

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

Olaoluwa Osuntokun

Lightning Labs

`laolu@lightning.engineering`

2026-10-15

DRAFT (commit `ba8422e`)

Full version

Abstract. In LAYERTREE 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 32 virtual bytes per level, 1,020 at depth 16). In transaction-tree constructions a user who leaves publishes $O(\log N)$ transactions, the interior must be pre-signed or template-committed, and one unresponsive user forces the whole tree on-chain at expiry. Here the output commits to an off-chain Merkle-sum tree of balances and tapscript covenants enforce every state transition: the operator evicts an expired lease in one transaction too, adding members needs only its signature on the covenant (self-funders still sign their inputs), a cohort is added or swept in one transaction, and the shared output never expires.

We state what the shared output does as an ideal functionality, give an abstract protocol whose covenant predicates realise it, and compile the predicates to four tiers of opcodes already proposed for Bitcoin, some assumed in a tapleaf version their proposals do not define; the smallest tier needs `OP_CAT` and one point-tweak opcode, recovering introspection from a generator-key signature. We prove that the abstract protocol UC-realises the functionality in a global-ledger hybrid model, under a stated assumption on how many competing spends of the shared output can confirm ahead of an honest one, with the tree-level lemmas mechanised in Lean 4. Lightning channels compose with the functionality as leaves whose exit key is shared, so a hub sized by exit throughput occupies one output. A simulator, a byte-exact Go implementation and 3,019 executable checks accompany the design.

Keywords: Bitcoin · covenants · Merkle-sum trees · universal composability · 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 [39] share an output between two parties; most approaches for *groups* pay for the sharing when it is dissolved. In channel factories [8], Ark [23,40], timeout trees [24,53] and covenant trees built on `OP_CHECKTEMPLATEVERIFY` (CTV) [41] or `OP_CHECKCONTRACTVERIFY` [20,19], 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` [47], CoinPool’s `OP_MERKLESUB` [29], `OP_EVICT` [52]) 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.

We present the design from the interface inwards: an ideal functionality in the universal composability (UC) framework [9] (§3); an abstract protocol of covenant *predicates* and honest-party behaviour, independent of opcodes (§4); and four realisations by opcode tier (§5), from restored string opcodes (BIPs 440 and 441 [44,45]), introspection (`OP_TX` [43]), stack signature checks (`OP_CHECKSIGFROMSTACK` [6]) and elliptic-curve arithmetic [34,42] down to `OP_CAT` and one point-tweak opcode, plus one built on `OP_CHECKCONTRACTVERIFY` [20]. We propose no opcode for this design, but the tiers are not deployable as proposals stand: FULL, MID and both CCV variants assume opcodes in a tapleaf version their proposals do not define, and two drafts claim the same opcode numbers (§5). The elliptic-curve draft [34] is by the author of this paper. The separation lets the security argument be made once, for the predicates.

Contributions. (1) $\mathcal{F}_{LT}$, with interfaces for exit, eviction, grafting, update, vouchers and sweeps, and liveness under an explicit contention assumption. (2) An abstract protocol built from one *step* predicate over a Merkle-sum tree. (3) Every predicate compiled in three tiers, and the exit in a fourth (MIN) that rebuilds the BIP-341 message in script and checks it with Poelstra’s generator-key signature [37]; an upper bound on the opcodes needed. (4) A UC proof in a global-ledger hybrid model with the tree-level lemmas mechanised in Lean 4, which isolated that sums are conserved unconditionally but exits are guaranteed only for members who verified their path. (5) Lightning channels as leaves, and the conditions under which they compose. (6) A simulator, a byte-exact Go implementation, and measured scripts with their modelled transaction costs: a 1,020 vB exit at depth 16, 665 vB for a second exit in the same transaction, 158

to 203 vB per swept leaf. (7) For scale, a hypothetical native Merkle-sum opcode, measured in the simulator and proposed by no one, would make the exit grow by 10 vB per doubling of the membership (§10). The specification, code and proofs are public [35]; this text describes revision ba8422e.

This version. A short version for developers walks through the operations, vouchers and trust model and only states what is proved. This full version is built around $\mathcal{F}_{\text{LT}}$: it carries the UC proof (§7) and the channel-leaf scripts (§9), the Lean statements inline, sections on costs against the alternatives (§11), the node format and radix (§12), batch shapes (§13), grafting (§14) and receiving over Lightning (§15), and the Lean development and script listings (Appendices A and D).

2 Background

Bitcoin. A pay-to-taproot output [49] 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 [48] under Q or by revealing one script with its Merkle path (the *control block*) and satisfying it. Tapscript [50] has no loops, so every “loop” below is unrolled. BIP-441 [45] 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 [44] (“varops”); BIP-347 [18] proposes OP_CAT alone in the current version, 0xC0. OP_TX [43] pushes fields of the spending transaction, optionally collated into one hash preimage; OP_CHECKSIGFROMSTACK (CSFS) [6] verifies a signature over a stack element; OP_TWEAKADD [42] computes $x(P + tG)$, and the EC-ops draft [34] 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 [27,33] hashes each node’s children with their sums, so an inclusion proof proves a leaf’s contribution to the root sum.

Composable security. We use the UC framework [9] with global subroutines [11]. A protocol Π *UC-realises* a functionality $\mathcal{F}$ if for every adversary $\mathcal{A}$ there is a simulator $\mathcal{S}$ such that no environment $\mathcal{Z}$ distinguishes Π with $\mathcal{A}$ from $\mathcal{F}$ with $\mathcal{S}$. Following Badertscher et al. [3] the chain is a global ledger $\mathcal{G}_{\text{ledger}}$ that accepts transactions satisfying a validity predicate, orders them at the adversary’s discretion within a liveness bound Δ_L , and never reverts them; time is a global clock $\mathcal{G}_{\text{clock}}$ [22] whose rounds are blocks; signatures are the ideal functionality $\mathcal{F}_{\text{sig}}$ [10]; channels follow [2,17].

3 The LayerTree Functionality

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,

Functionality $\mathcal{F}_{\text{LT}}[D, \text{profile}, pk_{\text{op}}, \text{exact}, \Delta]$

State. A *partial* tree T over 2^D slots. A node is *known* (empty, a leaf $\ell = (a, pk_{\text{exit}}, pk_{\text{own}}, pk_{\text{ev}}, h_{\text{ev}}, \text{nonce}, \text{seq})$, or an inner node with its two children) or *opaque* (h, s), a hash and a sum never opened; known nodes may sit below an opaque one where a path has been shown. A is the root's sum; V is the set of outstanding vouchers. Every call is public (caller, arguments, effect). $\mathcal{A}$ decides when, within $\Delta(q)$ rounds of $\mathcal{G}_{\text{clock}}$, an accepted honest call takes effect, q being the honest calls queued ahead of it; effects are atomic and ordered.

On CREATE(T_0) **from** the operator, once: after $\sum T_0$ coins are deposited on $\mathcal{G}_{\text{ledger}}$, set $T := T_0$, or the opaque root of T_0 if $\mathcal{A}$ withholds it.

On JOIN(p, h, u) **from** the operator: require node p at height h empty and **profile** to admit p ; require $\sum u$ coins deposited; set $T[p] := u$, a tree of height h , or its opaque root if $\mathcal{A}$ withholds it.

On REVEAL(p, n_L, n_R) **from** $\mathcal{A}$: require $T[p]$ opaque and equal to the parent of n_L and n_R ; replace it by an inner node with children n_L, n_R (opaque, or leaves at height 1).

On EXIT(p, v, f) **from** $\mathcal{P}$: require $T[p] = \ell$ known, $\mathcal{P}$ holding pk_{own} , f a coin of $\mathcal{P}$, and $v = a$ if exact else $v \leq a$; set $T[p] := \text{empty}$; post the transaction of Fig. 2, funded by f , paying v to pk_{exit} .

On EVICT(p, f) **from** $\mathcal{P}$: as EXIT with pk_{ev} in place of pk_{own} , $\mathcal{G}_{\text{clock}} \geq h_{\text{ev}}$, and $v = a$.

On UPDATE(p, ℓ') **from** the owner of $T[p] = \ell$ and the operator jointly (and the holder of $\ell.pk_{\text{exit}}$ in variant UPDATE^{*}): set $T[p] := \ell'$; require $\ell'.a - \ell.a$ coins deposited if positive, else pay the difference to the callers.

On VOUCH(p, ℓ_c) **from** the owner of $T[p] = \ell$: require $\ell_c.a \leq \ell.a$, $\ell_c.pk_{\text{ev}} = \ell.pk_{\text{ev}}$ and $\ell_c.h_{\text{ev}} \leq \ell.h_{\text{ev}}$; add (p, ℓ, ℓ_c) to V and return it to the caller.

On CLAIM(p, ℓ_c, q) **from** any party holding $(p, \ell, \ell_c) \in V$ with $T[p]$ equal to ℓ in every field but a and $\ell_c.a \leq T[p].a$: set $T[p] := \text{empty}$ if $\ell_c.a = T[p].a$, else $T[p]$ with amount $T[p].a - \ell_c.a$ and $\text{seq} + 1$; then require $T[q]$ empty and set $T[q] := \ell_c$. (SPLIT is VOUCH and CLAIM by the owner at once.)

On SWEEP(p, h) **from** $\mathcal{P}$: require every leaf below node p (height h) known and naming $\mathcal{P}$ as evictor with $\mathcal{G}_{\text{clock}} \geq h_{\text{ev}}$; set the subtree empty; pay each leaf's a to its pk_{exit} .

On READ **from** any party: return T and A .

Fig. 1. The ideal functionality. Batched exits (EXIT_K) are K EXIT calls taking effect together. $\mathcal{A}$ may refuse to serve JOIN (there is no liveness for joining), schedules accepted calls within $\Delta(q)$, and may graft opaque subtrees and later reveal them; value under an opaque node that is never revealed is frozen.

an exit key paid on exit, and an evictor key, normally the operator's, that may remove the leaf after a lease height). The operator holds no key over the funds, only an *operator key* that authorises additions and co-signs cooperative changes. Figure 1 gives $\mathcal{F}_{\text{LT}}$; its parameters are the depth D , the addressing *profile* (dense, used throughout: the operator assigns slots; Appendix C.2 describes the sparse alternative), the operator key, the payout rule *exact* (whether an owner may leave part of its balance as fee), and the liveness bound Δ below. $\mathcal{F}_{\text{LT}}$ is protocol-aware, as in Generalized Channels [2]: it builds an honest party's transaction from a fee coin the caller names and posts it.

What the functionality promises. (G1) *Conservation*: A is the sum of T and every payout is funded by a removed leaf. (G2) *Unilateral exit*: the owner of a known leaf is paid its balance with no other party's consent; no party can prevent the exit except by outbidding it, and under [C] below it takes effect within $\Delta(q)$. (G3) *Eviction*: after the lease height the evictor can do the same, and since a carve keeps the evictor and a lease no later, this holds per balance. (G4) *Operator-only membership*: JOIN and SWEEP need the operator and no member. (G5) *No expiry of the shared output*, though each leaf has a lease. Not promised: liveness for joining, and anything for value under an opaque node never

revealed (frozen, not stolen); honest members verify their path before funding (§4.4), so their leaves are known.

The liveness bound. Every call spends and recreates one ledger output, and an honest call can be overtaken by any number of valid conflicting spends (a corrupt member alone can carve its balance back and forth), so the delay does not follow from the ledger model; we state it under an assumption whose parameters we do not derive. *[C] (contention):* at most $\kappa \geq 1$ spends of a tree output confirm per block, and while an honest transition is pending at the prevailing fee rate 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 [4,13], and as low as 1 once anyone hangs descendants off a payout or change output. β is economic, the adversary’s fee budget against the honest bid; unbounded, it makes exit a fee auction. Under [C] an honest call with q honest calls queued ahead takes effect within $\Delta(q) = \Delta_L(\lceil(\beta + q)/\kappa\rceil + 1)$ rounds, Δ_L being the ledger’s liveness parameter. We use $\kappa = 25$, since a deployment assumes the lowest limit its relay peers enforce (spec 10.5). Cooperatively ($\beta = 0$, $\Delta_L = 1$ block) a tree then drains 25 exits per block, or 64 once peers run cluster mempool, so 10,000 channel leaves closing at once take 400 blocks (157 at 64): trees are sized by exit throughput, not depth.

4 The Abstract Protocol

The protocol Π has a data structure whose root one output commits to, one *predicate* per interface, enforced by the ledger, and the behaviour of honest parties.

4.1 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

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

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 (Appendix C.3). 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 (Lemma 1). 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 (Theorem 1).

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 C.2.

4.2 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). 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.

4.3 The Step and the Transition Predicates

Every transition spends the tree output and recreates it as output 0 with the new root and total; payouts occupy outputs $1, \dots, K$; other inputs and outputs are free, for fees, change and anchors. The core is the *step*: $\text{fold}(n_{\text{old}}, \pi) = (r, S)$ with S the input's amount (CHECK_OLD), and $(r', S') = \text{fold}(n_{\text{new}}, \pi)$. A predicate Pred_X is a sequence of steps, each proving its old node against the previous step's root (CHECK_PREV), an authorisation, and a check that output 0 is $(S', \text{TreeSPK}(r', m))$ unless the tree is now empty (FINAL). Table 1 gives each interface's predicate, Figure 3 shows how they compose, and the boxes that follow it state each predicate exactly; Appendix B gives the step in pseudocode and tapscript.

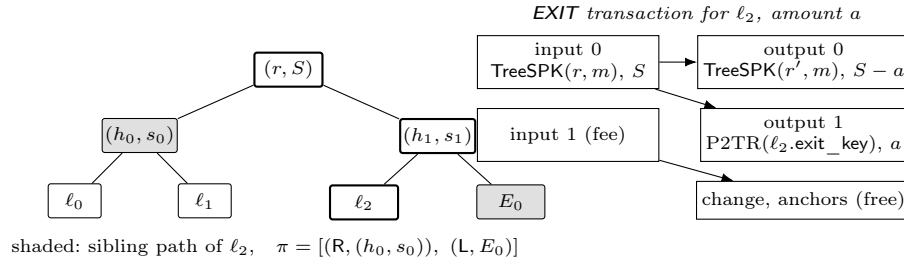


Fig. 2. A depth-2 tree and the exit of ℓ_2 . The predicate checks $\text{fold}((H_{\text{leaf}}(\ell_2), a), \pi) = (r, S)$, computes $r' = \text{fold}(E_0, \pi)$, verifies a signature by $\ell_2.\text{owner_key}$, and constrains outputs 0 and 1. Everything else in the transaction is free.

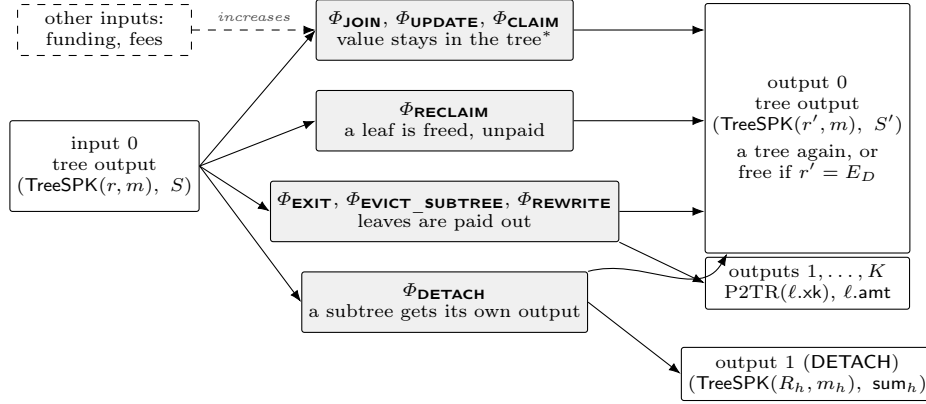


Fig. 3. How the predicates compose. Every transition spends the tree output, input 0, and recreates it as output 0 under the new root; payouts are outputs $1, \dots, K$ and every other input and output is free. *UPDATE's decrease is free. DETACH writes a second tree output, itself the tree output of a smaller tree under the same rules and `tree_id`.

Signatures must cover all outputs, or a relay could redirect the fee slack of a non-exact exit. JOIN_h grafts a whole subtree (with $h = 8$, 256 members) for a constant script cost, but cannot look inside the grafted node: this is where opaque nodes come from. EVICT_SUBTREE_h is a shape-specialised multiproof, one path to the subtree and its 2^h leaves hashed once. VOUCH is an off-chain signature by the source leaf's owner over the source leaf without its amount (spec 0.13.0), the claim leaf and the tree's lineage identifier `tree_id` (fixed at creation and inherited by detached cohorts, so a voucher verifies in no other tree), and every carve gives the claim leaf the source's evictor key and a lease no later than the source's, so moving a balance cannot escape eviction; both are spec 0.10.0 rules in every tier (Appendix C.3). FULL also builds CLAIM^x (the exit key consents to a redemption), REWRITE_h (a subtree's published leaves

Table 1. Transition predicates. Each row is one script leaf under m ; σ_k is a transaction signature under key k covering all outputs; steps are written old node $\rightarrow$ new node.

Interface (predicate)	Steps	Authorisation and outputs
EXIT (EXIT)	$(H_{\text{leaf}}(\ell), a) \rightarrow E_0$	σ_{own} ; out ₁ pays <code>exit_key</code> , $v \leq a$ ($v = a$ if exact)
EVICT (EXIT, flag)	same	σ_{ev} , lock time $\geq$ <code>evict_after</code> ; out ₁ pays exactly a
$K \times$ EXIT (EXIT _{K})	K leaf steps, chained	each step its own σ ; out _{1..K}
JOIN (JOIN _{h})	$E_h \rightarrow (h_{\text{new}}, s_{\text{new}})$	σ_{op} ; out ₀ = $S + s_{\text{new}}$, funded by other inputs
$K \times$ JOIN (JOIN _{K})	K steps $E_0 \rightarrow$ leaf	one σ_{op}
UPDATE (UPDATE)	$(H_{\text{leaf}}(\ell), a) \rightarrow (H_{\text{leaf}}(\ell'), a')$	$\sigma_{\text{own}}, \sigma_{\text{op}}$, plus σ_{exit} in <code>UPDATE</code> ^{x} ; out ₀ = $S - a + a'$
CLAIM (CLAIM)	source $\rightarrow$ source'; $E_0 \rightarrow \ell_c$	voucher signature checked from the stack; ℓ_c keeps the evictor, lease no later; out ₀ = S
CLAIM (CLAIM ^{h})	same	as above, over a message naming a payment hash, with the preimage in the witness
CLAIM (SPLIT)	same	σ_{own} of the source leaf; no stack signature, so no <code>OP_CHECKSIGFROMSTACK</code>
SWEEP (EVICT_SUBTREE _{h})	subtree, post-order $\rightarrow E_h$	σ_{ev} , every lease expired; one payout per leaf

replaced at once, each emptied leaf paid exactly to its exit key) and DETACH _{h} (a subtree moved to an output of its own).

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 `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.

4.4 Honest Parties

The honest parties are given as algorithms: the operator's (Algorithms 1, 2, 9 and 10), a member's (3, 4 and 5), the issuer's and the recipient's of a voucher (6, 7 and 8), and those of the receive protocol of §15 (11 to 14). Each is followed by a paragraph that walks its lines against the spec rule it implements and says where the spec is silent.

A member stores its leaf and path and follows every spend of the tree output; each witness contains the path it used, so the member updates its own path from the chain alone. Before funding a JOIN or a creation, it verifies its path against the grafted node (or r_0) and the deposit; one who skips the check may end up under an opaque node. A self-funded graft therefore needs every funder to sign one transaction after the cohort is fixed, an abort-and-retry round like

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 $\text{Csfs}_{\ell_i.\text{ok}}(\sigma_i, H_{\text{exit}}(\text{tree_id} \parallel \hat{\ell}_i))$ – 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]

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 3).

an Ark round or a coinjoin; in an operator-funded graft the operator fronts the value and members pay it off-chain. To exit, a member builds the transaction of Figure 2 with a fee input. Its signature commits to the tree outpost, which every transition changes, so a plain exit cannot be pre-signed or delegated: the owner key must be online to rebuild it after a competing spend and to bump its fee. The tower variant EXIT_K^t (FULL, MID, CCV) replaces that signature by one over the leaf without its amount and the tree's identifier, checked from the

Φ_{UPDATE} – replaces one leaf by another at the same position, with the owner and the operator

Witness a path π of D levels; leaves ℓ, ℓ' ; signatures σ_o, σ_p (and σ_x under x); r, m .

Constants D ; OPERATOR_KEY; the flag x (exit-key consent, UPDATE^x).

1. $\text{in}_0 = (\text{TreeSPK}(r, m), S)$ – the spent output is the tree output [7.1 r1, r3]
2. $\text{fold}(\text{nd}(\ell), \pi) = (r, S)$ and $(r', S') = \text{fold}(\text{nd}(\ell'), \pi)$ – the new leaf takes the old one's place [7.4 r1]
3. $\text{Sig}_{\ell.\text{ok}}(\sigma_o)$ and $\text{Sig}_{\text{OPERATOR_KEY}}(\sigma_p)$ – the owner and the operator both consent [7.4 r2]
4. under x : $\text{Sig}_{\ell.\text{xk}}(\sigma_x)$ – a shared exit key cannot be spliced down without consent [7.4 r5]
5. $\text{out}_0 = (S - \ell.\text{amt} + \ell'.\text{amt}, \text{TreeSPK}(r', m))$ – a decrease is free to spend; an increase is funded elsewhere [7.4 r3]
6. sparse profile: $H_{\text{pos}}(\ell'.\text{ok} \parallel \ell'.\text{nonce}) = H_{\text{pos}}(\ell.\text{ok} \parallel \ell.\text{nonce})$ – the leaf stays at its derived position [7.4 r4]

stack, so a tower holding it builds, fees and rebuilds the exit while the owner is offline. Transitions are final at the member's confirmation depth. The operator keeps the full tree, assigns slots, signs joins and updates, and evicts or sweeps expired leaves.

Algorithm 1 Create – operator, founding a tree

- 1: **choose** D , OPERATOR_KEY and the option set $\mathcal{O}$ (heights, sizes, variants)
 - 2: **choose** the outpoint o_0 that input 0 of the funding transaction will spend
 - 3: $\text{tree_id} \leftarrow H_{\text{tree-id}}(o_0)$
 - 4: $m \leftarrow$ Merkle root of the taptree for $(D, \text{OPERATOR_KEY}, \text{tree_id}, \mathcal{O})$
 - 5: **build** the initial leaves, at least one, and their root r_0
 - 6: **publish** $(D, \text{OPERATOR_KEY}, \text{tree_id}, \mathcal{O}, m, r_0)$ and each member's leaf and path
 - 7: **after** every member has run **AcceptLeaf**:
 - 8: spend o_0 at input 0; output 0 $\leftarrow (\text{TreeSPK}(r_0, m), \sum r_0)$
-

Lines 2 to 4 fix the order that the circularity of spec 6.2 forces: the scripts of CLAIM carry tree_id , so m depends on it, and the tree output's own outpoint cannot be its source, since the output's script depends on m . The recommended derivation (line 3) hashes the outpoint spent by input 0 of the funding transaction, which exists before the transaction is built and can be spent once, so it is unique by construction; the spec requires only that it be unique per lineage. Line 6 publishes the parameters from which any party recomputes m (spec 9.1: a participant MUST recompute m before treating an output as a tree, which also catches a wrong tree_id). Line 7 funds the output only after the founding members have verified their paths (spec 6.5, 9.2), so each founder runs **AcceptLeaf** before any funds move. The spec leaves the transport of leaves and paths to founders open (spec 9, introduction), and says nothing of a timeout. Lines 1 to 4 build the

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: $\text{Csfs}_{\ell_s.\text{ok}}(\nu, H_{\text{voucher}}^{\text{tree_id} \parallel \hat{\ell}_s \parallel \ell_c})$; 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]

$\Phi_{\text{EVICT_SUBTREE}}$ – the operator sweeps every remaining leaf of one aligned subtree of height h

Witness a path π of $D - h$ levels; the 2^h elements $o_1, \dots, o_{2^h}$ in position order, each a leaf or $\perp$; a signature σ ; r, m .

Constants D ; h ; OPERATOR_KEY .

Verification let R be the root recomputed from the elements ($\perp$ contributes E_0), and v_j, k_j the payout and its output index for the j -th leaf, numbering from output 1.

1. $\text{in}_0 = (\text{TreeSPK}(r, m), S)$ – the spent output is the tree output [7.1 r1, r3]
2. $\text{fold}(R, \pi) = (r, S), (r', S') = \text{fold}(E_h, \pi)$ – the published subtree is in the tree, with each internal node hashed once; it becomes empty [7.7 r3, r4]
3. $\text{Sig}_{\text{OPERATOR_KEY}}(\sigma)$ – the operator authorises [7.7 r1]
4. for every leaf o_j : $o_j.\text{ek} = \text{OPERATOR_KEY}$, $\text{lock} \geq o_j.\text{ea}$ with input 0 enabling lock time, and $\text{out}_{k_j} = (o_j.\text{amt}, \text{P2TR}(o_j.\text{xk}))$ – every swept leaf is expired, names the operator as evictor, and is paid in full to its exit key; one leaf with another evictor blocks the sweep [7.7 r2]
5. $\text{out}_0 = (S - \sum_j o_j.\text{amt}, \text{TreeSPK}(r', m))$ unless $r' = E_D$ – output 0 recreates the tree output without the cohort [7.7 r4; 7.8]

Algorithm 2 Graft – operator, adding a cohort at height h

- 1: **pick** an empty position (h, i) with path π , at root r and sum S
 - 2: **build** the subtree of the new leaves; $\bar{n} = (\bar{h}, \bar{s}) \leftarrow$ its node
 - 3: $r' \leftarrow \text{fold}(\bar{n}, \pi)$
 - 4: tx: input 0 spends the tree output; output 0 = $(S + \bar{s}, \text{TreeSPK}(r', m))$
 - 5: **add** funding inputs (the operator's, or each member's own)
 - 6: **if** more than one funder: add a change output or a P2A anchor
 - 7: **for each** member U : **send** (tx, ℓ_U , path of ℓ_U in the subtree, π, r, m)
 - 8: **collect** the signatures of the members' inputs, if any
 - 9: **if** a member withdraws or does not sign: drop it, rebuild, go to line 2
 - 10: sign input 0 with OPERATOR_KEY ; **broadcast**
-

transaction that Φ_{JOIN} accepts: the slot is empty, output 0 carries $S + \bar{s}$ and the operator signs. The slot's height fixes the script, since each height h is a tapleaf fixed at creation (spec 6.2), and the operator chooses the slot that later batching depends on (design/graft.md). Line 5 covers the three funding modes: in an operator-funded graft the operator supplies $\bar{s}$ and is paid off-chain, so there is nothing to collect and no round (spec 9.2); in a member-funded one each member adds an input. Line 6 is the spec's advice for more than one funder: because each funder signs with SIGHASH_DEFAULT or SIGHASH_ALL , the graft cannot be replaced at a higher fee without every funder signing again, so the operator must be able to bump it by CPFP (spec 9.2). Line 7 sends each member what **AcceptLeaf** needs, and lines 8 and 9 are the abort and retry of a signing round: the output commits to the whole subtree, so one member who changes its leaf or never signs invalidates every other signature (spec 9.2, informative). The spec states the round but gives the operator no rule for when to give up on a member

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]

or how long to wait. Lines 2 to 6 are checks (a) to (d) of spec 9.2, and they are the checks the covenant does not make: Φ_{JOIN} sees a hash and a sum and cannot tell a cohort from random bytes. Line 2 shows that input 0 is the unspent tree output, line 3 that the slot was empty in the current root, line 4 that U ’s own leaf folds to the grafted node, with the subtree path of exactly h levels so that the leaf lands at depth D (a leaf grafted shallower sits where no D -level path can reach it), and lines 5 and 6 that output 0 commits to the root containing it. This is the check that Theorem 4 and the claim “an exit is guaranteed only to a member who verified its path” rest on. Lines 7 and 8 add the agreed fields and, from spec 9.2, the rule against re-joining a lineage with an owner key, nonce and seq held before, which would revive every exit authorisation ever published for that leaf. A member who funds an input is then protected by construction,

Φ_{DETACH} – moves one aligned subtree of height h into a tree output of its own

Witness a path π of $D - h$ levels; the subtree's hash R_h and sum sum_h ; a signature σ ; r , m .

Constants D ; h ; OPERATOR_KEY ; m_h , the taptree root of the tree of depth h compiled from the parent's option set (spec 7.10 rule 3).

1. $\text{in}_0 = (\text{TreeSPK}(r, m), S)$ – the spent output is the tree output [7.1 r1, r3]
2. $\text{fold}((R_h, \text{sum}_h), \pi) = (r, S)$ and $(r', S') = \text{fold}(E_h, \pi)$ – the subtree is in the tree, and leaves it [7.10 r1, r2]
3. $\text{Sig}_{\text{OPERATOR_KEY}}(\sigma)$ – the operator decides; it spends nobody's balance [7.10 r4]
4. $\text{out}_1 = (\text{sum}_h, \text{TreeSPK}(R_h, m_h))$ ($\text{TreeSPK}_{\text{CCV}}$ in the CCV tier) – the cohort keeps its rules, its tree_id and every leaf byte for byte, at depth h [7.10 r3]
5. $\text{out}_0 = (S - \text{sum}_h, \text{TreeSPK}(r', m))$ unless $r' = E_D$ – the rest of the tree is recreated; value is conserved across the two outputs [7.10 r2; 7.8]

Φ_{RECLAIM} – removes a reclaimable leaf after its deadline and pays nothing for it

Witness a path π of D levels; a leaf ℓ ; a signature σ ; r , m .

Constants D ; RECLAIM_KEY (present in the tree exactly when the tree has this script).

1. $\text{in}_0 = (\text{TreeSPK}(r, m), S)$ – the spent output is the tree output [7.11 r1, r3]
2. $\ell.\text{ek} = \text{RECLAIM_KEY}$ – the leaf is reclaimable [7.11 r1]
3. $\text{lock} \geq \ell.\text{ea}$ with input 0 enabling lock time – the deadline has passed [7.11 r2]
4. $\text{Sig}_{\text{RECLAIM_KEY}}(\sigma)$ – the operator's reclaim key signs [7.11 r3]
5. $\text{fold}(\text{nd}(\ell), \pi) = (r, S)$, $(r', S') = \text{fold}(E_0, \pi)$ and $\text{out}_0 = (S - \ell.\text{amt}, \text{TreeSPK}(r', m))$ unless $r' = E_D$, with no output constrained for the leaf's value – the value becomes free for the transaction [7.11 r4]

Algorithm 3 AcceptLeaf – joiner U , before funding or paying for a leaf

- 1: **input:** unsigned tx, ℓ_U , subtree path ρ , path π of the slot, r , m
 - 2: **if** 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** ℓ_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** out_0 is unspent
 - 11: **and** $\text{fold}(\text{nd}(\ell_U), \text{its path in } r') = (r', S')$; only then pay
-

since the only transaction that spends its input commits to a root containing its leaf (line 9). A member who pays off-chain has nothing to sign, so lines 2 to 6 protect nothing and the spec requires the check of lines 10 and 11 on the *confirmed* transaction before payment; paying earlier trusts the operator to graft. The alternative that the covenant enforces is a hash-locked voucher against a leaf the operator already holds (spec 9.2, Algorithm 8). Lines 2 to 4 are Φ_{EXIT}

Algorithm 4 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  $\tau$  under  $\text{EXIT}_K^t$ ; broadcast
6 :   wait until tx is buried or a spend  $\text{tx}'$  of  $o$  confirms:
7 :     if tx is pending and too cheap: bump by replacement or an anchor child
8 :     if  $\text{tx}'$  confirms:  $(\ell, \pi, r, o) \leftarrow \text{Follow}(\text{tx}')$ 
9 :     if  $\ell$  is gone: stop; else continue loop
```

for $K = 1$: the leaf is removed along its own path (check 2), output 1 pays the exit key (check 3) and output 0 recreates the tree output (check 7); line 4 is the payout rule, in which a plain exit may pay less than the balance and the exact variants may not. Line 5 signs all outputs, which is what stops a relay from diverting the shortfall or the fee slack (spec 7.2 rule 3). The signature commits to the tree outputpoint o , which every transition changes, so it cannot be made in advance: a plain exit needs the owner online to rebuild it (lines 1 to 5 again). Under EXIT^t the owner makes $\tau = \text{Sign}_{\ell.\text{ok}}(H_{\text{exit}}(\text{tree_id} \parallel \hat{\ell}))$ once and hands (ℓ, τ) to a tower, which then runs the whole algorithm and pays the fee from its own input (spec 7.2 rule 7, 9.3); the tower also tracks the leaf's bytes, not only its path, since a **REWRITE** top-up changes the amount the exit must pay without the owner (spec 9.7). Lines 6 to 8 reflect that every transition spends the one tree output, so an exit can be overtaken by any conflicting spend that confirms first and has to be rebuilt against the new root (spec 10.5), after **Follow** (Algorithm 5) has brought the path up to date; under [C] this takes at most $\Delta(q)$ blocks. The spec names replacement (BIP-125) and anchor outputs as the ways to bump a fee but does not fix a policy, and for a cooperative exit it advises asking the operator first, who can fold it into its next transition (spec 9.3). An owner who stays in the tree after an exit built on a published authorisation was outbid must change the leaf to void it (spec 7.2 rule 7). Line 2 uses the fact that every transition publishes the sibling paths it used and its old and new nodes (spec 9.7). Line 5 is the whole of path maintenance: of the siblings along a member's path, only the ancestor of the changed position at the level where the two paths diverge can change, and the witness computes its new value (Lemma 3, the Lean `path_maintenance`). Line 4 is the case where the leaf itself changes, which the member's own signature accounts for except where only the

Algorithm 5 Follow – member or tower, on each spend of the tree output

```

1 : input:  $(\ell, \pi, r, o)$ , and a transaction  $\text{tx}'$  that spends  $o$  at input 0
2 : read from  $\text{tx}'$ 's witness: changed positions, paths used, old and new nodes
3 :  $r' \leftarrow$  the root committed by  $\text{out}_0$  of  $\text{tx}'$  ( $E_D$  if the tree is gone)
4 : if the position of  $\ell$  changed:  $\ell \leftarrow$  the new leaf the witness shows (or none)
5 : else replace the one entry of  $\pi$  where the two paths diverge
6 :   by the node the witness computes at that height
7 : if  $\ell \neq \text{none}$  and  $\text{fold}(\text{nd}(\ell), \pi) \neq r'$ :
8 :   ask the operator for the path and check it against  $r'$ 
9 :  $(r, o) \leftarrow (r', (\text{txid}(\text{tx}'), 0))$ 

```

amount moves: a top-up, a merge (spec 7.5.4) or a reclaim-form transfer (7.9 rule 8), so a tower that holds an exit authorisation must track the leaf's bytes (spec 7.9 rule 3, 9.7). Line 7 is the fall-back the spec allows to a participant that has fallen behind: it can always catch up from the chain, or obtain the current path from the operator and verify it against the confirmed root. A light client cannot find the spends by scriptPubKey, since $\text{TreeSPK}(r, m)$ changes with every transition; it follows the outpoint chain (design/towers.md). The spec does not say how often a member must run the algorithm, only that it **MUST** process every spend (spec 9.7). Lines 3 and 5 are the issuer's **MUSTs** of spec 9.6: never

Algorithm 6 Vouch – issuer I of a voucher against its leaf ℓ_s

```

1 : on request for a voucher paying  $\ell_c$  (for  $\text{CLAIM}^h$ , with a payment hash  $\eta$ ):
2 :   if another leaf has the body of  $\ell_s$  (dense profile): refuse
3 :   if  $\sum_{\text{outstanding}} + \ell_c.\text{amt} > \ell_s.\text{amt}$ : refuse
4 :    $\nu \leftarrow \text{Sign}_{\ell_s.\text{ok}}(H_{\text{voucher}}(\text{tree\_id} \parallel \hat{\ell}_s \parallel \ell_c \parallel \eta))$ 
5 :   record  $(\nu, \ell_s.\text{seq})$  as a liability; send  $(\text{tree\_id}, \ell_s, \ell_c, \nu)$ 
6 :   on a transition that changes a field of  $\ell_s$  other than amt:
7 :     re-sign every outstanding voucher over the new  $\ell_s$  and the new amount
8 :   on a decrease of  $\ell_s.\text{amt}$  that leaves seq alone:
9 :     keep the outstanding sum within it, or advance seq and re-sign
10 : retire a liability once  $\ell_s.\text{seq}$  exceeds its value at issuance

```

issue vouchers whose amounts sum to more than the source holds, and track each as a liability. The covenant prevents over-*redemption* but not over-issuance (spec 10.3). Line 2 is the bookkeeping rule of the replay paragraph of spec 7.5: a voucher binds its source by content, not by position, so two leaves with one body would let one voucher redeem against both, and the dense profile does not check this on chain. Line 4 binds the voucher to **tree_id**, the source without its amount and the claim leaf in full, and for CLAIM^h to the payment hash under a tag of its own, so that a plain and a hash-locked voucher are disjoint instruments. Lines 6 to 9 follow from what the message binds. A **CLAIM** or **SPLIT** advances **seq**, an **UPDATE** or **REWRITE** may rotate a key, renew a lease or change **seq**,

and each of these voids every outstanding voucher, so the issuer re-signs them against the new leaf, which also lowers the balance they draw on. A change of the amount alone voids nothing: a top-up is harmless, and after a decrease the issuer must keep the outstanding sum within the reduced amount or advance `seq` and rebuild. Line 10 is how a voucher is retired, since the covenant gives it no deadline (spec 7.5.1). The spec does not tell the issuer to check that ℓ_c obeys the carve rule or the dust floor; a voucher that violates either simply fails at redemption. Lines

Algorithm 7 Redeem – recipient of a voucher ($\text{tree_id}, \ell_s, \ell_c, \nu$)

```

1 : on receipt: if  $\nu$  does not verify under  $\ell_s.\text{ok}$  over the voucher message, or
2 :    $\text{tree\_id}$  is not the value from which  $m$  was recomputed: reject
3 :   obtain a path for a leaf with the body of  $\ell_s$  in the confirmed tree;
4 :   if it fails or  $\ell_s.\text{amt} < \ell_c.\text{amt}$ : reject
5 :   read  $\ell_s.\text{ek}, \ell_s.\text{ea}$ ; if  $\ell_s.\text{ea}$  can pass before a redemption confirms ( $\Delta(q)$ ) and
6 :      $\ell_s.\text{ek} \notin \{\text{NUMS}, U\}$  and  $\ell_s.\text{xk} \neq U$ : treat as unsecured credit
7 :   if  $\ell_c.\text{amt} < \ell_s.\text{amt}$  and no empty position is known in the source's cohort:
8 :     prefer a full-balance voucher
9 :   to redeem: build tx for  $\Phi_{\text{CLAIM}}$  (position,  $\nu$ , fee input, preimage if any);
10 :  broadcast; rebuild as in Exit on a conflicting spend; settled once it confirms

```

1 to 4 are the recipient's MUSTs of spec 9.6: verify the signature, check that `tree_id` is one of the published creation parameters from which m is recomputed (spec 9.1), and verify that a leaf with the voucher's source body is in the tree with an amount that covers the voucher. Lines 5 and 6 are the lease check. Eviction voids a voucher without the issuer doing anything, since the evictor can pay the whole balance to the source's exit key once $\ell_s.\text{ea}$ has passed, so a recipient treats a voucher whose lease can pass within $\Delta(q)$ as unsecured credit and should want the lease to outlast its own redemption deadline plus $\Delta(q)$; the check is moot when the evictor is NUMS or the recipient, or when the exit key is the recipient's, the credit leaf of spec 10.3. Lines 7 and 8 are the empty-position caveat: a voucher for less than the whole balance needs an empty position for its claim leaf, which the operator controls. Lines 9 and 10 are the redemption, which only the voucher's signature authorises; it can be built by anyone who holds the voucher, a tower included. The spec states that the voucher is settled only after the redemption, or after the issuer includes the claim leaf and the recipient verifies inclusion, and that until then the recipient bears the risk that the issuer double-issues (spec 9.6). For a hash-locked voucher the recipient does not treat it as payment until it also holds the preimage and redeems promptly, because the issuer can void it by advancing `seq` (spec 9.6.1). The hash orders the two legs and does not make the exchange atomic: the voucher is a cheque, the value stays in ℓ_s , and until the redemption confirms the issuer can void it by a conflicting spend of that leaf (spec 9.6.2). The five rules of the spec fix the order. Line 1: the party the HTLC pays draws the preimage (rule 1), because it reveals it to collect the HTLC and the revelation enables the voucher; were the voucher

Algorithm 8 Swap – issuer A and holder B of a CLAIM^h voucher paired with an HTLC (an ordered exchange)

