TensorCash Post-Quantum Spending: Witness v2 + ML-DSA from Genesis

Imosuke Takakuni[1]

2 May 2026

ABSTRACT

TensorCash is a **genesis fork**, not a Bitcoin upgrade. There is no installed base of UTXOs to retrofit, no soft-fork activation window, no backwards-compatibility tax. We took that freedom to ship two coexisting spending stacks from block 0: the familiar ECDSA / Schnorr stack on Witness v0 / v1, and a **post-quantum stack on a new Witness v2**, built around the NIST FIPS 204 standard ML-DSA (Module-Lattice-Based Digital Signature Algorithm). A v2 output is a 32-byte Taproot output that can only be spent by a script-path proof — key-path is disabled by consensus — and the only new script opcodes are `OP_CHECKMLSIG` and `OP_CHECKMLSIGVERIFY` (the canonical wallet template uses the VERIFY form). This document describes the encoding, the consensus rules, the address format, the wallet plumbing, and the threat model. Because both rails are live from height 0, there is no migration. Users pick a rail when they fund an address; both remain valid forever.

I. Why bake PQ in at genesis

A few reasons specific to a genesis fork:

- **No legacy outputs to rotate.** Bitcoin will eventually face a multi-year, contentious migration of P2PK / early-Taproot UTXOs whose raw pubkeys are exposed on chain. We start with zero such outputs. There is no "vulnerable balance" to drain into a quantum-safe address — the user simply funds whichever rail they prefer.
- **Wallet teams can make a one-time choice.** A custodian designing a TensorCash hot/cold setup decides today whether each wallet is on v0/v1 (mature ecosystem, ~64-byte signatures, HD-derivable) or on v2 (PQ-safe, 2.4–4.6 KB signatures, independent keys). The chain accepts both; the choice is operational, not protocol-level.
- **Consensus simplicity.** Because v2 is consensus-active from height 0, there are no deployment bits, no `version-bits` warnings, no "discourage-upgradable-witness-program" softeners. Either a witness v2 output passes the v2 rules (§4) or the transaction is rejected. End of story.
- **PoW + script decoupling.** TensorCash's mining work (inference + Wesolowski VDF; see the Core-Node whitepaper) is independent of the signature scheme used in transactions. Adding ML-DSA touches the *spending* side only: the proof blob, the VDF, the SPV sidecars and the reorg advisory are all untouched.

1. "Imosuke Takakuni" is a pseudonym. Contact: takakuni@tensorcash.org

II. ML-DSA in one page (FIPS 204)

ML-DSA is the lattice-based digital signature scheme NIST standardized in August 2024 as FIPS 204. Security reduces to the hardness of finding short vectors in module lattices, a problem believed to remain hard even against a large-scale quantum adversary running Shor's or Grover's algorithm.

Three parameter sets are exposed (`bcore/src/crypto/mldsaverify.h:17-21`):

```
enum class ParamSet : uint8_t {
    MLDSA_44 = 0x2C,  // NIST Security Level 2 (~128-bit quantum)
    MLDSA_65 = 0x41,  // NIST Security Level 3 (~192-bit quantum)  ← default
    MLDSA_87 = 0x57,  // NIST Security Level 5 (~256-bit quantum)
};
```

Sizes per parameter set, in bytes (`mldsaverify.h:33-43`, `mldsakeygen.h:21-23`):

Set	Public key	Secret key	Signature	Quantum security
ML-DSA-44	1 312	2 560	2 420	≈ AES-128
ML-DSA-65	1 952	4 032	3 309	≈ AES-192
ML-DSA-87	2 592	4 896	4 627	≈ AES-256

The signing wrapper signs the 32-byte TensorCash Taproot sighash with the FIPS 204 ML-DSA implementation exposed by liboqs:

```
// crypto/mldsakeygen.h
bool MLDSA_Keygen(ParamSet level,
                  std::vector<uint8_t>& pk_out,
                  std::vector<uint8_t>& sk_out);

bool MLDSA_Sign(std::span<const uint8_t> sk,
                std::span<const uint8_t> msg32,    // 32-byte sighash
                ParamSet level,
                std::vector<uint8_t>& sig_out);
```

The build enables ML-DSA when it can find liboqs headers and library support (`ENABLE_MLDSA`). Operational deployments should pin a liboqs version that contains the standardized ML-DSA algorithms and should verify signatures against known test vectors; the current CMake probe detects liboqs presence but does not enforce a minimum version. If liboqs is absent, key generation, signing, and verification compile as disabled stubs, so a node cannot follow a chain containing v2 spends rather than accepting unverified signatures.

