ML-ADSA: A Trapdoor-Free Post-Quantum Aggregate Signature from Module-Lattice Summation over ML-DSA

pq-cybarg

2026

Contents

1 ML-ADSA: A Trapdoor-Free Post-Quantum Aggregate Signature from Module-Lattice Summation over ML-DSA

Research paper draft (v1.0). Companion artifacts: formal specification (`docs/30`), end-to-end verification dossier (`docs/31`), publication/NIST roadmap (`docs/32`), reference implementation (`go-mladsa/`), and machine-checked proofs (`formal/`). This draft is structured for IACR ePrint / a peer-reviewed venue; it is LaTeX-convertible. Independent cryptanalysis and peer review are pending and are explicitly flagged as prerequisites for standardization.

1.1 Abstract

We present ML-ADSA (Module-Lattice Aggregate Digital Signature Algorithm), a non-interactive, decentralized, trapdoor-free aggregate signature scheme over ML-DSA (FIPS-204), instantiated at all three FIPS-204 parameter sets — ML-ADSA-44, -65, and -87, at NIST Categories 2, 3, and 5. A committee of signers, each holding an ordinary ML-DSA key, jointly produces a single constant-size signature that the unmodified FIPS-204 verifier accepts against an aggregate public key. ML-ADSA requires no trusted setup, no trapdoors, no SNARK/STARK/zero-knowledge proof system, and no trusted or intermediary aggregator; every value the combiner uses is public, so any party reconstructs the identical aggregate. The construction is parameter-generic — it depends only on ML-DSA's shared base ring $(q,\ n=256,\ d=13)$ and the Fiat–Shamir linearity, not on the per-level $(k, \ell,\ \eta,\ \tau,\ \gamma_1,\ \gamma_2,\ \omega)$ — so a single combine procedure serves every level; the aggregate sizes are exactly ML-DSA's (pk/sig of 1312/2420, 1952/3309, 2592/4627 bytes), independent of the committee size.

Our central observation is structural: BLS aggregation is the homomorphic product of signatures in a pairing group, $\sigma* = \Pi\ \sigma_i$; ML-ADSA is the homomorphic sum of Fiat–Shamir responses in the module $R_q = Z_q[X]/(X^{256}+1)$, $z* = \Sigma\ z_i$, $t* = \Sigma\ t_i$, $W* = \Sigma\ w_i$. The same homomorphic-aggregation algebra gives ML-ADSA the BLS-like properties of order- and grouping-independence and, for deterministic signers, byte-identical aggregates regardless of the sub-aggregation schedule. Unlike BLS, the Fiat–Shamir challenge binds the entire participant commitment, so finished aggregates are homomorphic but not freely mergeable: aggregation is a single-shot combine over contributions rather than an incremental product.

Security reduces, in both the ROM and the QROM, to the *same* assumptions as ML-DSA itself — de-

cisional Module-LWE, SelfTargetMSIS, and Module-SIS — at the NIST category of whichever parameter set is used (2/3/5 for -44/-65/-87), because the aggregate key is itself a bona fide ML-DSA key at that level (a sum of Module-LWE samples is a Module-LWE sample with the same parameters). The proofs are parameter-generic; they quantify over $(k, \ell, \eta, \tau, \gamma_1, \gamma_2, \omega)$ and instantiate to each level. We give a content-refresh layer that yields many-time security (advantage independent of the number of signed contents), an epoch Merkle key-tree for non-equivocation and accountability, and a registry/proof-of-possession layer for rogue-key resistance. We further prove that producing *any* member of the equivalence class of valid signatures for a fixed $(pk*, m)$ is as hard as a single ML-DSA forgery (no advantage from signature multiplicity), in ROM and QROM. The deployed default is the nonce-hiding F-OFFSET variant (§6b): signers broadcast only $(\mathrm{HighBits}(w_i), \mathrm{LowBits}(w_i)+r_i)$ rather than the full commitment, so secret recovery is the ≥native lattice problem (the basic combine exposes the per-content nonce by linear algebra); combined with a decision-bound deterministic nonce it is stateless many-time — a fixed committee reuses its keys for unlimited distinct decisions with no rotation, like an ordinary ML-DSA key — and a live decentralized multi-node devnet drives it end-to-end with byte-identical agreement across nodes.

The scheme is accompanied by 274 machine-checked lemmas (242 EasyCrypt + 32 Coq) across 43 prover artifacts (33 classical EasyCrypt + 5 quantum EasyPQC + 5 Coq/Rocq), plus 6 Gobra code-level theorems (tallies reproducible via `formal/count-artifacts.sh`), a reference implementation at all three parameter sets cross-validated against CIRCL and theQRL/go-qrllib, known-answer tests, a byte-identical AVX2 NTT kernel, and live multi-process demonstrations of decentralized aggregation and order/grouping independence. We discuss limitations — chiefly the need for independent cryptanalysis and a fully derived tight-QROM bound for the lossy (rejection-free) construction variant.

1.2 1. Introduction

1.2.1 1.1 The problem

Aggregate signatures compress many signatures on a common message into one short signature, a workhorse of consensus protocols (e.g., Ethereum's BLS-aggregated attestations), certificate transparency, and multi-party authorization. The dominant construction, BLS [BLS01, BGLS03], aggregates by multiplying group elements and is exquisitely simple — but it relies on pairings over elliptic curves, which Shor's algorithm breaks on a quantum computer. The NIST post-quantum standard for signatures, ML-DSA (Dilithium) [FIPS-204], is a Fiat–Shamir-with-aborts lattice scheme; crucially, ML-DSA signatures do not concatenate-combine — there is no public operation turning $\{\sigma_1, \ldots, \sigma_N\}$ on a common message into one short signature an unmodified verifier accepts. Post-quantum chains that port a BLS pipeline therefore fall back to *lists* of per-signer signatures (e.g., up to 128×4627 B $\approx$ 0.5 MB per aggregated attestation), losing the entire benefit of aggregation.

A line of work obtains *succinct proofs of knowledge of many signatures* (LaBRADOR [BS22] and its GPU implementations, Greyhound, lattice SNARKs/STARKs, zkVMs). These are powerful but heavy: they introduce a proof system, prover cost, and often a structured setup, and the output is a SNARK/STARK *about* signatures rather than a signature. Half-aggregation and lattice multi-signatures (MuSig-style two-round schemes, MuSig-L) reduce size or rounds but either remain linear-ish, require interaction, or need ROS/AGM-style assumptions.

We ask: can we get BLS-like, true aggregation — one constant-size signature an unmodified ML-DSA verifier accepts — with no trapdoor, no setup, no proof system, and no trusted aggregator, reducing to

exactly ML-DSA's assumptions? This paper answers yes.

1.2.2 1.2 The key idea: aggregation as module summation

ML-DSA signing produces `z = y + c·s1`, a response in `R_q^ℓ`, with a commitment `w = A·y` and a public key `t = A·s1 + s2`. These are *linear* in the secret. If `N` signers use a common challenge `c̃*` derived from the sum of their commitments `W* = Σ wᵢ`, then the sum of their responses, `z* = Σ zᵢ = (Σ yᵢ) + c̃*·(Σ s1ᵢ)`, is exactly the response of a *single* ML-DSA signer whose key is the summed key `t* = Σ tᵢ` and whose nonce is `Σ yᵢ`. The aggregate `σ* = (c̃*, z*, h*)` is therefore a syntactically and semantically valid ML-DSA signature under `pk* = (ρ, HighBits(t*))`. Aggregation is literally addition in **`R_q`**.

