

FlatMerkle: Logarithmic-Space Inclusion Proofs over a Single Linear Hash Stream

Sergio Demian Lerner

Fairgate Labs Rootstock Labs

`sergio@fairgate.io` `sergio@rootstocklabs.com*`

October 14, 2026

Abstract

Merkle trees are the standard way to obtain logarithmic-size, logarithmic-time inclusion proofs over a large dataset, but they require a tree-shaped hash structure distinct from the ordinary, linear (Merkle–Damgård) hashing that most deployed systems – including Bitcoin’s own transaction and witness hashing – actually perform. We present FlatMerkle, a construction that achieves the same asymptotic guarantees, $O(\log N)$ proof size and $O(\log N)$ verification time, over a single ordinary linear hash stream, by periodically re-injecting the hash function’s own past internal states (“checkpoints”) back into the stream as ordinary message content. The result is a flat byte sequence whose digest is a single, unmodified invocation of the underlying hash function – syntactically an ordinary digest of an ordinary byte string, not a distinguishable commitment format – while still supporting compact inclusion proofs for any element, and compact proofs that a published file is malformed, so that an honest party can always answer a malicious committer in logarithmic space. We give the recursive construction, a closed-form formula locating any element in the structured stream, worked examples with illustrations, a security reduction to free-start collision resistance of the hash function’s compression step, an optimal element-size analysis for both a flexible toy primitive and real SHA-256, a batched multi-element proof construction, a measured construction-speed comparison against Merkle trees, a concrete application to committing data inside a Bitcoin transaction annex (BIP-341), a design sketch using that application as a data-availability layer for an optimistic Bitcoin rollup, a from-scratch gas-cost comparison, on Ethereum, Rootstock, and Arbitrum, of an in-contract FlatMerkle commitment against an in-contract Merkle tree, tracing the three chains’ differing verdicts to specific protocol design choices rather than to anything intrinsic to either construction, and a comparison against the closest prior art we are aware of, authenticated append-only skip lists. Every number, table, and figure about the construction itself was checked against an executable, self-tested reference implementation, publicly available at <https://github.com/SergioDemianLerner/FlatMerkle> and described at the end of Section 1; the gas-cost figures are derived from cited protocol constants.

*This paper, its accompanying illustrations, and its reference implementation were prepared with the assistance of Anthropic’s Claude (Sonnet 5, Sonnet 5.5, and Opus 5.5) as an AI writing and coding assistant, and reviewed with the assistance of GPT 5.6 SOL. All claims about the construction were independently verified against the executable reference implementation described herein. *Competing interests:* the author is affiliated with Fairgate Labs and Rootstock Labs. Rootstock Labs contributes to the development of Rootstock, one of the three chains compared in Section 11 and Appendix A; BitVMX and ESSPI, used in Section 10, were developed with the support of both organizations and are co-authored by the author.

Contents

1	Introduction	3
2	Construction	6
2.1	Setup	6
2.2	Recursive block structure	6
2.3	Checkpoints	7
2.4	Size	7
2.5	Position formula	8
3	Inclusion proof: construction and verification	8
3.1	Two kinds of revealed value	8
3.2	Building a proof	12
3.3	Verifying a proof	13
3.4	Verifier compute cost varies by index; proof size does not	15
4	Worked examples: $N = 16$	16
4.1	Index 7 in detail	16
4.2	Reading the checkpoint labels	18
5	Security	18
5.1	Assumptions	18
5.2	Soundness	19
5.3	Completeness	21
5.4	Binding: what an accepting proof actually certifies	21
5.5	Fraud proofs: proving a published file is not canonically built	22
6	Optimal element size	28
6.1	A flexible toy primitive	28
6.2	Real SHA-256: the block size is fixed at 64 bytes	29
7	Batched multi-element inclusion proofs	29
7.1	Construction	30
7.2	Soundness	32
7.3	Measured savings	33
8	Construction speed vs. a Merkle tree	33
9	Application: committing data in a Bitcoin transaction annex	35
9.1	Why this needs a flat hash, not a tree	35
9.2	What this buys	35
9.3	Alternatives within a linearly hashed annex	36
9.4	What FlatMerkle verification does <i>not</i> give you: binding sha_annex to a specific transaction	36
9.5	Block alignment	38
9.6	Caveats	39

10 Application: a Bitcoin data-availability layer for an optimistic rollup	40
10.1 Setup: the state-transition commitment	40
10.2 Two independent watchtower checks	41
10.3 Enforcing the checks on Bitcoin	41
10.4 Scope	42
11 Application: an EVM rollup calldata commitment	42
11.1 Why calldata, not blobs	43
11.2 Four ways to commit to published calldata	43
11.3 Result	44
12 Related work	45
12.1 Linking-based time-stamping	45
12.2 Authenticated append-only skip lists	45
12.3 Other related, but structurally distinct, work	45
12.4 Scope of this comparison	46
13 Discussion and conclusion	46
13.1 Beyond SHA-256: sponge-based hash functions	46
13.2 Incremental growth	47
13.3 Non-power-of-two N	47
13.4 Summary	47
A EVM rollup calldata-commitment costs: full derivation	48
A.1 A protocol fact the naive comparison misses	48
A.2 The four options on Ethereum, computed	49
A.3 Rootstock: the same options, different protocol rules	50
A.4 Arbitrum: the missing combination	51
A.5 Two narrower points, and the general lesson	52

1 Introduction

An inclusion proof lets a verifier, given only a short commitment to a large dataset, confirm that a particular element is really part of that dataset – without holding the dataset itself. The standard tool for this is the Merkle tree [21]: leaves are hashed, pairs of hashes are hashed together recursively up to a root, and an inclusion proof consists of the $O(\log N)$ sibling hashes on the path from a leaf to the root. Both proof size and verification time are $O(\log N)$ in the number of elements N .

This efficiency comes at a structural cost: the committed data must be hashed *as a tree*. Many real systems, however, do not have that freedom. A Bitcoin transaction’s witness data, for instance, can carry an optional *annex*, whose commitment is defined by BIP-341 [36] as a single, ordinary SHA-256 of the annex bytes – a flat, linear Merkle–Damgård hash, with no tree structure available at the consensus layer: consensus only ever computes $\text{SHA256}(\text{annex bytes})$, once, linearly.

Threat model

The setting this paper targets is adversarial on the committing side. One party – the *committer* – publishes data and its digest; other parties – *opponents* – later need to convince a verifier who holds only the digest (a Bitcoin script, a dispute program, a contract) of facts about that data. The

requirement is that, for *every* byte string the committer might publish, an honest opponent who holds it can always prove something in $O(\log N)$ space and verification time: either the inclusion of any element, or – if the commitment is malformed so that inclusion proofs for the published data cannot be built – the malformation itself. A malicious committer will choose whichever of the two is weaker, so if either proof costs $O(N)$, the scheme as a whole offers only $O(N)$ guarantees.

This is what rules out the obvious way of embedding a Merkle tree in a linearly hashed field. Appending a Merkle root as the field’s last 32 bytes gives logarithmic inclusion proofs (one compression call from a revealed midstate opens the root, then an ordinary Merkle path), but a committer can publish a root that disagrees with the published data at a single position near the start. Exposing that requires proving which bytes physically sit at that position, and in a linear hash this means recomputing the chain from there to the end: $O(N)$. Section 9.3 compares this and other alternatives in detail.

This raises a natural question: can a *linear* hash stream support $O(\log N)$ inclusion proofs *and* $O(\log N)$ proofs of malformation, without changing what is actually hashed at the commitment layer?

This paper

We show that it can. The idea is simple to state: periodically, as the stream is hashed, re-inject the hash function’s own current internal state back into the stream as if it were ordinary message content. Later, a proof can reveal one of these re-injected states instead of the raw data it summarizes, letting the verifier skip an entire sub-range of the file in a single hash-function call. Applied recursively at doubling distances (2, 4, 8, 16, ...), this yields the same $O(\log N)$ proof size and $O(\log N)$ verification time as a Merkle tree – while the file, from the outside, is still just one ordinary flat hash of one ordinary byte string. Because the re-injected states make any position of the stream reachable in $O(\log N)$ calls, the same mechanism also lets an opponent prove, in $O(\log N)$, that a published file deviates from the canonical layout (Section 5.5), meeting both halves of the threat model. The price is size: the structured file is about twice the size of the raw data (Section 2). We call the resulting structure **FlatMerkle**.

The security of the construction rests on a specific, well-studied but non-standard assumption: *free-start* collision resistance of the hash function’s raw compression step (Section 5) – the hardness of finding a compression-function collision when the adversary is free to choose the starting state, not just the message. This is strictly stronger than the ordinary (IV-fixed) collision resistance that hash functions are normally analyzed under, and we state it as an explicit, separate assumption rather than treating it as automatic.

Contributions

- A formal recursive construction of the structured file, and a closed-form formula, $\text{position}(a) = 2a - \text{popcount}(a)$, that locates any original element’s *item index* within the structured stream without a lookup table – a count of items, not literally a byte offset; multiplying by the fixed per-item byte width gives the byte offset (Section 2). The inclusion-proof path itself comes from a separate, simpler branch rule (Section 3).
- Single-element proof construction and verification algorithms, with worked examples and illustrations for a 16-element file (Sections 3–4).
- A security reduction: forging a FlatMerkle inclusion proof, or making a single digest accept two conflicting answers for the same index, implies finding a free-start collision in the underlying

compression function. This binding guarantee does *not* extend to files that were never canonically built in the first place – we give a working forgery against that stronger claim – so we also give a complementary $O(N)$ -detect/ $O(\log N)$ -prove fraud-proof mechanism that lets any party holding the full file catch and prove a non-canonical layout, turning soundness against a malicious publisher into an optimistic rather than unconditional guarantee (Section 5).

- Optimal element-size analysis minimizing average proof size, both for a size-flexible toy primitive and for real, fixed-block SHA-256 (Section 6).
- A batched multi-element inclusion proof construction with measured space savings against m independent proofs (Section 7).
- A measured construction-speed comparison against Merkle trees, using an identical from-scratch SHA-256 compression function on both sides (Section 8).
- A concrete application: committing a FlatMerkle structured file inside a Bitcoin transaction annex, so that BIP-341’s existing SHA256(annex) commitment doubles as a FlatMerkle root, with no consensus changes (Section 9).
- A design sketch composing this annex application with BitVMX and BitVM3, two Bitcoin computation-verification protocols, and ESSPI [15, 16, 18] to use a FlatMerkle-committed annex as the data-availability layer for an optimistic rollup, with watchtowers disputing both non-canonical layouts and individually incorrect state transitions (Section 10).
- A from-scratch gas-cost comparison of four ways to commit rollup calldata on-chain, using current protocol constants, on Ethereum, Rootstock, and Arbitrum [1, 4, 5, 11, 14, 22, 23, 27–30, 33, 35]: the three chains disagree on whether an in-contract FlatMerkle commitment or an in-contract Keccak Merkle tree is cheaper, tracing the difference to two separable protocol design choices (a calldata floor and a warm-call discount for precompiles) rather than to anything about FlatMerkle itself; both remain a clear improvement over submitting an externally built FlatMerkle file or persisting the raw data in storage, on all three chains (Section 11).
- A comparison against the closest prior art we identified, authenticated append-only skip lists [8, 20] (Section 12).

Every number, table, and figure about the construction itself was produced by an executable, self-tested reference implementation (a size-flexible toy compression function and a real-SHA-256 variant), not asserted from hand calculation. The exceptions are the EVM gas-cost figures of Section 11 and Appendix A, which are derived by hand from cited protocol constants plus a flagged estimate, and the build-time extrapolation of Section 8. For single-element proofs, the self-tests flip every individual revealed value, one at a time, for every index at every tested N , and confirm rejection; multiproofs are checked the same way across many randomized target sets, tampering the data, the claimed data, one shared reveal, the index-to-pair assignment, and the pair set per trial. The reference implementation – filenames retain “LogLink,” this project’s working name before it was renamed FlatMerkle; the construction itself is unaffected – `loglink.py` (toy compression function), `loglink_annex.py` (real SHA-256), `illustrate.py` (figures), `fraud_proof.py` and `simulate_fraud_proofs.py` (Section 5.5), `experiments.py` and `benchmark.py` (Sections 7 and 8) – is publicly available, together with the paper’s source and the benchmark data, at <https://github.com/SergioDemianLerner/FlatMerkle>. The repository also contains a line-by-line Java transliteration of the core algorithms for cross-language sanity-checking, which should not be read as an independently designed second implementation.

2 Construction

2.1 Setup

Let $\mathcal{C} : \{0, 1\}^w \times \{0, 1\}^b \rightarrow \{0, 1\}^w$ be the raw compression function underlying a Merkle–Damgård hash function H : w is the internal state width in bits (256 for SHA-256), and b is the message block width in bits (512 for SHA-256). Let $\text{IV} \in \{0, 1\}^w$ be the function’s fixed public initial state.

Let $D[0], D[1], \dots, D[N-1]$ be $N = 2^L$ original data elements, each padded to *exactly* b bits – one full compression-function message block – so that $\mathcal{C}(\text{state}, D[i])$ typechecks the same way it does for every other item in the file; Section 6 discusses the optimal choice of the pre-padding element size subject to this constraint. The domain is $N = 2^L$ for an integer $L \geq 2$, i.e. $N \in \{4, 8, 16, \dots\}$: $L \geq 1$ would let $\text{block}(a, 1)$ itself be well-defined, but the top-level split into two root halves $\text{block}(0, L-1)$ and $\text{block}(2^{L-1}, L-1)$ that Section 3’s VERIFY relies on needs $L-1 \geq 1$, so $N = 2$ ($L = 1$) is excluded as degenerate. Elements are numbered left to right: $D[0]$ is hashed first, $D[N-1]$ last – followed only by the $L-1$ MID blocks that close the recursion’s still-open ancestor blocks (Figure 1), and then the file’s finalization.

2.2 Recursive block structure

The structured file is built by a recursive procedure $\text{block}(a, k)$, which emits the byte content covering the 2^k original elements $D[a], \dots, D[a + 2^k - 1]$:

$$\text{block}(a, 1) = D[a] \parallel D[a+1] \parallel \text{MID}(a) \tag{1}$$

$$\text{block}(a, k) = \text{block}(a, k-1) \parallel \text{block}(a + 2^{k-1}, k-1) \parallel \text{MID}(a), \quad k \geq 2 \tag{2}$$

where $\text{MID}(a)$ is a single hash block whose content is the running checkpoint $\text{CK}[a]$ (defined below), padded to the block width b :

$$\text{MID}(a) = \text{pad}_b(\text{CK}[a]). \tag{3}$$

Definition 2.1 (Block padding). *For a w -bit state value $x \in \{0, 1\}^w$ and target block width b bits, $\text{pad}_b : \{0, 1\}^w \rightarrow \{0, 1\}^b$ is any fixed, public, injective function of x alone – content-independent, not a new padding scheme beyond the ordinary single-block append-a-1-bit convention already used elsewhere in Merkle–Damgård padding. Injectivity is the only property Theorem 5.8’s and Theorem 5.9’s proofs actually use; nothing below depends on the specific encoding beyond that. Two instances are used in this paper:*

- **$b > w$ (real SHA-256, Sections 6.2 onward):** $\text{pad}_b(x) = x \parallel 0\mathbf{x}80 \parallel 0^{b-w-8}$ – a single 1-bit followed by zero bits, filling the block out to exactly b bits, trivially injective since x is recoverable as the first w bits. For SHA-256, $w = 256$ bits (32 bytes) and $b = 512$ bits (64 bytes), so this appends the single byte $0\mathbf{x}80$ followed by 31 zero bytes to the 32-byte checkpoint, filling the 64-byte block exactly.
- **$b = w$ (the toy primitive, Section 4):** $\text{pad}_w(x) = x$, the identity function – also trivially injective, but with no spare byte for a terminator, since block and checkpoint are the same width. This is the encoding the toy reference implementation actually uses.