III. The on-stack encoding for ML-DSA pubkeys

Bitcoin tapscript pushes raw 32-byte x-only pubkeys. ML-DSA pubkeys are 1.3 to 2.6 KB, and their length depends on the parameter set, so we use a self-describing encoding (`mldsaverify.h:54-57, 71-75`):

```
pk_blob = alg_id (1 byte, 0x01)
       || level  (1 byte, 0x2C / 0x41 / 0x57)
       || varint(pk_len)
       || pk_bytes      // exactly 1312 / 1952 / 2592 bytes
```

`ParsePublicKey` enforces:

- the algorithm identifier is exactly `ALG_ID_MLDSA = 0x01`,
- the level byte matches one of the three legal `ParamSet` values,
- the varint decodes to **exactly** the canonical public key length for that level (off-by-one is a hard reject),
- the varint is itself **minimally encoded** — no `FD 03 00` for a value that fits in one byte, etc.

The strictness matters because the parser sits inside the script interpreter and is therefore consensus-critical: a malleable encoding here would fork the chain on the first non-canonical witness.

IV. Witness version 2 — script-path-only Taproot

Witness v2 outputs look exactly like v1 Taproot at the scriptPubKey layer:

```
scriptPubKey:  OP_2 <32-byte program>
```

The 32 bytes are the Taproot **output key**, the standard internal-key-tweaked-by-Merkle-root construction. What changes is the spend-time semantics, encoded in the script interpreter (`bcore/src/script/interpreter.cpp:2296-2339`):

```
} else if (witversion == 2 && program.size() == WITNESS_V1_TAPROOT_SIZE && !is_p2sh) {
    // Taproot v2 (script-only): 32-byte non-P2SH witness v2 program.
    // Semantics mirror Taproot, except key-path spending is disabled by consensus.
    if (!(flags & SCRIPT_VERIFY_TAPROOT_SCRIPT_ONLY)) return set_success(serror);
    if (stack.size() == 0) return set_error(serror, SCRIPT_ERR_WITNESS_PROGRAM_WITNESS_EMPTY);

    execdata.m_witness_version = 2;     // unlocks larger stack elements

    // Drop optional annex (same wire layout as v1)
    if (stack.size() >= 2 && !stack.back().empty() && stack.back()[0] == ANNEX_TAG) {
        const valtype& annex = SpanPopBack(stack);
        execdata.m_annex_hash = (HashWriter{} << annex).GetSHA256();
        execdata.m_annex_present = true;
    }

    if (stack.size() == 1) {
        // KEY-PATH SPEND IS NOT PERMITTED FOR V2.
        return set_error(serror, SCRIPT_ERR_TAPROOT_KEYPATH_DISABLED);
    } else {
        // Script-path: pop control block + script, verify Taproot commitment,
        // then execute the leaf script under SigVersion::TAPSCRIPT.
        ...
        return ExecuteWitnessScript(stack, exec_script, flags,
                                    SigVersion::TAPSCRIPT, checker, execdata, serror);
    }
}
```

Two design decisions to highlight.

(a) Key-path spending is disabled by consensus. A v1 Taproot output can be spent either by a single Schnorr signature against the output key (key-path) or by a script-path proof. For v2 we hard-disable the first option. Why: a key-path spend would authorize with the Taproot output key on the secp256k1/Schnorr rail. If a quantum adversary breaks secp256k1 discrete log, every v2 output that allowed key-path spends would be drainable just like a v1 output. By forcing every v2 spend through the script path, we guarantee the spending condition is whatever the leaf script says — and our default leaf script is a single ML-DSA verify, so the spend is post-quantum end-to-end.

(b) The 32-byte program is still a Taproot-tweaked output. A raw ML-DSA public key is 1.3+ KB — far too long to be the witness program itself. So we keep the v1 Taproot construction (`internal_key` tweaked by Merkle root over leaves), and put the ML-DSA pubkey *inside* a tapscript leaf. The internal key still exists, but it cannot be used to spend; it can be a NUMS point or any wallet-tracked key — semantics are determined purely by the leaf scripts.

