eventually migrates to post-quantum signatures, will need exactly this: aggregated ML-DSA multisig where the output is a single standard signature indistinguishable from a solo signer. PERMAFROST is that solution.

## 4. Comparison with Existing Approaches

---

### 4.1 vs MuSig2 (Bitcoin Schnorr)

MuSig2 is the state-of-the-art for Bitcoin Schnorr key aggregation (N-of-N). It achieves elegant 2-round signing by exploiting Schnorr's linearity. However, MuSig2 is fundamentally limited: it only supports N-of-N (all parties must sign), and it relies entirely on the discrete logarithm problem over secp256k1, which Shor's algorithm destroys.

### 4.2 vs FROST (Schnorr Threshold)

FROST (RFC 9591) extends Schnorr aggregation to T-of-N thresholds using Shamir secret sharing. It achieves 2-round signing with beautiful simplicity. But like MuSig2, FROST's security depends entirely on the discrete logarithm problem. PERMAFROST is the conceptual successor to FROST: same T-of-N threshold signing, same invisible on-chain signature, but built on post-quantum lattice foundations.

### 4.3 vs Smart Contract Multisig

Smart contract multisig (Gnosis Safe, opMultisig, etc.) operates at the application layer. It verifies N individual signatures on-chain, costs  $O(N)$  gas, is visible to everyone as multisig, and only works for operations the contract explicitly supports. PERMAFROST operates below the contract layer with  $O(1)$  verification cost and universal scope.

## 4.4 Feature Matrix

Property	MuSig2	FROST	SC Multisig	PERMAFROST
Threshold	N-of-N only	T-of-N	T-of-N	T-of-N
Signing rounds	2	2	1 (on-chain)	3
Quantum-safe	No	No	No	YES (FIPS 204)
Signature size	64 bytes	64 bytes	$N \times \text{sig}$	2,420-4,627 bytes
On-chain visible	No	No	Yes	No
Verification cost	$O(1)$	$O(1)$	$O(N)$	$O(1)$
Math basis	Discrete log	Discrete log	Varies	MLWE + MSIS
Standard	BIP-327	RFC 9591	N/A	FIPS 204
Scope	Signing	Signing	Limited	ALL operations
DKG	None needed	2-round	N/A	4-phase dealerless

### WHY 3 ROUNDS?

Why 3 rounds instead of 2? ML-DSA's non-linearity (rejection sampling, decomposition) requires stronger commitment binding than Schnorr's linear structure. The commit-then-reveal pattern prevents adaptive manipulation of the combined commitment. The 3rd round also enables commitment hash verification, catching cheating parties before they can corrupt the signature.

## 5. Mathematical Foundations

### 5.1 The Polynomial Ring $R_q$

All arithmetic in PERMAFROST operates in the polynomial ring:

$$R_q = \mathbb{Z}_q[X] / (X^{256} + 1)$$

where  $q = 8,380,417$  (the Dilithium prime, 23 bits:  $2^{23} - 2^{13} + 1$ ) and  $N = 256$  (polynomial degree). Elements of  $R_q$  are polynomials of degree  $\leq 255$  with coefficients in  $\mathbb{Z}_q = \{0, 1, \dots, q-1\}$ .

This ring is chosen for three critical reasons: (1)  $q \equiv 1 \pmod{2N}$ , enabling efficient Number Theoretic Transform; (2)  $X^{256} + 1$  is the 512th cyclotomic polynomial, irreducible over  $\mathbb{Z}$ , ensuring the ring has the correct algebraic structure; and (3) the resulting ring is isomorphic (via NTT) to  $\mathbb{Z}_q^{256}$ , making pointwise multiplication possible in  $O(N \log N)$  time instead of  $O(N^2)$ .

#### ANALOGY

The polynomial ring is like a clock face, but instead of 12 hours it has 8,380,417 positions, and instead of one hand it has 256 hands all spinning simultaneously. Every operation in PERMAFROST happens on this multi-handed clock.

### 5.2 Number Theoretic Transform (NTT)

The NTT is the finite-field analogue of the Fast Fourier Transform. It maps polynomials from their coefficient representation to their evaluation representation:

$$\text{NTT}: R_q \rightarrow \mathbb{Z}_q^{256}$$

Given a primitive 512th root of unity  $\zeta = 1753$  in  $\mathbb{Z}_q$  (i.e.,  $\zeta^{512} \equiv 1 \pmod{q}$  and  $\zeta^{256} \equiv -1 \pmod{q}$ ), the NTT of a polynomial  $a$  is:

$$\text{NTT}(a)[i] = \sum_{j=0}^{255} a[j] \cdot \zeta^{\{\text{brv}(i) \cdot (2j+1)\}} \pmod{q}$$

where  $\text{brv}(i)$  is the 8-bit bit-reversal of  $i$ . In NTT domain, polynomial multiplication is pointwise:  $\text{NTT}(a \cdot b) = \text{NTT}(a) \cdot \text{NTT}(b)$ . This is critical for threshold signing performance, as computing  $A \cdot z$ ,  $c \cdot s$ , etc. requires many polynomial multiplications.

Implementation note: Since  $q$  is 23 bits, the product  $a[i] \cdot b[i]$  is at most 46 bits. JavaScript's Number type provides 53 bits of integer precision (IEEE 754 double), so all NTT arithmetic stays within safe integer range. This avoids the need for BigInt in ring operations — a critical performance decision.

## 5.3 Modular Arithmetic

Two modular reduction functions are used throughout:

$\text{mod}(a, q)$ : Standard unsigned reduction to  $[0, q)$ :  $\text{mod}(a) = ((a \% q) + q) \% q$

$\text{smod}(a, q)$ : Centered/signed reduction to  $[-(q-1)/2, (q-1)/2]$ :  $\text{smod}(a) = \text{mod}(a + (q-1)/2) - (q-1)/2$

The centered form is essential for norm checks (e.g., `polyChknorm`) where we need to test if coefficients are "small" — checking  $|\text{smod}(a[i])| < B$  requires centering around zero.

## 5.4 Decomposition Functions

ML-DSA uses several decomposition functions parameterized by  $\gamma_2$  (`gamma_2`):

`Power2Round(r)`: Splits  $r$  into high and low parts relative to  $2^d$  ( $d=13$ ).  $r_0 = \text{smod}(r, 2^d)$ ,  $r_1 = (r - r_0) / 2^d$ . Used to split the public key:  $t = t_1 \cdot 2^d + t_0$ .

`Decompose(r)`: Splits  $r$  relative to  $2\gamma_2$ .  $r_0 = \text{smod}(r, 2\gamma_2)$ ,  $r_1 = (r - r_0) / (2\gamma_2)$ . Special case: if  $r - r_0 = q-1$ , return  $(r_1=0, r_0=r_0-1)$ .

$\text{HighBits}(r) = \text{Decompose}(r).r_1$ ,  $\text{LowBits}(r) = \text{Decompose}(r).r_0$ .

`MakeHint(z, r)`: Returns 1 if adding  $z$  to  $r$  changes `HighBits`. `UseHint(h, r)`: Adjusts `HighBits` according to hint  $h$ . These are central to how ML-DSA achieves signature compression.

## 5.5 Sampling Functions

`RejNTTPoly(xof)`: Samples a uniformly random polynomial in NTT domain by rejection sampling from SHAKE-128. Each 3-byte block yields a 23-bit candidate; values  $\geq q$  are rejected.

`RejBoundedPoly(xof)`: Samples a polynomial with small coefficients in  $[-\eta, \eta]$  by rejection sampling from SHAKE-256. For  $\eta=2$ , each nibble yields a candidate via  $\eta - (t \bmod 5)$  for  $t < 15$ .

`SampleInBall(seed)`: Samples a sparse polynomial  $c \in \mathbb{R}_q$  with exactly  $\tau$  nonzero coefficients, each  $\pm 1$ . This is the challenge polynomial in the Fiat-Shamir transform.

## 6. ML-DSA (FIPS 204) Background

### 6.1 Standard Parameters

Parameter	ML-DSA-44	ML-DSA-65	ML-DSA-87
NIST Level	2 (128-bit)	3 (192-bit)	5 (256-bit)
K (rows of A)	4	6	8
L (columns of A)	4	5	7
$\eta$ (secret bound)	2	4	2
$\tau$ (challenge weight)	39	49	60
$\gamma_1$ (masking range)	$2^{17}$	$2^{19}$	$2^{19}$
$\gamma_2$ (decomposition)	$(q-1)/88$	$(q-1)/32$	$(q-1)/32$
$\omega$ (hint weight)	80	55	75
$\beta = \tau \cdot \eta$	78	196	120
d (Power2Round)	13	13	13
c bytes	32	48	64
Public key bytes	1,312	1,952	2,592
Signature bytes	2,420	3,309	4,627

### 6.2 Key Generation

Standard ML-DSA key generation proceeds as follows:

1. Sample  $\rho$  (public seed, 32 bytes),  $\rho'$  (secret seed),  $K$  (signing key) from random seed via SHAKE-256.



2. Expand  $A \in \mathbb{R}_q^{K \times L}$  from  $\rho$  using XOF-128 (SHAKE-128).
3. Sample  $s_1 \in \mathbb{R}_q^L$ ,  $s_2 \in \mathbb{R}_q^K$  with small coefficients (bound  $\eta$ ) via rejection sampling.
4. Compute  $t = A \cdot s_1 + s_2$  in  $\mathbb{R}_q^K$ .
5. Split  $t$  into  $(t_1, t_0)$  via Power2Round.
6. Public key:  $pk = \text{Encode}(\rho, t_1)$ . Secret key:  $sk = (\rho, K, tr, s_1, s_2, t_0)$  where  $tr = \text{SHAKE-256}(pk)$ .