$b < w$ is never valid under either scheme, or any other injective encoding: a w -bit value cannot be injectively embedded in a strictly narrower b -bit block, so $b \geq w$ is a hard requirement on the compression function’s parameters, not a property of the padding scheme chosen (Section 6 returns to this as a feasibility constraint on element size).

The whole file is:

$$F = \text{block}(0, L-1) \parallel \text{block}(2^{L-1}, L-1). \quad (4)$$

Note there is no top-level MID closing the two halves together – it would be redundant, since the file’s own finalization (the hash function’s standard length-padding) already plays that role.

2.3 Checkpoints

Definition 2.2 (Checkpoint). *For each original index a that starts some $\text{block}(a, k)$, $\text{CK}[a]$ is the running compression-chain state immediately before the file’s content for element a begins. $\text{CK}[a]$ is recorded exactly once, the first time position a begins a block, and reused verbatim by every later reference to it – in particular, by $\text{block}(a, k)$ ’s own closing $\text{MID}(a) = \text{pad}_b(\text{CK}[a])$, regardless of how many different k a single position a happens to start a block for (Section 3’s worked $N=8$ trace shows $a = 0$ starting two nested blocks, with $\text{MID}(0)$ emitted – carrying the same $\text{CK}[0]$ bytes both times – at two different points in the file). The one exception is $\text{CK}[0]$ itself, which is not any block’s ending but the hash function’s fixed public IV.*

Every $\text{block}(a, k)$ other than the two top-level ones has a parent $\text{block}(p, k+1)$, of which it is either the *left child* ($a = p$, equivalently $a \bmod 2^{k+1} = 0$) or the *right child* ($a = p + 2^k$). The checkpoint labels used in the figures and examples follow this distinction:

- A **left child**’s closing value $\text{ending}(a, k)$ (Definition 3.1) becomes $\text{CK}[a + 2^k]$, the checkpoint of the sibling that follows it; we label it CK_{a+2^k} .
- A **right child**’s $\text{ending}(a, k)$ is never re-injected as a checkpoint; it only feeds its parent’s closing MID. We label it by the span it summarizes, $\text{CK}_{[a, a+2^k-1]} := \text{ending}(a, k)$.

A bracketed label therefore always names an ending. The state one compression call earlier, $\text{prewrap}(a, k)$, is for $k \geq 2$ the ending of $\text{block}(a, k)$ ’s right child, so it is labelled $\text{CK}_{[a+2^{k-1}, a+2^k-1]}$, not $\text{CK}_{[a, a+2^k-1]}$; confusing the two is the label-level form of the verifier bug of Remark 3.5. For $k = 1$, $\text{prewrap}(a, 1)$ (the state after $D[a], D[a+1]$) has no label and is written out.

2.4 Size

Each call to block at level $k = 1$ emits 2 data elements and 1 MID block; each call at level $k \geq 2$ emits 1 additional MID block on top of its two children. Summing over the full recursion, the total number of items in the file is exactly

$$M = 2N - 2, \quad (5)$$

i.e. N data elements and $N - 2$ MID blocks (no top-level MID, as noted above). The expansion factor $M/N = 2 - 2/N$ approaches exactly 2 as $N \rightarrow \infty$: asymptotically, the structured file is twice the size of the raw data, independent of N .

Building the file also has a modest memory footprint: although M items are eventually emitted, the recursive definition of $\text{block}(a, k)$ only ever needs the checkpoints of currently-open ancestor blocks – at most L of them at any one time – so a streaming implementation can construct the whole file in a single left-to-right pass using $O(\log N)$ working memory, discarding each checkpoint once its block closes. (The reference implementation instead keeps every checkpoint in one dictionary for the whole build, for simplicity, not because $O(N)$ memory is required.)

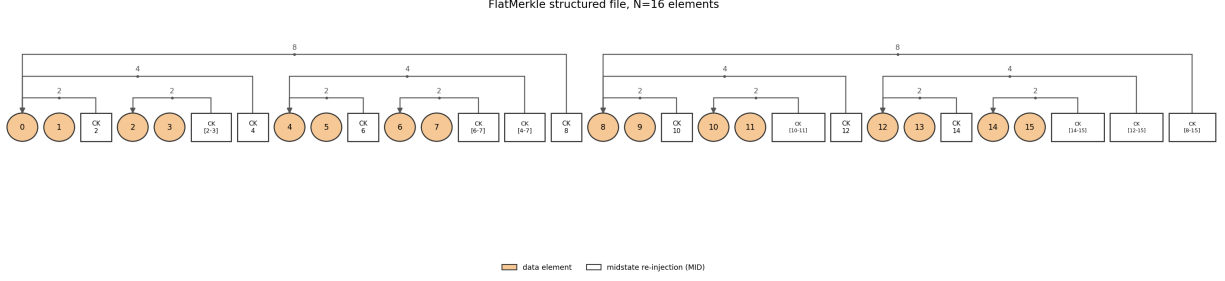


Figure 1: The structured file for $N = 16$ elements, in full: data elements as circles, MID (checkpoint re-injection) blocks as boxes labelled by the rule above, and the distance- 2^k jump arcs implicit in the recursion, drawn above the row with an arrowhead into the element each arc's checkpoint traces back to. Read each box's label as naming the state that box's arc points *at* (the state *entering* the block it opens), not the block immediately to its left: the box labelled CK_4 , for instance, physically *contains* $CK[0] = IV$ and its arc points back at element 0 – it is promoting that value forward to be the checkpoint of the block starting four elements later, not describing anything about its own immediate neighbors.

2.5 Position formula

To build or verify a proof for original index a without materializing the whole file, we need to know exactly where in the structured file's byte stream element a 's content sits.

Lemma 2.3 (Position formula). *For any original index $a \in [0, N)$ – even or odd, not only positions that start a block – the 0-indexed position of $D[a]$ in the structured file is*

$$\text{position}(a) = 2a - \text{popcount}(a), \quad (6)$$

where $\text{popcount}(a)$ is the number of 1-bits in the binary representation of a .

Proof sketch. By induction on the recursive structure of Section 2.2. Every element contributes exactly one data slot; every completed pair contributes one additional MID slot, and more generally every completed block at level $k \geq 1$ contributes one MID slot for each level it closes. The number of MID slots emitted strictly before position a is exactly the number of block-closing boundaries crossed on the way to a , which – by the standard correspondence between binary counter increments and block closures in a doubling recursive structure – equals $a - \text{popcount}(a)$. Adding the a data slots already emitted gives $\text{position}(a) = a + (a - \text{popcount}(a)) = 2a - \text{popcount}(a)$. $\square$

The prover and verifier of Section 3 never call $\text{position}(\cdot)$: they branch only on whether the target lies below a block's midpoint. The formula is for a party holding only the raw byte stream, who needs to locate $D[a]$ directly – for example, to extract an element from a downloaded Bitcoin annex (Section 9) before building a proof. It is verified in the accompanying code against the recursive construction for every index at every $N \in \{4, 8, 16, 32, 64\}$.

3 Inclusion proof: construction and verification

3.1 Two kinds of revealed value

Besides the target pair's two raw data elements, every proof step reveals exactly one 32-byte (in general, w -bit) state value per level of recursion. Before saying what each reveal means, we need

two pieces of notation: the two snapshots of the running hash-chain state taken while a block is built, and the notion of a *sibling*.

Definition 3.1 (Prewrap and ending). *Building $\text{block}(a, k)$ (Section 2) is a prefix of the single, ongoing compression-chain computation for the whole file, and it ends with one specific compression call: the one that applies the block's own closing checkpoint re-injection, $\text{MID}(a)$. Write*

$$\text{prewrap}(a, k) = \text{the running state immediately before that final call,}$$

i.e. right after every element and every nested MID belonging to $\text{block}(a, k)$ other than its own closing one has been hashed, and write

$$\text{ending}(a, k) = \mathcal{C}(\text{prewrap}(a, k), \text{MID}(a))$$

for the state immediately after that call. Both are snapshots of the same single running state used to build the whole file – they are not separately computed side channels – taken one compression call apart, on either side of $\text{block}(a, k)$'s own closing step. (A checkpoint $\text{CK}[a']$ for $a' \neq 0$, Section 2, is always equal to $\text{ending}(a, k)$ for whichever block happens to finish immediately before position a' begins; the one exception is $\text{CK}[0] = \text{IV}$, which is not any block's ending.)

The generic step. Before a concrete example, here is the same pattern once, abstractly, for a single call $\text{block}(a, k)$ with $k \geq 2$. Every line below is one of exactly two kinds, and the rule is about the *word*, not the font: a line that says **emit** writes bytes to the file and feeds them into the hash chain; every other line merely *names* a value the running state already has, and writes nothing. (*Execute* is a third thing, strictly speaking – a recursive call that itself expands into a sequence of exactly this same shape, one level down – so it is set in plain text, neither bold nor colored.) To make the two real kinds easy to tell apart at a glance even where the line is mostly math, *emit* lines are **bold black**, and every value-naming line is shown in gray:

1. if not already set: $\text{CK}[a] := \text{state}$
2. execute $\text{block}(a, k-1)$
3. execute $\text{block}(a + 2^{k-1}, k-1)$
4. $\text{prewrap}(a, k) := \text{state}$
5. **emit** $\text{MID}(a)$
6. $\text{ending}(a, k) := \text{state}$

The base case $k = 1$ is the same six steps with two data emissions, **emit** $D[a]$ and **emit** $D[a+1]$, standing in for the two recursive calls (steps 2–3). Every call to $\text{block}(\cdot, \cdot)$ made anywhere in the whole construction – there are exactly $N - 2$ of them, one per MID emission – follows one of these two shapes.

The $N = 8$ example below is exactly what you get from inlining every one of the $N - 2 = 6$ calls that construction makes, in the order they actually execute, down to the individual data elements.

Example: the full build sequence for $N = 8$. To make Definition 3.1 concrete, here is the complete, real sequence of events building the structured file for a small $N = 8$ example ($D[0..7]$), exactly as produced by the reference implementation, using the same **bold** (emit) / gray (value-naming only) convention as above:

1. $\text{CK}[0] := \text{IV}$
2. **emit** $D[0]$

```

3. emit  $D[1]$ 
4.  $\text{prewrap}(0, 1) := \text{state}$ 
5. emit  $\text{MID}(0)$  (bytes =  $\text{pad}_b(\text{CK}[0])$ )
6.  $\text{ending}(0, 1) := \text{state}$ 
7.  $\text{CK}[2] := \text{state}$  (=  $\text{ending}(0, 1)$ )
8. emit  $D[2]$ 
9. emit  $D[3]$ 
10.  $\text{prewrap}(2, 1) := \text{state}$ 
11. emit  $\text{MID}(2)$  (bytes =  $\text{pad}_b(\text{CK}[2])$ )
12.  $\text{ending}(2, 1) := \text{state}$ 
13.  $\text{prewrap}(0, 2) := \text{state}$  (=  $\text{ending}(2, 1)$ )
14. emit  $\text{MID}(0)$  (again) (bytes =  $\text{pad}_b(\text{CK}[0])$ , same bytes as step 5)
15.  $\text{ending}(0, 2) := \text{state}$ 
16.  $\text{CK}[4] := \text{state}$  (=  $\text{ending}(0, 2)$ )
17. emit  $D[4]$ 
18. emit  $D[5]$ 
19.  $\text{prewrap}(4, 1) := \text{state}$ 
20. emit  $\text{MID}(4)$  (bytes =  $\text{pad}_b(\text{CK}[4])$ )
21.  $\text{ending}(4, 1) := \text{state}$ 
22.  $\text{CK}[6] := \text{state}$  (=  $\text{ending}(4, 1)$ )
23. emit  $D[6]$ 
24. emit  $D[7]$ 
25.  $\text{prewrap}(6, 1) := \text{state}$ 
26. emit  $\text{MID}(6)$  (bytes =  $\text{pad}_b(\text{CK}[6])$ )
27.  $\text{ending}(6, 1) := \text{state}$ 
28.  $\text{prewrap}(4, 2) := \text{state}$  (=  $\text{ending}(6, 1)$ )
29. emit  $\text{MID}(4)$  (again) (bytes =  $\text{pad}_b(\text{CK}[4])$ , same bytes as step 20)
30.  $\text{ending}(4, 2) := \text{state}$  (this is the file's raw chain state, before finalization)

```

Two things are worth noticing. First, every gray, **non-emit** line is free: it is a name for a value the running state already has at that point, not a separate computation and not a separate byte written anywhere – Definition 3.1's *prewrap*/*ending* pair is nothing more than *before* and *after* labels for step 5 (or step 11, or 20, or 26 – whichever *MID* call is currently closing). Second, *MID*(0) is emitted twice (steps 5 and 14) and *MID*(4) is emitted twice (steps 20 and 29): same checkpoint bytes both times, but compressed against a different running state each time, since the same original position can start more than one nested block (here $a = 0$ starts both $\text{block}(0, 1)$ and $\text{block}(0, 2)$). This is exactly the mechanism that lets one checkpoint, computed once, be reused at every level whose block happens to start at that position – the same reuse Definition 3.1 already alludes to for $\text{CK}[a']$ in general. (14 emitted items total – 8 data, 6 *MID* – matching $2N - 2$ for $N = 8$, Section 2.)

Definition 3.2 (Sibling). *At recursion level k , block $\text{block}(a, k)$ has two children, $\text{block}(a, k - 1)$ and $\text{block}(a + h, k - 1)$ with $h = 2^{k-1}$ (Section 2). The target index t lies in exactly one of the two; call that one the target child. The **sibling** (at this level) is the other child – the one not containing t .*

Since the sibling does not contain the target, the proof does not recurse into it: it reveals a single state value that stands in for the sibling's whole contents. Which value depends on whether the sibling was built before or after the target child:

Left sibling: becomes the new checkpoint. The sibling $\text{block}(a, k-1)$ is already closed when the target child begins. Its closing state $\text{ending}(a, k-1) = \text{CK}_{a+2^{k-1}}$ is revealed and used directly as the checkpoint that starts the evaluation of the target child.

Right sibling: combined with the verified value. The sibling $\text{block}(a+h, k-1)$ is built after the target child. The proof reveals its *pre*-closing state $\text{prewrap}(a+h, k-1)$, not its ending, and the verifier performs the sibling's closing fold itself, using left_end , its own result for the target child, as the checkpoint. Revealing $\text{ending}(a+h, k-1)$ instead would make the final check independent of left_end , so any target data below this level would verify (Remark 3.5). For $k-1 \geq 2$, $\text{prewrap}(a+h, k-1)$ is the ending of the sibling's own right child and carries the label $\text{CK}_{[a+h+2^{k-2}, a+h+2^{k-1}-1]}$ (Section 2); for $k-1 = 1$ it is unlabelled.

Concretely, the fold-in step in the second case computes

$$\mathcal{C}(\text{prewrap}(a+h, k-1), \text{pad}_b(\text{left_end})), \quad (7)$$

and in a genuine proof $\text{left_end} = \text{ending}(a, k-1) = \text{CK}[a+h]$, the checkpoint that $\text{block}(a+h, k-1)$'s closing $\text{MID}(a+h)$ re-injects, so by Definition 3.1

$$\mathcal{C}(\text{prewrap}(a+h, k-1), \text{pad}_b(\text{CK}[a+h])) = \text{ending}(a+h, k-1) \quad (8)$$

exactly: the verifier recomputes the sibling's closing value from a checkpoint it has itself just verified.

Closing $\text{block}(a, k)$ itself. After both children have been accounted for, in left-to-right order, the state is $v = \text{prewrap}(a, k)$ in either case, and one further call closes $\text{block}(a, k)$ with its own checkpoint $\text{CK}[a]$, passed down from the level above:

$$\mathcal{C}(v, \text{pad}_b(\text{CK}[a])) = \mathcal{C}(\text{prewrap}(a, k), \text{MID}(a)) = \text{ending}(a, k). \quad (9)$$

This is the last line of `EVALBLOCK` below, `return C(right_end, pad_b(ckpt))`. Every level performs Equation (9) once, in addition to resolving the sibling. The two folds are easy to conflate in the pseudocode, but Equation (7) closes the *sibling* with $\text{CK}[a+h]$, while Equation (9) closes $\text{block}(a, k)$ with $\text{CK}[a]$, one level up.