The witness stack for the canonical "single-leaf ML-DSA" template is:

```
   <ml_dsa_signature || sighash_flag>     // 2 421 / 3 310 / 4 628 bytes
   <tapscript = <encoded_pk> OP_CHECKMLSIGVERIFY OP_TRUE>
   <control_block = leaf_version || parity || internal_pubkey || merkle_path>
```

V. `OP_CHECKMLSIG` and `OP_CHECKMLSIGVERIFY`

Two new opcodes are added to the tapscript interpreter (`bcore/src/script/script.h:215`):

```
OP_CHECKMLSIG       = 0xbb,
OP_CHECKMLSIGVERIFY = 0xbc,
```

They are explicitly excluded from the OP_SUCCESS upgrade range (`script.cpp:364`) so they are *consensus-meaningful* from genesis — not NOPs that future soft-forks can repurpose.

The execution rule (`interpreter.cpp:1112-1153`):

```
case OP_CHECKMLSIG:
case OP_CHECKMLSIGVERIFY:
{
    // ML-DSA verification only inside Tapscript (witness v1 or v2).
    if (sigversion != SigVersion::TAPSCRIPT)
        return set_error(serror, SCRIPT_ERR_TAPSCRIPT_ONLY);

    // Stack:  <sig_with_sighash_flag>  <pk_blob>  OP_CHECKMLSIG[VERIFY]
    if (stack.size() < 2)
        return set_error(serror, SCRIPT_ERR_INVALID_STACK_OPERATION);

    const valtype& pk_blob       = stacktop(-1);
    const valtype& sig_with_flag = stacktop(-2);

    if (sig_with_flag.empty()) return set_error(serror, SCRIPT_ERR_MLDSA_ENCODING);
    uint8_t hash_type = sig_with_flag.back();
    valtype sig(sig_with_flag.begin(), sig_with_flag.end() - 1);

    // SIGHASH byte must be one of {DEFAULT=0x00, ALL=0x01,
    // NONE=0x02, SINGLE=0x03}, or ANYONECANPAY with ALL/NONE/SINGLE
    // encoded as 0x81..0x83.
    if (!(hash_type <= 0x03 || (hash_type >= 0x81 && hash_type <= 0x83)))
        return set_error(serror, SCRIPT_ERR_MLDSA_ENCODING);

    bool fSuccess = checker.CheckMLDSASignature(sig, pk_blob, hash_type,
                                                sigversion, execdata, serror);
    if (serror && *serror != SCRIPT_ERR_OK) return false;  // fatal: bad encoding

    popstack(stack); popstack(stack);
    if (opcode == OP_CHECKMLSIGVERIFY) {
        if (!fSuccess) return set_error(serror, SCRIPT_ERR_MLDSA_VERIFY);
    } else {
        stack.push_back(fSuccess ? vchTrue : vchFalse);
    }
}
```

`CheckMLDSASignature` (`interpreter.cpp:1938-1956`):

1. parses the encoded `pk_blob` (§3),
2. computes the ML-DSA-flavoured Taproot sighash (§6),
3. calls the constant-time FIPS 204 `mldsa::MLDSA_Verify` on `(pk, sighash, sig)`.

There is no batch verification: each `OP_CHECKMLSIGVERIFY` independently runs ~2–4 ms of lattice arithmetic. That is acceptable because tapscript already ships a generous block-validation weight budget and ML-DSA verify is several orders of magnitude faster than the chain's existing per-block VDF check.

VI. The ML-DSA TapSighash

ML-DSA does **not** reuse the Schnorr/BIP-341 sighash. It uses a separate tagged hash with a distinct domain separator (`bcore/src/script/interpreter.cpp:1592, 1701-1727`):

```
const HashWriter HASHER_TAPSIGHASH_MLDSA{TaggedHash("TapSighash/ML-DSA")};

bool SignatureHashMLDSA(uint256& hash_out,
                        ScriptExecutionData& execdata,
                        const T& tx_to,
                        uint32_t in_pos,
                        uint8_t hash_type,
                        SigVersion sigversion,
                        const PrecomputedTransactionData& cache,
                        MissingDataBehavior mdb)
{
    HashWriter ss{HASHER_TAPSIGHASH_MLDSA};
    ss << uint8_t(0)                          // EPOCH
       << hash_type
       << tx_to.version << tx_to.nLockTime
       << ( prevouts / amounts / scripts / sequences hashes per ANYONECANPAY )
       << ( outputs hash per SIGHASH_ALL/SINGLE )
       << spend_type                           // (ext_flag<<1) | annex_present
       << ( prevout / spent_output / sequence  per ANYONECANPAY )
       << ( in_pos                             otherwise )
       << ( annex hash if present )
       << ( single-output hash if SIGHASH_SINGLE )
       << execdata.m_tapleaf_hash << key_version
       << execdata.m_codeseparator_pos;
    hash_out = ss.GetSHA256();
    return true;
}
```

The shape mirrors BIP-341 exactly so wallet implementers can reuse the same precomputed transaction data — only the **tag string** differs. This is what makes the two domains cryptographically separate: a Schnorr signature over `TapSighash` is provably *not* an ML-DSA signature over `TapSighash/ML-DSA` for the same transaction, even if an attacker crafted a signature that happened to be a valid byte sequence in both schemes.