## 6.3 Signing Algorithm (Rejection Sampling Loop)

The ML-DSA signing algorithm uses a "Fiat-Shamir with Aborts" paradigm, where most signing attempts are rejected to ensure the signature does not leak information about the secret key:

1. Compute message representative:  $\mu = \text{SHAKE-256}(tr \parallel M)$ .
2. Derive private randomness:  $\rho' = \text{SHAKE-256}(K \parallel \text{rnd} \parallel \mu)$ .
3. Sample masking vector  $y \in \mathbb{R}_q^L$  with coefficients in  $[-\gamma_1+1, \gamma_1]$ .
4. Compute  $w = A \cdot y$ , then  $w_1 = \text{HighBits}(w)$ .
5. Compute challenge:  $c = \text{SHAKE-256}(\mu \parallel \text{W1Encode}(w_1))$ ,  $c = \text{SampleInBall}(c)$ .
6. Compute response:  $z = y + c \cdot s_1$ .
7. Reject if  $\|z\|_\infty \geq \gamma_1 - \beta$  or  $\|\text{LowBits}(w - c \cdot s_2)\|_\infty \geq \gamma_2 - \beta$ .
8. Compute hint  $h = \text{MakeHint}(-c \cdot t_0, w - c \cdot s_2 + c \cdot t_0)$ . Reject if  $\text{weight}(h) > \omega$ .
9. Output signature:  $\sigma = (c, z, h)$ .

## 6.4 Verification

Verification is straightforward: decode public key  $(\rho, t_1)$  and signature  $(c, z, h)$ , check  $\|z\|_\infty < \gamma_1 - \beta$ , compute  $c = \text{SampleInBall}(c)$ ,  $w'_{\text{approx}} = A \cdot z - c \cdot t_1 \cdot 2^d$ ,  $w'_1 = \text{UseHint}(h, w'_{\text{approx}})$ , then verify  $c = \text{SHAKE-256}(\mu \parallel \text{W1Encode}(w'_1))$ .

## 6.5 Why Threshold is Hard for ML-DSA

Unlike Schnorr/ECDSA where threshold signing is relatively straightforward (linear secret sharing + linear signature equation), ML-DSA has three fundamental obstacles:



### FUNDAMENTAL OBSTACLES

1. REJECTION SAMPLING: ~75% of signing attempts are rejected. With threshold signing, parties must coordinate their rejection — if one party rejects, all must restart.
2. NON-LINEAR OPERATIONS: Decomposition functions (HighBits, LowBits, MakeHint) cannot be computed independently on shares and then combined.
3. JOINT MASKING: The masking vector  $y$  must be jointly generated without any party learning the full  $y$ , while still producing a valid distribution for the sum.

The "Threshold Signatures Reloaded" paper solves these using: hyperball sampling (replace uniform  $y$  with a continuous sphere distribution), additive secret sharing over bitmasks, and parallel iterations ( $K_{\text{iter}}$  independent attempts per round to amortize rejection).



## 7. Threshold Construction Overview

### 7.1 High-Level Protocol Flow

The PERMAFROST protocol has two major phases: key generation (one-time setup) and distributed signing (per transaction). Key generation produces a standard ML-DSA public key and  $N$  threshold key shares. Distributed signing uses any  $T$  shares to produce a standard FIPS 204 signature via three rounds of communication.

### 7.2 The Hyperball Masking Breakthrough

Instead of sampling  $y$  uniformly from  $[-\gamma_1+1, \gamma_1]^{\{N \cdot L\}}$  as in standard ML-DSA, the threshold protocol:

1. Samples a point on a high-dimensional hypersphere (radius  $r'$ ) using Box-Muller + normalization.
2. Rounds the continuous samples to get integer vectors  $(y, e)$ .
3. Computes  $w = A \cdot y + e$  (instead of  $w = A \cdot y$ ).

This continuous sampling enables additive sharing: each party samples independently on the hypersphere, and the sum of their samples is "well-shaped" enough to pass the norm checks after rounding. The error term  $e$  absorbs the rounding differences.

#### ANALOGY

Imagine you need to bake a cake, but no single person is allowed to know the full recipe. Hyperball sampling is like having each baker add ingredients by feel — measured from a carefully calibrated internal compass — such that the combined result always tastes correct, even though no baker knew the exact quantities anyone else added.

### 7.3 Additive Secret Sharing over Bitmasks

The secret key  $(s_1, s_2)$  is split into additive shares indexed by bitmasks. For a  $(T, N)$  configuration, we enumerate all  $C(N, N-T+1)$  subsets of  $(N-T+1)$  potentially honest signers. Each subset corresponds to a bitmask, and for each bitmask, a separate additive share is generated.

Each party stores shares for all subsets they belong to. A "share recovery" algorithm combines the right shares based on which  $T$  parties are active. This combinatorial approach ensures that any  $T$  parties can sign, even though the underlying sharing is additive (not Shamir-based).



For example, in a 3-of-5 configuration:  $C(5, 3) = 10$  honest-signer subsets. Each party belongs to  $C(4, 2) = 6$  of these subsets. The total secret  $s_1 = \sum s_1^{\{hs\}}$  over all 10 subsets. When 3 specific parties want to sign, the share recovery algorithm selects the right combination of stored shares for each party.



## 8. Key Generation Internals

### 8.1 How Shares Are Constructed (3-of-5 Example)

This section explains the mathematics of how threshold key shares are constructed. The dealerless DKG (Section 9) distributes this process across all parties so no single entity ever sees the full picture. Understanding the underlying math is essential for understanding why the DKG is secure.

Step 1: Seed generation. The dealer generates a 32-byte random seed. This is the only source of randomness — everything else is deterministic from this seed.

Step 2: SHAKE-256 expansion. The seed is fed into  $\text{SHAKE-256}(\text{seed} \parallel K \parallel L)$ . First 32 bytes of output  $\rightarrow p$  (public randomness for matrix A). Then  $5 \times 32$  bytes  $\rightarrow$  one key per party (per-party randomness).

Step 3: Matrix A expansion. A is a  $4 \times 4$  matrix of polynomials (for ML-DSA-44). Each polynomial has 256 coefficients in  $[-q, q]$ . Expansion uses SHAKE-128 with rejection sampling.

Step 4: Gosper's hack enumerates all subsets. For  $T=3$ ,  $N=5$ , we need  $N-T+1 = 3$  signers in each honest subset.  $C(5,3) = 10$  such subsets, each represented as a 5-bit bitmask.

### 8.2 Gosper's Hack for Subset Enumeration

Gosper's hack iterates through all bitmasks with exactly  $k$  bits set, in ascending order, using only bitwise arithmetic:

**TYPESCRIPT**

```
1 let honestSigners = (1 << (N - T + 1)) - 1; // smallest bitmask
2 while (honestSigners < (1 << N)) {
3   // Process this subset...
4   const c = honestSigners & -honestSigners; // lowest set bit
5   const r = honestSigners + c;              // carry propagation
6   honestSigners = (((r ^ honestSigners) >> 2) / c) | r;
7 }
```

Step 5: For each subset, sample a secret share. The SHAKE-256 stream produces a 64-byte share seed for each subset. From this seed, `deriveUniformLeqEta` samples  $s_1$  ( $L$  polynomials) and  $s_2$  ( $K$  polynomials), each with 256 coefficients in  $[-\eta, \eta]$ .



Step 6: Accumulate the total secret. All 10 shares'  $s_1$  vectors are summed (mod  $q$ ) to get  $\text{totalS}_1$ . Same for  $s_2$ . No single party knows this total.

Step 7: Compute public key.  $t = A \cdot \text{NTT}(\text{totalS}_1) + \text{totalS}_2$ , split via Power2Round into  $(t_1, t_0)$ .  $\text{pk} = \text{Encode}(\rho, t_1)$  — standard FIPS 204 format, 1,312 bytes.  $\text{tr} = \text{SHAKE-256}(\text{pk}, 64)$ .

Step 8: Distribute shares. Each party receives their ThresholdKeyShare containing: party ID,  $\rho$ , per-party key,  $\text{tr}$ , and a map from bitmask to  $(s_1, s_2, \hat{s}_1, \hat{s}_2)$  for all subsets containing that party.

#### FROM MATH TO PRACTICE

In the dealerless DKG (Section 9), each of these steps is distributed across all parties. No single party performs the full computation — each party contributes entropy per-bitmask, and shares are derived jointly so nobody ever sees the full secret. The math is identical; only the trust model changes from centralized to fully distributed.

## 8.3 deriveUniformLeqEta

This function samples a polynomial with small coefficients in  $[-\eta, \eta]$  from a 64-byte seed. It uses SHAKE-256 (not SHAKE-128 as in standard ML-DSA) with input:  $\text{seed}[0:64] \parallel \text{nonce\_le16}$ .

For  $\eta=2$ : each nibble  $t < 15$  yields coefficient  $Q + \eta - (t - \text{floor}(205 \cdot t / 1024) \cdot 5)$ . The magic constant  $\text{floor}(205 \cdot t / 1024)$  computes  $\text{floor}(t/5)$  without division for  $t < 15$ .

