

LayerTree: A Shared Bitcoin UTXO with an $O(1)$ Unilateral Exit

Olaoluwa Osuntokun

Lightning Labs

laolu@lightning.engineering

2026-10-15

DRAFT (commit `ba8422e`)

Abstract. LayerTree lets any number of users share one Bitcoin output, and any one of them leaves alone in one transaction: the unilateral exit is $O(1)$ in transactions, whatever the number of members, and $O(D)$ in bytes for a tree of depth D (about 1,020 virtual bytes at depth 16). The output commits to an off-chain Merkle-sum tree of balances, and tapscript covenants enforce every change to it, with no pre-signed interior and no expiry of the shared output; an expired lease is evicted, and a cohort joins, settles or is swept, in one transaction too. We describe each operation in the order a member meets it, who authorises it and what it costs, and the voucher system: a signature over a source leaf and a new leaf that anyone can redeem once, which an issuer re-signs for a running total, so repeated payments to one recipient, including payments received over Lightning into a funded credit leaf, cost nothing on-chain per payment. The design compiles to four tiers of proposed opcodes, the smallest needing `OP_CAT` and one point-tweak opcode; channels live in leaves; the tree rules are mechanised in Lean 4, and a simulator, a byte-exact Go implementation and 3,019 executable checks accompany it.

Keywords: Bitcoin · covenants · Merkle-sum trees · vouchers · payment channels · formal verification

1 Introduction

LayerTree lets N users share one Bitcoin unspent transaction output (UTXO) and lets any one of them leave alone in a single transaction: the unilateral exit is $O(1)$ in transactions for any N , and $O(D)$ in bytes for a tree of depth D . Payment channels [29] share an output between two parties; most approaches for *groups* pay for the sharing when it is dissolved. In channel factories [5], Ark [16,30], timeout trees [17,43] and covenant trees built on `OP_CHECKTEMPLATEVERIFY` (CTV) [31] or `OP_CHECKCONTRACTVERIFY` [14,13], the shared output is the root of a tree of *transactions*. A member who leaves unilaterally publishes the path from the root to its leaf, $O(\log N)$ transactions each paying a fee; the interior must exist in advance as pre-signed transactions (every member signs, online) or covenant templates; and when the tree expires, reclaiming the funds of one unresponsive member

means publishing everything. Payment-pool opcode proposals (OP_TAPLEAF_UPDATE_VERIFY [37], CoinPool’s OP_MERKLESUB [22], OP_EVICT [42]) target one-transaction withdrawal instead, with purpose-built opcodes.

LayerTree takes a different route. The tree stays off-chain and is a tree of *data*: a binary Merkle-sum tree of balances whose root one output commits to. A member exits by presenting its leaf and sibling path to the output itself; the script recomputes the root from the proof, checks it against the commitment, recomputes it with the leaf removed, and requires the transaction to recreate the output under the new root with the leaf’s amount paid out. The member needs nobody else’s signature, and a deeper tree adds bytes to the exit, not transactions. The same covenant lets the operator evict members whose lease has expired, graft a pre-built subtree of new members in one step, and sweep an expired cohort cheaply.

This paper explains the design for those who would build on it: the tree (§3), every operation as a predicate with the member’s side of it as an algorithm (§4), vouchers and receiving from Lightning (§5), and who can do what (§6); Appendix E gives what an implementer stores and pays for. The operations compile to four tiers of proposed opcodes (§7); we propose none, and three tiers assume opcodes in a tapleaf version their proposals do not define. A hypothetical native opcode would cut the exit to 10 vB per doubling (§10). The elliptic-curve draft [24] is by the author. The ideal functionality, its UC proof and the Lean development are in the full version (§8). The specification, code and proofs are public [25]; this text describes revision ba8422e.

2 Background

Bitcoin. A pay-to-taproot output [39] commits to an internal key P and the Merkle root m of a *taptree* of scripts through $Q = P + H_{\text{TapTweak}}(P \parallel m) \cdot G$, and is spent either by a Schnorr signature [38] under Q or by revealing one script with its Merkle path (the *control block*) and satisfying it. Tapscript [40] has no loops, so every “loop” below is unrolled. BIP-441 [35] defines tapleaf version 0xC2, restoring the string opcodes disabled in 2010 (OP_CAT, OP_SUBSTR, OP_LEFT, OP_RIGHT, bit operations) with unsigned numbers of any length, metered by BIP-440 [34] (“varops”); BIP-347 [12] proposes OP_CAT alone in the current version, 0xC0. OP_TX [33] pushes fields of the spending transaction, optionally collated into one hash preimage; OP_CHECKSIGFROMSTACK (CSFS) [4] verifies a signature over a stack element; OP_TWEAKADD [32] computes $x(P + tG)$, and the EC-ops draft [24] adds point arithmetic. All are proposals. Sizes are in virtual bytes (vB), a quarter of a transaction’s weight in weight units (WU), where a witness byte weighs one unit and any other byte four. A Merkle-sum tree [20,23] hashes each node’s children with their sums, so an inclusion proof proves a leaf’s contribution to the root sum.

3 How LayerTree Works

A *tree* has one *operator* and any number of *members*. Each member owns a *leaf*: a balance and three keys (an owner key that authorises the member’s actions, an exit key that an exit pays, and an evictor key, normally the operator’s, that may remove the leaf once its lease height has passed). The operator holds no key over the funds, only an *operator key* that admits new members and co-signs cooperative changes. The balances live in a Merkle-sum tree kept off-chain; one taproot output holds their total and commits to the root, and each operation is a tapscript leaf of that output, which we call its *predicate*.

3.1 The Tree

We write $\text{le}_k(x)$ for the k -byte little-endian encoding of x . A leaf is a fixed 116-byte record

$$\begin{aligned} &\text{le}_{64}(\text{amount}) \parallel \text{exit_key} \parallel \text{owner_key} \parallel \text{evictor_key} \\ &\quad \parallel \text{le}_{32}(\text{evict_after}) \parallel \text{le}_{32}(\text{nonce}) \parallel \text{le}_{32}(\text{seq}), \end{aligned}$$

with keys as 32-byte x-only points (P2TR(k) is the output script $\text{OP_1 } \langle k \rangle$, so exit_key is the payout’s output key itself, and a single-key wallet supplies its tweaked BIP-86 output key), evict_after a block height, nonce a per-leaf identifier and seq a counter advanced by vouchers (§5). A *node* is a pair (h, s) of a 32-byte hash and an unsigned sum. A position holding leaf ℓ has node $(H_{\text{leaf}}(\ell), \ell.\text{amount})$; an empty position has node $E_0 = (0^{32}, 0)$. The parent of (h_L, s_L) and (h_R, s_R) is

$$(H_{\text{node}}(h_L \parallel \text{enc}_{64}(s_L) \parallel h_R \parallel \text{enc}_{64}(s_R)), s_L + s_R), \quad \text{enc}_{64}(s) = \text{le}_{64}(s) \parallel 0x01.$$

Every hash is a BIP-340 tagged hash with its own tag; $\text{enc}_{64}(s)$ is the minimal BIP-441 encoding of $s + 2^{64}$, produced from a computed sum with one addition. E_h is the node of the empty tree of height h .

The *sibling path* π of a position is the list of D pairs $(\text{dir}_k, (h_k, s_k))$, one per level, giving the side the position lies on and the sibling node not taken. $\text{fold}(n, \pi)$ combines a node with a path from the bottom up, hashing with each sibling in the order the direction gives and adding the sums. The honest path folds a leaf’s node to the root, and folding a *different* node up the *same* path yields the root with that position replaced (§8).

Profiles. In the *dense* profile, which the paper measures, the operator assigns positions and each level costs 88 bytes of script and 41 of witness. A *sparse* profile, in which a leaf’s position is a hash of its owner key and a nonce, needs no slot allocation but cannot graft and costs more on-chain (1,781 vB per exit at 2^{16} members against 1,020); see Appendix D.7.

3.2 The Tree Output

Let r be the root hash and S the root sum. The tree output is a taproot output with internal key $P = \text{lift_x}(x(\text{NUMS} + H_{\text{state}}(r) \cdot G))$, where NUMS is the BIP-341 nothing-up-my-sleeve point and lift_x the BIP-340 even- y normalisation, so the key path is unspendable and every spend goes through a script. Its taptree has one leaf per predicate; its Merkle root m identifies the rule set and is fixed for the tree's life. TreeSPK hides the scripts, so a member, a voucher holder or a tower recomputes m from the published parameters before treating an output as a tree, rejecting any taptree with a leaf outside the expected set (spec 9.1, Appendix E). A *tree output* for state (r, S) has scriptPubKey $\text{TreeSPK}(r, m)$ and amount S . Every predicate recomputes $\text{TreeSPK}(r, m)$ from its witness and checks it against the spent input, binding the root the proofs are checked against and the script set the successor carries. FULL, MID and MIN predicates read input 0, so two such trees cannot be spent in one transaction; a CCV tree's input may sit at any index.

4 Operations

Every operation is one spend of the tree output through one tapleaf, its *predicate* (Figure 1): the tree output is input 0, output 0 recreates it under the new root and total, payouts are outputs $1, \dots, K$, and every other input and output is free. The witness proves each changed position with a *step*: the old node folded up its sibling path gives the committed (r, S) , the new node folded up the same path gives (r', S') , and steps chain. Costs are whole transactions at $D = 16$ in FULL (§10). The four operations a member meets most are stated as predicate boxes and a member's side of leaving and joining as algorithms; Appendix B gives who authorises each operation and what it costs, and the full version states every predicate and every party.

Notation for the predicates. A leaf ℓ has fields $\ell.\text{amt}$, $\ell.\text{xk}$ (exit key), $\ell.\text{ok}$ (owner key), $\ell.\text{ek}$ (evictor key), $\ell.\text{ea}$ (evict_after), $\ell.\text{seq}$ and a nonce; $\hat{\ell}$ is ℓ without its amount, $\text{nd}(\ell) = (H_{\text{leaf}}(\ell), \ell.\text{amt})$ and $\text{nd}(\perp) = E_0$. The spending transaction has inputs in_i , outputs $\text{out}_i = (v_i, \text{spk}_i)$ and lock time lock ; $\text{P2TR}(k)$ is the output script paying k . $\text{Sig}_k(\sigma)$ states that σ is a signature by k over the transaction (the hash type is the signer's obligation, spec 10.7), and $\text{Csfs}_k(\sigma, M)$ that it is a signature by k over the message M , checked with $\text{OP_CHECKSIGFROMSTACK}$. Every box starts by checking that input 0 is the tree output for (r, S) under m and ends by checking that output 0 is the tree output for the root the transition produces, unless that root is E_D and the last leaf has left (spec 7.1, 7.8). *Constants* come from the tree's option set (spec 6.2) and are never witness elements.

4.1 Joining

JOIN_h grafts a subtree of up to 2^h new leaves, built off-chain, into an empty aligned slot on the operator's signature alone. The covenant sees a hash and a

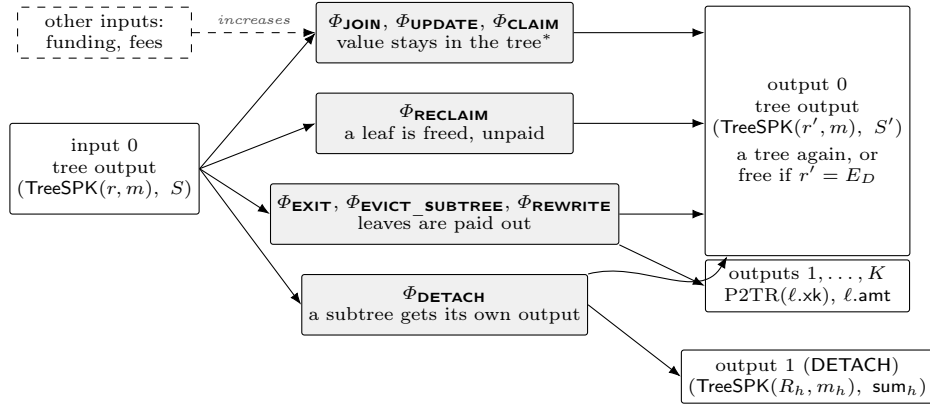


Fig. 1. How the predicates compose. Every transition spends the tree output, input 0, and recreates it as output 0 under the new root; payouts are outputs 1, ..., K and every other input and output is free. *UPDATE's decrease is free. DETACH writes a second tree output, itself the tree output of a smaller tree under the same rules and `tree_id`.

Φ_{JOIN} – grafts K nodes of height h into empty slots, on the operator's signature Witness for $i \in [K]$: a path π_i of $D - h$ levels and a new node $n_i = (\bar{h}_i, \bar{s}_i)$ (a new leaf ℓ_i and a path of D levels in the sparse profile, where $h = 0$ and the position is $H_{\text{pos}}(\ell_i.\text{ok} \parallel \ell_i.\text{nonce})$); a signature σ; and r, m. Constants D; h ($K = 1$ for JOIN_h; $h = 0$ and $K \geq 1$ for JOIN_K); OPERATOR_KEY. Verification write $(r_0, S_0) = (r, S)$. 1. $\text{in}_0 = (\text{TreeSPK}(r, m), S)$ – the spent output is the tree output [7.1 r1, r3] 2. for $i \in [K]$: $\text{fold}(E_h, \pi_i) = (r_{i-1}, S_{i-1})$ and $(r_i, S_i) = \text{fold}(n_i, \pi_i)$ – the slot was empty and now holds n_i [7.3 r1] 3. $\text{Sig}_{\text{OPERATOR_KEY}}(\sigma)$ – only the operator grafts [7.3 r2] 4. $\text{out}_0 = (S + \sum_i \bar{s}_i, \text{TreeSPK}(r_K, m))$ – the grafted sums come from other inputs [7.3 r3] There is no check on the contents of n_i (spec 7.3 rule 4): a joiner verifies its own path before funding (Algorithm 1).

sum and walks only the $D - h$ levels above the slot, so a larger cohort is a smaller transaction: 937 vB for one leaf, 872 vB for four, an estimated 679 vB for 256. Since it cannot look inside the graft, each joiner verifies it (Algorithm 1).

Lines 2 to 6 are checks (a) to (d) of spec 9.2, which Φ_{JOIN} cannot make since it sees only a hash and a sum: the output is unspent (2), the slot was empty (3), U 's leaf folds to the grafted node along a subtree path of exactly h levels (4), and output 0 commits to the root containing it (5, 6). Lines 7 and 8 add the agreed fields and the rule against re-joining a lineage with an owner key, nonce and `seq` held before, which would revive every exit authorisation published for that leaf. A member that funds an input is protected by construction (line 9); one that pays off-chain has nothing to sign, so lines 10 and 11 must hold on the *confirmed* graft before it pays.