VII. Address format — `tc1z…`

Witness v2 addresses are 32-byte programs encoded with bech32m, identical in shape to Taproot v1 addresses except for the version character (`bcore/src/key_io.cpp:68-74`):

```
std::string operator()(const WitnessV2Taproot& tap) const
{
    std::vector<unsigned char> data = {2};                    // ← witness v2
    ConvertBits<8, 5, true>([&](unsigned char c) {
        data.push_back(c);
    }, tap.begin(), tap.end());
    return bech32::Encode(bech32::Encoding::BECH32M,
                          m_params.Bech32HRP(), data);
}
```

Decoding (`key_io.cpp:196-201`) parses any bech32m string with version `2` and a 32-byte program directly into a `WitnessV2Taproot` destination. Because the v2 payload size matches v1's `WITNESS_V1_TAPROOT_SIZE`, the parsing path is the simplest possible diff.

The bech32 alphabet maps witness versions to characters; the v2 character is `z`. Hence the visible prefix per network:

Network	HRP	Visible prefix
TensorMain	`tc`	`tc1z…`
TensorTest	`tct`	`tct1z…`
TensorRegtest	`tcrt`	`tcrt1z…`

Total length is 64 characters (HRP + `1` separator + 52 data + 6 checksum), matching v1 Taproot.

VIII. Generating a v2 address

The wallet RPC `generatemldsaaddress` (`bcore/src/wallet/rpc/pq.cpp:47-200`) ties everything together. The flow:

1. **Pick parameter set.** Default ML-DSA-65 (NIST L3, ~192-bit quantum security ≈ AES-192) — chosen for the same reason FIPS 204 designates it as the recommended default: best size/security tradeoff for contemporary signing workloads. Users can opt up to ML-DSA-87 (extra 1.3 KB per signature) or down to ML-DSA-44.
2. **Generate the keypair** with `CMLDSAKey::MakeNewKey(level)` — fresh randomness via liboqs.
3. **Build the encoded pubkey** (§3): `alg_id(0x01) || level || varint(len) || pk_bytes`.
4. **Build the canonical leaf script:** `<encoded_pk> OP_CHECKMLSIGVERIFY OP_TRUE`. The `OP_TRUE` makes the leaf evaluate to `1` after the signature check, satisfying the tapscript "stack must end with truthy" rule.
5. **Compute the tapleaf hash** with BIP-341 leaf-version `0xc0`.
6. **Mint a fresh wallet x-only key** as the Taproot internal key. The internal key is *unspendable in practice* on v2 (key-path is consensus- disabled), but it is still part of the output-key construction and therefore indispensable. Using a wallet-owned key — rather than a well-known NUMS point — lets the wallet recognize ownership of the resulting v2 output during scanning.
7. **Compute the Taproot tweak** with the leaf hash as the (single-leaf) Merkle root and emit the output key + parity.
8. **Encode** as `tc1z…` on TensorMain (§7).
9. **Persist** the ML-DSA key and the Taproot construction metadata so the wallet can later spend the output without re-deriving anything.

The RPC returns enough material for cold-signing flows (`address`, `pubkey`, `seckey`, `tapscript`, `internal_pubkey`, `merkle_root`, `parity`, `output_pubkey`, `encoded_pubkey`).

A second RPC, `signmldsatransaction`, takes a raw transaction plus the key/Taproot bundle and produces a fully-signed v2 input (`pq.cpp:289`). A third, `signmldsatransactionwithwallet`, looks up the Taproot metadata and ML-DSA material from the wallet database (`pq.cpp:490`); today that wallet-backed signing path is complete for unencrypted stored ML-DSA keys. Encrypted-wallet ML-DSA signing is deliberately rejected with an explicit "not yet fully implemented" error, so encrypted-wallet operators must sign via the direct RPC flow with externally supplied key material until that path is completed.