```

1 :  $A$ : draw  $R$ ;  $\eta \leftarrow \text{SHA256}(R)$ ; issue  $B$  a  $\text{CLAIM}^h$  voucher and an invoice on  $\eta$ 
2 :  $B$ : if the source's lease can pass before  $\text{cltv\_expiry} + \Delta(q)$ : abort
3 :  $B$ : pay the invoice (offer the HTLC)
4 :  $A$ : from the offer until the voucher is redeemed or the HTLC fails or times out:
5 :   do not advance  $\ell_s.\text{seq}$ 
6 :  $A$ : settle the HTLC off-chain, releasing  $R$  through the payment
7 :  $B$ : on learning  $R$ : redeem the voucher at once (Redeem)
8 :  $B$ : call the swap complete only when the redemption is buried so deep
9 :   that no reorganisation can replace it by a transition advancing seq
10 :  $A$ : if the HTLC failed or timed out: retire the voucher by advancing  $\ell_s.\text{seq}$ 

```

holder to draw it, it could redeem and never settle. Lines 6 and 7: the voucher is the second leg to resolve (rule 2), since it has no deadline and the HTLC does, and an HTLC must never wait on a preimage that only an on-chain redemption would reveal; where that order cannot be arranged the parties use two HTLCs. Lines 4 and 5 are rule 3, which no script enforces and which binds the issuer even after the HTLC settles: an issuer that collects and then voids has been paid for a voucher that can no longer be redeemed. Line 2 is rule 5, the lease check before the holder offers the HTLC. Lines 7 to 9 are rule 4: the holder carries the issuer risk until its redemption confirms and is buried, and the interval includes contention for the tree output, so it is governed by $\Delta(q)$ and not by a constant. Line 10 retires the voucher as spec 9.6.1 requires, and the issuer must not reuse η against the same leaf until it has. Usually only the voucher reaches the chain, at the cost of one CLAIM^h . The algorithm produces a transaction Φ_{REWRITE} accepts.

Algorithm 9 Rewrite – operator coordinating a cohort rewrite at height h

```

1 : collect the changes for one aligned subtree;  $n_j \leftarrow$  new element,  $\circ$  if unchanged
2 :  $\text{tx}:\text{in}_0 =$  tree output, plus funding for any increase
3 :  $\text{out}_0 = (S - \sum o_j.\text{amt} + \sum n_j.\text{amt}, \text{TreeSPK}(r', m))$ 
4 :  $\text{out}_1, \dots \leftarrow (o_j.\text{amt}, \text{P2TR}(o_j.\text{xk}))$  for each emptied leaf, in position order (a reclaimable leaf emptied under check 8 is not paid)
5 : for each  $j$  whose leaf changes (not an amount-only increase):
6 :   send  $(\text{tx}, o_j, n_j, \pi, r, m)$  to its owner
7 : each owner: if  $o_j$  is in  $r$  (fold), and  $n_j$  and every output are as agreed:
8 :   sign (and, under  $x$ , the exit key)
9 : if a required signature is missing: set  $n_j \leftarrow \circ$ , rebuild, collect from line 2
10 :  $\sigma_p \leftarrow \text{Sign}_{\text{OPERATOR\_KEY}}(\text{tx})$ ; assemble the witness; broadcast

```

Lines 3 and 4 are checks 7 and 6: each emptied leaf is paid exactly, which is why the operator cannot remove a leaf without paying it, and the output 0 amount nets the subtree's old and new amounts. Lines 5 and 6 send the transaction only to the owners whose leaves change in a way check 5 names, since a leaf that only gains amount, unless it is **SEQ_FINAL**, and a leaf nobody signs for pass through

unchanged; positions in the reclaim form need the operator alone (check 8). Lines 7 and 8 are the owner’s side, which the spec states for **UPDATE**, whose signers **MUST** verify the full transaction (spec 9.5), and which carries over to the cohort form by analogy; an owner who emptied its position is paid by the covenant, and one who did not sign keeps its leaf. Line 9 follows from the signature coverage: every signature covers all outputs, so any change to the transaction invalidates the others and the round restarts, as for a self-funded graft (spec 9.2). The spec does not specify the message flow, when to give up on an owner, or how requests are collected (design/batching.md describes the transition, not the coordination). Lines 4 to 8 are the sweep of spec 7.7: one transaction pays every leaf of the

Algorithm 10 Evict – operator, removing expired leaves

```

1 : admit a leaf to a cohort subtree only if  $\ell.\text{ek} = \text{OPERATOR\_KEY}$ ,
2 :   and align the leases per cohort
3 : before  $\ell.\text{ea}$ : notify the owner; offer an UPDATE extending  $\ell.\text{ea}$ 
4 : when  $\text{lock} \geq T = \max_j o_j.\text{ea}$  over an aligned subtree:

5 :    $\text{tx} \leftarrow \Phi_{\text{EVICT\_SUBTREE}}$  with the  $2^h$  elements published,  $\text{lock} = T$ 
6 :    $\text{out}_{k_j} \leftarrow (o_j.\text{amt}, \text{P2TR}(o_j.\text{xk}))$  for each leaf
7 :    $\text{out}_0 \leftarrow (S - \sum o_j.\text{amt}, \text{TreeSPK}(r', m))$ ; sign with OPERATOR\_KEY;
8 :   broadcast, paying the fee
9 :   for a leaf that blocks the sweep (later lease, other evictor):
10 :    wait, or remove it with  $\Phi_{\text{EXIT}}$ ,  $e = 1$ 

```

subtree in full to its exit key, so an eviction returns the member’s money and does not take it, at a small fraction of the per-leaf cost of evicting each leaf alone (spec 9.4). The sweep requires a lock time of at least the lease of every leaf in the subtree and **OPERATOR_KEY** as every evictor, which is why lines 1 and 2 make the evictor key a condition of joining a cohort and align the leases: one leaf with a later lease holds up its whole cohort, and one with another evictor blocks it outright. Line 3 is the spec’s advice (**SHOULD**) to notify and to offer a renewal; a renewal by **UPDATE** in place blocks the sweep the same way, so the spec recommends relocating the renewed leaf into a cohort whose lease matches, by an **UPDATE** followed by a **SPLIT** of the whole balance or by a cooperative exit and a **JOIN**₀ (spec 9.4). Lines 9 and 10 are the spec’s remedy for a blocking leaf (spec 7.7). A member whose lease is later than the cohort’s can carve into the cohort’s empty positions and delay its sweep; the spec asks operators to keep cohorts full or detach them, and gives no mechanism.

5 Realisations by Opcode Tier

Each predicate of Table 1 is compiled to tapscript against four opcode sets (Table 3), 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

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 [43] (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 [34] 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 <code>bitcoin/bips</code> 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 §3 (policy and fees, not consensus)	all

Table 3. 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 EXIT₁ 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). For scale, a hypothetical native Merkle-sum opcode, measured in the simulator and not a tier, would take EXIT₁ to 252 bytes of script at any depth and about 388 vB on a six-leaf tree, about 520 vB in a full depth-16 tree (§17).

	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

tier. FULL, MID and CCV compile every predicate of Table 1, MIN the dense single-leaf exit and SPLIT. Table 2 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.

FULL and MID. With `0xC2` the leaf is one 116-byte stack element read with `OP_SUBSTR`, sums use arbitrary-length arithmetic, one `LEVEL` of the fold is 88 bytes, and `TreeSPK` is recomputed with the EC opcodes; replacing the splice and EC opcodes by `OP_CAT` and `OP_TWEAKADD` costs 2,378 bytes of script at $D = 16$ against 2,101.

Under `0xC0` (`MID`) fields arrive as separate witness elements, checked with `OP_SIZE` and joined with `OP_CAT`, and a computed sum is widened to 8 bytes by a witness-supplied zero pad (Appendix B); a cap of 2^{31} satoshis (21.47 BTC) per tree and exact payouts are the price of 4-byte arithmetic.

MIN: introspection from a signature. Without `OP_TX` the only opcode that sees the transaction is `OP_CHECKSIG`. The script *constructs* the BIP-341 signature message from values it has derived (the recomputed $\text{TreeSPK}(r, m)$, the input amount, the new tree output, the payout) and raw witness pieces for what it does not constrain, and demands a Schnorr signature under the generator G with nonce G , Poelstra’s construction [37]: the challenge is $e = H_{\text{challenge}}(G_x \parallel G_x \parallel \text{msg})$, and $sG = (1 + e)G$ has the unique solution $s = e + 1 \bmod n$. The script computes e from its message and calls `OP_CHECKSIG` with signature $G_x \parallel s$ and key G_x ; since `OP_CHECKSIG` hashes the real transaction, it passes only if the two messages are equal, which is assumption [MIN]. Without long arithmetic, $e + 1$ is formed by grinding the fee input’s sequence number until e ends in 00 (256 trials on average, repeated on every fee bump; Appendix B). What is new is the use: the field-by-field enforcement and the signature as a tier’s only introspection. `EXIT1` costs 3,640 bytes of script at $D = 16$ (73% more than `FULL`) and 1,473 vB (44% more on the model, 47% serialised); vouchers and channel leaves need a signature over stack data and are out of reach, `SPLIT` is not (6,651 bytes). Whether a 64-leaf sweep (16 kB in `FULL`) fits `MIN` is open.

CCV: binding by opcode. `OP_CHECKCONTRACTVERIFY` [20] replaces every taproot-key recomputation of `FULL`: one call binds the old root to the input under the input’s own taptree, one commits output 0 to the new root with the residual assigned, one per payout asserts a `P2TR` of the leaf’s exit key and deducts its amount. The state key becomes the BIP-443 data tweak of `NUMS` by the root; the fold and the authorisation are `FULL`’s, in either numeric encoding, and r and m leave every witness. One `OP_TX` read per constrained amount survives, hence exact payouts.

6 Security

The tree-level facts are theorems about the abstract predicates, mechanised in Lean 4 [28] in about 4,500 lines without Mathlib (Appendix A) and cited by name; the composable claim is a simulation argument in the hybrid model of §2, sketched here and given in §7. We work in the idealised hash model: H_{leaf} and H_{node} are injective on the domains the protocol uses, and digests under different tags never coincide or equal the zero string. Predicates and bytecode correspond by execution (§10), not proof.

6.1 Tree-Level Lemmas

Lemma 1 (Encodings). *For $s < 2^{64}$ the minimal BIP-441 encoding of $s + 2^{64}$ is $\text{le}_{64}(s) \parallel 01$; leaf and node serialisations are injective; 2^{32} has a five-byte minimal encoding and 2^{31} a five-byte `CScriptNum`, while $-(2^{31} - 1)$ is `ffffffff` with successor `feffffff`. (`enc64_eq_minimal`, `serializeLeaf_inj`, `serializeNode_inj`, `minimalLE_two_pow_32_length`, `scriptNum_two_pow_31_length`, `scriptNum_neg_max31_succ`.)*

Theorem 1 (Binding). *Roots bind trees: two trees of the same depth with equal roots are equal. Any node n and path π with $\text{fold}(n, \pi)$ equal to the root of a tree T are the root of one of T 's subtrees and that subtree's honest path; in particular the sum in n is the true sum of that subtree, and a leaf node that verifies with a full-length path is the leaf at that position. (`root_inj`, `fold_sound`.)*

```
theorem root_inj {d} (t1 t2 : Tree d) : (t1.root : Node H) = t2.root → t1 = t2
theorem fold_sound {d} (t : Tree d) (m : Node H) (π : List (Bool × Node H)) :
  fold m π = t.root → ∃ (h : Nat) (p : Pos d h), m = (t.getSub p).root ∧ π =
    t.pathSub p
theorem fold_setSub {d h} (t : Tree d) (p : Pos d h) (s : Tree h) :
  fold s.root (t.pathSub p) = ((t.setSub p s).root : Node H)
```

Proof (sketch). Induction on the path. A non-empty path claims an internal node, so the tree is not a tip by tag separation. Injectivity of H_{node} on the parent's preimage equates both children's hashes *and* sums with the claimed ones, and the induction hypothesis applies to the child the direction bit selects. This is the only place the hash axioms are used.

Lemma 2 (A step is a subtree replacement). *If the current root is the root of a tree T and a well-formed step is valid from it to r' , its old node is the root of T 's subtree at some position p of the step's height, and r' is the root of T with that subtree replaced by any tree whose root is the step's new node. (`Step_sound`.)*

Theorem 2 (Conservation). *For any sequence of valid steps from (r, S) to (r', S') , $S' + \sum s_{\text{old}} = S + \sum s_{\text{new}}$; for EXIT_K , $S' + \sum_k \text{payout}_k \leq S$, with equality under exact payouts. No hash assumption is needed. (`Steps.sum`, `ExitK.no_inflation`.)*

```
theorem Steps.sum {r r' : Node H} {ss : List (Step H)} (h : Steps r ss r') :
  r'.sum + oldSum ss = r.sum + newSum ss
theorem ExitK.no_inflation {exact} {r r'} {blocks} {out0} (h : ExitK exact r blocks r'
  out0) :
  out0 + payoutSum blocks ≤ r.sum -- ExitK.exact_conservation: equality if exact
```

Theorem 3 (Payability). *For any tree T and position p holding leaf ℓ , the honest witness (the leaf, its path, E_0) is a valid step from T 's root to the root of T with p emptied, the sum falls by $\ell.\text{amount}$, and every other position is unchanged. (`payable`.)*

```
theorem payable {d} (t : Tree d) (p : Pos d 0) (l : Leaf) (hl : t.get p = some l) :
  (leafStep (t.path p) (some l) none).valid t.root (t.set p none).root ∧
  (t.set p none).root.sum + l.amount = t.root.sum ∧ ∀ q, q ≠ p → (t.set p none).get q
  = t.get q
```

Theorem 4 (Real-tree invariant). *If the current root is the root of a tree and every step is well formed and grounded (its new node is the root of some tree of the step’s height), the final root is the root of a tree of the same depth. Leaf-level steps are grounded by construction; a JOIN graft is grounded exactly when the operator’s node has a preimage tree. (Steps.sound, join_sound.)*

```

theorem Steps.sound {d} (t : Tree d) {r'} {ss : List (Step H)} (h : Steps t.root ss r')
  (hwf : ∀ s ∈ ss, s.WF d) (hg : ∀ s ∈ ss, s.Grounded) : ∃ t' : Tree d, t'.root = r'
theorem join_sound {auth : Prop} {d h} (t : Tree d) (path) (new r' : Node H) (out0 : Nat)
  (hwf : path.length + h = d) (hj : Join auth h t.root path new r' out0) :
  ∃ p : Pos d h, t.getSub p = Tree.empty h ∧ out0 = t.root.sum + new.sum
  ∧ ∀ u : Tree h, u.root = new → r' = (t.setSub p u).root

```

Lemma 3 (Path maintenance). *If leaf ℓ at p with path π folds to the root r , and a valid step at a position p' that is neither p nor above it takes r to r' , then ℓ folds to r' with the path π in which the one sibling on the step’s path is replaced by the node the step computes at that height.*

The Lean model proves the statement for a real tree (`Tree.path_maintenance`, `Tree.path_maintenance_valid`). The lemma as stated needs no real tree, so it also covers a path through an opaque node, for which §7 gives a paper proof. Other transitions have theorems of their own (Appendix A). Authorisation and lock times are an opaque proposition `auth` in the theorems, discharged by $\mathcal{F}_{\text{sig}}$ and $\mathcal{G}_{\text{clock}}$; which key a payout pays (the leaf’s exit key, not the evictor) is established by execution, not by these theorems. The predicates bind the witness root and taptree to the spent output through $\text{TreeSPK}(r', m') = \text{TreeSPK}(r, m)$ (`witness_root_binds`), which reduces to the discrete logarithm of NUMS and hash collisions when the two internal keys have opposite signs, and to taproot binding in the random-oracle model, as in BIP-341’s own argument, when they share one (Appendix A).

Predicates, algorithms and lemmas. The lemmas meet the boxes and the algorithms at fixed places. Check 1 of every box is the binding of the witness root and taptree to the spent output (`witness_root_binds`). The fold checks of every box are the hypothesis of Lemma 2 and their conclusion is a step, so the amounts that the last check of each box fixes are Theorem 2 applied to its steps. Lines 2 to 4 of Algorithm 4 build the honest witness of Theorem 3 (the leaf, its path, E_0), and line 5 of Algorithm 5 is Lemma 3. Lines 3 to 6 of Algorithm 3 are the checks that Theorem 4 needs and that Φ_{JOIN} cannot make: a graft without them can give an opaque node, which freezes the value it brought and nothing else. The signature checks of the boxes are the opaque authorisation of the Lean structures, discharged by $\mathcal{F}_{\text{sig}}$ and $\mathcal{G}_{\text{clock}}$. Appendix A lists every box, algorithm and result with its Lean name.

6.2 The Composable Statement

Let Π_τ be the protocol with the predicates compiled by tier τ , and $\mathcal{G}_{\text{ledger}}[\text{Pred}_\tau]$ the global ledger whose validity predicate, for a spend of a tree output, is that

tier’s predicate. Let $\mathcal{F}_{\text{LT}}|_\tau$ be $\mathcal{F}_{\text{LT}}$ restricted to what τ compiles into m : the interfaces of Table 3, the heights h and batch sizes K in the taptree, exact and amounts below 2^{31} in MID and MIN (exact in CCV), no graft in the sparse profile, at most 54 occupied positions per MID sweep, and for MIN only the dense one-leaf EXIT and SPLIT, without the terminal exit. Trees whose taptree carries REWRITE_h or DETACH_h are excluded: neither is an interface of $\mathcal{F}_{\text{LT}}$, and their Lean theorems are all we claim for them.

Theorem 5 (UC realisation). *In the idealised hash model, with taproot binding in the ROM and under $[C]$, for each tier τ , Π_τ UC-realises $\mathcal{F}_{\text{LT}}|_\tau$ with $\Delta(q) = \Delta_L(\lceil(\beta + q)/\kappa\rceil + 1)$, in the hybrid model with $\mathcal{G}_{\text{ledger}}[\text{Pred}_\tau]$, $\mathcal{G}_{\text{clock}}$ and $\mathcal{F}_{\text{sig}}$ (in MIN, under $[MIN]$), against a static adversary corrupting the operator and any members, for trees without REWRITE_h and DETACH_h .*

Claim 1 (Simulation invariant). *After each ledger event, $\mathcal{F}_{\text{LT}}$ ’s partial tree has the root the tree output commits to, and agrees with every honest party’s leaf and path.*

The proof is the next section.

What the theorem leaves out. The bytecode is matched to Pred_τ by 3,019 executable checks, not a verified compiler. Fee pinning is outside the ledger model: a low-feerate transition early in the chain caps the feerate behind it, a descendant hung off any payout or change output consumes the chain’s policy limit, and TRUC transactions [51] cannot help, since they cap an unconfirmed chain at two; $[C]$ assumes this away, and cooperative batching, anchors and sizing by exit throughput mitigate it. The operator can evict any leaf after its lease, a cohort at about 160 vB per leaf, so griefing is cheap; a longer guarantee needs a later lease at entry, and no carve can extend it.

7 The UC Proof

This section gives the simulator and the hybrid argument behind Theorem 5. We follow the conventions of the composable ledger treatment of Badertscher et al. [3] and of Generalized Channels [2]. $\mathcal{F}_{\text{LT}}$ is protocol-aware: for an honest EXIT or EVICT it constructs the transaction of Figure 2 itself, from the honest party’s leaf and path and the fee coin f named in the call, obtains the signature through $\mathcal{F}_{\text{sig}}$, and submits it to $\mathcal{G}_{\text{ledger}}$. The simulator never builds an honest transaction; it only schedules. The two worlds therefore agree on the ledger contents and differ only in how the tree state is kept.

7.1 The Hybrid World

The real protocol Π_τ runs in the presence of:

$\mathcal{G}_{\text{ledger}}[\text{Pred}_\tau]$ the global ledger functionality of Badertscher et al., parameterised by a validity predicate that, for a transaction spending a tree output, evaluates the tier’s covenant predicate Pred_τ on the input state, the witness and the outputs (§4). It guarantees persistence, and liveness with parameter Δ_L : a valid transaction with sufficient fee that an honest party submits, and that stays valid, is in the ledger state of every honest party within Δ_L rounds. The adversary orders transactions within the rules, and may get a conflicting spend in first, which invalidates the honest one; [C] bounds how often.

$\mathcal{G}_{\text{clock}}$ the global clock of Katz et al. [22]; rounds are blocks.

$\mathcal{F}_{\text{sig}}$ the ideal signature functionality [10], which we use in place of an EUF-CMA reduction; every **CHECKSIG** and **CHECKSIGFROMSTACK** in the scripts is a call to its verification interface, with one exception: MIN’s **OP_CHECKSIG** under the generator G , whose key has no honest signer, is real BIP-340 verification, and its binding is assumption [MIN] (Table 2).

Hashes are treated in the idealised model of §6 (injectivity plus tag separation), and the taproot commitment in the random-oracle model where the two internal keys share a sign (Appendix A). Every abort event below is one in which the simulator holds two distinct preimages of one digest, so the argument transfers to a collision-resistance reduction.

7.2 The Simulator and the Invariant

$\mathcal{S}$ runs an internal copy of $\mathcal{A}$ and of every honest party’s protocol state, and keeps a partial tree $\hat{T}$ equal to $\mathcal{F}_{\text{LT}}$ ’s. Claim 1 is the invariant: after each ledger event, $\hat{T}$ has the root the tree output commits to, and every honest party’s leaf is a known node of $\hat{T}$ whose path folds to that root.

Ledger events. Whenever a transaction tx spending a tree output appears in the ledger, $\mathcal{S}$ parses its witness:

1. Identify the script leaf from the control block; this fixes the transition type and its parameters $(K, h, \text{profile}, \text{variant})$.
2. For each step, fold the supplied node up the supplied path. Where the path passes through an opaque node of $\hat{T}$, call **REVEAL** for each level with the nodes the witness supplies (they hash to the opaque node because the fold reaches the committed root); where it passes through a known node and the fold disagrees with it, $\mathcal{S}$ holds two preimages of one digest (abort event **Coll**). Otherwise Theorem 1 gives the unique position p the step proves.
3. Map the steps to the interface: a leaf replaced by E_0 is **EXIT** (owner flag) or **EVICT** (evictor flag); E_h replaced by a node is **JOIN**, with the subtree $\mathcal{S}$ obtains from the operator’s join messages, or its opaque root; a leaf replaced by a leaf is **UPDATE**; the two-step carve is **VOUCH** followed by **CLAIM** (for **SPLIT**, or a voucher a corrupt owner issued) or **CLAIM** of a recorded voucher; the post-order pattern is **SWEEP**.
4. Check the authorising signatures with $\mathcal{F}_{\text{sig}}$. A valid signature under an honest key that the honest party did not produce is a forgery (abort event **Forge**).

5. If tx is an honest call that $\mathcal{F}_{\text{LT}}$ posted, let it take effect; otherwise invoke the interface on behalf of the corrupt party that authorised it.

Honest calls. When an honest party calls an interface, $\mathcal{F}_{\text{LT}}$ builds and posts the transaction and $\mathcal{S}$ chooses, within $\Delta(q)$, when it takes effect, matching the round in which $\mathcal{G}_{\text{ledger}}$ includes it. When a corrupt operator grafts or creates a node whose preimage $\mathcal{S}$ cannot obtain, $\hat{T}$ holds it as opaque; an honest member never funds such a node without verifying its own path, and $\mathcal{S}$ reveals that path from the member's state, so the member's leaf is known.

The invariant, by case. Assume Claim 1 before an accepted spend. *Exit, evict, update, carve.* Each step is valid from the committed root, so by Lemma 2 the new root is the root of $\hat{T}$ with the step's position replaced, which is $\mathcal{F}_{\text{LT}}$'s tree after the call; Theorem 2 gives that the amounts agree, and the bytecode that the destinations do. A drained carve source becomes E_0 in both (claimedSource). *Join and create.* The new node is placed at an empty position (join_sound), known or opaque. *Sweep.* evictSubtree_sound gives the emptied subtree and one payout per leaf. *Reveal.* It changes no root. In every case an honest party's leaf outside the changed position keeps a valid path by Lemma 3, and one inside it was removed by its owner's or evictor's own call.

Proof of Lemma 3. Let the step at p' have old node n , path π_s and new node n' , and let k be the height at which the paths of p and p' meet. Both $\text{fold}(\ell, \pi)$ and $\text{fold}(n, \pi_s)$ equal r ; reading the two folds from the root down, injectivity of H_{node} equates, level by level, the two parents' preimages, so the siblings above height k coincide, ℓ 's sibling at height $k - 1$ is the node the step's fold reaches at that height, and vice versa. Folding n' up π_s changes only that node and the nodes above it; replacing ℓ 's sibling at that level by the new node and folding with the unchanged siblings above therefore yields r' . Below height $k - 1$ nothing on ℓ 's path changed. No real tree is needed, only the two folds.

7.3 Hybrids

- $\mathcal{H}_0$ the real execution of Π_τ with $\mathcal{A}$ in the $(\mathcal{G}_{\text{ledger}}, \mathcal{G}_{\text{clock}}, \mathcal{F}_{\text{sig}})$ -hybrid world.
- $\mathcal{H}_1$ as $\mathcal{H}_0$, but the execution aborts on Forge. Indistinguishable by the definition of $\mathcal{F}_{\text{sig}}$.
- $\mathcal{H}_2$ as $\mathcal{H}_1$, but the execution aborts on Coll, on a taproot-commitment collision, and in MIN on a violation of [MIN]. In the idealised hash model and the ROM these have negligible probability; in each, the simulator outputs the colliding preimages.
- $\mathcal{H}_3$ as $\mathcal{H}_2$, but the honest parties' tree state is replaced by $\hat{T}$ and their transactions are built and posted by $\mathcal{F}_{\text{LT}}$. The invariant above makes the two executions identical transaction by transaction; payability (Theorem 3, and Lemma 3 for a path through an opaque node) gives that every honest exit $\mathcal{F}_{\text{LT}}$ builds is valid.
- $\mathcal{H}_4$ the ideal execution of $\mathcal{F}_{\text{LT}}$ with $\mathcal{S}$. Identical to $\mathcal{H}_3$ by construction.

Liveness. $\mathcal{F}_{\text{LT}}$ lets $\mathcal{S}$ delay an honest call by up to $\Delta(q)$. In the real world the honest transaction confirms unless a conflicting spend confirms first; under [C] at most κ spends of the output confirm per block, at most β of the conflicting ones are adversarial, and q honest calls are queued ahead, so after at most $\lceil(\beta+q)/\kappa\rceil$ blocks of Δ_L rounds each the honest party’s rebuilt transaction, valid by Lemma 3, is included within one further Δ_L . Nothing beyond [C] is claimed: the model does not bound β , and pinning (§6) is one way it grows.

7.4 Per-Tier Remarks

The argument is the same for every tier because the predicates are the same; what differs is $\mathcal{F}_{\text{LT}}|_\tau$ (§6) and one detail each of MIN and CCV. In MIN, CHECK_OLD and FINAL are enforced through the constructive signature check of §5, so $\mathcal{S}$ recovers the enforced fields from the witness’s raw pieces rather than from OP_TX selectors, and their binding is [MIN]. In CCV the witness carries neither r nor m : the input’s commitment to the old root and output 0’s to the new one are checked by OP_CHECKCONTRACTVERIFY under the input’s own taptree, so $\mathcal{S}$ reads r and m from the spent output, and the binding of witness_root_binds rests on the BIP-443 data tweak in place of TreeSPK (CCVTweak.binding, ccv_input_binds). In every tier the voucher message names the lineage’s tree_id (spec 0.10.0), so a voucher for one instance of $\mathcal{F}_{\text{LT}}$ does not verify in another, and $\mathcal{F}_{\text{LT}}$ ’s VOUCH records it per instance. REWRITE_{*h*} and DETACH_{*h*} are outside the theorem (§6).

7.5 Composition for Channel Leaves

Conjecture 1 would follow the pattern of Generalized Channels [2], in the hybrid model with $\mathcal{F}_{\text{LT}}$, $\mathcal{G}_{\text{ledger}}$, $\mathcal{G}_{\text{clock}}$ and $\mathcal{F}_{\text{sig}}$. The channel simulator would reuse the channel-level simulator of that work for the update and settlement transactions, which are ordinary ledger transactions once the funding output exists, and use $\mathcal{F}_{\text{LT}}$ ’s EXIT and EVICT to materialise it. The three tree-layer preconditions of §8 are the conditions under which $\mathcal{F}_{\text{LT}}$ guarantees that the materialised output carries the full capacity to the channel key: exact payouts (ExitK.exact_conservation), non-claimability (the SEQ_FINAL facts), and a lease at or below the tip so that EVICT is available to the counterparty. The missing pieces are the HTLC deadline argument under [C] and the rebinding of pre-signed states across tree transitions; they are exercised block by block on the simulator’s chain model (§9), not proved.

8 Channel Leaves as Composition

A leaf whose exit key is a 2-of-2 between its owner and a counterparty (normally the operator) can host a payment channel that exists only off-chain and whose funding output materialises at the payout of the EXIT or EVICT that removes the leaf, so a tree of such leaves is a Lightning hub behind one UTXO. The channel protocol runs in the $\mathcal{F}_{\text{LT}}$ -hybrid model and treats the payout as its funding transaction; §9 gives scripts, state machine and costs. The funding outpoint

changes with every tree transition, so states are signed without committing to it (OP_TX collates the signed fields behind a per-channel context, and CSFS checks the signature over the hash, emulating SIGHASH_ANYPREVOUT [15]), and the channel uses the eltoo ratchet [14,46] with zero-fee version-3 transactions [51] and an ephemeral anchor [5]. This needs OP_TX and CSFS: FULL, MID, or CCV with the channel scripts in their own outputs.

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 [25]. 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). MID allows seven HTLCs per state beside two balances (the 520-byte element limit) and 2^{31} satoshis per tree.

Composition conditions. Let $\mathcal{F}_{\text{chan}}$ be a channel functionality in the style of Generalized Channels [2]. Three of $\mathcal{F}_{\text{LT}}$ ’s interfaces are too lenient for a leaf whose balance is not the owner’s alone, and each becomes a precondition: (i) `exact` is set, so the owner cannot underfund the channel; (ii) `seq` is the value whose successor does not fit in four bytes (0xFFFFFFFF under BIP-441 numerics, 0x7FFFFFFFF under CScriptNum, Lemma 1), so the owner cannot carve the leaf; (iii) the lease height is zero, giving the counterparty EVICT (between two members, UPDATE^x instead).

Conjecture 1. Under [C], in the $(\mathcal{F}_{\text{LT}}, \mathcal{G}_{\text{ledger}}, \mathcal{G}_{\text{clock}}, \mathcal{F}_{\text{sig}})$ -hybrid model with conditions (i) to (iii), the channel-leaf protocol realises $\mathcal{F}_{\text{chan}}$ with closing bound $\Delta(q) + 3 + \text{csv_delay}$ for channels whose HTLC deadlines respect the budget above.

The proof obligations are the three conditions, the HTLC deadline argument under [C], and the rebinding of pre-signed states across tree transitions (§7); the state machine and the budget run end to end on the simulator’s chain model, with two blocks of margin, but are not proved.

REBIND _n (0xC2, 81 B)	SETTLE _n (0xC2, 79 B)
$\langle S_0 + n + 1 \rangle$ OP_CHECKLOCKTIMEVERIFY OP_DROP	$\langle \text{csv_delay} \rangle$ OP_CHECKSEQUENCEVERIFY OP_DROP
$\langle \text{SEL_CHAN_UPD} \rangle$ OP_TX // collated blob	$\langle \text{SEL_CHAN_SET} \rangle$ OP_TX // + amount, spk
$\langle \text{ctx}_{\text{upd}} \rangle$ OP_SWAP OP_CAT OP_SHA256	$\langle \text{ctx}_{\text{set}} \rangle$ OP_SWAP OP_CAT OP_SHA256
$\langle P_{\text{chan}} \rangle$ OP_CHECKSIGFROMSTACK	$\langle P_{\text{chan}} \rangle$ OP_CHECKSIGFROMSTACK

Fig. 4. The two tapleaves (version 0xC2) of a channel leaf’s update output for state n , sized at $\text{csv_delay} = 144$; under 0xC0 SETTLE_n is 80 B, since 144 needs two bytes as a CScriptNum.

9 Channel Leaves: Scripts and Costs

Selectors and context. Two OP_TX selectors are used, both with the collate flag so that one call leaves one blob: SEL_CHAN_UPD selects version, number of inputs, the current input’s sequence, number of outputs, all output amounts and scripts, and lock time; SEL_CHAN_SET adds the current input’s amount and scriptPubKey. The per-channel context is

$$\begin{aligned} \text{chan_id} = H_{\text{chanid}}(\text{tree_id} \parallel \text{owner_key} \parallel \text{le}_{32}(\text{nonce}) \parallel x(P_{\text{chan}}) \\ \parallel \text{le}_{64}(C) \parallel \text{le}_{32}(\text{epoch}) \parallel \text{le}_{32}(\text{csv_delay})), \end{aligned}$$

with ctx_{upd} and ctx_{set} two tagged hashes of it, so that an update signature never verifies where a settlement signature is required. The signed messages are $\text{SHA256}(\text{ctx} \parallel \text{OP_TX}(\text{SEL}))$.

Update output scripts. The update transaction for state n pays its whole input value to an output U_n with internal key P_{chan} (a MuSig2 [31,30] key of the two parties) and the two-leaf taptree of Figure 4. A later update binds to U_n through REBIND_n, whose lock-time check admits only states above n ; the settlement of state n binds through SETTLE_n after csv_delay blocks, and only to U_n because SEL_CHAN_SET commits the spent script, which embeds n .

Transactions. The update transaction is version 3, lock time $S_0 + n$, one input with sequence 0xFFFFFFFF, one output of the full capacity C to U_n and a zero-value pay-to-anchor output; it pays no fee and is valid whether bound to the funding output or to any earlier update output, both of value C . The settlement is version 3 with input sequence csv_delay , outputs for the two balances and each in-flight HTLC, and an anchor; $b_U + b_V + \sum \text{htlc} = C$. Fees for both come from a child spending the anchor, so both parties need an on-chain fee reserve to close unilaterally. The signing order is settlement before update for each state, as in eltoo.

Force-close cost. At $D = 16$ dense, $\text{csv_delay} = 144$ and no HTLCs in flight:

step	vB earliest
EXIT ₁ by owner or evictor, with fee input and change	$\approx 1,020\ t$
update _n bound to the funding output	$153\ t + 1$
anchor child for update _n	$152\ t + 1$
settlement _n (two payouts, anchor)	$204\ t + 1 + \text{csv}$
anchor child for settlement _n	$152\ t + 1 + \text{csv}$
total	$\approx 1,681 \approx 146\ \text{blocks}$
contested rebind to an earlier update output	161 per contest
settlement with h HTLC outputs	$204 + 43h$
HTLC-success sweep	155 after settlement
HTLC-timeout sweep	138 after <code>cltv_expiry</code>

An HTLC output has two leaves, so each sweep carries a 65-byte control block. The lifetime on-chain footprint of a channel leaf that is force-closed is therefore about 1,680 vB. For a simple taproot channel [36] priced the same way, the funding transaction (one P2TR key-path input, the funding output and change) is 154 vB, the commitment (a MuSig2 key-path input, two balance outputs and two anchors) 240 vB, its anchor child 152 vB and the delayed sweep of the broadcaster’s output 138 vB: about 680 vB in all, with both constructions carrying their fee-bumping children (BOLT 3’s anchor commitment [26] is 1,124 WU, 281 vB, in the non-taproot case). Cooperative closes, splices and lease renewals of a channel leaf cost one tree transition or nothing. The update transaction cannot be packaged with the exit because a version-3 transaction may not spend an unconfirmed version-2 parent, so G_{exit} in §8 is measured to confirmation, and the costs above are for a tree of the occupancy that budget allows.

Limits and execution. Under 0xC0 every stack element is at most 520 bytes, including the collated settlement blob of $156 + 43h$ bytes with both balances present, so a MID channel carries at most seven HTLC outputs per state beside two balances, or eight beside one; FULL has no such limit. The scripts and transactions above run, in both tapleaf versions, on a chain model that enforces BIP-65 and BIP-68 (210 checks): a leaf leaves a tree by EXIT₁[−], a state signed before the exit binds to whichever funding output an exit or an eviction creates, a stale update is answered in the last block of its window, and both HTLC sweeps confirm at their deadlines. Counted as those rules count, the budget of §8 has two blocks of margin: with $G_{\text{exit}} = 72$ and $G_{\text{sweep}} = 2$ the payee’s sweep confirms at $t + 219$ against a bound of 221. A Go mirror generates the same scripts byte for byte, with channel messages in Lightning’s type-length-value (TLV) encoding.

10 Evaluation

Implementation. A Python tapscript interpreter for both leaf versions models BIP-441 numerics, OP_TX, CSFS, the EC opcodes and varops, computes the real BIP-341 signature message and enforces MINIMALDATA; it models consensus limits, not standardness. Every compiled predicate runs against a reference tree with

positive and negative cases per rule (forged leaf, stale root, attacker-chosen m , unconserved amount, early eviction, voucher replay, and others): 1,610 FULL, 1,391 MID, 57 MIN and 4,295 CCV checks, failing if a script uses an opcode outside its tier, plus 220 for channel leaves and 133 for the hypothetical opcode of §17. A Go implementation on `btcd` and `lnd` generates the same scripts, byte-exact against 145 golden scripts, and builds, signs and verifies every transition of every tier (427 cases), each witness replayed through the simulator; its BIP-341 signature hash equals the simulator's.

The transaction model. Every vB figure is a whole transaction: script, witness elements and a 129-byte control block (the eight-leaf reference script set) as measured, plus a 946 WU skeleton (the tree input, a P2WPKH fee input, the tree output, one P2TR payout, a P2WPKH change output, version, lock time and counts) and 172 WU per further payout, with $vB = \lceil WU/4 \rceil$; transitions without a payout (joins, updates, carves) are charged 172 WU less. The witness item count and per-item length prefixes are excluded (on a dense $D = 16$ exit, 10.5 vB in FULL, 10 in CCV, 42.5 in MID, 44.75 in MIN), and the baselines omit the same overhead. A deployment that carries the sweeps, rewrites and variants needs 16 to 32 tapleaves, which adds 8 to 16 vB to every spend.

Exit cost. A dense exit (Figure 5, Table 9) 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 5 is a *model*: an exiter publishes D transactions of about 150 vB (one key-path input, two P2TR outputs) and a 111 vB claim, anchors and their fee children omitted, which favours the tree. LayerTree is about 60% smaller at $D = 16$; a radix-4 tree needs 2,031 vB, and an Ark member of a 256-member round unrolls 1,311 vB. Under mass exit a tree's interior is shared, about 261 vB per member at radix 2 and 191 vB at radix 4 with the claim, against 165 vB for a cohort leaving by `REWRITE6` (Table 12, Appendix C). The model omits the latency of D confirmations and the larger opcode set.

A native 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` 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. 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.

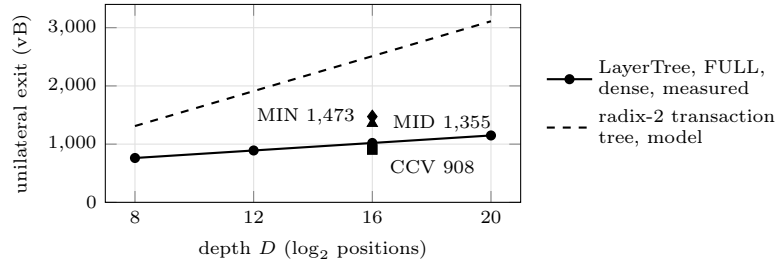


Fig. 5. Unilateral exit cost against tree depth: measured for LayerTree (FULL, and the other tiers at $D = 16$), modelled for an unamortised radix-2 transaction tree.

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. Figure 6 plots this. The six-leaf 388 vB of Figure 7 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 (§15) 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 (§17).

Batching and joins. A second exit in the same transaction adds 665 vB (its marginal weight rounded up on its own, one more than $1,684 - 1,020$), against 158 to 165 vB per leaf for a sweep or a rewrite (Table 4). A graft costs the operator under 3 vB per member, so a self-funded member’s own input dominates its entry. Off-chain, a member follows every spend of the tree output (about 1 kB of witness each), and the operator stores the whole tree.

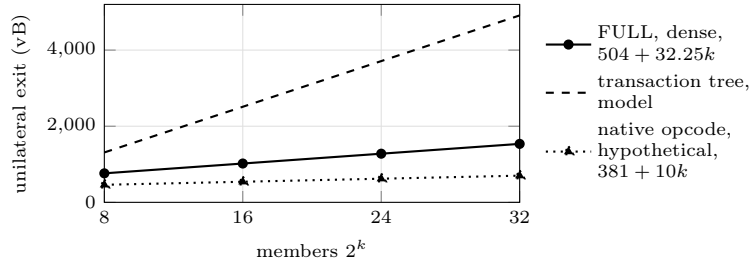


Fig. 6. Exit cost against membership 2^k under the scaling model: a key-addressed depth-64 tree for the hypothetical native opcode, dense depth k for FULL, a radix-2 transaction tree. The dotted line needs an opcode no one has proposed.

Table 4. 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 §4.4. 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

Fees and budgets. At 10 sat/vB a $D = 16$ exit costs 10,200 sat, a 1% fee on 1,020,000 sat (Appendix C). Varops usage is 5.6 to 7.7% of the BIP-440 budget in the dense profile, falling with depth because the budget scales with weight; 0xC0 tiers use one to three signature operations of the BIP-342 sigop budget.

Artefact. Every number here is regenerated from the repository [35] at the revision on the title page (ba8422e) by the scripts under `tools/`: `test_scripts.py` (1,610 checks; `--rewrite` and `--subtree` print the cohort shapes), `test_mid.py` (1,391), `test_min.py` (57), `test_ccv.py` (4,295), `test_chan.py` (220), `test_msu.py` (133), `gen_listings.py` (the size table), `fee_analysis.py` (the fee model), `gen_golden.py` and `experiments/radix4.py` (55); by `go test ./...`; and by `lake build` in `formal/`, which prints the axiom audit. Figure coordinates are copied from these outputs by hand.

11 Costs Against the Alternatives

Joining, updating and leaving a shared output each cost more bytes than the same operation on a bare UTXO. What sharing buys is a different placement

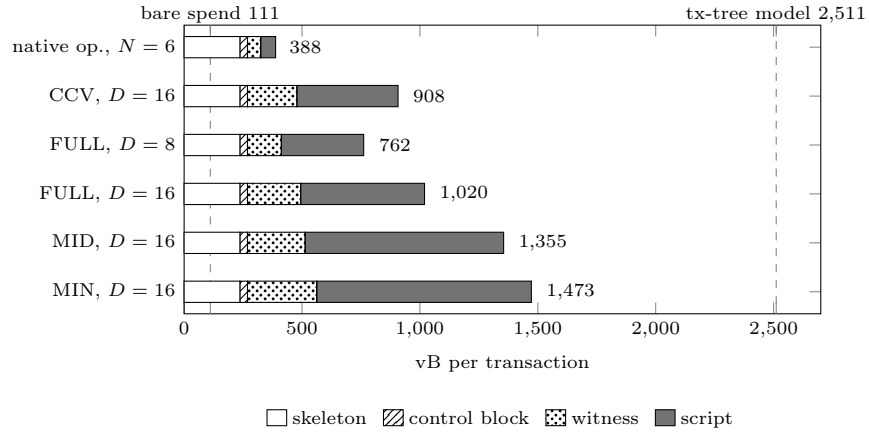


Fig. 7. One unilateral exit by variant, dense unless noted. Skeleton (236.5 vB, white) and control block (32.25 vB, hatched) are common to every row, then witness (dotted) and script (grey); the script dominates because the levels are unrolled. Dashed: a bare P2TR spend and the radix-2 transaction-tree model at $D = 16$. The native-opcode row is measured on a six-leaf tree; it is a hypothetical opcode in the simulator (§17) with a bitmap-compressed proof.

of the cost: entry is nearly free in bulk, value moves off-chain for free, a bulk exit costs about a bare spend, and a solo exit costs about ten times one but is a single transaction with no timelock and no counterparty. All figures in this and the following sections use the model of §10. Figures marked *estimate* are built from measured sizes for scripts or populations not measured directly.

Exit. Figure 7 decomposes exits; taking the fee from the payout brings the FULL exit at $D = 16$ to 921 vB. The hypothetical native opcode’s 388 vB is measured on a six-leaf tree, where the whole witness (leaf, signature and compressed proof) is 56 vB; with 2^{16} members the proof grows by about 14 non-empty siblings to roughly 530 vB, the better estimate of the floor for a design that publishes a proof in script. The transaction-tree model (Table 12) ignores the leaf’s timelock and the fee of each of the exiter’s transactions. In bulk the cost per leaf depends on who signs: with EXIT_K every leaf carries its own path, so a second exit adds 665 vB, while the operator’s sweep of expired leases (only for members who accepted a lease) and the owners’ cohort rewrite share the path and reach about 160 vB per leaf, under 1.5 times a bare spend (§13). The chain limit counts transactions, not leaves.

Entry and moving value. JOIN_h walks only the levels above its slot, so at $D = 16$ an operator-funded 1,024-member graft costs 614 vB (estimate, without a payout output), under 1 vB per member, against 43 vB for one output in an on-chain batch and about 154 vB for a channel opened in its own transaction (§14); a self-funded member adds its own 58 vB input and the signing round of §4.4. An

Table 5. 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 C.2). 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

on-chain UPDATE is 1,030 vB and a voucher redemption 1,618 vB (neither has a payout output; on the skeleton that carries one, an UPDATE is 1,073 vB), so moving value by rewriting leaves would cost about ten times a bare spend per payment. Value moves off-chain instead, as it does in Lightning and Ark: in channel leaves as HTLC forwards (§8), or by vouchers that the issuer usually settles rather than the holder redeems (§15). UPDATE is maintenance, not payment.

Lifetime and latency. Table 5 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.

12 Node Format and Radix

The tree is a Merkle-sum tree whose position assignment, in the dense profile, is the operator’s. This section gives the node format and the choice of radix; the sparse alternative is in Appendix C.2.

One node format. A parent hashes each child’s sum, not only their total, and this is a soundness property. A tree hashing $h_L \parallel h_R \parallel (s_L + s_R)$ admits inflation: two leaves of 100 under a root of sum 100 each have a proof whose sibling claims sum zero, and together the proofs claim 200. Here the two proofs hash the distinct preimages $(A, 100, B, 0)$ and $(A, 0, B, 100)$, so at most one reaches the root; in Lean this is `serializeNode_inj` with `hNode_inj`, from which `fold_sound` gives that a verified leaf carries its true amount.

Table 6. Radix in the dense FULL tier, measured: a radix- r level generated and run in the simulator against a radix- r reference tree (55 checks), and EXIT₁ for 65,536 slots (262,144 at radix 8). The radix-2 row reproduces the shipped LEVEL.

radix	level script (B)	level siblings (B)	levels	EXIT ₁ script (B)	witness (B)	tx (vB)
2	88	41	16	2,101	902	1,020
4	146	123	8	1,861	1,230	1,042
8	215	287	6	1,983	1,968	1,257

Dense positions. An exit proves inclusion and a join proves that its slot is E_h , the same way in either profile. What the dense profile adds is aligned subtrees, on which JOIN _{h} , EVICT_SUBTREE _{h} and every shape of §13 depend: cohort grafts at one price and sweeps at about 160 vB per leaf, at the price of an operator who keeps the slot map (spec 5.4, `spec/layertree.md`).

Radix. A radix- r path carries $41(r-1)D/\log_2 r$ sibling bytes, $41D$ at radix 2 and $61.5D$ at radix 4, so radix 2 minimises the witness and a higher radix can only save the fixed part of the unrolled level. Table 6 measures it. Of the radix-4 level’s 146 bytes about 62 are hashing and sums shared with radix 2 and the rest sibling and direction handling, so the fixed part a higher radix amortises is about 60 bytes while the sibling part is what it multiplies; radix 4 comes out about 2% larger overall. With a native Merkle-sum opcode a level costs only its sibling and radix 2 is optimal. The sparse profile’s radix is in Appendix C.2.

13 Batching

The path is half of a transition’s cost, about 516 vB of a 1,020 vB exit at $D = 16$, so batching is the question of how many leaves can share one. Read off the measured EXIT₁ and EXIT₂, a transition has a fixed part (skeleton, state binding, epilogue, control block) of about 355 vB paid once per transaction, 32 vB per level paid once per path, and about 148 vB per leaf touched (its 116 bytes, signature, payout and hashing script), which no shape removes; for one exit at $D = 16$ these sum to 1,020 vB. Batching therefore helps only leaves under one aligned subtree, which cohort grafts create.

The sweep. EXIT _{K} chains single-leaf steps, each with its full path, and saves only the skeleton (665 vB for a second exit). EVICT_SUBTREE _{h} is the real batch: the witness carries the path above an aligned subtree once and every position of the subtree, the script folds the subtree in post order checking each lease and evictor key, and folds the result and E_h up the shared path (158 vB per leaf for 51 leaves). It is the operator’s tool, and it needs a lease on every leaf it sweeps.

The cohort rewrite, REWRITE _{h} . Running the sweep’s fold over two subtrees gives the general cohort transition. For each position of an aligned subtree the witness carries an old and a new element (a leaf or the empty string); the old root, folded

from the old elements, must fold with the shared path to (r, S) ; the new root is folded from the new elements; output 0 is $(S - \sum \text{old} + \sum \text{new}, \text{TreeSPK}(r', m))$, an increase funded by other inputs. The owner of each changed leaf signs, except for a leaf whose only change is a larger amount, and the operator signs once. The rule that makes the rewrite a safe mass exit is the payout: a position that goes from a leaf to E_0 must be paid exactly its amount, to its exit key, from the next unused output. An owner who signs a transaction that empties its position is therefore paid by the covenant, and the operator cannot remove a leaf without that owner's signature and that payout; the owner multiproof of spec 13.4 (`spec/layertree.md`) is this special case, not a separate script. In one transaction a cohort can exit, join, resize, renew, rotate keys and pay between members with the sum conserved, with only the members whose leaves change taking part: the settlement step of an Ark round without a round, an expiry, or unchanged members online.

Figure 8 plots the measured costs; `REWRITE6` is 18,236 bytes of script. At $h = 6$ each position costs 261 bytes of script and 234 of witness for two leaf elements, or 118 when the new element is the one-byte unchanged marker; an owner signature adds 16 vB, and an exit adds its 43-byte output and drops the new element, so an unchanged position costs about 95 vB, a changed one 140 vB and an exiting one 154 vB, with 16 vB more per signed position under the exit-consent form. No shape lowers this per-leaf floor, and the unchanged positions' bytes make the rewrite a tool for cohorts that settle together. The Lean model covers it (`rewrite_sound`): the old and new subtrees are published, so a cohort that joins this way is grounded by construction and cannot be opaque, and a rewrite that empties the subtree pays out exactly what it held (`rewrite_exit_conservation`).

Caching the path, DETACH_h. Script cannot read a previous witness, so the upper levels of a cohort's path can only be cached in an output of its own. `DETACHh`, signed by the operator, proves an aligned subtree's node (R_h, s_h) against (r, S) , leaves E_h in output 0 for $S - s_h$, and creates output 1 as a depth- h tree with root R_h , amount s_h and a taptree m_h fixed at creation; members' paths truncated to the lower h levels are their new proofs (`detach_sound`, `detach_member_payable`). At $D = 16$, $h = 8$ it is 1,548 bytes of script and 497 of witness, about 780 vB, and every later exit from the detached tree costs the depth-8 figure, 762 vB instead of 1,020 vB, so it pays for itself at the fourth exit; a cohort of 256 that all leave alone saves about a quarter. The larger effect is throughput: the cohort's exits form their own chain of unconfirmed spends, so assumption [C] applies to it separately, and the operator can shard on demand when, say, channel leaves near their HTLC deadlines. Under spec 0.10.0, m_h must be compiled with the parent's creation options, so a detached channel leaf keeps exact payouts and the other parameters the composition conditions of §8 need; vouchers bind the lineage's `tree_id`, which the detached tree inherits, so they survive a detach; and channel leaves need nothing, as their `chan_id` binds the inherited `tree_id` and a detach leaves every leaf's bytes unchanged. Theorem 5 does not cover trees that carry `DETACHh` or `REWRITEh`.

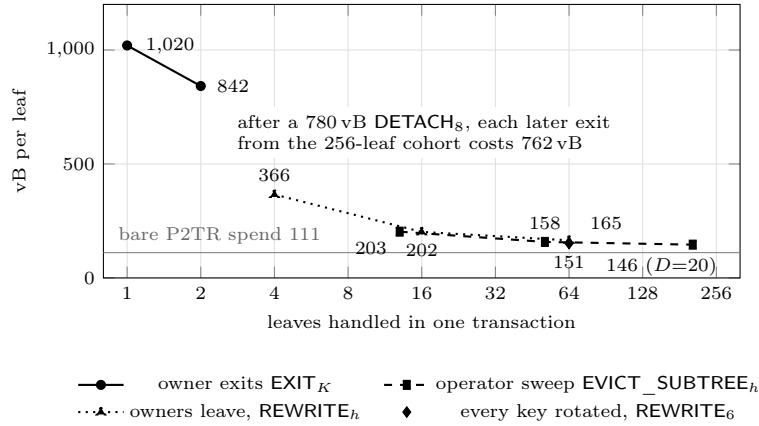


Fig. 8. Batch shapes on the model of §10, all measured at $D = 16$ except the 205-leaf sweep ($EVICT_SUBTREE_8$), measured at $D = 20$: vbytes per leaf that leaves or changes against the leaves one transaction handles. The detach is noted in the plot, not plotted, since its cost is per later exit.

14 Grafting and Whom It Trusts

$JOIN_h$ replaces an empty node E_h with a node (R_h, s_h) that the operator built off-chain from up to 2^h new leaves, and adds s_h to the tree output. Nothing below height h appears on-chain and the script walks only the $D - h$ levels above the slot, so a larger cohort is a smaller transaction (Table 7). Each height is a tapleaf fixed at creation.

What the covenant checks. The predicate requires that the slot held E_h , that output 0 carries $S + s_h$, and the operator’s signature. It sees a hash and a sum and cannot tell a real cohort from random bytes, which is what makes the graft’s size independent of the cohort’s and is the source of every trust question. `join_sound` states the consequences: the slot was empty, output 0 is the old sum plus s_h , and for every tree whose root is the grafted node the new root is the old tree with it inserted. A graft therefore cannot move, shrink or rewrite an existing leaf. A node with no preimage is an opaque node of $\mathcal{F}_{LT}$: its s_h , exactly the value the graft added, is frozen, every other leaf stays payable (Theorem 3, with Lemma 3 for a path past the opaque node), and the terminal exit is lost.

Who carries what risk. Existing members never sign and take no risk. The operator can fill an empty aligned slot with any node it signs and funds, so an opaque or zero-sum graft freezes only the value it brings and wastes only slots no one else could fill. Assurance is needed only by those who put money in. A member funding with an input checks the unsigned transaction (the slot folds as E_h to the current root, its leaf folds to (R_h, s_h) with sums recomputed, output 0 commits to the new root for $S + s_h$) and signs over all outputs, so its input can

Table 7. Entry cost per member at $D = 16$ on the model of §10, with no payout output. JOIN_0 and JOIN_2 are measured; JOIN_4 and JOIN_8 are the measured JOIN_0 with the same number of levels ($D = 12$ and $D = 8$), and JOIN_{10} is JOIN_8 less two levels (estimate). Per member is operator-funded; member-funded adds each member’s P2TR key-path input (58 vB) and the signing round of §4.4.

graft	members	tx (vB)	per member	member-funded
JOIN_0	1	937	937	≈ 995
JOIN_2	4	872	218	≈ 276
JOIN_4	16	808	51	≈ 109
JOIN_8	256	679	2.7	≈ 61
JOIN_{10}	1,024	614	0.60	≈ 59

only be spent into a root containing its leaf. A member buying in off-chain checks the same path against the confirmed root before paying over Lightning, or pays through a hash-locked voucher (§15). Neither checks anyone else’s leaf: leaves whose paths verify sum to at most the root, so a co-member’s bad leaf cannot make a good one unpayable.

Funding. In an operator-funded graft the operator supplies s_h from the input the model already counts and members buy their balances off-chain; the operator bears the credit risk of a member who never pays, a risk to it and not to other members. In a member-funded graft each member adds an input and signs one transaction after the cohort is fixed, so one absent member stalls the cohort and the rest retry (a liveness cost, not a safety one), and the saving against an on-chain batch, 43 vB per output, is gone. Had the operator grafted a node with no tree behind it, every member’s check would fail, nobody would pay, and only the operator’s own coins would be frozen.

Designing a cohort. The slot the operator picks is what later batching depends on, and a cohort should share a lease, since one leaf the operator admits with a NUMS evictor or a later lease blocks the sweep of its subtree (a member cannot create one by carving, which keeps the evictor and the lease, spec 0.10.0). REWRITE_h is a graft that publishes its leaves: grounded by construction, at about 95 vB per unchanged position of the rewritten subtree (measured at $h = 6$, §13) against under 3 per member for JOIN_8 . A graft also settles off-chain credit as exitable leaves (§15).

15 Receiving over Lightning

A user with no on-chain funds but a fee coin receives a Lightning payment and wants a balance it can leave with alone. An Ark client gets an out-of-round output co-signed by the operator that expires with its round, or, below the operator’s floor, a ledger credit with no on-chain claim. The tree takes the same payment with less trust, through a *cumulative* voucher (Appendix C.3): since a redemption advances the source leaf’s seq and voids every other voucher against it, repeated

receives are one voucher that the issuer re-signs for the running total against the same claim leaf. The recipient keeps the latest and needs no contest window, since it never prefers a smaller state. The covenant stops the issuer from redeeming more than the source holds; it does not stop over-issuance or voiding (advancing seq, or redeeming an older, smaller voucher, before the recipient redeems). Both are closed by the source leaf's shape.

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 (§8).

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, in the FULL tier, the only one with CLAIM^x . The operator issues the invoice and draws its preimage, so the payer's receipt proves that the operator was paid, not the user; a user who wants a receipt naming it hands the operator preimages in advance, and can then receive offline only as many payments as it has pre-shared. The payer's HTLC reaches the operator, which settles it *first*; only then does it sign the user a voucher for the new running total, a plain cumulative voucher against the credit leaf. Nothing reaches the chain per payment and the user need not be online, but a voucher draws only what the credit leaf holds: the operator

pre-funds it, with capital it recovers only with the user's consent, or tops it up as credit accrues, about 122 vB per credit leaf in a cohort REWRITE_h^x [35], which voids no voucher. The order matters. A hash-locked voucher issued before the HTLC settles, with the preimage held by the recipient, breaks the rule that the party the HTLC pays draws the preimage (spec 9.6.2), and on a credit leaf it cannot be repaired: if the HTLC then fails, retiring the voucher means advancing the credit leaf's seq, which needs the recipient's exit key, so the voucher stays redeemable for a payment that never arrived. 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 (§15): 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 places a leaf of the recipient's own in its next cohort graft, at under 3 vB per member in a 256-member JOIN_8 , and retires the voucher by advancing the credit leaf's seq and lowering its amount by what it has paid out: one UPDATE^x per credit leaf (1,048 vB), which the recipient co-signs, or one REWRITE_h^x for a cohort of credit leaves, about 151 vB per changed leaf at $h = 6$ plus 16 vB for each exit-key signature. If the operator stops cooperating, the recipient redeems its latest voucher alone with one CLAIM^x : 1,636 vB at $D = 16$, with no timelock, against 1,311 to 2,511 vB and a relative timelock to unroll an Ark round tree of 256 to 65,536 members (§10). A carve conserves the tree's total and the exact exit pays in full, so both fees come from a confirmed coin of the recipient's: with the exit, 2,656 vB, or 26,560 sat at 10 sat/vB, below which the fallback costs more than it recovers (Appendix C).

Where one operator leaf backs many recipients instead, over-issuance is equivocation with a compact proof, which a slashable bond can deter; voiding has no compact proof, so no bond reaches it.

Atomic receive with pending leaves. Settling first still leaves the recipient exposed for one payment. The tree closes the gap with three added script variants, none needing an opcode outside its tier's set. *R1*: a leaf whose evictor is a separate reclaim key can, after its deadline, be removed by the operator alone with no payout. *R2*: a hash-locked voucher (CLAIM^{hm}) that raises the source leaf's sibling to an absolute level and lowers the source by the difference without advancing its seq; the sibling must be owned by the source's exit key and exit to it. *R3*: a REWRITE_h that moves value from a reclaimable leaf into that sibling with the operator's signature alone. A recipient holds a pair, a pending leaf the operator owns, naming the recipient's key as exit key and the reclaim key as evictor, next to the recipient's own landing leaf. The operator signs the merge voucher before the recipient releases the preimage. The voucher can only move value from the pending leaf into a leaf the recipient alone can exit, and the level it names is absolute, so an older voucher redeemed after a newer moves nothing and one voucher covers a whole window. A window ends in one of three ways: the operator settles it with an R3 rewrite, a tower redeems the voucher before the deadline,

or the operator reclaims the pending leaf after it (R1) and only then fails any inbound HTLC whose voucher is outstanding. The recipient's liveness duty is one check of its landing leaf per window, which a tower performs holding no key of the recipient's. The model of Appendix A proves that a merge or a transfer moves value only between leaves that exit to one key, that a reclaim removes only flagged leaves past their deadline, and that replaying a voucher moves nothing. The credit leaf remains the choice for a wallet that accepts one payment of trust and wants no deadline.

The parties of the pending-leaf receive. The recipient and the operator run Algorithm 11 for each payment; the operator runs Algorithms 12 and 14 at the close of each window, and the recipient's tower runs Algorithm 13.

Algorithm 11 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  $E \geq T + \Delta(q) + \text{time to settle from an on-chain preimage}$ 
3 :  $O$ :  $X \leftarrow \text{settled level of } L + a$ 
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 :    $X = \text{settled level of } L + a$ ,  $P.\text{amt} + L.\text{amt} - X' \geq \text{DUST\_MIN}$  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
```