Algorithm 1 AcceptLeaf – joiner U , before funding or paying for a leaf

```

1 : input: unsigned tx,  $\ell_U$ , subtree path  $\rho$ , path  $\pi$  of the slot,  $r$ ,  $m$ 
2 : if  $\text{in}_0$  is not an unspent output with  $\text{spk} = \text{TreeSPK}(r, m)$  and amount  $S$ : reject
3 : if  $\text{fold}(E_h, \pi) \neq (r, S)$ : reject
4 : if  $|\rho| \neq h$  or  $\text{fold}(\text{nd}(\ell_U), \rho) \neq (\bar{h}, \bar{s})$  (all sums recomputed): reject
5 :  $r' \leftarrow \text{fold}((\bar{h}, \bar{s}), \pi)$ 
6 : if  $\text{out}_0 \neq (S + \bar{s}, \text{TreeSPK}(r', m))$ : reject
7 : if  $\ell_U$ 's fields or  $U$ 's inputs and change differ from the agreed, or
8 :    $(\ell_U.\text{ok}, \ell_U.\text{nonce}, \ell_U.\text{seq})$  was held in this lineage before: reject
9 : if  $U$  has an input to sign: sign it (SIGHASH_DEFAULT); store  $\ell_U, \rho, \pi, r, r', \text{txid}$ 
10 : else (pays off-chain) : wait for the confirmed graft; check  $\text{out}_0$  is unspent
11 :   and  $\text{fold}(\text{nd}(\ell_U), \text{its path in } r') = (r', S')$ ; only then pay

```

4.2 Moving Value

Most payments never touch the tree: channel leaves pay over Lightning (§9) and vouchers (§5) move value with a signature. The tree changes by UPDATE (the owner and the operator, 1,030 vB) or when a carve is redeemed. A carve writes a new leaf out of a source leaf's balance in two steps and conserves the total: CLAIM on a voucher, SPLIT on the owner's transaction signature (1,586 vB). Checks 7 and 8 bind every carve: the new leaf keeps the source's evictor and a lease no later, so a balance cannot escape eviction, and neither it nor a residual may fall below DUST_MIN, which default policy could not relay a payout for.

4.3 Settling a Cohort

REWRITE $_h$ replaces the contents of one aligned subtree in a single spend: a position may go from a leaf to empty (an exit, paid out), from empty to a leaf (a join) or from one leaf to another. The operator and the owner of every changing leaf authorise it, but a leaf that only gains amount needs no signature, so a payment within a cohort needs only the payer's. At $h = 6$ a mass exit costs 165 vB per leaf and rewriting every balance 151 vB, against 1,030 vB for an UPDATE each. The reclaim form (check 8) serves the receive of §5.3.

4.4 Leaving

EXIT $_K$ removes one leaf, or K in one transaction, and pays each out on the owner's signature alone: no operator, no other member, no timelock. It costs 1,020 vB, and 665 vB more for a second leaf. The signature commits to the tree outpoint, which every operation changes, so a plain exit cannot be pre-signed and the owner must be online to rebuild it after a competing spend. The tower variant EXIT $_K^t$ signs the leaf without its amount, checked from the stack, so a tower builds and rebuilds the exit with its own fee input; the owner voids it by changing the leaf, and draws a fresh nonce at every join (spec 9.2).

Φ_{CLAIM} – carves a new leaf out of a source leaf, by voucher or by the owner’s signature

Witness paths π_c (claim) and π_s (source) of D levels; leaves ℓ_s, ℓ_c ; an authorisation ν (the voucher signature, or under **split** the owner’s transaction signature); a preimage ρ under h ; ξ under x ; r, m . The merge form has instead one path π_p of $D - 1$ levels, the pair (ℓ_s, ℓ_t) with the side b of ℓ_s , a level X , ν (and ρ under hm).

Constants D ; **tree_id**; **DUST_MIN**; **SEQ_FINAL**; **RECLAIM_KEY** if named. The forms h (hash lock), x (exit-key consent), **split** and **merge** (the merge form below) are chosen at creation and are visible in the taptree root m .

Verification, carve forms

1. $\text{in}_0 = (\text{TreeSPK}(r, m), S)$ – the spent output is the tree output [7.1 r1, r3]
2. authorisation, by form: $\text{Csfs}_{\ell_s.\text{ok}}(\nu, H_{\text{voucher}}^m(\text{tree_id} \parallel \hat{\ell}_s \parallel \ell_c))$; under h , $|\rho| = 32$ and the message is $H_{\text{voucher}}^h(\text{tree_id} \parallel \hat{\ell}_s \parallel \ell_c \parallel \text{SHA256}(\rho))$; under **split**, $\text{Sig}_{\ell_s.\text{ok}}(\nu)$ – the voucher binds the tree, the source body and the claim leaf; a wrong preimage fails as a bad signature [7.5 r1; 7.5.1; 7.5.2]
3. under x : $\text{Sig}_{\ell_s.\text{xk}}(\xi)$ – the exit key must also consent [7.5.3 r7]
4. $\ell'_s = \ell_s$ with $\text{amt} = \ell_s.\text{amt} - \ell_c.\text{amt} \geq 0$ and $\text{seq} = \ell_s.\text{seq} + 1$, where $\ell_s.\text{seq} \neq \text{SEQ_FINAL}$; $\text{fold}(\text{nd}(\ell_s), \pi_s) = (r, S)$ and $(r_1, S_1) = \text{fold}(\text{nd}(\ell'_s), \pi_s)$, with E_0 in place of $\text{nd}(\ell'_s)$ if the amount is 0 – step 1: the source pays; its seq advances, which retires the voucher [7.5 r2]
5. $\text{fold}(E_0, \pi_c) = (r_1, S_1)$ and $(r', S') = \text{fold}(\text{nd}(\ell_c), \pi_c)$ – step 2: the claim leaf fills a position empty after step 1 [7.5 r3]
6. $S' = S$ and $\text{out}_0 = (S, \text{TreeSPK}(r', m))$ – a carve conserves the total; no payout [7.5 r4]
7. $\ell_c.\text{ek} = \ell_s.\text{ek}$ and $\ell_c.\text{ea} \leq \ell_s.\text{ea}$ – nothing carved is harder to evict than its source [7.5 r5]
8. $\ell_c.\text{amt} \geq \text{DUST_MIN}$ and $\ell'_s.\text{amt} \in \{0\} \cup [\text{DUST_MIN}, \infty)$ – no sub-dust leaves [7.5 r6]

Verification, merge forms CLAIM^m, CLAIM^{hm} check 1 of the carve form, and

9. $\text{Csfs}_{\ell_s.\text{ok}}(\nu, H_{\text{voucher}}^m(\text{tree_id} \parallel \hat{\ell}_s \parallel \hat{\ell}_t \parallel \text{le}_{64}(X)))$, with $\text{SHA256}(\rho)$ appended and $|\rho| = 32$ under hm – the voucher names an absolute level X for the source’s sibling [7.5.4 r1]
10. $\ell_t.\text{ok} = \ell_t.\text{xk} = \ell_s.\text{xk}$; neither seq is **SEQ_FINAL**; $\ell_t.\text{ek} \neq \text{RECLAIM_KEY}$ when the tree names one – value moves only into a leaf the source’s exit key owns and exits to [7.5.4 r2]
11. $d = X - \ell_t.\text{amt} \geq 1$ and $\ell_s.\text{amt} - d \in \{0\} \cup [\text{DUST_MIN}, \infty)$ – replay moves nothing; the residual stays above the floor [7.5.4 r3]
12. the pair of ℓ_s, ℓ_t folds to (r, S) along π_p ; the pair with $\ell_t.\text{amt} = X$ and $\ell_s.\text{amt} - d$, every other field of both (the source’s seq included) unchanged, folds to (r', S) ; $\text{out}_0 = (S, \text{TreeSPK}(r', m))$ – a merge changes no body, so it voids no voucher against its source [7.5.4 r4, r5]

Algorithm 2 Exit – member U with wallet (ℓ, π, r, o) , or a tower holding τ

- 1: **loop** (rebuild) :
 - 2: $(r', S') \leftarrow \text{fold}(E_0, \pi)$
 - 3: **tx**: $\text{in}_0 = o$, a fee input; $\text{out}_0 = (S - \ell.\text{amt}, \text{TreeSPK}(r', m))$
 - 4: $\text{out}_1 \leftarrow (v, \text{P2TR}(\ell.\text{xk}))$, $v = \ell.\text{amt}$ (less, for a plain exit that pays the fee)
 - 5: $\sigma \leftarrow \text{Sign}_{\ell.\text{ok}}(\text{tx})$, or τ under EXIT_K^t ; **broadcast**
 - 6: **wait** until **tx** is buried or a spend tx' of o confirms:
 - 7: **if** **tx** is pending and too cheap: bump by replacement or an anchor child
 - 8: **if** tx' confirms: $(\ell, \pi, r, o) \leftarrow \text{Follow}(\text{tx}')$
 - 9: **if** ℓ is gone: **stop**; **else continue loop**
-

Lines 2 to 4 build Φ_{EXIT} for $K = 1$, and line 5 signs all outputs, so that a relayar cannot divert the shortfall or the fee slack. The signature commits to the tree outpoint, which every transition changes, so a plain exit needs the owner online to rebuild it; under EXIT^t the owner signs τ once and a tower runs the algorithm with its own fee input (spec 7.2 rule 7, 9.3). Lines 6 to 8 reflect that a conflicting spend that confirms first forces a rebuild against the new root (spec 10.5), within $\Delta(q)$ under $[C]$; **Follow** updates the path from that spend’s

Φ_{REWRITE} – replaces the contents of one aligned subtree of height h , with every changing owner and the operator

Witness a path π of $D - h$ levels; for each position $j \in [2^h]$ an old element o_j (a leaf or $\perp$) and a new element n_j (a leaf, $\perp$, or the marker $\circ$ for “unchanged”, read as o_j); a signature σ_j (and ξ_j under x) for each position that check 5 names; σ_p ; r , m .

Constants D ; h ; OPERATOR_KEY; SEQ_FINAL; RECLAIM_KEY if named. The flags x (exit-key consent), REWRITE ^{x} , which a tree with UPDATE ^{x} or CLAIM ^{x} must use) and reclaim (the tree names RECLAIM_KEY and $h \geq 1$).

Verification let inc_j hold if $n_j = o_j$ in every field but amt , with $n_j.\text{amt} \geq o_j.\text{amt}$ and $o_j.\text{seq} \neq \text{SEQ_FINAL}$.

1. $\text{in}_0 = (\text{TreeSPK}(r, m), S)$ – the spent output is the tree output [7.1 r1, r3]
2. $\text{fold}(R_{\text{old}}, \pi) = (r, S)$, with R_{old} the root of the old elements – the published old subtree is in the tree [7.9 r1]
3. $(r', S') = \text{fold}(R_{\text{new}}, \pi)$, with R_{new} the root of the new elements – the same path gives the new root [7.9 r2]
4. $\text{Sig}_{\text{OPERATOR_KEY}}(\sigma_p)$ – the operator signs once [7.9 r4]
5. for each j with $n_j \neq o_j$ and o_j a leaf, and not inc_j : $\text{Sig}_{o_j.\text{ok}}(\sigma_j)$, and under x also $\text{Sig}_{o_j.\text{xk}}(\xi_j)$ – every changing owner consents; a pure top-up needs nobody, unless the leaf is SEQ_FINAL [7.9 r3, r7]
6. for each j with o_j a leaf and $n_j = \perp$, in position order, the next unused output from output 1 is $(o_j.\text{amt}, \text{P2TR}(o_j.\text{xk}))$ – an emptied leaf is paid exactly its amount [7.9 r5]
7. $\text{out}_0 = (S - \sum_j o_j.\text{amt} + \sum_j n_j.\text{amt}, \text{TreeSPK}(r', m))$ unless $r' = E_D$ – a decrease is free to spend; an increase is funded elsewhere [7.9 r6; 7.8]
8. reclaim form: for o_j with $o_j.\text{ek} = \text{RECLAIM_KEY}$, checks 5 and 6 are replaced as follows.
Transfer: $j = 2i$ even, $n_j = o_j$ but for a lower amt , and o_{j+1} is a leaf with $o_{j+1}.\text{ek} \neq \text{RECLAIM_KEY}$ and $o_{j+1}.\text{ok} = o_{j+1}.\text{xk} = o_j.\text{xk}$, with $n_{j+1} = o_{j+1}$ but for an amt raised by exactly the decrease, and $o_{j+1}.\text{seq} \neq \text{SEQ_FINAL}$ (else its owner signs, as in check 5); no signature and no payout. *Reclaim*: any other change but an amount-only increase, with $\text{lock} \geq o_j.\text{ea}$; no signature and no payout, and the position may be emptied or replaced – value moves from a pending leaf into the recipient’s landing leaf, or the leaf is taken back after its deadline [7.9 r8]

Φ_{EXIT} – spends the tree output (r, S) under m , removes K leaves and pays them out

Witness for $i \in [K]$: a path π_i of D levels, a leaf ℓ_i , a signature σ_i and a flag $e_i \in \{0, 1\}$ (1 for an eviction); and r , m .

Constants D ; the variant flags = (exact payout) and t (tower, implies =), with tree_id under t .

Verification write $(r_0, S_0) = (r, S)$.

1. $\text{in}_0 = (\text{TreeSPK}(r, m), S)$ – the spent output is the tree output [7.1 r1, r3; 6.4]
2. for $i \in [K]$: $\text{fold}(\text{nd}(\ell_i), \pi_i) = (r_{i-1}, S_{i-1})$ and $(r_i, S_i) = \text{fold}(E_0, \pi_i)$ – each leaf is in the previous step’s root and becomes empty [7.2 r1; 7.6]
3. $\text{out}_i = (v_i, \text{P2TR}(\ell_i.\text{xk}))$ – payout i goes to the exit key [7.2 r2]
4. if $e_i = 0$: $\text{Sig}_{\ell_i.\text{ok}}(\sigma_i)$, and under t instead $\text{Csfs}_{\ell_i.\text{ok}}(\sigma_i, H_{\text{exit}}(\text{tree_id} \parallel \ell_i))$ – the owner authorises; the tower form names no transaction [7.2 r3, r7]
5. if $e_i = 0$: $v_i \leq \ell_i.\text{amt}$, and $v_i = \ell_i.\text{amt}$ under $=$ and t – a plain exit’s shortfall is the fee the owner signed for [7.2 r3, r6]
6. if $e_i = 1$: $\text{Sig}_{\ell_i.\text{ek}}(\sigma_i)$, $\text{lock} \geq \ell_i.\text{ea}$ with input 0 enabling lock time, and $v_i = \ell_i.\text{amt}$ – eviction only after the lease, paid in full [7.2 r4]
7. $\text{out}_0 = (S_K, \text{TreeSPK}(r_K, m))$ unless $r_K = E_D$ – output 0 recreates the tree output [7.1 r2, r5; 7.8]

witness alone (spec 9.7). The spec fixes no fee-bumping policy and advises asking the operator for a cooperative exit first.

Eviction and scaling out. The exit predicate with its evict flag removes a leaf whose lease has passed, on the evictor’s signature, and pays the leaf’s whole amount to its exit key; EVICT_SUBTREE_h sweeps an aligned subtree in one spend

at 158 vB per leaf for 51 leaves at $h = 6$, against 1,020 vB each alone. DETACH_h moves a cohort into a tree output of its own (780 vB at $h = 8$, then 762 vB per exit), which raises throughput, since each output is spent independently (§6).

5 Vouchers

A voucher lets a member, or the operator receiving from Lightning, pay with a signature alone; its holder can redeem it later without the issuer or the operator. Vouchers take the place of Ark’s out-of-round payments with a different trade: an Ark transfer is undone only if sender and server collude, a voucher by its issuer alone, or by the source’s evictor after the lease, until its redemption confirms. In exchange a redeemed voucher is a leaf its holder exits alone, whose eviction pays it on-chain where an expired Ark round forfeits to the operator.

5.1 The Voucher and CLAIM

A voucher is a BIP-340 signature by the owner key of a *source* leaf over $H_{\text{voucher}}(\text{tree_id} \parallel \hat{\ell}_{\text{src}} \parallel \ell_{\text{claim}})$, where $\hat{\ell}_{\text{src}}$ is the source leaf without its amount (spec 0.13.0), ℓ_{claim} is the full leaf being paid (an amount and the recipient’s keys), and tree_id is a 32-byte constant fixed when the tree is created (Appendix E). Anyone who holds the voucher can redeem it with CLAIM (Φ_{CLAIM} , check 2), which checks the signature with OP_CHECKSIGFROMSTACK and needs no other. Replay is impossible because redemption advances the source’s *seq*, which the voucher binds; the amount is not bound, so an amount-only top-up voids nothing, and the sum check stops a voucher drawing more than the source holds. Two rules that no script enforces carry the rest: a leaf’s *seq* only ever advances (an UPDATE winding it back would revive old vouchers), and no two leaves share an owner key and nonce, since a voucher binds its source by content, not position (spec 10.3). Binding tree_id rather than the taptree root m keeps a voucher out of other trees built with the same parameters, and a detached cohort inherits it, so vouchers survive a DETACH_h .