For  $\eta=4$ : each nibble  $t \leq 2\eta$  yields coefficient  $Q + \eta - t$ .

# 9. Dealerless DKG: The PERMAFROST Key Ceremony

The dealerless distributed key generation is the core innovation of PERMAFROST beyond the signing protocol. It eliminates the trusted dealer entirely, achieving all four critical properties simultaneously: no trusted dealer, structural secrecy, forced randomness, and minimal public leakage.

ANALOGY

The dealerless DKG is like a group of people mixing paint together. Each person adds their secret color to each bucket. Nobody knows what color the others added. But because they all committed to their colors beforehand (sealed envelopes), nobody can cheat. The final mixed color is the public key — everyone can see it, but nobody knows the individual contributions.

## 9.1 Design Decisions: Per-Bitmask Seed Derivation

Three approaches were considered for dealerless key generation:

Property	Global Seed	Per-Generator Local	Per-Bitmask Seed (chosen)
Structural secrecy	No (all know S)	Yes	Yes
Unbiased shares	Yes	No	Yes (honest minority)
No equivocation	Yes (deterministic)	No (needs complaints)	Yes (deterministic)
No sabotage	Yes	No	Yes
Honest-party req	1 of N	N/A	1 per bitmask subgroup

Core idea: For each bitmask *b*, the holders of *b* jointly derive *seed\_b* via a mini coin-flip. Shares are then deterministically derived from *seed\_b*. This gives structural secrecy (non-holders never learn *seed\_b*), forced randomness (unbiased if at least one holder is honest), no equivocation (all holders independently compute the same share), and no sabotage (shares forced by the jointly-determined seed).

## 9.2 Hard Invariants

### HARD INVARIANTS

I1: Global rho is used ONLY for matrix A expansion and generator assignment. No share material is ever derived from global rho.

I2: Every share ( $s1^b$ ,  $s2^b$ ) is derived from seed\_b, which is known ONLY to holders of b. Non-holders never see seed\_b, its inputs, or the resulting share. This is a structural guarantee — it does not depend on any party deleting data.

## 9.3 Prerequisites

P1. Authenticated confidential channels (REQUIRED): Pairwise authenticated + encrypted channels between all parties. Without confidential channels, the protocol provides NO security guarantees. Required for bitmask seed reveals and mask piece delivery.

P2. Broadcast channel: Reliable broadcast visible to all parties. Required for commitments,  $\rho$  reveals, and aggregates.

P3. Liveness: All N parties must complete all phases. Abort → restart with new session ID.

## 9.4 The Four Phases

1

### Phase 1: Commit

Each party independently samples entropy, computes commitment hashes, and broadcasts them.

Each party  $i$  independently:

- Samples  $\rho_i$  — 256-bit entropy for the public matrix seed.
- For each bitmask  $b$  where  $i$  is a holder: samples  $r_{\{i,b\}}$  — 256-bit entropy contribution.
- Computes and broadcasts commitment hashes:

```
1 C_i^rho = SHAKE-256("DKG-RHO-COMMIT" || sid || i || rho_i)
2 C_{i,b} = SHAKE-256("DKG-BSEED-COMMIT" || sid || b || i || r_{i,b})
```

All parties collect all Phase 1 broadcasts before proceeding.



## 2

**Phase 2: Reveal & Derive**

Parties reveal entropy, verify commitments, derive joint matrix and per-bitmask seeds.

Step 2a: Each party broadcasts  $\rho_i$ . All verify against  $C_i^{\rho}$ . Abort if mismatch.

Step 2b: For each bitmask  $b$ , party  $i$  sends  $r_{\{i,b\}}$  to fellow holders via confidential channel.

Step 2c: Each holder verifies received  $r_{\{i,b\}}$  against  $C_{\{i,b\}}$ . Abort if mismatch.

Step 2d: Derive joint  $\rho$ :

```
1 rho = SHAKE-256("DKG-RHO-AGG" || sid || rho_0 || ... || rho_{N-1})
```

Expand matrix  $A$  from  $\rho$  via XOF-128. Assign generators per bitmask using balanced deterministic assignment.

Step 2e: Each holder of bitmask  $b$  independently computes:

```
1 seed_b = SHAKE-256("DKG-BSEED" || sid || b || r_{p0,b} || ... || r_{pk,b})
2
3 for j = 0..L-1: s1^b[j] = deriveUniformLeqEta(seed_b, j)
4 for j = 0..K-1: s2^b[j] = deriveUniformLeqEta(seed_b, j + L)
```

All holders of  $b$  arrive at the identical share — no distribution needed.

## 3

**Phase 3: Masked Aggregation**

Generators compute partial public keys and distribute info-theoretic masks to all parties.

For each bitmask  $b$ , the designated generator  $\text{gen}(b)$ :

- Computes partial public key contribution:  $w^b = \text{InvNTT}(A * s1^b) + s2^b$
- Splits  $w^b$  into  $N$  info-theoretic masks via `splitVectorK`:
  - Sample  $N-1$  uniform random masks  $r_{\{b,j\}}$  for all  $j \neq \text{gen}(b)$
  - Compute residual:  $r_{\{b,\text{gen}(b)\}} = w^b - \sum \text{other masks (mod } q)$
- Sends  $r_{\{b,j\}}$  to party  $j$  via confidential channel.
- Keeps  $r_{\{b,\text{gen}(b)\}}$  locally — the residual is never transmitted.

## 4

**Phase 4: Aggregate & Finalize**

Parties aggregate masks, reconstruct the public key, and verify via test signing.

Each party  $j$ :

- Computes local aggregate:  $R_j = \sum \text{over all } b \text{ of } r_{\{b,j\}} \pmod{q}$
- Broadcasts  $R_j$

Finalization (all parties, deterministic):

```

1  t  = sum_{j=0}^{N-1} R_j  (mod q)    // masks cancel: yields A*s1 + s2
2  (t0, t1) = Power2Round(t)
3  publicKey = Encode(rho, t1)
4  tr = SHAKE-256(publicKey)

```

Each party constructs their ThresholdKeyShare containing: their party id, rho, tr, a locally-generated 32-byte key, and their shares for all bitmasks where their bit is set.

**POST-DKG VERIFICATION**

After DKG completes, parties perform a test threshold signing of a known message. If the resulting signature verifies against the public key, the DKG succeeded. If not, a party cheated during mask distribution — restart with a fresh session.

## 9.5 Mask Cancellation: Why It Works

The correctness of Phase 4 depends on mask cancellation:

$$t = \sum_j R_j = \sum_j \sum_b r_{\{b,j\}} = \sum_b \sum_j r_{\{b,j\}} = \sum_b w^b = \sum_b (A \cdot s1^b + s2^b) = A \cdot (\sum_b s1^b) + (\sum_b s2^b) = A \cdot s1_{\text{total}} + s2_{\text{total}}$$

The key insight: swapping the order of summation makes the random masks cancel. For each bitmask  $b$ , the  $N$  mask pieces  $r_{\{b,j\}}$  were constructed so that  $\sum_j r_{\{b,j\}} = w^b$ . Therefore the double sum over all  $j$  and all  $b$  yields the sum of all  $w^b$  values, which equals  $A \cdot s1 + s2$ .

## 9.6 Complaint and Abort Protocol

The per-bitmask seed design eliminates most complaint scenarios:

Attack	Per-Generator Local	Per-Bitmask Seed (PERMAFROST)
Biased share	Undetectable	IMPOSSIBLE (forced by seed)
Equivocated share	Detected by commitment	IMPOSSIBLE (local derivation)
Wrong share sent	Detected by commitment	IMPOSSIBLE (no sending needed)
Withheld reveal	N/A	Detected → abort + restart
Wrong mask pieces	Signing fails	Detected by test sign → restart
Withheld mask piece	Protocol stuck	Detected → abort + restart

All abort scenarios lead to restart with a new `session_id`. Per Theorem 5, this is safe — session isolation ensures no cross-session information leakage.

## 9.7 Message Complexity

For a DKG with parameters  $(T, N)$ , let  $|B| = C(N, N-T+1)$  bitmask count and  $k = N-T+1$  holders per bitmask:

Phase	Broadcasts	Private Messages
1 (Commit)	$N$ messages with hashes	0
2 (Reveal)	$N$ messages ( $\rho_{i,j}$ )	$ B  \cdot k \cdot (k-1)$ msgs of 32 bytes
3 (Mask)	0	$ B  \cdot (N-1)$ mask pieces (~3 KB each)
4 (Aggregate)	$N$ messages ( $R_j$ )	0

Worst case ( $T=4, N=6$ ):  $|B|=20, k=3$ . Phase 2: 120 private messages of 32 bytes = 3.8 KB. Phase 3: 100 mask pieces of ~3 KB each = ~300 KB. Total: ~304 KB — well within practical limits for a one-time DKG ceremony.

## 9.8 Security Proofs

The following theorems formally establish the security properties of the PERMAFROST dealerless DKG. These proofs rely on no new cryptographic assumptions beyond what ML-DSA already requires.

## Theorem 1: Correctness

**Statement.** The distributed DKG produces a valid ML-DSA public key  $(p, t_1)$ , and the resulting ThresholdKeyShare values are compatible with the existing threshold signing protocol.

**Proof.**

The total secret:  $s_1 = \sum_{b \in B} s_1^b$ ,  $s_2 = \sum_{b \in B} s_2^b$ .