This is the construction of spec 9.6.3. The recipient holds a *receive pair*: a pending leaf P at an even position, owned by the operator, whose exit key is the recipient's, whose evictor is RECLAIM_KEY and whose deadline is T , and its sibling, the landing leaf L , which the recipient owns and exits to. Line 1 has the recipient draw the preimage, the party the HTLC pays, as the ordering rules of **Swap** require. Line 2 is rule 4 of the spec: T must be chosen so that the HTLC cannot expire before the operator learns a preimage published late on-chain. Lines 3 to 5 are rule 1: the headroom is required, since a voucher that drains P exactly would be blocked by any amount-only top-up, which leaves a sub-dust residual that Φ_{CLAIM} check 11 rejects, after which the operator could reclaim the payment at T . Line 6 signs the merge voucher of Φ_{CLAIM} (checks 9 to 12): it raises L to the absolute level X and lowers P by the difference without advancing P 's seq, so it voids nothing and an older, lower voucher is a no-op after a newer one. Lines 7 to 10 are the recipient's verification of rule 2, against the *confirmed* tree and before it releases anything, which also excludes a pending leaf that is not L 's sibling or an L that the operator could evict inside the window; line 11 is the release rule,

one preimage at a time per pair, because the targets are levels of one leaf and a second voucher that included an unsettled payment would hand the recipient that payment without its preimage. Before line 11 the operator has signed nothing it can lose, since V needs a preimage that only R holds and publishing it on-chain hands it to O ; after it R can redeem alone. The spec fixes none of the invoice's expiry arithmetic beyond rule 4 (design/pending-receive.md derives a minimum final delta in terms of the window length). Line 1 is the operator's duty under

Algorithm 12 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

```

rule 3 of spec 9.6.3: before $T - \Delta(q) - \varepsilon$ it should raise L to the level the recipient has settled in the window, so that the recipient's vouchers move only what is still in flight above it. The two ways of lines 2 to 4 are the voucher the operator signs for itself, which needs no one else, and the transfer of the reclaim form of the rewrite (spec 7.9 rule 8), which moves value from P into the sibling that the same key owns and exits to and needs the operator's signature alone, so a cohort of recipients settles in one transaction without any of them signing. Pairs that received nothing cost nothing. The operator's promise is a SHOULD, because the recipient does not rely on it: if L has not reached the settled level by $T - \Delta(q) - \varepsilon$, the recipient or its tower MUST redeem the latest voucher (Watch). Lines 5 and 6 are rule 4 and a MUST: until the reclaim confirms, the recipient can still redeem and publish the preimage, and an HTLC already failed back would leave the operator paying for a payment it never received. The spec leaves open the length of a window, how many pairs a recipient holds for payments across a deadline, and how the operator learns which vouchers are outstanding. This is the one liveness duty of the recipient (spec 9.6.3 rule 3, which puts the

Algorithm 13 Watch – tower holding (V, ρ) for a pair (P, L)

```

1 : on every spend of the tree output: Follow the paths and the bytes of  $P$  and  $L$ 
2 : at height  $T - \Delta(q) - \varepsilon$  at the latest:
3 :   if  $L.\text{amt} < \text{the settled level (confirmed)}$ :
4 :     build  $\Phi_{\text{CLAIM}}$  (merge form) from the latest  $V$  and  $\rho$ , own fee input
5 :     broadcast; if a conflicting spend confirms, rebuild as in Exit
6 : stop once  $L.\text{amt} \geq X$  or the reclaim of  $P$  has confirmed

```

check on the recipient *or* a tower holding the voucher and preimage): once per window, at $T - \Delta(q) - \varepsilon$ (line 2), confirm that L holds at least the settled level and, if it does not, redeem the latest voucher with the preimage. A tower can do it because the merge voucher moves value only into a leaf the recipient alone can exit and needs no signature beyond the voucher’s (spec 7.5.4; design/towers.md); the tower holds no key of the recipient’s. Line 1 is **Follow**, with the leaf bytes, since a merge voucher binds L ’s body; lines 3 to 5 are the redemption, within the bound $\Delta(q)$ of spec 10.5. The spec gives no tower protocol (registration, payment, handing over a voucher) or incentive (design/towers.md). Line 1 is the

Algorithm 14 Reclaim – operator, taking back a pending leaf after its deadline

```

1 : when lock  $\geq P.\text{ea} = T$  for a pending leaf  $P$  with  $P.\text{ek} = \text{RECLAIM\_KEY}$ :
2 :    $\text{tx} \leftarrow \Phi_{\text{RECLAIM}}$ , or the reclaim branch of  $\Phi_{\text{REWRITE}}$  replacing  $P$  by the
3 :     next window’s pending leaf
4 :   sign with RECLAIM_KEY for  $\Phi_{\text{RECLAIM}}$ , OPERATOR_KEY for the rewrite
5 :   broadcast; rebuild on a conflicting spend
6 : only after the reclaim has confirmed: fail upstream the HTLCs not settled

```

deadline of Φ_{RECLAIM} (checks 2 and 3): a leaf whose evictor is RECLAIM_KEY is reclaimable, and after T the operator alone removes it with no payout, so that value in P at its deadline is the operator’s, which is what makes the timeout of the construction work. The batch form of lines 2 and 3 shares one transaction among many pairs and writes the next window’s pending leaf in the same rewrite (spec 7.9 rule 8). A reclaimable leaf is never swept by $\Phi_{\text{EVICT_SUBTREE}}$, whose evictor must be OPERATOR_KEY, and an ordinary eviction of one still pays its exit key. Line 5 is rule 4 of spec 9.6.3: only after the reclaim confirms may the operator fail an inbound HTLC whose merge voucher is outstanding, since until then the recipient can still redeem and publish the preimage. The spec does not say how the operator chooses between the two forms of line 2.

16 Related Work

Payment pools. CoinPool [29] is a shared UTXO with per-user balances whose one-transaction withdrawal rests on a new opcode, OP_MERKLESUB; OP_TAPLEAF_UPDATE_VERIFY [47] recreates a pool output with the leaver’s key and leaf removed, and OP_EVICT [52] 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 lease-based eviction, cohort grafts, channel leaves and a composable statement. *Transaction trees.* Channel factories [8] need all N members to sign every change. Ark [23], deployed today [1], batches users into rounds whose outputs are leaves of a pre-signed MuSig2 tree; covenant Ark replaces the signing with CTV templates [40] but keeps the tree, its unrolled exit and its expiry. Timeout trees [24] and SuperScalar [53] add laddered expiries and Decker-Wattenhofer factories; MATT-style trees [19,20]

replace signatures by templates. Table 11 (Appendix C) compares them. *Beyond Bitcoin*. Plasma’s forced exits [38] prove a balance against a committed root. Proofs of liabilities [21,12] use Merkle-sum trees, whose per-child sums block the inflation a total-only node admits. None of these states its guarantees as a functionality; we build on the ledger model of [3] and the channel functionalities of [2,17]. *Components*. The Merkle-sum sparse Merkle tree is Taproot Assets’ [33], truncated and verified in script; the data-tweaked key follows BIP-443 [20]; the generator-key signature is Poelstra’s [37], and its ECDSA analogue has been used on Bitcoin Cash as `OP_PUSH_TX`. Liquid’s tapscript [7] has `OP_CAT`, CSFS, per-field introspection and 64-bit arithmetic, so MID is close to expressible there; Simplicity [32] would express the predicates directly.

17 Discussion and Limitations

The functionality is deliberately modest, and its liveness more so: all activity is serialised through one output, so an honest exit’s delay is set by policy and fees, which [C] names rather than derives. Cluster mempool [13] (Bitcoin Core 31) replaced the 25-transaction chain limit with a 64-transaction cluster limit, which raises κ for a cooperative tree but shares the cluster with every descendant anyone attaches. Two changes were weighed against it. A relative lock on transitions that remove no value, so that only removals could chain, was declined: the first conflicting spend in a block already invalidates an honest exit, so the lock would cost throughput and remove nothing. The tower exit EXIT_K^t (spec 7.2 rule 7) was adopted: a signature over the leaf without its amount and the tree’s identifier, checked from the stack, lets a tower rebuild and rebroadcast an exit with the owner offline, which takes the owner’s liveness out of the auction without changing its price. A hypothetical native Merkle-sum opcode (simulator only, in no tier, not proposed here) takes EXIT_1 to 252 B of script at every depth and the exit to about $381 + 10k$ vB for 2^k members (§10).

18 Conclusion

LayerTree specifies a shared output by what it lets members do (join, leave alone, be evicted after a lease, be swept as a cohort) and realises it with a tree kept off-chain as data: a one-transaction exit of about a kilobyte, no pre-signing, no expiry of the shared output, from proposed opcodes; its price is that all activity funnels through one UTXO. Future work: a verified compiler, a proof of Conjecture 1, extending the mechanised path maintenance of Lemma 3 to opaque nodes, and a P model [16].

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. Ark Labs: arkd: Backend services powering Arkade. Repository, <https://github.com/arkade-os/arkd> (2023)
2. Aumayr, L., Ersoy, O., Erwig, A., Faust, S., Hostáková, K., Maffei, M., Moreno-Sanchez, P., Riahi, S.: Generalized channels from limited blockchain scripts and adaptor signatures. In: ASIACRYPT 2021. LNCS, vol. 13091, pp. 635–664. Springer (2021)
3. 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)
4. 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)
5. 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)
6. Black, B., Rubin, J.: BIP-348: CHECKSIGFROMSTACK. <https://github.com/bitcoin/bips/blob/master/bip-0348.md> (2024)
7. Blockstream: Elements tapscript opcodes: Introspection, streaming SHA256, 64-bit arithmetic, ECMULSCALARVERIFY, TWEAKVERIFY. https://github.com/ElementsProject/elements/blob/master/doc/tapscript_opcodes.md (2021)
8. Burchert, C., Decker, C., Wattenhofer, R.: Scalable funding of bitcoin micropayment channel networks. In: Royal Society Open Science. vol. 5 (2018)
9. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: FOCS 2001. pp. 136–145. IEEE (2001)
10. Canetti, R.: Universally composable signature, certification, and authentication. In: CSFW 2004. pp. 219–233. IEEE (2004)
11. Canetti, R., Dodis, Y., Pass, R., Walfish, S.: Universally composable security with global setup. In: TCC 2007. LNCS, vol. 4392, pp. 61–85. Springer (2007)
12. 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)
13. Daftuar, S.: An overview of the cluster mempool proposal. Delving Bitcoin, <https://delvingbitcoin.org/t/an-overview-of-the-cluster-mempool-proposal/393> (2024)
14. Decker, C., Russell, R., Osuntokun, O.: eltoo: A simple layer2 protocol for bitcoin. <https://blockstream.com/eltoo.pdf> (2018)
15. Decker, C., Towns, A.: BIP-118: SIGHASH_ANYPREVOUT for taproot scripts. <https://github.com/bitcoin/bips/blob/master/bip-0118.mediawiki> (2017)
16. 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)
17. Dziembowski, S., Ekey, L., Faust, S., Malinowski, D.: PERUN: Virtual payment hubs over cryptocurrencies. In: IEEE S&P 2019. pp. 106–123. IEEE (2019)
18. Heilman, E., Sabouri, A.: BIP-347: OP_CAT in tapscript. <https://github.com/bitcoin/bips/blob/master/bip-0347.mediawiki> (2023)
19. Ingala, S.: Merkleize all the things. bitcoin-dev mailing list, <https://merkle.fun> (2022)
20. Ingala, S.: BIP-443: OP_CHECKCONTRACTVERIFY. <https://github.com/bitcoin/bips/blob/master/bip-0443.mediawiki> (2025)

21. Ji, Y., Chalkias, K.: Generalized proof of liabilities. In: ACM CCS 2021. ACM (2021)
22. Katz, J., Maurer, U., Tackmann, B., Zikas, V.: Universally composable synchronous computation. In: TCC 2013. LNCS, vol. 7785, pp. 477–498. Springer (2013)
23. 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)
24. Law, J.: Scaling lightning with simple covenants. bitcoin-dev mailing list, <https://github.com/JohnLaw2/ln-scaling-covenants> (2023)
25. 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)
26. Lightning Network specification: BOLT #3: Bitcoin transaction and script formats. <https://github.com/lightning/bolts/blob/master/03-transactions.md> (2024)
27. Merkle, R.C.: A digital signature based on a conventional encryption function. In: CRYPTO 1987. LNCS, vol. 293, pp. 369–378. Springer (1988)
28. de Moura, L., Ullrich, S.: The Lean 4 theorem prover and programming language. In: CADE 2021. LNCS, vol. 12699, pp. 625–635. Springer (2021)
29. 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)
30. Nick, J., Ruffing, T., Jin, E.: BIP-327: MuSig2 for BIP340-compatible multi-signatures. <https://github.com/bitcoin/bips/blob/master/bip-0327.mediawiki> (2022)
31. Nick, J., Ruffing, T., Seurin, Y.: MuSig2: Simple two-round Schnorr multi-signatures. In: CRYPTO 2021. LNCS, vol. 12825, pp. 189–221. Springer (2021)
32. O’Connor, R.: Simplicity: A new language for blockchains. PLAS 2017, <https://blockstream.com/simplicity.pdf> (2017)
33. 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)
34. Osuntokun, O.: Elliptic curve operations for Bitcoin script. Draft BIP, bitcoin/bips PR #1945 (closed unmerged), <https://github.com/bitcoin/bips/pull/1945> (2025)
35. Osuntokun, O.: LayerTree: Specification, reference implementation and formal model. <https://github.com/roasbeef/layertree> (2026)
36. 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)
37. 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)
38. Poon, J., Buterin, V.: Plasma: Scalable autonomous smart contracts. White paper, <https://plasma.io/plasma.pdf> (2017)
39. Poon, J., Dryja, T.: The bitcoin lightning network: Scalable off-chain instant payments. <https://lightning.network/lightning-network-paper.pdf> (2016)

40. Roose, S.: The Ark case for CTV. Delving Bitcoin, <https://delvingbitcoin.org/t/the-ark-case-for-ctv/1528> (2025)
41. Rubin, J.: BIP-119: CHECKTEMPLATEVERIFY. <https://github.com/bitcoin/bips/blob/master/bip-0119.mediawiki> (2020)
42. Rubin, J.: BIP-449: OP_TWEAKADD, x-only key tweak addition. <https://github.com/bitcoin/bips/blob/master/bip-0449.md> (2026)
43. Russell, R.: OP_TX. Draft BIP on a personal fork, <https://github.com/rustyrussell/bips/blob/guilt/varops/bip-unknown-optx.mediawiki> (2024)
44. Russell, R., Moik, J.: BIP-440: Varops budget for script runtime constraint. <https://github.com/bitcoin/bips/blob/master/bip-0440.mediawiki> (2025)
45. Russell, R., Moik, J.: BIP-441: Restoration of disabled script (tapleaf 0xC2). <https://github.com/bitcoin/bips/blob/master/bip-0441.mediawiki> (2025)
46. Sanders, G.: LN-Symmetry (eltoo) transaction formats, draft. https://github.com/instagibbs/bolts/blob/eltoo_draft/XX-eltoo-transactions.md (2023)
47. 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)
48. Wuille, P., Nick, J., Ruffing, T.: BIP-340: Schnorr signatures for secp256k1. <https://github.com/bitcoin/bips/blob/master/bip-0340.mediawiki> (2020)
49. 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)
50. Wuille, P., Nick, J., Towns, A.: BIP-342: Validation of taproot scripts. <https://github.com/bitcoin/bips/blob/master/bip-0342.mediawiki> (2020)
51. Zhao, G.: BIP-431: Topology restrictions for pinning. <https://github.com/bitcoin/bips/blob/master/bip-0431.mediawiki> (2024)
52. 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)
53. 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 The Lean Formalisation

This appendix describes the mechanised model behind §6 in enough detail that a reader can judge what has been proved, under what assumptions, and what has not. The development is about 4,500 lines of Lean 4 [28] in eleven modules and depends on nothing but the core library: no Mathlib, no external tactics. It builds in seconds and prints an axiom audit for 72 headline theorems; as `lake build` prints it, every theorem lists a subset of `propext`, `Quot.sound` and `Classical.choice`, and none lists `sorryAx`. The code is in the artefact [35].

The appendix is organised as the Lean development was read against the paper. Section A.1 lists the headline theorems and states the main ones; §A.2 says what the model keeps and what it drops; §A.3 maps every predicate box, algorithm, lemma and theorem of the paper to its Lean names and shows the key definitions; the remaining sections give the assumption ledger, the module sizes, the binding to the output, the encodings and what is not proved.

A.1 Headline Theorems

The audit file `Axioms.lean` prints one line for each of the 72 headline theorems, in eight groups.

group	module	# what the theorems say
Tree	Tree	4 roots bind trees (<code>root_inj</code>); a node and path that fold to the root are a subtree and its honest path (<code>fold_sound</code>); a member's path for the next root is its old path with one entry replaced (<code>path_maintenance</code> , <code>path_maintenance_valid</code>)
Transitions	Transitions	21 a step is a subtree replacement; steps conserve sums with no hash assumption; the honest exit witness is valid; <code>EXIT_K</code> pays within the exit rule; <code>JOIN</code> , <code>UPDATE</code> and <code>CLAIM</code> are sound; the terminal branch; the sparse profile
Cohort	Cohort	14 sweep, <code>REWRITE</code> (with the reclaim form), <code>DETACH</code> , merge and reclaim are sound and conserve value
Trace	Trace	9 along a history the books balance, leaves stay well formed, above the dust floor and evictable, and every on-chain root is that of a reachable tree
Voucher	Voucher	6 a voucher cannot be replayed and survives a top-up and a detach; a final leaf is never carved
Receive	Receive	6 a merge changes no body, raises its target monotonically, cannot be replayed and cannot overpay; value held for one exit key never decreases
Binding	Binding	2 a witness root and taptree that pass the scriptPubKey check are the committed ones, in <code>FULL</code> and <code>CCV</code>
Encoding	Encoding	10 serialisations are injective; <code>enc64</code> is the minimal encoding; the <code>SEQ_FINAL</code> and <code>PADZ</code> facts

The statements of the main ones follow, from the development, with the implicit arguments the elaborator infers omitted (and some explicit binders elided in `join_sound`, `claim_sound`, `rewrite_sound` and `detach_sound`); the types they use (`Tree`, `Pos`, `fold`, `HashModel`) are in §A.2.

`t.root` is the root node of a tree, `t.getSub p` the subtree at position `p`, `t.setSub p u` the tree with that subtree replaced by `u`, `t.pathSub p` the honest sibling path, and `t.get`, `t.set`, `t.path` their leaf-level specialisations.

```
theorem fold_pathSub (t : Tree d) (p : Pos d h) :
  fold (t.getSub p).root (t.pathSub p) = t.root
```

Note that the conclusion of `fold_sound` equates *nodes*, so the sum the prover presented is the true sum of the subtree.

A **Step** is a path, a height, an old node and a new node; it is valid for roots r, r' when the old node folds to r and the new node to r' , and well formed for depth d when the path has $d - h$ entries. **Steps** chains steps, each against the root the previous one produced.