5.2 Cumulative Vouchers and Their Risks

Every voucher against a source draws on the same leaf and the first redemption voids the rest, so an issuer that pays one recipient repeatedly signs a *cumulative* voucher for the running total and re-signs it for more at each payment; the recipient keeps the latest and redeems it alone if the issuer stops cooperating, with no contest window (Appendix C.1). The covenant prevents over-*redemption* but not over-issuance or voiding: the issuer can advance *seq* first, and once the source’s lease passes its evictor can pay the whole balance to the source’s exit key. A holder therefore treats a voucher whose lease can pass before a redemption would confirm (within $\Delta(q)$, §6) as unsecured, and prefers a full-balance voucher, since a partial one needs an empty position that the operator controls. A dedicated *credit leaf*, owned by the issuer with the recipient’s key as exit key, in a tree with

CLAIM^x , UPDATE^x and the exact exit, removes the voiding: every change that could void a voucher needs the exit key (Appendix C.2). A hash-locked voucher CLAIM^h paired with an HTLC on the same hash is an *ordered* exchange, not an atomic swap, under five rules (spec 9.6.2, Appendix C.3); it is how a newcomer buys a leaf over Lightning.

5.3 Receiving over Lightning

The flow. A user receives a Lightning payment into a credit leaf that the operator owns and that names the user's key as exit key (any tier but MIN). The operator draws the preimage and settles the payer's HTLC *first*, only then signing the user a cumulative voucher for the new running total. Nothing reaches the chain per payment and the user need not be online, but a voucher draws only what the credit leaf holds, which the operator pre-funds or tops up at about 122 vB a leaf in a cohort rewrite (Appendix E.3). A hash-locked voucher issued before the HTLC settled would break the first swap rule and, if the HTLC failed, could never be retired, since advancing the credit leaf's *seq* needs the recipient's exit key. Settling first leaves the recipient exposed for one payment, the one the operator has settled and not yet vouched for. A plain voucher cannot close that gap, being a cheque and not escrow, but a *pending-leaf receive* can (below, and Appendix E.3): the voucher is hash-locked and raises a leaf the recipient owns, and the operator can take back what the recipient never claimed.

Settlement. Normally the voucher is never redeemed: the operator pays the recipient a leaf in its next cohort graft (under 3 vB a member) and retires the voucher by one UPDATE^x that the recipient co-signs (1,048 vB) or a cohort REWRITE_h^x (about 164 vB a leaf). If the operator stops cooperating, the recipient redeems its latest voucher with one CLAIM^x (1,636 vB, no timelock) and leaves by the exact exit, paying both fees from a confirmed coin: 2,656 vB, or 26,560 sat at 10 sat/vB, below which the fallback costs more than it recovers (Appendix E.4).

The pending-leaf receive. A tree that names a reclaim key and carries the merge voucher CLAIM^{hm} (the merge forms of Φ_{CLAIM}) and the reclaim form of Φ_{REWRITE} (FULL, MID and CCV) closes the gap. Each recipient holds a pending leaf P and, beside it, a landing leaf L of its own. For each payment the operator signs a hash-locked voucher that raises L and lowers P , which only the recipient's preimage releases (Algorithm 3); once per window the operator moves the settled total into L and then takes back what was never claimed (Algorithm 4). The voucher can only move value from P into a leaf the recipient alone can exit, so it needs no signature beyond the issuer's and a tower can redeem it.

The recipient holds a *receive pair*: a pending leaf P (owned by the operator, exit key the recipient's, evictor RECLAIM_KEY , deadline T) and its sibling, the landing leaf L , which the recipient owns (spec 9.6.3). Line 2 keeps the HTLC alive past a late preimage; lines 3 to 5 leave headroom, since a voucher that drains P exactly is blocked by an amount-only top-up. Line 6 signs a hash-locked merge voucher that raises L to the absolute level X without advancing P 's *seq*, so an

Algorithm 3 Receive – recipient R and operator O , for one payment into a pair (P, L)

```

1 :  $R$ : draw  $\rho$ ;  $\eta \leftarrow \text{SHA256}(\rho)$ ; invoice on  $\eta$ ; the HTLC  $(\eta, a, E)$  reaches  $O$ 
2 :  $O$ : require  $E \geq T + \Delta(q)$  + time to settle from an on-chain preimage
3 :  $O$ :  $X \leftarrow$  settled level of  $L + a$ 
4 :  $O$ : if  $P.\text{amt} < (X' - L.\text{amt}) + \text{DUST\_MIN}$  for  $X' = X$  or an outstanding target:
5 :   top up  $P$  by an amount-only increase
6 :  $O$ : sign a  $\text{CLAIM}^{hm}$  voucher  $V$  on  $(\eta, X)$  against  $P$ ; send  $V$  to  $R$ 
7 :  $R$ : verify in the confirmed tree:  $P$  reclaimable,  $L$  its sibling,  $R$ 's, not reclaimable,
8 :    $P.\text{xk} = L.\text{ok} = L.\text{xk}$ , no  $\text{SEQ\_FINAL}$ ,  $V$  verifies under  $P.\text{ok}$  over  $\text{tree\_id}, \hat{P}, \hat{L}, X$ ,
9 :    $X =$  settled level of  $L + a$ ,  $P.\text{amt} + L.\text{amt} - X' \geq \text{DUST\_MIN}$  for all outstanding  $X'$ ,
10 :    $T$  at least  $\Delta(q) + \varepsilon$  away, and  $L.\text{ea} > T + \Delta(q)$ ; else refuse
11 :  $R$ : release  $\rho$  (one preimage at a time per pair); hand  $(V, \rho)$  to its tower
12 :  $O$ : on learning  $\rho$ : settle the inbound HTLC; nothing reaches the chain

```

older, lower voucher is a no-op. Lines 7 to 10 verify against the *confirmed* tree before line 11 releases one preimage at a time per pair; before it the operator has signed nothing it can lose. Lines 1 to 4 are rule 3 of spec 9.6.3: before $T - \Delta(q) - \varepsilon$

Algorithm 4 SettleWindow – operator, at the close of a window of a pair (P, L)

```

1 : before  $T - \Delta(q) - \varepsilon$ : for each pair that settled amounts in the window:
2 :   raise  $L$  to its settled level, either by redeeming the merge voucher  $U$ 
3 :   that the operator signs itself,
4 :   or by a transfer in  $\Phi_{\text{REWRITE}}$  (reclaim form), batched across recipients
5 : hold every inbound HTLC with an outstanding merge voucher open
6 :   until the reclaim of  $P$  has confirmed
7 : after  $T$ : run Reclaim on every  $P$  of the window

```

the operator raises L to the settled level, by a voucher it signs itself or by the reclaim-form transfer of Φ_{REWRITE} , which needs only its signature and settles a cohort of recipients in one transaction. The recipient does not rely on it: if L is short by then, the recipient or its tower must redeem (the one liveness duty). Lines 5 and 6 are rule 4: until the reclaim of P confirms, an inbound HTLC with an outstanding voucher stays open.

Costs. Issuing a voucher costs nothing on-chain. A redemption proves two positions: 1,618 vB for CLAIM , 1,628 vB for CLAIM^h and 1,636 vB for CLAIM^x , against 1,586 vB for SPLIT , with no payout output (add 43 vB for one).

6 Trust Model

Custody never rests with the operator, given a taptree whose m the member has recomputed (§3.2). The key path is unspendable, and every script path needs a

proof against the committed root plus the affected owner’s signature, the evictor’s after the lease (paying the leaf in full), or the operator’s for operations that only add value or move a cohort intact. Table 4 (Appendix B) lists each party’s powers: the operator can refuse to co-sign, graft, evict after a lease (paying in full) and detach a cohort, but cannot move or reduce another leaf, delay an exit beyond [C] or freeze value it did not graft; an owner can always exit, and anyone can redeem a valid voucher or relay a transition.

What a member does. A member follows every spend of the tree output and updates its own path from the witness alone (spec 9.7), verifies its path before funding a join (Algorithm 1) and the source before accepting a voucher (§5.2), keeps a confirmed fee coin for every unilateral path but FULL’s fee-from-payout exit (Appendix E), and must be online to exit unless a tower holds its EXIT^t authorisation (Algorithm 2).

Liveness. Every operation spends and recreates one output, so an honest exit can be overtaken by conflicting spends (a corrupt member alone can carve its balance back and forth), and its delay depends on policy and fees. We state it as an assumption. *[C] (contention):* at most $\kappa \geq 1$ spends of a tree output confirm per block, and while an honest spend is pending at the prevailing feerate at most β conflicting adversarial spends confirm ahead of it. κ is mempool policy (25 under Bitcoin Core’s chain limits before version 31, at most 64 under its cluster limit since [2,8], as low as 1 once anyone hangs descendants off a payout); β is the adversary’s fee budget against the honest bid. Under [C] an honest spend with q others queued ahead confirms within $\Delta(q) = \Delta_L(\lceil(\beta + q)/\kappa\rceil + 1)$ blocks, Δ_L being the chain’s own delay. We use $\kappa = 25$, the lowest limit relay peers may enforce (spec 10.5): cooperatively 10,000 leaves leaving at once take 400 blocks (157 at 64), so a tree is sized by exit throughput, not depth. The operator can evict any leaf once its lease passes, so a member who wants a longer guarantee needs a later lease at entry; no carve extends one.

7 Realisations by Opcode Tier

Each operation of §4 is compiled to tapscript against four opcode sets (Table 1), so that the design’s dependence on each candidate opcode is measured rather than asserted; the tree format and the predicates are the same in every tier. FULL, MID and CCV compile every operation, MIN the dense single-leaf exit and SPLIT. Table 2 (Appendix A) lists what each tier assumes of the proposals it uses, mirroring the assumptions [A-1] to [A-6] of the specification. MIN is the tier closest to the proposals as written: it needs only OP_CAT (BIP-347) and OP_TWEAKADD (BIP-449), both defined for the current tapleaf version. MID and CCV in 0xC0 need OP_TX in a version its draft does not define; CCV in 0xC2 needs BIP-443 there; FULL needs CSFS and EC ops in 0xC2 and two drafts whose opcode numbers collide.

Table 1. The four opcode tiers. “Shape” says whether inputs and outputs the predicate does not constrain are free or fixed by the script. Sizes are bytes of script / witness data for EXIT_1 in the dense profile at $D = 16$; the last row is the transaction size on the model of §10 and serialised with its witness length prefixes. MID’s sweep fits at most 54 occupied positions at $h = 6$ (stack limit).

	FULL	MID	MIN	CCV
Tapleaf version	0xC2	0xC0	0xC0	0xC2 or 0xC0
Beyond BIP-342	OP_CAT, splice, bit ops, OP_TX, CSFS, EC ops	OP_CAT, OP_TX, CSFS, OP_TWEAKADD	OP_CAT, OP_TWEAKADD	OP_CCV (BIP-443), OP_TX
Numerics	any length	4-byte, $< 2^{31}$	4-byte, $< 2^{31}$	as FULL or MID
Owner payout	$\leq$ amount	exact	exact	exact
Predicates	all	all	EXIT_1 , SPLIT	all
Vouchers / channels / shape	yes / yes / free	yes / yes / free	no / no / fixed	option / yes / free
EXIT_1 script / witness (B)	2,101 / 902	3,367 / 978	3,640 / 1,175	1,716 / 838
EXIT_1 (vB), model / serialised	1,020 / 1,030	1,355 / 1,398	1,473 / 1,518	908 / 918

What each tier gives up. In FULL the leaf is one stack element and sums use arbitrary-length arithmetic. MID’s 4-byte numbers cap a tree at 2^{31} satoshis (21.47 BTC) and force exact payouts. MIN has no OP_TX: the script builds the BIP-341 signature message from the values it has verified and checks it with a signature under the generator G (Poelstra’s construction [27]), which passes only if the message equals the real transaction’s (assumption [MIN]); vouchers and channels, which need a signature over stack data, are out of its reach, but SPLIT is not (6,651 bytes of script). CCV replaces every taproot-key recomputation with an OP_CHECKCONTRACTVERIFY call and is the smallest tier.

8 Formal Verification

The rules of every operation are mechanised in Lean 4 [21] (about 4,500 lines, no Mathlib; 72 headline theorems in an axiom audit). They show that a root binds its tree; that every sequence of steps conserves the total, with no hash assumption; that every known leaf can be paid out by its honest witness; that a graft without a real preimage freezes only the value it brought; that a voucher cannot be replayed and a carve keeps the evictor, lease and dust floor; and that the cohort operations and encodings behave as the scripts assume. An exit is guaranteed only to a member who verified its path (lines 2 to 6 of Algorithm 1). Scripts are matched to the rules by executable checks (§10), not a verified compiler.

For each tier, the protocol UC-realises an ideal functionality of the shared output in a global-ledger hybrid model [6,1], with idealised hashes, taproot binding in the random-oracle model and [C], against a static adversary corrupting the operator and any members. The full version gives the functionality, the simulator, the proof and the Lean statements.

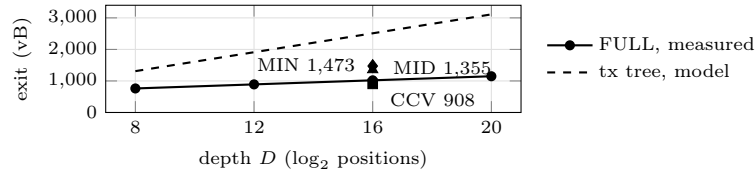


Fig. 2. Exit cost against depth: LayerTree measured, radix-2 transaction tree modelled.

9 Channel Leaves

A leaf whose exit key is a 2-of-2 between its owner and a counterparty (normally the operator) hosts a payment channel whose funding output appears only when an exit or eviction pays the leaf out, so a tree of such leaves is a Lightning hub behind one UTXO. That outpoint changes with every tree transition, so states are signed without committing to it (OP_TX collates the signed fields and CSFS checks the signature, emulating SIGHASH_ANYPREVOUT [10]), with the eltoo ratchet [9,36], zero-fee version-3 transactions [41] and an ephemeral anchor [3]. This needs FULL, MID or CCV.

Deadlines and conditions. A channel leaf’s HTLC deadlines add the counterparty’s stale updates to the exit’s confirmation time: with `csv_delay` = 144 the invoice’s minimum final delta is about 220 blocks, against `lnd`’s default of 80 [18], and about 1,775 channel leaves may close in one window (Appendix E.7). A channel leaf has exact payouts, so the owner cannot underfund the channel; `seq` = `SEQ_FINAL`, so no voucher or `SPLIT` can carve the capacity the signed states commit to; and lease zero with the counterparty as evictor, so eviction is the counterparty’s force-close. The state machine and the budget run end to end in the simulator with two blocks of margin; a proof that channel leaves compose is open. A force-close costs $\approx 1,681$ vB (its exit creates the funding output), against 684 vB for a simple taproot channel’s open plus force close [26].

10 Evaluation

Implementation and model. A Python tapscript interpreter for both leaf versions computes the real BIP-341 signature message and runs every compiled script against a reference tree, with positive and negative cases per rule; a Go implementation on `btcd` and `lnd` generates the same scripts byte for byte and builds, signs and verifies every transition of every tier. A vB figure is a whole transaction on a 946 WU skeleton (Appendix D.10).