This is the lattice analogue of BLS's group product. BLS: `e(σ*, g) = e(H(m), Σ pkᵢ)` because the pairing is bilinear and the group is homomorphic. ML-ADSA: `Verify(pk*, m, σ*)` because `R_q` is a ring and the Fiat–Shamir relation is linear in the summed key. The two schemes share one algebraic engine — homomorphic aggregation over an abelian group/module — instantiated in a pairing group (BLS) versus a module lattice (ML-ADSA).

1.2.3 1.3 What summation buys, and the one thing it costs

Because Σ over a fixed multiset of ring elements is associative and commutative, ML-ADSA inherits BLS's order- and grouping-independence: two parties aggregating the *same* signer set in different orders, or combining different sub-group "aggregates of aggregates," obtain the byte-identical result (signers are deterministic; §4). We prove this and demonstrate it live across OS processes (§7).

The cost — the single structural difference from BLS — is mergeability. BLS aggregates are *freely mergeable*: `σ_{S∪T} = σ_S · σ_T` for disjoint `S, T`. In ML-ADSA the challenge `c̃*` is a hash of the *entire* `W*`; a partial aggregate over `S` was computed under `c̃*_S ≠ c̃*_{S∪T}`, so finished aggregates cannot be combined. Aggregation is a single-shot combine over contributions (commitments, then responses), not an incremental product. We show this is not merely a limitation but a *security feature*: challenge-binding the full participant set is exactly what defeats the Drijvers/ROS attack on two-round lattice multi-signatures, letting us prove concurrent security with no ROS/AGM/OMDL assumption (§6.4).

1.2.4 1.4 Contributions

1. A trapdoor-free, setup-free, proof-system-free post-quantum aggregate signature whose output is a byte-exact ML-DSA signature — at any of the three FIPS-204 parameter sets (Categories 2/3/5) — and whose verifier is the unmodified FIPS-204 verifier (§3, §5).
2. A tight security reduction to ML-DSA's own assumptions (decisional MLWE + SelfTargetMSIS) in the ROM, and a QROM reduction (tight for the perfect-masking construction), all machine-checked in EasyCrypt; the aggregate key is a genuine ML-DSA key, so Category-5 hardness is inherited exactly (§6).
3. Many-time aggregation via content-key refresh, with advantage provably independent of the number of signed contents — overcoming the per-key one-time limitation of naive Fiat–Shamir aggregation (§4, §6).
4. Decentralization without an aggregator: every combiner input is public, so any party reconstructs the identical aggregate; we formalize and prove order/grouping independence and an equivalence relation for aggregates, and prove that guessing any equivalent signature is as hard as ML-DSA in ROM and QROM (§6.5).

5. Accountability layers — an epoch Merkle key-tree (non-equivocation), registry + proof-of-possession (rogue-key resistance), and a ZKP-free decoy mechanism for signer-set privacy (§4, §8).
6. A reproducible assurance package: 260 machine-checked lemmas across 39 prover artifacts (+6 Gobra code-level theorems), a CIRCL/go-qrllib-anchored reference implementation, KATs, and live decentralized demonstrations (§7, §9).

1.2.5 1.5 Honest scope

ML-ADSA has not yet undergone independent third-party cryptanalysis, which we regard as the decisive gate for standardization. The QROM bound for the *rejection-free* (Construction B) variant derives the distinct-per-query GHHM21 adaptive-reprogramming form in-prover (`ml_adsa_qrom_ghhm.ec`, §6.6) from a proven per-round perfect resampling and the elementary distinct-query bound — it is no longer imported as the loose Zhandry semi-constant-distribution axiom (which is retained only as an independent cross-check); the perfect-masking (Construction A) variant is tight and unconditional. All three parameter sets (ML-ADSA-44/65/87, Categories 2/3/5) are instantiated in the reference implementation and CIRCL-cross- validated, and ACVP-shaped vectors are produced; the remaining gaps are external (independent cryptanalysis; formal ACVP registration). None of these affect the core claims; all are itemized in §10 and `docs/32`.

1.3 2. Related work

Pairing-based aggregation. BLS [BLS01], aggregate BLS [BGLS03], and its consensus deployments (Ethereum) define the target behavior — one short signature, unmodified verifier, public aggregation — but are not post-quantum.

Succinct proofs of many signatures. LaBRADOR [BS22] and follow-ups (GPU-accelerated provers, Greyhound), lattice SNARKs/STARKs, and zkVM approaches produce a *proof* of knowledge of `N` valid signatures. They achieve sublinear size but at the cost of a proof system, prover time, and sometimes setup; the output is not itself a signature. ML-ADSA is complementary: it is *true* aggregation with no proof system, at the price of being single-message (a common message) rather than `N` distinct messages. (LaBRADOR-style proof-of-aggregation was a *separate* technique we evaluated; it is not the construction here.)

Lattice multi/threshold signatures. Two-round lattice multi-signatures (MuSig-style, MuSig-L [BTT22], DOTT, Damgård et al.) target a *single* aggregate key set up interactively; many rely on ROS/AGM/OMDL or on a key-aggregation round. ML-ADSA is non-interactive after a one-time registration, derives the aggregate key by public summation, and proves concurrent security without ROS/AGM.

Threshold Raccoon [dPKM+24, EUROCRYPT 2024] is a `t`-of-`n` lattice *threshold* signature built on a Raccoon-style (masking-friendly, discrete-Gaussian) FSwA scheme: signers run a 3-round interactive protocol to jointly produce one signature (≈13 KiB, independent of `n` and `t`; ~40 KiB communication *per signer*; thresholds up to ~1024) verified by a *new* Raccoon-style verifier, with security under Hint-MLWE + SelfTargetMSIS (Hint-MLWE reduces to MLWE) and a trusted dealer for key generation (distributed key generation is left as future work). It answers a different question from ML-ADSA — a threshold over a *new* scheme requiring interaction and a dealer-distributed key — whereas ML-ADSA non-interactively aggregates *independent* ML-DSA-87 signatures into a single bona-fide FIPS-204 signature the *unmodified* verifier accepts, with no threshold key, no interaction beyond one-time registration,

and a constant 4627-byte output.

Fusion [GF23, eprint 2023/303] is the closest design point: a post-quantum one-time, non-interactively + permissionlessly aggregatable signature on Module-SIS over ideal lattices (structurally Dilithium-like, built on the Boneh–Kim aggregatable OTS), targeting ≥128 bits against forgery (tightness-adjusted) with 256-bit parameterizations. Fusion's aggregate is a *bespoke* object with its own verifier (a weighted linear combination, ~46–80 KB, logarithmic in #keys), and its conservative parameters provision the underlying lattice at ≈2× the target (e.g. ~521→256) because its linear-combination forgery extraction is *lossy*. ML-ADSA occupies the adjacent corner: its aggregate is a byte-exact ML-DSA-87 signature, so forgery extraction is native ML-DSA's *tight* SelfTargetMSIS step (no ≈2× provisioning), giving guaranteed forgery security ≈ the underlying core-SVP (252 classical / 229 quantum core-SVP; §6c). The trade is target level vs. compatibility and assurance: Fusion reaches a higher *parameter* category at the cost of FIPS-incompatibility and (to date) no machine-checked verification and an unaudited reference implementation; ML-ADSA caps at ML-DSA-87's Category 5 but is FIPS-204 drop-in, ~4.6 KB, and machine-checked. To match Fusion's 256-bit quantum target tightly, ML-ADSA would need only ≈1.12× ML-DSA-87's lattice dimension (a non-FIPS parameter set), versus Fusion's ≈2.28× — a direct consequence of the tighter reduction. (External figures to be re-verified against the primary source for the final submission.)