IX. Wallet integration

ML-DSA keys and their Taproot spending metadata live in five new wallet-DB record types (`bcore/src/wallet/walletdb.h:77-81`):

```
extern const std::string MLDSA_KEY;          // mlkey:    pk_hash → (pk, sk_plain, level)
extern const std::string CRYPTED_MLDSA_KEY;  // cmlkey:   pk_hash → (pk, sk_AES256, level)
extern const std::string MLDSA_KEYMETA;      // mlkeymeta:pk_hash → CKeyMetadata
extern const std::string MLDSA_TAPDATA;      // mltapdata:addr → tapscript/control metadata
extern const std::string MLDSA_OUTPUT_INDEX; // mloutidx: outkey → addr
```

with `pk_hash = SHA256d(pk)`. A few intentional non-features:

- **No HD derivation.** ML-DSA does not have BIP-32-style hierarchical deterministic structure (the keys are FIPS-204 lattice samples). Each v2 address is generated independently; the wallet's HD seed cannot re-derive lost ML-DSA keys. This is a *backup discipline* requirement, not a bug — `backupwallet` and PSBT-based cold flows still work, but losing the wallet file means losing the keys.
- **Encryption is the same envelope as ECDSA keys.** When the wallet is encrypted, ML-DSA secret material is wrapped with the same `EncryptSecret` envelope (AES-256-CBC under the wallet master key, using `pk_hash` as the IV material), then stored under `cmlkey`. This mirrors the legacy wallet key-encryption primitive; it is not an AEAD construction with separate additional authenticated data. The handler is a one-line difference from the ECDSA path (`pq.cpp:173-185`).
- `generatemldsaaddress` **requires an unlocked wallet** when encryption is enabled — keys are encrypted *immediately* on disk, never written in plaintext.

The secret key is also returned in the RPC response, deliberately, so operators can capture an off-machine backup of post-quantum material at the moment of generation. (Compare: a Bitcoin wallet can replay an HD chain from the seed phrase. A TensorCash ML-DSA wallet cannot.)

X. Operational guidance: which rail to use?

Because both rails are live from genesis, the choice is operational:

Concern	Witness v0/v1 (ECDSA / Schnorr)	Witness v2 (ML-DSA)
Quantum-safe?	No (Shor-breakable)	Yes (lattice, FIPS 204)
Witness size	~64 B + script	2.4–4.6 KB + script
Verify cost	µs / signature	~ms / signature
HD-derivable?	Yes (BIP-32)	No — independent keys
Hardware wallet support	Mature	Limited / none today
Address visual	`tc1q…` (v0), `tc1p…` (v1) on TensorMain	`tc1z…` (v2) on TensorMain

Concern	Witness v0/v1 (ECDSA / Schnorr)	Witness v2 (ML-DSA)
Tooling maturity	High	Experimental

A reasonable default for 2026 traffic:

- **Hot operational wallets**, exchange deposits, day-to-day tx → v0/v1. They cycle quickly, and quantum cryptanalysis is not an immediate threat to a balance that lives in the address for hours.
- **Long-tenor cold storage**, treasury, time-locked vaults, ten-year inheritance plans → v2 (ML-DSA-65 or ML-DSA-87). This is exactly the "harvest-now, decrypt-later" surface that PQ exists to neutralize.
- **Multisig / quorum policies**: the tapscript leaf is unconstrained, so a wallet can mix `OP_CHECKSIG` (Schnorr) and `OP_CHECKMLSIGVERIFY` branches in the same Taproot tree — e.g. a 2-of-3 vault where two of the three are ML-DSA and one is Schnorr, allowing the operator to spend with a hardware wallet during normal operations and to fall back to PQ keys if Schnorr ever weakens.

XI. Security analysis

(a) What v2 protects. A v2 output's spending condition is ML-DSA over a domain-separated Taproot sighash. An attacker who breaks the underlying lattice problem (SVP / module-LWE) at scale could produce a forgery; this is the threat ML-DSA was designed to resist. An attacker who only breaks secp256k1 signing cannot spend a v2 output — every spend path is gated by an ML-DSA verify.

(b) What v2 does not protect. The *contents* of a v2 transaction (who sent what to whom, amounts, etc.) are still public on chain just like v0/v1. PQ signatures are about authentication, not privacy. Operators needing privacy should layer it (CoinJoin, payment-code rotation) on top of the v2 stack.

(c) Internal key residual risk. Even with key-path disabled, the output key is `internal_key + tagged_hash("TapTweak", internal || root)`. A quantum-equipped attacker who broke secp256k1 discrete log could recover wallet-controlled internal keys from exposed Schnorr/secp256k1 material — but they still cannot satisfy the v2 script path, which requires a fresh ML-DSA signature. The internal key is functionally a deterministic salt, not an authority. Wallets that prefer a literal "no ECDSA anywhere" stance can use a NUMS internal key (BIP-341 `H` point) at the cost of losing wallet-ownership tagging on output scans.