Exit cost. A dense exit (Figure 2, Table 3) costs 762 vB at $D = 8$ and 1,149 vB at $D = 20$, about 32 vB per level. One tier step costs as much as 10 (MID, +33%, or +36% serialised) to 14 (MIN, +44%, or +47%) levels of depth.

Comparison with transaction trees. The dashed line of Figure 2 is a *model*: an exiter publishes D transactions of about 150 vB and a 111 vB claim, anchors and their fee children omitted, which favours the tree. LayerTree is about 60% smaller at $D = 16, 165$ against 261 vB a member under mass exit. The model omits the latency of D confirmations and the larger opcode set (Appendix D).

A native opcode (hypothetical). An unproposed `OP_MERKLESUMUPDATE` would fold a Merkle-sum path in one step, replacing the unrolled levels. `EXIT1` is then 252 bytes of script at every depth, and the exit about $381 + 10k$ vB for 2^k members (541 at 2^{16} , against 1,020 for FULL and 2,511 for a transaction tree). Batches gain little, since the per-position step dominates: settling pending receives costs 547 vB a pair at 64 positions. It fixes this tree’s encoding in consensus, so the work treats it as a measurement, not a tier (§12, Appendix D.8).

11 Related Work

Payment pools. CoinPool [22] is a shared UTXO with per-user balances whose one-transaction withdrawal rests on a new opcode, `OP_MERKLESUB`; `OP_TAPLEAF_UPDATE_VERIFY` [37] recreates a pool output with the leaver’s key and leaf removed, and `OP_EVICT` [42] removes unresponsive members. LayerTree reaches one-transaction exit from general-purpose opcodes, holds balances in a Merkle-sum tree rather than keys in a taptree, and adds leases, grafts and channel leaves. *Transaction trees.* Channel factories [5] need all N members to sign every change. Ark [16] batches users into rounds whose outputs are leaves of a pre-signed MuSig2 tree; covenant Ark replaces the signing with CTV templates [30] but keeps the tree, its unrolled exit and its expiry. Timeout trees [17] and SuperScalar [43] add laddered expiries and Decker-Wattenhofer factories; MATT-style trees [13,14] replace signatures by templates. Table 13 (Appendix D) compares them. *Beyond Bitcoin.* Plasma’s forced exits [28] prove a balance against a committed root. Proofs of liabilities [15,7] use Merkle-sum trees, whose per-child sums block the inflation a total-only node admits.

12 Discussion and Conclusion

All activity is serialised through one output, so an honest exit’s delay is set by policy and fees, which [C] names; cluster mempool [8] raises κ for a cooperative tree but shares the cluster with every descendant anyone attaches, and fee pinning is outside [C]. The price of the one-transaction exit is one UTXO for all activity. Next: a verified compiler, a proof that channel leaves compose, a P model [11], and costing a native Merkle-sum opcode (§10).

Use of AI. The concept, the initial design and its first version are the author’s own; AI assistants (Anthropic’s Claude) were used to develop the code, proofs and text from them, and the author reviewed all of it and is responsible for its content.

References

1. Badertscher, C., Maurer, U., Tschudi, D., Zikas, V.: Bitcoin as a transaction ledger: A composable treatment. In: CRYPTO 2017. LNCS, vol. 10401, pp. 324–356. Springer (2017)
2. Bitcoin Core: Mempool policy defaults: Ancestor and descendant count limits (25) and cluster limit (64). `src/policy/policy.h` at v31.1, <https://github.com/bitcoin/bitcoin/blob/v31.1/src/policy/policy.h> (2026)
3. Bitcoin Core: Pay to anchor and ephemeral dust policy (`src/policy/ephemeral_policy.h` at v31.1). https://github.com/bitcoin/bitcoin/blob/v31.1/src/policy/ephemeral_policy.h (2026)
4. Black, B., Rubin, J.: BIP-348: CHECKSIGFROMSTACK. <https://github.com/bitcoin/bips/blob/master/bip-0348.md> (2024)
5. Burchert, C., Decker, C., Wattenhofer, R.: Scalable funding of bitcoin micropayment channel networks. In: Royal Society Open Science. vol. 5 (2018)
6. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: FOCS 2001. pp. 136–145. IEEE (2001)
7. Chalkias, K., Chatzigiannis, P., Ji, Y.: Broken proofs of solvency in blockchain custodial wallets and exchanges. In: Financial Cryptography and Data Security, FC 2022 International Workshops. LNCS, Springer (2022)
8. Daftuar, S.: An overview of the cluster mempool proposal. Delving Bitcoin, <https://delvingbitcoin.org/t/an-overview-of-the-cluster-mempool-proposal/393> (2024)
9. Decker, C., Russell, R., Osuntokun, O.: eltoo: A simple layer2 protocol for bitcoin. <https://blockstream.com/eltoo.pdf> (2018)
10. Decker, C., Towns, A.: BIP-118: SIGHASH_ANYPREVOUT for taproot scripts. <https://github.com/bitcoin/bips/blob/master/bip-0118.mediawiki> (2017)
11. Desai, A., Gupta, V., Jackson, E., Qadeer, S., Rajamani, S., Zufferey, D.: P: Safe asynchronous event-driven programming. In: PLDI 2013. pp. 321–332. ACM (2013)
12. Heilman, E., Sabouri, A.: BIP-347: OP_CAT in tapscript. <https://github.com/bitcoin/bips/blob/master/bip-0347.mediawiki> (2023)
13. Ingala, S.: Merkleize all the things. bitcoin-dev mailing list, <https://merkle.fun> (2022)
14. Ingala, S.: BIP-443: OP_CHECKCONTRACTVERIFY. <https://github.com/bitcoin/bips/blob/master/bip-0443.mediawiki> (2025)
15. Ji, Y., Chalkias, K.: Generalized proof of liabilities. In: ACM CCS 2021. ACM (2021)
16. Keceli, B.: Ark: An alternative privacy-preserving second layer solution. bitcoin-dev mailing list, 22 May 2023, <https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2023-May/021694.html> (2023)
17. Law, J.: Scaling lightning with simple covenants. bitcoin-dev mailing list, <https://github.com/JohnLaw2/ln-scaling-covenants> (2023)
18. Lightning Labs: lnd: the default time-lock delta of 80 blocks. `chainreg/chainregistry.go` at v0.18.4-beta, <https://github.com/lightningnetwork/lnd/blob/v0.18.4-beta/chainreg/chainregistry.go> (2024)
19. Lightning Network specification: BOLT #3: Bitcoin transaction and script formats. <https://github.com/lightning/bolts/blob/master/03-transactions.md> (2024)

20. Merkle, R.C.: A digital signature based on a conventional encryption function. In: CRYPTO 1987. LNCS, vol. 293, pp. 369–378. Springer (1988)
21. de Moura, L., Ullrich, S.: The Lean 4 theorem prover and programming language. In: CADE 2021. LNCS, vol. 12699, pp. 625–635. Springer (2021)
22. Naumenko, G., Riard, A.: CoinPool: Efficient off-chain payment pools for Bitcoin. <https://github.com/coinpool-dev/paper/blob/master/coinpool-v0.1.pdf>; OP_MERKLESUB draft BIP, <https://github.com/ariard/bips/blob/coinpool-bips/bip-merklesub.mediawiki> (2022)
23. Osuntokun, O.: Merkle sum sparse Merkle trees. Draft BIP (Taproot Assets), <https://github.com/Roasbeef/bips/blob/bip-tap/bip-tap-ms-smt.mediawiki> (2021)
24. Osuntokun, O.: Elliptic curve operations for Bitcoin script. Draft BIP, bitcoin/bips PR #1945 (closed unmerged), <https://github.com/bitcoin/bips/pull/1945> (2025)
25. Osuntokun, O.: LayerTree: Specification, reference implementation and formal model. <https://github.com/roasbeef/layertree> (2026)
26. Osuntokun, O.: Simple taproot channels (extension BOLT, feature bits 80/81). lightning/bolts PR #995, merged May 2026, <https://github.com/lightning/bolts/blob/master/bolt-simple-taproot.md> (2026)
27. Poelstra, A.: CAT and Schnorr tricks I and II. Blog posts, 20 Jan and 10 Feb 2021, <https://www.wpsoftware.net/andrew/blog/cat-and-schnorr-tricks-i.html>, <https://www.wpsoftware.net/andrew/blog/cat-and-schnorr-tricks-ii.html> (2021)
28. Poon, J., Buterin, V.: Plasma: Scalable autonomous smart contracts. White paper, <https://plasma.io/plasma.pdf> (2017)
29. Poon, J., Dryja, T.: The bitcoin lightning network: Scalable off-chain instant payments. <https://lightning.network/lightning-network-paper.pdf> (2016)
30. Roose, S.: The Ark case for CTV. Delving Bitcoin, <https://delvingbitcoin.org/t/the-ark-case-for-ctv/1528> (2025)
31. Rubin, J.: BIP-119: CHECKTEMPLATEVERIFY. <https://github.com/bitcoin/bips/blob/master/bip-0119.mediawiki> (2020)
32. Rubin, J.: BIP-449: OP_TWEAKADD, x-only key tweak addition. <https://github.com/bitcoin/bips/blob/master/bip-0449.md> (2026)
33. Russell, R.: OP_TX. Draft BIP on a personal fork, <https://github.com/rustyrussell/bips/blob/guilt/varops/bip-unknown-optx.mediawiki> (2024)
34. Russell, R., Moik, J.: BIP-440: Varops budget for script runtime constraint. <https://github.com/bitcoin/bips/blob/master/bip-0440.mediawiki> (2025)
35. Russell, R., Moik, J.: BIP-441: Restoration of disabled script (tapleaf 0xC2). <https://github.com/bitcoin/bips/blob/master/bip-0441.mediawiki> (2025)
36. Sanders, G.: LN-Symmetry (eltoo) transaction formats, draft. https://github.com/instagibbs/bolts/blob/eltoo_draft/XX-eltoo-transactions.md (2023)
37. Towns, A.: TAPLEAF_UPDATE_VERIFY covenant opcode. bitcoin-dev mailing list, 9 Sep 2021, <https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2021-September/019419.html> (2021)
38. Wuille, P., Nick, J., Ruffing, T.: BIP-340: Schnorr signatures for secp256k1. <https://github.com/bitcoin/bips/blob/master/bip-0340.mediawiki> (2020)
39. Wuille, P., Nick, J., Towns, A.: BIP-341: Taproot: SegWit version 1 spending rules. <https://github.com/bitcoin/bips/blob/master/bip-0341.mediawiki> (2020)
40. Wuille, P., Nick, J., Towns, A.: BIP-342: Validation of taproot scripts. <https://github.com/bitcoin/bips/blob/master/bip-0342.mediawiki> (2020)

Table 2. Deployment assumptions (spec 4.1, `spec/layertree.md`). A proposal *defines* its opcode for one tapleaf version; a tier may *assume* it in another.

	assumption	tiers
[A-1]	OP_TX [33] (a draft on a personal fork, defined for 0xC2 with BIP-440 costs) pushes amounts as 8-byte little-endian strings, which FULL compares with BIP-441 numerics	FULL, MID, CCV
[A-2]	CSFS (BIP-348, defined for 0xC0) available in 0xC2 at the cost of one signature check	FULL, CCV
[A-3]	the EC-ops draft [24] available in 0xC2 at its stated costs	FULL
[A-4]	opcode numbers: OP_TX and OP_EC_POINT_NEGATE both claim 0xbd, BIP-443 and OP_EC_POINT_ADD both 0xbb; the simulator parks OP_TX, OP_TWEAKADD and CCV at 0xbf, 0xc0, 0xc1, so the golden scripts are byte-exact under that numbering only (sizes are unaffected)	all
[A-5]	OP_TX in 0xC0 with CScriptNum outputs, which no proposal defines	MID, CCV
[A-6]	BIP-443 (defined for 0xC0) also in 0xC2; its sigop cost, open upstream, charged as one signature check; amount rules as of bitcoin/bips commit 24e96e87. Conservation rests on the tier's OP_TX equalities, so a change to BIP-443 can only make honest spends fail	CCV
[MIN]	OP_CHECKSIG under key G with $s = e + 1$ accepts only if the script-built message is the real one, barring a collision in the TapSighash or challenge hash	MIN
[C]	contention: κ and β of §6 (policy and fees, not consensus)	all

41. Zhao, G.: BIP-431: Topology restrictions for pinning. <https://github.com/bitcoin/bips/blob/master/bip-0431.mediawiki> (2024)
42. ZmnSCPxj: OP_EVICT: An alternative to OP_TAPLEAFUPDATEVERIFY. bitcoin-dev mailing list, 18 Feb 2022, <https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2022-February/019926.html> (2022)
43. ZmnSCPxj: SuperScalar: Laddered timeout-tree-structured Decker-Wattenhofer factories. Delving Bitcoin, <https://delvingbitcoin.org/t/superscalar-laddered-timeout-tree-structured-decker-wattenhofer-factories/1143> (2024)

A Assumptions and Measured Sizes

Table 2 lists the deployment assumptions of §7, and Table 3 the script, witness and transaction sizes behind Figure 2 and Table 1.

Table 3. Measured EXIT₁ cost (FULL unless noted) on the model of §10. Witness weight is script, witness data and a 129-byte control block for the eight-leaf reference script set (each doubling of the set adds 8 vB); the transaction adds the 946 WU skeleton. Last column: unamortised radix-2 transaction tree of 2^D leaves, $150D + 111$ vB. The sparse profile is in Table 14.

profile	D	script (B)	witness data (B)	witness (WU)	tx (vB)	radix-2 tx tree (vB)
dense	8	1,397	574	2,100	762	1,311
dense	12	1,749	738	2,616	891	1,911
dense	16	2,101	902	3,132	1,020	2,511
dense	20	2,453	1,066	3,648	1,149	3,111
MID dense	16	3,367	978	4,474	1,355	
MIN dense	16	3,640	1,175	4,944	1,473	
CCV dense	16	1,716	838	2,683	908	

B Operations in Detail

The operations of §4 in the order a member meets them. Every operation is one spend of the tree output through one tapleaf; costs are whole transactions at $D = 16$ in FULL.

B.1 Joining

JOIN_h: grafting a cohort. The operator inserts a subtree of up to 2^h new leaves, built off-chain, into an empty aligned slot of height h ; $h = 0$ adds one leaf. *Authorised by* the operator key alone; no existing member signs or is affected. *Proves* that the slot held the empty node E_h , and adds the new subtree’s root and sum. *On chain* the tree output grows by the subtree’s sum, which other inputs must supply. The covenant sees a hash and a sum, not the leaves, and walks only the $D - h$ levels above the slot, so a larger cohort is a smaller transaction: 937 vB for one leaf, 872 vB for four (218 each), and an estimated 679 vB for 256 (2.7 each). A taptree carries one JOIN_h leaf per height it supports.

Because the covenant cannot look inside the graft, each joiner checks it. A member funding its own leaf verifies, before signing its input over all outputs, that the slot folds as empty to the current root, that its leaf folds to the grafted node with every sum recomputed, and that output 0 commits to the new root; its input can then only be spent into a root containing its leaf. A self-funded cohort is thus a signing round, like an Ark round or a coinjoin, and each member adds its funding input (58 vB). In an operator-funded graft the member pays off-chain, after checking its path against the confirmed root, or through a hash-locked voucher (§5.2).

JOIN_K: batched single joins. K single-leaf joins under one operator signature, the batching primitive of the sparse profile, which cannot graft; 1,494 vB for two.

B.2 Moving Value

Most payments never touch the tree. Members with channel leaves pay over Lightning through the operator (§9), and vouchers (§5) move value with a signature that is redeemed only if the issuer stops cooperating. The tree changes when a balance’s terms change or a carve is redeemed.