```
theorem Step.sound (t : Tree d) (s : Step H) (r' : Node H)
  (hwf : s.WF d) (hv : s.valid t.root r') :
  ∃ p : Pos d s.height, s.old = (t.getSub p).root ∧
    ∀ (u : Tree s.height), u.root = s.new → r' = (t.setSub p u).root
```

Steps.sum is Theorem 2 and needs no hash assumption. **Steps.sound** is Theorem 4; **s.Grounded** says the new node is the root of some tree of the step's height, which leaf-level steps satisfy by construction (`grounded_of_leaf`) and which a **JOIN** graft must be given.

The transitions are structures bundling the steps with the output rule and an opaque authorisation. Their theorems:

```
theorem ExitK.exact_conservation (h : ExitK true r blocks r' out0) :
  out0 + payoutSum blocks = r.sum

theorem claim_sound (t : Tree d) (hwf1 : srcPath.length = d)
  (hwf2 : claimPath.length = d)
  (hc : Claim auth  $\pi$  t.root srcPath claimPath src claim r' out0) :
  out0 = t.root.sum  $\wedge$ 
   $\exists$  (p1 p2 : Pos d 0), t.get p1 = some src  $\wedge$ 
  (t.set p1 (claimedSource src claim.amount)).get p2 = none  $\wedge$ 
  r' = ((t.set p1 (claimedSource src claim.amount))
    .set p2 (some claim)).root
```

payable is Theorem 3. join_sound uses root_inj (via eq_empty_of_root): since JOIN_h hard-codes E_h as the old node, the slot it fills is provably the empty tree, and the theorem gives the successor root for every tree whose root is the grafted node. claim_sound exposes the drained-source case: the claim position must be empty in the tree *after* the source step, which is why a source reduced to zero may host the claim.

The terminal branch of Final and the sparse profile have their own statements: terminal_no_leaf (a tree whose root is E_D holds no leaf, so the terminal branch is reachable only when the last leaf leaves), ExitKT.terminal_conservation (on that branch the payouts are at most S , exactly S under exact payouts), Tree.Sparse.unique (two leaves with the same owner key and nonce are the same leaf at the same position) and sparseJoin_sound (a sparse join lands at the position derived from the leaf, which was empty, and keeps the tree sparse).

Cohort transitions. The sweep, REWRITE_h and DETACH_h are one step each from a subtree root at a position $p : \text{Pos } d \ h$; a position inside it is $p.\text{append } q$. The sweep and the rewrite take the published subtrees as values of Tree h , since the witness carries their 2^h elements and leaves_inj says those name one tree, so neither needs a Grounded hypothesis. In Rewrite, authorisation is a predicate required at every position whose element changes, and the payouts are those of the leaves replaced by E_0 , in position order:

```
theorem rewrite_sound (t : Tree d) (hwf : path.length + h = d)
  (hr : Rewrite h  $\pi$  auth lockTime t.root path old new r' out0 payouts) :
   $\exists$  p : Pos d h, t.getSub p = old  $\wedge$  r' = (t.setSub p new).root  $\wedge$ 
  ( $\forall$  q : Pos h 0, t.get (p.append q) = old.get q  $\wedge$ 
    (t.setSub p new).get (p.append q) = new.get q)  $\wedge$ 
  ( $\forall$  q : Pos d h, q  $\neq$  p  $\rightarrow$  (t.setSub p new).getSub q = t.getSub q)  $\wedge$ 
  out0 + old.total = t.root.sum + new.total  $\wedge$ 
  out0 + payoutTotal payouts  $\leq$  t.root.sum + new.total

theorem detach_sound (t : Tree d) (hwf : path.length + h = d)
  (hd : Detach auth h t.root path sub r' out0 r1 out1) :
   $\exists$  p : Pos d h, out0 + out1 = t.root.sum  $\wedge$ 
  r' = (t.setSub p (Tree.empty h)).root  $\wedge$ 
  r1 = (t.getSub p).root.hash  $\wedge$  out1 = (t.getSub p).total  $\wedge$ 
   $\forall$  q : Pos h 0, (t.getSub p).get q = t.get (p.append q)  $\wedge$ 
  (t.getSub p).path q = (t.path (p.append q)).drop (d - h)
```

rewrite_exit_conservation specialises rewrite_sound to an emptied subtree: output 0 plus the payouts equals the input amount. detach_member_payable applies payable inside the detached tree to the truncated path. evictSubtree_sound is the

sweep's analogue of `rewrite_sound` with the new subtree empty, on either Final branch. `Detach` states its second tree output as a root hash and an amount rather than generalising Step: the detached output's taptree m_h is a binding fact, outside the step.

The pending-leaf receive. Params carries the optional reclaim key, and a leaf is Reclaimable when its evictor is that key; a tree without a reclaim key has no reclaimable leaf, and every rule reduces to its form before the receive. Merge (CLAIM^m , CLAIM^{hm}) is one step at height 1 over a sibling pair, and Reclaim (RECLAIM_1) one leaf-level step to E_0 with the deadline compared against a lock-time parameter:

```

theorem merge_sound (t : Tree d) (path : List (Bool × Node H))
  (b : Bool) (src tgt : Leaf) (X : Nat) (r' : Node H) (out0 : Nat)
  (hwf : path.length + 1 = d)
  (hm : Merge auth π t.root path b src tgt X r' out0) :
  out0 = t.root.sum ∧ tgt.ownerKey = src.exitKey ∧
  tgt.exitKey = src.exitKey ∧ tgt.amount < X ∧
  (Tree.tip (mergedSource src (X - tgt.amount))).total + X
  = src.amount + tgt.amount ∧
  ∃ c : Pos d 1, t.get (c.pairAt b) = some src ∧
  t.get (c.pairAt (!b)) = some tgt ∧ ...

theorem reclaim_sound (t : Tree d) (path : List (Bool × Node H))
  (l : Leaf) (r' : Node H) (out0 : Nat) (hwf : path.length = d)
  (hc : Reclaim auth π lockTime t.root path l r' out0) :
  ∃ p : Pos d 0, t.get p = some l ∧ π.Reclaimable l ∧
  l.evictAfter ≤ lockTime ∧ r' = (t.set p none).root ∧
  (∀ q : Pos d 0, q ≠ p → (t.set p none).get q = t.get q) ∧
  out0 + l.amount = t.root.sum

```

The tail of `merge_sound` is elided; it says the successor tree is `mergeTree`, in which the source is lowered by the draw, with every field but its amount unchanged and its `seq` included, and the target is raised to exactly X . `reclaim_only_flagged` adds that every leaf that is not reclaimable survives a reclaim and that a tree with no reclaim key accepts none. `rewrite_transfer_sound` is the analogue for the reclaim form of REWRITE_h (spec 7.9 rule 8): an amount-only decrease of a reclaimable leaf at an even position is absorbed exactly by its sibling, which the same key owns and exits to, and `rewrite_reclaimable_change` says a reclaimable leaf otherwise changes only by an amount-only increase or after its deadline, never needing its owner.

Three theorems state the properties the construction was designed for. `merge_stale_rejected`: if the target already holds X or more, no merge to X is accepted, so replaying a voucher, or redeeming an older and lower one after a newer, moves nothing; `merge_target_monotone` gives the accepted case, where the target's level rises to exactly X . `merge_body_preserved`: a merge changes no leaf's body, so it voids no voucher against its source. `merge_no_overpay`: along any history, merges into one target move at most the largest level signed for it plus what merges move out of it, under the hypothesis that nothing but a merge lowers the target (the operator's duty to advance the target's `seq` in any decrease it co-signs, which the scripts do not check), and every such merge moves value between leaves exiting to one key. `merge_cycle_overpays` is the case the bound allows for, a cycle between two leaves of one key. The first version of the rules

required the merge target to be owned by the source’s exit key but not to exit to it, and the model showed two consequences: value could leave a key’s exits through a leaf it owned that exits elsewhere, and two leaves each owned by the other’s exit key could merge back and forth. Both rules now require equal exit keys, after which `Reachable.exit_value` holds: along any history of merges and reclaims, the amount held in leaves that are not reclaimable and exit to one key never decreases.

Histories. `Trace.lean` composes the transitions along a whole history. `Trans` is a transition record in the tree’s vocabulary, `Trans.apply` the tree it produces and `Trans.Pre` what the script requires of the current tree; `Chain` is a history of accepted spends. `Chain.sound` is the bridge:

```
theorem Chain.sound (t : Tree d) (h : Chain  $\pi$  d t.root r') :
   $\exists$  (xs : List (Trans d)) (t' : Tree d),
    r' = t'.root  $\wedge$  Reachable  $\pi$  t xs t'

theorem Reachable.ledger (h : Reachable  $\pi$  t xs t') :
  t'.total + removedSum xs t = t.total + addedSum xs t
```

Every root that a chain of accepted spends reaches from the root of a real tree is the root of a tree reachable by the transition records, and the books balance over the records: merges move nothing in or out, and a reclaim removes its leaf’s amount with no payout. `Reachable.dust` and `Reachable.evictable` carry the dust floor and the evictability of every leaf through a history, `Reachable.wf` well-formedness, and `chain_state` the statement that every on-chain root is the root of a reachable tree whose every leaf has an accepted exit.

A.2 Model Abstractions

The model keeps the rules and drops the machinery that carries them out. It states the *rules* of §4.3 as relations over the objects the scripts manipulate (leaves, nodes, positions, sibling paths, amounts), with one structure per transition whose fields are the checks of that transition’s predicate box, in protocol order, and it composes the transitions along histories. What it abstracts, following `formal/README.md`:

- *Tiers and bytecode.* There is no model of tapscript, of BIP-441 numerics, of `OP_TX` or of the BIP-341 message. The four tiers (FULL, MID, MIN, CCV) differ in how a script computes a rule, not in the rule, so the model states each rule once; the tier-dependent constants are parameters of `Params` (`seqFinal`, `dustMin`, the optional `reclaimKey`). That a script implements the relation is established by execution (§10), not by proof.
- *Authorisation.* Signatures, lock times and the hash lock are an opaque proposition `auth` of each transition (a predicate on positions for `Rewrite`). The variants that differ only in what is signed, `EXITt`, `UPDATEx`, `CLAIMh`, `CLAIMx` and `SPLIT`, are therefore the same relation in the model, and the theorems hold for each. The lock time is a parameter that the deadline checks compare against, not a height that grows along a chain.

- *Transaction fields.* The input and output amounts the scripts read with `OP_TX` are parameters (`out0`); a payout is the pair of key and amount that output i carries, with its key required to be the leaf’s exit key (`ExitBlock.payoutOk`).
- *Paths and positions.* Depth is a type index and a position is `Pos d h`; a sibling path is *root-first*, the reverse of the witness, which supplies the deepest pair first. In the sparse profile the position is an arbitrary function of the owner key and nonce; the hash that derives it is not modelled.
- *Hashes.* Parameters of a `HashModel`: collision resistance as injectivity, tag separation as inequalities. Nothing quantifies an adversary’s advantage. Sums are unbounded naturals; the 2^{64} range belongs to the encodings.
- *What is outside.* Concurrency, mempool policy, reorganisations and the off-chain protocol are not theorems about hash trees; the party algorithms of §4.4 are not formalised, only the facts about the tree and its histories that their lines rely on.

The types. The definitions below are those the theorems of §A.1 are stated over.

The model states the *rules* of §4.3 as relations over the objects the scripts manipulate: nodes, paths, amounts. It does not model tapscript bytecode, signatures or keys, and a lock time is a parameter, not a block height that grows along a chain. Authorisation is carried as an opaque proposition, and the input and output amounts the scripts read with `OP_TX` are parameters. Three choices keep the proofs short.

Depth as a type index. A tree of depth d is `Tree d`, a perfect binary tree whose tips are `Option Leaf`. Positions are `Pos d h`, paths from depth d down to a subtree of height h , so that `Pos d 0` addresses a leaf and `Pos d h` addresses the slot a `JOINh` fills. Indexing by both endpoints avoids all arithmetic on depths: the recursions on `getSub`, `setSub` and `pathSub` pattern-match on the position and the tree simultaneously, and the one impossible case (a `cons` step below height 0) is dismissed by `Pos.height_le`.

```

inductive Tree : Nat → Type
| tip : Option Leaf → Tree 0
| bin {d : Nat} : Tree d → Tree d → Tree (d + 1)

inductive Pos : Nat → Nat → Type
| nil {h : Nat} : Pos h h
| cons {d h : Nat} : Bool → Pos d h → Pos (d + 1) h

def parent (a b : Node H) : Node H :=
  (M.hNode a.hash a.sum b.hash b.sum, a.sum + b.sum)

def combine (n : Node H) (x : Bool × Node H) : Node H :=
  if x.1 then parent x.2 n else parent n x.2

def fold (n : Node H) : List (Bool × Node H) → Node H
| [] => n
| x :: rest => combine (fold n rest) x

```

Root-first paths. The model orders a sibling path from the root down, so `fold` is a structural recursion on the list and the soundness proof is a plain induction. The witness of §5 supplies the same pairs leaf-first, because the script consumes the deepest level first; nothing in the theorems depends on the order.

Hashes as parameters. Everything is parametric in a class `HashModel` whose fields are the two hash functions, the zero digest, and the assumptions the argument uses:

```
class HashModel (Leaf : Type) (H : outParam Type) where
  hLeaf : Leaf → H
  hNode : H → Nat → H → Nat → H
  empty0 : H
  hLeaf_inj : ∀ {l1 l2}, hLeaf l1 = hLeaf l2 → l1 = l2
  hNode_inj : ∀ {a1 b1 a2 b2 sa1 sb1 sa2 sb2},
    hNode a1 sa1 b1 sb1 = hNode a2 sa2 b2 sb2 →
      a1 = a2 ∧ sa1 = sa2 ∧ b1 = b2 ∧ sb1 = sb2
  hNode_ne_hLeaf : ∀ a sa b sb l, hNode a sa b sb ≠ hLeaf l
  hNode_ne_empty : ∀ a sa b sb, hNode a sa b sb ≠ empty0
  hLeaf_ne_empty : ∀ l, hLeaf l ≠ empty0
```

Injectivity is collision resistance idealised; the three inequalities are the tag separation that distinct BIP-340 tags provide. The encodings module discharges the serialisation half of each injectivity field (Lemma 1): given SHA-256 collision resistance, `serializeLeaf_inj` and `serializeNode_inj` are what remains. Nothing quantifies the adversary’s advantage; a proof under injectivity transfers to any adversary that cannot exhibit a collision.

A.3 Correspondence with the Paper

Each predicate box is represented by a structure or inductive predicate whose fields are the checks of the box, in protocol order; the table maps the paper’s boxes, algorithms and results to the Lean names.

Table 8: Correspondence between the paper and the Lean development.

Paper	Lean
<i>Objects and the step</i>	
tree, positions, paths, fold (§4.1)	Tree, Pos, Tree.pathSub, fold, combine, parent
hashes, tag separation	HashModel
leaf, body ℓ , node	Leaf, Leaf.WF, Leaf.body, leafNode
step, chained steps (check 2 of each box)	Step, Step.valid, Step.WF, Steps
check 1 of each box (binding of r , m)	BindingModel, witness_root_binds, output_binds, CCVTweak, ccv_input_binds
last check, terminal branch	Final, ExitKT
<i>Predicate boxes</i>	
Φ_{EXIT} , = and evict branch	ExitBlock, ExitBlock.payoutOk, ExitK, ExitKT, ExitK.no_inflation, ExitK.exact_conservation, ExitK.pays_exit_key
Φ_{EXIT} , t	as above; the tower signature is the opaque ExitBlock.authorised
Φ_{JOIN} , dense	Join, join_sound
Φ_{JOIN} , sparse and JOINK	SparseJoin, Tree.Sparse, sparseJoin_sound, sparse_collision_rejected; K joins are Steps
Φ_{UPDATE} , x	Update, update_sound; x is in auth
Φ_{CLAIM} , carve forms	Claim, claimedSource, claim_sound, claim_evictable, claim_not_final; h , x , split are in auth
Φ_{CLAIM} , merge forms	Merge, MergeOk, mergeTree, merge_sound
$\Phi_{\text{EVICT_SUBTREE}}$	EvictSubtree, Tree.sweepPayouts, evictSubtree_sound, sweepPayouts_total

Paper	Lean
Φ_{REWRITE}	Rewrite, Tree.rewritePayouts, Leaf.AmountOnlyIncrease, rewrite_sound, rewrite_unsigned_change, rewrite_final_authorised, rewrite_payout_conservation, rewrite_exit_conservation
Φ_{REWRITE} , reclaim form (check 8)	Rewrite.transfer, Rewrite.deadline, Leaf.TransferShape, Tree.Transfer, rewrite_transfer_sound, rewrite_reclaimable_change
Φ_{DETACH}	Detach, detach_sound, detach_member_payable; m_h is a binding fact, outside the step
Φ_{RECLAIM}	Reclaim, reclaim_sound, reclaim_only_flagged
<i>Algorithms (the facts their lines rely on)</i>	
Alg. 1 Create	none; its outputs are the hypotheses treeSpk_inj and the real-tree invariant
Alg. 2 Graft, 3 AcceptLeaf	Join, join_sound, Step.Grounded, Steps.sound, fold_sound payable, exit_available, ExitK.single_sound, ExitKT.single_terminal, exit_blocks_reachable
Alg. 4 Exit	Tree.path_maintenance, Tree.path_maintenance_valid
Alg. 5 Follow	Voucher.issue, voucher_no_replay, voucher_survives_topup, self_voucher_fixed, final_never_carved, voucher_survives_detach
Alg. 6 Vouch	Voucher.Redeemable, Voucher.issue_redeemable, Voucher.redeem_pre, claim_sound
Alg. 7 Redeem	the voucher half only (Voucher); the ordering with an HTLC is outside the model
Alg. 8 Swap	Rewrite, rewrite_sound, rewrite_payout_conservation
Alg. 9 Rewrite	evictSubtree_sound, claim_evictable, Reachable.evictable
Alg. 10 Evict	MergeVoucher, merge_body_preserved, merge_stale_rejected, merge_no_overpay, merge_cycle_overpays
Alg. 11 Receive	rewrite_transfer_sound, merge_target_monotone, Reachable.exit_value
Alg. 12 SettleWindow	MergeVoucher.Redeemable, merge_target_monotone, Tree.path_maintenance
Alg. 13 Watch	Reclaim, reclaim_sound, reclaim_only_flagged
Alg. 14 Reclaim	
<i>Lemmas and theorems of §6</i>	
Lemma 1 (encodings)	Encoding.enc64_eq_minimal, serializeLeaf_inj, serializeNode_inj, minimalLE_two_pow_32_length, scriptNum_two_pow_31_length, scriptNum_neg_max31_succ
Theorem 1 (binding)	root_inj, fold_sound
Lemma 2	Step.sound
Theorem 2 (conservation)	Steps.sum, ExitK.no_inflation
Theorem 3 (payability)	payable
Theorem 4 (real-tree invariant)	Steps.sound, join_sound
Lemma 3 (path maintenance)	Tree.path_maintenance, path_maintenance_valid for a real tree; the opaque-node case is a paper proof
histories, §6	Trans, Reachable, Accepts, Chain, Chain.sound, chain_state, Reachable.ledger
Theorem 5 (UC realisation)	not in Lean; the paper proof of §7

Two entries differ from the paper's boxes in what they check, and the difference is deliberate. `Update` carries a hypothesis `new_dust`, that the written leaf is at least `DUST_MIN`, which the script of Φ_{UPDATE} does not check: the spec (7.8) asks operators not to co-sign such a leaf and the model takes the operator to honour it, so that `Reachable.dust` holds along a whole history. And `Claim` states the

carve rules as fields (`evictorKey_eq`, `evictAfter_le`, `claim_dust`, `residual_dust`), which are checks 7 and 8 of Φ_{CLAIM} .

The key definitions follow, as they stand in the Lean source. A step is a path, a height, an old and a new node, valid when the old node folds to the current root and the new node to the root written to output 0:

```
structure Step (H : Type) where
  height : Nat
  path : List (Bool × Node H)
  old : Node H
  new : Node H

def Step.valid (s : Step H) (r r' : Node H) : Prop :=
  fold s.old s.path = r ∧ fold s.new s.path = r'

inductive Steps : Node H → List (Step H) → Node H → Prop
| nil (r : Node H) : Steps r [] r
| cons {r r₁ r' : Node H} {s : Step H} {rest : List (Step H)} :
  s.valid r r₁ → Steps r₁ rest r' → Steps r (s :: rest) r'
```

The predicates are structures over steps. Φ_{EXIT} , Φ_{JOIN} , Φ_{UPDATE} and Φ_{CLAIM} (carve forms):

```
structure ExitK (exact : Bool) (r : Node H) (blocks : List (ExitBlock H)) (r' : Node H)
  (out0 : Nat) : Prop where
  steps : Steps r (blocks.map ExitBlock.step) r'
  out0_eq : out0 = r'.sum
  payouts : ∀ b ∈ blocks, b.payoutOk exact
  auth : ∀ b ∈ blocks, b.authorised

structure Join (auth : Prop) (h : Nat) (r : Node H) (path : List (Bool × Node H)) (new :
  Node H)
  (r' : Node H) (out0 : Nat) : Prop where
  step : (Step.mk h path (Tree.emptyNode h) new).valid r r'
  out0_eq : out0 = r'.sum
  authorised : auth

structure Update (auth : Prop) (π : Params) (r : Node H) (path : List (Bool × Node H))
  (old new : Leaf) (r' : Node H) (out0 : Nat) : Prop where
  step : (leafStep path (some old) (some new)).valid r r'
  out0_eq : out0 = r'.sum
  new_dust : π.dustMin ≤ new.amount
  authorised : auth

structure Claim (auth : Prop) (π : Params) (r : Node H)
  (srcPath claimPath : List (Bool × Node H)) (src claim : Leaf) (r' : Node H) (out0 :
  Nat) :
  Prop where
  amount_le : claim.amount ≤ src.amount
  seq_lt : src.seq < π.seqFinal
  evictorKey_eq : claim.evictorKey = src.evictorKey
  evictAfter_le : claim.evictAfter ≤ src.evictAfter
  claim_dust : π.dustMin ≤ claim.amount
  residual_dust : claim.amount = src.amount ∨ π.dustMin ≤ src.amount - claim.amount
  steps : Steps r [leafStep srcPath (some src) (claimedSource src claim.amount),
    leafStep claimPath none (some claim)] r'
  out0_eq : out0 = r'.sum
  authorised : auth
```

The cohort and receive predicates, Φ_{REWRITE} (with the reclaim form as transfer and deadline), Φ_{DETACH} , Φ_{CLAIM} (merge forms) and Φ_{RECLAIM} :

```
structure Rewrite (h : Nat) (π : Params) (auth : Pos h 0 → Prop) (lockTime : Nat) (r :
  Node H)
  (path : List (Bool × Node H)) (old new : Tree h) (r' : Node H) (out0 : Nat)
  (payouts : List Payout) : Prop where
```

```

steps : Steps r [Step.mk h path old.root new.root] r'
out0_eq : out0 = r'.sum
authorised : ∀ (q : Pos h 0) (l : Leaf), old.get q = some l → new.get q ≠ some l →
  ¬ l.AmountOnlyIncrease π (new.get q) → ¬ π.Reclaimable l → auth q
transfer : ∀ (c : Pos h 1) (l : Leaf), old.get (c.pairAt false) = some l →
  l.TransferShape π (new.get (c.pairAt false)) → old.Transfer π new c
deadline : ∀ (q : Pos h 0) (l : Leaf), old.get q = some l → new.get q ≠ some l →
  π.Reclaimable l → ¬ l.AmountOnlyIncrease π (new.get q) →
  (¬ ∃ c : Pos h 1, q = c.pairAt false ∧ l.TransferShape π (new.get q)) →
  l.evictAfter ≤ lockTime
payouts_eq : payouts = old.rewritePayouts π new

structure Detach (auth : Prop) (h : Nat) (r : Node H) (path : List (Bool × Node H))
  (sub : Node H) (r' : Node H) (out0 : Nat) (r1 : H) (out1 : Nat) : Prop where
  steps : Steps r [Step.mk h path sub (Tree.emptyNode h)] r'
  out0_eq : out0 = r'.sum
  r1_eq : r1 = sub.hash
  out1_eq : out1 = sub.sum
  authorised : auth

structure MergeOk (π : Params) (src tgt : Leaf) (X : Nat) : Prop where
  owner_eq : tgt.ownerKey = src.exitKey
  exit_eq : tgt.exitKey = src.exitKey
  src_not_final : src.seq ≠ π.seqFinal
  tgt_not_final : tgt.seq ≠ π.seqFinal
  tgt_not_reclaimable : ¬ π.Reclaimable tgt
  raise : tgt.amount < X
  draw_le : X - tgt.amount ≤ src.amount
  residual_dust : X - tgt.amount = src.amount ∨ π.dustMin ≤ src.amount - (X - tgt.amount)

structure Reclaim (auth : Prop) (π : Params) (lockTime : Nat) (r : Node H)
  (path : List (Bool × Node H)) (l : Leaf) (r' : Node H) (out0 : Nat) : Prop where
  reclaimable : π.Reclaimable l
  deadline : l.evictAfter ≤ lockTime
  step : (leafStep path (some l) none).valid r r'
  out0_eq : out0 = r'.sum
  authorised : auth

```

A spend accepted by some predicate is an `Accepts` step between two roots, and a history is a `Chain` of them; `Chain.sound` bridges every on-chain root back to a reachable tree (§A.1).

A.4 Assumption Ledger

Every hypothesis outside Lean's logic, what it stands for in the deployment, and which theorems consume it.

hypothesis	stands for	used by
hLeaf_inj	SHA-256 collision resistance + se-leafNode_inj, root_inj	
hNode_inj	SHA-256 collision resistance + se-root_inj, fold_sound	
hNode_ne_hLeaf, hNode_ne_empty, hLeaf_ne_empty	distinct tags (random-oracle heuristic)	leafNode_inj, fold_sound
treeSpk_inj	DL of NUMS + CR of H_{state} , TapTweak (opposite-sign keys); taproot binding in the ROM (same-sign keys)	witness_root_binds
CCVTweak.dataTweak_inj, as treeSpk_inj, one per stage		CCVTweak.binding, ccv_input_binds
tapTweak_inj		
real-tree invariant	creation check + grounded grafts (off-chain)	every theorem stated for a Tree
<i>used by Theorem 5 but outside Lean</i>		
compiler correctness	the bytecode implements Pred_r (tested, 3,019 checks)	$\mathcal{G}_{\text{ledger}}[\text{Pred}_r]$
opcode semantics	[A-1] to [A-6] (Table 2)	every tier
[C]	contention: κ, β (§3)	liveness
[MIN]	CR of TapSighash and the challenge hash	chal-MIN
Lemma 3, opaque nodes	paper proof from hNode_inj (real honest paths tree: path_maintenance)	
(none)		Steps.sum, Ex- itK.no_inflation, encodings

The conservation results use no hypothesis at all, which is the formal content of “the covenant conserves sums whatever it is shown”. The hash axioms are stated for all leaves and all Nat sums, while the serialisation injectivities hold for well-formed leaves and sums below 2^{64} ; the bridge is that every sum on a verified path is at most the input amount, and the 41-byte size check on each sibling bounds its sum below 2^{64} , so the axioms are only ever applied inside the encodings’ range.

A.5 Modules and Effort

module	lines	contents
Hash.lean	59	the HashModel class
Leaf.lean	132	Leaf, Leaf.WF, Params, Node, leafNode, leafNode_inj
Tree.lean	646	trees, positions, sibling pairs, paths, fold, subtree positions (Pos.append), all tree theorems
Transitions.lean	774	Step, Steps, the transitions, the terminal branch, sparse positions
Cohort.lean	673	EVICT_SUBTREE _h , REWRITE _h with rule 8, DETACH _h on published subtrees, Merge, Reclaim
Voucher.lean	340	vouchers, non-replay, survival of a top-up and of a detach
Receive.lean	459	merge vouchers, target monotonicity, the overpay bound, Reachable.exit_value
Trace.lean	794	transition records, Reachable, Chain, the trace invariants
Binding.lean	123	BindingModel, the binding theorems, the CCV tweak (CCVTweak)
Encoding.lean	447	encodings, serialisation injectivity, SEQ_FINAL
Axioms.lean	94	the axiom audit

The only proofs that took real effort were `fold_sound`, where the two-index positions and the domain-separation case for a tip had to be found, and `Steps.sound`, where the height of the grounding tree must be reconciled with the height the soundness lemma returns, which is why `Step` carries its height explicitly. A verified interpreter for the tapscript subset would need a model of the stack machine, of BIP-441 numerics, of `OP_TX`'s serialisations and of the BIP-341 message, and a proof that each generated script's execution relation equals the corresponding `Step` relation; we estimate it at several thousand lines and regard it as the natural sequel.

A.6 Binding to the Output

`Binding.lean` records the one cryptographic step outside the hash model. A structure `BindingModel` carries an abstract `treeSpk : H → TapRoot → Spk` with the hypothesis that it is injective in both arguments, and two one-line theorems conclude that a witness (r', m') passing the `scriptPubKey` check equals the committed (r, m) , and that the successor output commits to the computed root under the same m . In the deployment the map is $(r, m) \mapsto x(P_r + H_{\text{TapTweak}}(x(P_r) \parallel m) \cdot G)$ with $P_r = \text{lift_x}(x(\text{NUMS} + H_{\text{state}}(r)G))$, and the two x-projections identify a point with its negation. A collision $(r, m) \neq (r', m')$ yields one of: $x(P_r) = x(P_{r'})$ with $r \neq r'$, which is a collision in H_{state} or the relation $\text{NUMS} + tG = -(\text{NUMS} + t'G)$, solvable for the discrete logarithm of `NUMS`; $x(P_r) = x(P_{r'})$ with $m \neq m'$, a collision in H_{TapTweak} ; or distinct internal keys with $Q = \pm Q'$. Write $P_r = \sigma(\text{NUMS} + hG)$ with $\sigma = \pm 1$ and $h = H_{\text{state}}(r)$, and $t = H_{\text{TapTweak}}(x(P_r) \parallel m)$. When the two keys have opposite signs, $Q = \pm Q'$ is a linear relation in `NUMS` and G , solvable for the discrete logarithm of `NUMS`. When they have the same sign the `NUMS` terms cancel and what remains is $h + \sigma t = h' + \sigma t' \pmod n$, a collision of the composed map $(r, m) \mapsto H_{\text{state}}(r) + \sigma H_{\text{TapTweak}}(x(P_r) \parallel m)$: neither a discrete logarithm nor a collision of either hash. We assume it hard in the random-oracle model, which is how BIP-341's own binding argument goes. Making the hypothesis explicit is the point: it is what a future script change must not break. For the CCV tier `CCVTweak.binding` decomposes `treeSpk_inj` into two stage assumptions, the BIP-443 data tweak of `NUMS` by the root and the taproot tweak by m , each of which carries the same case split, so the binding theorems hold for a CCV tree unchanged (`ccv_input_binds` states the first for the input's own taptree).

A.7 Encodings

`Encoding.lean` proves the byte-level facts the design leans on, over natural-number bytes so that every proof stays within linear arithmetic: fixed-width little-endian encodings are injective on their range; `minimalLE (n + 2^(8k)) = leBytes k n ++ [1]` for $n < 2^{8k}$, which with $k = 8$ is `enc64` and justifies producing it with one addition; the 116-byte leaf and the 82-byte node preimage are injective; and the `SEQ_FINAL` facts of Lemma 1, computed by `decide` on a bounded-recursion model of `CScriptNum`.

A.8 What Is Not Proved

The tapscript bytecode is outside the model. That the scripts of §5 implement the relations above is established by execution: every script of every tier runs in the simulator against a reference tree with positive and negative cases, and the Go generators are compared byte for byte with the Python output. A verified interpreter for the tapscript subset (BIP-441 numerics, OP_TX, OP_CHECKSIGFROMSTACK) would close this gap and is a project of a different size. Signatures are authorisation and deliberately opaque. A lock time is a parameter that the deadline checks of Reclaim and Rewrite compare against; that it is a height that only grows along a chain is not modelled. Payout destinations are modelled as the payout the script reads from output k , whose key `ExitBlock.payoutOk` requires to be the leaf's exit key, for an eviction as for an owner exit (`ExitK.pays_exit_key`); that the bytecode makes the output script P2TR of that key is established by execution. Lemma 3 is mechanised for a real tree; its extension to a path through an opaque node has a paper proof only. Concurrency, mempool policy, reorganisations and the off-chain protocol are not theorems about hash trees and belong to a model checker; a design for a P [16] model is in the artefact.

B Script Listings

Figure 9 states the step every transition is built from; the listings below are taken from the reference generators and are what the size figures of §10 measure. Stack comments show the main stack left to right with the top on the right, and the alt stack after a bar.

Step $(\pi, n_{\text{old}}, n_{\text{new}}, (r, S))$	
1 :	$(r_0, S_0) \leftarrow \text{fold}(n_{\text{old}}, \pi)$ // $D - h$ unrolled levels
2 :	assert $r_0 = r \wedge S_0 = S$ // CHECK_OLD: S is input 0's amount
3 :	$(r', S') \leftarrow \text{fold}(n_{\text{new}}, \pi)$ // same siblings, same sums
4 :	return (r', S')
 Final $((r', S'), m)$	
1 :	if $(r', S') = E_D$ then return // terminal: output 0 free
2 :	$(a_0, \text{spk}_0) \leftarrow \text{OP_TX}(\text{output } 0)$
3 :	assert $\text{spk}_0 = \text{TreeSPK}(r', m) \wedge a_0 = S'$

Fig. 9. The Merkle-sum step shared by all transitions, and the output rule. A transition is a sequence of steps, each proving its old node against the root the previous step produced (CHECK_PREV), followed by Final. Authorisation and payout checks are specific to each transition.

One level of the fold (FULL tier, 88 bytes). LEVEL advances both accumulators, the old node (oh, os) and the new node (nh, ns) , by one level using the same direction bit and sibling $sc = h \parallel \text{enc}_{64}(s)$ from the witness. T is the 64-byte tag prefix of H_{node} ; PAD8 is $\langle 2^{64} \rangle$ OP_ADD, which yields enc_{64} of the sum by Lemma 1.

```

OP_6 OP_ROLL OP_6 OP_ROLL      # T oh os nh ns dir sc
OP_SIZE 41 OP_EQUALVERIFY      # sc is exactly hash || enc64(sum)
OP_DUP 32 OP_8 OP_SUBSTR      # ... dir sc ss8 (sibling sum, raw 1e64)
OP_DUP OP_TOALTSTACK          # | ss8
OP_3 OP_ROLL OP_TUCK OP_ADD    # T oh os nh dir sc ns ns'
OP_TOALTSTACK PAD8            # T oh os nh dir sc ns9 | ss8 ns'
OP_3 OP_ROLL OP_SWAP OP_CAT    # T oh os dir sc nc (nc = nh || enc64(ns))
OP_OVER OP_SWAP OP_3 OP_PICK  # T oh os dir sc sc nc dir
OP_NOTIF OP_SWAP OP_ENDIF OP_CAT # T oh os dir sc pre_n (dir ? sc || nc : nc || sc)
OP_5 OP_PICK OP_SWAP OP_CAT OP_SHA256 # T oh os dir sc nh'
OP_TOALTSTACK                  # | ss8 ns' nh'
OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK
                                # T oh os dir sc nh' ns' ss8
OP_5 OP_ROLL OP_TUCK OP_ADD    # T oh dir sc nh' ns' os os'
OP_TOALTSTACK PAD8            # T oh dir sc nh' ns' os9 | os'
OP_5 OP_ROLL OP_SWAP OP_CAT    # T dir sc nh' ns' oc (oc = oh || enc64(os))
OP_3 OP_ROLL OP_SWAP OP_4 OP_ROLL # T nh' ns' sc oc dir
OP_NOTIF OP_SWAP OP_ENDIF OP_CAT # T nh' ns' pre_o
OP_3 OP_PICK OP_SWAP OP_CAT OP_SHA256 # T nh' ns' oh'
OP_FROMALTSTACK OP_2SWAP      # T oh' os' nh' ns'

```

Checking the old root and writing the new one. After D levels, CHECK_OLD compares the old accumulator with the committed state: the sum against input 0's amount and the hash against the witness root r parked on the alt stack (itself bound to the input scriptPubKey by the prologue). FINAL constrains output 0 unless the new root is the empty tree.

```

# CHECK_OLD: [T oh os nh ns] | ... r -> [T nh ns]
OP_2SWAP <SEL_IN_AMOUNT> OP_TX OP_NUMEQUALVERIFY OP_FROMALTSTACK OP_EQUALVERIFY

# FINAL: [T nh ns] | m -> [1]
OP_ROT OP_DROP                # nh ns
OP_OVER <E_D.hash> OP_EQUAL
OP_IF OP_2DROP OP_FROMALTSTACK OP_DROP OP_1 # terminal: output 0 free
OP_ELSE OP_SWAP OP_FROMALTSTACK TREE_SPK    # ns spk'
      OP_0 <SEL_OUT_SINGLE> OP_TX           # ns spk' amt0 spk0
      OP_ROT OP_EQUALVERIFY OP_NUMEQUAL    # 1
OP_ENDIF

```

The MID gadget. Under 0xC0 a computed 4-byte number x is widened to 8 bytes by two witness elements z, w : the script checks $\text{size}(x) + \text{size}(z) = 4$ and $z \parallel w = 00000000$ and uses $x \parallel z \parallel w$ as $\text{le}_{64}(x)$. Two such gadgets per level and the per-field size checks account for almost all of MID's premium over FULL.

The MIN message. The MIN exit assembles the BIP-341 signature message for input 0 with hash type SIGHASH_DEFAULT and checks it as described in §5. Fields marked *derived* are computed by the script from values it has verified; fields marked *raw* are witness elements the script does not constrain but must be presented consistently, since the whole message is hashed and compared with the real one through OP_CHECKSIG.

field	source	note
epoch, hash type	constant 00 00	SIGHASH_DEFAULT
version, lock time	raw	free
sha_prevouts	raw	free
sha_amounts	derived: $\text{SHA256}(\text{le}_{64}(S) \parallel \text{amt}_1)$	S verified by CHECK_OLD ; amt_1 raw; two inputs exactly
sha_scriptpubkeys	derived: $\text{SHA256}(22 \parallel \text{TreeSPK}(r, m) \parallel \text{spk}_1)$	TreeSPK recomputed; spk_1 raw
sha_sequences	raw	the field the spender grinds
sha_outputs	derived: $\text{SHA256}(\text{out}_0 \parallel \text{out}_1 \parallel \text{extra})$	out_0 $\text{le}_{64}(S') \parallel 22 \parallel \text{TreeSPK}(r', m),$ out_1 the payout; extra raw
spend type, input index	constant 02 00000000	script path, no annex, input 0
tapleaf hash	raw	the leaf being executed
key version, code-sep	constant 00 ffffffff	

The message is hashed with the TapSighash tag and the challenge e is computed from $G_x \parallel G_x \parallel \text{msg}$ under the challenge tag. The script verifies $e_{[0,31)} \parallel 00 = e$ against the witness-supplied $e_{[0,31)}$ and finally executes

$$\langle G_x \parallel e_{[0,31)} \parallel 01 \rangle \langle G_x \rangle \text{ OP_CHECKSIG.}$$

Grinding and fee bumping. The spender grinds the fee input’s **nSequence** until e ends in 00 (so that $e + 1$ is formed by replacing the last byte), 256 hashes on average, choosing values that set no BIP-68 relative lock (the disable bit set) and stay below 0xFFFFFFFF so the transaction keeps signalling replaceability. Any change to the transaction changes the message, so a replacement with a higher fee must be re-ground and re-signed by the owner; a third party bumps it only by a child spending the fee change or an anchor. The raw *extra* outputs travel as one witness element, and Bitcoin Core’s policy limits 0xC0 witness items to 80 bytes: one P2TR change output and a P2A anchor (56 bytes) fit, two P2TR change outputs (86 bytes) make the exit non-standard. The simulator checks consensus limits only.

C Measurements, Vouchers and Fee Tables

C.1 Measured Sizes

Table 9 gives the script, witness and transaction sizes behind Figure 5 and Table 3.

C.2 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 [33] 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,

Table 9. 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 10.

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	

Table 10. 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
JOIN ₂ , two leaves	32					2,821

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 JOIN _{K} . It still hosts channel leaves and vouchers and still exits in one transaction.

Where the bytes go. A sparse proof is the smaller witness and the larger script, and the script wins (Table 10). 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 JOIN_0 (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 5); 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).

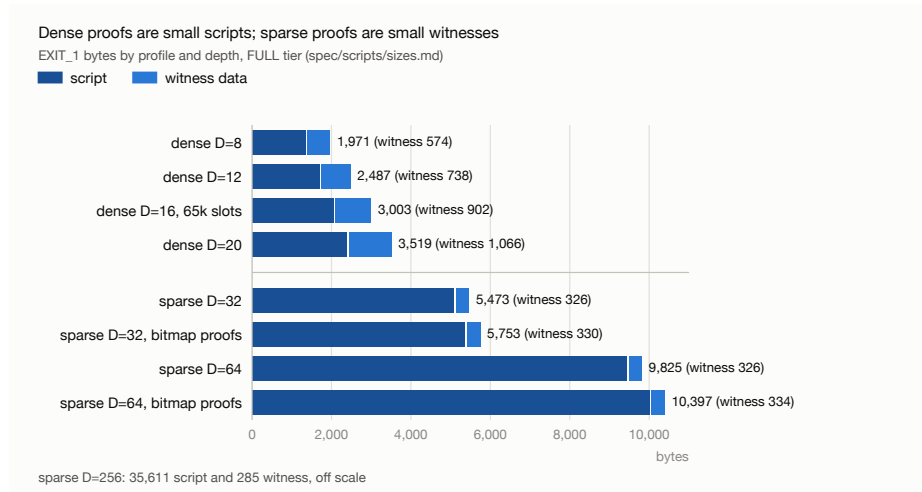


Fig. 10. EXIT₁ bytes by profile and depth, FULL tier: dense proofs are small scripts, sparse proofs small witnesses. Sparse rows are measured on the six-leaf tree.

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.

C.3 Vouchers (CLAIM)

A voucher is a BIP-340 signature by the source leaf’s owner 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), verified with `OP_CHECKSIGFROMSTACK`. Under spec 0.10.0 every tier binds `tree_id`, a constant fixed when a tree is created and inherited by the cohorts detached from it, in place of `m`, which names a rule set and so is shared by trees created with the same parameters; a voucher therefore verifies only in its own lineage, and survives a detach. The claim leaf must keep the source's evictor key and a lease no later than the source's (spec 0.10.0), and neither the claim leaf nor a non-zero residual may be below `DUST_MIN`, a creation constant of 330 sat by default (spec 0.11.0), so that no carve writes a leaf whose removal default policy would not relay. **CLAIM** is two steps: the source leaf's amount is reduced by the claim amount and its `seq` advanced by one (or the position emptied if drained), then the claim leaf is placed at an empty position. The total is unchanged. Replay is impossible because the first step advances the source's `seq`, which the message binds, so it never again matches a leaf in the tree. The amount is left out of the message so that an amount-only top-up of the source, which `REWRITEh` allows on the operator's signature, voids no voucher and no `EXITt` authorisation; the sum check still stops a voucher drawing more than the source holds. Because a voucher lets an owner move value with a lone signature, leaves whose balance is not the owner's alone set `seq` to the tier's `SEQ_FINAL` (§8), whose successor does not fit in four bytes, so the **CLAIM** script rejects them. Vouchers are the mechanism for operator-to-member push payments without the operator's later participation.

Hash-locked vouchers (CLAIM^h). Adding a payment hash to the signed message, under its own tag, and requiring the preimage at redemption turns a voucher into a hash-locked cheque that can be paired with a channel-leaf HTLC or a Lightning invoice carrying the same hash. It is not an HTLC and the pairing is not atomic: the value stays in the source leaf, and until the redemption confirms the issuer can void the cheque with a conflicting `EXIT` or `SPLIT` of that leaf, so the redeemer bears issuer risk for that window, which contention on the tree output lengthens. The script does not compare a witness-supplied hash against the preimage: it computes `SHA256(preimage)` and feeds it into the signed message, so a wrong preimage fails as an invalid signature. That costs one hash opcode, eight bytes of script in every tier, and no opcode beyond those **CLAIM** already needs. The separate tag is required: reusing H_{voucher} for both forms would let a redeemer strip the hash lock.

The variant adds ordering, not security. It cannot create an on-chain deadline, because script can require that a spend not happen before a height but not that it happen before one, and the value backing a voucher never leaves the source leaf, so there is nothing to refund; deadlines live in the other leg of the swap, and an issuer retires an unredeemed voucher by advancing the source leaf's `seq`. Redemption publishes the preimage, which is what settles the other leg and also what obliges a redeemer to secure its own leg first.

Exit-key consent (CLAIM^x , CLAIM^{hx}). The **FULL** tier also builds both forms with one further signature over the transaction by the source leaf's exit key, as `UPDATEx` adds to `UPDATE`: 7 bytes of script over the plain form, as `UPDATEx`

Table 11. Comparison with shared-output constructions. Pools: OP_TAPLEAF_UPDATE_VERIFY [47], CoinPool’s OP_MERKLESUB [29], OP_EVICT [52]. 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 none	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 12. 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

over UPDATE, and 64 of witness. Its use is a credit leaf owned by an issuer and exiting to one recipient: every voucher against it pays that recipient, whose own redemptions carry the exit-key signature at no cost, and the issuer cannot carve the leaf even with a voucher it signs to itself.

C.4 Comparisons

C.5 Fee Tables

Fees for the measured sizes at fee rates from 1 to 250 sat/vB, on the model of §10; every fee is the tabulated vB times the rate. The sparse row is the six-leaf measurement.

case	vB	1	10	25	50	250 sat/vB
dense EXIT ₁ , $D = 8$	762	762	7,620	19,050	38,100	190,500
dense EXIT ₁ , $D = 16$	1,020	1,020	10,200	25,500	51,000	255,000
dense EXIT ₁ , $D = 20$	1,149	1,149	11,490	28,725	57,450	287,250
sparse EXIT ₁ , $D = 32, N = 6$	1,637	1,637	16,370	40,925	81,850	409,250
MID dense EXIT ₁ , $D = 16$	1,355	1,355	13,550	33,875	67,750	338,750
MIN dense EXIT ₁ , $D = 16$	1,473	1,473	14,730	36,825	73,650	368,250
second exit in EXIT ₂ , marginal	665	665	6,650	16,625	33,250	166,250
EVICT ₄ SUBTREE ₄ , per leaf (13)	203	203	2,030	5,075	10,150	50,750
EVICT ₆ SUBTREE ₆ , per leaf (51)	158	158	1,580	3,950	7,900	39,500
plain P2TR to P2TR	111	111	1,110	2,775	5,550	27,750

Break-even balances for the dense $D = 16$ exit, i.e. the leaf amount at which the fee is 1%, 5% or 10% of the balance:

sat/vB	fee (sat)	balance at 1%	at 5%	at 10%
2	2,040	204,000	40,800	20,400
10	10,200	1,020,000	204,000	102,000
50	51,000	5,100,000	1,020,000	510,000
250	255,000	25,500,000	5,100,000	2,550,000

Operators should size cohorts and minimum leaf amounts so that a sweep stays below about 10% of the smallest leaf at the fee-rate tail they intend to survive; the FULL-tier fee-from-payout option saves a constant 99 vB and is unavailable to channel leaves.

D Complete Script Listings

This appendix reproduces every script the reference generators emit, for the smallest depths that exercise every construct: dense $D = 4$ and sparse $D = 8$, and dense $D = 8$ for the cohort shapes of FULL. Tapscript has no subroutines, so a compiled script is one flat opcode stream; the generators, however, build every script from a small set of helpers, and the listings below are those streams *collapsed back* into named macro calls, one per line, with the remaining raw opcodes wrapped between them. Each tier’s macros are defined once (§D.1, §D.6, §D.8, §D.10), themselves collapsed against the other macros; the CCV tier under 0xC0 numerics uses MID’s. The collapse is mechanical (`tools/gen_structured.py`) and verified: expanding a structured listing reproduces the compiled bytes exactly, so these listings and the flat ones in `spec/scripts/` describe the same scripts, and the byte count in each heading is the compiled size. A run of n identical calls is written NAME xn . Angle brackets denote pushes of named constants: <TAG_NODE> is the 64-byte tagged-hash prefix, <EMPTY_k> the hash of E_k , <ENC0> the constant 2^{64} whose addition implements PAD8, <NUMS> the BIP-341 nothing-up-my-sleeve point, and <SEL_*> the OP_TX selectors of §4.2. Bare integers are minimal number pushes. Sizes at $D = 16$ and $D = 32$ are in Table 9; each further level adds one LEVEL (88 bytes dense, 135 sparse, 160 MID).

D.1 FULL Tier Macros

```

# Macro definitions, FULL tier. Each expansion is itself collapsed
# against the other macros; a macro never contains itself.