3.2 Building a proof

Algorithm 3.1 BUILDPROOF(a_0, L) – prover, given target pair start a_0

```

function Recurse(a, k, t): # t = target index, in [a, a+2^k)
    if k = 1: return # base case: no reveal here,
                    # base pair always sent raw
    h := 2^(k-1)
    if t < a + h: # target in left child
        Recurse(a, k-1, t)
        reveal prewrap(a+h, k-1) # right sibling, not yet closed
    else: # target in right child
        reveal ending(a, k-1) # left sibling, already closed
        Recurse(a+h, k-1, t)

h_top := 2^(L-1)
if a0 < h_top:
    Recurse(0, L-1, a0)
    reveal prewrap(h_top, L-1)
else:
    reveal ending(0, L-1)
    Recurse(h_top, L-1, a0)

return (D[a0], D[a0+1], reveals in the order emitted above)

```

The proof carries a_0 's two data elements, not a_0 itself: the verifier (Algorithm 3.3 below) always has its own trusted target *index* and derives the pair start from *that*, never from anything the proof claims – a_0 is prover-side bookkeeping used to build the reveal sequence, not part of the wire format.

Lemma 3.3 (Index-independent proof size, for fixed N). *For every index $0 \leq a < N = 2^L$, this procedure emits exactly $L - 1$ reveals – the same count for every index at that N , not merely an average over indices. (Proof size is, as usual, still $O(\log N)$ in N itself; the point here is that it does not additionally vary with which index is proven once N is fixed.)*

This follows directly from the algorithm: each of the $L - 1$ recursive levels above the base case emits exactly one reveal, regardless of which branch is taken. (Section 3.4 discusses *verifier compute cost*, which is *not* constant across indices, even though proof size is.)

Prover time and storage. BUILDPROOF as written treats $\text{ending}(a, k)$ and $\text{prewrap}(a, k)$ as values simply available to read off – but, unlike the checkpoints $\text{CK}[a]$, neither is a literal byte range stored anywhere in the structured file (Definition 3.1): both are internal running-chain- state snapshots that existed only transiently while the file was originally built. Section 2's $O(\log N)$ working-memory claim is for the *build* step alone, and explicitly discards each checkpoint once its block closes; it says nothing about the cost of later answering an arbitrary proof request. Two honest options exist, with a real space/time trade-off between them, and a deployment needs to pick one explicitly:

- **Cache every snapshot at build time.** Retain all $O(N)$ intermediate ending/prewrap values (one 32-byte, or in general w -bit, entry per MID emission) alongside the file, rather than discarding them. BUILDPROOF then costs $O(\log N)$ time per request, matching proof

size, at the cost of $O(N)$ storage held indefinitely by whoever answers proof requests (in addition to the file itself). This is what the reference implementation does, for simplicity, exactly as Section 2 already notes for the analogous build-time choice.

- **Recompute on demand.** Keep only the file bytes and answer each proof request by re-running the hash chain from the nearest available checkpoint up to the needed snapshot. Without any cached snapshots this costs up to $O(N)$ compression calls per proof request – no better than a full re-hash – though a partial cache (e.g. one snapshot every $O(\sqrt{N})$ positions) trades a smaller, tunable storage budget against a correspondingly higher, but still sublinear, per-request recomputation cost.

Either way, the $O(\log N)$ figures emphasized elsewhere in this paper are proof *size* and verifier *compute*, not prover storage: a FlatMerkle prover that wants every request answered in $O(\log N)$ time, matching the verifier’s own cost, needs $O(N)$ storage to do it, the same asymptotic footprint an equivalent Merkle-tree prover needs to keep the whole tree (not just the root) on hand.

3.3 Verifying a proof

Algorithm 3.2 EVALBLOCK($a, k, t, ckpt$) – verifier, symmetric to RECURSE

```

function EvalBlock(a, k, t, ckpt):
  if k = 1:
    s := C(C(ckpt, D0), D1) # the proof's base pair
    return C(s, pad_b(ckpt)) # re-inject checkpoint,
                              # closing the base pair
  h := 2^(k-1)
  if t < a + h: # left branch
    left_end := EvalBlock(a, k-1, t, ckpt)
    sib_prewrap := read next reveal
    right_end := C(sib_prewrap, pad_b(left_end))
                  # fold in the value just verified
    return C(right_end, pad_b(ckpt))
  else: # right branch
    left_end := read next reveal
    right_end := EvalBlock(a+h, k-1, t, left_end)
    return C(right_end, pad_b(ckpt))

```

Algorithm 3.3 $\text{VERIFY}(N, \Delta, \text{index}, \text{data}, \pi)$ – top-level verifier

```
function Verify(N, Delta, index, data, pi):
  if N < 4 or N is not a power of two: return false # domain: N = 2^L, L >= 2
  if index < 0 or index >= N: return false # index out of range
  (D0, D1, reveals) := pi # reveals: a shared queue,
                           # consumed front-to-back by
                           # every "read next reveal"
                           # line below and inside
                           # EvalBlock (Section 3.3)
  a0 := index - (index mod 2) # pair start, derived from
                              # the verifier's OWN index --
                              # NEVER taken from pi
  if index = a0: expected := D0 else: expected := D1
  if data != expected: return false # supplied data disagrees with proof
  L := log2(N)
  h_top := 2^(L-1)
  if a0 < h_top:
    left_end := EvalBlock(0, L-1, a0, IV) # consumes from reveals
    if reveals is empty: return false # ran out before the top fold
    sib_prewrap := read next reveal
    raw_state := C(sib_prewrap, pad_b(left_end))
  else:
    if reveals is empty: return false
    left_end := read next reveal
    raw_state := EvalBlock(h_top, L-1, a0, left_end)
  if reveals is not empty: return false # leftover reveals: malformed
  return finalize(raw_state, N) = Delta # public length-padding + compare
```

Remark 3.4 (A natural-looking but insecure alternative: trusting a proof-supplied pair start). A verifier might instead accept a pair start a_0 as part of the proof itself, $\pi = (a_0, D_0, D_1, \text{reveals})$, and branch on whether $\text{index} = a_0$ or $\text{index} = a_0 + 1$ to decide which of D_0, D_1 is “the” claimed data – exactly as `BUILDPROOF` computes a_0 internally. This is a genuine soundness hole, and a collision-free one: `EVALBLOCK`’s recursion branches only on whether the recursion target lies below a block’s midpoint, and every branch point above the base case compares against a power-of-two threshold $h = 2^{k-1} \geq 2$, so the least-significant bit of the recursion target never affects any branch decision – indices $i-1$ and i (for odd i) take an identical recursion path. An honest proof for index $i-1$ carries $D_0 = D[i-1]$, and a verifier that trusts a proof-supplied a_0 can be asked to check $(\text{index} = i, \text{data} = D[i-1], \pi)$ with a_0 relabelled to i instead of $i-1$ (same D_0, D_1 , same reveals, unchanged): the trusting verifier sees $\text{index} = a_0$, sets $\text{expected} := D_0 = D[i-1]$, and accepts – reaching the genuine digest with no free-start collision at all, since the recursion itself never used a_0 for anything but this one comparison. The construction above avoids this by design: a_0 is never read from π , only derived from the verifier’s own index, which the verifier is asking about in the first place and has no reason to distrust. (The reference implementations already follow this discipline – `verify_proof` derives its pair start solely from the caller-supplied target index, and the wire-level *Proof* object carries no separate pair-start field at all.)

`EVALBLOCK`’s “read next reveal” lines share the same queue and must reject if it is empty: too few reveals is as malformed as too many, and an implementation that substitutes a default value for a missing reveal reopens a gap of the same kind as Remark 3.5’s. `VERIFY` first enforces the domain $N = 2^L$, $L \geq 2$ (Section 2; for $N = 2$ there is no top-level split), then $\text{index} \in [0, N)$,

derives a_0 from *index* alone, checks *data* against the element of the base pair that *index* selects, mirrors BUILDPROOF’s top-level split (there is no MID joining the two root blocks; the finalization plays that role), rejects missing or leftover reveals, and only then compares the finalized chain state with the digest. Both reference implementations perform all of these checks; a re-implementation of EVALBLOCK’s recursive core alone should add them.

Remark 3.5 (A natural-looking but insecure alternative). *A verifier might be tempted to compute the left branch’s left_end recursively (as required) but then discard it, returning*

$$\mathcal{C}(\text{right_end}, \text{pad}_b(\text{ckpt}))$$

without ever folding left_end into the check. This is a genuine soundness hole: tampering with any value inside the left recursion goes undetected, because the final check never actually depends on it. The construction above avoids this by design (left branch of Algorithm 3.2):

$$\text{right_end} := \mathcal{C}(\text{sib_prewrap}, \text{pad}_b(\text{left_end})).$$

The verifier must combine the just-verified left_end with the revealed sibling pre-close state, not simply trust a separately revealed post-close state for the sibling and ignore what it just computed. We recommend testing for exactly this class of omission directly – flipping a single bit in every individually revealed value and confirming rejection – since it is the kind of gap that is easy to miss by inspection alone.

3.4 Verifier compute cost varies by index; proof size does not

Although every proof has exactly $L - 1$ reveals, the number of compression-function calls the *verifier* performs to check a proof is not constant: it depends on how often the target’s recursion path takes the left branch (which costs 2 calls at that level: the fold, then the checkpoint re-injection) versus the right branch (which costs 1 call: only the checkpoint re-injection, since the sibling’s closed value is used as-is). For $N = 32$ ($L = 5$), measured directly against the real verifier implementation:

Index	0	8	15	23	30
Compression calls (incl. finalization)	11	10	8	8	7

Table 1: Verifier compression-function calls by target index, $N = 32$. The minimum, 7, is achieved by indices that take the cheap (right) branch at every level; the maximum, 11, by index 0, which takes the left branch at every level. In general the range is $[L + 2, 2L + 1]$ (including one constant finalization call): the ratio $(2L + 1)/(L + 2)$ approaches, but never reaches, $2\times$ as $L \rightarrow \infty$, and is strictly less than $2\times$ for every finite L (here, $L = 5$: $11/7 \approx 1.57\times$) – bounded, not asymptotically growing relative to L .

This asymmetry is a lever, not just a cost – but the right way to describe it is by *bit pattern*, not by index magnitude. The exact cost is $L + 2 + \#\{\text{zero bits among bits } 1..L-1 \text{ of the target}\}$, which depends only on which of those bits are set, not on how large the index is: for $N = 32$, index 16 (10000_2) costs 10 – almost as expensive as index 0’s 11, and no cheaper than index 8’s 10 – despite being numerically closer to the “high-index end” than either. Index 30 (11110_2) gets the minimum cost of 7 not because it is large, but because its high-order bits happen to all be 1. The correct lever is therefore: if the elements most likely to be queried are known in advance, placing them at indices whose high-order bits are all 1 – concretely, the last few elements of the file – favors

the cheap, right-branch-heavy verification path and reduces their average verification cost for free, with no change to the construction or to the constant proof-size guarantee. Since the whole range is bounded, $[L + 2, 2L + 1]$, the payoff of this placement is at most a factor of roughly 2 and does not grow with N – a genuine but modest optimization, not an asymptotic one.

A proof always discloses its whole base pair. BUILDPROOF returns $D[a_0]$ and $D[a_0+1]$ unconditionally, where $a_0 = a - (a \bmod 2)$ – so a proof requested for target index a always discloses *both* raw elements of its pair, not just $D[a]$: proving index 7 necessarily also reveals element 6, whether or not anyone asked about element 6. This is a real, semantic difference from an ordinary Merkle proof, whose leaf hash need not expose any neighboring leaf, and it is worth stating plainly rather than leaving implicit. It also affects the size comparison in Section 4’s table and Section 6: with element size equal to the state width, $D = W$, FlatMerkle’s proof size $2D + (L - 1)W$ and a same-shape Merkle proof’s size $LW + D$ come out numerically *equal* – but FlatMerkle’s byte count buys one fewer sibling hash *and* one extra disclosed element, not a strictly smaller disclosure. Anyone relying on FlatMerkle proofs where the neighboring element’s privacy matters should account for this explicitly.

4 Worked examples: $N = 16$

All examples in this section are built from 16 real 32-byte pseudorandom elements, hashed with a toy but real compression function ($C(s, b) = \text{SHA256}(0x01 \parallel s \parallel b)$, 32-byte state and block – so $b = w$, using the identity checkpoint encoding $\text{pad}_w(x) = x$ of Definition 2.1 rather than the $b > w$ terminator-byte scheme), and every proof below was independently re-verified, including confirming that flipping a single bit anywhere in the target data, or in any one revealed value, is rejected.

Target index	Base pair proved (both)	Reveals	Proof size
0	$D[0], D[1]$	3	160 B
4	$D[4], D[5]$	3	160 B
7	$D[6], D[7]$	3	160 B
8	$D[8], D[9]$	3	160 B
11	$D[10], D[11]$	3	160 B
14	$D[14], D[15]$	3	160 B

Table 2: Six representative target indices out of the 16 possible, spread across the file; all 16 were checked and every one produces the same *constant* 160-byte proof (2×32 bytes of raw pair data plus 3×32 bytes of state reveals, $L - 1 = 3$), since proof size does not depend on which index is targeted (Section 3). A proof for target index a always discloses *both* raw elements of its base pair, $D[a_0]$ and $D[a_0+1]$ – proving index 7 necessarily also discloses element 6 – so the “Base pair proved” column names both elements the proof exposes, not just the requested index (Section 3.4 discusses this). For comparison, a same-shape Merkle proof over 16 leaves needs $\lceil \log_2 16 \rceil = 4$ sibling hashes (128 bytes) plus the leaf’s own preimage, and does *not* disclose a neighboring leaf.

4.1 Index 7 in detail

Figure 2 shows the full 16-element structured file with the proof path for index 7 highlighted: the base pair $D[6], D[7]$, and the checkpoint values CK_4 , CK_6 , and $\text{CK}_{[12,15]}$, in the order the proof

actually reveals them.

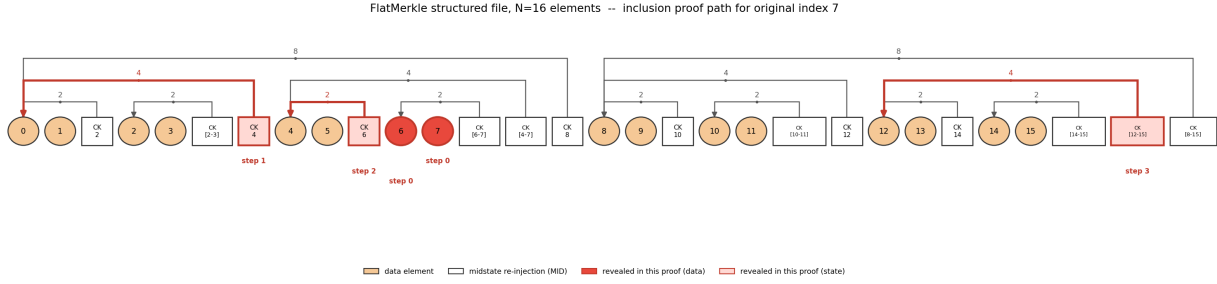


Figure 2: Structured file for $N = 16$, proof path for index 7 highlighted in red, numbered by the order each value is used.

A natural question when first looking at Figure 2 is how the verifier gets from “just after element 7” to the digest: no arc in the drawing runs directly from that point to the end. The reason is that the verifier does not walk the file’s own drawn jump-back arcs at all – it performs a straight, ordered fold of $\mathcal{C}(\cdot, \cdot)$ calls, each combining the previous result with either the next raw element or the next revealed checkpoint. Figure 3 makes this explicit.