Each  $(s_1^b, s_2^b)$  is derived via `deriveUniformLeqEta(seed_b, ...)`, which produces polynomials with coefficients in  $[-\eta, \eta]$  — identical to the key generation method's derivation.

By linearity of matrix multiplication and polynomial addition:

```

1  t = A * s1 + s2
2    = sum_b (A * s1^b + s2^b)      [linearity]
3    = sum_b w^b                    [definition of w^b]
4    = sum_b sum_j r_{b,j}          [mask reconstruction: sum_j r_{b,j} = w^b]
5    = sum_j sum_b r_{b,j}          [swap finite sums]
6    = sum_j R_j                    [definition of R_j]
```

Mask cancellation holds because  $\sum_j r_{b,j} = w^b$  by construction:  $N-1$  masks are sampled uniformly, and the residual is computed as  $w^b$  minus the sum of the other masks.

`Power2Round(t)` yields  $(t_0, t_1)$ , `publicKey = Encode(p, t_1)` is a valid FIPS 204 public key. For signing: the share recovery algorithm (`#recoverShare`) selects shares by bitmask and sums them. It depends only on share values and the sharing pattern. Shares from the distributed DKG are functionally identical to single-device shares.

## Theorem 2: Structural Secrecy

**Statement.** For every party  $i$ , there exists at least one bitmask  $b^* \in B$  such that  $i \notin b^*$ . Party  $i$  never learns  $\text{seed}_{\{b^*\}}$  or  $(s_1^{\{b^*\}}, s_2^{\{b^*\}})$ , and therefore cannot reconstruct  $(s_1, s_2)$ .

**Proof.**

**Existence of  $b^*$ :** Bitmasks have  $(N-T+1)$  bits set. The number NOT containing bit  $i$  is  $C(N-1, N-T+1) > 0$  for  $T \geq 2$ . ✓

**Structural separation:**  $\text{seed}_{\{b^*\}}$  is derived from  $\{r_{\{i,b^*\}} : i \in b^*\}$ . Since  $i \notin b^*$ , party  $i$ :

- Did not contribute any  $r_{\{i,b^*\}}$  (they are not a holder)
- Never received any  $r_{\{k,b^*\}}$  (reveals are private to holders of  $b^*$ )



- Cannot compute  $\text{seed}_{\{b^*\}}$  (missing all inputs except the broadcast commitments, which are SHAKE-256 hashes)

Therefore party  $i$  cannot derive  $(s_1^{\{b^*\}}, s_2^{\{b^*\}})$ .

Computational residual: The public transcript reveals only  $t = A \cdot s_1 + s_2$ . Party  $i$  can subtract their known shares' contributions to get  $A \cdot s_1^{\{b^*\}} + s_2^{\{b^*\}} + [\text{other unknown terms}]$ . In the best case for the adversary (only one unknown bitmask), this is a single Module-LWE sample. Recovery requires solving MLWE.

### Theorem 3: Forced Randomness

Statement. If at least one holder of bitmask  $b$  is honest, then  $\text{seed}_b$  is computationally indistinguishable from uniform, and the derived share  $(s_1^b, s_2^b)$  has the canonical LeqEta distribution.

Proof.

Commit-before-reveal prevents adaptive choice: All parties commit to  $r_{\{i,b\}}$  in Phase 1 before any reveals in Phase 2. Changing  $r_{\{i,b\}}$  after committing requires finding a SHAKE-256 second preimage — computationally infeasible.

Honest contribution forces uniformity: Let party  $h$  be the honest holder. After all commitments are broadcast,  $r_{\{h,b\}}$  is uniformly random and independent of all other  $r_{\{i,b\}}$  values (which were committed before  $r_{\{h,b\}}$  was chosen — the binding property ensures no party can adapt after committing).

$\text{seed}_b = \text{SHAKE-256}(\text{"DKG-BSEED"} \parallel \text{sid} \parallel \text{encode\_u16le}(b) \parallel r_{\{p_o,b\}} \parallel \dots \parallel r_{\{pk,b\}}, \text{dkLen}=64)$

In the random oracle model, if  $r_{\{h,b\}}$  is uniform and independent of the other inputs (given the binding of their commitments), then  $\text{seed}_b$  is indistinguishable from uniform.

Deterministic derivation preserves distribution:  $\text{deriveUniformLeqEta}(\text{seed}_b, \text{nonce})$  uses SHAKE-256 as a PRF to produce coefficients with the uniform LeqEta distribution. If  $\text{seed}_b$  is pseudorandom, the output is computationally indistinguishable from the canonical distribution.

No equivocation: All holders independently compute  $\text{seed}_b$  from the same inputs, arriving at the same share. There is no "distribution" of shares between holders — each computes locally.

### Theorem 4: Public-Key Transcript Equivalence

Statement. The public transcript reveals only information equivalent to the standard ML-DSA public key  $(p, t_1)$ . No per-bitmask MLWE samples are leaked.

Proof.

The public transcript consists of:



- $\{C_i^p\}$  — SHAKE-256 hashes (preimage-resistant)
- $\{C_{\{i,b\}}\}$  — SHAKE-256 hashes (preimage-resistant)
- $\{p_i\}$  — random nonces, aggregate determines  $p$  (already public)
- $\{R_j\}$  — per-party masked aggregates

$R_j$  values are statistically uniform:  $R_j = \sum_b r_{\{b,j\}}$ . For at least one bitmask  $b'$  where  $\text{gen}(b') \neq j$ , the term  $r_{\{b',j\}}$  was sampled uniformly by the generator. A sum including at least one uniform independent term is itself uniform.

Only the sum carries signal:  $(R_0, \dots, R_{\{N-1\}})$  is constrained by  $\sum_j R_j = t$ . Any strict subset  $\{R_j : j \in S, |S| < N\}$  is independent of individual  $w^b$  values. This is the standard property of additive N-out-of-N secret sharing.

The extractable information equals one MLWE sample  $(A, t)$  — matching standard ML-DSA key generation exactly.

### Theorem 5: Abort Safety and Session Isolation

Statement. Abort at any phase is safe. Restarting with a fresh `session_id` produces an independent DKG instance. No cross-session information leakage occurs.

Proof.

Domain separation: `session_id` is incorporated into every hash and commitment. Cross-session replay requires a SHAKE-256 collision.

Phase 1 abort: Only hash commitments have been broadcast — no secret material exposed.

Phase 2 abort: Revealed  $p_i$  are non-sensitive random nonces. Revealed  $r_{\{i,b\}}$  values (sent to fellow holders) belong to an aborted session's seed. The new session samples fresh  $r_{\{i,b\}}'$  values from independent randomness. Even if an adversary learns some  $r_{\{i,b\}}$  from the old session, these are uncorrelated with the new session's values.

Phase 3 abort: Mask pieces  $r_{\{b,j\}}$  are uniform random (statistically independent of secrets without the full set).

Phase 4 abort:  $R_j$  values are uniform (Theorem 4).

### Theorem 6: LeqEta Bias Non-Exploitability

Statement. In the edge case where ALL holders of some bitmask  $b$  are malicious (possible when  $N < 2T - 1$ ), they can bias  $\text{seed}_b$  and hence  $(s_1^b, s_2^b)$ . This does not compromise: (a) signature unforgeability, (b) secret recovery hardness, or (c) overall MLWE security.



Proof.

Even with adversarial choice of `seed_b`, the derived shares satisfy `coeff [-η, η]` (enforced by `deriveUniformLeqEta`'s rejection sampling, which always produces valid `LeqEta` coefficients regardless of seed — there is no seed that produces out-of-range coefficients).

(a) Unforgeability: Rejection sampling bounds (`r`, `r'`, `K_iter`) assume worst-case `[-η, η]` norms. Biased distribution can only reduce norms below worst case, improving (not degrading) rejection sampling success rate. Unforgeability follows from the same Module-SIS/LWE reduction.

(b) Secret recovery: The adversary's own biased shares don't help recover non-held shares. Non-held shares are generated by subgroups containing at least one honest party (different bitmask). The residual is still MLWE-hard.

(c) MLWE hardness: The public key  $t = A \cdot \sum_b s_1^b + \sum_b s_2^b$  has  $L_\infty$  norm at most  $|B| \cdot \eta$  per coefficient. MLWE hardness depends on the norm bound, not the distribution within it.

#### ASSUMPTION SUMMARY

No new cryptographic assumption beyond what ML-DSA already requires. The DKG uses only: SHAKE-256 collision/preimage resistance (ML-DSA requires this), SHAKE-256 as PRF/XOF in the random oracle model (ML-DSA requires this), Module-LWE hardness (ML-DSA requires this), and authenticated confidential channels (standard for all DKGs).

## 10. Hyperball Sampling

### 10.1 Motivation

In standard ML-DSA, the masking vector  $y$  is sampled uniformly from  $[-\gamma_1+1, \gamma_1]^{N \cdot L}$ . This works for single-signer because the signer knows the full secret and can perform rejection sampling locally. In threshold ML-DSA, each party independently samples a "partial masking" vector. These must: (1) sum to something that looks like a valid masking vector, (2) not leak information about individual parties' secrets, and (3) enable efficient rejection sampling.

The hyperball construction achieves this by sampling from a continuous distribution on a high-dimensional sphere, then rounding to integers.

### 10.2 The sampleHyperball Algorithm

Input: radius  $r'$ , scaling factor  $v$ , dimensions  $(K, L)$ , seed  $\text{rhop}$  (64 bytes), nonce.