## TAGGED(LEAF)
<TAG_LEAF> OP_SWAP OP_CAT OP_SHA256

## TAGGED(NODE)
<TAG_NODE> OP_SWAP OP_CAT OP_SHA256

## TAGGED(STATE)
<TAG_STATE> OP_SWAP OP_CAT OP_SHA256

## TAGGED(VOUCHER)
<TAG_VOUCHER> OP_SWAP OP_CAT OP_SHA256

## TAGGED(EXIT)
<TAG_EXIT> OP_SWAP OP_CAT OP_SHA256

## TAGGED(POS)
<TAG_POS> OP_SWAP OP_CAT OP_SHA256

## TAGGED(TAPTWEAK)
<TAG_TAPTWEAK> OP_SWAP OP_CAT OP_SHA256

## PAD8 ([n] -> [enc64(n)]: minimal encoding of n + 2^64 is le64(n)||01)
<ENCO> OP_ADD

## LE64 ([n] -> [le64(n)])
PAD8
OP_8 OP_LEFT

## PAD4 ([n] -> [le32(n)])
<ZERO4> OP_CAT OP_4 OP_LEFT

## TREE_SPK ([r m] -> [TREE_SPK(r, m)] with EC ops)
OP_SWAP
TAGGED(STATE)
OP_0 OP_SWAP OP_EC_POINT_MUL <NUMS> OP_EC_POINT_ADD OP_EC_POINT_X_COORD
OP_TUCK OP_SWAP OP_CAT
TAGGED(TAPTWEAK)
OP_0 OP_SWAP OP_EC_POINT_MUL OP_SWAP OP_2 OP_SWAP OP_CAT OP_EC_POINT_ADD
OP_EC_POINT_X_COORD <0x5120> OP_SWAP OP_CAT

## TREE_SPK_MIN ([r m] -> [TREE_SPK(r, m)] with OP_TWEAKADD)
OP_SWAP
TAGGED(STATE)
<NUMS_X> OP_TWEAKADD OP_TUCK OP_SWAP OP_CAT
TAGGED(TAPTWEAK)
OP_SWAP OP_TWEAKADD <0x5120> OP_SWAP OP_CAT

## LEAF_ACC ([leaf] -> [H_leaf(leaf) amount])
OP_DUP OP_8 OP_LEFT OP_SWAP
TAGGED(LEAF)
OP_SWAP

## LEAF_POS ([leaf] -> [leaf H_pos(owner_key||nonce)])
OP_DUP 40 32 OP_SUBSTR OP_OVER 108 OP_4 OP_SUBSTR OP_CAT
TAGGED(POS)

## PROLOGUE (input is #0; (r, m) match the input scriptPubKey; alt <- m r)
<SEL_CURR_INPUT> OP_TX OP_0 OP_NUMEQUALVERIFY OP_2DUP
TREE_SPK
<SEL_IN_SPK> OP_TX OP_EQUALVERIFY OP_TOALTSTACK OP_TOALTSTACK

## CARVE_RULES ([src claim] -> [src claim]: both 116 bytes, claim.evictor_key ==
src.evictor_key, claim.evict_after <= src.evict_after (7.5 rule 5), claim.amount >=
DUST_MIN (7.5 rule 6))
OP_SIZE 116 OP_EQUALVERIFY OP_SWAP OP_SIZE 116 OP_EQUALVERIFY OP_SWAP OP_2DUP

```

```

72 32 OP_SUBSTR OP_SWAP 72 32 OP_SUBSTR OP_EQUALVERIFY OP_2DUP 104 OP_4
OP_SUBSTR OP_SWAP 104 OP_4 OP_SUBSTR OP_LESSTHANOREQUAL OP_VERIFY 330
OP_OVER OP_8 OP_LEFT OP_LESSTHANOREQUAL OP_VERIFY

## DRAIN_RULE  ([.. nh ns] -> [.. nh' ns]: a zero residual makes the new node E_0, any
                other is at least DUST_MIN (7.5 rules 2 and 6))
OP_DUP OP_NOT OP_IF OP_SWAP OP_DROP <EMPTY_0> OP_SWAP OP_ELSE 330 OP_OVER
OP_LESSTHANOREQUAL OP_VERIFY OP_ENDIF

## LEAF_BODY  ([leaf] -> [body]: the 108 bytes after the amount field (5.2), what vouchers
               and exit authorisations bind (7.5, 7.2 rule 7))
108 OP_RIGHT

## TOWER_AUTH  ([leaf sig] -> [leaf]: sig is a BIP-340 signature by leaf.owner_key over
                H_exit(tree_id || body(leaf)), the leaf without its amount (LEAF_BODY), checked by
                OP_CHECKSIGFROMSTACK (EXIT_K~t, 7.2 rule 7))
OP_OVER
LEAF_BODY
<TREE_ID> OP_SWAP OP_CAT
TAGGED(EXIT)
OP_2 OP_PICK 40 32 OP_SUBSTR OP_CHECKSIGFROMSTACK OP_VERIFY

## LEVEL  (dense level: dir sc T oh os nh ns -> T oh' os' nh' ns')
OP_6 OP_ROLL OP_6 OP_ROLL OP_SIZE 41 OP_EQUALVERIFY OP_DUP 32 OP_8 OP_SUBSTR
OP_DUP OP_TOALTSTACK OP_3 OP_ROLL OP_TUCK OP_ADD OP_TOALTSTACK
PAD8
OP_3 OP_ROLL OP_SWAP OP_CAT OP_OVER OP_SWAP OP_3 OP_PICK OP_NOTIF OP_SWAP
OP_ENDIF OP_CAT OP_5 OP_PICK OP_SWAP OP_CAT OP_SHA256 OP_TOALTSTACK
OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK OP_5 OP_ROLL OP_TUCK OP_ADD
OP_TOALTSTACK
PAD8
OP_5 OP_ROLL OP_SWAP OP_CAT OP_3 OP_ROLL OP_SWAP OP_4 OP_ROLL OP_NOTIF
OP_SWAP OP_ENDIF OP_CAT OP_3 OP_PICK OP_SWAP OP_CAT OP_SHA256
OP_FROMALTSTACK OP_2SWAP

## LEVEL_MIN  (dense level without splice opcodes: sibling as two elements)
OP_7 OP_ROLL OP_7 OP_ROLL OP_7 OP_ROLL OP_SIZE OP_8 OP_EQUALVERIFY OP_OVER
OP_SIZE 32 OP_EQUALVERIFY OP_DROP OP_DUP OP_TOALTSTACK OP_SWAP OP_OVER
PAD8
OP_CAT OP_SWAP OP_3 OP_ROLL OP_TUCK OP_ADD OP_TOALTSTACK
PAD8
OP_3 OP_ROLL OP_SWAP OP_CAT OP_OVER OP_SWAP OP_3 OP_PICK OP_NOTIF OP_SWAP
OP_ENDIF OP_CAT OP_5 OP_PICK OP_SWAP OP_CAT OP_SHA256 OP_TOALTSTACK
OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK OP_5 OP_ROLL OP_TUCK OP_ADD
OP_TOALTSTACK
PAD8
OP_5 OP_ROLL OP_SWAP OP_CAT OP_3 OP_ROLL OP_SWAP OP_4 OP_ROLL OP_NOTIF
OP_SWAP OP_ENDIF OP_CAT OP_3 OP_PICK OP_SWAP OP_CAT OP_SHA256
OP_FROMALTSTACK OP_2SWAP

## SPARSE_SETUP  ([pos_hash] -> [] ; alt <- pos E_0)
OP_TOALTSTACK <EMPTY_0> OP_TOALTSTACK

## SPARSE_TEARDOWN  (drop pos and E_k from the alt stack)
OP_FROMALTSTACK OP_DROP OP_FROMALTSTACK OP_DROP

## CHECK_OLD  (os == input 0 amount, oh == r (from alt))
OP_2SWAP <SEL_IN_AMOUNT> OP_TX OP_NUMEQUALVERIFY OP_FROMALTSTACK
OP_EQUALVERIFY

## CHECK_OLD_S  (CHECK_OLD with the sparse teardown)
OP_2SWAP <SEL_IN_AMOUNT> OP_TX OP_NUMEQUALVERIFY
SPARSE_TEARDOWN
OP_FROMALTSTACK OP_EQUALVERIFY

## CHECK_PREV  (os == ns_prev, oh == nh_prev (from alt); chained steps)
OP_2SWAP OP_FROMALTSTACK OP_NUMEQUALVERIFY OP_FROMALTSTACK OP_EQUALVERIFY

```

```

## CHECK_PREV_S    (CHECK_PREV with the sparse teardown)
OP_2SWAP
SPARSE_TEARDOWN
OP_FROMALTSTACK OP_NUMEQUALVERIFY OP_FROMALTSTACK OP_EQUALVERIFY

## LEAF_STEP    (cohort sweep: one leaf or empty -> its node, paying the next output)
OP_SIZE OP_ONOTEQUAL OP_IF OP_SIZE 116 OP_EQUALVERIFY OP_DUP 72 32 OP_SUBSTR
<OPERATOR_KEY> OP_EQUALVERIFY OP_DUP 104 OP_4 OP_SUBSTR
OP_CHECKLOCKTIMEVERIFY OP_DROP OP_FROMALTSTACK OP_DUP OP_1ADD OP_TOALTSTACK
<SEL_OUT_SINGLE> OP_TX <0x5120> OP_3 OP_PICK OP_8 32 OP_SUBSTR OP_CAT
OP_EQUALVERIFY OP_OVER OP_8 OP_LEFT OP_NUMEQUALVERIFY
LEAF_ACC
OP_ELSE OP_DROP <EMPTY_0> OP_0 OP_ENDIF

## LEVEL_S(k)    [shown for k = 0; <EMPTY_k> and the bit index vary with k]    (sparse level
k: sc T oh os nh ns | pos E_k -> T oh' os' nh' ns' | pos E_k+1)
OP_5 OP_ROLL OP_FROMALTSTACK OP_SWAP OP_SIZE OP_ONOTEQUAL OP_NOTIF OP_DROP
OP_DUP <ENCO> OP_CAT OP_ENDIF OP_SWAP <ENCO> OP_CAT OP_DUP OP_CAT OP_6
OP_PICK OP_SWAP OP_CAT OP_SHA256 OP_FROMALTSTACK OP_DUP OP_TOALTSTACK
OP_SWAP OP_TOALTSTACK OP_0 OP_DOWNSHIFT OP_1 OP_AND OP_ONOTEQUAL OP_SWAP
OP_SIZE 41 OP_EQUALVERIFY OP_DUP 32 OP_8 OP_SUBSTR OP_DUP OP_TOALTSTACK
OP_3 OP_ROLL OP_TUCK OP_ADD OP_TOALTSTACK
PAD8
OP_3 OP_ROLL OP_SWAP OP_CAT OP_OVER OP_SWAP OP_3 OP_PICK OP_NOTIF OP_SWAP
OP_ENDIF OP_CAT OP_5 OP_PICK OP_SWAP OP_CAT OP_SHA256 OP_TOALTSTACK
OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK OP_5 OP_ROLL OP_TUCK OP_ADD
OP_TOALTSTACK
PAD8
OP_5 OP_ROLL OP_SWAP OP_CAT OP_3 OP_ROLL OP_SWAP OP_4 OP_ROLL OP_NOTIF
OP_SWAP OP_ENDIF OP_CAT OP_3 OP_PICK OP_SWAP OP_CAT OP_SHA256
OP_FROMALTSTACK OP_2SWAP

## FINAL    [shown for D = 4; <EMPTY_D> is the empty root of the tree's depth]    (output 0 is
(ns, TREE_SPK(nh, m)) unless nh == E_D)
OP_ROT OP_DROP OP_OVER <EMPTY_4> OP_EQUAL OP_IF OP_2DROP OP_FROMALTSTACK
OP_DROP OP_1 OP_ELSE OP_SWAP OP_FROMALTSTACK
TREE_SPK
OP_0 <SEL_OUT_SINGLE> OP_TX OP_ROT OP_EQUALVERIFY OP_NUMEQUAL OP_ENDIF

## HASH_STEP(t)    [shown for t = 3; the OP_PICK depth is t-3]    (cohort sweep: two child
nodes -> parent node)
OP_2 OP_PICK OP_OVER OP_ADD OP_TOALTSTACK
PAD8
OP_CAT OP_SWAP
PAD8
OP_ROT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_0 OP_PICK OP_SWAP OP_CAT OP_SHA256
OP_FROMALTSTACK

## REWRITE_STEP(d)    [shown for d = 0; the OP_ROLL depths are d+3, d+4 and d+5, the picks
d+3 to d+5]    (cohort rewrite: old and new elements of one position -> oh os nh ns,
with 7.9 rules 3 and 5)
OP_3 OP_ROLL OP_4 OP_ROLL OP_SIZE OP_1 OP_EQUAL OP_IF OP_DROP OP_DUP OP_ENDIF
OP_SIZE OP_8 OP_EQUAL OP_IF OP_OVER
LEAF_BODY
OP_CAT OP_ENDIF OP_2DUP OP_EQUAL OP_NOTIF OP_OVER OP_SIZE OP_NIP OP_ONOTEQUAL
OP_IF OP_DUP OP_SIZE OP_NIP OP_ONOTEQUAL OP_IF OP_2DUP OP_8 108 OP_SUBSTR
OP_SWAP OP_8 108 OP_SUBSTR OP_EQUAL OP_IF OP_2DUP OP_SWAP OP_8 OP_LEFT
OP_SWAP OP_8 OP_LEFT OP_LESSTHANOREQUAL OP_OVER OP_4 OP_RIGHT <SEQ_FINAL>
OP_EQUAL OP_NOT OP_BOOLAND OP_ELSE OP_0 OP_ENDIF OP_NOTIF
OWNER_AUTH(0)
OP_ENDIF OP_ELSE
OWNER_AUTH(0)
OP_FROMALTSTACK OP_DUP OP_1ADD OP_TOALTSTACK <SEL_OUT_SINGLE> OP_TX <0x5120>
OP_4 OP_PICK OP_8 32 OP_SUBSTR OP_CAT OP_EQUALVERIFY OP_2 OP_PICK OP_8
OP_LEFT OP_NUMEQUALVERIFY OP_ENDIF OP_ENDIF OP_ENDIF OP_SWAP
ELEM_NODE(0)
OP_ROT
ELEM_NODE(1)

```

```

## REWRITE_STEP_X(d)    [shown for d = 0; the OP_ROLL depths are d+3, d+4 and d+5, the picks
                        d+3 to d+5]    (REWRITE_STEP with OWNER_AUTH_X)
OP_3 OP_ROLL OP_4 OP_ROLL OP_SIZE OP_1 OP_EQUAL OP_IF OP_DROP OP_DUP OP_ENDIF
    OP_SIZE OP_8 OP_EQUAL OP_IF OP_OVER
LEAF_BODY
OP_CAT OP_ENDIF OP_2DUP OP_EQUAL OP_NOTIF OP_OVER OP_SIZE OP_NIP OP_ONOTEQUAL
    OP_IF OP_DUP OP_SIZE OP_NIP OP_ONOTEQUAL OP_IF OP_2DUP OP_8 108 OP_SUBSTR
    OP_SWAP OP_8 108 OP_SUBSTR OP_EQUAL OP_IF OP_2DUP OP_SWAP OP_8 OP_LEFT
    OP_SWAP OP_8 OP_LEFT OP_LESSTHANOREQUAL OP_OVER OP_4 OP_RIGHT <SEQ_FINAL>
    OP_EQUAL OP_NOT OP_BOOLLAND OP_ELSE OP_0 OP_ENDIF OP_NOTIF
OWNER_AUTH_X(0)
OP_ENDIF OP_ELSE
OWNER_AUTH_X(0)
OP_FROMALTSTACK OP_DUP OP_1ADD OP_TOALTSTACK <SEL_OUT_SINGLE> OP_TX <0x5120>
    OP_4 OP_PICK OP_8 32 OP_SUBSTR OP_CAT OP_EQUALVERIFY OP_2 OP_PICK OP_8
    OP_LEFT OP_NUMEQUALVERIFY OP_ENDIF OP_ENDIF OP_ENDIF OP_SWAP
ELEM_NODE(0)
OP_ROT
ELEM_NODE(1)

## HASH_STEP2(d)    [shown for d = 8; the HASH_STEP pick depths are d-3 and d-5]    (cohort
                    rewrite: two old and two new child nodes -> old and new parent nodes)
OP_2SWAP OP_7 OP_ROLL OP_7 OP_ROLL OP_2SWAP
HASH_STEP(8)
OP_2SWAP OP_5 OP_ROLL OP_5 OP_ROLL OP_2SWAP
HASH_STEP(6)

## ELEM_NODE(d)    [shown for d = 0; picks L at d+4 and Z at d+3 for the old element, one
                    deeper for the new]    ([elem] -> [h s]: a leaf's node, or E_0 for the empty element;
                    picks the hoisted TAG_LEAF prefix (L) and E_0 (Z))
OP_SIZE OP_ONOTEQUAL OP_IF OP_SIZE 116 OP_EQUALVERIFY OP_DUP OP_8 OP_LEFT
    OP_SWAP OP_4 OP_PICK OP_SWAP OP_CAT OP_SHA256 OP_SWAP OP_ELSE OP_DROP OP_3
    OP_PICK OP_0 OP_ENDIF

## OWNER_AUTH(d)    [shown for d = 0; the OP_ROLL depth is d+5]    (cohort rewrite: the next
                    witness element is a signature by old.owner_key)
OP_5 OP_ROLL OP_2 OP_PICK 40 32 OP_SUBSTR OP_CHECKSIGVERIFY

## OWNER_AUTH_X(d)    [shown for d = 0; the OP_ROLL depth is d+5]    (OWNER_AUTH plus a
                    signature by old.exit_key (REWRITE_h~x))
OWNER_AUTH(0)
OP_5 OP_ROLL OP_2 OP_PICK OP_8 32 OP_SUBSTR OP_CHECKSIGVERIFY

## EXIT_CONSENT(k)    [shown for k = 3; k = 4 under a preimage (CLAIM~hx) or for UPDATE~x]
OP_3 OP_PICK OP_8 32 OP_SUBSTR OP_CHECKSIGVERIFY

```

D.2 FULL Tier, Dense Profile ($D = 4$)

```

# EXIT_1 (dense, D=4): 1045 bytes
# macros in CAPITALS are defined in MACROS_FULL.txt; raw opcodes are wrapped and indented

PROLOGUE
OP_DUP OP_TOALTSTACK OP_IF OP_OVER 104 OP_4 OP_SUBSTR OP_CHECKLOCKTIMEVERIFY
    OP_DROP OP_OVER 72 32 OP_SUBSTR OP_ELSE OP_OVER 40 32 OP_SUBSTR OP_ENDIF
    OP_CHECKSIGVERIFY OP_SIZE 116 OP_EQUALVERIFY OP_1 <SEL_OUT_SINGLE> OP_TX
    <0x5120> OP_3 OP_PICK OP_8 32 OP_SUBSTR OP_CAT OP_EQUALVERIFY OP_OVER OP_8
    OP_LEFT OP_FROMALTSTACK OP_IF OP_NUMEQUALVERIFY OP_ELSE OP_LESSTHANOREQUAL
    OP_VERIFY OP_ENDIF
LEAF_ACC
<TAG_NODE> OP_ROT OP_ROT <EMPTY_0> OP_0
LEVEL    x4
CHECK_OLD
FINAL

# EXIT_2 (dense, D=4): 1635 bytes
# macros in CAPITALS are defined in MACROS_FULL.txt; raw opcodes are wrapped and indented

```

```

PROLOGUE
OP_DUP OP_TOALTSTACK OP_IF OP_OVER 104 OP_4 OP_SUBSTR OP_CHECKLOCKTIMEVERIFY
  OP_DROP OP_OVER 72 32 OP_SUBSTR OP_ELSE OP_OVER 40 32 OP_SUBSTR OP_ENDIF
  OP_CHECKSIGVERIFY OP_SIZE 116 OP_EQUALVERIFY OP_1 <SEL_OUT_SINGLE> OP_TX
  <0x5120> OP_3 OP_PICK OP_8 32 OP_SUBSTR OP_CAT OP_EQUALVERIFY OP_OVER OP_8
  OP_LEFT OP_FROMALTSTACK OP_IF OP_NUMEQUALVERIFY OP_ELSE OP_LESSTHANOREQUAL
  OP_VERIFY OP_ENDIF
LEAF_ACC
<TAG_NODE> OP_ROT OP_ROT <EMPTY_0> OP_0
LEVEL x4
CHECK_OLD
OP_SWAP OP_TOALTSTACK OP_TOALTSTACK OP_DROP OP_DUP OP_TOALTSTACK OP_IF
  OP_OVER 104 OP_4 OP_SUBSTR OP_CHECKLOCKTIMEVERIFY OP_DROP OP_OVER 72 32
  OP_SUBSTR OP_ELSE OP_OVER 40 32 OP_SUBSTR OP_ENDIF OP_CHECKSIGVERIFY
  OP_SIZE 116 OP_EQUALVERIFY OP_2 <SEL_OUT_SINGLE> OP_TX <0x5120> OP_3
  OP_PICK OP_8 32 OP_SUBSTR OP_CAT OP_EQUALVERIFY OP_OVER OP_8 OP_LEFT
  OP_FROMALTSTACK OP_IF OP_NUMEQUALVERIFY OP_ELSE OP_LESSTHANOREQUAL
  OP_VERIFY OP_ENDIF
LEAF_ACC
<TAG_NODE> OP_ROT OP_ROT <EMPTY_0> OP_0
LEVEL x4
CHECK_PREV
FINAL

# JOIN_0 (dense, D=4): 961 bytes
# macros in CAPITALS are defined in MACROS_FULL.txt; raw opcodes are wrapped and indented

PROLOGUE
<OPERATOR_KEY> OP_CHECKSIGVERIFY OP_SIZE OP_8 OP_EQUALVERIFY OP_OVER OP_SIZE
  32 OP_EQUALVERIFY OP_DROP <TAG_NODE> OP_ROT OP_ROT <EMPTY_0> OP_0 OP_2SWAP
LEVEL x4
CHECK_OLD
FINAL

# JOIN_2 (dense, D=4): 785 bytes
# macros in CAPITALS are defined in MACROS_FULL.txt; raw opcodes are wrapped and indented

PROLOGUE
<OPERATOR_KEY> OP_CHECKSIGVERIFY OP_SIZE OP_8 OP_EQUALVERIFY OP_OVER OP_SIZE
  32 OP_EQUALVERIFY OP_DROP <TAG_NODE> OP_ROT OP_ROT <EMPTY_2> OP_0 OP_2SWAP
LEVEL x2
CHECK_OLD
FINAL

# JOINK_2 (dense, D=4): 1433 bytes
# macros in CAPITALS are defined in MACROS_FULL.txt; raw opcodes are wrapped and indented

PROLOGUE
<OPERATOR_KEY> OP_CHECKSIGVERIFY OP_SIZE OP_8 OP_EQUALVERIFY OP_OVER OP_SIZE
  32 OP_EQUALVERIFY OP_DROP <TAG_NODE> OP_ROT OP_ROT <EMPTY_0> OP_0 OP_2SWAP
LEVEL x4
CHECK_OLD
OP_SWAP OP_TOALTSTACK OP_TOALTSTACK OP_DROP OP_SIZE OP_8 OP_EQUALVERIFY
  OP_OVER OP_SIZE 32 OP_EQUALVERIFY OP_DROP <TAG_NODE> OP_ROT OP_ROT
  <EMPTY_0> OP_0 OP_2SWAP
LEVEL x4
CHECK_PREV
FINAL

# UPDATE (dense, D=4): 1081 bytes
# macros in CAPITALS are defined in MACROS_FULL.txt; raw opcodes are wrapped and indented

PROLOGUE
<OPERATOR_KEY> OP_CHECKSIGVERIFY OP_2 OP_PICK 40 32 OP_SUBSTR
  OP_CHECKSIGVERIFY OP_SIZE 116 OP_EQUALVERIFY OP_SWAP OP_SIZE 116
  OP_EQUALVERIFY <TAG_NODE> OP_ROT OP_ROT

```

```

LEAF_ACC
OP_ROT
LEAF_ACC
LEVEL x4
CHECK_OLD
FINAL

# UPDATE_X (exit-key consent, dense, D=4): 1088 bytes
# macros in CAPITALS are defined in MACROS_FULL.txt; raw opcodes are wrapped and indented

PROLOGUE
EXIT_CONSENT(4)
<OPERATOR_KEY> OP_CHECKSIGVERIFY OP_2 OP_PICK 40 32 OP_SUBSTR
  OP_CHECKSIGVERIFY OP_SIZE 116 OP_EQUALVERIFY OP_SWAP OP_SIZE 116
  OP_EQUALVERIFY <TAG_NODE> OP_ROT OP_ROT
LEAF_ACC
OP_ROT
LEAF_ACC
LEVEL x4
CHECK_OLD
FINAL

# CLAIM (dense, D=4): 1765 bytes
# macros in CAPITALS are defined in MACROS_FULL.txt; raw opcodes are wrapped and indented

PROLOGUE
OP_2 OP_PICK
LEAF_BODY
OP_2 OP_PICK OP_CAT <TREE_ID> OP_SWAP OP_CAT
TAGGED(VOUCHER)
OP_3 OP_PICK 40 32 OP_SUBSTR OP_CHECKSIGFROMSTACK OP_VERIFY
CARVE_RULES
OP_OVER OP_8 OP_LEFT OP_OVER OP_8 OP_LEFT OP_SUB
LE64
OP_2 OP_PICK OP_8 104 OP_SUBSTR OP_CAT OP_2 OP_PICK 112 OP_4 OP_SUBSTR
  OP_1ADD OP_SIZE OP_4 OP_LESSTHANOREQUAL OP_VERIFY
PAD4
OP_CAT OP_FROMALTSTACK OP_ROT OP_TOALTSTACK OP_TOALTSTACK <TAG_NODE> OP_ROT
  OP_ROT OP_SWAP
LEAF_ACC
OP_ROT
LEAF_ACC
DRAIN_RULE
LEVEL x4
CHECK_OLD
OP_FROMALTSTACK OP_2 OP_ROLL OP_2 OP_ROLL OP_SWAP OP_TOALTSTACK OP_TOALTSTACK
LEAF_ACC
<EMPTY_O> OP_0 OP_2SWAP
LEVEL x4
CHECK_PREV
FINAL

# CLAIM_H (hash-locked voucher, dense, D=4): 1773 bytes
# macros in CAPITALS are defined in MACROS_FULL.txt; raw opcodes are wrapped and indented

PROLOGUE
OP_SIZE 32 OP_EQUALVERIFY OP_SHA256 OP_TOALTSTACK OP_2 OP_PICK
LEAF_BODY
OP_2 OP_PICK OP_CAT <TREE_ID> OP_SWAP OP_CAT OP_FROMALTSTACK OP_CAT
  <TAG_VOUCHER_H> OP_SWAP OP_CAT OP_SHA256 OP_3 OP_PICK 40 32 OP_SUBSTR
  OP_CHECKSIGFROMSTACK OP_VERIFY
CARVE_RULES
OP_OVER OP_8 OP_LEFT OP_OVER OP_8 OP_LEFT OP_SUB
LE64
OP_2 OP_PICK OP_8 104 OP_SUBSTR OP_CAT OP_2 OP_PICK 112 OP_4 OP_SUBSTR
  OP_1ADD OP_SIZE OP_4 OP_LESSTHANOREQUAL OP_VERIFY
PAD4

```

```

OP_CAT OP_FROMALTSTACK OP_ROT OP_TOALTSTACK OP_TOALTSTACK <TAG_NODE> OP_ROT
  OP_ROT OP_SWAP
LEAF_ACC
OP_ROT
LEAF_ACC
DRAIN_RULE
LEVEL x4
CHECK_OLD
OP_FROMALTSTACK OP_2 OP_ROLL OP_2 OP_ROLL OP_SWAP OP_TOALTSTACK OP_TOALTSTACK
LEAF_ACC
<EMPTY_O> OP_0 OP_2SWAP
LEVEL x4
CHECK_PREV
FINAL

# CLAIM_X (exit-key consent, dense, D=4): 1772 bytes
# macros in CAPITALS are defined in MACROS_FULL.txt; raw opcodes are wrapped and indented

PROLOGUE
EXIT_CONSENT(3)
OP_2 OP_PICK
LEAF_BODY
OP_2 OP_PICK OP_CAT <TREE_ID> OP_SWAP OP_CAT
TAGGED(VOUCHER)
OP_3 OP_PICK 40 32 OP_SUBSTR OP_CHECKSIGFROMSTACK OP_VERIFY
CARVE_RULES
OP_OVER OP_8 OP_LEFT OP_OVER OP_8 OP_LEFT OP_SUB
LE64
OP_2 OP_PICK OP_8 104 OP_SUBSTR OP_CAT OP_2 OP_PICK 112 OP_4 OP_SUBSTR
  OP_1ADD OP_SIZE OP_4 OP_LESSTHANOREQUAL OP_VERIFY
PAD4
OP_CAT OP_FROMALTSTACK OP_ROT OP_TOALTSTACK OP_TOALTSTACK <TAG_NODE> OP_ROT
  OP_ROT OP_SWAP
LEAF_ACC
OP_ROT
LEAF_ACC
DRAIN_RULE
LEVEL x4
CHECK_OLD
OP_FROMALTSTACK OP_2 OP_ROLL OP_2 OP_ROLL OP_SWAP OP_TOALTSTACK OP_TOALTSTACK
LEAF_ACC
<EMPTY_O> OP_0 OP_2SWAP
LEVEL x4
CHECK_PREV
FINAL

# SPLIT (owner-authorized carve, dense, D=4): 1653 bytes
# macros in CAPITALS are defined in MACROS_FULL.txt; raw opcodes are wrapped and indented

PROLOGUE
OP_2 OP_PICK 40 32 OP_SUBSTR OP_CHECKSIGVERIFY
CARVE_RULES
OP_OVER OP_8 OP_LEFT OP_OVER OP_8 OP_LEFT OP_SUB
LE64
OP_2 OP_PICK OP_8 104 OP_SUBSTR OP_CAT OP_2 OP_PICK 112 OP_4 OP_SUBSTR
  OP_1ADD OP_SIZE OP_4 OP_LESSTHANOREQUAL OP_VERIFY
PAD4
OP_CAT OP_FROMALTSTACK OP_ROT OP_TOALTSTACK OP_TOALTSTACK <TAG_NODE> OP_ROT
  OP_ROT OP_SWAP
LEAF_ACC
OP_ROT
LEAF_ACC
DRAIN_RULE
LEVEL x4
CHECK_OLD
OP_FROMALTSTACK OP_2 OP_ROLL OP_2 OP_ROLL OP_SWAP OP_TOALTSTACK OP_TOALTSTACK
LEAF_ACC

```

```

<EMPTY_0> OP_0 OP_2SWAP
LEVEL    x4
CHECK_PREV
FINAL

```

D.3 FULL Tier, Cohort Shapes ($D = 8$, $h = 2$)

```

# EVICT_SUBTREE_2 (dense, D=8): cohort sweep of a 4-position subtree: 2005 bytes
# macros in CAPITALS are defined in MACROS_FULL.txt; raw opcodes are wrapped and indented

```

```

PROLOGUE
<OPERATOR_KEY> OP_CHECKSIGVERIFY OP_1 OP_TOALTSTACK <TAG_NODE> OP_1 OP_ROLL
LEAF_STEP
OP_3 OP_ROLL
LEAF_STEP
HASH_STEP(4)
OP_3 OP_ROLL
LEAF_STEP
OP_5 OP_ROLL
LEAF_STEP
HASH_STEP(6)
HASH_STEP(4)
OP_FROMALTSTACK OP_DROP <EMPTY_2> OP_0
LEVEL    x6
CHECK_OLD
FINAL

```

```

# REWRITE_2 (dense, D=8): cohort rewrite of a 4-position subtree: 2139 bytes
# macros in CAPITALS are defined in MACROS_FULL.txt; raw opcodes are wrapped and indented

```

```

PROLOGUE
<OPERATOR_KEY> OP_CHECKSIGVERIFY OP_1 OP_TOALTSTACK <EMPTY_0> <TAG_LEAF>
  <TAG_NODE>
REWRITE_STEP(0)
REWRITE_STEP(4)
HASH_STEP2(8)
REWRITE_STEP(4)
REWRITE_STEP(8)
HASH_STEP2(12)
HASH_STEP2(8)
OP_FROMALTSTACK OP_DROP OP_6 OP_ROLL OP_DROP OP_5 OP_ROLL OP_DROP
LEVEL    x6
CHECK_OLD
FINAL

```

```

# DETACH_2 (dense, D=8): detach a 4-position subtree into its own tree output: 1372 bytes
# macros in CAPITALS are defined in MACROS_FULL.txt; raw opcodes are wrapped and indented

```

```

PROLOGUE
<OPERATOR_KEY> OP_CHECKSIGVERIFY OP_SIZE OP_8 OP_EQUALVERIFY OP_OVER OP_SIZE
  32 OP_EQUALVERIFY OP_DROP OP_OVER <M_2 (FULL reference set)>
TREE_SPK
OP_1 <SEL_OUT_SINGLE> OP_TX OP_ROT OP_EQUALVERIFY OP_OVER OP_NUMEQUALVERIFY
  <TAG_NODE> OP_ROT OP_ROT <EMPTY_2> OP_0
LEVEL    x6
CHECK_OLD
FINAL

```

D.4 FULL Tier, Sparse Profile ($D = 8$)

```

# EXIT_1 (sparse, D=8): 1892 bytes
# macros in CAPITALS are defined in MACROS_FULL.txt; raw opcodes are wrapped and indented

```

```

PROLOGUE
OP_DUP OP_TOALTSTACK OP_IF OP_OVER 104 OP_4 OP_SUBSTR OP_CHECKLOCKTIMEVERIFY
  OP_DROP OP_OVER 72 32 OP_SUBSTR OP_ELSE OP_OVER 40 32 OP_SUBSTR OP_ENDIF

```

```

    OP_CHECKSIGVERIFY OP_SIZE 116 OP_EQUALVERIFY OP_1 <SEL_OUT_SINGLE> OP_TX
    <0x5120> OP_3 OP_PICK OP_8 32 OP_SUBSTR OP_CAT OP_EQUALVERIFY OP_OVER OP_8
    OP_LEFT OP_FROMALTSTACK OP_IF OP_NUMEQUALVERIFY OP_ELSE OP_LESSTHANOREQUAL
    OP_VERIFY OP_ENDIF
LEAF_POS
SPARSE_SETUP
LEAF_ACC
<TAG_NODE> OP_ROT OP_ROT <EMPTY_0> OP_0
LEVEL_S(k) for k = 0 to 7
CHECK_OLD_S
FINAL

# JOIN_0 (sparse, D=8): 1876 bytes
# macros in CAPITALS are defined in MACROS_FULL.txt; raw opcodes are wrapped and indented

PROLOGUE
<OPERATOR_KEY> OP_CHECKSIGVERIFY OP_SIZE 116 OP_EQUALVERIFY
LEAF_POS
SPARSE_SETUP
LEAF_ACC
<TAG_NODE> OP_ROT OP_ROT <EMPTY_0> OP_0 OP_2SWAP
LEVEL_S(k) for k = 0 to 7
CHECK_OLD_S
FINAL

# JOINK_2 (sparse, D=8): 3263 bytes
# macros in CAPITALS are defined in MACROS_FULL.txt; raw opcodes are wrapped and indented

PROLOGUE
<OPERATOR_KEY> OP_CHECKSIGVERIFY OP_SIZE 116 OP_EQUALVERIFY
LEAF_POS
SPARSE_SETUP
LEAF_ACC
<TAG_NODE> OP_ROT OP_ROT <EMPTY_0> OP_0 OP_2SWAP
LEVEL_S(k) for k = 0 to 7
CHECK_OLD_S
OP_SWAP OP_TOALTSTACK OP_TOALTSTACK OP_DROP OP_SIZE 116 OP_EQUALVERIFY
LEAF_POS
SPARSE_SETUP
LEAF_ACC
<TAG_NODE> OP_ROT OP_ROT <EMPTY_0> OP_0 OP_2SWAP
LEVEL_S(k) for k = 0 to 7
CHECK_PREV_S
FINAL

# UPDATE (sparse, D=8): 2092 bytes
# macros in CAPITALS are defined in MACROS_FULL.txt; raw opcodes are wrapped and indented

PROLOGUE
<OPERATOR_KEY> OP_CHECKSIGVERIFY OP_2 OP_PICK 40 32 OP_SUBSTR
    OP_CHECKSIGVERIFY OP_SIZE 116 OP_EQUALVERIFY OP_SWAP OP_SIZE 116
    OP_EQUALVERIFY
LEAF_POS
OP_ROT
LEAF_POS
OP_ROT OP_EQUALVERIFY OP_SWAP
LEAF_POS
SPARSE_SETUP
<TAG_NODE> OP_ROT OP_ROT
LEAF_ACC
OP_ROT
LEAF_ACC
LEVEL_S(k) for k = 0 to 7
CHECK_OLD_S
FINAL

# CLAIM (sparse, D=8): 3461 bytes

```

```
# macros in CAPITALS are defined in MACROS_FULL.txt; raw opcodes are wrapped and indented

PROLOGUE
OP_2 OP_PICK
LEAF_BODY
OP_2 OP_PICK OP_CAT <TREE_ID> OP_SWAP OP_CAT
TAGGED(VOUCHER)
OP_3 OP_PICK 40 32 OP_SUBSTR OP_CHECKSIGFROMSTACK OP_VERIFY
CARVE_RULES
OP_OVER OP_8 OP_LEFT OP_OVER OP_8 OP_LEFT OP_SUB
LE64
OP_2 OP_PICK OP_8 104 OP_SUBSTR OP_CAT OP_2 OP_PICK 112 OP_4 OP_SUBSTR
  OP_1ADD OP_SIZE OP_4 OP_LESSTHANOREQUAL OP_VERIFY
PAD4
OP_CAT OP_FROMALTSTACK OP_ROT OP_TOALTSTACK OP_TOALTSTACK OP_OVER
LEAF_POS
OP_NIP
SPARSE_SETUP
<TAG_NODE> OP_ROT OP_ROT OP_SWAP
LEAF_ACC
OP_ROT
LEAF_ACC
DRAIN_RULE
LEVEL_S(k) for k = 0 to 7
CHECK_OLD_S
OP_FROMALTSTACK OP_2 OP_ROLL OP_2 OP_ROLL OP_SWAP OP_TOALTSTACK OP_TOALTSTACK
LEAF_POS
SPARSE_SETUP
LEAF_ACC
<EMPTY_0> OP_0 OP_2SWAP
LEVEL_S(k) for k = 0 to 7
CHECK_PREV_S
FINAL
```

D.5 FULL Numerics Without Splice or EC Opcodes ($D = 4$)

The intermediate data point of §5: BIP-441 numerics kept, while OP_SUBSTR, OP_LEFT, OP_RIGHT and the EC point opcodes are replaced by OP_CAT, OP_SIZE and OP_TWEAKADD.