UPDATE and UPDATE^x: changing a leaf. Replaces one leaf with another at the same position: a splice in or out, a lease renewal, a key rotation or a transfer of ownership. *Authorised by* the leaf’s owner and the operator together; UPDATE^x, a variant chosen at creation, adds the leaf’s exit key as a third signer. *Proves* the old leaf against the root and writes the new one. *On chain* the tree output changes by the difference: an increase is funded by other inputs, a decrease is free for the transaction to spend, and both signers see where it goes. 1,030 vB, or 1,048 vB with the third signature. UPDATE^x exists for leaves whose exit key is shared with or owned by someone else (a channel’s counterparty, a credit leaf’s

recipient, §5.2): without it the owner and the operator could splice such a leaf down together.

SPLIT: paying a member directly. Carves a new leaf out of the payer’s leaf. *Authorised by* the payer’s owner key, signing this transaction. *Proves* two positions in sequence: the source leaf, rewritten with its amount reduced by the payment and its `seq` raised by one (or emptied if drained), then an empty position under the intermediate root, which receives the new leaf. *On chain* the total is unchanged and no payout is made. The new leaf keeps the source’s evictor key and a lease no later than the source’s, and neither it nor a non-zero residual may be below `DUST_MIN` (both rules are shared with `CLAIM`, §5.1). 1,586 vB. `SPLIT` suits a payer that broadcasts and pays the fee itself, leaves no outstanding promise, and is the one member-to-member transfer `MIN` supports. Its deferred counterpart, a voucher redeemed by `CLAIM`, is the subject of §5.

B.3 Settling a Cohort

REWRITE_h: everything at once. Replaces the contents of one aligned subtree of height h in a single spend: any position may go from a leaf to empty (an exit, paid out), from empty to a leaf (a join, funded) or from one leaf to another (a payment, renewal, resize or key rotation). *Authorised by* the operator and by the owner of every leaf that changes; a leaf that only gains amount needs no signature unless its `seq` is `SEQ_FINAL` (a channel leaf, §9, whose capacity its signed states fix), so a payment within a cohort needs only the payer’s, and a leaf nobody signs for passes through unchanged. *Proves* the old subtree, every leaf of it published, against the root, and computes the new one. *On chain* each emptied leaf is paid exactly its amount at its exit key. At $h = 6$ a mass exit costs 165 vB per leaf, about what a sweep costs the operator, and rewriting every balance 151 vB per leaf against 1,030 vB for an `UPDATE` each; a position whose leaf does not change is published once, behind a one-byte marker, for about 95 vB. `REWRITEhx` adds the exit key’s signature (16 vB per signed position) and replaces `REWRITEh` wherever `UPDATEx` or `CLAIMx` is present. It exists in every tier but `MIN`; under `0xC0` the stack limit holds a full cohort only to $h = 5$.

B.4 Leaving

EXIT_K: unilateral exit. Removes one leaf, or K leaves in one transaction, and pays each out (Figure 1). *Authorised by* the leaf’s owner key alone; no operator, no other member, no timelock. *Proves* the leaf and its path against the root and folds the empty node up the same path for the new root. *On chain* the tree output shrinks by the leaf’s amount and output 1 pays its exit key. In `FULL` the payout may be less than the balance, the difference paying the fee, which saves the fee input (99 vB); the exact-payout variant `EXITK-`, chosen at creation and visible in the tree’s taptree root, requires the full amount and is what `MID`, `MIN`, `CCV` and channel leaves use. 1,020 vB for one exit, and 665 vB more for a second in the same transaction (`EXIT2` is 1,684 vB). The owner’s signature

commits to the tree outpoint, which every operation changes, so a plain exit cannot be pre-signed: the owner must be online to rebuild it after a competing spend. The tower variant EXIT_K^t (FULL, MID and CCV, exact payout) replaces that signature by one over the leaf without its amount and the tree's identifier, checked from the stack with `OP_CHECKSIGFROMSTACK`; whoever holds it builds the exit with its own fee input and rebuilds it after a competing spend, so an owner signs once in advance and a tower does the rest. The authorisation is a bearer right until a field other than the amount changes (a top-up leaves it valid), so an owner who stays voids it by changing the leaf. Rejoining any tree of the lineage with an owner key, nonce and seq held before would revive every authorisation published for that leaf, so a wallet draws a fresh nonce at every join (spec 9.2). 110 bytes of script over the exact exit. Owners first ask for a cooperative exit, which the operator can batch, and a held authorisation makes that one message.

B.5 Being Evicted

Eviction by EXIT. A lease is how the operator recovers its capital and slots from a member who has gone away, and eviction is the only way it can remove a leaf without the owner. The exit script, with a flag, removes a leaf whose lease has passed. *Authorised by* the leaf's evictor key, with a lock time at or after the lease height. *On chain* the payout is exactly the leaf's amount, to the leaf's own exit key, so an eviction returns the member's money rather than taking it. Same cost as an exit; the evictor pays the fee.

EVICT_SUBTREE_h: the cohort sweep. Evicts every remaining leaf of one aligned subtree in one spend. *Authorised by* the operator key, which must be every swept leaf's evictor, with a lock time past every lease. *Proves* the subtree from all its leaves (empty positions contribute E_0), each internal node hashed once, against the root. *On chain* one exact payout per leaf, and the subtree becomes empty. 203 vB per leaf for 13 leaves at $h = 4$ and 158 vB per leaf for 51 at $h = 6$, against 1,020 vB each alone. One leaf with a later lease holds up its cohort's sweep until that lease passes, and one with another evictor blocks it, so operators admit only leaves that name the operator as evictor into a cohort, align leases per cohort, and renew a member by moving it to a cohort with the new lease (an UPDATE and a SPLIT of the whole balance, 2,616 vB), or renews a whole cohort in place by one REWRITE₆ that every renewing member signs (151 vB each; a member who does not sign keeps the old lease).

B.6 Scaling Out

DETACH_h: a cohort's own output. Moves one aligned subtree into a tree output of its own. *Authorised by* the operator key. *Proves* the subtree's root and sum against the parent root. *On chain* output 0 is the parent minus the cohort and output 1 the cohort, under its own root and the parent's rules compiled at depth h ; the detached tree inherits the parent's tree identifier, so vouchers and channel states against its leaves stay valid. No member needs new data: its proof in the

detached tree is its old path cut to the lower h levels. 780 vB at $h = 8$, after which every exit from the cohort costs 762 vB instead of 1,020; the detach pays for itself at the fourth exit. Its larger effect is throughput, since each output is spent independently (§6).

B.7 Who Can Do What

Table 4. Who can do what (spec 10.1). “Carve” is SPLIT or a redeemed voucher.

party	can	cannot
Operator	refuse to co-sign; graft leaves, including ones it funds; evict a leaf after its lease, paying it in full, with everything carved from it; issue vouchers on its own leaves; detach a cohort; fill a cohort’s empty positions, stalling partial-balance vouchers	move or reduce another leaf; delay an exit beyond [C]; freeze value it did not graft and fund; change the taptree, or unilaterally a lease or a detached cohort’s rules, keys or balances, other than raising an amount
Owner	exit at any time; co-sign updates; issue vouchers against its leaf; carve its leaf, into any empty position (blocking a graft there; a later lease than the cohort’s delays its sweep)	change another leaf; withdraw more than its amount; carve a leaf with another evictor or a later lease than its own
Anyone	redeem a valid voucher; relay transitions; exit a leaf whose EXIT ^t authorisation it holds	trigger an owner’s exit without one; create leaves; alter amounts

C Vouchers in Detail

C.1 Cumulative Vouchers

Every voucher against a source draws on the same leaf, and the first redemption voids the rest. An issuer that pays one recipient repeatedly therefore signs a *cumulative* voucher for the running total against the same claim leaf, and re-signs it for a larger amount at each payment: 2, then 5, then 9, one signature per payment. The recipient keeps only the latest and, if the issuer stops cooperating, redeems that one alone; a smaller one is never worth more to it, so no contest window.

Settled credit is retired by advancing **seq**. Once the issuer pays the recipient another way, it advances the source’s **seq**, which voids every outstanding voucher against it: one UPDATE per source leaf (1,030 vB), or one REWRITE _{h} for a cohort of them (about 151 vB per leaf at $h = 6$). Until then the issuer tracks every voucher it has signed as a liability. Plain CLAIM leaves a hole: anyone holding an old voucher can redeem it, and the issuer holds them all, so redeeming the voucher for 2 voids the one for 9. The credit leaf below closes it.

C.2 The Dedicated Credit Leaf

A dedicated credit leaf is a source leaf that the issuer owns and whose exit key is one recipient's, in a tree whose taptree carries CLAIM^x , UPDATE^x and the exact-payout exit, carries REWRITE_h^x in place of REWRITE_h (or no rewrite at all), and omits SPLIT . CLAIM^x is CLAIM with one further signature over the transaction, by the source leaf's exit key: 7 bytes of script and 64 of witness in FULL , the only tier that builds it. The recipient's own redemptions carry that signature at no cost, and the issuer cannot carve the leaf at all, not even with a voucher it signs to itself. Over-issuance then has no second victim, since every voucher pays the same recipient, who can watch the leaf's amount. Voiding needs a field the voucher binds to change, and every transition that could change one needs the exit key: CLAIM^x , UPDATE^x and REWRITE_h^x by rule, while the exact exit and an eviction pay the whole balance to that key and so void nothing the recipient is owed. The one change the operator makes alone, an amount-only top-up, touches no bound field (spec 0.13.0). The balance can only grow or pay its recipient, whose remaining exposure is the issuer's liveness: a provider that stops issuing.

Three details keep it that way. The recipient checks from the taptree root m that no plain REWRITE_h is present, since it needs only the owner's and the operator's signatures, both the issuer's. The issuer funds the leaf only with credit it has issued or is about to, since any excess is the recipient's on every path the issuer can take alone, and tops it up by an amount-only increase under REWRITE_h^x , which needs only the operator's signature and leaves the latest voucher valid. And the leaf has a finite lease with the operator as evictor, not a NUMS evictor, which every carve would inherit, blocking the sweep of whatever cohort the claim leaf lands in. The credit is one-way: if the recipient could also spend from it, an older, larger voucher would be a stale state needing a window to supersede, as in a bidirectional channel. To spend, the recipient redeems into a channel leaf (§9).

C.3 Hash-Locked Vouchers and Swaps

CLAIM^h adds a payment hash to the signed message, under its own tag so that the lock cannot be stripped, and requires the 32-byte preimage: the script hashes it into the message, so a wrong preimage fails as a bad signature. Paired with an HTLC or invoice on the same hash, it is an ordered exchange, not an atomic swap: the issuer can void the voucher until the redemption confirms, and script cannot give a voucher a deadline. Five rules (spec 9.6.2) order it. (1) The party the HTLC pays draws the preimage. (2) The voucher resolves second; no HTLC waits on a preimage that only a redemption would reveal. (3) The issuer leaves the source's seq alone from the moment an HTLC on the hash is offered to it until the voucher is redeemed or the HTLC fails. (4) No script enforces rule 3, so the holder redeems as soon as it learns the preimage and bears the issuer risk until that confirms. (5) The holder first checks that the source's lease outlasts the HTLC's expiry plus $\Delta(q)$. Usually the HTLC settles off-chain, so such a swap costs at most one CLAIM^h ; it is also how a newcomer buys a leaf over Lightning.

D Costs Against the Alternatives

Joining, updating and leaving a shared output each cost more bytes than the same operation on a bare UTXO. What the sharing buys is a different placement of the cost. Entry is close to free in bulk, value moves off-chain for nothing, a bulk exit costs about a bare spend, and a solo exit costs about nine times one but is a single transaction with no timelock and no counterparty. This appendix walks through each of those claims with the measured figures, then says where LayerTree loses and how the numbers are made (§D.10).

D.1 The Unit of Cost

Every LayerTree operation is one transaction that spends the tree output and recreates it. The witness carries a leaf and its Merkle path, and the script folds the path twice, once to check the old root and once to compute the new one. Each level costs 88 bytes of script and 41 of witness in FULL, 129 weight units or about 32 vB. The depth D is fixed when the tree is created, so an exit from a given tree is a constant size whatever the population: 1,020 vB at $D = 16$, which holds 65,536 slots.

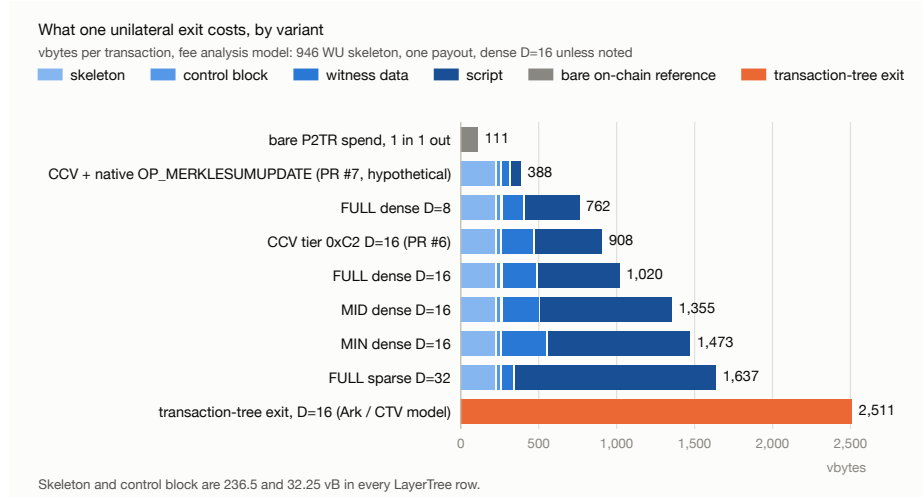


Fig. 3. One unilateral exit by variant. The skeleton (236.5 vB) and control block (32.25 vB) are the same in every LayerTree row; the script dominates because tapscript has no loops and the levels are unrolled. The sparse and native-opcode rows are measured on the simulator’s six-leaf tree. The “PR” labels name the repository changes that added the CCV tier and the hypothetical opcode.

Figure 3 splits the exit into its parts, and Table 5 lists every variant against a bare spend. MID and MIN pay more for the same work because they lack

the restored arithmetic and point opcodes; CCV saves about a tenth by letting `OP_CHECKCONTRACTVERIFY` bind the state. The hypothetical native Merkle-sum opcode of spec 13 item 12 is a measurement, never a tier: on the simulator’s six-leaf tree its exit is 388 vB, with 252 bytes of script at any depth, but the witness still carries one 40-byte sibling per occupied level, so a full depth-16 tree costs about 520 vB. That is the floor for any design that publishes a proof and checks it in script. The transaction-tree row is the model of §10: D transactions of about 150 vB to unroll the branch, plus a 111 vB claim. It flatters the tree, since it omits the relative timelock at the leaf and the fee each of the 16 transactions pays separately; Table 6 gives the same model at other round sizes and radix 4.

Table 5. One exit, dense $D = 16$ unless noted. Model: the transaction model of §D.10. Serialised: the real transaction the test suites build, with the witness length prefixes the model leaves out (`spec/scripts/weights.md`).

variant	model (vB)	serialised (vB)
bare P2TR spend, one input and one output	111	
one P2TR output inside a batch transaction	43	
FULL EXIT ₁	1,020	1,030
FULL, fee taken from the payout	921	
FULL, $D = 8$ and $D = 20$	762 and 1,149	
FULL tower exit EXIT ₁ ^t	1,046	
FULL sparse $D = 32$, six-leaf tree	1,637	1,648
FULL sparse $D = 32$, 2^{16} members (measured)	1,781	
MID	1,355	1,398
MIN	1,473	1,518
CCV, tapleaf 0xC2	908	918
native Merkle-sum opcode (hypothetical), six-leaf tree	388	
native Merkle-sum opcode (hypothetical), full $D = 16$ tree	≈ 520	
radix-2 transaction-tree exit (model)	2,511	

Table 6. The transaction-tree model across round sizes N . A level is one key-path P2TR input and r P2TR outputs, $68 + 43r$ vB (154 at radix 2, used as 150; 240 at radix 4); the claim is a plain P2TR spend, 111 vB; anchors and their fee children are omitted, which favours the tree. Mass exit: the tree’s interior shared by all members, against LayerTree’s REWRITE₆ or sweep of a 64-slot cohort. LayerTree is measured at $D = \log_2 N$.