Output: Float64Array of dimension  $N \cdot (K+L)$ .

The algorithm proceeds in three steps:

1. Initialize SHAKE-256 with domain separator:  $H = \text{SHAKE-256}('H' \parallel \text{rhop} \parallel \text{nonce\_le16})$ .
2. Generate  $\text{dim}+2$  random floats via Box-Muller pairs (need even count). For each pair  $(f1, f2)$  of uniform random floats:  $z1 = \sqrt{-2 \cdot \ln(f1)} \cdot \cos(2\pi \cdot f2)$ ,  $z2 = \sqrt{-2 \cdot \ln(f1)} \cdot \sin(2\pi \cdot f2)$ . Accumulate L2 norm:  $\text{sq} += z1^2 + z2^2$ . Scale first  $N \cdot L$  components by  $v$  ( $=3.0$ ).
3. Normalize to sphere of radius  $r'$ :  $\text{factor} = r' / \sqrt{\text{sq}}$ ,  $\text{result}[i] = \text{samples}[i] \times \text{factor}$ .

### 10.3 Box-Muller Transform

The Box-Muller transform converts two independent uniform random variables  $U_1, U_2 \in (0, 1)$  into two independent standard normal variables:

$$Z_1 = \sqrt{-2 \ln U_1} \cdot \cos(2\pi U_2), Z_2 = \sqrt{-2 \ln U_1} \cdot \sin(2\pi U_2)$$

This works because the projection of a multivariate standard normal onto the unit sphere is uniformly distributed on that sphere (a well-known result from spherical distribution theory). The hyperball construction generates  $\text{dim}+2$  standard normal samples and normalizes their L2 norm to radius  $r'$ .

## 10.4 BigInt Precision for Uniform Floats

Converting 64-bit random bytes to uniform floats in  $[0, 1)$  requires care. The naive approach (Go: `float64(uint64) / (1 << 64)`) double-rounds because `uint64` values above  $2^{53}$  cannot be exactly represented as `float64`.

### TYPESCRIPT

```
1 // Extract exactly 53 bits of precision
2 const u: bigint = dataView.getBigUint64(offset, true);
3 const f: number = Number(u >> 11n) * TWO_NEG_53;
4 // TWO_NEG_53 = 2^(-53) = 1.1102230246251565e-16 (exact)
```

Shifting right by 11 bits extracts the top 53 bits of the 64-bit value. Since `Number.MAX_SAFE_INTEGER` =  $2^{53} - 1$ , the result of `Number(u >> 11n)` is always exactly representable. Multiplying by  $2^{-53}$  gives a uniform value in  $[0, 1 - 2^{-53}]$ .

## 10.5 The C1 Fix: Avoiding $\log(0)$

Box-Muller requires  $\ln(U_1)$ , which is  $-\infty$  when  $U_1 = 0$ . The probability of  $U_1 = 0$  is  $2^{-53}$ . The fix:

### TYPESCRIPT

```
1 const f1 = f1Raw === 0 ? Number.MIN_VALUE : f1Raw;
2 // MIN_VALUE ~ 5e-324, gives sqrt(-2*ln(MIN_VALUE)) ~ 38.6
3 // This sample will be rejected by excess check
```

`Number.MIN_VALUE`  $\approx 5 \times 10^{-324}$  is the smallest positive denormalized `float64`. Using it instead of 0 gives a large but finite value that will almost certainly be rejected by the excess check, not affecting the distribution meaningfully.

# 11. Share Recovery

Given  $T$  active parties out of  $N$  total, each party must reconstruct their combined secret share ( $s1Hat$ ,  $s2Hat$ ) for the specific active set. The key generation distributes shares indexed by honest-signer bitmasks. Share recovery maps the current active set to the appropriate combination of stored shares.

## 11.1 Sharing Patterns

Sharing patterns are precomputed by a max-flow optimal assignment algorithm and stored as lookup tables indexed by  $(T, N)$ . Each `pattern[i]` lists the bitmasks that party  $i$  must sum to reconstruct their share.

Example ( $T=2, N=3$ ): `pattern = [[3, 5], [6]]`. Party 0 sums shares with bitmasks 3 (`=0b011`) and 5 (`=0b101`). Party 1 sums share with bitmask 6 (`=0b110`).

## 11.2 Permutation-Based Bitmask Translation

Sharing patterns are defined for a canonical ordering where the first  $T$  parties are active. When the actual active set differs, a permutation translates bitmask indices:

### TYPESCRIPT

```
1 // Build permutation: active -> 0..T-1, inactive -> T..N-1
2 const perm = new Uint8Array(N);
3 let i1 = 0, i2 = T;
4 for (let j = 0; j < N; j++) {
5   if ((act & (1 << j)) !== 0) perm[i1++] = j;
6   else perm[i2++] = j;
7 }
8
9 // Translate bitmask through permutation
10 let u_ = 0;
11 for (let i = 0; i < N; i++) {
12   if ((u & (1 << i)) !== 0) u_ |= (1 << perm[i]);
13 }
14 const s = share.shares.get(u_);
```



## 11.3 Base Case: $T = N$

When  $T = N$  (all parties required), each party has exactly one share corresponding to the single  $C(N, 1) = N$  subset of size 1. No permutation is needed — each party returns their single stored share directly.

## 11.4 Share Accumulation

After translation, the party sums all assigned shares: for each bitmask  $u$  in the sharing pattern, translate to actual bitmask  $u'$ , look up the stored share, and accumulate via polynomial addition mod  $q$ . The result ( $s1Hat$ ,  $s2Hat$ ) is this party's additive share of the full secret in NTT domain.

## 12. The 3-Round Distributed Signing Protocol

The distributed signing protocol is PERMAFROST's core contribution: any  $T$  parties independently generate their own randomness on their own machines, exchange only commitments and responses, and produce a single standard FIPS 204 signature. No party's secret ever leaves their device.

Round	Communication	Purpose	Data Size
Round 1	Broadcast 32-byte hash	Commit to randomness	32 bytes
Round 2	Broadcast packed $w$ vectors	Reveal commitment	$K_{\text{iter}} \times K \times 736$ bytes
Round 3	Broadcast packed $z$ vectors	Partial response	$K_{\text{iter}} \times L \times 736$ bytes
Combine	None (local computation)	Produce FIPS 204 signature	0 bytes

### 12.1 Round 1: Commitment

1

#### Round 1: Commitment

Each party independently generates randomness, samples hyperball points, and broadcasts a commitment hash.

Each party independently:

1. Generate randomness:  $\text{rhop} = \text{random } 64 \text{ bytes}$
2. Sample  $K_{\text{iter}}$  hyperball points:  $\text{stw}[\text{iter}] = \text{sampleHyperball}(r', v, K, L, \text{rhop}, \text{nonce})$
3. Round to integer vectors:  $(y, e) = \text{fvecRound}(\text{stw})$
4. Compute commitment:  $w = A \cdot \text{NTT}(y) + e$
5. Pack commitments into bytes
6. Hash commitment:

```
1 commitmentHash = SHAKE-256(tr || partyId || packedW) // 32 bytes
```

Output: {commitmentHash (32 bytes), state}. Broadcast hash to all parties.

### WHY COMMIT-THEN-REVEAL?

Without commit-then-reveal, a malicious party could see others' commitments and choose their own adaptively to manipulate the combined commitment. The hash commitment prevents this: all parties commit before any reveals.

## 12.2 Round 2: Reveal

2

### Round 2: Reveal

After receiving all T commitment hashes, each party validates, stores hashes, and reveals their commitment.

After receiving all T commitment hashes, each party:

1. Validate: correct number of hashes, no duplicate party IDs.
2. Store hashes: save copies for verification in Round 3.
3. Compute message digest:

```
1 |  $\mu$  = SHAKE-256(tr || message) // 64 bytes
```

4. Reveal commitment: return packed w data from Round 1.

Output: {commitment (packed bytes), state}. Broadcast commitment to all parties.

## 12.3 Round 3: Partial Response

3

### Round 3: Partial Response

After receiving all reveals, each party verifies commitments, recovers shares, and computes their partial signature.

After receiving all T commitment reveals, each party:

1. Verify commitments against stored hashes (constant-time comparison). Abort if any mismatch detected.
2. Unpack and aggregate commitments:  $w_{\text{final}} = \sum$  of all parties' w vectors.
3. Recover combined secret share:  $(s1_{\text{Hat}}, s2_{\text{Hat}}) = \text{recoverShare}(\text{share}, \text{activePartyIds})$ .
4. For each iteration: compute challenge c, then  $cs = c \cdot s$ ,  $zf = cs + stw$ , check excess. Apply H1 fix: always execute fvecRound regardless of excess (see Section 12.4).



5. Pack and return partial z response.

Output: packed partial response. Broadcast to all parties.

## 12.4 The H1 Timing Fix

The excess check `fvecExcess(zf, r, v)` reveals whether the current iteration was accepted or rejected. If `fvecRound()` is only called on accepted iterations, the execution time correlates with the rejection pattern, which correlates with the secret key.