Synchronized aggregation. Chipmunk [FHSZ23, CCS 2023] (succeeding Squirrel [FSZ22]) is a *synchronized* lattice multi-signature (a SIS-based key-homomorphic one-time signature plus a homomorphic vector commitment): signers aggregate signatures on the same message at one time slot into a single aggregate (≈118 KB at 1024 signers, ≈136 KB at 8192, at 112-bit security — *not* constant-small), where each signer holds a large secret-key tree state (re-derivable from a seed; the public key itself is small) and aggregation runs in the ROM under (ring-)SIS. ML-ADSA does not require synchronization or per-period slots, allows *distinct* per-signer participation rather than a single common-message slot, keeps small ML-DSA keys, and produces a true 4627-byte ML-DSA-87 signature rather than a ~100 KB scheme-specific aggregate — at the cost of being single-common-message and "homomorphic-but-not-freely-mergeable." A fuller, source-checked head-to-head (sizes, trust model, assumptions, interaction, verification) is in `docs/35` §4.

Half-aggregation / sequential aggregation. Reduce size or assume sequential signing; generally not constant-size and not verifier-transparent.

Position. To our knowledge ML-ADSA is the first construction that is simultaneously (i) true constant-size aggregation, (ii) verified by the *unmodified* FIPS-204 verifier, (iii) trapdoor/setup/proof- system-free, (iv) decentralized (no aggregator), (v) many-time, and (vi) reduced — with machine-checked proofs — to exactly ML-DSA's assumptions in ROM and QROM.

1.4 3. Preliminaries

We use FIPS-204 notation (§2 of `docs/30`): ring `R_q`, public matrix `A ∈ R_q^{k×ℓ}` from `ExpandA(ρ)`, secret `(s1, s2)`, key `t = A·s1 + s2`, challenge `c = SampleInBall(c̃)` with `τ` nonzero ±1 coefficients, masking nonce `y` with `‖y‖_∞ < γ1`, commitment `w = A·y`, response `z = y + c·s1`, decomposition `Decompose/HighBits/LowBits`, `Power2Round`, hints `MakeHint/UseHint`, and `μ = H(H(pk) ‖ ctx ‖ m)`. Assumptions: decisional MLWE, SelfTargetMSIS (FIPS-204's forgery assumption), Module-SIS; plus a secure PRF and a collision-resistant hash.

Parameter sets. All three FIPS-204 sets share the base ring `q = 8380417`, `n = 256`, `d = 13`, and the

NTT root $\zeta = 1753$; only the module dimensions (k,ℓ), the secret/challenge distributions (η, τ), the masking/rounding magnitudes $(\gamma_1, \gamma_2, \beta{=}\tau\eta, \omega)$, and the challenge-hash length $\tilde{c}$ change. Because ML-ADSA depends only on the shared ring and Fiat–Shamir linearity, the *same* combine and the *same* proofs serve all three; ML-ADSA inherits each set's NIST category and its byte-exact ML-DSA sizes:

ML-ADSA set	NIST cat.	(k,ℓ)	η	τ	γ_1	γ_2	ω	$\tilde{c}$ (B)	pk (B)	sig (B)
ML-ADSA-44	2	(4,4)	2	39	2^{17}	$(q-1)/88$	80	32	1312	2420
ML-ADSA-65	3	(6,5)	4	49	2^{19}	$(q-1)/32$	55	48	1952	3309
ML-ADSA-87	5	(8,7)	2	60	2^{19}	$(q-1)/32$	75	64	2592	4627

The aggregate is a byte-exact ML-DSA signature at the chosen level, so the pk/sig columns are *also* the aggregate's sizes — constant in the committee size `N`. The reference implementation realizes all three (`go-mladsa Params44/65/87`, parameterized verifier/signer/aggregator), cross-validated against CIRCL's `mldsa44/65/87` (every aggregate accepted by their unmodified verifier; §9). Where a running example is helpful we use ML-ADSA-87 (Category 5), QRL 2.0's deployment target; the statements hold at each level with the table's substitutions.

1.5 4. Construction

ML-ADSA (Construction F) is layered.

L0 — core aggregate. The summation combine of §1.2 / §5; in the single-signer case it is exactly ML-DSA-87.

L1 — content-key refresh (many-time). A signer's only long-term secret is a 32-byte master seed `msk`. For each content `C`, the per-content key `sk_{C} = (s1,s2,t)` is derived by `s1‖s2 = ExpandS(PRF(msk, "F.refresh", C))`, `t = A·s1 + s2`, and the nonce by `y = DeriveNonce(msk, C, σ)` (a PRF stream, centered). Distinct contents give independent keys (PRF security), so aggregating across many contents is safe and the per-content security equals single ML-DSA (many-time, F-C4). Nonces are deterministic (no per-signature RNG; RFC-6979/EdDSA style), which eliminates RNG-failure key leaks and is what makes aggregates byte-reproducible — subject to a one-time discipline: a `(signer, C)` must sign at most once (else two responses share a nonce under two challenges → key recovery). With predictable `C = (slot, committee)` this is the one-vote-per-slot rule.

L2 — epoch key tree (non-equivocation, accountability). At epoch start a signer Merkle-commits all its `t_{C}` for the epoch's content range and signs the root with its registration key. A verifier accepts a `t_{C}` only if the root is registration-signed (`VerifyEpochRoot`) and `t_{C}` is a proven member (`VerifyContentInTree`). This binds every key used in an aggregate to a single commitment — defeating injected or equivocating keys (F-C8/C9).

L3 — registry + proof-of-possession (rogue-key). Each signer registers `(id, regpk, PoP)`; recognized weight is determined by the public registry. Non-registered signers have weight 0.

Decisions & decoys. The aggregate is decision-agnostic (YES/NO, multi-choice, approval, threshold, subset, weighted tally). A decoy is a weight-0 (non-registered) signer included to disguise the real signer set via refresh; because recognized weight is public, decoys cannot change outcomes (F-C7) and need no zero-knowledge proof. An optional hidden-value tally uses a homomorphic commitment (hiding = MLWE, binding = Module-SIS).

1.6 5. The decentralized combine (and the homomorphic-summation structure)

1.6.1 5.1 The combine

Two non-interactive rounds (commit, then respond) over public data; any party combines:

1. Each signer `i` publishes `(t_i, w_i)` where `w_i = A·y_i`.
2. Any party computes `t* = Σ t_i`, `(t1*,t0*) = Power2Round(t*)`, `pk* = (ρ, t1*)`, `W* = Σ w_i`, `μ = H(H(pk*) ‖ ctx ‖ m)`, `c̃* = H(μ ‖ w1Encode(HighBits(W*)))`, `ĉ = NTT(SampleInBall(c̃*))`.
3. Each signer `i` publishes `z_i = y_i + c·s1_i`.
4. Any party computes `z* = Σ z_i` and the public hint identity `rr2 = A·z* − c·t1*·2^d` (`= W* − c·s2* + c·t0*`, computable without secrets), sets `h* = MakeHint(−c·t0*, rr2)`, checks the FIPS-204 bounds, and outputs `σ* = (c̃*, z*, h*)`.

This is byte-identical to the secret-key procedure (it sources `rr2` from the public identity instead of `s2*`), so `Verify(pk*, m, σ*)` is the unmodified FIPS-204 verifier. There is no privileged aggregator: all of `t*, W*, c̃*, z*, h*` are functions of public values, so every party derives the same `σ*`.