	LayerTree, dense	radix-2 tree	radix-4 tree
isolated exit, $N = 2^8$	762	1,311	1,071
isolated exit, $N = 2^{12}$	891	1,911	1,551
isolated exit, $N = 2^{16}$	1,020	2,511	2,031
mass exit, per member	158 to 165	≈ 261	≈ 191

Table 7. Batch shapes at dense $D = 16$ on the transaction model: vB per leaf that leaves or changes, or per member that joins. Self-funded joins add each member’s P2TR key-path funding input (58 vB) and the signing round of §4.1. The REWRITE_h rows are FULL’s figures; JOIN_8 is an estimate, the measured JOIN_0 at $D = 8$, which walks the same eight levels. A renewal alone is an **UPDATE** and a whole-balance **SPLIT**; a cohort renewed (or its keys rotated) in place by REWRITE_6 needs every renewing member’s signature.

removal	leaves	tx	per leaf	addition or change	leaves	tx	per leaf
EXIT_1	1	1,020	1,020	JOIN_0	1	937	937
EXIT_2	2	1,684	842	JOIN_2	4	872	218
EVICT_SUBTREE_4	13	2,638	203	JOIN_8 (operator-funded)	256	679	2.7
EVICT_SUBTREE_6	51	8,057	158				
REWRITE_2 , all leave	4	1,465	366	REWRITE_6 , renewals	64	9,656	151
REWRITE_6 , all leave	64	10,552	165	UPDATE , one leaf	1	1,030	1,030
				renewal, alone	1	2,616	2,616

D.2 Exits in Bulk

A single exit is the expensive case. The cost per leaf falls as more leaves leave in one transaction, and how far it falls depends on who has to sign (Figure 4, Table 7).

Owners who coordinate leave together with EXIT_K . Each leaf still carries its own path, so EXIT_2 is 1,684 vB: 842 vB a leaf, and 665 vB for the second exit alone. *The operator* sweeps an aligned cohort of expired leases with EVICT_SUBTREE_h , which publishes the cohort’s leaves once and shares the path above it: 203 vB a leaf for 13 leaves at $h = 4$, 158 vB for 51 at $h = 6$, and about 147 vB for the 205 leaves of a height-8 cohort at $D = 20$. The sweep needs no member, but it only works on leaves whose evictor is the operator and whose lease has run out. A leaf with the **NUMS** evictor, or with a lease far in the future, blocks the sweep of its whole cohort (spec 7.7), so the operator’s cheap removal is something members opt into at join time by accepting a lease.

Owners who want the sweep’s price without a lease use the cohort rewrite REWRITE_h (§4) with every position leaving. It is the same post-order fold, authorised by one plain signature per leaf and the operator’s, with no nonce round. At $D = 16$ four owners leaving a height-2 cohort pay 366 vB each, sixteen at $h = 4$ pay 202 vB, and sixty-four at $h = 6$ pay 165 vB, within seven vbytes of the operator’s sweep. The same script settles a cohort in place: rotating every key of a height-6 cohort is 151 vB per changed leaf against 1,030 vB for an **UPDATE** each, and a position that does not change costs about 95 vB and needs no signature, so an absent member is carried through rather than excluded.

Batching changes what the chain limit counts. Every transition spends the same output, so unconfirmed transitions form a chain, and relay policy caps it: 25 transactions under the ancestor and descendant limits of Bitcoin Core before 31.0 [2], at most 64 under cluster mempool [8] (κ of §6). The cap counts transactions, not leaves, so a 51-leaf sweep is one entry. Uncoordinated exits chain only if each exiter rebuilds on the previous unconfirmed exit, or a tower or the operator holding EXIT^t authorisations builds them in sequence; otherwise

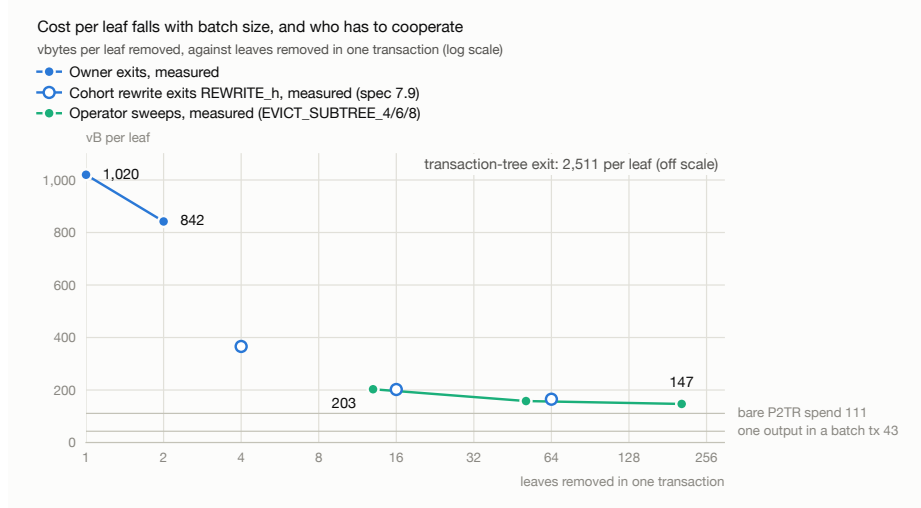


Fig. 4. Cost per leaf against leaves removed in one transaction (log scale): owner exits (EXIT₁, EXIT₂), owners leaving by REWRITE_h (366, 202 and 165 vB at 4, 16 and 64 positions), and operator sweeps (EVICT_SUBTREE_{4,6,8}: 13, 51 and 205 leaves).

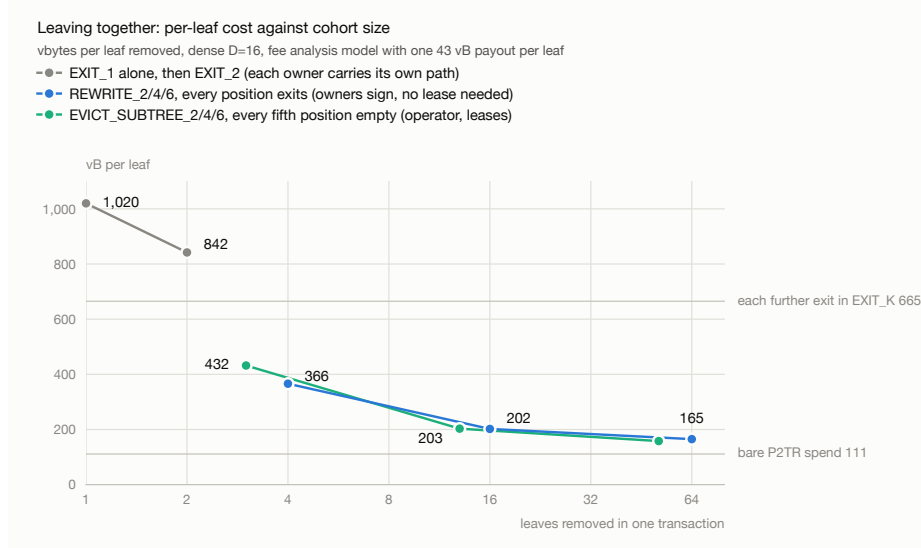


Fig. 5. The same three ways out at $D = 16$, per leaf removed against the number removed in one transaction: owners alone or in pairs, owners leaving together by REWRITE_h, and the operator's sweep. The rewrite starts below the EXIT_K floor and meets the sweep near $h = 6$ (165 against 158 vB per leaf).

they conflict and replace one another. So 25 a block is the cooperative ceiling for single exits, and the worst case is set by β . A deployment that expects more shards into several trees, each sized by exit throughput rather than by depth. Each batch shape is its own tapleaf, and each doubling of the taptree adds 8 vB to every spend, so a tree that carries the sweeps and rewrites (16 to 32 tapleaves) pays 8 to 16 vB more on every exit.

D.3 Entry

JOIN_h grafts a prebuilt subtree of height h into an empty slot and walks only the $D - h$ levels above it, so a bigger cohort is a cheaper transaction (Figure 6). At $D = 16$ with no payout output, as spec 11 prices a join, one member costs 937 vB, a four-member graft 872 vB (218 vB a member), a 256-member graft about 679 vB (2.7 vB) and a 1,024-member graft about 614 vB (0.6 vB), both estimates from measured joins that walk as many levels, against 43 vB for one output in an on-chain batch and about 154 vB for a channel opened in its own transaction.

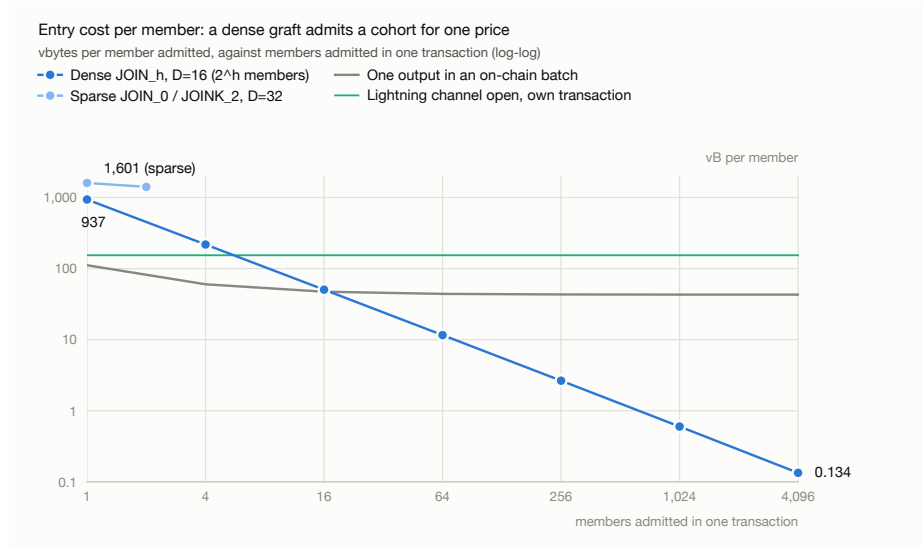


Fig. 6. Entry cost per member against members admitted in one transaction (log-log). The figure prices each join with one payout output, 43 vB above the no-payout pricing of spec 11 that the text and Table 7 use: its 980 and 1,644 are 937 and 1,601 there. The dense points above 4 members are extrapolated from the measured JOIN_2 at 129 weight units a level.

Who funds the value grafted in decides who has to be online. When the operator funds it from the transaction’s fee input, the graft needs the operator’s

signature alone, and the whole cohort costs nothing beyond the figure; members pay for their balances off-chain after the graft confirms, or beforehand with a hash-locked voucher (§5). When each member funds its own leaf, each adds a key-path input of about 58 vB, the saving against an on-chain batch shrinks to the output side, and the cohort becomes a signing round among the funders: every signature commits to the new root, so a joiner who drops out invalidates the others and the operator collects again (§4.1). Either way, a member checks its own path inside the subtree before funding, because a grafted node with no real tree behind it freezes its value. The sparse profile cannot graft at all: a sparse join is one leaf for 1,601 vB.

D.4 Moving Value

On-chain transitions are the wrong path for payments. An `UPDATE` at $D = 16$ is 1,030 vB and a voucher redemption with `CLAIM` 1,618 vB, priced as spec 11 prices them, without a payout output (1,073 and 1,661 vB on the skeleton that carries one), against 111 vB for a bare spend. A member who moved value by rewriting its leaf would pay about nine times what separate UTXOs pay per payment (Table 8).

Value moves off-chain instead, and the chain carries only membership changes and the exercise of exit rights. Payments between channel leaves are ordinary HTLC forwards; their funding outputs do not exist until a leaf exits (§9). A voucher is one signed message from payer to payee, credit until redeemed, and an issuer usually settles it off-chain rather than the holder redeeming it (§5). `UPDATE` is maintenance: resizing a balance, renewing a lease, rotating a key. When a cohort's balances do have to settle on-chain, `REWRITEh` does it for about 151 vB per changed leaf with only the changed leaves' owners signing, which is the settlement an Ark round performs without the round.

Table 8. 1,000 members who each make 1,000 payments, entry and exit included where a design has them. The Lightning row opens and closes each channel cooperatively in its own transaction; the Ark row boards and leaves as outputs of round transactions, whose shared overhead is left out; the LayerTree rows enter by one 1,024-slot graft.

how value moves	on-chain bytes
bare UTXOs, one spend per payment	111 MvB
LayerTree, one on-chain <code>UPDATE</code> per payment	1,030 MvB
Lightning channels, payments off-chain	308 kvB
Ark, payments off-chain (rounds and refreshes not priced)	86 kvB
LayerTree, payments off-chain, cohort exit or sweep (renewals not priced)	about 160 kvB
LayerTree, payments off-chain, members never exit (balances sold by voucher)	under 1 kvB

D.5 The Lightning Baseline and Lifetime per Member

The Lightning figures are a simple taproot channel [26] priced on the same model as the LayerTree rows, fee-bumping children included (Table 9). The non-taproot anchor commitment of BOLT 3 [19] is 281 vB in place of the 240 used here.

Table 9. A simple taproot channel on the transaction model.

transaction	vB
funding: one P2TR key-path input, the funding output and change	154
cooperative close: the MuSig2 key-path spend of the funding output	154
commitment: a MuSig2 key-path input, two balance outputs and two anchors	240
anchor child that pays the commitment’s fee	152
delayed sweep of the broadcaster’s output after the CSV	138
force close: commitment, anchor child and sweep	530
lifetime, open alone and force close	684
lifetime, open alone and cooperative close	308
lifetime, open as one output of a batch and cooperative close	197
lifetime, open as one output of a batch and force close	573

Against this baseline a member who leaves a dense $D = 16$ tree alone pays 1,020 vB, about twice a channel’s force close and 1.5 times its 684-byte lifetime. A channel leaf inside the tree costs its exit plus an update, a settlement and their two anchor children, about 1,681 vB in all, against 684 vB for a channel with its own funding output, because materialising the funding output carries the Merkle-sum proof. The fair comparison adds entry and exit for one member and asks how the member leaves (Figure 7, Table 10).

Lifetime and latency. Table 10 adds entry and exit. The dense tree wins outright for a member who never exits alone or who is swept, and loses to Lightning and to a bare UTXO for a member who exits alone, though it beats an Ark unroll there. Members who hold a lease and let the operator sweep them, or who sell their balance by voucher and never exit, pay almost nothing; members who insist on leaving alone pay for the proof.

D.6 Time and Liveness

Bytes are half of an exit’s cost. The other half is how long it takes and who has to be online (Table 11). A LayerTree exit has no pre-signed transaction to lose, no counterparty to wait for, no timelock and no expiry: the covenant checks a proof and pays. A plain exit’s signature commits to the tree outpoint, which every operation changes, so the owner has to be online to build it and to rebuild it after a competing spend. The tower exit EXIT_K^t (§4) lifts that: the owner signs the leaf without its amount and the tree’s identifier once, in advance, and whoever holds the signature builds and rebuilds the exit against whatever root is current. A channel leaf gives up some of the rest in exchange for free payments.