(d) Encoding strictness. `ParsePublicKey` (§3) rejects non-canonical varints, wrong sizes, and unknown algorithm/level bytes. `OP_CHECKMLSIG` rejects empty signatures and out-of-range SIGHASH bytes. There is no "tolerant" path — the consensus rule is strict equality.

(e) Domain separation. `TapSighash` (Schnorr) and `TapSighash/ML-DSA` are independent BIP-340 tagged hashes, so a signature in one domain cannot collide with a signature in the other. Cross-protocol forgeries (taking a Schnorr signature and reusing its bytes inside an ML-DSA witness) are computationally infeasible.

(f) Post-FIPS-204-amendment robustness. Should NIST amend ML-DSA in a future revision (e.g. tweaking the rejection-sampling bound), the `alg_id` byte gives us a clean version bump: a future `ALG_ID_MLDSA = 0x02` encoding is a soft-fork-quality change to the tapscript interpreter and does not require a new witness version.

XII. Summary

TensorCash ships post-quantum spending as a first-class, day-zero feature. Witness v2 is a 32-byte Taproot output that is **always** spent through the script path, where the canonical leaf script is a single `OP_CHECKMLSIGVERIFY` against an FIPS 204 ML-DSA public key. TensorMain addresses are bech32m-encoded with a `tc1z…` prefix. Wallet integration is parallel to the existing ECDSA storage but without HD derivation: each v2 key is generated independently, stored directly or under the legacy encrypted-key envelope, and must be backed up explicitly.

The Core-Node whitepaper covers what makes a TensorCash *block* different from Bitcoin's. This whitepaper covers what makes a TensorCash *spending proof* different. The two are deliberately decoupled: a v2 transaction is mined, propagated, and validated by the same Wesolowski-VDF / sidecar / reorg-advisory machinery as a v0/v1 transaction. Choosing quantum-safe spending costs you signature size and signing throughput; it costs the network nothing.

Appendix A. File map

Concern	Path
ML-DSA crypto wrapper	`bcore/src/crypto/mldsakeygen.{h,cpp}`, `bcore/src/crypto/mldsaverify.{h,cpp}`
Wallet-side `CMLDSAKey`	`bcore/src/mldsakey.{h,cpp}`
New opcode definitions	`bcore/src/script/script.h:215`, `script.cpp:154,364`
Tapscript handler & sighash	`bcore/src/script/interpreter.cpp:1112-1153, 1592, 1701-1727, 1938-1956`
Witness v2 consensus rule	`bcore/src/script/interpreter.cpp:2296-2339`
Address type & variant	`bcore/src/addresstype.h:94-101, 152`
Bech32m encode / decode (`tc1z` on TensorMain)	`bcore/src/key_io.cpp:68-74, 196-201`
Wallet DB record types	`bcore/src/wallet/walletdb.h:77-81`
Wallet RPCs	`bcore/src/wallet/rpc/pq.cpp` (`generatemldsaaddress`, `signmldsatransaction`, `signmldsatransactionwithwallet`)

Appendix B. Defaults and constants

Constant	Value	Source
`ALG_ID_MLDSA`	`0x01`	`crypto/mldsaverify.h:24`
Default parameter set	`MLDSA_65` (= `0x41`, NIST L3)	`crypto/mldsaverify.h:19`, RPC default
Public key sizes (44 / 65 / 87)	1 312 / 1 952 / 2 592 B	`crypto/mldsaverify.h:33-43`
Signature sizes (44 / 65 / 87)	2 420 / 3 309 / 4 627 B	`crypto/mldsaverify.h:33-43`
Secret key sizes (44 / 65 / 87)	2 560 / 4 032 / 4 896 B	`crypto/mldsakeygen.h:21-23`

Constant	Value	Source
`OP_CHECKMLSIG`	`0xbb`	`script/script.h:215`
`OP_CHECKMLSIGVERIFY`	`0xbc`	`script/script.h:215`
Tapscript sighash tag	`"TapSighash/ML-DSA"`	`script/interpreter.cpp:1592`
Witness v2 program size	32 B	`interpreter.cpp:2296`
Bech32 HRPs	`tc` / `tct` / `tcrt`	`kernel/chainparams.cpp`, `key_io.cpp`
Bech32m version char (v2)	`z`	bech32 alphabet