1.6.2 5.2 Homomorphic aggregation: ML-ADSA as the additive dual of BLS

	BLS	ML-ADSA
aggregation operation	group product $\sigma^* = \prod \sigma_i$	module sum $z^* = \sum z_i$, $t^* = \sum t_i$, $W^* = \sum w_i$
algebraic structure	pairing group (abelian)	ring/module R_q (abelian)
key aggregation	$pk^* = \sum pk_i$ (group)	$pk^* = (\rho, \mathrm{HighBits}(\sum t_i))$ (module)
verifier	unmodified BLS verify	unmodified ML-DSA verify
order/grouping independence	yes (commutative product)	yes (commutative sum) — byte-identical for deterministic signers
free mergeability of finished aggregates	yes	no (challenge binds full W^*)
post-quantum	no (pairings)	yes (MLWE/MSIS)

The order/grouping independence is the *same* phenomenon in both schemes: the aggregate is a function of a sum/product over a fixed multiset, which is associative and commutative. In ML-ADSA the sums are over R_q and reduce mod q at each step; mod-q reduction preserves associativity (Z_q is a ring), so

a hierarchical "aggregate of aggregates" — summing sub-group partial sums in any order — yields the identical total and hence the identical σ∗. We make this precise and machine-relevant in §7.

The non-mergeability is the price of Fiat–Shamir: the challenge is a hash of W∗, so the response set is "locked" to the participant set at challenge time. This is the only place ML-ADSA departs from the BLS template, and §6.4 turns it into a security advantage.

1.7 6. Security

All theorems are machine-checked; lemma names/provers are in `docs/31`. We summarize statements and the reduction structure. The proofs are parameter-generic: they quantify over (k, ℓ, η, τ, γ_1, γ_2, ω) and instantiate to each of the three sets. Throughout, the aggregate key pk∗ = (ρ, HighBits(Σ t_i)) is a genuine ML-DSA key *at the cohort's parameter set*: a sum of MLWE samples is an MLWE sample with the same parameters, so the NIST category of that set (2 for -44, 3 for -65, 5 for -87) is inherited exactly. We state bounds in the level-agnostic form `adv_mlwe + Adv^{SelfTargetMSIS}`; substituting the set's parameters gives the concrete category. The running example is ML-ADSA-87 (Category 5).

1.7.1 6.1 EUF-CMA (keystone, ROM)

Theorem 1. For any forger A against ML-ADSA with one honest signer and a chosen-message signing oracle, `Adv^{EUF-CMA}(A)` $\leq$ `adv_mlwe + Adv^{SelfTargetMSIS}(B(A))`, a tight reduction to ML-DSA's two assumptions. *Proof structure:* a perfect HVZK-simulation hop (the signing oracle is simulated from public data, `masking_ok`), an MLWE key-indistinguishability hop (`mlwe_assumption`), and an extraction hop (`extract_sound`: a PoP-valid verifying forgery *is* a SelfTargetMSIS solution). EasyCrypt, `msufcma_uncond`, admit-free.

1.7.2 6.2 Strong unforgeability (ROM)

Theorem 2. `Adv^{SUF-CMA}` $\leq$ `adv_mlwe + Adv^{StrongSIS}`, where `StrongSIS = SelfTargetMSIS v Module-SIS` captures both fresh-message forgery and the same-`(μ,c)`-new-`(z,h)` collision. EasyCrypt `sufcma_uncond`.

1.7.3 6.3 Many-time (ROM)

Theorem 3 (F-C4). The advantage is independent of the number of signed contents Q, because content refresh makes each content an independent single-ML-DSA instance (PRF security). EasyCrypt `ml_adsa_F_manytime.ec`, `ml_adsa_F_refresh.ec`.

1.7.4 6.4 Concurrent security / ROS-resistance (ROM)

Theorem 4. `Adv^{concurrent} = Adv^{single-session}` with no ROS/AGM/OMDL assumption. The Drijvers/ROS attack on two-round lattice multisignatures needs the adversary to bias the aggregate challenge across concurrent sessions (Wagner's algorithm on a ROS system). Construction F's commit-then-reveal binds each cosigner's commitment before the challenge, making the challenge independent of the adversary's nonce (`unbiasable_challenge`, `challenge_adversary_independent`); the ROS system is unsolvable. EasyCrypt `ml_adsa_F_concurrent.ec`.

1.7.5 6.5 Equivalence-class hardness (ROM and QROM)

ML-DSA is not a unique-signature scheme: the equivalence class `class(pk*, m) = {σ : Verify(pk*,m,σ)}` may contain more than one element. Theorem 5. Producing *any* member of `class(pk*, m)` for an un-queried `m` (with a PoP-valid cohort) is, by definition, the EUF-CMA win, so it inherits Theorem 1's bound — the multiplicity of valid signatures gives the adversary no advantage. EasyCrypt `equiv_class_guess_bound` (ROM) and `qrom_equiv_class_uncond` (QROM). This is the formal content of "guessing an equivalent signature is as hard as a normal ML-DSA forgery, same bits."

1.7.6 6.6 Post-quantum (QROM)

Theorem 6 (Construction A, tight). `Adv^{QROM-EUF-CMA} ≤ adv_mlwe + Adv^{QROM-SelfTargetMSIS}` with no reprogramming/O2H loss for perfect masking. EasyCrypt `qrom_eufcma_uncond`. Theorem 7 (Construction B, lossy). A self-contained capstone keeps the masking + adaptive-reprogramming loss explicit (`qrom_eufcma_lossy`). Theorem 7′ (distinct-per-query reprogramming, derived). The Construction-B reprogramming loss is *derived in-prover*, not imported as a black box: the tight linear bound `reprog_ghhm(q_s,q_h) = q_s·(q_h+1)/|from|` is obtained by composing a *proven* per-round perfect-resampling equivalence (FunSamplingLib `LambdaReprogram.main_theorem` — a single fresh reprogramming is *exactly* indistinguishable, advantage 0, which is precisely why GHHM21 is tight) with the elementary distinct-query search bound (`T_Distinct.bound_distinct`) over `q_s` distinct, high-min-entropy commitment points, summed by a machine-checked hybrid. EasyCrypt `ml_adsa_qrom_ghhm.ec : ghhm_hybrid` (with `reprog_single_perfect`, `distinct_detect_bound`, `reprog_ghhm_ge0`). This replaces the imported Zhandry semi-constant-distribution quartic `(2q+2)⁴/6·λ²` with a proven per-round equivalence plus one elementary axiom, shrinking the QROM-B trust base; the quartic is retained only as an independent cross-check (`ml_adsa_qrom_reprog.ec`). Moreover the per-round hypothesis is discharged hypothesis-free (`ghhm_multiround_perfect`), and the signing-round program-equivalence — real-RO signer vs. reprogramming HVZK simulator — is itself machine-checked as a concrete module equivalence (`reprog_round_equiv`, `reprog_round_pr_eq`, modules `ReprogRound.roundReal`/`roundSim`, advantage 0). The only remaining input is the lattice response's HVZK-simulatability — which is *not a gap and not aggregation-specific*: it is ML-DSA's own perfect-HVZK property (`masking_ok = masking_perfect`), whose consequences are machine-checked both classically (`zero_leakage_perfect_A`) and quantumly (`sq_perfect`), with the aggregate-vs-single-signer reduction machine-checked too (Theorem of §6.7, `F_zk_per_content`). Going further, even ML-DSA's rejection-sampling change-of-variables — the finite-combinatorial kernel of perfect HVZK ("the accepted response is uniform on the norm window, independent of the secret shift `c·s1`") — is now machine-checked from first principles in `ml_adsa_masking.ec` (`reject_uniform`, `masking_perfect_concrete`), resting only on the trivial parameter bounds `0 ≤ β ≤ γ1−1`. The sole remaining unmodelled item is the bit-level *functional correctness of the NTT/rounding arithmetic*, the same boundary every ML-DSA formalization shares (mitigated here by byte-exact cross-validation against two independent FIPS-204 implementations). Thus both the *bound* and the *program-level reprogramming step* are mechanized, and the residual is exactly ML-DSA's own primitive, not a hole.