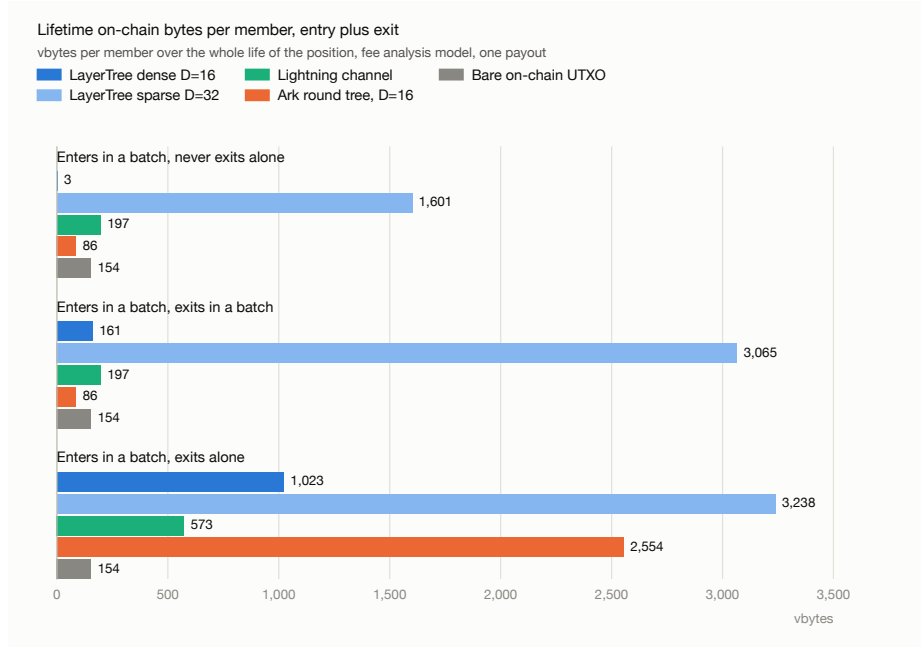


Fig. 7. Lifetime on-chain bytes per member, entry plus exit. The dense, Lightning, Ark and bare rows are those of Table 10. The sparse bars price the join with a payout output and the exit on the six-leaf tree (1,644, 3,108 and 3,281); Table 14 gives the sparse join and exit sizes, and its Lifetime paragraph prices the join without a payout output and the exit at 2^{16} members (1,601, 3,209 and 3,382).

Table 10. Lifetime on-chain bytes per member, by how it leaves, and the unilateral exit’s latency (a tower watches for stale states; an Ark member must refresh before expiry). LayerTree rows are dense at $D = 16$ (the sparse profile is in Appendix D.7). Entry is one 43 vB output in a shared transaction; for LayerTree it is a share of a 256-slot operator-funded graft, or that plus the member’s 58 vB input when self-funded. LayerTree’s batch exit is the 51-leaf sweep, which applies only to expired leases aligned per cohort; keeping a lease costs a renewal per period, 2,616 vB alone or 151 vB in a cohort rewrite, not included. Lightning is a simple taproot channel: cooperative close 154 vB; force close a 240 vB commitment, its 152 vB anchor child and a 138 vB sweep. Ark’s unilateral exit is the radix-2 unroll of its round. *Nobody with EXIT^t , whose holder rebuilds the exit.

	vB per member, leaving			unilateral exit		
	never alone	in a batch	alone	txs	blocks	online
LayerTree, operator-funded	3	161	1,023	1	1	owner*
LayerTree, self-funded	61	219	1,081	1	1	owner*
Lightning channel	197	197	573	3	≈ 145	tower
Ark, round of 256	86	86	1,354	9	8+	timelock member
Ark, round of 65,536	86	86	2,554	17	16+	timelock member
bare UTXO	154	154	154	1	1	nobody

Table 11. The unilateral exit’s latency and liveness. Ark is a round of 2^{16} members; the channel leaf and the Lightning channel close by force.

	LayerTree EXIT _K	LayerTree EXIT _K ^t	LayerTree channel leaf	Lightning	Ark
transactions	1	1	3, plus fee children	3: commitment, anchor child, sweep	16, plus the claim
blocks until spendable timelock	1 none	1 none	about 146 CSV on the settlement	about 145 CSV on the commitment	16 plus the leaf timelock relative lock at the leaf
who must be online	the owner, to build, sign and rebuild if outbid	nobody; a tower holds a one-time authorisation	the closer; a tower can finish	a tower, for stale states	the member, before the round expires
what expires	nothing; a lease is the price of the cheap batch exit	nothing; a changed leaf (not a top-up) voids the authorisation	nothing	nothing	the round: refresh or forfeit

The exit is one column of a wider question: who has to be awake for each thing a member does. Table 12 answers it for the designs of §11, and Table 13 sets their structure side by side. Across LayerTree’s row the only party ever needed for *another* member’s operation is the operator, and only to add members, co-sign an UPDATE or settle a cohort; no member is needed for another’s exit, join, payment or eviction. Pre-signed designs need the whole set at creation because the interior is signatures, and the round-based ones make every member come back before an expiry. Covenant trees remove the signatures but keep the $O(\log N)$ exit. The pools match the one-transaction exit, each with a purpose-built opcode and with every change to the pool signed by every participant.

D.7 The Sparse Profile

In the sparse profile a leaf’s position is the low D bits of $H_{\text{pos}}(\text{owner_key} \parallel \text{nonce})$, so the tree is the Merkle-sum sparse Merkle tree of Taproot Assets [23] truncated to $D = 32$ or 64 . Direction bits are derived in script from the position hash, and empty siblings are elided from the witness (the script substitutes E_k), so the witness grows with $\log_2 N$ rather than with D . A sparse position is a function of the leaf: a joiner computes it alone, a key and nonce cannot appear twice, the root depends only on the leaf set, and absence is an inclusion proof of E_0 . None of LayerTree’s predicates needs these properties. An exit proves inclusion and a join proves that its slot is E_h , the same way in both profiles. The dense profile has aligned subtrees instead, on which JOIN_h, EVICT_SUBTREE_h and every batch shape depend, so the sparse profile cannot graft or sweep and batches only by JOINK_K. It still hosts channel leaves and vouchers and still exits in one transaction.

Table 12. Who has to be awake, by operation. “All” is every member of the shared output; a round is a signing session that restarts if one signer drops.

	join	pay off-chain	settle on-chain	exit	evict	expiry
LayerTree	operator alone, or a round among self-funders	one message (voucher), or two parties (channel); from Lightning, via the operator	UPDATE: one exchange; REWRITE _h : the changed owners	alone, 1 tx	operator alone, after the lease; member paid	none
Lightning [29] Factory [5]	two parties all	two parties two parties, in a channel	two parties all, every state	alone $O(\log N)$ pre-signed txs	n/a no	none none
Ark, pre-signed [16]	the round, in a window	server co-signs, trusted until the next round	the next round, with everyone	$O(\log N)$ txs plus a timelock	no; the round expires	every round
Ark, covenant [30]	funders sign their inputs	as above	the next round	$O(\log N)$ txs	no; the round expires	every round
Timeout tree [17] Payment pool [37,22,42]	all at creation, or CTV all sign the new state	channels inside all sign	a new tree all sign	$O(\log N)$ txs alone, 1 tx	no; the tree expires OP_EVICT: the others sign	per tree none
CTV or CCV tree [31,13]	operator builds the templates	varies	a new tree	$O(\log N)$ txs	no	per design

Table 13. Comparison with shared-output constructions. Pools: OP_TAPLEAF_UPDATE_VERIFY [37], CoinPool’s OP_MERKLESUB [22], OP_EVICT [42]. GSR: the string opcodes BIP-441 restores; APO: SIGHASH_ANYPREVOUT. LayerTree’s interactive signing: the covenant needs the operator only; a self-funded cohort’s funders sign one transaction together.

	LayerTree	Pools	Ark (deployed)	Ark (covenant)	Timeout tree	Factory
Unilateral exit	1 tx, $O(D)$ bytes	1 tx	$O(\log N)$ txs	$O(\log N)$ txs	$O(\log N)$ txs	$O(\log N)$ txs
Evict one member	1 tx after its lease	1 tx (EVICT)	round expires	round expires	tree expires	no
Member position expiry	lease, renewed by the operator; eviction pays the member	none	per round; operator reclaims	per round	per tree	none
Interactive signing	operator only; a self-funded cohort once	all, to update	round participants	funders, per round	none or all	all
Shared output expiry	none	none	per round	per round	per tree	none
Batch add	1 tx, any subtree	re-sign	per round	per round	per tree	re-sign all
Opcodes	FULL: GSR, TX, CSFS, EC; MID: CAT, TX, CSFS, TWEAKADD; MIN: CAT, TWEAKADD; CCV: CCV, TX	a pool opcode (and APO for CoinPool)	none	CTV or CCV	CTV	none

Table 14. The sparse profile, FULL tier, on the model of §10. The witness is $244 + 41k$ bytes for k non-empty siblings: $k = 2$ on the measured six-leaf tree (326 B), and $k = 16$ on the measured 2^{16} -member tree (900 B), the mean over the members being $\approx \log_2 N$; the $D = 64$ row is an estimate. Joins are charged without a payout output.

case	D	N	script (B)	witness data (B)	witness (WU)	tx (vB)
EXIT ₁ , measured	32	6	5,147	326	5,602	1,637
EXIT ₁ , measured	32	2^{16}	5,147	900	6,176	1,781
EXIT ₁ , estimate	64	2^{16}	9,499	≈ 900	$\approx 10,528$	$\approx 2,869$
JOIN ₀ , one leaf	32					1,601
JOINK ₂ , two leaves	32					2,821

Where the bytes go. A sparse proof is the smaller witness and the larger script, and the script wins (Table 14). Both causes follow from hashing the position. Random positions collide, with $N^2/2^{D+1}$ expected collisions among N members (about one half at $D = 32$ for $N = 2^{16}$, about 32,000 at $D = 16$), so the sparse tree needs about twice the depth, and every level is unrolled. A sparse level is also 135 bytes of script against 88, because it derives the direction bit from the position, substitutes E_k for an elided sibling and computes E_{k+1} for the next level. The full-width tree of Taproot Assets ($D = 256$) costs 35,611 bytes of script per exit, about 17 times the dense script and 9 times the dense transaction, and MID caps sparse depth at 31. An exit is 1,637 vB on the six-leaf tree and 1,781 vB at 2^{16} members (measured), against 1,020 vB for dense at $D = 16$; at 10 sat/vB the six-leaf exit costs 16,370 sat. Varops usage is 3.6 to 4.7% of the BIP-440 budget ($D = 64$ to 32), against 5.6 to 7.7% in the dense profile.

Lifetime. A member who enters by one sparse JOIN₀ (1,601 vB) and leaves in a batch or alone at the 2^{16} -member exit pays 3,209 vB or 3,382 vB in all, against 161 and 1,023 vB for an operator-funded dense member (Table 10); the sparse profile is the most expensive design in every cost column of that table.

Radix. A radix- r path carries $41(r-1)D/\log_2 r$ sibling bytes, so radix 2 minimises the dense witness, and a measured radix-4 level brings the dense exit to 1,042 vB and radix 8 to 1,257 vB, against 1,020 vB at radix 2. In the sparse profile most siblings are empty and cost one witness byte, so a radix-4 level of about 180 bytes could replace two levels of 135 and save about 360 vB per exit (estimate; no sparse radix- r level has been written).

Dense is therefore the default (spec 5.4, `spec/layertree.md`): 1.7 to 2.8 times cheaper per exit at 2^{16} members, with cohort grafts at one price, sweeps at about 160 vB per leaf and every batch shape, at the price of an operator who keeps the slot map. The sparse profile is kept, specified and tested as the measured alternative, and for deployments that are key-addressed registries.

D.8 A Native Merkle-Sum Opcode (Hypothetical)

Tapscript has no loops, so the tiers pay for every level of a path in script, and that is the cost they change only by a constant. A hypothetical OP_MERKLESUMUPDATE

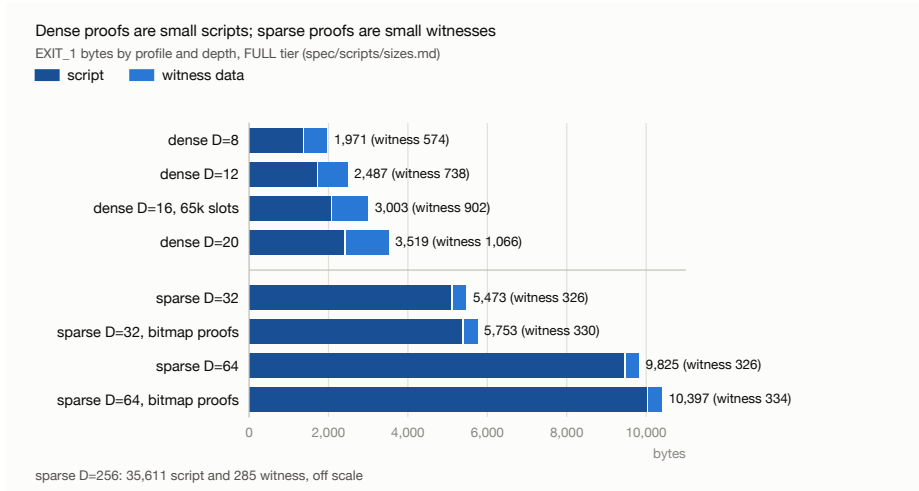


Fig. 8. EXIT₁ bytes by profile and depth, FULL tier: dense proofs are small scripts, sparse proofs small witnesses. Sparse rows are measured on the six-leaf tree.

would take a proof, an old node and a new node and return the root of the old node's path and the root with the new node in its place. The proof is a depth, a position and a bitmap naming the levels that carry a sibling, then those 40-byte siblings; every other level uses the empty node E_k . The opcode adds the sums, fails on overflow past 2^{64} or on a non-canonical proof, and costs $2 \times$ the depth in hashes, charged as BIP-440 varops. An old node equal to the new one is plain verification. It replaces the unrolled fold and the sum arithmetic along the path; the script that remains checks the leaf, the payout and, for a pending receive, the amounts it moves. Nobody has proposed it and nothing here depends on it; it runs only in the simulator, behind a flag, in no tier and in no golden script.

EXIT₁ then has 252 bytes of script at every depth (2,101 in FULL and 1,716 in CCV at $D = 16$) and one 40-byte witness sibling per occupied level. Empty levels cost nothing, so a hashed-position tree (Appendix efapp:sparse) costs about the same as a dense one. For 2^k members in a key-addressed depth-64 tree the exit is about $381 + 10k$ vB (541 at 2^{16} , 701 at 2^{32}), against $504 + 32.25k$ for FULL (1,020; 1,536) and $150k + 111$ for a transaction tree (2,511; 4,911): 10 vB per doubling and no fixed depth. The 388 vB of Figure 3 and the 520 vB of a full depth-16 tree are the simulator's smaller model; 541 is the key-addressed one. The CLAIM^{hm} redemption of a pending leaf is 626 vB at 2^{16} members against 1,136 vB in FULL. This and the settlement figure below assume a proof that can start above height 0 (a start-height field), or a dense tree of depth exactly k : the opcode's empty chain starts at E_0 , so without either a height- h proof carries every empty level, about 480 vB more per merge or rewrite at 2^{16} members.

It does little for batches. A rewrite is dominated by a per-position step of about 376 bytes of script at $h = 6$, which the opcode does not touch: settling pending receives costs 547 vB a pair at 64 positions (two transactions of 8,347 and 9,155 vB at 2^{16} members, 24,065 bytes of script each against 25,036 in FULL). A multi-leaf exit can chain K calls, each call's old root the previous call's new root; a sweep needs the subtree root of its 2^h leaves, which the proof format above does not express. The price is that the opcode fixes in consensus this tree's node encoding, empty-node chain, parent hash and overflow rule, so a different tree or a format change would need a different opcode. That is why the work composes proposed opcodes and proposes none, and treats the opcode as a measurement of what the general approach pays, not as a tier.