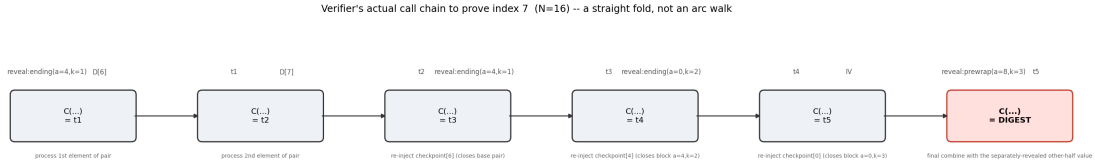


Figure 3: The verifier’s actual sequence of compression-function calls for index 7 – a straight-line fold ending at the digest, not a walk along the structured file’s own arcs.

Finally, Figure 4 shows the proof exactly as it is stored and transmitted: a flat, ordered sequence of items with no tree and no arcs, matching the compact wire format

$$(D[a_0], D[a_0+1], \text{reveals}[0..L-2]),$$

where $a_0 = a - (a \bmod 2)$ is derived by the verifier from whichever index it is asking about, never transmitted as part of the proof itself (Remark 3.4 explains why the proof must not carry its own copy of a_0).

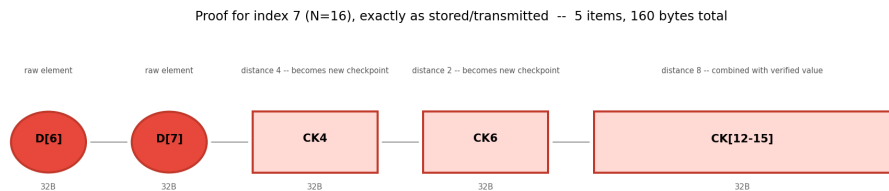


Figure 4: The index-7 proof exactly as stored: $D[6], D[7], CK_4, CK_6, CK_{[12,15]}$ – 5 items, 160 bytes, verified byte-for-byte against the real prover output.

4.2 Reading the checkpoint labels

CK_4 and CK_6 are each a single, unambiguous position: each is the closing value (ending) of a *left*-child block, immediately promoted to the checkpoint that starts the next block (Section 2). Using a single trailing index for this kind of value (e.g. naming it by the last element it covers) would be ambiguous, since different-sized blocks can share the same last-covered index; the span notation avoids that collision by construction.

$CK_{[12,15]}$ is different in kind, and worth reading carefully. The third reveal is $prewrap(8, 3)$ – the state right before $block(8, 3)$ ’s own closing $MID(8)$ folds in, *not* ending(8, 3) (which is the file’s own final raw chain state, and is never revealed to any verifier, since it only exists after the verifier’s own last fold). $prewrap(8, 3)$ happens to equal ending(12, 2) – $block(8, 3)$ ’s own right child’s ending, one level down – since that child closes immediately before $block(8, 3)$ ’s own $MID(8)$ is written, so it names the right child’s span, $CK_{[12,15]}$, *not* $block(8, 3)$ ’s own span $CK_{[8,15]}$ (which would name a value – ending(8, 3) – this proof never touches at all). The verifier does not receive $CK_{[12,15]}$ as a promoted checkpoint the way it does CK_4 or CK_6 : it folds this reveal together with $D[6], D[7]$ ’s already-verified result to *compute* $block(8, 3)$ ’s own closing value, which the verifier itself performs as the last fold in Figure 3.

5 Security

5.1 Assumptions

Assumption 5.1 (Free-start collision resistance). *It is computationally infeasible to find $(s_1, b_1) \neq (s_2, b_2) \in \{0, 1\}^w \times \{0, 1\}^b$ with $C(s_1, b_1) = C(s_2, b_2)$.*

This differs from the collision resistance ordinarily assumed of a hash function H , which fixes the starting state to IV and only requires hardness of finding two distinct *messages* colliding under that fixed start. Assumption 5.1 is strictly stronger: it must hold for *adversarially chosen* starting states, because a FlatMerkle verifier accepts several state values as untrusted input (the proof’s reveals) and only ever ties them to the truth via later equality checks against the published digest – never by re-deriving them from IV itself. This is exactly the property the construction needs, and we state it as a distinct, named assumption rather than assume it follows automatically from ordinary collision resistance of H ; in general it does not – free-start collision resistance of a Davies–Meyer-style compression step is a materially stronger, separately studied property [7].

Remark 5.2 (This assumption has failed for real, deployed compression functions). *The gap between Assumption 5.1 and ordinary collision resistance is not merely theoretical. Free-start collisions were found for MD5’s compression function in 1996 [7], years before any collision was known for MD5 itself, and for the full SHA-1 compression function in 2015 [32] – two years before the first ordinary SHA-1 collision (SHAttered, 2017). A FlatMerkle instance built on SHA-1 was therefore already insecure in 2015, at a time when SHA-1 was still considered safe for ordinary hashing by most practitioners. Assumption 5.1 should be checked against the free-start-collision literature for the specific compression function in use, not against that function’s ordinary collision-resistance track record. For SHA-256, this paper’s main instantiation, the best published result we are aware of is a practical semi-free-start collision for 39 of its 64 steps [17]; no free-start collision is known for the full compression function. For BLAKE2b, used in Section 11, Guo et al. report weaknesses of the compression function when its entire 16-word internal state is chosen [9]; a FlatMerkle verifier only lets an adversary choose the 8-word chaining value and the message block, and fixes the remaining inputs itself (see Section 11), but this margin is narrower than for SHA-256 and should be reassessed before deployment.*

Assumption 5.3 (Raw compression-function access). *The hash function H is built from C via the standard Merkle–Damgård iteration with a public finalization, and both prover and verifier have direct access to C itself (not merely to H).*

This rules out hash functions where the internal chaining step is not separately exposed or well-defined (e.g. most sponge constructions – see the discussion of adapting FlatMerkle to Keccak-256 in Section 13), and is satisfied by the Merkle–Damgård family generally, including SHA-256, SHA-1, and MD5. Satisfying Assumption 5.3 says nothing about whether Assumption 5.1 also holds for a given member of that family – see Remark 5.2.

Remark 5.4 (No domain separation between data and MID blocks). *A data element may be byte-identical to some MID block; nothing in the construction prevents it. This does not affect Theorems 5.8 or 5.9: which positions are data and which are checkpoints is fixed by (index, N) before any byte is examined, never inferred from content. A deployment wanting an extra margin can still tag MID blocks, provided the tag uses the $b - w$ bits that pad_b leaves free (256 bits for SHA-256) and application data is escaped to avoid the same pattern. Overwriting any of the checkpoint’s own w bits instead would break the injectivity of pad_b , which both theorems use. In a Bitcoin annex, any such convention must be known to anyone parsing the raw annex bytes, since consensus does not distinguish the two.*

5.2 Soundness

All three security arguments in this section – soundness, binding, and fraud-proof soundness (Section 5.5) – compare two executions of the same verifier recursion that reach the same output, and propagate that equality from outputs back to inputs. The one subtlety is the checkpoint argument: in a right branch, EVALBLOCK seeds the child with a *revealed* value, not with its own *ckpt*, so a naive “descend and compare” walk cannot assume a child’s checkpoint argument is already known to agree in both executions. We therefore state the induction once, with a hypothesis that quantifies over *unequal* checkpoint arguments, in a form general enough to cover the modified evaluator the fraud-proof verifier uses.

Fix $N = 2^L$, a target t , and a set V of t ’s own containing levels (Definition 5.5 below; $V = \emptyset$ for ordinary verification). EVALBLOCK $_V$ is EVALBLOCK except that, at each level $(a, k) \in V$, its own closing fold uses a supplied b -bit string $m_{(a,k)}$ in place of $\text{pad}_b(\text{ckpt})$; EVALBLOCK $_{\emptyset} = \text{EVALBLOCK}$.

Definition 5.5 (Anchored level). *For $1 \leq k \leq L - 1$ let $a_k = \lfloor t/2^k \rfloor 2^k$, so that $\text{block}(a_k, k)$ is t ’s containing block at level k . The checkpoint argument at level k is anchored (with respect to t and V), written $\text{bnd}(k)$, if*

$$\text{bnd}(1) = \text{true}, \quad \text{bnd}(k) = [(a_k, k) \notin V] \vee [\text{bit}_{k-1}(t) = 0 \wedge \text{bnd}(k-1)] \quad (2 \leq k \leq L-1),$$

and, additionally, $\text{bnd}(L-1) = \text{true}$ whenever $t < 2^{L-1}$. In words: a checkpoint is anchored if it survives in its own closing fold; or if the level takes the left branch, in which case the same value is passed down to the child and the question recurses; the base case is always anchored because $C(\text{ckpt}, d_0)$ consumes it whether or not its closing fold is substituted; and the top-level left root’s checkpoint is the public constant IV, fixed by the verifier itself. With $V = \emptyset$ every level is anchored.

Lemma 5.6 (Extraction). *Let $1 \leq k \leq L-1$ and consider two executions of EVALBLOCK $_V(a_k, k, t, \cdot)$ with checkpoint arguments $\text{ckpt}_1, \text{ckpt}_2$ – not assumed equal – and their own reveals, base elements and substituted strings. If both return the same value, then either a free-start collision in C can be extracted from the two executions with $O(k)$ compression calls, or all of the following hold: (i)*

both consumed the same base elements d_0, d_1 , the same left-branch sibling reveals, and the same substituted strings $m_{(a_j, j)}$ for every $(a_j, j) \in V$, $j \leq k$; (ii) for every $j \leq k$ with $\text{bnd}(j)$, the two executions' checkpoint arguments at level j coincide – in particular $\text{ckpt}_1 = \text{ckpt}_2$ if $\text{bnd}(k)$, and every right-branch reveal that seeds an anchored child coincides.

Proof. Induction on k . Every invocation ends with one compression call $\mathcal{C}(x_i, c_i)$, where $c_i = \text{pad}_b(\text{ckpt}_i)$ if $(a_k, k) \notin V$ and $c_i = m_{(a_k, k), i}$ otherwise. Equal outputs give either a collision or $x_1 = x_2$ and $c_1 = c_2$; in the first sub-case injectivity of pad_b (Definition 2.1) gives $\text{ckpt}_1 = \text{ckpt}_2$, in the second the substituted strings agree.

Base, $k = 1$: $x_i = \mathcal{C}(\mathcal{C}(\text{ckpt}_i, d_{0,i}), d_{1,i})$, so equal x gives a collision or $d_{1,1} = d_{1,2}$ and equal inner outputs, which in turn gives a collision or $\text{ckpt}_1 = \text{ckpt}_2$ and $d_{0,1} = d_{0,2}$ – regardless of V , matching $\text{bnd}(1)$.

Left branch ($\text{bit}_{k-1}(t) = 0$): $x_i = \mathcal{C}(s_i, \text{pad}_b(\ell_i))$ with s_i the sibling's revealed prewrap and $\ell_i = \text{EVALBLOCK}_V(a_k, k-1, t, \text{ckpt}_i)$. Equal x gives a collision or $s_1 = s_2$ and $\ell_1 = \ell_2$; the induction hypothesis applied to the child (whose checkpoint argument is the *same* ckpt_i) gives a collision or (i)–(ii) below level k , including $\text{ckpt}_1 = \text{ckpt}_2$ if $\text{bnd}(k-1)$. Together with the closing fold: ckpt is pinned iff $(a_k, k) \notin V$ or $\text{bnd}(k-1)$, which is $\text{bnd}(k)$.

Right branch ($\text{bit}_{k-1}(t) = 1$): $x_i = \text{EVALBLOCK}_V(a_k + 2^{k-1}, k-1, t, r_i)$, where r_i is a *revealed* value, not ckpt_i . The induction hypothesis gives a collision or (i)–(ii) below level k , including $r_1 = r_2$ if $\text{bnd}(k-1)$. Here ckpt_i appears *only* in the closing fold, so it is pinned iff $(a_k, k) \notin V$, which is again $\text{bnd}(k)$.

Each level inspects $O(1)$ calls, so the collision, if any, is found within $O(k)$ calls by recomputing both executions. $\square$

The only place the top-level split of `VERIFY` differs from a generic level is that the left root is seeded with `IV` in both executions – which is why Definition 5.5 marks it anchored unconditionally – and that the top-level sibling's closing fold, $\mathcal{C}(\text{sib_prewrap}, \text{pad}_b(\text{left_end}))$, is never substituted. Finalization is one further fixed public suffix of compression calls on blocks determined only by N (for SHA-256 and 64-byte items, exactly one padding block), so equal digests give either a collision or equal raw chain states. Combining these with Lemma 5.6 gives the following top-level statement, used by every theorem below.

Corollary 5.7 (Top-level extraction). *If two executions of `VERIFY` (with $V = \emptyset$), or of the fraud-proof fold of Section 5.5 (with the same t and V), both for the same N and target t , finalize to the same digest, then either a free-start collision in $\mathcal{C}$ can be extracted with $O(\log N)$ compression calls, or the two executions consumed the same d_0, d_1 , the same left-branch reveals, the same substituted strings, and agree on the checkpoint argument at every anchored level of t 's path. With $V = \emptyset$ this means the two proofs are identical.*

Theorem 5.8 (Soundness). *Under Assumptions 5.1 and 5.3, no adversary can produce $(\text{index}, \text{data}, \pi)$ accepted by `VERIFY` against a genuine digest Δ , where data disagrees with the true element $D[\text{index}]$ that Δ actually commits to, without also producing a free-start collision in $\mathcal{C}$.*

Proof. Let π^* be the genuine proof for *index*, built by `BUILDPROOF` against the real file; it is accepted (Section 5.3). Suppose an adversary's $(\text{index}, \text{data}, \pi)$ is also accepted, with $\text{data} \neq D[\text{index}]$. Both executions use the same *index* and N , hence the same target pair start a_0 and the same branch decisions, and both finalize to Δ . Corollary 5.7 with $V = \emptyset$ yields either a free-start collision or $\pi = \pi^*$; in the latter case `VERIFY` selects *data* from the same (d_0, d_1) by the same parity of *index*, so $\text{data} = D[\text{index}]$, a contradiction. $\square$

5.3 Completeness

Completeness is immediate from construction: for any index and any honestly built file, EVAL-BLOCK run against the reveals produced by BUILDPROOF recomputes, level by level, exactly the same sequence of compression calls the builder performed – by inspection of the two algorithms in Section 3, they are symmetric by design. This was also confirmed empirically: every index of every tested $N \in \{4, 8, 16, 32, 64\}$ verifies successfully against a genuinely built file.

5.4 Binding: what an accepting proof actually certifies

It might seem that Theorem 5.8’s technique establishes something stronger than it actually does: that for *any* byte string F' a publisher chooses to publish – canonically built or not – no accepting proof could name data other than the bytes genuinely present at position(*index*) in F' . That claim is false, and Remark 5.10 below gives a working forgery against it. What Theorem 5.8’s technique actually establishes, for an arbitrary (possibly non-canonical) F' , is a strictly weaker and correct property: *binding*, not *provenance*.