```
# EXIT_1 (minimal inventory, dense, D=4): 1118 bytes
# macros in CAPITALS are defined in MACROS_FULL.txt; raw opcodes are wrapped and indented

<SEL_CURR_INPUT> OP_TX OP_0 OP_NUMEQUALVERIFY OP_2DUP
TREE_SPK_MIN
<SEL_IN_SPK> OP_TX OP_EQUALVERIFY OP_TOALTSTACK OP_TOALTSTACK OP_DUP
  OP_TOALTSTACK OP_IF OP_3 OP_PICK OP_CHECKLOCKTIMEVERIFY OP_DROP OP_4
  OP_PICK OP_ELSE OP_5 OP_PICK OP_ENDIF OP_CHECKSIGVERIFY OP_SIZE OP_4
  OP_EQUALVERIFY OP_OVER OP_SIZE OP_4 OP_EQUALVERIFY OP_DROP OP_2 OP_PICK
  OP_SIZE OP_4 OP_EQUALVERIFY OP_DROP OP_3 OP_PICK OP_SIZE 32 OP_EQUALVERIFY
  OP_DROP OP_4 OP_PICK OP_SIZE 32 OP_EQUALVERIFY OP_DROP OP_5 OP_PICK OP_SIZE
  32 OP_EQUALVERIFY OP_DROP OP_6 OP_PICK OP_SIZE OP_8 OP_EQUALVERIFY OP_DROP
  OP_6 OP_PICK OP_TOALTSTACK OP_5 OP_PICK OP_TOALTSTACK OP_CAT OP_CAT OP_CAT
  OP_CAT OP_CAT OP_CAT
TAGGED(LEAF)
OP_1 <SEL_OUT_SINGLE> OP_TX <0x5120> OP_FROMALTSTACK OP_CAT OP_EQUALVERIFY
  OP_FROMALTSTACK OP_FROMALTSTACK OP_IF OP_2DUP OP_NUMEQUALVERIFY OP_ELSE
  OP_2DUP OP_LESSTHANOREQUAL OP_VERIFY OP_ENDIF OP_NIP <TAG_NODE> OP_ROT
  OP_ROT <EMPTY_0> OP_0
LEVEL_MIN x4
CHECK_OLD
OP_ROT OP_DROP OP_OVER <EMPTY_4> OP_EQUAL OP_IF OP_2DROP OP_FROMALTSTACK
  OP_DROP OP_1 OP_ELSE OP_SWAP OP_FROMALTSTACK
TREE_SPK_MIN
OP_0 <SEL_OUT_SINGLE> OP_TX OP_ROT OP_EQUALVERIFY OP_NUMEQUAL OP_ENDIF
```

D.6 MID Tier Macros

```

# Macro definitions, MID tier. Each expansion is itself collapsed
# against the other macros; a macro never contains itself.

## TAGGED(LEAF)
<TAG_LEAF> OP_SWAP OP_CAT OP_SHA256

## TAGGED(NODE)
<TAG_NODE> OP_SWAP OP_CAT OP_SHA256

## TAGGED(STATE)
<TAG_STATE> OP_SWAP OP_CAT OP_SHA256

## TAGGED(VOUCHER)
<TAG_VOUCHER> OP_SWAP OP_CAT OP_SHA256

## TAGGED(EXIT)
<TAG_EXIT> OP_SWAP OP_CAT OP_SHA256

## TAGGED(POS)
<TAG_POS> OP_SWAP OP_CAT OP_SHA256

## TAGGED(TAPTWEAK)
<TAG_TAPTWEAK> OP_SWAP OP_CAT OP_SHA256

## TOWER_AUTH_MID ([leaf(10) ea_num ze we sig] -> [leaf(10) ea_num ze we]: sig is a
    BIP-340 signature by ok over H_exit(tree_id || body(leaf)), the leaf body rejoined
    from its six fields after amt8, checked by OP_CHECKSIGFROMSTACK (EXIT_K^t, 7.2 rule 7))
OP_12 OP_PICK OP_12 OP_PICK OP_12 OP_PICK OP_12 OP_PICK OP_12 OP_PICK OP_12
    OP_PICK OP_CAT OP_CAT OP_CAT OP_CAT OP_CAT OP_CAT <TREE_ID> OP_SWAP OP_CAT
TAGGED(EXIT)
OP_12 OP_PICK OP_CHECKSIGFROMSTACK OP_VERIFY

## NONNEG ([x] -> [x]: x == |x|, a minimal non-negative CScriptNum (rejects a negative, a
    negative zero and a non-minimal encoding))
OP_DUP OP_DUP OP_ABS OP_EQUALVERIFY

## PADZ ([x z w] -> [le32(x)]: z||w == 00000000 and size(x)+size(z) == 4)
OP_OVER OP_SWAP OP_CAT <ZERO4> OP_EQUALVERIFY OP_SIZE OP_ROT OP_SIZE OP_ROT
    OP_ADD OP_4 OP_NUMEQUALVERIFY OP_SWAP OP_CAT

## ENC64 ([x z w] -> [enc64(x)])
PADZ
<00000000_01> OP_CAT

## LE64 ([x z w] -> [le64(x)])
PADZ
<ZERO4> OP_CAT

## TREE_SPK_MIN ([r m] -> [TREE_SPK(r, m)] with OP_TWEAKADD)
OP_SWAP
TAGGED(STATE)
<NUMS_X> OP_TWEAKADD OP_TUCK OP_SWAP OP_CAT
TAGGED(TAPTWEAK)
OP_SWAP OP_TWEAKADD <0x5120> OP_SWAP OP_CAT

## PROLOGUE (input is #0; (r, m) match the input scriptPubKey; alt <- m r)
32768 OP_TX OP_0 OP_NUMEQUALVERIFY OP_2DUP
TREE_SPK_MIN
<SEL_IN_SPK> OP_TX OP_EQUALVERIFY OP_TOALTSTACK OP_TOALTSTACK

## LEVEL_MID (dense level with PADZ gadgets)
OP_13 OP_ROLL OP_13 OP_ROLL OP_13 OP_ROLL OP_13 OP_ROLL OP_13 OP_ROLL OP_13
    OP_ROLL OP_13 OP_ROLL OP_13 OP_ROLL OP_13 OP_ROLL OP_2 OP_PICK
NONNEG
OP_TOALTSTACK
ENC64

```

```

OP_CAT OP_FROMALTSTACK OP_7 OP_PICK OP_OVER OP_ADD OP_TOALTSTACK OP_7 OP_ROLL
  OP_5 OP_ROLL OP_5 OP_ROLL
ENC64
OP_6 OP_ROLL OP_SWAP OP_CAT OP_2 OP_PICK OP_SWAP OP_4 OP_PICK OP_NOTIF
  OP_SWAP OP_ENDIF OP_CAT OP_8 OP_PICK OP_SWAP OP_CAT OP_SHA256 OP_TOALTSTACK
  OP_5 OP_PICK OP_OVER OP_ADD OP_TOALTSTACK OP_DROP OP_4 OP_ROLL OP_4 OP_ROLL
  OP_4 OP_ROLL
ENC64
OP_3 OP_ROLL OP_SWAP OP_CAT OP_ROT OP_NOTIF OP_SWAP OP_ENDIF OP_CAT OP_OVER
  OP_SWAP OP_CAT OP_SHA256 OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK

## LEVEL_MID_S (sparse level: dir folded into the position accumulator)
OP_13 OP_ROLL OP_13 OP_ROLL OP_13 OP_ROLL OP_13 OP_ROLL OP_13 OP_ROLL OP_13
  OP_ROLL OP_13 OP_ROLL OP_13 OP_ROLL OP_13 OP_ROLL OP_3 OP_ROLL
  OP_FROMALTSTACK OP_SWAP OP_SIZE OP_ONOTEQUAL OP_NOTIF OP_DROP OP_DUP
  OP_ENDIF OP_SWAP <ENCO> OP_CAT OP_DUP OP_CAT
TAGGED(NODE)
OP_FROMALTSTACK OP_FROMALTSTACK OP_7 OP_PICK OP_IF OP_OVER OP_ADD OP_ENDIF
  OP_SWAP OP_DUP OP_ADD OP_SWAP OP_TOALTSTACK OP_TOALTSTACK OP_TOALTSTACK
  OP_3 OP_ROLL OP_3 OP_ROLL OP_3 OP_ROLL OP_2 OP_PICK
NONNEG
OP_TOALTSTACK
ENC64
OP_CAT OP_FROMALTSTACK OP_7 OP_PICK OP_OVER OP_ADD OP_TOALTSTACK OP_7 OP_ROLL
  OP_5 OP_ROLL OP_5 OP_ROLL
ENC64
OP_6 OP_ROLL OP_SWAP OP_CAT OP_2 OP_PICK OP_SWAP OP_4 OP_PICK OP_NOTIF
  OP_SWAP OP_ENDIF OP_CAT OP_8 OP_PICK OP_SWAP OP_CAT OP_SHA256 OP_TOALTSTACK
  OP_5 OP_PICK OP_OVER OP_ADD OP_TOALTSTACK OP_DROP OP_4 OP_ROLL OP_4 OP_ROLL
  OP_4 OP_ROLL
ENC64
OP_3 OP_ROLL OP_SWAP OP_CAT OP_ROT OP_NOTIF OP_SWAP OP_ENDIF OP_CAT OP_OVER
  OP_SWAP OP_CAT OP_SHA256 OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK

## SPARSE_SETUP ([ok nonce plo phi] -> [] ; alt <- plo 0 1 E_0)
OP_2SWAP OP_CAT
TAGGED(POS)
OP_ROT OP_DUP OP_TOALTSTACK OP_SIZE OP_4 OP_EQUALVERIFY OP_ROT OP_CAT
  OP_EQUALVERIFY OP_0 OP_TOALTSTACK OP_1 OP_TOALTSTACK <EMPTY_0>
  OP_TOALTSTACK

## SPARSE_TEARDOWN (check the accumulated position against plo; drop bookkeeping)
OP_FROMALTSTACK OP_DROP OP_FROMALTSTACK OP_DROP OP_FROMALTSTACK
  OP_FROMALTSTACK OP_NUMEQUALVERIFY

## CHECK_OLD (os (padded) == input 0 amount, oh == r)
OP_2SWAP OP_6 OP_ROLL OP_6 OP_ROLL
LE64
<SEL_IN_AMOUNT> OP_TX OP_EQUALVERIFY OP_FROMALTSTACK OP_EQUALVERIFY

## CHECK_OLD_S (CHECK_OLD with the sparse teardown)
OP_2SWAP OP_6 OP_ROLL OP_6 OP_ROLL
LE64
<SEL_IN_AMOUNT> OP_TX OP_EQUALVERIFY
SPARSE_TEARDOWN
OP_FROMALTSTACK OP_EQUALVERIFY

## CHECK_PREV (os == ns_prev, oh == nh_prev; chained steps)
OP_2SWAP OP_FROMALTSTACK OP_NUMEQUALVERIFY OP_FROMALTSTACK OP_EQUALVERIFY

## CHECK_PREV_S (CHECK_PREV with the sparse teardown)
OP_2SWAP
SPARSE_TEARDOWN
OP_FROMALTSTACK OP_NUMEQUALVERIFY OP_FROMALTSTACK OP_EQUALVERIFY

## LEAF_JOIN ([amt8 ek ok evk ea nonce seq amt_num za wa] -> [leaf] | amt_num)
OP_2 OP_PICK
NONNEG

```

```

OP_TOALTSTACK
LE64
OP_7 OP_PICK OP_EQUALVERIFY OP_DUP OP_SIZE OP_4 OP_EQUALVERIFY OP_DROP OP_1
  OP_PICK OP_SIZE OP_4 OP_EQUALVERIFY OP_DROP OP_2 OP_PICK OP_SIZE OP_4
  OP_EQUALVERIFY OP_DROP OP_3 OP_PICK OP_SIZE 32 OP_EQUALVERIFY OP_DROP OP_4
  OP_PICK OP_SIZE 32 OP_EQUALVERIFY OP_DROP OP_5 OP_PICK OP_SIZE 32
  OP_EQUALVERIFY OP_DROP OP_6 OP_PICK OP_SIZE OP_8 OP_EQUALVERIFY OP_DROP
  OP_CAT OP_CAT OP_CAT OP_CAT OP_CAT OP_CAT

## CARVE_RULES  ([src(10) claim(10) extras(7) sea zse wse cea zce wce] -> [src(10)
  claim(10) extras(7)]: claim.evk == src.evk and claim.evict_after <= src.evict_after,
  each evict_after a minimal number checked against its field by PADZ and the claim's
  NONNEG-checked before the compare (7.5 rule 5, 10.6), and claim.amt_num >= DUST_MIN
  (rule 6))
19 OP_PICK 30 OP_PICK OP_EQUALVERIFY OP_2 OP_PICK OP_TOALTSTACK
PADZ
OP_16 OP_PICK OP_EQUALVERIFY OP_2 OP_PICK OP_TOALTSTACK
PADZ
23 OP_PICK OP_EQUALVERIFY OP_FROMALTSTACK OP_FROMALTSTACK
NONNEG
OP_GREATERTHANOREQUAL OP_VERIFY 330 OP_10 OP_PICK OP_LESSTHANOREQUAL
  OP_VERIFY

## DRAIN_RULE  ([.. nh ns] -> [.. nh' ns]: a zero residual makes the new node E_0, any
  other is at least DUST_MIN (7.5 rules 2 and 6))
OP_DUP OP_NOT OP_IF OP_SWAP OP_DROP <EMPTY_0> OP_SWAP OP_ELSE 330 OP_OVER
  OP_LESSTHANOREQUAL OP_VERIFY OP_ENDIF

## FINAL  [shown for D = 4; <EMPTY_D> is the empty root of the tree's depth]  (output 0 is
  (le64(ns), TREE_SPK(nh, m)) unless nh == E_D)
OP_ROT OP_DROP OP_OVER <EMPTY_4> OP_EQUAL OP_IF OP_2DROP OP_2DROP
  OP_FROMALTSTACK OP_DROP OP_1 OP_ELSE OP_3 OP_ROLL OP_3 OP_ROLL
LE64
OP_SWAP OP_FROMALTSTACK
TREE_SPK_MIN
OP_0 <SEL_OUT_SINGLE> OP_TX OP_ROT OP_EQUALVERIFY OP_EQUAL OP_ENDIF

## EVICT_LEAF_STEP  (cohort sweep pre-pass: [slot idx] -> [idx'] | s sc, paying the next
  output)
OP_SWAP OP_IF OP_TOALTSTACK OP_2 OP_PICK OP_CHECKLOCKTIMEVERIFY OP_DROP
PADZ
OP_6 OP_PICK OP_EQUALVERIFY OP_6 OP_PICK <OPERATOR_KEY> OP_EQUALVERIFY
  OP_FROMALTSTACK OP_DUP OP_1ADD OP_TOALTSTACK <SEL_OUT_SINGLE> OP_TX
  <0x5120> OP_11 OP_PICK OP_CAT OP_EQUALVERIFY OP_10 OP_PICK OP_EQUALVERIFY
  OP_9 OP_PICK OP_1 OP_CAT OP_TOALTSTACK
LEAF_JOIN
TAGGED(LEAF)
OP_FROMALTSTACK OP_FROMALTSTACK OP_ROT OP_SWAP OP_CAT OP_FROMALTSTACK OP_ROT
  OP_TOALTSTACK OP_SWAP OP_TOALTSTACK OP_ELSE OP_0 OP_TOALTSTACK <E_0.sc>
  OP_TOALTSTACK OP_ENDIF

## EVICT_HASH_STEP(4)
EVICT_HASH_ROOT(4)
OP_DUP OP_5 OP_ROLL OP_5 OP_ROLL
ENC64
OP_ROT OP_SWAP OP_CAT OP_SWAP

## EVICT_HASH_STEP(5)
EVICT_HASH_ROOT(5)
OP_DUP OP_6 OP_ROLL OP_6 OP_ROLL
ENC64
OP_ROT OP_SWAP OP_CAT OP_SWAP

## EVICT_HASH_STEP(6)
OP_2 OP_ROLL OP_ADD OP_TOALTSTACK OP_SWAP OP_CAT OP_3 OP_PICK OP_SWAP OP_CAT
  OP_SHA256 OP_FROMALTSTACK OP_DUP OP_7 OP_ROLL OP_7 OP_ROLL
ENC64
OP_ROT OP_SWAP OP_CAT OP_SWAP

```

```

## EVICT_HASH_STEP(7)
OP_2 OP_ROLL OP_ADD OP_TOALTSTACK OP_SWAP OP_CAT OP_4 OP_PICK OP_SWAP OP_CAT
  OP_SHA256 OP_FROMALTSTACK OP_DUP OP_8 OP_ROLL OP_8 OP_ROLL
ENC64
OP_ROT OP_SWAP OP_CAT OP_SWAP

## EVICT_HASH_STEP(8)
OP_2 OP_ROLL OP_ADD OP_TOALTSTACK OP_SWAP OP_CAT OP_5 OP_PICK OP_SWAP OP_CAT
  OP_SHA256 OP_FROMALTSTACK OP_DUP OP_9 OP_ROLL OP_9 OP_ROLL
ENC64
OP_ROT OP_SWAP OP_CAT OP_SWAP

## EVICT_HASH_STEP(9)
OP_2 OP_ROLL OP_ADD OP_TOALTSTACK OP_SWAP OP_CAT OP_6 OP_PICK OP_SWAP OP_CAT
  OP_SHA256 OP_FROMALTSTACK OP_DUP OP_10 OP_ROLL OP_10 OP_ROLL
ENC64
OP_ROT OP_SWAP OP_CAT OP_SWAP

## EVICT_HASH_STEP(10)
OP_2 OP_ROLL OP_ADD OP_TOALTSTACK OP_SWAP OP_CAT OP_7 OP_PICK OP_SWAP OP_CAT
  OP_SHA256 OP_FROMALTSTACK OP_DUP OP_11 OP_ROLL OP_11 OP_ROLL
ENC64
OP_ROT OP_SWAP OP_CAT OP_SWAP

## EVICT_HASH_STEP(11)
OP_2 OP_ROLL OP_ADD OP_TOALTSTACK OP_SWAP OP_CAT OP_8 OP_PICK OP_SWAP OP_CAT
  OP_SHA256 OP_FROMALTSTACK OP_DUP OP_12 OP_ROLL OP_12 OP_ROLL
ENC64
OP_ROT OP_SWAP OP_CAT OP_SWAP

## EVICT_HASH_STEP(12)
OP_2 OP_ROLL OP_ADD OP_TOALTSTACK OP_SWAP OP_CAT OP_9 OP_PICK OP_SWAP OP_CAT
  OP_SHA256 OP_FROMALTSTACK OP_DUP OP_13 OP_ROLL OP_13 OP_ROLL
ENC64
OP_ROT OP_SWAP OP_CAT OP_SWAP

## EVICT_HASH_STEP(13)
OP_2 OP_ROLL OP_ADD OP_TOALTSTACK OP_SWAP OP_CAT OP_10 OP_PICK OP_SWAP OP_CAT
  OP_SHA256 OP_FROMALTSTACK OP_DUP OP_14 OP_ROLL OP_14 OP_ROLL
ENC64
OP_ROT OP_SWAP OP_CAT OP_SWAP

## EVICT_HASH_STEP(14)
OP_2 OP_ROLL OP_ADD OP_TOALTSTACK OP_SWAP OP_CAT OP_11 OP_PICK OP_SWAP OP_CAT
  OP_SHA256 OP_FROMALTSTACK OP_DUP OP_15 OP_ROLL OP_15 OP_ROLL
ENC64
OP_ROT OP_SWAP OP_CAT OP_SWAP

## EVICT_HASH_STEP(15)
OP_2 OP_ROLL OP_ADD OP_TOALTSTACK OP_SWAP OP_CAT OP_12 OP_PICK OP_SWAP OP_CAT
  OP_SHA256 OP_FROMALTSTACK OP_DUP OP_16 OP_ROLL OP_16 OP_ROLL
ENC64
OP_ROT OP_SWAP OP_CAT OP_SWAP

## EVICT_HASH_STEP(16)
OP_2 OP_ROLL OP_ADD OP_TOALTSTACK OP_SWAP OP_CAT OP_13 OP_PICK OP_SWAP OP_CAT
  OP_SHA256 OP_FROMALTSTACK OP_DUP 17 OP_ROLL 17 OP_ROLL
ENC64
OP_ROT OP_SWAP OP_CAT OP_SWAP

## EVICT_HASH_STEP(17)
OP_2 OP_ROLL OP_ADD OP_TOALTSTACK OP_SWAP OP_CAT OP_14 OP_PICK OP_SWAP OP_CAT
  OP_SHA256 OP_FROMALTSTACK OP_DUP 18 OP_ROLL 18 OP_ROLL
ENC64
OP_ROT OP_SWAP OP_CAT OP_SWAP

## EVICT_HASH_STEP(18)

```

```

OP_2 OP_ROLL OP_ADD OP_TOALTSTACK OP_SWAP OP_CAT OP_15 OP_PICK OP_SWAP OP_CAT
  OP_SHA256 OP_FROMALTSTACK OP_DUP 19 OP_ROLL 19 OP_ROLL
ENC64
OP_ROT OP_SWAP OP_CAT OP_SWAP

## EVICT_HASH_STEP(19)
OP_2 OP_ROLL OP_ADD OP_TOALTSTACK OP_SWAP OP_CAT OP_16 OP_PICK OP_SWAP OP_CAT
  OP_SHA256 OP_FROMALTSTACK OP_DUP 20 OP_ROLL 20 OP_ROLL
ENC64
OP_ROT OP_SWAP OP_CAT OP_SWAP

## EVICT_HASH_STEP(20)
OP_2 OP_ROLL OP_ADD OP_TOALTSTACK OP_SWAP OP_CAT 17 OP_PICK OP_SWAP OP_CAT
  OP_SHA256 OP_FROMALTSTACK OP_DUP 21 OP_ROLL 21 OP_ROLL
ENC64
OP_ROT OP_SWAP OP_CAT OP_SWAP

## EVICT_HASH_STEP(21)
OP_2 OP_ROLL OP_ADD OP_TOALTSTACK OP_SWAP OP_CAT 18 OP_PICK OP_SWAP OP_CAT
  OP_SHA256 OP_FROMALTSTACK OP_DUP 22 OP_ROLL 22 OP_ROLL
ENC64
OP_ROT OP_SWAP OP_CAT OP_SWAP

## EVICT_HASH_STEP(22)
OP_2 OP_ROLL OP_ADD OP_TOALTSTACK OP_SWAP OP_CAT 19 OP_PICK OP_SWAP OP_CAT
  OP_SHA256 OP_FROMALTSTACK OP_DUP 23 OP_ROLL 23 OP_ROLL
ENC64
OP_ROT OP_SWAP OP_CAT OP_SWAP

## EVICT_HASH_STEP(23)
OP_2 OP_ROLL OP_ADD OP_TOALTSTACK OP_SWAP OP_CAT 20 OP_PICK OP_SWAP OP_CAT
  OP_SHA256 OP_FROMALTSTACK OP_DUP 24 OP_ROLL 24 OP_ROLL
ENC64
OP_ROT OP_SWAP OP_CAT OP_SWAP

## EVICT_HASH_STEP(24)
OP_2 OP_ROLL OP_ADD OP_TOALTSTACK OP_SWAP OP_CAT 21 OP_PICK OP_SWAP OP_CAT
  OP_SHA256 OP_FROMALTSTACK OP_DUP 25 OP_ROLL 25 OP_ROLL
ENC64
OP_ROT OP_SWAP OP_CAT OP_SWAP

## EVICT_HASH_STEP(25)
OP_2 OP_ROLL OP_ADD OP_TOALTSTACK OP_SWAP OP_CAT 22 OP_PICK OP_SWAP OP_CAT
  OP_SHA256 OP_FROMALTSTACK OP_DUP 26 OP_ROLL 26 OP_ROLL
ENC64
OP_ROT OP_SWAP OP_CAT OP_SWAP

## EVICT_HASH_STEP(26)
OP_2 OP_ROLL OP_ADD OP_TOALTSTACK OP_SWAP OP_CAT 23 OP_PICK OP_SWAP OP_CAT
  OP_SHA256 OP_FROMALTSTACK OP_DUP 27 OP_ROLL 27 OP_ROLL
ENC64
OP_ROT OP_SWAP OP_CAT OP_SWAP

## EVICT_HASH_STEP(27)
OP_2 OP_ROLL OP_ADD OP_TOALTSTACK OP_SWAP OP_CAT 24 OP_PICK OP_SWAP OP_CAT
  OP_SHA256 OP_FROMALTSTACK OP_DUP 28 OP_ROLL 28 OP_ROLL
ENC64
OP_ROT OP_SWAP OP_CAT OP_SWAP

## EVICT_HASH_STEP(28)
OP_2 OP_ROLL OP_ADD OP_TOALTSTACK OP_SWAP OP_CAT 25 OP_PICK OP_SWAP OP_CAT
  OP_SHA256 OP_FROMALTSTACK OP_DUP 29 OP_ROLL 29 OP_ROLL
ENC64
OP_ROT OP_SWAP OP_CAT OP_SWAP

## EVICT_HASH_STEP(29)
OP_2 OP_ROLL OP_ADD OP_TOALTSTACK OP_SWAP OP_CAT 26 OP_PICK OP_SWAP OP_CAT
  OP_SHA256 OP_FROMALTSTACK OP_DUP 30 OP_ROLL 30 OP_ROLL

```

```

ENC64
OP_ROT OP_SWAP OP_CAT OP_SWAP

## EVICT_HASH_STEP(30)
OP_2 OP_ROLL OP_ADD OP_TOALTSTACK OP_SWAP OP_CAT 27 OP_PICK OP_SWAP OP_CAT
  OP_SHA256 OP_FROMALTSTACK OP_DUP 31 OP_ROLL 31 OP_ROLL
ENC64
OP_ROT OP_SWAP OP_CAT OP_SWAP

## EVICT_HASH_STEP(31)
OP_2 OP_ROLL OP_ADD OP_TOALTSTACK OP_SWAP OP_CAT 28 OP_PICK OP_SWAP OP_CAT
  OP_SHA256 OP_FROMALTSTACK OP_DUP 32 OP_ROLL 32 OP_ROLL
ENC64
OP_ROT OP_SWAP OP_CAT OP_SWAP

## EVICT_HASH_STEP(32)
OP_2 OP_ROLL OP_ADD OP_TOALTSTACK OP_SWAP OP_CAT 29 OP_PICK OP_SWAP OP_CAT
  OP_SHA256 OP_FROMALTSTACK OP_DUP 33 OP_ROLL 33 OP_ROLL
ENC64
OP_ROT OP_SWAP OP_CAT OP_SWAP

## EVICT_HASH_STEP(33)
OP_2 OP_ROLL OP_ADD OP_TOALTSTACK OP_SWAP OP_CAT 30 OP_PICK OP_SWAP OP_CAT
  OP_SHA256 OP_FROMALTSTACK OP_DUP 34 OP_ROLL 34 OP_ROLL
ENC64
OP_ROT OP_SWAP OP_CAT OP_SWAP

## EVICT_HASH_STEP(34)
OP_2 OP_ROLL OP_ADD OP_TOALTSTACK OP_SWAP OP_CAT 31 OP_PICK OP_SWAP OP_CAT
  OP_SHA256 OP_FROMALTSTACK OP_DUP 35 OP_ROLL 35 OP_ROLL
ENC64
OP_ROT OP_SWAP OP_CAT OP_SWAP

## EVICT_HASH_STEP(35)
OP_2 OP_ROLL OP_ADD OP_TOALTSTACK OP_SWAP OP_CAT 32 OP_PICK OP_SWAP OP_CAT
  OP_SHA256 OP_FROMALTSTACK OP_DUP 36 OP_ROLL 36 OP_ROLL
ENC64
OP_ROT OP_SWAP OP_CAT OP_SWAP

## EVICT_HASH_STEP(36)
OP_2 OP_ROLL OP_ADD OP_TOALTSTACK OP_SWAP OP_CAT 33 OP_PICK OP_SWAP OP_CAT
  OP_SHA256 OP_FROMALTSTACK OP_DUP 37 OP_ROLL 37 OP_ROLL
ENC64
OP_ROT OP_SWAP OP_CAT OP_SWAP

## EVICT_HASH_STEP(37)
OP_2 OP_ROLL OP_ADD OP_TOALTSTACK OP_SWAP OP_CAT 34 OP_PICK OP_SWAP OP_CAT
  OP_SHA256 OP_FROMALTSTACK OP_DUP 38 OP_ROLL 38 OP_ROLL
ENC64
OP_ROT OP_SWAP OP_CAT OP_SWAP

## EVICT_HASH_STEP(38)
OP_2 OP_ROLL OP_ADD OP_TOALTSTACK OP_SWAP OP_CAT 35 OP_PICK OP_SWAP OP_CAT
  OP_SHA256 OP_FROMALTSTACK OP_DUP 39 OP_ROLL 39 OP_ROLL
ENC64
OP_ROT OP_SWAP OP_CAT OP_SWAP

## EVICT_HASH_STEP(39)
OP_2 OP_ROLL OP_ADD OP_TOALTSTACK OP_SWAP OP_CAT 36 OP_PICK OP_SWAP OP_CAT
  OP_SHA256 OP_FROMALTSTACK OP_DUP 40 OP_ROLL 40 OP_ROLL
ENC64
OP_ROT OP_SWAP OP_CAT OP_SWAP

## EVICT_HASH_ROOT(4)
OP_2 OP_ROLL OP_ADD OP_TOALTSTACK OP_SWAP OP_CAT OP_1 OP_PICK OP_SWAP OP_CAT
  OP_SHA256 OP_FROMALTSTACK

## EVICT_HASH_ROOT(5)

```

```

OP_2 OP_ROLL OP_ADD OP_TOALTSTACK OP_SWAP OP_CAT OP_2 OP_PICK OP_SWAP OP_CAT
  OP_SHA256 OP_FROMALTSTACK

## LEAF_JOIN_OLD (cohort rewrite: [amt8 ek ok evk ea nonce seq amt_num za wa] -> [amt8
  rest ek ok fin amt], fin = (seq == SEQ_FINAL))
OP_2 OP_PICK
NONNEG
OP_TOALTSTACK
LE64
OP_7 OP_PICK OP_EQUALVERIFY OP_DUP OP_SIZE OP_4 OP_EQUALVERIFY OP_DROP OP_1
  OP_PICK OP_SIZE OP_4 OP_EQUALVERIFY OP_DROP OP_2 OP_PICK OP_SIZE OP_4
  OP_EQUALVERIFY OP_DROP OP_3 OP_PICK OP_SIZE 32 OP_EQUALVERIFY OP_DROP OP_4
  OP_PICK OP_SIZE 32 OP_EQUALVERIFY OP_DROP OP_5 OP_PICK OP_SIZE 32
  OP_EQUALVERIFY OP_DROP OP_6 OP_PICK OP_SIZE OP_8 OP_EQUALVERIFY OP_DROP
  OP_DUP 2147483647 OP_EQUAL OP_TOALTSTACK OP_5 OP_PICK OP_5 OP_PICK
  OP_TOALTSTACK OP_TOALTSTACK OP_CAT OP_CAT OP_CAT OP_CAT OP_CAT
  OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK

## LEAF_JOIN_NEW (cohort rewrite: [amt8 ek ok evk ea nonce seq amt_num za wa] -> [amt8
  rest amt])
OP_2 OP_PICK
NONNEG
OP_TOALTSTACK
LE64
OP_7 OP_PICK OP_EQUALVERIFY OP_DUP OP_SIZE OP_4 OP_EQUALVERIFY OP_DROP OP_1
  OP_PICK OP_SIZE OP_4 OP_EQUALVERIFY OP_DROP OP_2 OP_PICK OP_SIZE OP_4
  OP_EQUALVERIFY OP_DROP OP_3 OP_PICK OP_SIZE 32 OP_EQUALVERIFY OP_DROP OP_4
  OP_PICK OP_SIZE 32 OP_EQUALVERIFY OP_DROP OP_5 OP_PICK OP_SIZE 32
  OP_EQUALVERIFY OP_DROP OP_6 OP_PICK OP_SIZE OP_8 OP_EQUALVERIFY OP_DROP
  OP_CAT OP_CAT OP_CAT OP_CAT OP_CAT OP_FROMALTSTACK

## REWRITE_CONV (cohort rewrite: [idx L Z amt8 rest] -> [idx L Z sc], a leaf's 41-byte
  node or Z (E_0) for the empty element)
OP_SIZE OP_ONOTEQUAL OP_IF OP_2DUP OP_CAT OP_4 OP_PICK OP_SWAP OP_CAT
  OP_SHA256 OP_NIP OP_SWAP OP_1 OP_CAT OP_CAT OP_ELSE OP_2DROP OP_DUP
  OP_ENDIF

## REWRITE_PAY (cohort rewrite: output idx pays exactly old.amt8 to P2TR(old.ek), idx
  advances (7.9 rule 5))
OP_FROMALTSTACK OP_DUP OP_1ADD OP_TOALTSTACK <SEL_OUT_SINGLE> OP_TX <0x5120>
  OP_6 OP_PICK OP_CAT OP_EQUALVERIFY OP_6 OP_PICK OP_EQUALVERIFY

## OWNER_AUTH_MID (cohort rewrite: the next witness element (nine deep) is a signature by
  old.ok)
OP_9 OP_ROLL OP_3 OP_PICK OP_CHECKSIGVERIFY

## OWNER_AUTH_MID_X (OWNER_AUTH_MID plus a signature by old.ek (REWRITE_h`x))
OWNER_AUTH_MID
OP_9 OP_ROLL OP_4 OP_PICK OP_CHECKSIGVERIFY

## REWRITE_STEP_MID (cohort rewrite pre-pass: [new_slot old_slot idx L Z] -> [idx' L Z] |
  ns nsc os osc, with 7.9 rules 3 and 5)
OP_TOALTSTACK OP_TOALTSTACK OP_TOALTSTACK OP_IF
LEAF_JOIN_OLD
OP_ELSE OP_0 OP_0 OP_0 OP_0 OP_0 OP_0 OP_ENDIF OP_6 OP_PICK OP_2 OP_EQUAL
  OP_IF OP_6 OP_ROLL OP_DROP OP_TOALTSTACK OP_2DROP OP_DROP OP_FROMALTSTACK
  OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK OP_5 OP_ROLL OP_5 OP_ROLL
REWRITE_CONV
OP_4 OP_ROLL OP_2DUP OP_TOALTSTACK OP_TOALTSTACK OP_TOALTSTACK OP_TOALTSTACK
  OP_ELSE OP_6 OP_PICK OP_3 OP_EQUAL OP_IF OP_6 OP_ROLL OP_DROP OP_8 OP_ROLL
  OP_8 OP_ROLL OP_8 OP_ROLL OP_2 OP_PICK
NONNEG
OP_TOALTSTACK
LE64
OP_5 OP_PICK OP_SIZE OP_ONOTEQUAL OP_VERIFY OP_FROMALTSTACK OP_8 OP_ROLL OP_8
  OP_ROLL OP_8 OP_ROLL OP_8 OP_ROLL OP_8 OP_ROLL OP_8 OP_ROLL OP_ELSE
  OP_TOALTSTACK OP_TOALTSTACK OP_TOALTSTACK OP_TOALTSTACK OP_TOALTSTACK
  OP_TOALTSTACK OP_IF

```

```

LEAF_JOIN_NEW
OP_ELSE OP_0 OP_0 OP_0 OP_ENDIF OP_FROMALTSTACK OP_FROMALTSTACK
  OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK OP_ENDIF
  OP_4 OP_PICK OP_8 OP_PICK OP_EQUAL OP_6 OP_PICK OP_10 OP_PICK OP_EQUAL
  OP_BOOLAND OP_NOTIF OP_4 OP_PICK OP_SIZE OP_NIP OP_ONOTEQUAL OP_IF OP_7
  OP_PICK OP_SIZE OP_NIP OP_ONOTEQUAL OP_IF OP_4 OP_PICK OP_8 OP_PICK
  OP_EQUAL OP_OVER OP_8 OP_PICK OP_LESSTHANOREQUAL OP_BOOLAND OP_2 OP_PICK
  OP_NOT OP_BOOLAND OP_NOTIF
OWNER_AUTH_MID
OP_ENDIF OP_ELSE
OWNER_AUTH_MID
REWRITE_PAY
OP_ENDIF OP_ENDIF OP_ENDIF OP_TOALTSTACK OP_2DROP OP_DROP OP_FROMALTSTACK
  OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK OP_8 OP_ROLL OP_8 OP_ROLL
REWRITE_CONV
OP_7 OP_ROLL OP_TOALTSTACK OP_TOALTSTACK OP_5 OP_ROLL OP_5 OP_ROLL
REWRITE_CONV
OP_4 OP_ROLL OP_TOALTSTACK OP_TOALTSTACK OP_ENDIF