1.7.7 6.7 Other proved properties

Rogue-key collapse to single-forge (`ml_adsa_rogue_proof.ec`), no-new-power extraction to Module-SIS (`ml_adsa_nnp_proof.ec`), hiding from Module-LWE (`ml_adsa_F_hiding.ec`), decision integrity for weight-0 decoys (Coq `ml_adsa_F_decisions.v`), Merkle/provenance binding (Coq `ml_adsa_F_provenance.v`), and the N-fold reconstruction identity (Coq `ml_adsa_identity.v`).

1.8 6b. Hiding the aggregate nonce: the F-OFFSET instantiation

In the base decentralized combine, each signer broadcasts its full commitment $w_i = A \cdot y_i$. Because A is tall, an observer with w_i and the response $z_i = y_i + c \cdot s1_i$ recovers y_i by linear algebra and hence $s1_i = c^{-1}(z_i - y_i)$ — the per-content one-time key. This is contained by message-binding and the one-time discipline (a recovered key signs only the already-agreed message), but it does expose the nonce. F-OFFSET is the instantiation that removes the exposure while remaining a byte-exact ML-DSA aggregate, and it is our recommended, deployed default: a fixed committee combined with a message-bound deterministic nonce ($\sigma{=}3\beta$) is stateless many-time — it reuses its keys for unlimited distinct decisions with no rotation, like an ordinary ML-DSA key — and a live decentralized multi-node devnet drives it end-to-end with every node reconstructing the identical byte-exact aggregate (verified by an independent FIPS-204 verifier). The base full-w combine is retained only as the byte-exactness reference and fallback.

Construction. Each signer broadcasts only $\mathrm{HighBits}(w_i)$ (what a native ML-DSA signature already reveals) and a *noised* low part $q_i = \mathrm{LowBits}(w_i) + r_i$, where r_i is a fresh, secret, independent offset of width $\pm R$ — tolerated as noise, never subtracted. The combiner forms the aggregate challenge over the *estimated* high bits $w1* = \mathrm{HighBits}(\Sigma(\mathrm{HighBits}(w_i) \cdot \alpha + q_i))$ and, after responses, builds the FIPS-204 hint to *target* $w1*$ (each coordinate: keep the high bits, apply the one-step ± 1 hint, or — on the rare carry-miss that no ± 1 reaches — advance the content index and retry, exactly the rejection already native to ML-DSA). The verifier recomputes $\mathrm{UseHint}(h*, A \cdot z* - c \cdot t1* \cdot 2^d) = w1*$, the value hashed into the challenge, so it accepts byte-exact.

Security ($\geq$ native). Recovering $s1_i$ from $(\mathrm{HighBits}(w_i), q_i, z_i)$ reduces to an LWE instance $b = (A \cdot c) \cdot s1_i + e$ with $|e| \leq R$. This instance is the native key-recovery instance with *additional* fresh noise on the same secret-image, so by a noise-flooding reduction it is at least as hard as native against *any* attack (primal, dual, hybrid). Concretely, the Albrecht et al. `lattice-estimator` (full attack suite) gives native ML-DSA-87 at 267 bits versus the offset instance at 455 / 489 bits for $R = 2^{11} / 2^{12}$ (best attack `dual_hybrid` in both cases); the aggregate one-time key is harder still. The carry budget bounds $R \leq \sim 2^{12}$ ($\leq \omega$ carry-misses) and $\geq$-native bounds $R \geq \sim 3$, a wide operating window; the hint weight stays within ω for cohorts up to at least 32 signers, flat in cohort size. Forward-secret per-content keying (a hash-chain ratchet that erases spent epoch seeds) ensures a later compromise cannot recover the spent keys whose offset transcript is already public, and the fresh-per-content masking prevents cross-content accumulation.

The instantiation is machine-checked in the same corpus (`ml_adsa_F_offset.ec`: the noise-flooding reduction, the combiner's choice-rule correctness over the real high-bits/`UseHint` model, and the full combine loop as a Hoare-verified procedure) and exercised end-to-end (byte-exact verification, scale envelope, provenance, and a deterministic known-answer test). It is a drop-in alternative to the base aggregate with the same registry/ binding/one-time scaffolding.

1.9 6c. Leakage, the σ-independence of the deployed bound, and tightness-adjusted bit security

The deployed bound is independent of the nonce width. The deployed scheme publishes the full round transcript and so, in the worst case, leaks the entire per-content one-time key. We model exactly that

(the *key-leak* model): the signing oracle hands the adversary the whole one-time key, and the forgery target is an *un-queried* content whose key is fresh and never exposed. That target is a *no-message* (key-only) attack, which we prove is bounded by `adv_mlwe + STMSIS` without any HVZK/**`masking_ok`** term (`ml_adsa_F_keyonly.ec : konly_uncond`). Hence `Pr[deployed forge] ≤ adv_prf + Q·(adv_mlwe + STMSIS)`, and no term depends on the nonce width σ. The previous σ=12β figure was an artifact of the atomic-masking proof route, which the deployed scheme does not use.

Consequence — σ=3β default. Since σ does not enter the deployed security, it is chosen for the correctness norm wall, which *prefers* a smaller σ. We set the deployment default to $\sigma = 3\beta$ (`SigmaOffsetDefault`), raising the per-committee single-aggregate ceiling ≈8× (≈32 768 → ≈262 144) at no EUF cost, while staying ~120× above the recovery floor. The single published hint at σ=3β has recovery hardness 387 bits (real `lattice-estimator`), well above native.

Leakage register. Every value the aggregation *process* exposes beyond a bare signature is independently ≥ native: the offset broadcast (estimator 455–489), the response/hint (387), the aggregate key (352), and the combined three-view (= native). The key-leak model is a conservative upper bound on all of them; F-OFFSET's hiding leaks strictly less. Across unlimited aggregation cycles, the base-wallet root key is unrecoverable: recovering it from any number of leaked per-content keys reduces to breaking the refresh PRF, with advantage `adv_prf + p_coll` independent of the cycle count (`ml_adsa_F_rootsafe.ec`).

Tighter (non-key-leak) model and ZK-parity. For the stricter model where the per-content key is *not* treated as leaked, we machine-check that the residual signing-simulation gap equals a Hint-MLWE distinguishing advantage (`ml_adsa_F_hintmlwe.ec`: the response `z = c·s1 + y` is a Gaussian-noised secret inner product), giving `Pr[forge] ≤ adv_hint + adv_mlwe + STMSIS` and, as an optional configuration, computational zero-knowledge parity with a native signature under the (published) Hint-MLWE assumption. The single-hint regime (one-time refresh) places σ in the mild KLSS parameter range (floor ≈0.5β).

Tightness-adjusted security, classical/quantum. Because the aggregate *is* a byte-exact ML-DSA signature, its forgery extraction is native ML-DSA's *tight* SelfTargetMSIS step, and the key-leak model adds no forking/reprogramming loss. So the guaranteed forgery security ≈ the underlying core-SVP, in either cost model: ML-ADSA-87 = 267 (gate-count) / 252 (core-SVP) classical, and 217 (ChaLoy21) / 229 (core-SVP) quantum — not halved by the reduction. (Contrast a forking/linear-combination aggregate, which provisions ≈2× underlying for the same guarantee.) This is the honest apples-to-apples basis for comparison with bespoke aggregate schemes (§2).

1.10 7. Order- and grouping-independence: statement and live proof

Proposition. For deterministic signers over a fixed final signer set `S` and content `C`, the aggregate `(pk*, σ*)` is a function of the multiset `{(t_i, w_i, z_i) : i ∈ S}` through sums only; hence it is byte-identical under any permutation of `S` and any hierarchical partition of `S` into sub-groups whose partial sums are later summed.