Theorem 5.9 (Binding). *Fix $N = 2^L$ ($L \geq 2$, the domain VERIFY itself checks – see Section 3), any published digest Δ' , and any index $\in [0, N)$. Under Assumption 5.1, there is at most one value data for which some proof π makes VERIFY accept $(N, \text{index}, \text{data}, \pi)$ against Δ' – regardless of whether Δ' commits to a canonically built file. (N is fixed throughout, not just Δ' and index, because VERIFY’s recursion structure – how many levels deep it goes, and where each branch falls – is itself a function of N ; two different N values would in general give two entirely different recursions over the same Δ' , so the claim is only meaningful, and only proved, for one N at a time.)*

Proof. Suppose two accepting triples $(\text{index}, \text{data}_1, \pi_1)$ and $(\text{index}, \text{data}_2, \pi_2)$, both against the same fixed N , verify against Δ' , with $\text{data}_1 \neq \text{data}_2$. Corollary 5.7 (with $V = \emptyset$) only ever compares two *accepting executions* against each other; it never assumes either was built by the honest BUILD procedure, only that both recursion structures are fixed by (index, N) – identical here, since N is fixed – and that both reach the same Δ' . It therefore yields either a free-start collision or $\pi_1 = \pi_2$, hence $\text{data}_1 = \text{data}_2$, contradicting the assumption. $\square$

Remark 5.10 (Binding does not imply the accepted value matches F' ’s own bytes – a working forgery). *Take a target index in the left half of the file. The verifier’s very last step is $\text{raw_state} = \mathcal{C}(r, \text{pad}_b(y))$, where y is the value the left-half recursion just computed and r is an unverified reveal – nothing above binds r to anything. In a canonically built file, the file’s own last 64 bytes are exactly $\text{MID}(2^{L-1}) = \text{pad}_b(\text{CK}[2^{L-1}]) = \text{pad}_b(\text{ending}(0, L-1))$, i.e. the true ending of the real left-half data – but nothing forces a publisher to write that value there. A publisher can instead: pick arbitrary fake $D[0], D[1]$ and arbitrary fake inner reveals; run the left-half recursion forward on this fake path to obtain some y_{fake} ; write $\text{pad}_b(y_{\text{fake}})$, instead of the true checkpoint, as the file’s actual last 64 bytes; and separately place whatever real data they like at position(0) elsewhere in the file. Reading off the true chain state r entering that last block – a public quantity, computable by anyone holding F' , no secret or collision required – the fake $(D[0], D[1], \text{reveals})$ now verifies against the real, published $\Delta' = H(F')$, even though the bytes genuinely sitting at position(0) in F' are something else entirely. This costs no free-start collision: it is exactly the deviation Theorem 5.9 does not rule out (both fake and genuine index = 0 claims cannot simultaneously be accepting, by binding – but binding says nothing about which one, if either, matches what a reader who simply parses F' would call “the data at position(0)”). Confirmed against the real SHA-256 reference implementation (`loglink_annex.py`): a file identical to a canonical one except for a single 32-byte checkpoint, whose SHA-256 is a completely ordinary digest, admits an accepting proof for forged index = 0 data.*

An accepting proof therefore certifies membership in *the dataset defined by the file’s own accepting proofs*, not “the bytes at position(i) of F ” as literally parsed off the byte string – the two coincide only when F' is canonically built to begin with, a property Theorem 5.9 says nothing about and which cannot be checked from Δ' alone in less than $O(N)$ work. Section 5.5 gives that missing check.

5.5 Fraud proofs: proving a published file is not canonically built

Binding rules out equivocation but, as Remark 5.10 shows, does not rule out a publisher shipping a file that is simply not canonically built in the first place. Recovering a soundness guarantee that actually protects a reader of F' needs a second, independent mechanism: a way for anyone holding the *whole* file – a *watcher* – to detect a non-canonical checkpoint and turn that one finding into a compact, $O(\log N)$ proof that a lightweight verifier, who never sees more than a handful of 32/64-byte values, can check without re-hashing F' .

Definition 5.11 (Violation). $\text{block}(a, k)$ is a violation in F' if the bytes physically present at $\text{MID}(a)$ ’s position disagree with $\text{pad}_b(\text{CK}[a])$, where $\text{CK}[a]$ is the real running compression state that genuinely enters position a when F' is hashed left to right – i.e. $\text{MID}(a)$ fails to honestly re-inject the checkpoint it claims to.

Algorithm 5.1 $\text{SCANFORVIOLATIONS}(F') - \text{watcher}, O(N)$, one linear pass

```

state := IV
for each item in  $F'$ , left to right:
    if item is  $D[a]$  and  $\text{CK}[a]$  not yet set:
         $\text{CK}[a] := \text{state}$  # entry checkpoint, captured once
    if item is  $\text{MID}(a)$ :
        expected :=  $\text{pad}_b(\text{CK}[a])$ 
        if item.bytes != expected:
            record violation ( $a, k, \text{item.bytes}$ )
        state :=  $C(\text{state}, \text{item.bytes})$  # always advance with the REAL bytes

```

Advancing with the file’s real bytes regardless of whether a violation was just found – rather than the value that *should* have been there – is what lets a single pass surface every independent violation in one scan, including several at once.

A single disputed level is not always enough, however: if two or more violations lie on the *same* root-to-leaf path – an ancestor block and one of its own descendants both corrupted – substituting real bytes at only one of them leaves the *other* corrupted level still folded in using the canonical (wrong) value, and the reconstruction fails to reach Δ' regardless of which one is chosen (Remark 5.14 below gives the mechanism). BUILD FRAUD PROOF and $\text{VERIFY FRAUD PROOF}$ therefore take a *set* of disputed levels, all lying on one target index’s own root-to-leaf path, and substitute at every one of them simultaneously.

BUILD FRAUD PROOF must *not* simply call the ordinary BUILD PROOF , which recomputes everything canonically from scratch (using $\text{pad}_b(\text{CK}[a])$ for every MID , by definition – the very thing that would be wrong for a non-canonical F'). Instead it reads directly off the snapshots SCANFORVIOLATIONS (Algorithm 5.1) already recorded while walking F' ’s own real bytes left to right:

Algorithm 5.2 $\text{EXTRACTPHYSICALPROOF}(F', t)$ – reads BuildProof-shaped values off a real, possibly non-canonical hash trace

```
# Same shape as BuildProof(t) -- L-1 reveals, two base-case data
# elements -- but every value below is read from the ACTUAL scan
# state ScanForViolations recorded while hashing F' left to right,
# never recomputed canonically:
d0 := the real bytes physically at D[a0]'s position in F', a0 = t - (t mod 2)
d1 := the real bytes physically at D[a0+1]'s position in F'
reveals := the sequence of scan.prewrap_of(.,.) / scan.ending_of(.,.)
          values BuildProof(t)'s own recursion would visit, each read
          from the scan's recorded running state at that exact
          position in F' -- not recomputed from a canonical checkpoint
return (d0, d1, reveals)
```

Algorithm 5.3 $\text{BUILDFRAUDPROOF}(F', t, V)$ – watcher, $O(\log N)$, for known violations V on target t 's path

```
# t: target index: any (a,k) in V must be one of t's OWN containing
# blocks, i.e. a = t with its low k bits cleared -- all of V lies on
# t's single root-to-leaf recursion path (self_positions(L,t)).
(d0, d1, reveals) := ExtractPhysicalProof(F', t)
actual_mid_bytes := { (a,k): the real bytes physically at MID(a,k)'s
                       position in F', for each (a,k) in V }
return (t, V, d0, d1, reveals, actual_mid_bytes)
```

Algorithm 5.4 $\text{VERIFYFRAUDPROOF}(N, \Delta', t, V, d_0, d_1, \text{reveals}, \text{actual_mid_bytes}) - O(\log N)$

```
# Structural checks, mirroring EVERY check ordinary Verify makes
# (Section 3.3) before it does anything else -- explicit, not implicit:
if N < 4 or N is not a power of two: return false
if t < 0 or t >= N: return false
if d0 or d1 is not exactly b bits wide: return false
if any V level (a,k) is not in self_positions(log2(N), t): return false
if actual_mid_bytes's key set != V: return false # exactly one entry
                                         # per disputed level,
                                         # no unreferenced ones
if any actual_mid_bytes[.] is not exactly b bits wide: return false

# Runs EvalBlock exactly as ordinary Verify does for target index t,
# using the real d0, d1 and reveals, except that at EVERY disputed level
# (a,k) in V the closing fold uses the real bytes claimed for that level
# instead of pad_b(ckpt). A mismatch is only COUNTED at an anchored level
# (bnd, "Anchored level" definition above): at an unanchored one, ckpt
# feeds no compression call
# once its closing fold is substituted, so it is a free variable and a
# comparison against it proves nothing.
A := { (a,k) in self_positions(log2(N), t) : bnd(k) holds for (t, V) }
mismatch := false
levels_reached := {}
function EvalBlock'(a', k', target, ckpt):
    ... exactly EvalBlock (Section 3.3), including its own "read next
    reveal, reject if the queue is empty" checks, except at its own
    closing step:
    if (a', k') in V:
        levels_reached.add((a', k'))
        closing_block := actual_mid_bytes[(a', k')] # not pad_b(ckpt)
        if (a', k') in A and actual_mid_bytes[(a', k')] != pad_b(ckpt):
            mismatch := true
    else:
        closing_block := pad_b(ckpt) # unchanged
    ... fold closing_block in exactly as EvalBlock does

raw_state := (top-level call of EvalBlock', mirroring Verify's own split)
if raw_state is malformed (reveal queue underflowed): return false
if reveals is not empty after the top-level call: return false # leftover
                                         # reveals
if levels_reached != V: return false # some claimed level was never
                                         # actually on t's own path
if finalize(raw_state, N) != Delta': return false # binding check
if mismatch != true: return false # a truthful non-accusation is
                                         # not a fraud proof

return true
```

Theorem 5.12 (Fraud-proof soundness). *Under Assumption 5.1, if VERIFYFRAUDPROOF accepts against a genuine digest Δ' , then the file Δ' commits to is not canonically built – some block (a, k) somewhere is a violation – except by also producing a free-start collision in $\mathcal{C}$. It does not, on its own, certify that every individual level in V is itself a violation (Remark 5.18).*

Proof. Suppose the file is canonically built – every $\text{MID}(a)$ equals $\text{pad}_b(\text{CK}[a])$ – yet VERIFYFRAUDPROOF accepts $(N, \Delta', t, V, d_0, d_1, \text{reveals}, \text{actual_mid_bytes})$. Compare this *submitted* execution with a *genuine* one for the same t and the same V : the honest proof $\text{BUILDPROOF}(t)$, with substituted strings equal to the file’s actual bytes at each $(a, k) \in V$, which by canonicity are $\text{pad}_b(\text{CK}[a])$. The genuine execution is therefore an ordinary EVALBLOCK run in disguise, reaches Δ' , and its checkpoint argument at every level (a_k, k) of t ’s path is $\text{CK}[a_k]$. Both executions finalize to Δ' , so Corollary 5.7 gives either a free-start collision, or (a) $\text{actual_mid_bytes}[(a, k)] = \text{pad}_b(\text{CK}[a])$ for every $(a, k) \in V$ and (b) at every anchored level the submitted checkpoint argument equals $\text{CK}[a_k]$. Acceptance requires a counted mismatch, i.e. some *anchored* $(a, k) \in V$ with $\text{actual_mid_bytes}[(a, k)] \neq \text{pad}_b(\text{ckpt})$; by (a) and (b) both sides equal $\text{pad}_b(\text{CK}[a])$ – a contradiction. $\square$

Remark 5.13 (Why the anchoring condition is necessary). *Counting mismatches at unanchored levels would make the theorem false. For $N = 8$, $t = 6$, $V = \{(4, 2)\}$ on a canonical file, the level’s checkpoint is the first reveal, used only by the substituted closing fold (the right child is seeded with a separate reveal). Replacing that reveal by any other value leaves the fold to Δ' unchanged but forces a mismatch – an accepted accusation against a canonical file, with no collision. Substitution at unanchored levels is still permitted, since it is needed to reach Δ' when an ancestor on t ’s path is genuinely corrupted; only the mismatch there is not counted.*

Remark 5.14 (General nested/chained violations, not just one point). *Substituting only one of several violations chained along a single path does not work, in either direction. Suppose $\text{block}(a, k)$ (deeper) and $\text{block}(a, k')$ (an ancestor, $k' > k$) are both violations. A fraud proof disputing only (a, k) folds the ancestor’s closing step with the canonical $\text{pad}_b(\text{CK}[a])$ – but the real file’s ancestor $\text{MID}(a)$ is genuinely corrupted, so the reconstructed path departs from the file’s actual hash chain at that ancestor step and, absent a collision, will not reach Δ' . Symmetric reasoning shows a fraud proof disputing only the ancestor (a, k') fails too, since it folds the deeper, also-corrupted level canonically. Substituting the real physical bytes at every violated level on the path simultaneously avoids this: the reconstruction then matches the file’s true hash chain at every step along the path, violated or not, so it reaches Δ' regardless of how many violations are chained together. It might seem that proving only the deepest violation on a chained path should always suffice, on the reasoning that nothing lies below it to be corrupted – but that reasoning only accounts for corruption below the disputed level, not above it, and is false whenever an ancestor higher up the same path is also corrupted, exactly as the paragraph above shows – confirmed directly against real SHA-256 (`simulate_fraud_proofs.py`, Cases F and G): a single-substitution proof at either the deepest or an ancestor violation alone fails to verify for a deliberately nested pair (and triple), while the multi-level substitution above succeeds. Violations on independent (non-nested) paths are unaffected by any of this and can each be proven with their own (possibly single-level) proof.*

Remark 5.15 (Sibling substitution is unsound). *A proof anchored at target t can only substitute real bytes at one of t ’s own containing blocks – never at a sibling block visited along the way (whose prewrap is revealed but whose own closing step EVALBLOCK still folds in canonically, exactly as an ordinary proof does). One might be tempted to widen V to include such sibling positions too, so that a violation sitting there could also be disputed. That does not work: the sibling’s canonical closing step is the only place the checkpoint value threaded down from t ’s own base data, d_0 and d_1 , is ever used again once computed, so substituting a claimed value there instead severs d_0 , d_1 , and every reveal on t ’s path from Δ' entirely – the fold-to-digest check then passes regardless of what base data is handed to VERIFYFRAUDPROOF , letting a forger fabricate an accepting proof against a fully canonical file with no collision needed at all. (This is exactly the kind of unsoundness a tamper-injection test catches directly: flip a bit of the submitted d_0 in an otherwise-genuine proof*

built this way, and verification is unaffected – which is only possible if d_0 was never actually bound to Δ' in the first place.) So substitution must stay confined to t 's own containing blocks, and a violation sitting on some sibling's own closing step, relative to a particular t , is simply not provable via that t . This leaves a genuine question: does some target always work, for any non-canonical file, no matter how its violations are scattered? Theorem 5.17 below answers yes.

Lemma 5.16 (Dyadic containment). *For any index t and any block $\text{block}(a, k)$ in the recursion with $a \leq t$, exactly one of the following holds: either $\text{block}(a, k)$ contains t (i.e. $a \leq t < a + 2^k$), or it lies entirely at or before t (i.e. $a + 2^k \leq t$). These are the only two possibilities once $a \leq t$, immediately from $\text{block}(a, k)$ being the contiguous index range $[a, a + 2^k)$: either t falls inside that range, or the range has already ended by the time t is reached.*

Theorem 5.17 (Fraud-proof completeness). *For every non-canonical FlatMerkle file F' of valid size N , if $\text{SCANFORVIOLATIONS}(F')$ finds at least one violation, then choosing $t =$ the a of the violation with the largest such value, and $V =$ every violation on t 's own containment path (i.e. $V = \{(a, k) : (a, k) \text{ is one of } t\text{'s own containing blocks and a violation}\}$), makes $\text{BUILD FRAUD PROOF}(F', t, V)$ produce a proof $\text{VERIFY FRAUD PROOF}$ accepts.*

Proof. Let (a', k') be any violation the scan found; by the choice of t (the maximum violation start), $a' \leq t$. By Lemma 5.16, either (i) $\text{block}(a', k')$ contains t , or (ii) it lies entirely at or before t .

In case (i): since a' is, by construction of the recursion, the unique multiple of $2^{k'}$ whose block contains t at level k' , (a', k') is exactly one of t 's own containing blocks – i.e. $(a', k') \in V$ – so it is substituted explicitly with its real physical bytes.

In case (ii): $\text{block}(a', k')$'s entire index range lies at or before t . Every index $\leq t$ falls either inside t 's own base pair, or inside a sibling range EVALBLOCK visits going *right* somewhere along t 's own recursion path (a right branch at level j has $t \geq c + h$ for that level's midpoint $c + h$, and its sibling is the range $[c, c + h)$, entirely $< c + h \leq t$) – *never* inside a sibling range visited going *left* (a left branch has $t < c + h$, and its sibling is $[c + h, c + 2h)$, entirely $> t$, the wrong side for case (ii)). Every right-branch sibling's value is read as an opaque, real ending straight from F' 's own scan trace ($\text{EXTRACT PHYSICAL PROOF}$ never recomputes it canonically), so it already reflects $\text{block}(a', k')$'s real, possibly-corrupted bytes automatically – no substitution is needed, or possible, to fix what is already correct by construction.