### TYPESCRIPT

```
1  const excess = fvecExcess(zf, params.r, params.nu, K, L);
2  const { z } = fvecRound(zf, K, L); // ALWAYS executed (H1 fix)
3
4  if (excess) {
5      zs.push(zeroVector); // Use zero, but fvecRound already ran
6  } else {
7      zs.push(z);          // Accept this iteration
8  }
```

### H1 TIMING LEAK

Without the H1 fix, the execution time of `fvecRound` (which involves `Math.round` on every coefficient) is secret-dependent. This timing leak could allow an attacker to determine the rejection pattern across multiple signing sessions, gradually extracting information about the secret key. Always executing `fvecRound` ensures constant execution time.

## 13. Combine: Signature Finalization

The combine step produces the final standard FIPS 204 signature from the aggregated data. It requires only the threshold public key, the message, all parties' commitments, and all parties' responses. It does NOT require any secret key material — anyone with the public key can perform this step.

### 13.1 The Combine Algorithm

For each iteration  $\text{iter} = 0$  to  $K_{\text{iter}}-1$ :

1. Aggregate commitments:  $w_{\text{final}}[\text{iter}][j] = \sum \text{allWs}[\text{party}][\text{iter}][j] \bmod q$ .
2. Aggregate responses:  $z_{\text{final}}[\text{iter}][j] = \sum \text{allZs}[\text{party}][\text{iter}][j] \bmod q$ .
3. Check  $\|z_{\text{final}}\|_{\infty} < \gamma_1 - \beta$ . Skip if fails.
4. Decompose:  $(w_0, w_1) = \text{Decompose}(w_{\text{final}}[\text{iter}])$ .
5. Recompute challenge:  $c = \text{SHAKE-256}(\mu \parallel W1\text{Encode}(w_1))$ ,  $c = \text{SampleInBall}(c)$ .
6. Compute  $Az$  in NTT domain:  $Az[i] = \sum_j A[i][j] \cdot \text{NTT}(z[j])$ .
7. Compute  $Az - 2^d \cdot c \cdot t_1$ :  $ct12d[i] = \text{NTT}(c) \cdot \text{NTT}(t_1[i] \ll d)$ .
8. Compute  $f = \text{InvNTT}(Az - ct12d) - w_{\text{final}}$ . Check  $\|f\|_{\infty} < \gamma_2$ .
9. Compute hint:  $h = \text{MakeHint}(w_0 + f, w_1)$ . Check  $\text{weight}(h) \leq \omega$ .
10. Success! Encode signature:  $\sigma = \text{sigCoder.encode}([c, z, h])$ . Return  $\sigma$ .

### 13.2 Why This Produces Valid FIPS 204 Signatures

The key mathematical identity that makes threshold signing work:

$$Az - c \cdot t_1 \cdot 2^d = A \cdot (\sum y_i + c \cdot \sum s_{1_i}) - c \cdot (A \cdot \sum s_{1_i} + \sum s_{2_i} - \sum t_{0_i}) \cdot 2^d \approx A \cdot \sum y_i + \sum e_i + c \cdot \sum s_{2_i} = w_{\text{final}} + c \cdot s_2$$

The "approximately" comes from rounding errors in `fvecRound`. The hint  $h$  corrects for these small differences, exactly as in standard ML-DSA. The verifier computes  $w'_{\text{approx}} = Az - c \cdot t_1 \cdot 2^d$ , applies `UseHint(h, w'_{\text{approx}})` to get  $w'_1$ , and checks  $\text{SHAKE-256}(\mu \parallel W1\text{Encode}(w'_1)) = c$ . This is identical to standard FIPS 204 verification.



### IMPLEMENTATION NOTE

Protection of `t1`: When computing `NTT(polyShiftl(t1[i]))`, the `polyShiftl` operation modifies the polynomial in-place (left-shifts by `d=13` bits). To prevent corrupting the decoded public key data, we operate on a copy: `t1[i].slice()`. This defensive copy is essential for correctness.

## 14. Polynomial Packing: 23-Bit Encoding

The distributed protocol needs to transmit full  $R_q$  polynomials (coefficients in  $[0, q)$ ) between parties. Since  $q = 8,380,417 < 2^{23}$ , each coefficient requires exactly 23 bits.

### 14.1 Encoding: polyPackW

Input: `Int32Array[256]` with values in  $[0, 2^{23})$ . Output: `Uint8Array[736]` ( $256 \times 23 / 8 = 736$  bytes).

#### TYPESCRIPT

```
1 // Bit-packing algorithm
2 v = 0, j = 0
3 for each coefficient p[i]:
4   v |= (p[i] & 0x7FFFFFF) << j // append 23 bits
5   j += 23
6   while j >= 8: // flush complete bytes
7     buf[k++] = v & 0xFF
8     v >>= 8
9     j -= 8
```

### 14.2 Decoding: polyUnpackW (with M7 Validation Fix)

#### TYPESCRIPT

```
1 // Bit-unpacking with M7 bounds check
2 v = 0, j = 0
3 for each output coefficient:
4   while j < 23: // need 23 bits
5     v += buf[k++] << j
6     j += 8
7   coeff = v & ((1 << 23) - 1) // extract 23 bits
8   if (coeff >= Q) throw Error // M7 FIX: validate!
9   v >>= 23
10  j -= 23
```

**M7 VALIDATION FIX**

M7 Fix: The unpacking MUST validate that each coefficient is in  $[0, q)$ . This prevents an adversary in a distributed protocol from injecting malformed polynomials with coefficients  $\geq q$ , which could cause undefined behavior in ring arithmetic.

## 14.3 Bit Safety Analysis

All intermediate values in the packing/unpacking fit within 32-bit integers. Maximum accumulator value during packing:  $v = \text{coeff} \ll j$  where  $\text{coeff} < 2^{23}$  and  $j < 8$ , so  $v < 2^{31}$ . Maximum accumulator during unpacking:  $v = \text{byte} \ll j$  where  $\text{byte} < 2^8$  and  $j < 23$ , so  $v < 2^{31}$ . No 53-bit overflow is possible.

Communication sizes: Commitment =  $K_{\text{iter}} \times K \times 736$  bytes (e.g.,  $2 \times 4 \times 736 = 5,888$  bytes for ML-DSA-44, 2-of-2). Response =  $K_{\text{iter}} \times L \times 736$  bytes.

# 15. Parameter Tables

## 15.1 ML-DSA-44 (K=4, L=4, NIST Level 2)

T\N	2	3	4	5	6
2	K=2	K=3	K=3	K=3	K=4
3	—	K=4	K=7	K=14	K=19
4	—	—	K=8	K=30	K=74
5	—	—	—	K=16	K=100
6	—	—	—	—	K=37

Full parameters for ML-DSA-44 (selected configurations):

(T,N)	K_iter	r	r'	v
(2,2)	2	252,778	252,833	3.0
(2,3)	3	310,060	310,138	3.0
(3,3)	4	246,490	246,546	3.0
(3,5)	14	282,800	282,912	3.0
(4,5)	30	259,427	259,526	3.0
(4,6)	74	268,705	268,831	3.0
(5,6)	100	250,590	250,686	3.0
(6,6)	37	219,245	219,301	3.0

## 15.2 ML-DSA-65 (K=6, L=5, NIST Level 3)

(T,N)	K_iter	r	r'
(2,2)	2	344,000	344,080
(2,3)	3	421,700	421,810
(3,3)	4	335,200	335,290
(3,5)	14	384,600	384,750
(5,6)	100	340,700	340,830
(6,6)	37	298,000	298,080

## 15.3 ML-DSA-87 (K=8, L=7, NIST Level 5)

(T,N)	K_iter	r	r'
(2,2)	2	442,000	442,100
(2,3)	3	541,600	541,740
(3,3)	4	430,600	430,710
(3,5)	14	494,200	494,400
(5,6)	100	437,400	437,570
(6,6)	37	382,800	382,910

## 15.4 Why $N \leq 6$

N is capped at 6 for mathematical and practical reasons:



1. Combinatorial explosion:  $C(N, N-T+1)$  subsets.  $N=6$ : up to 20 subsets.  $N=10$ : up to 252 subsets.
2.  $K_{\text{iter}}$  growth: For  $(T=5, N=6)$ ,  $K_{\text{iter}} = 100$ , meaning each signing round requires 100 parallel attempts. Communication:  $100 \times 4 \times 736 = 294$  KB per party per round.
3. Parameter table size: Each  $(T, N)$  needs empirically optimized  $(K_{\text{iter}}, r, r')$  values.
4. Share storage: Each party stores shares for all subsets they belong to. More subsets = more key storage.

The signature size NEVER changes — it's always standard FIPS 204. The extra communication during signing is off-chain between parties. Beyond  $N=6$ , this communication becomes impractical.

## 16. Security Analysis and Hardening

The Go reference implementation has known vulnerabilities identified in its audit report. PERMAFROST addresses all five findings and adds additional hardening.



### 16.1 C1 (Critical): Box-Muller NaN/Infinity from log(0)

Vulnerability: When  $U_1 = 0$  (probability  $2^{-53}$ ),  $\text{Math.log}(0) = -\text{Infinity}$ , causing NaN propagation through all subsequent arithmetic.

Fix: Clamp to `Number.MIN_VALUE`:

TYPESCRIPT

```
1 const f1 = f1Raw === 0 ? Number.MIN_VALUE : f1Raw;
2 // MIN_VALUE ~ 5e-324, gives sqrt(-2*ln(MIN_VALUE)) ~ 38.6
```

### 16.2 C8 (Critical): No Input Validation

Vulnerability: Passing wrong-length or undefined values to `sign()` could cause silent misbehavior — wrong polynomial sizes, buffer overflows, or silently incorrect signatures.