Proof. `pk*`, `W*`, `z*` are Σ over the multiset, associative and commutative under mod-q; `c̃*`, `h*`, and the encoding are deterministic functions of those sums; nonces are deterministic. The partial sums of a partition differ, but their sum collapses to the identical total (exactly as `(1+2+3)+(4+5) = 6+(7+8)`), and σ* depends only on the total. ■ (Go test `TestHierAgg_PartialsDiffer_TotalIdentical`: partials differ, total/σ* byte-identical.)

Live demonstration. `cmd/mladsa-hieragg` runs N real `user` processes (each a distinct ML-DSA key) and two independent `aggregator` processes that combine the same users via *different* sub-group partitions and *different* orders; a `judge` process independently re-derives `pk*` from the public commitments, byte-compares both aggregators' `σ*`, and re-verifies both with go-qrllib's native verifier — exiting non-zero on any mismatch. Result: byte-identical `σ*`, both natively verified. Negative controls (dropping a signer; tampering a published `σ*`) are correctly rejected.

1.11 8. Decentralized deployment and accountability

We integrate ML-ADSA into a post-quantum proof-of-stake consensus client (the QRL 2.0 `qrysm` fork), replacing the per-attester signature *list* with one constant-size aggregate. Each node is beacon + validator; aggregation is the existing permissionless rotating duty, but the combine is the §5 decentralized procedure, so every node derives the identical `σ*` (no trusted aggregator). The epoch key-tree is populated at epoch boundaries from on-chain registration keys (the trust anchor) plus a gossiped, validated commitment cache; an injected/equivocating key is rejected and the signer excluded while the honest set still aggregates (live: `cmd/mladsa-epochnet`, `cmd/mladsa-devnet`). Compression vs the list is constant 4627 B regardless of committee size (8–128× at typical/maximum committee sizes; e.g. a 16-member committee with 12 attesters compresses 12×), with no needed information lost: signer set = the public aggregation bits, key = Σ epoch-tree `t_i`, validity = one FIPS-204 verify. Appendix A gives the full case study: methodology, measured compression and agreement, the live epoch-key-tree authentication, the adversarial and no-leakage validation, and the engineering lessons.

1.12 8b. Performance (measured, reference implementation)

On Apple M5 (reference Go), the per-content refresh is ~0.10 ms, single-signer aggregation ~0.40 ms, and `N=128` aggregation ~24 ms at ML-ADSA-87 (one-time per slot, split across the committee in deployment); the -44/-65 levels are proportionally cheaper (smaller (k,ℓ)). Verification is **`O(1)`** in **`N`** — one unmodified ML-DSA verify (~0.14 ms native go-qrllib at -87) — against the constant aggregate sizes of the chosen level (2420/3309/4627-byte signature and 1312/1952/2592-byte aggregate key for -44/-65/-87), versus an `O(N)`-byte / `O(N)`-verify per-attester list (e.g. 128× smaller at a 128-member committee). Full tables, allocations, and reproduction commands are in `docs/33`; ACVP-style KAT vectors generated by the reference implementation (digests matching the pinned KATs) are in `vectors/`. The cost-dominant NTT has a byte-identical AVX2 kernel (Montgomery-domain `VPMULDQ` reduction; enabled on AVX2 CPUs, differentially validated against the generic kernel under `docker linux/amd64`); native-x86 timing is the one open perf item (§10).

1.13 9. Implementation and assurance

The reference implementation (`go-mladsa/`) is cross-validated against CIRCL and theQRL/go-qrllib at all three parameter sets: every aggregate is a byte-exact ML-DSA `(pk, sig)` pair their unmodified verifiers accept — `(1312, 2420)` at -44, `(1952, 3309)` at -65, `(2592, 4627)` at -87 (the parameterized signer/verifier/aggregator are exercised against CIRCL's `mldsa44/65/87`; the -87 path is additionally the legacy reference and is asserted byte-identical to it). Assurance has three surfaces (full traceability in `docs/31`): (i) algorithm-level machine-checked proofs — 260 lemmas (228 EasyCrypt + 32 Coq) across 39 artifacts, plus 6 Gobra code-level theorems, spanning EasyCrypt (classical + QROM via the

EasyPQC fork), Coq, and Gobra; (ii) implementation conformance — KATs and CIRCL/go-qrllib cross-checks; (iii) code-level structural proofs — Gobra theorems for the Merkle/one-time/framing/decision-linearity invariants. We are explicit about boundaries: the machine-checked proofs are about the algorithm/model; Go conformance is measurement/testing (plus the Gobra structural theorems). The perfect-HVZK masking change-of-variables — historically the one axiomatized "lattice fact" — is now derived (`ml_adsa_masking.ec`); the only remaining lattice content not source-proved is the bit-level NTT/rounding *functional correctness*, which is cross-checked byte-for-byte against two independent FIPS-204 implementations rather than fully source-proved (the shared Formosa-Crypto-scope boundary).

1.14 10. Limitations and open problems

1. Independent cryptanalysis — none yet; the decisive standardization gate.
2. Tight QROM-B — the rejection-free variant's distinct-per-query GHHM21 reprogramming bound is now *derived in-prover* (Thm 7′, `ml_adsa_qrom_ghhm.ec`) from a proven per-round perfect-resampling equivalence plus the elementary distinct-query bound; the per-round bound is discharged hypothesis-free (each round's gap is *proven* 0), so nothing about the bound is assumed; the signing-round program-equivalence (real-RO signer ~ reprogramming HVZK simulator) is itself machine-checked (`reprog_round_equiv`), leaving only the lattice response's HVZK-simulatability (the `masking_ok` axiom shared with the rest of the proof) factored out. Construction A is tight and unconditional.
3. Parameter sets — all three (ML-ADSA-44/65/87, Categories 2/3/5) are now instantiated in the reference implementation and cross-validated against CIRCL's `mldsa44/65/87`; the parameter-generic proofs cover each level. (Formal ACVP *registration* with NIST remains an external step.)
4. Optimized implementation — the cost-dominant NTT has a byte-identical AVX2 kernel (Montgomery-domain `VPMULDQ`), differentially validated under `docker linux/amd64`; the open items are a native-x86 timing/throughput measurement and a fully constant-time-hardened build (the NTT's data-independence is already machine-backed; §2 of `docs/34`).
5. Single common message — like BLS aggregate-on-common-message; distinct-message aggregation is out of scope (that is the regime of proof-of-aggregation / LaBRADOR).
6. F-OFFSET (nonce hiding) — the offset instantiation (§6b) is machine-checked for its core lemmas (the noise-flooding reduction and the combine-loop correctness over the real high-bits model) and validated end-to-end, but its EC port abstracts the q-ring arithmetic and `UseHint`'s mod-`m`/boundary at the same level as the base scheme's rounding model, and its absolute hardness figures use the estimator's gate-count model (the relative $\geq$-native claim is reduction-backed and calibration-robust). Integration into a live consensus transport is future work.

1.15 11. Conclusion

ML-ADSA shows that post-quantum aggregate signatures need neither a proof system nor a trapdoor nor a trusted aggregator: the linearity of Fiat–Shamir-with-aborts already provides BLS-style homomorphic aggregation, with module summation playing the role of the pairing-group product. The aggregate is a real ML-DSA signature, security reduces to ML-DSA's own assumptions in ROM and QROM, and the construction is many-time, decentralized, and accountable. The work is backed by machine-

checked proofs, an anchored implementation, and live demonstrations. What remains for standardization is external scrutiny — which we invite.