## REWRITE_STEP_MID_X (REWRITE_STEP_MID with OWNER_AUTH_MID_X)
OP_TOALTSTACK OP_TOALTSTACK OP_TOALTSTACK OP_IF
LEAF_JOIN_OLD
OP_ELSE OP_0 OP_0 OP_0 OP_0 OP_0 OP_0 OP_0 OP_ENDIF OP_6 OP_PICK OP_2 OP_EQUAL
  OP_IF OP_6 OP_ROLL OP_DROP OP_TOALTSTACK OP_2DROP OP_DROP OP_FROMALTSTACK
  OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK OP_5 OP_ROLL OP_5 OP_ROLL
REWRITE_CONV
OP_4 OP_ROLL OP_2DUP OP_TOALTSTACK OP_TOALTSTACK OP_TOALTSTACK OP_TOALTSTACK
  OP_ELSE OP_6 OP_PICK OP_3 OP_EQUAL OP_IF OP_6 OP_ROLL OP_DROP OP_8 OP_ROLL
  OP_8 OP_ROLL OP_8 OP_ROLL OP_2 OP_PICK
NONNEG
OP_TOALTSTACK
LE64
OP_5 OP_PICK OP_SIZE OP_ONOTEQUAL OP_VERIFY OP_FROMALTSTACK OP_8 OP_ROLL OP_8
  OP_ROLL OP_8 OP_ROLL OP_8 OP_ROLL OP_8 OP_ROLL OP_8 OP_ROLL OP_ELSE
  OP_TOALTSTACK OP_TOALTSTACK OP_TOALTSTACK OP_TOALTSTACK OP_TOALTSTACK
  OP_TOALTSTACK OP_IF
LEAF_JOIN_NEW
OP_ELSE OP_0 OP_0 OP_0 OP_0 OP_ENDIF OP_FROMALTSTACK OP_FROMALTSTACK
  OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK OP_ENDIF
  OP_4 OP_PICK OP_8 OP_PICK OP_EQUAL OP_6 OP_PICK OP_10 OP_PICK OP_EQUAL
  OP_BOOLAND OP_NOTIF OP_4 OP_PICK OP_SIZE OP_NIP OP_ONOTEQUAL OP_IF OP_7
  OP_PICK OP_SIZE OP_NIP OP_ONOTEQUAL OP_IF OP_4 OP_PICK OP_8 OP_PICK
  OP_EQUAL OP_OVER OP_8 OP_PICK OP_LESSTHANOREQUAL OP_BOOLAND OP_2 OP_PICK
  OP_NOT OP_BOOLAND OP_NOTIF
OWNER_AUTH_MID_X
OP_ENDIF OP_ELSE
OWNER_AUTH_MID_X
REWRITE_PAY
OP_ENDIF OP_ENDIF OP_ENDIF OP_TOALTSTACK OP_2DROP OP_DROP OP_FROMALTSTACK
  OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK OP_8 OP_ROLL OP_8 OP_ROLL
REWRITE_CONV
OP_7 OP_ROLL OP_TOALTSTACK OP_TOALTSTACK OP_5 OP_ROLL OP_5 OP_ROLL
REWRITE_CONV
OP_4 OP_ROLL OP_TOALTSTACK OP_TOALTSTACK OP_ENDIF

## REWRITE_HASH_STEP2(t) [shown for t = 8; the EVICT_HASH_STEP depths are t and t-2]
(cohort rewrite: [Ros Ros Rnsc Rns Losc Los Lnsc Lns] -> [posc pos pns pns], two pad
  pairs from below T)
OP_2SWAP OP_7 OP_ROLL OP_7 OP_ROLL OP_2SWAP
EVICT_HASH_STEP(8)
OP_2SWAP OP_5 OP_ROLL OP_5 OP_ROLL OP_2SWAP
EVICT_HASH_STEP(6)

## REWRITE_HASH_ROOT2(t) [shown for t = 8; the EVICT_HASH_STEP depths are t and t-2]
(cohort rewrite: the same at the subtree root, -> [oh os nh ns])
OP_2SWAP OP_7 OP_ROLL OP_7 OP_ROLL OP_2SWAP OP_2 OP_ROLL OP_ADD OP_TOALTSTACK
  OP_SWAP OP_CAT OP_5 OP_PICK OP_SWAP OP_CAT OP_SHA256 OP_FROMALTSTACK
  OP_2SWAP OP_5 OP_ROLL OP_5 OP_ROLL OP_2SWAP OP_2 OP_ROLL OP_ADD

```

```
OP_TOALTSTACK OP_SWAP OP_CAT OP_3 OP_PICK OP_SWAP OP_CAT OP_SHA256
OP_FROMALTSTACK
```

```
## EXIT_CONSENT_MID(k) [shown for k = 21 (UPDATE^x); 33 for CLAIM^x, 34 for CLAIM^hx]
21 OP_PICK OP_CHECKSIGVERIFY
```

D.7 MID Tier, Dense Profile ($D = 4$)

```
# EXIT_1 (MID tier, dense, D=4): 1447 bytes
# macros in CAPITALS are defined in MACROS_MID.txt; raw opcodes are wrapped and indented
```

```
PROLOGUE
OP_DUP OP_TOALTSTACK OP_IF OP_3 OP_PICK OP_CHECKLOCKTIMEVERIFY OP_DROP OP_10
  OP_PICK OP_CHECKSIGVERIFY
PADZ
OP_6 OP_PICK OP_EQUALVERIFY OP_ELSE OP_11 OP_PICK OP_CHECKSIGVERIFY OP_2DROP
  OP_DROP OP_ENDIF OP_1 <SEL_OUT_SINGLE> OP_TX <0x5120> OP_11 OP_PICK OP_CAT
  OP_EQUALVERIFY OP_10 OP_PICK OP_EQUALVERIFY OP_FROMALTSTACK OP_DROP
LEAF_JOIN
TAGGED(LEAF)
OP_FROMALTSTACK <TAG_NODE> OP_ROT OP_ROT <EMPTY_0> OP_0
LEVEL_MID x4
CHECK_OLD
FINAL
```

```
# EXIT_2 (MID tier, dense, D=4): 2405 bytes
# macros in CAPITALS are defined in MACROS_MID.txt; raw opcodes are wrapped and indented
```

```
PROLOGUE
OP_DUP OP_TOALTSTACK OP_IF OP_3 OP_PICK OP_CHECKLOCKTIMEVERIFY OP_DROP OP_10
  OP_PICK OP_CHECKSIGVERIFY
PADZ
OP_6 OP_PICK OP_EQUALVERIFY OP_ELSE OP_11 OP_PICK OP_CHECKSIGVERIFY OP_2DROP
  OP_DROP OP_ENDIF OP_1 <SEL_OUT_SINGLE> OP_TX <0x5120> OP_11 OP_PICK OP_CAT
  OP_EQUALVERIFY OP_10 OP_PICK OP_EQUALVERIFY OP_FROMALTSTACK OP_DROP
LEAF_JOIN
TAGGED(LEAF)
OP_FROMALTSTACK <TAG_NODE> OP_ROT OP_ROT <EMPTY_0> OP_0
LEVEL_MID x4
CHECK_OLD
OP_SWAP OP_TOALTSTACK OP_TOALTSTACK OP_DROP OP_DUP OP_TOALTSTACK OP_IF OP_3
  OP_PICK OP_CHECKLOCKTIMEVERIFY OP_DROP OP_10 OP_PICK OP_CHECKSIGVERIFY
PADZ
OP_6 OP_PICK OP_EQUALVERIFY OP_ELSE OP_11 OP_PICK OP_CHECKSIGVERIFY OP_2DROP
  OP_DROP OP_ENDIF OP_2 <SEL_OUT_SINGLE> OP_TX <0x5120> OP_11 OP_PICK OP_CAT
  OP_EQUALVERIFY OP_10 OP_PICK OP_EQUALVERIFY OP_FROMALTSTACK OP_DROP
LEAF_JOIN
TAGGED(LEAF)
OP_FROMALTSTACK <TAG_NODE> OP_ROT OP_ROT <EMPTY_0> OP_0
LEVEL_MID x4
CHECK_PREV
FINAL
```

```
# JOIN_0 (MID tier, dense, D=4): 1284 bytes
# macros in CAPITALS are defined in MACROS_MID.txt; raw opcodes are wrapped and indented
```

```
PROLOGUE
<OPERATOR_KEY> OP_CHECKSIGVERIFY
NONNEG
OP_OVER OP_SIZE 32 OP_EQUALVERIFY OP_DROP <TAG_NODE> OP_ROT OP_ROT <EMPTY_0>
  OP_0 OP_2SWAP
LEVEL_MID x4
CHECK_OLD
FINAL
```

```
# JOIN_2 (MID tier, dense, D=4): 964 bytes
# macros in CAPITALS are defined in MACROS_MID.txt; raw opcodes are wrapped and indented
```

```

PROLOGUE
<OPERATOR_KEY> OP_CHECKSIGVERIFY
NONNEG
OP_OVER OP_SIZE 32 OP_EQUALVERIFY OP_DROP <TAG_NODE> OP_ROT OP_ROT <EMPTY_2>
    OP_0 OP_2SWAP
LEVEL_MID x2
CHECK_OLD
FINAL

# JOINK_2 (MID tier, dense, D=4): 2045 bytes
# macros in CAPITALS are defined in MACROS_MID.txt; raw opcodes are wrapped and indented

PROLOGUE
<OPERATOR_KEY> OP_CHECKSIGVERIFY
NONNEG
OP_OVER OP_SIZE 32 OP_EQUALVERIFY OP_DROP <TAG_NODE> OP_ROT OP_ROT <EMPTY_0>
    OP_0 OP_2SWAP
LEVEL_MID x4
CHECK_OLD
OP_SWAP OP_TOALTSTACK OP_TOALTSTACK OP_DROP
NONNEG
OP_OVER OP_SIZE 32 OP_EQUALVERIFY OP_DROP <TAG_NODE> OP_ROT OP_ROT <EMPTY_0>
    OP_0 OP_2SWAP
LEVEL_MID x4
CHECK_PREV
FINAL

# UPDATE (MID tier, dense, D=4): 1559 bytes
# macros in CAPITALS are defined in MACROS_MID.txt; raw opcodes are wrapped and indented

PROLOGUE
<OPERATOR_KEY> OP_CHECKSIGVERIFY 18 OP_PICK OP_CHECKSIGVERIFY
LEAF_JOIN
OP_TOALTSTACK
LEAF_JOIN
TAGGED(LEAF)
OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK OP_SWAP
TAGGED(LEAF)
OP_SWAP <TAG_NODE> OP_4 OP_ROLL OP_4 OP_ROLL OP_4 OP_ROLL OP_4 OP_ROLL
LEVEL_MID x4
CHECK_OLD
FINAL

# UPDATE_X (MID tier, exit-key consent, dense, D=4): 1563 bytes
# macros in CAPITALS are defined in MACROS_MID.txt; raw opcodes are wrapped and indented

PROLOGUE
EXIT_CONSENT_MID(21)
<OPERATOR_KEY> OP_CHECKSIGVERIFY 18 OP_PICK OP_CHECKSIGVERIFY
LEAF_JOIN
OP_TOALTSTACK
LEAF_JOIN
TAGGED(LEAF)
OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK OP_SWAP
TAGGED(LEAF)
OP_SWAP <TAG_NODE> OP_4 OP_ROLL OP_4 OP_ROLL OP_4 OP_ROLL OP_4 OP_ROLL
LEVEL_MID x4
CHECK_OLD
FINAL

# CLAIM (MID tier, dense, D=4): 2686 bytes
# macros in CAPITALS are defined in MACROS_MID.txt; raw opcodes are wrapped and indented

PROLOGUE
32 OP_PICK 32 OP_PICK 32 OP_PICK 32 OP_PICK 32 OP_PICK 32 OP_PICK OP_CAT
    OP_CAT OP_CAT OP_CAT OP_CAT 24 OP_PICK 24 OP_PICK 24 OP_PICK 24 OP_PICK 24

```

```

    OP_PICK 24 OP_PICK 24 OP_PICK OP_CAT OP_CAT OP_CAT OP_CAT OP_CAT OP_CAT
    OP_CAT <TREE_ID> OP_SWAP OP_CAT
TAGGED(VOUCHER)
32 OP_PICK OP_CHECKSIGFROMSTACK OP_VERIFY
CARVE_RULES
20 OP_PICK OP_7 OP_PICK OP_7 OP_ROLL OP_7 OP_ROLL
PADZ
OP_EQUALVERIFY OP_TOALTSTACK OP_TOALTSTACK OP_TOALTSTACK OP_TOALTSTACK
    OP_TOALTSTACK
LEAF_JOIN
OP_TOALTSTACK OP_8 OP_PICK OP_8 OP_PICK OP_8 OP_PICK OP_8 OP_PICK OP_8
    OP_PICK OP_CAT OP_CAT OP_CAT OP_CAT OP_TOALTSTACK
LEAF_JOIN
TAGGED(LEAF)
OP_FROMALTSTACK OP_DUP OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK
    OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK
    OP_FROMALTSTACK OP_FROMALTSTACK OP_7 OP_PICK OP_TOALTSTACK OP_6 OP_PICK
    OP_TOALTSTACK OP_TOALTSTACK OP_6 OP_ROLL OP_DROP OP_7 OP_ROLL OP_6 OP_ROLL
    OP_SUB OP_DUP OP_0 OP_GREATERTHANOREQUAL OP_VERIFY OP_DUP OP_3 OP_ROLL OP_3
    OP_ROLL
LE64
OP_4 OP_ROLL OP_1ADD OP_4 OP_ROLL OP_4 OP_ROLL
PADZ
OP_3 OP_ROLL OP_SWAP OP_CAT OP_CAT
TAGGED(LEAF)
OP_SWAP
DRAIN_RULE
<TAG_NODE> OP_4 OP_ROLL OP_4 OP_ROLL OP_4 OP_ROLL OP_4 OP_ROLL
LEVEL_MID    x4
CHECK_OLD
OP_FROMALTSTACK OP_FROMALTSTACK
TAGGED(LEAF)
OP_SWAP OP_2SWAP OP_SWAP OP_TOALTSTACK OP_TOALTSTACK <EMPTY_0> OP_0 OP_2SWAP
LEVEL_MID    x4
CHECK_PREV
FINAL

# CLAIM_H (MID tier, hash-locked voucher, dense, D=4): 2694 bytes
# macros in CAPITALS are defined in MACROS_MID.txt; raw opcodes are wrapped and indented

PROLOGUE
OP_SIZE 32 OP_EQUALVERIFY OP_SHA256 OP_TOALTSTACK 32 OP_PICK 32 OP_PICK 32
    OP_PICK 32 OP_PICK 32 OP_PICK 32 OP_PICK OP_CAT OP_CAT OP_CAT OP_CAT OP_CAT
    24 OP_PICK 24 OP_PICK 24 OP_PICK 24 OP_PICK 24 OP_PICK 24 OP_PICK 24
    OP_PICK OP_CAT OP_CAT OP_CAT OP_CAT OP_CAT OP_CAT OP_CAT <TREE_ID> OP_SWAP
    OP_CAT OP_FROMALTSTACK OP_CAT <TAG_VOUCHER_H> OP_SWAP OP_CAT OP_SHA256 32
    OP_PICK OP_CHECKSIGFROMSTACK OP_VERIFY
CARVE_RULES
20 OP_PICK OP_7 OP_PICK OP_7 OP_ROLL OP_7 OP_ROLL
PADZ
OP_EQUALVERIFY OP_TOALTSTACK OP_TOALTSTACK OP_TOALTSTACK OP_TOALTSTACK
    OP_TOALTSTACK
LEAF_JOIN
OP_TOALTSTACK OP_8 OP_PICK OP_8 OP_PICK OP_8 OP_PICK OP_8 OP_PICK OP_8
    OP_PICK OP_CAT OP_CAT OP_CAT OP_CAT OP_TOALTSTACK
LEAF_JOIN
TAGGED(LEAF)
OP_FROMALTSTACK OP_DUP OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK
    OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK
    OP_FROMALTSTACK OP_FROMALTSTACK OP_7 OP_PICK OP_TOALTSTACK OP_6 OP_PICK
    OP_TOALTSTACK OP_TOALTSTACK OP_6 OP_ROLL OP_DROP OP_7 OP_ROLL OP_6 OP_ROLL
    OP_SUB OP_DUP OP_0 OP_GREATERTHANOREQUAL OP_VERIFY OP_DUP OP_3 OP_ROLL OP_3
    OP_ROLL
LE64
OP_4 OP_ROLL OP_1ADD OP_4 OP_ROLL OP_4 OP_ROLL
PADZ
OP_3 OP_ROLL OP_SWAP OP_CAT OP_CAT
TAGGED(LEAF)

```

```

OP_SWAP
DRAIN_RULE
<TAG_NODE> OP_4 OP_ROLL OP_4 OP_ROLL OP_4 OP_ROLL OP_4 OP_ROLL
LEVEL_MID x4
CHECK_OLD
OP_FROMALTSTACK OP_FROMALTSTACK
TAGGED(LEAF)
OP_SWAP OP_2SWAP OP_SWAP OP_TOALTSTACK OP_TOALTSTACK <EMPTY_0> OP_0 OP_2SWAP
LEVEL_MID x4
CHECK_PREV
FINAL

# SPLIT (MID tier, owner-authorized carve, dense, D=4): 2531 bytes
# macros in CAPITALS are defined in MACROS_MID.txt; raw opcodes are wrapped and indented

PROLOGUE
31 OP_PICK OP_CHECKSIGVERIFY
CARVE_RULES
20 OP_PICK OP_7 OP_PICK OP_7 OP_ROLL OP_7 OP_ROLL
PADZ
OP_EQUALVERIFY OP_TOALTSTACK OP_TOALTSTACK OP_TOALTSTACK OP_TOALTSTACK
  OP_TOALTSTACK
LEAF_JOIN
OP_TOALTSTACK OP_8 OP_PICK OP_8 OP_PICK OP_8 OP_PICK OP_8 OP_PICK OP_8
  OP_PICK OP_CAT OP_CAT OP_CAT OP_CAT OP_TOALTSTACK
LEAF_JOIN
TAGGED(LEAF)
OP_FROMALTSTACK OP_DUP OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK
  OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK
  OP_FROMALTSTACK OP_FROMALTSTACK OP_7 OP_PICK OP_TOALTSTACK OP_6 OP_PICK
  OP_TOALTSTACK OP_TOALTSTACK OP_6 OP_ROLL OP_DROP OP_7 OP_ROLL OP_6 OP_ROLL
  OP_SUB OP_DUP OP_0 OP_GREATERTHANOREQUAL OP_VERIFY OP_DUP OP_3 OP_ROLL OP_3
  OP_ROLL
LE64
OP_4 OP_ROLL OP_1ADD OP_4 OP_ROLL OP_4 OP_ROLL
PADZ
OP_3 OP_ROLL OP_SWAP OP_CAT OP_CAT
TAGGED(LEAF)
OP_SWAP
DRAIN_RULE
<TAG_NODE> OP_4 OP_ROLL OP_4 OP_ROLL OP_4 OP_ROLL OP_4 OP_ROLL
LEVEL_MID x4
CHECK_OLD
OP_FROMALTSTACK OP_FROMALTSTACK
TAGGED(LEAF)
OP_SWAP OP_2SWAP OP_SWAP OP_TOALTSTACK OP_TOALTSTACK <EMPTY_0> OP_0 OP_2SWAP
LEVEL_MID x4
CHECK_PREV
FINAL

# EVICT_SUBTREE_2 (MID tier, cohort sweep of a 4-position subtree, dense, D=4): 2249 bytes
# macros in CAPITALS are defined in MACROS_MID.txt; raw opcodes are wrapped and indented

PROLOGUE
<OPERATOR_KEY> OP_CHECKSIGVERIFY OP_1
EVICT_LEAF_STEP x4
OP_DROP <TAG_NODE> OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK
  OP_FROMALTSTACK
EVICT_HASH_STEP(4)
OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK
EVICT_HASH_STEP(6)
EVICT_HASH_ROOT(4)
<EMPTY_2> OP_0
LEVEL_MID x2
CHECK_OLD
FINAL

```

D.8 MIN Tier Macros

```
# Macro definitions, MIN tier. Each expansion is itself collapsed
# against the other macros; a macro never contains itself.

## TAGGED(LEAF)
<TAG_LEAF> OP_SWAP OP_CAT OP_SHA256

## TAGGED(NODE)
<TAG_NODE> OP_SWAP OP_CAT OP_SHA256

## TAGGED(STATE)
<TAG_STATE> OP_SWAP OP_CAT OP_SHA256

## TAGGED(TAPTWEAK)
<TAG_TAPTWEAK> OP_SWAP OP_CAT OP_SHA256

## TAGGED(TAPSIGHASH)
<TAG_TAPSIGHASH> OP_SWAP OP_CAT OP_SHA256

## TAGGED(CHALLENGE)
<TAG_CHALLENGE> OP_SWAP OP_CAT OP_SHA256

## NONNEG    ([x] -> [x]: x == |x|, a minimal non-negative CScriptNum (rejects a negative, a
               negative zero and a non-minimal encoding))
OP_DUP OP_DUP OP_ABS OP_EQUALVERIFY

## PADZ      ([x z w] -> [le32(x)]: z||w == 00000000 and size(x)+size(z) == 4)
OP_OVER OP_SWAP OP_CAT <ZERO4> OP_EQUALVERIFY OP_SIZE OP_ROT OP_SIZE OP_ROT
OP_ADD OP_4 OP_NUMEQUALVERIFY OP_SWAP OP_CAT

## LE64      ([x z w] -> [le64(x)])
PADZ
<ZERO4> OP_CAT

## TREE_SPK_MIN ([r m] -> [TREE_SPK(r, m)] with OP_TWEAKADD)
OP_SWAP
TAGGED(STATE)
<NUMS_X> OP_TWEAKADD OP_TUCK OP_SWAP OP_CAT
TAGGED(TAPTWEAK)
OP_SWAP OP_TWEAKADD <0x5120> OP_SWAP OP_CAT

## LEVEL_MID  (dense level with PADZ gadgets)
OP_13 OP_ROLL OP_13 OP_ROLL OP_13 OP_ROLL OP_13 OP_ROLL OP_13 OP_ROLL OP_13 OP_ROLL OP_13
OP_ROLL OP_13 OP_ROLL OP_13 OP_ROLL OP_13 OP_ROLL OP_2 OP_PICK
NONNEG
OP_TOALTSTACK
PADZ
<00000000_01> OP_CAT OP_CAT OP_FROMALTSTACK OP_7 OP_PICK OP_OVER OP_ADD
OP_TOALTSTACK OP_7 OP_ROLL OP_5 OP_ROLL OP_5 OP_ROLL
PADZ
<00000000_01> OP_CAT OP_6 OP_ROLL OP_SWAP OP_CAT OP_2 OP_PICK OP_SWAP OP_4
OP_PICK OP_NOTIF OP_SWAP OP_ENDIF OP_CAT OP_8 OP_PICK OP_SWAP OP_CAT
OP_SHA256 OP_TOALTSTACK OP_5 OP_PICK OP_OVER OP_ADD OP_TOALTSTACK OP_DROP
OP_4 OP_ROLL OP_4 OP_ROLL OP_4 OP_ROLL
PADZ
<00000000_01> OP_CAT OP_3 OP_ROLL OP_SWAP OP_CAT OP_ROT OP_NOTIF OP_SWAP
OP_ENDIF OP_CAT OP_OVER OP_SWAP OP_CAT OP_SHA256 OP_FROMALTSTACK
OP_FROMALTSTACK OP_FROMALTSTACK

## LEAF_JOIN  ([amt8 ek ok evk ea nonce seq amt_num za wa] -> [leaf] | amt_num)
OP_2 OP_PICK
NONNEG
OP_TOALTSTACK
LE64
OP_7 OP_PICK OP_EQUALVERIFY OP_DUP OP_SIZE OP_4 OP_EQUALVERIFY OP_DROP OP_1
OP_PICK OP_SIZE OP_4 OP_EQUALVERIFY OP_DROP OP_2 OP_PICK OP_SIZE OP_4
OP_EQUALVERIFY OP_DROP OP_3 OP_PICK OP_SIZE 32 OP_EQUALVERIFY OP_DROP OP_4
```

```

OP_PICK OP_SIZE 32 OP_EQUALVERIFY OP_DROP OP_5 OP_PICK OP_SIZE 32
OP_EQUALVERIFY OP_DROP OP_6 OP_PICK OP_SIZE OP_8 OP_EQUALVERIFY OP_DROP
OP_CAT OP_CAT OP_CAT OP_CAT OP_CAT OP_CAT

## CARVE_RULES ([src(10) claim(10) extras(7) sea zse wse cea zce wce] -> [src(10)
    claim(10) extras(7)]: claim.evk == src.evk and claim.evict_after <= src.evict_after,
    each evict_after a minimal number checked against its field by PADZ and the claim's
    NONNEG-checked before the compare (7.5 rule 5, 10.6), and claim.amt_num >= DUST_MIN
    (rule 6))
19 OP_PICK 30 OP_PICK OP_EQUALVERIFY OP_2 OP_PICK OP_TOALTSTACK
PADZ
OP_16 OP_PICK OP_EQUALVERIFY OP_2 OP_PICK OP_TOALTSTACK
PADZ
23 OP_PICK OP_EQUALVERIFY OP_FROMALTSTACK OP_FROMALTSTACK
NONNEG
OP_GREATERTHANOREQUAL OP_VERIFY 330 OP_10 OP_PICK OP_LESSTHANOREQUAL
    OP_VERIFY

## DRAIN_RULE ([.. nh ns] -> [.. nh' ns]: a zero residual makes the new node E_0, any
    other is at least DUST_MIN (7.5 rules 2 and 6))
OP_DUP OP_NOT OP_IF OP_SWAP OP_DROP <EMPTY_0> OP_SWAP OP_ELSE 330 OP_OVER
    OP_LESSTHANOREQUAL OP_VERIFY OP_ENDIF

```

D.9 MIN Tier, Dense Profile ($D = 4$)

```

# EXIT_1 (MIN tier, dense, D=4): 1720 bytes
# macros in CAPITALS are defined in MACROS_MIN.txt; raw opcodes are wrapped and indented

OP_2DUP
TREE_SPK_MIN
OP_TOALTSTACK OP_TOALTSTACK OP_TOALTSTACK OP_IF OP_3 OP_PICK
    OP_CHECKLOCKTIMEVERIFY OP_DROP OP_10 OP_PICK OP_CHECKSIGVERIFY
PADZ
OP_6 OP_PICK OP_EQUALVERIFY OP_ELSE OP_11 OP_PICK OP_CHECKSIGVERIFY OP_2DROP
    OP_DROP OP_ENDIF OP_9 OP_PICK 2117922 OP_CAT OP_9 OP_PICK OP_CAT
    OP_TOALTSTACK
LEAF_JOIN
TAGGED(LEAF)
OP_FROMALTSTACK <TAG_NODE> OP_ROT OP_ROT <EMPTY_0> OP_0
LEVEL_MID x4
OP_2SWAP OP_6 OP_ROLL OP_6 OP_ROLL
LE64
OP_FROMALTSTACK OP_FROMALTSTACK OP_3 OP_ROLL OP_EQUALVERIFY OP_4 OP_ROLL
    OP_DROP OP_ROT OP_5 OP_ROLL OP_5 OP_ROLL
LE64
OP_3 OP_ROLL OP_FROMALTSTACK
TREE_SPK_MIN
34 OP_SWAP OP_CAT OP_CAT OP_SWAP OP_CAT OP_2 OP_ROLL OP_CAT OP_SHA256 OP_2
    OP_ROLL OP_ROT OP_SWAP OP_CAT OP_SHA256 OP_FROMALTSTACK 34 OP_SWAP OP_CAT
    OP_3 OP_ROLL OP_CAT OP_SHA256 OP_3 OP_ROLL OP_4 OP_ROLL 0 OP_SWAP OP_CAT
    OP_SWAP OP_CAT OP_4 OP_ROLL OP_CAT OP_ROT OP_CAT OP_SWAP OP_CAT OP_3
    OP_ROLL OP_CAT OP_SWAP OP_CAT <0200000000> OP_CAT OP_SWAP OP_CAT
    <00ffffff> OP_CAT
TAGGED(TAPSIGHASH)
<G_X||G_X> OP_SWAP OP_CAT
TAGGED(CHALLENGE)
OP_OVER 0 OP_CAT OP_EQUALVERIFY <G_X> OP_SWAP OP_CAT OP_1 OP_CAT <G_X>
    OP_CHECKSIG

# SPLIT (MIN tier, owner-authorized carve, dense, D=4): 2815 bytes
# macros in CAPITALS are defined in MACROS_MIN.txt; raw opcodes are wrapped and indented

OP_2DUP
TREE_SPK_MIN
OP_TOALTSTACK OP_DUP OP_TOALTSTACK OP_SWAP OP_TOALTSTACK OP_DROP 31 OP_PICK
    OP_CHECKSIGVERIFY
CARVE_RULES

```

```

20 OP_PICK OP_7 OP_PICK OP_7 OP_ROLL OP_7 OP_ROLL
PADZ
OP_EQUALVERIFY OP_TOALTSTACK OP_TOALTSTACK OP_TOALTSTACK OP_TOALTSTACK
  OP_TOALTSTACK
LEAF_JOIN
OP_TOALTSTACK OP_8 OP_PICK OP_8 OP_PICK OP_8 OP_PICK OP_8 OP_PICK OP_8
  OP_PICK OP_CAT OP_CAT OP_CAT OP_CAT OP_TOALTSTACK
LEAF_JOIN
TAGGED(LEAF)
OP_FROMALTSTACK OP_DUP OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK
  OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK
  OP_FROMALTSTACK OP_FROMALTSTACK OP_7 OP_PICK OP_TOALTSTACK OP_6 OP_PICK
  OP_TOALTSTACK OP_TOALTSTACK OP_6 OP_ROLL OP_DROP OP_7 OP_ROLL OP_6 OP_ROLL
  OP_SUB OP_DUP OP_0 OP_GREATERTHANOREQUAL OP_VERIFY OP_DUP OP_3 OP_ROLL OP_3
  OP_ROLL
LE64
OP_4 OP_ROLL OP_1ADD OP_4 OP_ROLL OP_4 OP_ROLL
PADZ
OP_3 OP_ROLL OP_SWAP OP_CAT OP_CAT
TAGGED(LEAF)
OP_SWAP
DRAIN_RULE
<TAG_NODE> OP_4 OP_ROLL OP_4 OP_ROLL OP_4 OP_ROLL OP_4 OP_ROLL
LEVEL_MID x4
OP_2SWAP OP_6 OP_ROLL OP_6 OP_ROLL
LE64
OP_SWAP OP_FROMALTSTACK OP_EQUALVERIFY OP_FROMALTSTACK OP_FROMALTSTACK OP_ROT
  OP_TOALTSTACK
TAGGED(LEAF)
OP_SWAP OP_2SWAP OP_SWAP OP_TOALTSTACK OP_TOALTSTACK <EMPTY_0> OP_0 OP_2SWAP
LEVEL_MID x4
OP_2SWAP OP_FROMALTSTACK OP_NUMEQUALVERIFY OP_FROMALTSTACK OP_EQUALVERIFY
  OP_ROT OP_DROP OP_3 OP_ROLL OP_3 OP_ROLL
LE64
OP_SWAP OP_FROMALTSTACK OP_FROMALTSTACK OP_ROT OP_SWAP
TREE_SPK_MIN
34 OP_SWAP OP_CAT OP_ROT OP_SWAP OP_CAT OP_2 OP_ROLL OP_CAT OP_SHA256 OP_2
  OP_ROLL OP_ROT OP_SWAP OP_CAT OP_SHA256 OP_FROMALTSTACK 34 OP_SWAP OP_CAT
  OP_3 OP_ROLL OP_CAT OP_SHA256 OP_3 OP_ROLL OP_4 OP_ROLL 0 OP_SWAP OP_CAT
  OP_SWAP OP_CAT OP_4 OP_ROLL OP_CAT OP_ROT OP_CAT OP_SWAP OP_CAT OP_3
  OP_ROLL OP_CAT OP_SWAP OP_CAT <0200000000> OP_CAT OP_SWAP OP_CAT
  <00ffffff> OP_CAT
TAGGED(TAPSIGHASH)
<G_X||G_X> OP_SWAP OP_CAT
TAGGED(CHALLENGE)
OP_OVER 0 OP_CAT OP_EQUALVERIFY <G_X> OP_SWAP OP_CAT OP_1 OP_CAT <G_X>
  OP_CHECKSIG

```

D.10 CCV Tier Macros

```

# Macro definitions, CCV tier. Each expansion is itself collapsed
# against the other macros; a macro never contains itself.

```

```

## TAGGED(LEAF)
<TAG_LEAF> OP_SWAP OP_CAT OP_SHA256

## TAGGED(NODE)
<TAG_NODE> OP_SWAP OP_CAT OP_SHA256

## TAGGED(POS)
<TAG_POS> OP_SWAP OP_CAT OP_SHA256

## TAGGED(TAPTWEAK)
<TAG_TAPTWEAK> OP_SWAP OP_CAT OP_SHA256

## TAGGED(VOUCHER)
<TAG_VOUCHER> OP_SWAP OP_CAT OP_SHA256

```

```

## TAGGED(VOUCHER_H)
<TAG_VOUCHER_H> OP_SWAP OP_CAT OP_SHA256

## TAGGED(EXIT)
<TAG_EXIT> OP_SWAP OP_CAT OP_SHA256

## LEAF_BODY ([leaf] -> [body]: the 108 bytes after the amount field (5.2), what vouchers
and exit authorisations bind (7.5, 7.2 rule 7))
108 OP_RIGHT

## TOWER_AUTH ([leaf sig] -> [leaf]: sig is a BIP-340 signature by leaf.owner_key over
H_exit(tree_id || body(leaf)), the leaf without its amount (LEAF_BODY), checked by
OP_CHECKSIGFROMSTACK (EXIT_K~t, 7.2 rule 7))
OP_OVER
LEAF_BODY
<TREE_ID> OP_SWAP OP_CAT
TAGGED(EXIT)
OP_2 OP_PICK 40 32 OP_SUBSTR OP_CHECKSIGFROMSTACK OP_VERIFY

## CCV_CHECK_INPUT (BIP-443 CHECK_INPUT: [data] -> [], binds the old root to the current
input under its own taptree)
129 OP_0 129 129 OP_CHECKCONTRACTVERIFY

## CCV_PAYOUT(1)
OP_DUP OP_TOALTSTACK OP_TOALTSTACK OP_0 OP_1 OP_FROMALTSTACK OP_8 32
OP_SUBSTR OP_0 OP_2 OP_CHECKCONTRACTVERIFY OP_FROMALTSTACK

## CCV_PAYOUT(2)
OP_DUP OP_TOALTSTACK OP_TOALTSTACK OP_0 OP_2 OP_FROMALTSTACK OP_8 32
OP_SUBSTR OP_0 OP_2 OP_CHECKCONTRACTVERIFY OP_FROMALTSTACK

## LEAF_ACC
OP_DUP OP_8 OP_LEFT OP_SWAP
TAGGED(LEAF)
OP_SWAP

## LEAF_POS
OP_DUP 40 32 OP_SUBSTR OP_OVER 108 OP_4 OP_SUBSTR OP_CAT
TAGGED(POS)

## LEVEL
OP_6 OP_ROLL OP_6 OP_ROLL OP_SIZE 41 OP_EQUALVERIFY OP_DUP 32 OP_8 OP_SUBSTR
OP_DUP OP_TOALTSTACK OP_3 OP_ROLL OP_TUCK OP_ADD OP_TOALTSTACK <ENCO>
OP_ADD OP_3 OP_ROLL OP_SWAP OP_CAT OP_OVER OP_SWAP OP_3 OP_PICK OP_NOTIF
OP_SWAP OP_ENDIF OP_CAT OP_5 OP_PICK OP_SWAP OP_CAT OP_SHA256 OP_TOALTSTACK
OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK OP_5 OP_ROLL OP_TUCK OP_ADD
OP_TOALTSTACK <ENCO> OP_ADD OP_5 OP_ROLL OP_SWAP OP_CAT OP_3 OP_ROLL
OP_SWAP OP_4 OP_ROLL OP_NOTIF OP_SWAP OP_ENDIF OP_CAT OP_3 OP_PICK OP_SWAP
OP_CAT OP_SHA256 OP_FROMALTSTACK OP_2SWAP

## SPARSE_SETUP
OP_TOALTSTACK <EMPTY_0> OP_TOALTSTACK

## SPARSE_TEARDOWN
OP_FROMALTSTACK OP_DROP OP_FROMALTSTACK OP_DROP

## CHECK_PREV
OP_2SWAP OP_FROMALTSTACK OP_NUMEQUALVERIFY OP_FROMALTSTACK OP_EQUALVERIFY

## CHECK_PREV_S
OP_2SWAP
SPARSE_TEARDOWN
OP_FROMALTSTACK OP_NUMEQUALVERIFY OP_FROMALTSTACK OP_EQUALVERIFY

## CHECK_OLD (the input-amount equality plus one CCV_CHECK_INPUT on the folded old root)
OP_2SWAP <SEL_IN_AMOUNT> OP_TX OP_NUMEQUALVERIFY OP_ROT OP_ROT OP_TOALTSTACK
OP_TOALTSTACK
CCV_CHECK_INPUT

```

```

OP_FROMALTSTACK OP_FROMALTSTACK

## CHECK_OLD_S
OP_2SWAP <SEL_IN_AMOUNT> OP_TX OP_NUMEQUALVERIFY
SPARSE_TEARDOWN
OP_ROT OP_ROT OP_TOALTSTACK OP_TOALTSTACK
CCV_CHECK_INPUT
OP_FROMALTSTACK OP_FROMALTSTACK

## PROLOGUE_CCV (empty: CCV's -1 index binds the current input without OP_TX)

## CARVE_RULES ([src claim] -> [src claim]: both 116 bytes, claim.evictor_key ==
src.evictor_key, claim.evict_after <= src.evict_after (7.5 rule 5), claim.amount >=
DUST_MIN (7.5 rule 6))
OP_SIZE 116 OP_EQUALVERIFY OP_SWAP OP_SIZE 116 OP_EQUALVERIFY OP_SWAP OP_2DUP
72 32 OP_SUBSTR OP_SWAP 72 32 OP_SUBSTR OP_EQUALVERIFY OP_2DUP 104 OP_4
OP_SUBSTR OP_SWAP 104 OP_4 OP_SUBSTR OP_LESSTHANOREQUAL OP_VERIFY 330
OP_OVER OP_8 OP_LEFT OP_LESSTHANOREQUAL OP_VERIFY

## DRAIN_RULE ([.. nh ns] -> [.. nh' ns]: a zero residual makes the new node E_0, any
other is at least DUST_MIN (7.5 rules 2 and 6))
OP_DUP OP_NOT OP_IF OP_SWAP OP_DROP <EMPTY_0> OP_SWAP OP_ELSE 330 OP_OVER
OP_LESSTHANOREQUAL OP_VERIFY OP_ENDIF

## LEVEL_S(k) [shown for k = 0; <EMPTY_k> and the bit index vary with k]
OP_5 OP_ROLL OP_FROMALTSTACK OP_SWAP OP_SIZE OP_ONOTEQUAL OP_NOTIF OP_DROP
OP_DUP <ENCO> OP_CAT OP_ENDIF OP_SWAP <ENCO> OP_CAT OP_DUP OP_CAT OP_6
OP_PICK OP_SWAP OP_CAT OP_SHA256 OP_FROMALTSTACK OP_DUP OP_TOALTSTACK
OP_SWAP OP_TOALTSTACK OP_0 OP_DOWNSHIFT OP_1 OP_AND OP_ONOTEQUAL OP_SWAP
OP_SIZE 41 OP_EQUALVERIFY OP_DUP 32 OP_8 OP_SUBSTR OP_DUP OP_TOALTSTACK
OP_3 OP_ROLL OP_TUCK OP_ADD OP_TOALTSTACK <ENCO> OP_ADD OP_3 OP_ROLL
OP_SWAP OP_CAT OP_OVER OP_SWAP OP_3 OP_PICK OP_NOTIF OP_SWAP OP_ENDIF
OP_CAT OP_5 OP_PICK OP_SWAP OP_CAT OP_SHA256 OP_TOALTSTACK OP_FROMALTSTACK
OP_FROMALTSTACK OP_FROMALTSTACK OP_5 OP_ROLL OP_TUCK OP_ADD OP_TOALTSTACK
<ENCO> OP_ADD OP_5 OP_ROLL OP_SWAP OP_CAT OP_3 OP_ROLL OP_SWAP OP_4 OP_ROLL
OP_NOTIF OP_SWAP OP_ENDIF OP_CAT OP_3 OP_PICK OP_SWAP OP_CAT OP_SHA256
OP_FROMALTSTACK OP_2SWAP

## FINAL [shown for D = 4; <EMPTY_D> is the empty root of the tree's depth] (output 0 is
the CCV assign-mode tree output with amount == ns, unless nh == E_D)
OP_ROT OP_DROP OP_OVER <EMPTY_4> OP_EQUAL OP_IF OP_2DROP OP_1 OP_ELSE
OP_TOALTSTACK OP_0 OP_0 129 OP_0 OP_CHECKCONTRACTVERIFY OP_FROMALTSTACK
OP_0 <SEL_OUT_SINGLE> OP_TX OP_DROP OP_NUMEQUALVERIFY OP_1 OP_ENDIF

## LEVEL_MIN
OP_7 OP_ROLL OP_7 OP_ROLL OP_7 OP_ROLL OP_SIZE OP_8 OP_EQUALVERIFY OP_OVER
OP_SIZE 32 OP_EQUALVERIFY OP_DROP OP_DUP OP_TOALTSTACK OP_SWAP OP_OVER
<ENCO> OP_ADD OP_CAT OP_SWAP OP_3 OP_ROLL OP_TUCK OP_ADD OP_TOALTSTACK
<ENCO> OP_ADD OP_3 OP_ROLL OP_SWAP OP_CAT OP_OVER OP_SWAP OP_3 OP_PICK
OP_NOTIF OP_SWAP OP_ENDIF OP_CAT OP_5 OP_PICK OP_SWAP OP_CAT OP_SHA256
OP_TOALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK OP_5 OP_ROLL
OP_TUCK OP_ADD OP_TOALTSTACK <ENCO> OP_ADD OP_5 OP_ROLL OP_SWAP OP_CAT OP_3
OP_ROLL OP_SWAP OP_4 OP_ROLL OP_NOTIF OP_SWAP OP_ENDIF OP_CAT OP_3 OP_PICK
OP_SWAP OP_CAT OP_SHA256 OP_FROMALTSTACK OP_2SWAP