So every violation the scan found is accounted for by $\text{BUILD FRAUD PROOF}(F', t, V)$: either substituted explicitly (case (i)) or already reflected in a genuine reveal (case (ii)) – none can fall inside a sibling visited going left, the one place substitution is unavailable (Remark 5.15). Since $\text{EXTRACT PHYSICAL PROOF}(F', t)$ builds d_0, d_1 , and every reveal directly from F' 's real physical bytes (never a canonical reconstruction), the fold $\text{VERIFY FRAUD PROOF}$ performs – substituting real bytes at every $(a, k) \in V$ and folding in F' 's genuine reveals everywhere else – exactly reproduces F' 's true physical hash chain end to end, which by definition reaches F' 's real digest Δ' : the binding check passes. Finally, t is the start of a genuine violation $\text{block}(t, k)$, so $(t, k) \in V$, and the checkpoint argument the fold carries into that level is the real running state entering position t , i.e. $\text{CK}[t]$, so $\text{actual_mid_bytes}[(t, k)] \neq \text{pad}_b(\text{CK}[t])$. This level is anchored: t is a multiple of 2^k , so $\text{bit}_{j-1}(t) = 0$ for every $j \leq k$, and Definition 5.5 gives $\text{bnd}(k)$ through the left-branch clause down to $\text{bnd}(1)$, whatever else is in V . The mismatch is therefore counted, and $\text{VERIFY FRAUD PROOF}$ accepts. $\square$

Combined with Theorem 5.12, this gives full completeness: for *any* non-canonical file, an accepting $O(\log N)$ fraud proof always exists, even though not every individual violation is provable via every target.

Remark 5.18 (Fraud-proof soundness proves non-canonicity, not exact localization). *It might seem that Theorem 5.12 should establish something stronger: that an accepting proof certifies $\text{block}(a, k)$ specifically, for $(a, k) \in V$, is a violation. That stronger claim overreaches: Binding (Theorem 5.9) pins down the outermost accepted (index, data) pair for an ordinary, unmodified EVALBLOCK execution, but EVALBLOCK' is a modified evaluator with permitted deviation points at V – and nothing prevents an adversary from supplying self-consistent but entirely fabricated data and reveals below some level, engineering a spurious mismatch at an innocent level while the genuine corruption (needed to make the fabricated path still land on the real Δ') sits somewhere else in the file entirely, disguised as an honestly-revealed value. Concretely, for $N = 8$: a publisher picks an arbitrary 64-byte m , computes $\ell = \mathcal{C}(\mathcal{C}(\mathcal{C}(\text{IV}, D[0]), D[1]), m)$, writes $\text{pad}_b(\ell)$ as $\text{block}(2, 1)$'s closing MID, and builds the rest of the file consistently with that choice. The only violation in the file is $\text{block}(2, 1)$, yet a fraud proof for $t = 0$ with $V = \{(0, 1)\}$ and claimed bytes m is accepted (checked against the reference implementation): level $(0, 1)$ is anchored and its mismatch is real with respect to the submitted fold, but the physical bytes at $\text{MID}(0, 1)$ are canonical. The proof of Theorem 5.12 does not apply here, because for a non-canonical file there is in general no genuine execution along t 's path that reaches Δ' to compare against. What that proof actually establishes is the disjunctive, but still fully actionable, guarantee: at least one level somewhere in the walk disagrees between the genuine and submitted executions, and since the genuine file must actually be non-canonical for such a submission to reach Δ' without a collision at all, an accepting fraud proof is sufficient grounds to reject F' – just not, by itself, sufficient grounds to conclude the specific levels named in V are where the true corruption lives. Applications that only need “is this file trustworthy” (e.g. the Bitcoin-annex use case below) are unaffected by this distinction; applications that need to pinpoint and, say, penalize whoever introduced a specific corrupted byte range would need a strictly stronger construction than the one given here.*

Cost and trust model. Finding a violation costs the watcher $O(N)$, one linear pass – unavoidable, since nothing short of reading the whole file can distinguish a canonical build from a non-canonical one. Building and verifying the resulting proof costs $O(\log N)$: exactly the same shape as an ordinary single-element proof (Section 3), plus one extra 64-byte value per disputed level, and at most $L - 1$ levels can ever lie on one path, so the proof stays $O(\log N)$ even when every level on a path is disputed at once. This turns FlatMerkle's soundness story, for a file whose canonicity is not otherwise attested to, into an *optimistic* one: a reader may treat F' as canonical by default, and that assumption is enforceable – not merely hoped for – as long as at least one honest watcher holds the full file within some challenge window and is willing to publish an $O(\log N)$ fraud proof, structurally the same trust assumption optimistic rollups make about their own fraud-proof games [12]. We implemented and tested SCANFORVIOLATIONS, BUILDFRAUDPROOF, and VERIFYFRAUDPROOF against real SHA-256, including honest files with zero false positives, the exact forgery of Remark 5.10, single violations at every level, up to four independent simultaneous violations at $N = 1024$ (21 compression calls to verify a fraud proof there, against 2046 for a full rehash), deliberately nested chains of two and three violations on one path – specifically targeting Remark 5.14 – (7 and 9 compression calls respectively, against 14 and 30 for a full rehash), and – specifically targeting Theorem 5.17 – scattered violations including one on a sibling block's own closing step relative to an otherwise-natural target, confirming that target's own proof is correctly blocked while the max- a target's proof succeeds unaffected (6 and 8 compression calls at $N = 8$ and $N = 16$ respectively); every tamper variant of a genuine fraud proof – a flipped base-data byte, a flipped reveal, a flipped disputed-level byte string, a claim disguised as non-fraudulent, or a bogus off-path disputed level – was correctly rejected, including against the completeness-targeting proofs

specifically (ruling out the sibling-substitution unsoundness Remark 5.15 describes). Two further cases target the anchoring rule (Remark 5.13): every disputed-level set V on every target’s path in canonical files at $N \in \{8, 16, 32\}$, with one or two reveals replaced by random values (5,544 accusations, 308 configurations containing an unanchored disputed level), is rejected; and 300 random non-canonical files with up to five violations each, nested or not, at $N \in \{4, \dots, 64\}$, all yield an accepting fraud proof at the max- a target, confirming Theorem 5.17 under the anchored rule.

This is the property that makes the Bitcoin-annex application (Section 9) deployable: not that a malicious publisher’s non-canonical annex is harmless (Remark 5.10 shows it is not), but that binding prevents equivocation once a file is accepted as canonical, and the fraud-proof mechanism above gives any interested party a concrete, $O(\log N)$ way to challenge an annex that is not.

6 Optimal element size

6.1 A flexible toy primitive

Fix a total payload size P bytes and a state width W bytes, and suppose the element size D can be freely chosen (i.e. the compression function’s block width can be made to match D exactly, as in our toy 32-byte primitive, Section 4). D is not a free real number, however: MID(a) re-injects the entire W -byte checkpoint into one D -byte block via an injective encoding (Definition 2.1), and no injective encoding can embed a wider value into a narrower one, so

$$D \geq W \tag{10}$$

is a hard feasibility floor, not a property of any particular padding scheme – it holds regardless of whether the encoding used has a spare terminator byte (the $b > w$ scheme) or none (the $b = w$ identity encoding). The construction also requires $N = 2^L$ for an integer L (Section 2; non-power-of-two N is not yet handled – Section 13.3), so the genuine free variable is the integer L , with $D = P/2^L$ determined by it, not D itself. The problem is therefore the constrained optimization

$$\min_{L \in \mathbb{Z}} \text{size}(L) = 2D + (L - 1)W = \frac{2P}{2^L} + (L - 1)W \quad \text{subject to } D = P/2^L \geq W, \tag{11}$$

two raw elements plus $L - 1$ state reveals, minimized only over the integer L values whose resulting D actually satisfies the floor.

Ignoring the constraint first, to see where it binds: treating L as continuous and minimizing over (continuous) $D = P/2^L$ gives the stationarity condition

$$\frac{d}{dD} \text{size}(D) = 2 - \frac{W}{D \ln 2} = 0 \implies D^* = \frac{W}{2 \ln 2}. \tag{12}$$

For $W = 32$ bytes, $D^* \approx 23.08$ bytes – strictly *less* than $W = 32$, i.e. the unconstrained stationary point lies entirely outside the feasible region $D \geq W$. Since $\text{size}(D)$ is convex and its derivative is already positive at $D = W$ (the constraint boundary), the constrained optimum is not this stationary point at all, but simply the *smallest feasible* D : $D = W$ exactly, using the identity encoding. Every power of two below W is infeasible outright – not merely suboptimal – because the checkpoint cannot physically fit.

L	$N = 2^L$	$D = P/N$ (bytes)	Proof size (bytes)	Feasible?
21	2,097,152	512.00	1664.00	yes
22	4,194,304	256.00	1184.00	yes
23	8,388,608	128.00	960.00	yes
24	16,777,216	64.00	864.00	yes
25	33,554,432	32.00	832.00	yes ($D = W$)
26	67,108,864	16.00	(832.00)	no : $D < W$
27	134,217,728	8.00	(848.00)	no : $D < W$
28	268,435,456	4.00	(872.00)	no : $D < W$

Table 3: Sweep at fixed payload $P = 2^{30}$ bytes (1024 MiB), $W = 32$ bytes. Rows with $D < W = 32$ (bracketed sizes) are not achievable FlatMerkle configurations at all – a 32-byte checkpoint cannot be injectively embedded in a smaller block, under any encoding – not merely worse than the optimum; their $\text{size}(L)$ values are shown only to illustrate that the unconstrained continuous formula would (wrongly) suggest they help. The true, unique discrete minimum is $L = 25$ ($D = 32 = W$, using the identity checkpoint encoding of Definition 2.1), 832.00 B – not a tie with any smaller D , and not the non-power-of-two, sub- W point $D \approx 23.08$ the unconstrained continuous relaxation alone would suggest. A deployment that instead needs the $b > w$ terminator-byte encoding (e.g. to keep one spare byte for a domain-separation tag, Remark 5.4) faces the stricter floor $D \geq W + 1 = 33$ bytes; since D must itself be a power of two here ($D = P/2^L$ with both P and N powers of two), the smallest such D is 64, i.e. $L = 24$, 864.00 B.

6.2 Real SHA-256: the block size is fixed at 64 bytes

The toy analysis assumes D can be chosen freely, but SHA-256’s block width is fixed at $b = 512$ bits (64 bytes): every item of the structured file, data or MID, occupies exactly one 64-byte block, and Section 2 does not pack several application records into one element. With a payload of P bytes split into 64-byte elements, $N = P/64$, and since each proof reveal (a checkpoint) carries only its 32-byte state, a single-element proof costs

$$\text{size}(P) = 2 \times 64 + (\log_2(P/64) - 1) \times 32 = 96 + 32 \log_2(P/64) \text{ bytes.} \quad (13)$$

This was verified directly against the real-SHA-256 implementation: for $N = 32$ real 64-byte elements ($P = 2048$ bytes), the formula predicts $96 + 32 \log_2(32) = 96 + 160 = 256$ bytes, which matches the measured proof size exactly.

Unlike the toy case, where the two coincide at $D = W$, here FlatMerkle’s proof is *larger* than a Merkle proof over the same 64-byte elements: a Merkle proof carries one 64-byte leaf and $\log_2 N$ 32-byte siblings, $64 + 32 \log_2(P/64)$ bytes, exactly 32 bytes less than FlatMerkle’s, and it discloses no neighbouring element (Table 4). The difference is the second raw element of FlatMerkle’s base pair (Section 3.4).

7 Batched multi-element inclusion proofs

When proving inclusion of $m > 1$ target indices at once, independent single-element proofs waste space: two targets whose recursion paths share a common ancestor block need only reveal that shared checkpoint once, not once per proof – exactly the same idea used by Merkle multiproofs.

N	$P = 64N$ (bytes)	FlatMerkle proof (bytes)	Merkle proof (bytes)
4	256	160	128
8	512	192	160
16	1024	224	192
32	2048	256	224
64	4096	288	256

Table 4: Real-SHA-256 FlatMerkle proof sizes, measured directly against the `loglink_annex.py` implementation, matching $96 + 32 \log_2(P/64)$ exactly at every scale tested, next to the size of a Merkle proof over the same 64-byte elements, $64 + 32 \log_2 N$ (computed).

7.1 Construction

BUILDMULTIPROOF runs the same recursive traversal as BUILDPROOF (Section 3), but over the set of all target pairs simultaneously: at each level, a block is only recursed into if it contains at least one target; a sibling block is only revealed (as ending or prewrap, exactly as before) if it contains *no* target – if both children contain targets, both are walked and no reveal is needed at that level at all. The verifier’s EVALMULTIBLOCK mirrors this exactly, computing each shared value once for every target whose path passes through it. Algorithms 7.1–7.3 give all three procedures; they correspond line by line to `build_multiproof` and `verify_multiproof` in the reference implementation. Throughout, P is the set of distinct target *pair starts*, and for a set S of pair starts inside `block(a, k)` we write $S_L = \{s \in S : s < a + 2^{k-1}\}$ and $S_R = S \setminus S_L$.

Algorithm 7.1 BUILDMULTIPROOF(T) – prover

```

P := { t - (t mod 2) : t in T } # distinct pair starts
function MRecurse(a, k, S): # S: non-empty, pair starts in block(a,k)
    if k = 1: return # base pair always sent raw
    h := 2^(k-1)
    if S_L and S_R both non-empty:
        MRecurse(a, k-1, S_L)
        MRecurse(a+h, k-1, S_R) # no reveal at this level
    else if S_R is empty:
        MRecurse(a, k-1, S_L)
        reveal prewrap(a+h, k-1) # right sibling, as in BuildProof
    else:
        reveal ending(a, k-1) # left sibling, as in BuildProof
        MRecurse(a+h, k-1, S_R)

h_top := 2^(L-1); P_L := {s in P : s < h_top}; P_R := P \ P_L
if P_L and P_R both non-empty:
    MRecurse(0, L-1, P_L); MRecurse(h_top, L-1, P_R)
else if P_R is empty:
    MRecurse(0, L-1, P_L); reveal prewrap(h_top, L-1)
else:
    reveal ending(0, L-1); MRecurse(h_top, L-1, P_R)
return ({ s -> (D[s], D[s+1]) : s in P }, reveals in the order emitted)

```

Algorithm 7.2 EVALMULTIBLOCK($a, k, S, ckpt$) – verifier

```
function EvalMultiBlock(a, k, S, ckpt):
  if k = 1: # here S = {a}
    (d0, d1) := pairs[a]
    return C(C(ckpt, d0), d1), pad_b(ckpt))
  h := 2^(k-1)
  if S_L and S_R both non-empty:
    left_end := EvalMultiBlock(a, k-1, S_L, ckpt)
    right_end := EvalMultiBlock(a+h, k-1, S_R, left_end)
    # right child seeded with the COMPUTED left_end
  else if S_R is empty:
    left_end := EvalMultiBlock(a, k-1, S_L, ckpt)
    sib_prewrap := read next reveal # reject if queue empty
    right_end := C(sib_prewrap, pad_b(left_end))
  else:
    left_end := read next reveal # reject if queue empty
    right_end := EvalMultiBlock(a+h, k-1, S_R, left_end)
  return C(right_end, pad_b(ckpt))
```

Algorithm 7.3 VERIFYMULTIPROOF($N, \Delta, \{data_t\}_{t \in T}, \pi$) – top-level verifier

```
if N < 4 or N is not a power of two: return false
if T is empty or any t in T is outside [0, N]: return false
(pairs, reveals) := pi # reveals: shared FIFO queue
P := { t - (t mod 2) : t in T } # derived from the verifier's OWN T,
    # never from pi (the a0 rule, Sec. 3)
if keys(pairs) != P: return false # exactly one pair per pair start
for t in T:
  (d0, d1) := pairs[t - (t mod 2)]
  if t is even: expected := d0 else: expected := d1
  if data_t != expected: return false
h_top := 2^(L-1); P_L := {s in P : s < h_top}; P_R := P \ P_L
if P_L and P_R both non-empty:
  y := EvalMultiBlock(0, L-1, P_L, IV)
  raw_state := EvalMultiBlock(h_top, L-1, P_R, y)
else if P_R is empty:
  y := EvalMultiBlock(0, L-1, P_L, IV)
  sib_prewrap := read next reveal # reject if queue empty
  raw_state := C(sib_prewrap, pad_b(y))
else:
  r := read next reveal # reject if queue empty
  raw_state := EvalMultiBlock(h_top, L-1, P_R, r)
if reveals is not empty: return false # leftover reveals: malformed
return finalize(raw_state, N) = Delta
```

Deriving P from the verifier's own T , and requiring the proof's pair set to equal it exactly, is the multi-target form of the rule Remark 3.4 establishes for single proofs: the recursion never looks at a pair's low bit, so a verifier that took pair starts, or the assignment of indices to pair elements, from the proof could be fooled by the same relabeling.