D.9 Comparison with Other Tree-Based Protocols

A member who never leaves alone pays almost nothing: about 3 vB for a share of a graft, and about 160 vB in all if it leaves with its cohort. A member who does leave alone pays 1,020 vB, about nine times a bare spend and about twice a Lightning force close; the proof is the price of the shared output. An on-chain update is 1,030 vB, so payments must be off-chain or batched. Renewing a lease costs 2,616 vB alone or 151 vB a member in a signed cohort rewrite. Everything serialises through one output, at about 25 single exits a block even when they cooperate, which bounds how many time-sensitive leaves one tree should host; fee pinning on that output is outside the model. The sparse profile costs 1.7 to 2.8 times the dense one and cannot graft or sweep. The MID tier caps a tree at 2^{31} satoshis. And the opcodes do not exist: every tier needs proposals that are not active, three of them in a tapleaf version their proposals do not define (Table 2).

D.10 How the Numbers Are Made

Every figure is a whole transaction on one model, the same in spec 11, `spec/fee-analysis.md`, the evaluation of §10 and here: the measured script and witness and a 129-byte control block (the eight-leaf reference taptree), plus a 946 WU skeleton (the tree input, a P2WPKH fee input, the tree output, one P2TR payout and a P2WPKH change output) and 172 WU per further payout, with $vB = \lceil WU/4 \rceil$ once per transaction. A transition without a payout (JOIN, UPDATE, CLAIM, SPLIT, a rewrite that only updates) is priced on 774 WU, and the fee-from-payout exit on 550 WU. The model leaves out the witness item count and the per-item length prefixes; `tools/measure_weight.py` serialises the transactions the suites build and measures them (`spec/scripts/weights.md`): 10.5 vB on a FULL EXIT₁ at $D = 16$, 10 in CCV, and 42.5 and 44.75 in MID and MIN, whose witnesses carry each sibling as several small elements. The sizes come from `tools/gen_listings.py` (`spec/scripts/sizes.md`), the fee tables from `tools/fee_analysis.py`, and the figures of this appendix from `tools/gen_cost_figures.py`; the baselines are models, not measurements. The suites behind the sizes run 1,610 FULL, 1,391 MID, 57 MIN, 4,295 CCV and

220 channel-leaf checks, and 133 for the hypothetical opcode, which stays out of every tier.

E Implementer's Notes

The body says what each operation does and who signs it. This appendix collects what a wallet, a tower or an operator also has to get right: how a tree is created and recognised, what each party stores and rolls back, the Lightning receive as a message sequence, the fee reserve every unilateral path needs, mempool behaviour, leases and channel leaves. Section 4 serves as the per-operation card (who authorises, what the witness proves, what the outputs must be); the witness order of each tapleaf is in spec 8, the builders and signers are in the Go package `txn/`, one per transition for every tier (`BuildExit`, `BuildClaim`, `BuildRewrite`, `SignVoucher` and the rest, each replayed through the simulator), and what a co-signer verifies before signing is spec 9.2 step 3 for joins, 9.5 for updates and 9.6 for vouchers.

E.1 Creating and Recognising a Tree

The operator fixes the tree's parameters: the profile, the depth D , the operator key, the dust floor `DUST_MIN`, the supported `JOINh` heights and `EXITK` sizes, the variants (the exact-payout and tower exits, `UPDATEx`, `CLAIMx`, `REWRITEh` or `REWRITEhx`, `DETACHh`) and `tree_id`. It compiles the taptree from them and publishes the parameters with its root m (spec 6.2, 9.1). The recommended `tree_id` is a tagged hash of the outpoint spent by input 0 of the creation transaction, which is known before that transaction is built and can be spent only once, so it is unique per lineage; a chosen constant is not, and the tree output's own outpoint cannot serve, since m depends on `tree_id`. The creator therefore fixes input 0 before compiling the scripts. A zero-value output is invalid, so the operator seeds the tree with a leaf of its own, or with initial leaves whose owners have checked their inclusion before funding (spec 6.5).

Before treating an output as a LayerTree, a member, a voucher holder and a tower each recompute m from the published parameters and check `TreeSPK( $r$ ,  $m$ )` against the output for the root r they hold a path under. The scripts are visible only through m , so an operator that adds one tapleaf of its own (say, "the operator key spends the output") produces an output indistinguishable on-chain from an honest one; the recomputation, and rejecting any taptree with a leaf outside the expected set, is the check that custody rests on. A credit-leaf recipient also confirms that no plain `REWRITEh` is present, and a voucher holder that the voucher's `tree_id` is the one the `CLAIM` scripts carry.

E.2 State, Backups and Reorganisations

Table 15 lists what each party keeps. Every transition's witness publishes the paths it used, so a party holding a path for root r_k derives its path for r_{k+1} from

the witness of that spend alone (spec 9.7); a light client finds the spend with BIP-158 filters, which match the spent output’s scriptPubKey. The one piece of data that is not on-chain is a grafted member’s path inside its subtree, which the operator sends at the join and which a member who loses it can recover only from the operator or a cohort peer, so it is the backup that matters (spec 10.9).

Table 15. What each party stores. “Derives” is what it updates from each spend of the tree output; on a reorganisation every row is rolled back to the surviving (txid, r) .

party	stores	must back up	derives from each spend
Member	its leaf and path; r , m , <code>tree_id</code> and the published parameters; the txids of its join and of the latest spend	the leaf, and its path inside a grafted subtree	its new path, from the witness
Voucher holder	the latest voucher, ℓ_{src} and ℓ_{claim} ; the source’s path; a full-balance voucher or an empty position’s path; the source’s evictor and lease	the voucher and both leaves	the source’s path and <code>seq</code> ; whether the lease still outlasts a redemption
Tower	the <code>EXIT^t</code> authorisation, the leaf, its path and m ; a funded fee coin; for a channel leaf, $(\text{update}_n, \text{settle}_n)$ with their signatures (spec C.9.3)	the authorisation and the latest state	the leaf’s path; whether a bound field of the leaf changed
Operator	the whole tree: every leaf and path, cohort leases, the vouchers it issued as liabilities	the whole tree	every node; which leases have passed

A holder who is not a member has no join-time path to start from: it follows the tree from creation or asks the operator for the source’s path and checks it against the confirmed root. The operator may be the party it later falls back against; it can refuse the path but not falsify it, since a path either folds to the root or does not.

Reorganisations. Vouchers, paths and roots are keyed by (txid, r) and roll back with the chain (spec 10.11). A transition reorganised out is rebuilt against the surviving root and rebroadcast; a pending exit is rebuilt the same way after any competing spend. A JOIN reorganised out leaves each funder’s input its own. A reorganised CLAIM^h puts the swap back at the issuer’s discretion, so the holder rebroadcasts it at once. A member paying off-chain for a grafted leaf waits for its usual confirmation depth before paying (spec 9.2).

Messages. Spec 9 leaves message encoding out of scope, but the Go implementation carries TLV codecs, on lnd’s `tlv` package, for the join offer (`msg/join.go`), vouchers (`msg/voucher.go`), proofs (`msg/proof.go`) and every channel-leaf message (`msg/chan.go`: open, state proposals and signatures, HTLC add, fulfil and fail, rebind and splice).

E.3 Receiving over Lightning, Step by Step

The flow of §5.3, in any tier but MIN, with what goes wrong at each step:

(1) *Setup*. The operator grafts a credit leaf that it owns, with the recipient's key as exit key, the operator as evictor and a finite lease. The recipient checks m (Appendix E.1) and its path. If the operator never grafts it, nothing has been paid yet. (2) *Invoice*. The operator draws the preimage and issues the invoice, so the payer's receipt proves that the operator was paid. A recipient that wants the receipt to name it draws preimages and hands them over in advance, and can then be paid offline only that many times. (3) *HTLC*. The payer's HTLC reaches the operator, which settles it by revealing the preimage. If it fails, no voucher is issued and the recipient is owed nothing. (4) *Voucher*. Only then does the operator sign a cumulative voucher for the new running total and send it, store and forward if the recipient is offline. A provider that settles and never issues takes this one payment; that is the recipient's whole exposure. No order of the steps removes it. A voucher is a cheque against value that stays in the operator's leaf, so a hash lock orders the exchange without securing it, and the recipient-preimage order moves the risk to the operator with no time limit, since a credit leaf gives it no way to retire a voucher. A voucher that expires is not available either: Bitcoin enforces only lower bounds on time, so the most a script can give is the HTLC shape, an operator void branch after a deadline, and the deadline would have to sit in the leaf. Per payment that is a chain write per payment; shared across payments it lets the operator void settled vouchers too, unless the recipient redeems before each deadline. What closes the gap is a conditional claim in the tree, which step (8) describes, or a channel leaf and an online recipient. (5) *Top-up*. Before the running total exceeds the credit leaf's amount, the operator raises the amount by an amount-only REWRITE_h^x , about 122 vB per credit leaf batched per cohort (a changed position without its 16 vB member signature), or pre-funds the leaf. A top-up voids no voucher, since the message does not bind the amount (§5.1). Pre-funded capital beyond what has been vouched for is the recipient's on every path the operator can take alone, and comes back only with the recipient's consent. (6) *Retirement*. The operator pays the recipient another way, for instance with a leaf of its own in the next graft, then advances the credit leaf's seq and lowers its amount by one UPDATE^x that the recipient co-signs, or one REWRITE_h^x for a cohort of credit leaves. (7) *Fallback*. If the operator stops cooperating, the recipient redeems its latest voucher with CLAIM^x and leaves with the exact exit, paying both fees from a confirmed coin of its own (Appendix E.4). (8) *Atomic variant*. A tree that names a reclaim key and carries the merge claim and the reclaim-form rewrite (spec 6.2, 7.5.4, 7.9 rule 8, 7.11; FULL, MID and CCV) replaces steps (1) to (4) with a receive pair: a pending leaf P that the operator owns, with the recipient's key as exit key and the reclaim key as evictor, and the recipient's own landing leaf L as its sibling. For each payment the recipient draws the preimage and the operator signs a hash-locked merge voucher against P whose target is L 's settled level plus the payment. The recipient checks it against the confirmed tree and only then releases the preimage, so the operator has signed nothing it can lose before the reveal

and the recipient holds an enforceable claim after it. Before the pending leaf's deadline T the operator raises L to the settled level, by redeeming a voucher it signs itself or in one batch rewrite for every recipient of the window, which needs no recipient signature because it can only pay the recipient; if it does not, the recipient's tower redeems the latest voucher with the preimage, holding no key of the recipient's. After T the operator reclaims P , and only once the reclaim has confirmed does it fail an inbound HTLC whose voucher is outstanding. The recipient accepts a voucher only if L 's lease outlasts T plus the confirmation bound, since evicting L voids its vouchers.

E.4 Fee Reserves and Break-Even Balances

Only FULL's fee-from-payout exit can pay its fee out of tree value. A carve (CLAIM, CLAIM^h, CLAIM^x, SPLIT) conserves the tree's total; the exact-payout exit, which a credit-leaf tree, every channel leaf and the MID, MIN and CCV tiers use, pays the whole amount; and a channel leaf's update and settlement are zero-fee and are bumped only by an anchor child (spec C.5.5). So a member, a voucher holder and each party to a channel leaf keep a confirmed fee coin as a condition of holding the position, and a channel-leaf party's reserve grows with the HTLC value it has in flight, since a counterparty can delay the settlement at about 313 vB a block. A user with no on-chain coin at all has no unilateral recourse except, in FULL, the fee-from-payout exit of an ordinary leaf. Table 16 gives each path's fee at three feerates: a balance smaller than the fee is not worth recovering alone, and ten times the fee is the balance at which the path costs 10% (spec 11.2).

Table 16. The fee of each unilateral path at $D = 16$, FULL, on the transaction model, against the balance it recovers.

path	fee coin	vB	2 sat/vB	10 sat/vB	50 sat/vB
EXIT ₁ , fee from the payout	no	921	1,842	9,210	46,050
EXIT ₁ or EXIT ₁ ⁼ , fee input	yes	1,020	2,040	10,200	51,000
channel-leaf force close	yes	1,681	3,362	16,810	84,050
CLAIM ^x then EXIT ₁ ⁼	yes	2,656	5,312	26,560	132,800

E.5 Mempool Behaviour

An exit is bumped by replacement, which a plain exit's owner must re-sign and an EXIT^t holder rebuilds with its own fee input, or by a child spending its change or a pay-to-anchor output, which any transition may carry at an index past its payouts (spec 4.7). Version-3 (TRUC) transactions are not an option for transitions: TRUC caps an unconfirmed chain at two, and the chain of spends of the tree output is the throughput (spec 10.5). When a competing transition confirms first, the pending one is invalid; the wallet updates its path from the

winner’s witness and rebuilds. A graft with several funders cannot be replaced without every funder signing again, so it carries an operator change output or an anchor and is bumped by a child (spec 9.2).

E.6 Leases

A lease is what lets the operator sweep a cohort at 158 vB a leaf, and it has a recurring price. Leases are aligned per cohort, since one later lease holds up a sweep. A renewal moves a member into a cohort with the new lease, by an UPDATE and a whole-balance SPLIT (2,616 vB), or renews a cohort in place with one REWRITE₆ at 151 vB per member, which needs every renewing member’s signature; a member who does not sign keeps the old lease; a sweep needs every lease in the cohort to have passed, so the operator evicts it alone or moves it when its lease passes. So a member pays between 151 and 2,616 vB per lease period it stays, and the operator chooses the period. When a lease passes, the operator should warn the member and offer renewal (spec 9.4); if it evicts instead, the member sees its whole balance arrive at its exit key, at the operator’s fee.

E.7 Channel Leaves

A channel leaf is a new channel type, not a BOLT channel: LN-Symmetry updates signed with OP_TX and CSFS so that they rebind to whatever funding output the exit creates. Its funding output does not exist until the leaf leaves the tree, so there is nothing to announce to the gossip network; the leaf is reached through its counterparty, normally the operator. Its tower holds ($\text{update}_n, \text{settle}_n$), their signatures and a funded fee coin, and, in a tree with the tower exit, the leaf’s EXIT^t authorisation, which lets it start a close as well as finish one (spec C.9.3). A splice is one UPDATE^x (1,048 vB) that replaces the leaf’s amount and exit key, with no new funding output and no period in which two versions of the channel are live (spec C.5.2); it voids the tower’s authorisation, so the owner hands over a fresh one with the first state of the new epoch.

Deadlines and hub size. Settlement outputs include one per in-flight hashed time-locked contract (HTLC). A forwarding node keeps `cltv_expiry_delta` blocks between an incoming and an outgoing HTLC’s expiry, and a channel leaf needs $\text{cltv_expiry_delta} \geq G_{\text{exit}} + 1 + \beta_{\text{chan}} + 1 + \text{csv_delay} + 1 + G_{\text{sweep}}$, with `csv_delay` the settlement’s relative delay, G_{sweep} the time to confirm an HTLC sweep, G_{exit} the time to confirm the exit, $\Delta(q)$ under [C], and β_{chan} the stale updates the counterparty confirms first, a block each at about 313 vB a block (spec C.4.4). A leaf that receives sets it as the invoice’s `min_final_cltv_expiry_delta`; for one that pays it is the hub’s delta on the leaf-facing hop. With `csv_delay` = 144, G_{sweep} = 2, G_{exit} = 72 and β_{chan} = 0 this is about 220 blocks, against lnd’s default of 80 [18]. At $\Delta_L = 1$, $G_{\text{exit}} = 72$ needs $\beta + q \leq 71\kappa$; with $\kappa = 25$ and $\beta = 0$, about 1,775 channel leaves may close in one window, and larger hubs shard. The counterparty is normally the operator, which can issue competing spends, so the budget holds only while the member can outbid them, which is

what β and β_{chan} assume; both parties keep a confirmed fee coin and a reserve that grows with the HTLC value in flight (spec C.5.5).