## LEAF_STEP
OP_SIZE OP_ONOTEQUAL OP_IF OP_SIZE 116 OP_EQUALVERIFY OP_DUP 72 32 OP_SUBSTR
<OPERATOR_KEY> OP_EQUALVERIFY OP_DUP 104 OP_4 OP_SUBSTR
OP_CHECKLOCKTIMEVERIFY OP_DROP OP_FROMALTSTACK OP_DUP OP_1ADD OP_TOALTSTACK
OP_DUP OP_TOALTSTACK OP_0 OP_SWAP OP_2 OP_PICK OP_8 32 OP_SUBSTR OP_0 OP_2
OP_CHECKCONTRACTVERIFY OP_FROMALTSTACK <SEL_OUT_SINGLE> OP_TX OP_DROP
OP_OVER OP_8 OP_LEFT OP_NUMEQUALVERIFY
LEAF_ACC
OP_ELSE OP_DROP <EMPTY_0> OP_0 OP_ENDIF

## HASH_STEP(t) [shown for t = 3; the OP_PICK depth is t-3]
OP_2 OP_PICK OP_OVER OP_ADD OP_TOALTSTACK <ENCO> OP_ADD OP_CAT OP_SWAP <ENCO>
OP_ADD OP_ROT OP_SWAP OP_CAT OP_SWAP OP_CAT OP_0 OP_PICK OP_SWAP OP_CAT

```

```

OP_SHA256 OP_FROMALTSTACK

## REWRITE_PAY_CCV    (cohort rewrite: output idx pays exactly old.amount to old.exit_key by
    a CHECK_OUTPUT|DEDUCT call, idx advances (7.9 rule 5))
OP_FROMALTSTACK OP_DUP OP_1ADD OP_TOALTSTACK OP_DUP OP_TOALTSTACK OP_0
OP_SWAP OP_3 OP_PICK OP_8 32 OP_SUBSTR OP_0 OP_2 OP_CHECKCONTRACTVERIFY
OP_FROMALTSTACK <SEL_OUT_SINGLE> OP_TX OP_DROP OP_2 OP_PICK OP_8 OP_LEFT
OP_NUMEQUALVERIFY

## REWRITE_STEP_CCV(d) [shown for d = 0; the OP_ROLL depths are d+3, d+4 and d+5, the
    picks d+3 to d+5] (cohort rewrite: FULL's REWRITE_STEP with REWRITE_PAY_CCV as the
    exit payout)
OP_3 OP_ROLL OP_4 OP_ROLL OP_SIZE OP_1 OP_EQUAL OP_IF OP_DROP OP_DUP OP_ENDIF
    OP_SIZE OP_8 OP_EQUAL OP_IF OP_OVER
LEAF_BODY
OP_CAT OP_ENDIF OP_2DUP OP_EQUAL OP_NOTIF OP_OVER OP_SIZE OP_NIP OP_ONOTEQUAL
    OP_IF OP_DUP OP_SIZE OP_NIP OP_ONOTEQUAL OP_IF OP_2DUP OP_8 108 OP_SUBSTR
    OP_SWAP OP_8 108 OP_SUBSTR OP_EQUAL OP_IF OP_2DUP OP_SWAP OP_8 OP_LEFT
    OP_SWAP OP_8 OP_LEFT OP_LESSTHANOREQUAL OP_OVER OP_4 OP_RIGHT <SEQ_FINAL>
    OP_EQUAL OP_NOT OP_BOOLAND OP_ELSE OP_0 OP_ENDIF OP_NOTIF
OWNER_AUTH(0)
OP_ENDIF OP_ELSE
OWNER_AUTH(0)
REWRITE_PAY_CCV
OP_ENDIF OP_ENDIF OP_ENDIF OP_SWAP
ELEM_NODE(0)
OP_ROT
ELEM_NODE(1)

## REWRITE_STEP_CCV_X(d) [shown for d = 0; the OP_ROLL depths are d+3, d+4 and d+5, the
    picks d+3 to d+5] (REWRITE_STEP_CCV with OWNER_AUTH_X)
OP_3 OP_ROLL OP_4 OP_ROLL OP_SIZE OP_1 OP_EQUAL OP_IF OP_DROP OP_DUP OP_ENDIF
    OP_SIZE OP_8 OP_EQUAL OP_IF OP_OVER
LEAF_BODY
OP_CAT OP_ENDIF OP_2DUP OP_EQUAL OP_NOTIF OP_OVER OP_SIZE OP_NIP OP_ONOTEQUAL
    OP_IF OP_DUP OP_SIZE OP_NIP OP_ONOTEQUAL OP_IF OP_2DUP OP_8 108 OP_SUBSTR
    OP_SWAP OP_8 108 OP_SUBSTR OP_EQUAL OP_IF OP_2DUP OP_SWAP OP_8 OP_LEFT
    OP_SWAP OP_8 OP_LEFT OP_LESSTHANOREQUAL OP_OVER OP_4 OP_RIGHT <SEQ_FINAL>
    OP_EQUAL OP_NOT OP_BOOLAND OP_ELSE OP_0 OP_ENDIF OP_NOTIF
OWNER_AUTH_X(0)
OP_ENDIF OP_ELSE
OWNER_AUTH_X(0)
REWRITE_PAY_CCV
OP_ENDIF OP_ENDIF OP_ENDIF OP_SWAP
ELEM_NODE(0)
OP_ROT
ELEM_NODE(1)

## HASH_STEP2(d) [shown for d = 8; the HASH_STEP pick depths are d-3 and d-5] (cohort
    rewrite: two HASH_STEPS, the old pair then the new pair)
OP_2SWAP OP_7 OP_ROLL OP_7 OP_ROLL OP_2SWAP
HASH_STEP(8)
OP_2SWAP OP_5 OP_ROLL OP_5 OP_ROLL OP_2SWAP
HASH_STEP(6)

## ELEM_NODE(d) [shown for d = 0; picks L at d+4 and Z at d+3 for the old element, one
    deeper for the new] (cohort rewrite: a 116-byte leaf -> (H_leaf, amount), the empty
    element -> E_0)
OP_SIZE OP_ONOTEQUAL OP_IF OP_SIZE 116 OP_EQUALVERIFY OP_DUP OP_8 OP_LEFT
    OP_SWAP OP_4 OP_PICK OP_SWAP OP_CAT OP_SHA256 OP_SWAP OP_ELSE OP_DROP OP_3
    OP_PICK OP_0 OP_ENDIF

## OWNER_AUTH(d) [shown for d = 0; the OP_ROLL depth is d+5] (cohort rewrite: the next
    witness element is a signature by old.owner_key)
OP_5 OP_ROLL OP_2 OP_PICK 40 32 OP_SUBSTR OP_CHECKSIGVERIFY

## OWNER_AUTH_X(d) [shown for d = 0; the OP_ROLL depth is d+5] (OWNER_AUTH plus a
    signature by old.exit_key (REWRITE_h~x))

```

```

OWNER_AUTH(0)
OP_5 OP_ROLL OP_2 OP_PICK OP_8 32 OP_SUBSTR OP_CHECKSIGVERIFY

## EXIT_CONSENT(k) [shown for k = 3; k = 4 under a preimage (CLAIM^hx) or for UPDATE^x]
OP_3 OP_PICK OP_8 32 OP_SUBSTR OP_CHECKSIGVERIFY

```

D.11 CCV Tier, 0xC2 Numerics, Dense Profile ($D = 4$)

```

# EXIT_1 (CCV tier, dense, D=4): 660 bytes
# macros in CAPITALS are defined in MACROS_CCV.txt; raw opcodes are wrapped and indented

OP_DUP OP_TOALTSTACK OP_IF OP_OVER 104 OP_4 OP_SUBSTR OP_CHECKLOCKTIMEVERIFY
  OP_DROP OP_OVER 72 32 OP_SUBSTR OP_ELSE OP_OVER 40 32 OP_SUBSTR OP_ENDIF
  OP_CHECKSIGVERIFY OP_SIZE 116 OP_EQUALVERIFY
CCV_PAYOUT(1)
OP_1 <SEL_OUT_SINGLE> OP_TX OP_DROP OP_OVER OP_8 OP_LEFT OP_FROMALTSTACK
  OP_DROP OP_NUMEQUALVERIFY
LEAF_ACC
<TAG_NODE> OP_ROT OP_ROT <EMPTY_0> OP_0
LEVEL x4
CHECK_OLD
FINAL

# EXIT_2 (CCV tier, dense, D=4): 1250 bytes
# macros in CAPITALS are defined in MACROS_CCV.txt; raw opcodes are wrapped and indented

OP_DUP OP_TOALTSTACK OP_IF OP_OVER 104 OP_4 OP_SUBSTR OP_CHECKLOCKTIMEVERIFY
  OP_DROP OP_OVER 72 32 OP_SUBSTR OP_ELSE OP_OVER 40 32 OP_SUBSTR OP_ENDIF
  OP_CHECKSIGVERIFY OP_SIZE 116 OP_EQUALVERIFY
CCV_PAYOUT(1)
OP_1 <SEL_OUT_SINGLE> OP_TX OP_DROP OP_OVER OP_8 OP_LEFT OP_FROMALTSTACK
  OP_DROP OP_NUMEQUALVERIFY
LEAF_ACC
<TAG_NODE> OP_ROT OP_ROT <EMPTY_0> OP_0
LEVEL x4
CHECK_OLD
OP_SWAP OP_TOALTSTACK OP_TOALTSTACK OP_DROP OP_DUP OP_TOALTSTACK OP_IF
  OP_OVER 104 OP_4 OP_SUBSTR OP_CHECKLOCKTIMEVERIFY OP_DROP OP_OVER 72 32
  OP_SUBSTR OP_ELSE OP_OVER 40 32 OP_SUBSTR OP_ENDIF OP_CHECKSIGVERIFY
  OP_SIZE 116 OP_EQUALVERIFY
CCV_PAYOUT(2)
OP_2 <SEL_OUT_SINGLE> OP_TX OP_DROP OP_OVER OP_8 OP_LEFT OP_FROMALTSTACK
  OP_DROP OP_NUMEQUALVERIFY
LEAF_ACC
<TAG_NODE> OP_ROT OP_ROT <EMPTY_0> OP_0
LEVEL x4
CHECK_PREV
FINAL

# JOIN_0 (CCV tier, dense, D=4): 576 bytes
# macros in CAPITALS are defined in MACROS_CCV.txt; raw opcodes are wrapped and indented

<OPERATOR_KEY> OP_CHECKSIGVERIFY OP_SIZE OP_8 OP_EQUALVERIFY OP_OVER OP_SIZE
  32 OP_EQUALVERIFY OP_DROP <TAG_NODE> OP_ROT OP_ROT <EMPTY_0> OP_0 OP_2SWAP
LEVEL x4
CHECK_OLD
FINAL

# JOIN_2 (CCV tier, dense, D=4): 400 bytes
# macros in CAPITALS are defined in MACROS_CCV.txt; raw opcodes are wrapped and indented

<OPERATOR_KEY> OP_CHECKSIGVERIFY OP_SIZE OP_8 OP_EQUALVERIFY OP_OVER OP_SIZE
  32 OP_EQUALVERIFY OP_DROP <TAG_NODE> OP_ROT OP_ROT <EMPTY_2> OP_0 OP_2SWAP
LEVEL x2
CHECK_OLD
FINAL

```

```

# JOINK_2 (CCV tier, dense, D=4): 1048 bytes
# macros in CAPITALS are defined in MACROS_CCV.txt; raw opcodes are wrapped and indented

<OPERATOR_KEY> OP_CHECKSIGVERIFY OP_SIZE OP_8 OP_EQUALVERIFY OP_OVER OP_SIZE
  32 OP_EQUALVERIFY OP_DROP <TAG_NODE> OP_ROT OP_ROT <EMPTY_0> OP_0 OP_2SWAP
LEVEL   x4
CHECK_OLD
OP_SWAP OP_TOALTSTACK OP_TOALTSTACK OP_DROP OP_SIZE OP_8 OP_EQUALVERIFY
  OP_OVER OP_SIZE 32 OP_EQUALVERIFY OP_DROP <TAG_NODE> OP_ROT OP_ROT
  <EMPTY_0> OP_0 OP_2SWAP
LEVEL   x4
CHECK_PREV
FINAL

# UPDATE (CCV tier, dense, D=4): 704 bytes
# macros in CAPITALS are defined in MACROS_CCV.txt; raw opcodes are wrapped and indented

<OPERATOR_KEY> OP_CHECKSIGVERIFY OP_2 OP_PICK 40 32 OP_SUBSTR
  OP_CHECKSIGVERIFY OP_SIZE 116 OP_EQUALVERIFY OP_SWAP OP_SIZE 116
  OP_EQUALVERIFY <TAG_NODE> OP_ROT OP_ROT
LEAF_ACC
OP_ROT
LEAF_ACC
LEVEL   x4
CHECK_OLD
OP_ROT OP_DROP OP_OVER <EMPTY_4> OP_EQUAL OP_IF OP_2DROP OP_DROP OP_1 OP_ELSE
  OP_TOALTSTACK OP_SWAP OP_IF OP_2 OP_ELSE OP_0 OP_ENDIF OP_0 OP_0 129 OP_3
  OP_ROLL OP_CHECKCONTRACTVERIFY OP_FROMALTSTACK OP_0 <SEL_OUT_SINGLE> OP_TX
  OP_DROP OP_NUMEQUALVERIFY OP_1 OP_ENDIF

# UPDATE_X (CCV tier, exit-key consent, dense, D=4): 711 bytes
# macros in CAPITALS are defined in MACROS_CCV.txt; raw opcodes are wrapped and indented

EXIT_CONSENT(4)
<OPERATOR_KEY> OP_CHECKSIGVERIFY OP_2 OP_PICK 40 32 OP_SUBSTR
  OP_CHECKSIGVERIFY OP_SIZE 116 OP_EQUALVERIFY OP_SWAP OP_SIZE 116
  OP_EQUALVERIFY <TAG_NODE> OP_ROT OP_ROT
LEAF_ACC
OP_ROT
LEAF_ACC
LEVEL   x4
CHECK_OLD
OP_ROT OP_DROP OP_OVER <EMPTY_4> OP_EQUAL OP_IF OP_2DROP OP_DROP OP_1 OP_ELSE
  OP_TOALTSTACK OP_SWAP OP_IF OP_2 OP_ELSE OP_0 OP_ENDIF OP_0 OP_0 129 OP_3
  OP_ROLL OP_CHECKCONTRACTVERIFY OP_FROMALTSTACK OP_0 <SEL_OUT_SINGLE> OP_TX
  OP_DROP OP_NUMEQUALVERIFY OP_1 OP_ENDIF

# CLAIM (CCV tier, dense, D=4): 1378 bytes
# macros in CAPITALS are defined in MACROS_CCV.txt; raw opcodes are wrapped and indented

OP_2 OP_PICK
LEAF_BODY
OP_2 OP_PICK OP_CAT <TREE_ID> OP_SWAP OP_CAT
TAGGED(VOUCHER)
OP_3 OP_PICK 40 32 OP_SUBSTR OP_CHECKSIGFROMSTACK OP_VERIFY
CARVE_RULES
OP_OVER OP_8 OP_LEFT OP_OVER OP_8 OP_LEFT OP_SUB <ENCO> OP_ADD OP_8 OP_LEFT
  OP_2 OP_PICK OP_8 104 OP_SUBSTR OP_CAT OP_2 OP_PICK 112 OP_4 OP_SUBSTR
  OP_1ADD OP_SIZE OP_4 OP_LESSTHANOREQUAL OP_VERIFY <ZERO4> OP_CAT OP_4
  OP_LEFT OP_CAT OP_SWAP OP_TOALTSTACK <TAG_NODE> OP_ROT OP_ROT OP_SWAP
LEAF_ACC
OP_ROT
LEAF_ACC
DRAIN_RULE
LEVEL   x4
CHECK_OLD
OP_FROMALTSTACK OP_2 OP_ROLL OP_2 OP_ROLL OP_SWAP OP_TOALTSTACK OP_TOALTSTACK

```

```

LEAF_ACC
<EMPTY_0> OP_0 OP_2SWAP
LEVEL x4
CHECK_PREV
FINAL

# CLAIM_H (CCV tier, dense, D=4): 1386 bytes
# macros in CAPITALS are defined in MACROS_CCV.txt; raw opcodes are wrapped and indented

OP_SIZE 32 OP_EQUALVERIFY OP_SHA256 OP_TOALTSTACK OP_2 OP_PICK
LEAF_BODY
OP_2 OP_PICK OP_CAT <TREE_ID> OP_SWAP OP_CAT OP_FROMALTSTACK OP_CAT
TAGGED(VOUCHER_H)
OP_3 OP_PICK 40 32 OP_SUBSTR OP_CHECKSIGFROMSTACK OP_VERIFY
CARVE_RULES
OP_OVER OP_8 OP_LEFT OP_OVER OP_8 OP_LEFT OP_SUB <ENCO> OP_ADD OP_8 OP_LEFT
  OP_2 OP_PICK OP_8 104 OP_SUBSTR OP_CAT OP_2 OP_PICK 112 OP_4 OP_SUBSTR
  OP_1ADD OP_SIZE OP_4 OP_LESSTHANOREQUAL OP_VERIFY <ZERO4> OP_CAT OP_4
  OP_LEFT OP_CAT OP_SWAP OP_TOALTSTACK <TAG_NODE> OP_ROT OP_ROT OP_SWAP
LEAF_ACC
OP_ROT
LEAF_ACC
DRAIN_RULE
LEVEL x4
CHECK_OLD
OP_FROMALTSTACK OP_2 OP_ROLL OP_2 OP_ROLL OP_SWAP OP_TOALTSTACK OP_TOALTSTACK
LEAF_ACC
<EMPTY_0> OP_0 OP_2SWAP
LEVEL x4
CHECK_PREV
FINAL

# SPLIT (CCV tier, owner-authorized carve, dense, D=4): 1266 bytes
# macros in CAPITALS are defined in MACROS_CCV.txt; raw opcodes are wrapped and indented

OP_2 OP_PICK 40 32 OP_SUBSTR OP_CHECKSIGVERIFY
CARVE_RULES
OP_OVER OP_8 OP_LEFT OP_OVER OP_8 OP_LEFT OP_SUB <ENCO> OP_ADD OP_8 OP_LEFT
  OP_2 OP_PICK OP_8 104 OP_SUBSTR OP_CAT OP_2 OP_PICK 112 OP_4 OP_SUBSTR
  OP_1ADD OP_SIZE OP_4 OP_LESSTHANOREQUAL OP_VERIFY <ZERO4> OP_CAT OP_4
  OP_LEFT OP_CAT OP_SWAP OP_TOALTSTACK <TAG_NODE> OP_ROT OP_ROT OP_SWAP
LEAF_ACC
OP_ROT
LEAF_ACC
DRAIN_RULE
LEVEL x4
CHECK_OLD
OP_FROMALTSTACK OP_2 OP_ROLL OP_2 OP_ROLL OP_SWAP OP_TOALTSTACK OP_TOALTSTACK
LEAF_ACC
<EMPTY_0> OP_0 OP_2SWAP
LEVEL x4
CHECK_PREV
FINAL

# EVICT_SBTREE_1 (CCV tier, dense, D=4): 906 bytes
# macros in CAPITALS are defined in MACROS_CCV.txt; raw opcodes are wrapped and indented

<OPERATOR_KEY> OP_CHECKSIGVERIFY OP_1 OP_TOALTSTACK <TAG_NODE> OP_1 OP_ROLL
LEAF_STEP
OP_3 OP_ROLL
LEAF_STEP
HASH_STEP(4)
OP_FROMALTSTACK OP_DROP <EMPTY_1> OP_0
LEVEL x3
CHECK_OLD
FINAL

```

```
# EVICT_SUBTREE_2 (CCV tier, dense, D=4): 1284 bytes
# macros in CAPITALS are defined in MACROS_CCV.txt; raw opcodes are wrapped and indented

<OPERATOR_KEY> OP_CHECKSIGVERIFY OP_1 OP_TOALTSTACK <TAG_NODE> OP_1 OP_ROLL
LEAF_STEP
OP_3 OP_ROLL
LEAF_STEP
HASH_STEP(4)
OP_3 OP_ROLL
LEAF_STEP
OP_5 OP_ROLL
LEAF_STEP
HASH_STEP(6)
HASH_STEP(4)
OP_FROMALTSTACK OP_DROP <EMPTY_2> OP_0
LEVEL x2
CHECK_OLD
FINAL
```

D.12 CCV Tier, 0xC2 Numerics, Sparse Profile ($D = 8$)

```
# EXIT_1 (CCV tier, sparse, D=8): 1507 bytes
# macros in CAPITALS are defined in MACROS_CCV.txt; raw opcodes are wrapped and indented

OP_DUP OP_TOALTSTACK OP_IF OP_OVER 104 OP_4 OP_SUBSTR OP_CHECKLOCKTIMEVERIFY
  OP_DROP OP_OVER 72 32 OP_SUBSTR OP_ELSE OP_OVER 40 32 OP_SUBSTR OP_ENDIF
  OP_CHECKSIGVERIFY OP_SIZE 116 OP_EQUALVERIFY
CCV_PAYOUT(1)
OP_1 <SEL_OUT_SINGLE> OP_TX OP_DROP OP_OVER OP_8 OP_LEFT OP_FROMALTSTACK
  OP_DROP OP_NUMEQUALVERIFY
LEAF_POS
SPARSE_SETUP
LEAF_ACC
<TAG_NODE> OP_ROT OP_ROT <EMPTY_0> OP_0
LEVEL_S(k) for k = 0 to 7
CHECK_OLD_S
FINAL

# JOIN_0 (CCV tier, sparse, D=8): 1491 bytes
# macros in CAPITALS are defined in MACROS_CCV.txt; raw opcodes are wrapped and indented

<OPERATOR_KEY> OP_CHECKSIGVERIFY OP_SIZE 116 OP_EQUALVERIFY
LEAF_POS
SPARSE_SETUP
LEAF_ACC
<TAG_NODE> OP_ROT OP_ROT <EMPTY_0> OP_0 OP_2SWAP
LEVEL_S(k) for k = 0 to 7
CHECK_OLD_S
FINAL

# JOIN_2 (CCV tier, sparse, D=8): 2878 bytes
# macros in CAPITALS are defined in MACROS_CCV.txt; raw opcodes are wrapped and indented

<OPERATOR_KEY> OP_CHECKSIGVERIFY OP_SIZE 116 OP_EQUALVERIFY
LEAF_POS
SPARSE_SETUP
LEAF_ACC
<TAG_NODE> OP_ROT OP_ROT <EMPTY_0> OP_0 OP_2SWAP
LEVEL_S(k) for k = 0 to 7
CHECK_OLD_S
OP_SWAP OP_TOALTSTACK OP_TOALTSTACK OP_DROP OP_SIZE 116 OP_EQUALVERIFY
LEAF_POS
SPARSE_SETUP
LEAF_ACC
<TAG_NODE> OP_ROT OP_ROT <EMPTY_0> OP_0 OP_2SWAP
LEVEL_S(k) for k = 0 to 7
CHECK_PREV_S
```

FINAL

```
# UPDATE (CCV tier, sparse, D=8): 1715 bytes
# macros in CAPITALS are defined in MACROS_CCV.txt; raw opcodes are wrapped and indented

<OPERATOR_KEY> OP_CHECKSIGVERIFY OP_2 OP_PICK 40 32 OP_SUBSTR
  OP_CHECKSIGVERIFY OP_SIZE 116 OP_EQUALVERIFY OP_SWAP OP_SIZE 116
  OP_EQUALVERIFY
  LEAF_POS
  OP_ROT
  LEAF_POS
  OP_ROT OP_EQUALVERIFY OP_SWAP
  LEAF_POS
  SPARSE_SETUP
  <TAG_NODE> OP_ROT OP_ROT
  LEAF_ACC
  OP_ROT
  LEAF_ACC
  LEVEL_S(k) for k = 0 to 7
  CHECK_OLD_S
  OP_ROT OP_DROP OP_OVER <EMPTY_8> OP_EQUAL OP_IF OP_2DROP OP_DROP OP_1 OP_ELSE
    OP_TOALTSTACK OP_SWAP OP_IF OP_2 OP_ELSE OP_0 OP_ENDIF OP_0 OP_0 129 OP_3
    OP_ROLL OP_CHECKCONTRACTVERIFY OP_FROMALTSTACK OP_0 <SEL_OUT_SINGLE> OP_TX
    OP_DROP OP_NUMEQUALVERIFY OP_1 OP_ENDIF
```

D.13 CCV Tier, 0xC0 Numerics, Dense Profile ($D = 4$)

```
# CLAIM (CCV with 0xC0 numerics, in MID macros, dense, D=4): 2316 bytes
# macros in CAPITALS are defined in MACROS_MID.txt; raw opcodes are wrapped and indented

32 OP_PICK 32 OP_PICK 32 OP_PICK 32 OP_PICK 32 OP_PICK 32 OP_PICK OP_CAT
  OP_CAT OP_CAT OP_CAT OP_CAT 24 OP_PICK 24 OP_PICK 24 OP_PICK 24 OP_PICK 24
  OP_PICK 24 OP_PICK 24 OP_PICK OP_CAT OP_CAT OP_CAT OP_CAT OP_CAT OP_CAT
  OP_CAT <TREE_ID> OP_SWAP OP_CAT
  TAGGED(VOUCHER)
  32 OP_PICK OP_CHECKSIGFROMSTACK OP_VERIFY
  CARVE_RULES
  20 OP_PICK OP_7 OP_PICK OP_7 OP_ROLL OP_7 OP_ROLL
  PADZ
  OP_EQUALVERIFY OP_TOALTSTACK OP_TOALTSTACK OP_TOALTSTACK OP_TOALTSTACK
    OP_TOALTSTACK
  LEAF_JOIN
  OP_TOALTSTACK OP_8 OP_PICK OP_8 OP_PICK OP_8 OP_PICK OP_8 OP_PICK OP_8
    OP_PICK OP_CAT OP_CAT OP_CAT OP_CAT OP_TOALTSTACK
  LEAF_JOIN
  TAGGED(LEAF)
  OP_FROMALTSTACK OP_DUP OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK
    OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK
    OP_FROMALTSTACK OP_6 OP_PICK OP_TOALTSTACK OP_5 OP_PICK OP_TOALTSTACK OP_6
    OP_ROLL OP_DROP OP_7 OP_ROLL OP_6 OP_ROLL OP_SUB OP_DUP OP_0
    OP_GREATERTHANOREQUAL OP_VERIFY OP_DUP OP_3 OP_ROLL OP_3 OP_ROLL
  LE64
  OP_4 OP_ROLL OP_1ADD OP_4 OP_ROLL OP_4 OP_ROLL
  PADZ
  OP_3 OP_ROLL OP_SWAP OP_CAT OP_CAT
  TAGGED(LEAF)
  OP_SWAP
  DRAIN_RULE
  <TAG_NODE> OP_4 OP_ROLL OP_4 OP_ROLL OP_4 OP_ROLL OP_4 OP_ROLL
  LEVEL_MID x4
  OP_2SWAP OP_6 OP_ROLL OP_6 OP_ROLL
  LE64
  <SEL_IN_AMOUNT> OP_TX OP_EQUALVERIFY 129 OP_0 129 129 OP_CHECKCONTRACTVERIFY
    OP_FROMALTSTACK OP_FROMALTSTACK
  TAGGED(LEAF)
  OP_SWAP OP_2SWAP OP_SWAP OP_TOALTSTACK OP_TOALTSTACK <EMPTY_0> OP_0 OP_2SWAP
  LEVEL_MID x4
```

```

CHECK_PREV
OP_ROT OP_DROP OP_OVER <EMPTY_4> OP_EQUAL OP_IF OP_2DROP OP_2DROP OP_1
  OP_ELSE OP_3 OP_ROLL OP_3 OP_ROLL
LE64
OP_TOALTSTACK OP_0 OP_0 129 OP_0 OP_CHECKCONTRACTVERIFY OP_FROMALTSTACK OP_0
  <SEL_OUT_SINGLE> OP_TX OP_DROP OP_EQUALVERIFY OP_1 OP_ENDIF

# CLAIM_H (CCV with 0xC0 numerics, in MID macros, dense, D=4): 2324 bytes
# macros in CAPITALS are defined in MACROS_MID.txt; raw opcodes are wrapped and indented

OP_SIZE 32 OP_EQUALVERIFY OP_SHA256 OP_TOALTSTACK 32 OP_PICK 32 OP_PICK 32
  OP_PICK 32 OP_PICK 32 OP_PICK 32 OP_PICK OP_CAT OP_CAT OP_CAT OP_CAT OP_CAT
  24 OP_PICK 24 OP_PICK 24 OP_PICK 24 OP_PICK 24 OP_PICK 24 OP_PICK 24
  OP_PICK OP_CAT OP_CAT OP_CAT OP_CAT OP_CAT OP_CAT OP_CAT OP_CAT <TREE_ID> OP_SWAP
  OP_CAT OP_FROMALTSTACK OP_CAT <TAG_VOUCHER_H> OP_SWAP OP_CAT OP_SHA256 32
  OP_PICK OP_CHECKSIGFROMSTACK OP_VERIFY
CARVE_RULES
20 OP_PICK OP_7 OP_PICK OP_7 OP_ROLL OP_7 OP_ROLL
PADZ
OP_EQUALVERIFY OP_TOALTSTACK OP_TOALTSTACK OP_TOALTSTACK OP_TOALTSTACK
  OP_TOALTSTACK
LEAF_JOIN
OP_TOALTSTACK OP_8 OP_PICK OP_8 OP_PICK OP_8 OP_PICK OP_8 OP_PICK OP_8
  OP_PICK OP_CAT OP_CAT OP_CAT OP_CAT OP_TOALTSTACK
LEAF_JOIN
TAGGED(LEAF)
OP_FROMALTSTACK OP_DUP OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK
  OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK
  OP_FROMALTSTACK OP_6 OP_PICK OP_TOALTSTACK OP_5 OP_PICK OP_TOALTSTACK OP_6
  OP_ROLL OP_DROP OP_7 OP_ROLL OP_6 OP_ROLL OP_SUB OP_DUP OP_0
  OP_GREATERTHANOREQUAL OP_VERIFY OP_DUP OP_3 OP_ROLL OP_3 OP_ROLL
LE64
OP_4 OP_ROLL OP_1ADD OP_4 OP_ROLL OP_4 OP_ROLL
PADZ
OP_3 OP_ROLL OP_SWAP OP_CAT OP_CAT
TAGGED(LEAF)
OP_SWAP
DRAIN_RULE
<TAG_NODE> OP_4 OP_ROLL OP_4 OP_ROLL OP_4 OP_ROLL OP_4 OP_ROLL
LEVEL_MID x4
OP_2SWAP OP_6 OP_ROLL OP_6 OP_ROLL
LE64
<SEL_IN_AMOUNT> OP_TX OP_EQUALVERIFY 129 OP_0 129 129 OP_CHECKCONTRACTVERIFY
  OP_FROMALTSTACK OP_FROMALTSTACK
TAGGED(LEAF)
OP_SWAP OP_2SWAP OP_SWAP OP_TOALTSTACK OP_TOALTSTACK <EMPTY_0> OP_0 OP_2SWAP
LEVEL_MID x4
CHECK_PREV
OP_ROT OP_DROP OP_OVER <EMPTY_4> OP_EQUAL OP_IF OP_2DROP OP_2DROP OP_1
  OP_ELSE OP_3 OP_ROLL OP_3 OP_ROLL
LE64
OP_TOALTSTACK OP_0 OP_0 129 OP_0 OP_CHECKCONTRACTVERIFY OP_FROMALTSTACK OP_0
  <SEL_OUT_SINGLE> OP_TX OP_DROP OP_EQUALVERIFY OP_1 OP_ENDIF

# SPLIT (CCV with 0xC0 numerics, in MID macros, dense, D=4): 2161 bytes
# macros in CAPITALS are defined in MACROS_MID.txt; raw opcodes are wrapped and indented

31 OP_PICK OP_CHECKSIGVERIFY
CARVE_RULES
20 OP_PICK OP_7 OP_PICK OP_7 OP_ROLL OP_7 OP_ROLL
PADZ
OP_EQUALVERIFY OP_TOALTSTACK OP_TOALTSTACK OP_TOALTSTACK OP_TOALTSTACK
  OP_TOALTSTACK
LEAF_JOIN
OP_TOALTSTACK OP_8 OP_PICK OP_8 OP_PICK OP_8 OP_PICK OP_8 OP_PICK OP_8
  OP_PICK OP_CAT OP_CAT OP_CAT OP_CAT OP_TOALTSTACK
LEAF_JOIN

```

```

TAGGED(LEAF)
OP_FROMALTSTACK OP_DUP OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK
  OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK
  OP_FROMALTSTACK OP_6 OP_PICK OP_TOALTSTACK OP_5 OP_PICK OP_TOALTSTACK OP_6
  OP_ROLL OP_DROP OP_7 OP_ROLL OP_6 OP_ROLL OP_SUB OP_DUP OP_0
  OP_GREATERTHANOREQUAL OP_VERIFY OP_DUP OP_3 OP_ROLL OP_3 OP_ROLL
LE64
OP_4 OP_ROLL OP_1ADD OP_4 OP_ROLL OP_4 OP_ROLL
PADZ
OP_3 OP_ROLL OP_SWAP OP_CAT OP_CAT
TAGGED(LEAF)
OP_SWAP
DRAIN_RULE
<TAG_NODE> OP_4 OP_ROLL OP_4 OP_ROLL OP_4 OP_ROLL OP_4 OP_ROLL
LEVEL_MID x4
OP_2SWAP OP_6 OP_ROLL OP_6 OP_ROLL
LE64
<SEL_IN_AMOUNT> OP_TX OP_EQUALVERIFY 129 OP_0 129 129 OP_CHECKCONTRACTVERIFY
  OP_FROMALTSTACK OP_FROMALTSTACK
TAGGED(LEAF)
OP_SWAP OP_2SWAP OP_SWAP OP_TOALTSTACK OP_TOALTSTACK <EMPTY_0> OP_0 OP_2SWAP
LEVEL_MID x4
CHECK_PREV
OP_ROT OP_DROP OP_OVER <EMPTY_4> OP_EQUAL OP_IF OP_2DROP OP_2DROP OP_1
  OP_ELSE OP_3 OP_ROLL OP_3 OP_ROLL
LE64
OP_TOALTSTACK OP_0 OP_0 129 OP_0 OP_CHECKCONTRACTVERIFY OP_FROMALTSTACK OP_0
  <SEL_OUT_SINGLE> OP_TX OP_DROP OP_EQUALVERIFY OP_1 OP_ENDIF

# EVICT_SUBTREE_2 (CCV with 0xC0 numerics, in MID macros, dense, D=4): 1897 bytes
# macros in CAPITALS are defined in MACROS_MID.txt; raw opcodes are wrapped and indented

<OPERATOR_KEY> OP_CHECKSIGVERIFY OP_1 OP_SWAP OP_IF OP_TOALTSTACK OP_2
  OP_PICK OP_CHECKLOCKTIMEVERIFY OP_DROP
PADZ
OP_6 OP_PICK OP_EQUALVERIFY OP_6 OP_PICK <OPERATOR_KEY> OP_EQUALVERIFY
  OP_FROMALTSTACK OP_DUP OP_1ADD OP_TOALTSTACK OP_DUP OP_TOALTSTACK OP_0
  OP_SWAP OP_10 OP_PICK OP_0 OP_2 OP_CHECKCONTRACTVERIFY OP_FROMALTSTACK
  <SEL_OUT_SINGLE> OP_TX OP_DROP OP_10 OP_PICK OP_EQUALVERIFY OP_9 OP_PICK
  OP_1 OP_CAT OP_TOALTSTACK
LEAF_JOIN
TAGGED(LEAF)
OP_FROMALTSTACK OP_FROMALTSTACK OP_ROT OP_SWAP OP_CAT OP_FROMALTSTACK OP_ROT
  OP_TOALTSTACK OP_SWAP OP_TOALTSTACK OP_ELSE OP_0 OP_TOALTSTACK <E_0.sc>
  OP_TOALTSTACK OP_ENDIF OP_SWAP OP_IF OP_TOALTSTACK OP_2 OP_PICK
  OP_CHECKLOCKTIMEVERIFY OP_DROP
PADZ
OP_6 OP_PICK OP_EQUALVERIFY OP_6 OP_PICK <OPERATOR_KEY> OP_EQUALVERIFY
  OP_FROMALTSTACK OP_DUP OP_1ADD OP_TOALTSTACK OP_DUP OP_TOALTSTACK OP_0
  OP_SWAP OP_10 OP_PICK OP_0 OP_2 OP_CHECKCONTRACTVERIFY OP_FROMALTSTACK
  <SEL_OUT_SINGLE> OP_TX OP_DROP OP_10 OP_PICK OP_EQUALVERIFY OP_9 OP_PICK
  OP_1 OP_CAT OP_TOALTSTACK
LEAF_JOIN
TAGGED(LEAF)
OP_FROMALTSTACK OP_FROMALTSTACK OP_ROT OP_SWAP OP_CAT OP_FROMALTSTACK OP_ROT
  OP_TOALTSTACK OP_SWAP OP_TOALTSTACK OP_ELSE OP_0 OP_TOALTSTACK <E_0.sc>
  OP_TOALTSTACK OP_ENDIF OP_SWAP OP_IF OP_TOALTSTACK OP_2 OP_PICK
  OP_CHECKLOCKTIMEVERIFY OP_DROP
PADZ
OP_6 OP_PICK OP_EQUALVERIFY OP_6 OP_PICK <OPERATOR_KEY> OP_EQUALVERIFY
  OP_FROMALTSTACK OP_DUP OP_1ADD OP_TOALTSTACK OP_DUP OP_TOALTSTACK OP_0
  OP_SWAP OP_10 OP_PICK OP_0 OP_2 OP_CHECKCONTRACTVERIFY OP_FROMALTSTACK
  <SEL_OUT_SINGLE> OP_TX OP_DROP OP_10 OP_PICK OP_EQUALVERIFY OP_9 OP_PICK
  OP_1 OP_CAT OP_TOALTSTACK
LEAF_JOIN
TAGGED(LEAF)
OP_FROMALTSTACK OP_FROMALTSTACK OP_ROT OP_SWAP OP_CAT OP_FROMALTSTACK OP_ROT
  OP_TOALTSTACK OP_SWAP OP_TOALTSTACK OP_ELSE OP_0 OP_TOALTSTACK <E_0.sc>

```

```

    OP_TOALTSTACK OP_ENDIF OP_SWAP OP_IF OP_TOALTSTACK OP_2 OP_PICK
    OP_CHECKLOCKTIMEVERIFY OP_DROP
PADZ
OP_6 OP_PICK OP_EQUALVERIFY OP_6 OP_PICK <OPERATOR_KEY> OP_EQUALVERIFY
    OP_FROMALTSTACK OP_DUP OP_1ADD OP_TOALTSTACK OP_DUP OP_TOALTSTACK OP_0
    OP_SWAP OP_10 OP_PICK OP_0 OP_2 OP_CHECKCONTRACTVERIFY OP_FROMALTSTACK
    <SEL_OUT_SINGLE> OP_TX OP_DROP OP_10 OP_PICK OP_EQUALVERIFY OP_9 OP_PICK
    OP_1 OP_CAT OP_TOALTSTACK
LEAF_JOIN
TAGGED(LEAF)
OP_FROMALTSTACK OP_FROMALTSTACK OP_ROT OP_SWAP OP_CAT OP_FROMALTSTACK OP_ROT
    OP_TOALTSTACK OP_SWAP OP_TOALTSTACK OP_ELSE OP_0 OP_TOALTSTACK <E_0.sc>
    OP_TOALTSTACK OP_ENDIF OP_DROP <TAG_NODE> OP_FROMALTSTACK OP_FROMALTSTACK
    OP_FROMALTSTACK OP_FROMALTSTACK
EVICT_HASH_STEP(4)
OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK OP_FROMALTSTACK
EVICT_HASH_STEP(6)
EVICT_HASH_ROOT(4)
<EMPTY_2> OP_0
LEVEL_MID x2
OP_2SWAP OP_6 OP_ROLL OP_6 OP_ROLL
LE64
<SEL_IN_AMOUNT> OP_TX OP_EQUALVERIFY 129 OP_0 129 129 OP_CHECKCONTRACTVERIFY
    OP_ROT OP_DROP OP_OVER <EMPTY_4> OP_EQUAL OP_IF OP_2DROP OP_2DROP OP_1
    OP_ELSE OP_3 OP_ROLL OP_3 OP_ROLL
LE64
OP_TOALTSTACK OP_0 OP_0 129 OP_0 OP_CHECKCONTRACTVERIFY OP_FROMALTSTACK OP_0
    <SEL_OUT_SINGLE> OP_TX OP_DROP OP_EQUALVERIFY OP_1 OP_ENDIF

```