7.2 Soundness

EVALMULTIBLOCK (Algorithm 7.2) is a new verifier composition, not a parameter change to EVALBLOCK, and this paper already flags one soundness pitfall that a superficially similar-looking verifier design invites (Remark 3.5), so it needs its own argument rather than an appeal to Section 5 by analogy alone.

Lemma 7.1 (Multiproof soundness reduces to single-element soundness). *Fix a genuine digest Δ committing to a canonically built file (the same scope Theorem 5.8 requires). Under Assumptions 5.1 and 5.3, no adversary can produce a target set T , values $\{data_t\}_{t \in T}$, and a multiproof π accepted by VERIFYMULTIPROOF against Δ , with $data_t \neq D[t]$ for some $t \in T$, without also producing a free-start collision in $\mathcal{C}$.*

Proof. Fix any $t \in T$ with $data_t \neq D[t]$, and let $a_0 = t - (t \bmod 2)$. Run VERIFYMULTIPROOF on π and record every intermediate value. From this record we extract an ordinary single-element proof $\pi_t = (d_0, d_1, reveals_t)$ for index t : $(d_0, d_1) = pairs[a_0]$, and $reveals_t$ is built by walking t 's own root-to-leaf path in BUILDPROOF order, taking at each level (the top-level split included) the value that EVALBLOCK would read there:

- If t 's sibling at that level contains no other target pair, π revealed it explicitly, at the same point of the same fold; copy that reveal.
- If the sibling is the *left* child and contains targets, EVALMULTIBLOCK computed its value $left_end$ recursively and seeded t 's child with it; use that computed value as the reveal EVALBLOCK reads in its right branch.
- If the sibling is the *right* child and contains targets, EVALMULTIBLOCK computed $right_end = EVALMULTIBLOCK(a+h, k-1, S_R, left_end)$, whose last step is the call $\mathcal{C}(x, pad_b(left_end))$; use the pre-closing state x as the *sib_prewrap* reveal. EVALBLOCK then performs the same call, $\mathcal{C}(x, pad_b(left_end))$, with the same inputs.

By induction on block size along t 's path, EVALBLOCK on π_t performs exactly the compression calls EVALMULTIBLOCK performs along that path, with identical inputs: the base case uses the same $(ckpt, d_0, d_1)$, and each level up combines the same target-side value with the same sibling-side value through the same Equations (7)–(9). Both therefore reach the same raw chain state and the same Δ , and VERIFY selects $data_t$ from (d_0, d_1) by the same parity of t that VERIFYMULTIPROOF already checked. So VERIFY accepts $(t, data_t, \pi_t)$ with $data_t \neq D[t]$, and Theorem 5.8 supplies a free-start collision. The extraction costs $O(\log N)$ beyond the multiproof verification itself. $\square$

Remark 7.2 (Scope: this is soundness against physical data, not binding). *Lemma 7.1's conclusion is tied to the true, physical $D[t]$, exactly as Theorem 5.8's is – and, exactly as with the single-element case, this does not extend to a possibly non-canonical file. The extraction above still goes through unchanged for an arbitrary (possibly non-canonical) Δ' : it only relies on π and the honestly-extracted π_t reaching the same Δ' , not on Δ' committing to a canonical file. What that weaker setting yields is the multi-target analog of Theorem 5.9 (binding, not soundness against physical data): for a fixed target set T and digest Δ' , at most one tuple $\{data_t\}_{t \in T}$ is accepted by VERIFYMULTIPROOF, regardless of whether Δ' commits to a canonical file – since two distinct accepted tuples would disagree at some $t \in T$, extract to two distinct accepted single-element proofs for that t , and Theorem 5.9 already rules that out. It does not follow that an accepted $data_t$ equals the bytes physically present at position(t) in a non-canonical Δ' – Remark 5.10 is a standing counterexample to exactly that stronger claim, and nothing about batching several targets together repairs it.*

This reduction is why EVALMULTIBLOCK’s two-case rule (reveal directly vs. recurse) needs to be exactly the single-proof rule applied independently to each shared block, with no shortcut that lets a multi-target block reuse a value *without* recomputing or revealing it per level – precisely the discipline the insecure alternative of Remark 3.5 violates in the single-target case, and the same tamper-rejection discipline was applied to the multiproof implementation as well – tampering the target data, the claimed data, one shared or individual reveal per trial, relabeling a claim to the other index of its pair, and adding an extra unclaimed pair, across many randomized target sets (Section 1), with every tamper correctly rejected.

7.3 Measured savings

We measured average multiproof size (over 12 random trials per row) against m independent single-element proofs, for m uniformly random distinct target indices, at several scales.

N	m	Independent (B)	Multiproof (B)	Saved
2^{10}	2	704.0	597.3	15.2%
2^{10}	8	2816.0	1845.3	34.5%
2^{10}	32	11264.0	5306.7	52.9%
2^{10}	64	22528.0	8645.3	61.6%
2^{14}	2	960.0	874.7	8.9%
2^{14}	8	3840.0	2816.0	26.7%
2^{14}	32	15360.0	9434.7	38.6%
2^{14}	64	30720.0	16634.7	45.9%
2^{18}	2	1216.0	1120.0	7.9%
2^{18}	8	4864.0	3896.0	19.9%
2^{18}	32	19456.0	13509.3	30.6%
2^{18}	64	38912.0	24965.3	35.8%
2^{20}	2	1344.0	1232.0	8.3%
2^{20}	8	5376.0	4418.7	17.8%
2^{20}	32	21504.0	15450.7	28.1%
2^{20}	64	43008.0	28853.3	32.9%

Table 5: Multiproof space savings for uniformly random target sets, $D = W = 32$ bytes. Savings grow with m (more sharing opportunities) and shrink with N (a fixed m becomes a sparser sample of a larger tree, so paths share fewer ancestors), ranging from 7.9% to 61.6% across the scales tested.

8 Construction speed vs. a Merkle tree

Proof size is only half the comparison against Merkle trees; the other is build time. We measured this directly, using the identical hand-rolled SHA-256 compression function of the reference implementation (`sha256_core.py`) for every hash performed on every side – same language, same implementation, no library shortcut on one side and not the other – across three constructions:

1. **FlatMerkle**, as described in this paper: $2N - 2$ items plus one finalization, $2N - 1$ compression calls total.

2. **Merkle tree, same convention:** one compression call per node (leaf = $\mathcal{C}(\text{IV}, \text{element})$, internal = $\mathcal{C}(\text{IV}, \text{left} \parallel \text{right})$), no per-node padding – an equally-optimized tree, isolating the tree-vs-chain question with no artificial handicap on either side. N leaves + $(N - 1)$ internal nodes = $2N - 1$ calls.
3. **Merkle tree, standard:** every node is its own independently padded, finalized SHA-256 call – i.e. what many real Merkle tree implementations that hash with a single, ordinary SHA-256 per node actually do. A 64-byte node input needs 2 compression calls once padded and length-encoded, so this costs $4N - 2$ calls total. (Bitcoin’s own transaction Merkle tree is not an instance of this convention: it combines each pair of nodes with *double*-SHA256 – SHA256 applied twice, i.e. two independent finalized hashes chained together – which this benchmark does not measure and is more expensive still than either convention above.)

Method	N	Seconds	Compress calls	Calls/element
FlatMerkle	8,192	1.37	16,383	2.000
Merkle, same convention	8,192	1.36	16,383	2.000
Merkle, standard	8,192	2.73	32,766	4.000
FlatMerkle	65,536	11.03	131,071	2.000
Merkle, same convention	65,536	10.78	131,071	2.000
Merkle, standard	65,536	21.68	262,142	4.000

Table 6: Construction speed, real measurements. Calls/element ratios are the exact rationals $2 - 1/N$ (FlatMerkle and the same-convention tree) and $4 - 2/N$ (the standard tree), which already round to 2.000/4.000 to three decimals at these scales ($16383/8192 = 1.99988\dots$) – not an empirical coincidence, but not literally exact integers either, only in the $N \rightarrow \infty$ limit.

Two findings, both borne out at both measured scales:

1. **Against an equally-optimized Merkle tree, FlatMerkle is not faster to build** – measured construction time is within 1–2% either way (ratio 0.98–0.99 $\times$) at both scales tested, well inside the noise expected from a single run each; we do not have repeated-trial variance data to call this a rigorously established statistical tie, only a close match consistent with the two methods’ identical call counts. This is expected: FlatMerkle emits $2N - 2$ items and a same-convention Merkle tree performs $2N - 1$ calls – essentially the same work either way. The value of FlatMerkle was never faster construction; it is that the resulting commitment is a flat hash of raw bytes (Section 9), which a Merkle root is not, and that proofs need no separate tree-traversal metadata.
2. **Against the independently-finalized single-SHA-256 Merkle-tree convention benchmarked here** (*Merkle tree, standard* above, matching many real Merkle tree implementations that hash each node with one ordinary, self-contained SHA-256 call) – **FlatMerkle is almost exactly 2 $\times$ faster** (ratio 1.96–2.00 $\times$), because a 64-byte node input needs two compression calls once padded and length-encoded, where FlatMerkle’s continuous stream needs only one call per item and pays for padding exactly once, for the whole file. (This specific comparison does not include Bitcoin’s own transaction Merkle tree, which uses double-SHA256 per node rather than this convention – see the note above.)

Extrapolating to $N = 2^{20} = 1,048,576$ elements from the exact call-count formulas ($2N - 1$ for FlatMerkle and the same-convention tree, $4N - 2$ for the standard tree) and the throughput measured at $N = 65,536$, pooled across the three methods ($\approx 12,057$ calls/s; the per-method figures, 11,880 / 12,164 / 12,093, differ by about $\pm 2\%$, which we treat as noise) gives ≈ 174 s for FlatMerkle and the same-convention tree and ≈ 348 s for the standard tree. That the first two come out identical is an artifact of applying one pooled rate to equal call counts, not a claim that equal call counts guarantee equal wall-clock time. These are computed extrapolations, not timed runs.

All of these comparisons are single-threaded. FlatMerkle’s chaining is inherently sequential: each checkpoint is the running state after everything before it. The message-schedule expansion of each of the N data blocks, a substantial part of a compression call, can still be precomputed in parallel, since data blocks are known in advance; only the MID blocks and the chaining itself must run in order. A Merkle tree’s subtrees, by contrast, can be hashed entirely in parallel, so on multi-core hardware a parallel Merkle build can pull ahead of FlatMerkle by close to the core count at large N . How much of FlatMerkle’s sequential fraction a parallel implementation could hide is not something we measured here.

9 Application: committing data in a Bitcoin transaction annex

9.1 Why this needs a flat hash, not a tree

BIP-341 [36] defines an optional *annex* field in a Taproot witness stack: any item present when other conditions are met is treated as annex data, and its commitment inside the transaction’s signature hash is

$$\text{sha_annex} = \text{SHA256}(\text{compact_size}(\text{len}(\text{annex})) \parallel \text{annex}), \quad (14)$$

a single, ordinary, untagged SHA-256 of the length-prefixed annex bytes – consensus never computes anything tree-shaped over annex content. This is precisely the setting motivating this paper: if the annex bytes are the FlatMerkle structured file of Section 2 (with the short compact-size preamble accounted for), then `sha_annex` – already computed by every consensus-validating node, with zero protocol changes – *is* the FlatMerkle digest, and anyone holding the annex can produce $O(\log N)$ inclusion proofs for any element it contains, verifiable by anyone who has `sha_annex` (Section 9.2 below explains why this is *not* the same thing as the transaction’s final signature hash, and what a verifier actually needs).

9.2 What this buys

- A timestamped, immutable commitment to structured application data, inheriting Bitcoin’s ordering and immutability guarantees, at the cost of ordinary transaction fees for the annex bytes.
- $O(\log N)$ inclusion proofs verifiable by any third party who holds `sha_annex` (a public 32-byte value, obtainable from any node that has validated the transaction, or supplied directly alongside the proof) and a short proof – no need to download or re-hash the whole annex. How a lightweight verifier obtains an *authenticated* `sha_annex` for the confirmed transaction is a separate problem, with a real cost trade-off (Section 9.4). `sha_annex` itself is *not* the same object as the transaction’s final Taproot signature hash: it is one 32-byte component folded into the larger `SigMsg` that the signature hash is computed over, not recoverable from the signature hash alone (that would require inverting a hash function). A verifier who has

only the final signature digest, with no other data, cannot check a FlatMerkle proof against it directly; what must actually be transmitted alongside the proof is `sha_annex` itself.

- By Theorem 5.9, a given `sha_annex` value cannot be made to accept two different, conflicting values for the same index: an accepting proof pins down a unique answer. This does *not* mean a publisher gains nothing from an annex that deviates from the canonical layout – Remark 5.10 exhibits a real forgery against exactly that stronger claim – so a third party who merely downloads the annex, hashes it, and assumes every accepting proof describes bytes it can literally see at the corresponding offset is *not* safe. What is safe: any party willing to hold the full annex can run `SCANFORVIOLATIONS` once ($O(N)$) and, if the annex is non-canonical, publish an $O(\log N)$ fraud proof (Section 5.5) that any lightweight verifier – with only `sha_annex` – can check without downloading the annex at all.

9.3 Alternatives within a linearly hashed annex

Under the threat model of Section 1, a design must give an opponent $O(\log N)$ proofs both of inclusion and of malformation. Three natural alternatives to FlatMerkle meet the size goal but not both proof goals (Table 7).

Merkle root as a trailer. Lay the annex out as `0x50` header, data, then a 32-byte Merkle root ρ of the data, positioned so that $\rho \parallel 0x80 \parallel 0^* \parallel \text{len}$ fills the final SHA-256 block. Given the 32-byte midstate s entering that block, a verifier checks $\text{sha_annex} = \mathcal{C}(s, \rho \parallel \text{pad})$ with one call – binding ρ under Assumption 5.1, the same assumption FlatMerkle uses – and then an ordinary Merkle path against ρ . Overhead is about 96 bytes, and inclusion proofs are only 32 bytes longer than a plain Merkle proof. But a malicious committer can publish $\rho = \text{Merkle}(D')$, where D' equals the published data except at position 0. Every inclusion proof then certifies $D'[0]$, not the published $D[0]$, and exposing this requires proving the bytes physically at position 0 of the annex: a verifier must recompute the linear chain from there to the final block, $\Theta(N)$ compression calls. (We confirmed the one-call opening against the reference SHA-256 implementation.)

A fully serialized Merkle tree. Publishing every internal node in the annex, as well as the root, has the same weakness: an inconsistent node can be placed early in the stream, and proving the physical content of any position in a linear hash costs $\Theta(N)$ without re-injected states.

Root trailer plus an interactive bisection game. The $\Theta(N)$ malformation check can be turned into $O(\log N)$ rounds of an interactive game: the committer posts intermediate chain states, the opponent bisects to a single disputed compression call, and a dispute mechanism such as BitVMX [15] adjudicates it. This reaches logarithmic cost per round, but needs $O(\log N)$ on-chain rounds, the committer’s participation, and timeouts to handle a committer who stops responding. FlatMerkle’s fraud proofs (Section 5.5) are a single, non-interactive message that anyone holding the annex can publish.