Fix: `abytes()` length checks on all inputs:

TYPESCRIPT

```
1 abytes(publicKey, p.publicCoder.bytesLen, 'publicKey');
2 abytes(msg); // validates it's a Uint8Array
```

## 16.3 H1 (High): Timing Leak from Rejection Pattern

Vulnerability: The accept/reject decision depends on the secret key. If `fvecRound()` is only called on accepted iterations, execution time reveals the rejection pattern.

Fix: Always execute `fvecRound()` regardless of the excess check result (see Section 12.4).

## 16.4 H6 (High): No Zeroization of Secret Material

Vulnerability: Secret shares, intermediate products ( $c \cdot s_1$ ,  $c \cdot s_2$ ), and the message digest  $\mu$  remain in memory after use.

Fix: Explicit zeroization on both success and failure paths:

TYPESCRIPT

```
1 // After each iteration:
2 csVec.fill(0); zf.fill(0); cleanBytes(cs1, cs2);
3
4 // After all iterations:
5 cleanBytes(s1Hat, s2Hat); // recovered share
6
7 // In sign() - even on failure:
8 try { ... } finally { mu.fill(0); }
```

## 16.5 M7 (Medium): No Coefficient Validation on Unpack

Vulnerability: In distributed protocol, a malicious party could send polynomials with coefficients  $\geq q$ , leading to undefined behavior in ring arithmetic.

Fix: Bounds check in `polyUnpackW` (see Section 14.2).

## 16.6 Additional Hardening

Duplicate Share ID Validation: Prevents incorrect bitmask construction from duplicate IDs.

Commitment Hash Verification (Constant-Time): Uses bitwise OR of XOR differences to prevent timing-based attacks in Round 3.



State Destruction Classes: Round1State and Round2State use #private fields and provide destroy() methods that zero all sensitive data.

Failure-Path Cleanup: Message digest  $\mu$  is zeroed even when all 500 signing attempts are exhausted.

## 17. Frontend Integration: Communication Layer

---

The PERMAFROST protocol requires T wallets to exchange messages during the 3-round signing process. This section describes the recommended communication architecture for wallet-to-wallet coordination.

### 17.1 The Communication Problem

N wallets need to exchange messages securely for both DKG (4 phases, one-time) and signing (3 rounds, per transaction). Requirements: authenticated channels, message ordering, low latency, and end-to-end encryption.

### 17.2 Option 1: WebRTC DataChannels

WebRTC provides peer-to-peer encrypted communication via DataChannels, with a signaling server only for connection setup. Pros: true P2P, built-in DTLS encryption. Cons: complex NAT traversal, TURN server fallback, connection setup latency.

### 17.3 Recommended: Encrypted Mailbox over WebSocket

The recommended approach uses a WebSocket-based mailbox server where parties deposit and retrieve encrypted messages. Each party connects to the mailbox, encrypts messages using hybrid X25519 + ML-KEM-768 double encryption (Section 17.4), and polls for incoming messages.

Why WebSocket mailbox wins: Simple deployment (single server), no NAT traversal issues, works through firewalls, asynchronous message delivery (parties don't need to be online simultaneously), and the mailbox server sees only encrypted ciphertext.

### 17.4 E2E Encryption: Hybrid X25519 + ML-KEM (Dual System)

The DKG transport layer carries bitmask seed reveals and mask pieces. An attacker who records this traffic and later breaks the transport encryption can reconstruct every bitmask share and recover the full secret key. This is a textbook "harvest now, decrypt later" attack. The transport layer must therefore be resistant to both classical and quantum adversaries.

PERMAFROST mandates hybrid (double) encryption for all DKG traffic: X25519 ECDH combined with ML-KEM-768 (FIPS 203) key encapsulation. The shared secret is derived from both key agreements, so an attacker must break BOTH to decrypt the traffic. This follows the same principle deployed in production by



Google (CECPQ1/CECPQ2), Cloudflare, and Chrome — now approaching 50% of all TLS connections.

Why hybrid, not standalone ML-KEM? Daniel J. Bernstein documents extensively how post-quantum cryptosystems can fail unexpectedly. SIKE was applied to millions of real connections in 2019 via CECPQ2b; it was publicly broken in 2022. The ONLY reason user data wasn't immediately exposed is that CECPQ2b encrypted with SIKE AND with ECC, not just SIKE. ML-KEM's reference implementation required two rounds of security patches for timing leaks (KyberSlash, late 2023) and a third patch in mid-2024. Standalone post-quantum is driving without a seatbelt.

Protocol: Each party generates an ephemeral X25519 keypair AND an ephemeral ML-KEM-768 keypair per DKG session. For each pair of parties (i, j):

- Party i performs X25519 ECDH with party j's X25519 public key → shared\_ecdh (32 bytes)
- Party i encapsulates to party j's ML-KEM-768 encapsulation key → shared\_kem (32 bytes)
- Combined key: SHAKE-256("PERMAFROST-TRANSPORT" || shared\_ecdh || shared\_kem, dkLen=32)
- The combined key is used as the AES-256-GCM key for encrypting round messages between that pair

Security properties: If ML-KEM breaks (like SIKE did), X25519 still protects the traffic against classical adversaries. If X25519 breaks (quantum computer), ML-KEM-768 still provides 192-bit post-quantum security. An attacker must break both simultaneously to recover DKG secrets. Per-session ephemeral keys provide forward secrecy — compromising one ceremony reveals nothing about past or future DKG sessions.

### **HYBRID TRANSPORT IS MANDATORY**

Using standalone pre-quantum (X25519 only) OR standalone post-quantum (ML-KEM only) for DKG transport is insufficient. Pre-quantum alone falls to harvest-now-decrypt-later. Post-quantum alone risks a SIKE-class catastrophic break with no fallback. The hybrid approach ensures that the security of the transport layer is the STRONGER of the two components, not the weaker.

### **LIVE DEMO**

A live demo of the PERMAFROST communication layer is available at:  
<https://ipfs.opnet.org/ipfs/bafybeiejz5fc44scvp5qbgz4zzeg3jiecdd5t7dwr6o3dqqhbk54czlhia/>

## 18. Float64 Precision Analysis

---

### 18.1 Where Float64 is Used

Float64 (IEEE 754 double precision) is used exclusively in hyperball sampling and the FVec operations: `sampleHyperball` (Box-Muller), `fvecFrom` (Int32 → Float64 conversion), `fvecAdd` (vector addition), `fvecExcess` (weighted L2 norm), and `fvecRound` (Float64 → Int32 rounding).

### 18.2 Ring Arithmetic: Always Int32

All ring operations (`polyAdd`, `polySub`, `MultiplyNTTs`, NTT encode/decode) use `Int32Array` and stay within safe integer range. Maximum coefficient:  $q-1 = 8,380,416$  (23 bits). Maximum product:  $(q-1)^2 \approx 7.02 \times 10^{13}$  (47 bits) — within 53-bit safe range. All cryptographic ring arithmetic is exact.

### 18.3 Platform Dependence of Transcendental Functions

`Math.sqrt`, `Math.cos`, `Math.sin`, `Math.log` may differ by 1 ULP across platforms. This means: cross-platform deterministic signing is NOT guaranteed, but each platform independently produces valid signatures. The protocol's security depends on statistical properties, not determinism.

### 18.4 fvecRound Precision

The rounding step:  $u = \text{Math.round}(v) \mid 0$ . Since Float64 values are bounded by  $\pm 4,190,208$  and hyperball radii are  $\sim 250,000$ - $550,000$ , rounded values fit in int32. The  $\mid 0$  truncation is safe.

# 19. Architecture and Implementation

---

## 19.1 Module Structure

The reference implementation is available at [github.com/btc-vision/noble-post-quantum](https://github.com/btc-vision/noble-post-quantum), a fork of Paul Miller's noble-post-quantum library extended with threshold ML-DSA and dealerless DKG. The implementation consists of four key modules:

`_crystals.ts`: NTT, XOF (SHAKE-128/256), core finite field operations.

`ml-dsa-primitives.ts`: Ring arithmetic, coders, sampling, decomposition functions. The `createMLDSAPrimitives()` factory extracts shared functionality for reuse.

`ml-dsa.ts`: Standard FIPS 204 implementation (keygen/sign/verify).

`threshold-ml-dsa.ts`: `ThresholdMLDSA` class with the complete distributed protocol.

## 19.2 Class Design: ThresholdMLDSA

### TYPESCRIPT

```

1  class ThresholdMLDSA {
2      static readonly MAX_PARTIES = 6;
3      static create(securityLevel, T, N): ThresholdMLDSA;
4      static getParams(T, N, securityLevel): ThresholdParams;
5
6      // Key generation
7      keygen(seed?): ThresholdKeygenResult;
8
9      // Local convenience
10     sign(msg, publicKey, shares, opts?): Uint8Array;
11
12     // Distributed protocol
13     round1(share, opts?): Round1Result;
14     round2(share, ids, msg, hashes, state, opts?): Round2Result;
15     round3(share, commitments, state1, state2): Uint8Array;
16     combine(publicKey, msg, commitments, responses, opts?): Uint8Array | null;
17
18     // DKG methods
19     dkgSetup(sessionId): { bitmasks, holdersOf };
20     dkgPhase1(partyId, sessionId): { broadcast, state };
21     dkgPhase2(partyId, sessionId, state, allPhase1): { broadcast, privateToHolders };
22     dkgPhase2Finalize(partyId, sessionId, ...): { shares, generatorAssignment, privateToAll };
23     dkgPhase4(partyId, sessionId, ...): { broadcast };
24     dkgFinalize(partyId, sessionId, ...): DKGResult;
25 }

```

## 19.3 State Management

Round1State: Contains #stws (Float64Array[] — hyperball samples, SENSITIVE) and #commitment (packed w vectors). Uses ES2022 #private fields. Provides destroy() for zeroization.