1.16 References (to be completed for submission)

[BLS01] Boneh, Lynn, Shacham. *Short Signatures from the Weil Pairing.* ASIACRYPT 2001. [BGLS03] Boneh, Gentry, Lynn, Shacham. *Aggregate and Verifiably Encrypted Signatures…* EUROCRYPT 2003. [FIPS-204] NIST. *Module-Lattice-Based Digital Signature Standard.* 2024. [BS22] Beullens, Seiler. *LaBRADOR: Compact Proofs for R1CS from Module-SIS.* 2022. [BTT22] Boschini, Takahashi, Tibouchi. *MuSig-L: Lattice-Based Multi-Signature with Single-Round Online…* CRYPTO 2022. [GHHM21] … adaptive reprogramming in the QROM. [dPKM+24] del Pino, Katsumata, Maller, Mouhartem, Prest, Saarinen. *Threshold Raccoon: Practical Threshold Signatures from Standard Lattice Assumptions.* EUROCRYPT 2024 (LNCS 14652, pp. 219–248); ePrint 2024/184. [FHSZ23] Fleischhacker, Herold, Simkin, Zhang. *Chipmunk: Better Synchronized Multi-Signatures from Lattices.* ACM CCS 2023; ePrint 2023/1820. [FSZ22] Fleischhacker, Simkin, Zhang. *Squirrel: Efficient Synchronized Multi-Signatures from Lattices.* ACM CCS 2022; ePrint 2022/694. [Dilithium] Ducas et al. *CRYSTALS-Dilithium.* (Formosa-Crypto machine-checked ROM proof, eprint 2023/246.) *(Full bibliography — including the lattice-multisig and proof-of-aggregation lines surveyed in §2 — to be finalized against the ePrint template.)*

1.17 Appendix A. Case study: ML-ADSA in a post-quantum consensus client (QRL 2.0)

This appendix is an experimental report. Section 8 summarized the deployment; here we give the methodology, the measured results, the threat-model demonstrations, and the engineering lessons in full, so the case study is reproducible and the claims are auditable. All numbers below come from running code in the `qrysm` fork (the QRL 2.0 beacon client, itself a fork of Ethereum's Prysm); the live harnesses are `cmd/mladsa-devnet`, `cmd/mladsa-hieragg`, and `cmd/mladsa-epochnet`, and the in-package tests under `mladsa/`, `mladsaconsensus/`, and `mladsalegacy/`.

1.17.1 A.1 The problem in the host system

QRL 2.0 is a post-quantum proof-of-stake chain whose validators sign attestations with ML-DSA-87. Like every naïve post-quantum port of a BLS pipeline, the upstream client cannot combine signatures, so it keeps all of them: an `Attestation` carries a `repeated bytes signatures` field with one 4627-byte signature per attester, up to a 128-member committee — i.e. up to `128 × 4627 ≈ 592 KB` of signature per attestation, with the load-bearing invariant `len(signatures) == popcount(aggregation_bits)` woven through SSZ hash-tree-root, indexed-attestation conversion, the verification batch, slashing, and the gossip topic. This `O(N)` signature payload is exactly what an aggregate removes, and removing it is the real integration milestone (audit finding H5).

The aggregation step in the upstream client is *not cryptographic*: a rotating, permissionless aggregator (selected when `LE(hash(selection_proof)[0:8]) mod max(1, committee/16) == 0`) ORs the attesters' `aggregation_bits` together and appends their signatures. ML-ADSA changes *only this combine step*: the node/validator topology, the rotating-aggregator selection, and the verification entry points are unchanged.

1.17.2 A.2 What was integrated

We replaced the signature list with one ML-ADSA aggregate produced by the §5 decentralized combine (`mladsa.CombineFromPublic`) over the committee's *public* per-content contributions `(t_i, w_i, z_i)`. The integration was done in two phases:

- Phase 1 (`mladsaconsensus/`) carries exactly one aggregate in the existing field, with no proto regeneration — a drop-in that demonstrates the compression and verification against a reconstructed aggregate key `pk∗` while leaving the wire format untouched.
- Phase 2 makes the type honest: the proto becomes a single fixed `bytes signature = 3 [ssz_size 4627]` for both `Attestation` and `IndexedAttestation`; the SSZ hash-tree-root layout changes accordingly (a hard fork); the per-index verification loop in `attestation_utils.go` and `signature.go` collapses to *one* aggregate verify whose `pk∗` is reconstructed from the attesting validators' epoch-key-tree keys via the resolver seam `attestation.AggregateAttestationPubkey`; and the BLS-style max-cover signature-splice in `aggregation/attestations/` is removed (finished aggregates cannot be byte-merged — the challenge `c̃∗` binds the full `W∗`; merging distinct aggregates returns `ErrMLADSADistinctAggregates`). The whole non-test module builds clean.

Two design choices made the integration decentralized and trustless rather than a centralized optimization. First, the combine is the proven path: `TestDecentralized_EqualsAggregateF` asserts that `CombineFromPublic` is *byte-identical* to the formally-verified `AggregateF`, so the deployed code is the code the proofs are about. Second, every input the combiner needs is public — `t_i` is committed in the epoch key-tree, `w_i`/`z_i` are HVZK-simulatable, and the hint `h∗ = MakeHint(−c·t0∗, A·z∗ − c·t1∗·2^d)` uses only public data — so each node *independently* derives the byte-identical `σ∗` with no intermediary aggregator. The verifier seam is nil by default, so verification can never silently accept without the authenticated epoch key tree (closing audit risk H4-full).

1.17.3 A.3 Compression results (measured)

The aggregate is constant-size (4627 B) regardless of committee size; the only per-attester datum that remains is the `aggregation_bits` field (≤16 B for 128 members), which is already present upstream and is exactly the public information the verifier needs to select the `t_i` and rebuild `pk∗`.

Committee attesters `N`	Upstream signature list	ML-ADSA aggregate	Reduction
8	37,016 B	4,627 B (+1 B bits)	8×
16	74,032 B	4,627 B (+2 B bits)	16×
64	296,128 B	4,627 B (+8 B bits)	64×
128 (max)	592,256 B	4,627 B (+16 B bits)	128×

(The 8- and 16-node rows are from the live multi-process devnet; the 64/128 rows from the in-package `TestConsensus_∗` and `TestAggregateAttestation_SingleSigCompression`.) No needed information is dropped: the signer set is recovered exactly from the bits, the key is `Σ` of the epoch-tree `t_i`, and validity is one FIPS-204 verify. The only thing dropped is the *individual* signatures — exactly as BLS aggregation drops them in Ethereum. Rewards/participation are driven by the bits; attester-slashing matches BLS semantics (the evidence is two conflicting attestations + their bitfields); and targeted "validator X signed Y" attribution, if ever needed, is available via Construction-F provenance openings.

1.17.4 A.4 Decentralization and live agreement

In the live devnet (`run-devnet.sh`), `N` independent OS processes — each a full beacon+validator node — gossip over a shared bulletin directory; each signs its attestation, self-combines from the public contributions, and verifies. Every node derives the same $\sigma*$ (`all-nodes-agree=true`). For committees of size $\leq$16 every node is an aggregator (the selection modulus is 1), so all nodes independently combine, which is the strongest form of the decentralization claim. Every aggregate is verified by *both* our in-house verifier (CIRCL-equivalent, proven in `go-mladsa`) *and* QRL's own `go-qrllib/crypto/ml_dsa_87.Verify` — consensus accepts the aggregate with the verifier it already ships, because the aggregate *is* a byte-exact FIPS-204 ML-DSA-87 signature.

1.17.5 A.5 Epoch key-tree authentication (live)

Many-time security requires that each per-content key `t_i` be authentic. Each node commits *all* its one-time per-content keys for the epoch in a Merkle tree, signs the root with its registration key to produce `σ_reg`, and gossips `(kt_root, σ_reg)`. Every node verifies `σ_reg` at registration (`VerifyEpochRoot`); each per-slot commit then carries the Merkle path proving its `t_i` is the *committed* key for that content (`VerifyContentInTree`, property F-C9). Crucially, no new beacon-state field is required: the on-chain registration pubkey is the trust anchor, and `σ_reg` binds the off-chain, p2p-published epoch root to that on-chain identity. The store is populated at each epoch boundary directly from the real `ReadOnlyBeaconState` (`BuildEpochKeyTreeStore` iterates the active validator set), and the gossip cache (`GossipEpochKeyCache`) *rejects at ingest* any bundle whose root isn't signed by the claimed key or whose content keys aren't committed members of the root.