Round2State: Contains #hashes (stored commitment hashes), #mu (message digest, SENSITIVE), #act (active signer bitmask), #activePartyIds. Also #private with destroy().



## 19.4 FVec Layout

A Float64Array of size  $N \cdot (K+L)$  representing a vector in  $R_q^{K+L}$ . First  $N \cdot L$  floats: the  $z/y$  components. Last  $N \cdot K$  floats: the  $e$  (error) components. Total:  $256 \cdot 8 = 2048$  floats for ML-DSA-44. The  $L$ -components are scaled by  $v=3.0$  during hyperball sampling.

## 20. API Reference

### 20.1 ThresholdMLDSA.create(securityLevel, T, N)

Parameter	Type	Description
securityLevel	number	44, 65, 87 (or 128, 192, 256)
T	number	Threshold (minimum signers), $2 \leq T$
N	number	Total parties, $T \leq N \leq 6$

### 20.2 keygen(seed?)

Single-device key generation (for testing and development). Returns { publicKey, shares }. Optional 32-byte deterministic seed. For production use, the dealerless DKG methods (Section 9) are the correct path.

### 20.3 round1(share, opts?)

Distributed Round 1. Returns { commitmentHash (32 bytes), state (Round1State) }. Broadcast commitmentHash to all parties. Keep state private.

### 20.4 round2(share, activePartyIds, msg, hashes, state, opts?)

Distributed Round 2. Returns { commitment (packed bytes), state (Round2State) }. Broadcast commitment to all parties.

### 20.5 round3(share, commitments, state1, state2)

Distributed Round 3. Returns packed partial response (Uint8Array). Throws if any commitment doesn't match its hash. Broadcast response to all parties.

## 20.6 combine(publicKey, msg, commitments, responses, opts?)

Combine step. Returns standard FIPS 204 signature (Uint8Array) or null if all K\_iter iterations failed. Does NOT require secret key material.

## 20.7 DKG Methods

Method	Input	Output
dkgSetup	sessionId	bitmasks, holdersOf
dkgPhase1	partyId, sessionId	broadcast, state
dkgPhase2	partyId, sid, state, allPhase1	broadcast, privateToHolders
dkgPhase2Finalize	partyId, sid, state, phase1, phase2, reveals	shares, generators, masks
dkgPhase4	partyId, sid, phase2, masks, ownMasks, shares broadcast (R_j)	
dkgFinalize	partyId, sid, phase2, phase4, shares	DKGResult {publicKey, share}

## 20.8 Byte Lengths

Property	ML-DSA-44	ML-DSA-65	ML-DSA-87
Public key	1,312	1,952	2,592
Signature	2,420	3,309	4,627
Commitment hash	32	32	32
Polynomial (packed)	736	736	736
Commitment (2-of-2)	5,888	7,360	10,304
Response (2-of-2)	5,888	7,360	10,304

## 21. Known Limitations

---

### 21.1 No Side-Channel Protection in JavaScript

JavaScript cannot guarantee constant-time operations. Branch misprediction from if/else on secret-dependent values, cache-timing attacks on array indexing, JIT compiler optimizations, and garbage collector pauses may introduce timing variations. The H1 fix addresses one specific leak, but comprehensive side-channel protection requires hardware-level guarantees.

### 21.2 Float64 Platform Dependence

Transcendental functions are not required to be correctly rounded by IEEE 754. Cross-platform deterministic signing is NOT guaranteed. Each platform independently produces valid signatures.

### 21.3 $N \leq 6$ Cap

$N$  is limited to 6 due to: combinatorial explosion of subsets, rapid  $K_{\text{iter}}$  growth, parameter table size requirements, and share storage overhead. The signature size never changes — only off-chain communication grows.

### 21.4 Identifiable Aborts (Implicit Only)

The implementation supports identifiable abort detection implicitly (commitment hash verification catches cheating parties), but does not expose a standalone `identifyAbort()` method.

### 21.5 DKG Limitations

No full malicious robustness when  $N < 2T-1$ : If  $T-1$  colluders all land in the same bitmask, they can bias that share. Proven non-exploitable (Theorem 6) but not provably random.

No robustness against corrupt mask distribution: A malicious generator can send wrong mask pieces. Detected only by test signing (wrong public key → restart).

No proactive security / share refresh. Shares are static.



No abort tolerance during DKG. Liveness requires all N parties.

## 21.6 Unaudited Implementation

### UNAUDITED

This is a port of an academic reference implementation. It has NOT undergone formal security audit. The Go reference implementation's audit findings have been applied, but the TypeScript port itself is unaudited. Do not use for production custody without formal review.

# Appendix A: Notation Reference

Symbol	Meaning
$R_q$	$\mathbb{Z}_q[X]/(X^{256} + 1)$ , the polynomial ring
$q$	8,380,417 (the Dilithium prime)
$N$	256 (polynomial degree)
$K, L$	Matrix dimensions of $A \in R_q^{K \times L}$
$T$	Threshold (minimum signers)
$\eta$ (eta)	Bound on secret key coefficients
$\tau$ (tau)	Hamming weight of challenge polynomial
$\gamma_1$ (gamma1)	Masking range for $z$
$\gamma_2$ (gamma2)	Decomposition parameter
$\beta$ (beta)	$\tau \cdot \eta$ , norm bound adjustment
$\omega$ (omega)	Maximum hint weight
$d$	13, Power2Round parameter
$\nu$ (nu)	3.0, hyperball scaling factor

Symbol	Meaning
$r$	Primary L2 radius bound
$r'$	Sampling hypersphere radius
$K_{\text{iter}}$	Parallel signing iterations per round
$\rho$ (rho)	Public randomness seed (32 bytes)
$tr$	Public key hash (64 bytes)
$\mu$ (mu)	Message digest (64 bytes)
$c\sim$	Challenge hash (32/48/64 bytes)
$A$	Public matrix in $R_q^{K \times L}$ (NTT domain)
$s_1, s_2$	Secret key vectors
$t_1$	Public key polynomial vector
$w$	Commitment vector
$z$	Response vector
$h$	Hint vector
$\sigma$	Signature = $(c\sim, z, h)$

## Appendix B: Test Coverage

The PERMAFROST implementation includes 135 tests across 7 test files covering all aspects of the protocol:

### Threshold Tests (59 tests)

Category	Count	Coverage
Parameter Validation	10	T<2, T>N, N>6, N<2, invalid level, all valid combos
Key Generation	9	Deterministic, random, wrong seed, share count, pk lengths
Signing (ML-DSA-44)	11	2-of-3 all subsets, 2-of-2, 3-of-4, 3-of-3, 4-of-4
Signing (ML-DSA-65/87)	4	2-of-3 and 2-of-2 per level
Context & Edge Cases	5	Context, wrong context, empty/large message
Error Cases	5	Insufficient shares, wrong pk, duplicate IDs
NIST Level Mapping	3	128→44, 192→65, 256→87
Distributed Protocol	14	Full round trip, tampered commitment, state destruction

## Dealerless DKG Tests (55 tests)

Category	Count	Coverage
Correctness	6	Full 4-phase DKG for all (T,N) pairs, valid ML-DSA key verification
Signing Compatibility	6	DKG shares produce valid sigs via sign() and round1-round3
Seed Consistency	5	All holders of bitmask b derive identical seed_b
Mask Cancellation	5	$\sum_j R_j == \sum_b w^b$ computed independently
Structural Secrecy	5	Each party missing at least one bitmask
Commitment Binding	5	Wrong $r_{\{i,b\}}$ reveal detected by fellow holders
Session Isolation	5	Cross-session commitments rejected
Generator Balance	5	No party assigned excess bitmasks
Non-holder Exclusion	5	Non-holders cannot compute seed_b
Post-DKG Test Sign	5	Successful DKG always yields signable shares
Deterministic Vectors	3	Fixed-seed reproducibility across runs

## Standard ML-DSA Tests (21 tests)

All existing ML-DSA tests pass without regression after the primitives extraction refactor: 8 ACVP NIST vectors, 5 basic operations, 4 hybrid tests, 3 DKG vectors, 1 error handling.

## DKG Verification Points

#	Test	Description
1	Correctness	Full 4-phase DKG for all (T,N), verify valid ML-DSA key
2	Signing compat	DKG shares produce valid signatures via sign() and round1-round3
3	Seed consistency	All holders of bitmask b derive identical seed_b
4	Mask cancellation	$\sum_j R_j == \sum_b w^b$ (both computed independently)
5	Structural secrecy	Each party missing at least one bitmask
6	Commitment binding	Reveal different $r_{\{i,b\}}$ than committed $\rightarrow$ detected
7	Session isolation	Session A commitments invalid in session B
8	Generator balance	No party assigned $> \text{ceil}( B /k)$ bitmasks
9	Non-holder exclusion	Non-holders cannot compute seed_b from public transcript
10	Post-DKG test sign	Successful DKG always produces signable shares