Live (`run-epochnet.sh`): validators publish bundles and contributions; a node ingests/validates them, builds a `BeaconStateZond` from the published registration keys, installs the resolver, combines, and verifies through `attestation.VerifyIndexedAttestationSigs` (with `pk*` from the authenticated tree) and go-qrllib native. All-honest → verified (exit 0); with a forged validator → its bundle is rejected at ingest and it is excluded, and the honest set *still* produces a valid, verified aggregate (resilience, not denial). The end-to-end trust chain runs unbroken: on-chain registration pubkey → `VerifyEpochRoot` → `VerifyContentInTree` → `AggPkStar` → FIPS-204/go-qrllib verify.

1.17.6 A.6 Adversarial validation (fakes cannot change the result)

The harnesses include negative controls with teeth:

- Sybil flood. `TestSybil_FakesCannotChangeResult`: 1000 non-members are added to the cohort; the aggregate is *byte-identical* (`P ⊆ REG`, weight-0, property F-C7). Live, sybil nodes carry a bad proof-of-possession and are excluded at registration — they can re-derive and verify the honest $\sigma*$ but never alter it.
- Rogue key. A member with a tampered PoP is rejected by every node (F-C5).
- Equivocation / uncommitted key. Live (`EQUIV=1`): the equivocator (node 8) publishes a `t_i` not in its signed epoch tree; every node logs `EXCLUDE id 8 (t_i_C not in its epoch tree — F-C9)`, participants drop to 7/8, and *all* nodes — including the equivocator — still agree on the identical $\sigma*$ over the honest 7. The attacker cannot inject a key it did not commit.
- Forgery. A tampered or fabricated (keyless) signature is rejected by both our verifier and go-qrllib's.
- Wrong key set. `TestSybil_WrongKeySetCannotForge`: substituting a fabricated member's per-content key into the reconstructed set fails to verify, because `pk*` binds the *exact* attesting set.

1.17.7 A.7 Statistical no-leakage check

Against the rotated construction (each content a fresh one-time key — the point of the many-time refresh), `TestBreakIt_NoSecretLeakage` measures: max `|corr|` between distinct per-content keys = 0.15 (≈ the noise floor); correlation of a key recovered by averaging over 1500 rotated attestations with a one-time key = 0.02 (≈ 0); a second signature for the same content is refused (samples-per-key capped at 1, so accumulation is structurally impossible); and the Hellinger/total-variation distance between two keys' response distributions = 0.0026 over 5.4M samples — i.e. the optimal distinguisher's advantage is 0.26%. This is the empirical counterpart of the machine-checked zero-leakage (A3/F-C13) and many-time refresh (F-C4); it is not a substitute for lattice cryptanalysis, which reduces to MLWE+SelfTargetMSIS (proved in `formal/`).

1.17.8 A.8 Order-independence of aggregate-of-aggregates (live)

`cmd/mladsa-hieragg` is a multi-process demonstration that hierarchical aggregation is order- and grouping-independent. Real `user` processes (each a distinct ML-DSA key) publish genuine §5 commitments and responses; two independent `aggregator` processes perform a *hierarchical* combine — summing each sub-group's responses into a partial via `mladsa.CoeffSumL`, then summing the partials — using different partitions and different orders (Alice `[[1,2,3],[4,5],[6,7,8]]` order `0,1,2`; Bob `[[6,7],[8,1,2],[3,4,5]]` order `2,1,0`); a `judge` process independently re-derives `pk∗` from the public commitments, byte-compares both aggregators' `σ∗`, and re-verifies both with go-qrllib's native verifier. Result: byte-identical `σ∗` and `pk∗`, both natively verified. Negative controls (a byzantine aggregator that drops a signer; one that publishes a tampered `σ∗`) are correctly rejected. This is the only sound notion of "aggregate of aggregates" for ML-ADSA: finished aggregates cannot be merged (the challenge `č∗` depends on the full `Σ w_i`), but the underlying *contributions* sum associatively and commutatively mod `q`, hence the byte-identical total.

1.17.9 A.9 Backward compatibility and equivalence

Because the Phase-2 wire change drops the per-attester list, and ML-ADSA loses BLS's free mergeability, the transition is made non-destructive by a parallel validator (`mladsalegacy/`): `VerifyLegacyConcatenated` reproduces the *exact* former upstream per-attester verification, extracted verbatim from the pristine upstream tree; `UnmarshalLegacyAttestationSSZ` decodes the old wire layout; and `DetectFormat` distinguishes legacy bytes (first SSZ offset 136) from new single-aggregate bytes (first offset 4759), so a node runs both validators and routes historical/in-flight aggregates correctly. Atop that, since byte-canonicality no longer holds across all aggregation paths, statement-equivalence predicates (`SameStatement` / `EquivalentNew` / `EquivalentAcross`, and `mladsaconsensus.SignaturesEquivalent`) decide whether two aggregates certify the *same signer set over the same `AttestationData`* — across both the legacy-concat and ML-ADSA forms — while `ByteIdentical` captures the strong deterministic case. The formal equivalence-class result (`equiv_class_guess_bound`, and its QROM analogue) shows that the multiplicity of valid signatures gives an attacker no advantage: producing *any* member of `{σ : verify pk∗ m σ}` is, by definition, the EUF-CMA win, so the security is exactly ML-DSA-87 Category 5.

1.17.10 A.10 Engineering lessons

The integration surfaced several reproducible lessons worth recording for anyone porting an aggregate signature into a Prysm-class client:

1. The invariant, not the field, is the work. `len(signatures) == len(indices)` is woven through SSZ merkleization, indexed-attestation conversion, the verification batch, slashing, and gossip; replacing the *field* is mechanical, but replacing the *invariant* (one aggregate, signer set recovered from the bits) is what touches every subsystem.
2. A nil-by-default verifier seam is the safe migration. Wiring the `pk*` resolver as a seam that *refuses* until the epoch key tree is installed prevents the hot path from silently accepting an unauthenticated aggregate during a partial migration.
3. The combine must be the proven object. Reusing `CombineFromPublic` (byte-identical to `AggregateF`) rather than re-deriving an "optimized" combine in the client keeps the deployed code inside the scope of the machine-checked proofs.
4. Reproducing Bazel codegen standalone is feasible but fiddly. The proto/SSZ regeneration required rebuilding `protoc-gen-go-cast` against a current `protobuf`, and running `sszgen` over a temp dir of just the `.pb.go` files (it chokes on the gateway files' duplicate `context` import) — documented so the hard fork's generated code is reproducible without the Bazel sandbox.

1.17.11 A.11 What remains in the host system

Honest scope for the deployment (orthogonal to the algorithm's open problems in §10): the shared-directory gossip in the devnet is a stand-in for qrysm's libp2p topics (production wiring is the remaining integration step); batch ML-ADSA verification with gossip rate-limiting (audit H6) is heavier than BLS and is designed but not yet load-tuned; and the broader `_test.go` suite across the client still uses the old field and needs the same mechanical migration. None of these affect the result demonstrated here: QRL 2.0 consensus can replace its up-to-592 KB attestation-signature list with one 4627-byte aggregate, derived identically by every node, verified by the chain's own ML-DSA-87 verifier, with fakes provably powerless and no measurable secret leakage — keeping QRL's exact node/validator/rotating-aggregator structure and changing only the combine.