9.4 What FlatMerkle verification does *not* give you: binding `sha_annex` to a specific transaction

Everything above is about *FlatMerkle verification*: given a claimed `sha_annex` value and a short proof, confirming a specific element sits at a specific index within the annex that hashes to it. That is a completely separate question from *Bitcoin authentication*: given a claimed `sha_annex` value and a specific transaction instance, confirming that value genuinely is that instance’s annex hash, and not an arbitrary 32 bytes an adversary invented and handed to a lightweight verifier alongside an otherwise-valid-looking FlatMerkle proof. FlatMerkle’s binding guarantee (Theorem 5.9) says nothing about this: it is a statement about the relationship between a digest and the proofs it

Design	Annex size	Inclusion	Malformation	Interactive
Root trailer	$P + O(1)$	$O(\log N)$	$\Theta(N)$	no
Serialized Merkle tree	$\approx 2P$	$O(\log N)$	$\Theta(N)$	no
Root trailer + bisection	$P + O(1)$	$O(\log N)$	$O(\log N)$ per round*	yes
FlatMerkle	$\approx 2P$	$O(\log N)$	$O(\log N)$	no

Table 7: Committing P bytes (N elements) inside a linearly hashed annex. “Inclusion” and “Malformation” give the size of a proof an opponent can produce and the verifier’s work to check it. FlatMerkle pays roughly $2\times$ in size for non-interactive logarithmic proofs of both kinds. *Over $O(\log N)$ interactive rounds.

accepts, not about how a verifier learns which digest belongs to which transaction. A verifier who accepts *any* 32 bytes as “the” `sha_annex` for a given transaction, with no further check, can be handed a proof for a completely different, adversary-chosen annex and never know the difference.

A subtlety in identifying “a given transaction” at all: the annex is *witness* data, and a transaction’s `txid` – unlike its `wtxid` – is computed over the non-witness serialization only (BIP-141). Two witness variants of the same underlying transaction can therefore share an identical `txid` while carrying different annexes, different Taproot signatures, and different `wtxids`; only one variant is ever mined, but a party who knows merely a `txid` has no way to tell, from the `txid` alone, which witness/annex instance that was. So a specific transaction must be identified by its `wtxid`, or by some other authenticated source of the confirmed witness – a bare `txid` is not enough to pin down which annex is even being talked about.

Closing the authentication gap itself decomposes into two separate sub-properties, since a candidate `sha_annex` and a signature alone establish only the first:

1. *candidate* `sha_annex` $\rightarrow$ *a validly signed witness variant* – some witness instance exists for which this `sha_annex`, this signature, and these transaction/spent-output fields are mutually consistent;
2. *that witness variant* $\rightarrow$ *actually confirmed on-chain* – this specific witness instance, not some other validly-signed alternative the same signer could equally have produced, is the one Bitcoin’s consensus actually accepted.

Three candidate mechanisms, addressing these to different degrees:

- **Reconstruct *SigMsg* and check the signature.** `sha_annex` is one field embedded inside the larger *SigMsg* that BIP-341’s Taproot key-path signature is computed over (together with, among other fields, `sha_prevouts`, `sha_amounts`, `sha_scriptpubkeys`, `sha_sequences`, `sha_outputs`, and `input_index`) – it is *not* independently serialized or recoverable from the final signature hash alone. A verifier who is given the full unsigned transaction (or the specific fields *SigMsg* needs), the spending input’s Taproot public key, and its signature can recompute *SigMsg* – with the claimed `sha_annex` substituted in at the right position – and check the BIP-340 Schnorr signature over `hash_TapSighash(0x00||SigMsg)` directly. If it validates, the claimed `sha_annex` is, up to breaking Schnorr unforgeability, part of a genuinely, validly signed witness for this transaction’s inputs – *sub-property 1 only*. It does *not*, by itself, establish sub-property 2: the signature itself is witness data too, excluded from `txid` the same way the annex is, so a signer holding the private key can construct another equally valid signature over a different annex without changing the base transaction or its `txid` at all. This option proves

authorization and internal consistency of a witness variant, not that this particular variant is the one Bitcoin confirmed.

- **Trust a validating node or block explorer.** A lightweight verifier can simply ask a full node (or a service that has one) “what is the annex hash for input i of the transaction with this `wtxid` (or this block height and position)”, the same way it might already trust a block explorer for a confirmation status. Asking by `wtxid` (or an equivalent witness-committing identifier) rather than `txid` addresses sub-property 2 as well, at the cost of reintroducing a trusted third party the rest of this construction otherwise avoids.
- **An authenticated copy of the confirmed witness, tied to the block’s witness commitment.** BIP-141 commits every block’s transactions’ `wtxids` into the coinbase transaction. A verifier who is given the witness commitment path from the claimed witness serialization up to a block header they otherwise trust (e.g. via an SPV-style proof of work) gets both sub-properties without trusting any single third party. But this route is *not* $O(\log N)$: a `wtxid` is a double SHA-256 over the transaction’s *complete* serialization, witness included, and therefore over the annex bytes themselves. To check that a claimed `sha_annex` belongs to the confirmed `wtxid`, the verifier must recompute the `wtxid` from the full witness, i.e. hash all $\Theta(N)$ annex bytes. Nothing computed for one chain can be reused for the other: `sha_annex` and the `wtxid` are two independent SHA-256 chains over the same bytes, started from different states, so FlatMerkle’s checkpoints for the first say nothing about the second.

The three mechanisms therefore form a trilemma, and none of them gives a lightweight verifier everything at once:

Mechanism	Confirmed variant	Trustless	Verifier cost
Signature over <i>SigMsg</i>	no	yes	$O(1)$
Trusted node / explorer	yes	no	$O(1)$
Witness commitment to a header	yes	yes	$O(N)$

The first option, on its own, reduces the scheme to an ordinary digital signature over a commitment – which needs no blockchain at all – and so gives up the timestamping and ordering guarantees listed in Section 9.2; the second reintroduces exactly the trusted party the construction is meant to avoid; the third is trustless and confirmation-aware but costs as much as downloading the annex. Hence, for a purely lightweight, trustless verifier, the $O(\log N)$ cost of a FlatMerkle proof is the cost of checking membership *given* an authenticated `sha_annex`, not the end-to-end cost of learning what Bitcoin confirmed. The $O(N)$ authentication step can be done once per annex by any party and amortized across many proofs only if verifiers are willing to trust that party’s result, which is the second option again. Settings with their own accountability structure can avoid the trilemma: in the rollup of Section 10, sub-property 1 is sufficient for the dispute itself, as explained there. None of these mechanisms is implemented or benchmarked in this paper; a deployment needs to pick one explicitly rather than assume FlatMerkle’s own guarantees cover it.

9.5 Block alignment

`sha_annex = SHA256(compact_size(len(annex)) || annex)` prepends a length preamble whose own size, $p \in \{1, 3, 5, 9\}$ bytes, depends on `len(annex)` (BIP-341’s `CompactSize` encoding). Since p is essentially never a multiple of 64, treating this as a simple byte-offset adjustment is not sufficient: unless corrected, every 64-byte FlatMerkle item (Section 2) straddles two real SHA-256 compression-block boundaries, so no single item corresponds to one call of $\mathcal{C}$ at all, breaking the correspondence

the whole construction relies on. The fix is a small, fixed *alignment header* of $h = (64 - p) \bmod 64$ bytes – since $p \in \{1, 3, 5, 9\}$ is never a multiple of 64, h is always one of $\{63, 61, 59, 55\}$, fixed once p is known (e.g. $h = 59$ for the common case $p = 5$) – placed immediately after the compact-size preamble and before the FlatMerkle structured file proper, so that $p + h \equiv 0 \pmod{64}$: exactly one real SHA-256 compression block. Unlike an ordinary alignment padding scheme, this header’s content is *not* free: BIP-341 recognizes a witness item as an annex at all only if the annex’s own first byte is `0x50`, so the header’s first byte is fixed to `0x50`. The remaining $h - 1$ bytes carry no alignment requirement of their own – padding *length*, not content, is what restores block-alignment – but this paper fixes them canonically rather than leaving them to individual deployments: they SHALL be all-zero bytes. Any other fixed convention (e.g. a magic tag identifying the annex as a FlatMerkle structured file) would work equally well for alignment, but leaving this genuinely a per-deployment choice would mean two conforming FlatMerkle-in-annex implementations could disagree on effective CK[0] for the byte-identical annex, breaking interoperability; fixing it here removes that ambiguity.

With this header in place, define the *effective* checkpoint CK[0] (in place of the raw IV, Section 2) as the real SHA-256 chain state *after* processing that one preamble-plus-header block, starting from the ordinary IV: every FlatMerkle item from $D[0]$ onward then begins exactly on a genuine 64-byte compression-block boundary within the annex’s own SHA-256 computation, and $\text{position}(\cdot)$ (Lemma 2.3) converts directly to a byte offset by multiplying by 64 and adding $p + h$, with no further adjustment needed. (A degenerate edge case is worth flagging, not solving here: if $\text{len}(\text{annex})$ sits exactly at one of the three `CompactSize` length thresholds – 253, 65536, or $2^{32} = 4294967296$ bytes, where p transitions from 1 to 3, 3 to 5, or 5 to 9 bytes respectively – adding the header itself could change p and hence h ; a real implementation should compute p from the *final* annex length, header included, not iterate.)

Changing the starting chaining state alone is not enough to make this a correct SHA-256 computation, however: SHA-256’s finalization encodes the *total* number of bits processed from the true, original IV – not from whatever effective checkpoint a particular application chooses to resume from. Standalone FlatMerkle’s $\text{finalize}(\text{raw_state}, N)$ (Algorithm 3.3) implicitly assumes the entire hashed message is the $M = 2N - 2$ structured-file items themselves, $64M$ bytes total. In the annex, one additional real SHA-256 block – the compact-size-preamble-plus-header block derived above – was hashed *before* effective CK[0], so the true total message length is $64(M + 1)$ bytes, not $64M$: using the standalone $\text{finalize}(\cdot, N)$ unmodified would encode the wrong length and simply fail to reproduce the real `sha_annex`, independent of anything else being correct. The fix is to use an annex-specific finalization that takes the extra prefix block count explicitly,

$$\text{finalize_annex}(\text{raw_state}, N) := \text{finalize}_{64(M+1)}(\text{raw_state}) \quad (15)$$

where $\text{finalize}_L(\cdot)$ denotes SHA-256’s ordinary length-suffix finalization parameterized directly by the true total byte count L (rather than inferring it solely from N), and Algorithm 3.3’s `Verify` must be instantiated with `finalize_annex`, not the standalone `finalize`, when checking a proof against `sha_annex`.

9.6 Caveats

- **Relay and standardness policy.** As of this writing, annex data is consensus-valid but not relayed or mined by default nodes; alternative relay policies (e.g. Libre Relay, which relays annex-bearing transactions under a specific marker convention) exist but are not universal. Reliance on this application should account for the relay policy landscape at deployment time, which may change.

- **Future consensus use of the annex.** BIP-341 currently defines the annex only as a reserved field for future extensions, and its own text is explicit and severe about this: “[u]ntil the meaning of this field is defined by another softfork, users SHOULD NOT include annex in transactions, or it may lead to PERMANENT FUND LOSS” [36]. This is not a hypothetical or minor concern – it is BIP-341’s own strongest warning language, aimed at exactly this kind of use. It carries no consensus meaning today beyond being committed to via `sha_annex`, but a future soft fork could assign consensus-level semantics to some or all annex content (for instance, reserving a specific prefix byte or sub-format) that conflicts with using raw annex bytes for arbitrary application data as described here, in a way that could render funds in outputs spent through such a transaction unspendable. Anyone deploying this application should treat that risk as live, not theoretical, and track proposals that touch the annex field before relying on it.

10 Application: a Bitcoin data-availability layer for an optimistic rollup

Section 9 treated the annex as a place to commit an arbitrary structured file. This section sketches a specific consumer of that mechanism: an optimistic rollup that uses a Bitcoin transaction annex as its data-availability (DA) layer, with FlatMerkle giving watchtowers a compact way to point at exactly the step they dispute. This is an architectural proposal, not a protocol this paper implements or benchmarks; the composition below is intended to be checked, not taken on faith, and several of its pieces are explicitly flagged as open design choices rather than settled specifications.

10.1 Setup: the state-transition commitment

A rollup sequencer periodically batches a sequence of m transactions $tx_1, \dots, tx_m$ applied to its ledger, starting from a known initial state $state_0$ (the previous batch’s final, already-committed state), and producing $state_1, \dots, state_m$, one per transaction. The transactions themselves must be published *in plain* as part of the committed data, not reduced to a hash: a data-availability layer that only commits to $H(tx_i)$ without publishing tx_i itself gives nobody the bytes they would need to recompute $state_i$ and dispute it, defeating the point of using an annex for DA at all. And since each step’s starting state is just the previous step’s ending state, $state_{i-1}$ needs no separate storage of its own: the sequencer publishes exactly $state_0$ followed by the pairs $(tx_1, state_1), \dots, (tx_m, state_m)$, one initial value plus one transaction-and-resulting-state per step, as a single FlatMerkle structured file (Section 2) inside the annex of a Bitcoin transaction it signs, using the block-alignment convention of Section 9.5. Anyone who downloads the full annex can replay $state_0 \rightarrow state_1 \rightarrow \dots \rightarrow state_m$ directly and confirm the sequencer’s claimed final state follows from its claimed initial state – an ordinary $O(m)$ check that needs nothing from this paper. FlatMerkle’s contribution is what it adds on top: an $O(\log N)$ way to point at, and prove the exact bytes of, one disputed step without redistributing the whole batch.

Packing this sequence into FlatMerkle’s fixed-width elements $D[a]$ needs an explicit framing rule, because each tx_i has variable length and occupies as many consecutive elements as its encoding requires, while each $state_i$, a fixed-width digest, occupies exactly one. A dispute proof must show not only the bytes of some elements but that those elements *are* tx_i and its neighbouring states. We therefore let the file begin with a *step table*: element 0 holds $state_0$, and the next $\lceil m \cdot e/b \rceil$ elements hold, for each step i , the index of the first element of tx_i and its length in elements, encoded in a fixed e -bit entry; $state_i$ is the element immediately after tx_i . A table whose entries overlap, leave

gaps, run past N , or are not in increasing order makes the batch invalid in the same way an incorrect transition does, and is disputed the same way. A watchtower disputing step $i \geq 2$ then proves, in one batched multi-element proof (Section 7), the table element(s) holding entries $i - 1$ and i , the elements of tx_i , and the elements holding $state_{i-1}$ and $state_i$; for $i = 1$, the entry for step 1, tx_1 , $state_1$, and element 0. The proof has size $O(\ell_i + \log N)$, where ℓ_i is the number of elements tx_i occupies.

Embedding this same commitment in an `OP_RETURN` output instead of the annex raises a separate question – how a structured file of this kind maps onto `OP_RETURN`’s own size and structuring conventions – that this paper does not address; everything here is specific to the annex.

10.2 Two independent watchtower checks

A set of watchtowers, each holding the full published annex, run two independent checks against every batch.

1. **Canonicity of the layout itself.** A watchtower runs `SCANFORVIOLATIONS` (Algorithm 5.1, $O(N)$ where N is the number of elements in the batch’s structured file) against the published annex. If the file is not canonically built, binding (Theorem 5.9) still holds – each index has at most one accepted value – but that value need not be the bytes physically present at the corresponding position (Remark 5.10). A light client following the batch through inclusion proofs and a full node replaying the raw annex bytes would then see two different batches. A batch that fails this check has published something that cannot be relied on as a faithful, unambiguous state-transition record at all, independent of whether any individual transition it contains happens to be correct, which this paper treats as sufficient grounds on its own for the surrounding rollup protocol to slash the sequencer, hand off sequencing to a new party, or force an exit to Layer 1 – the specific choice among these is a protocol-design question this paper does not resolve.
2. **Correctness of an individual state transition.** For a canonical batch, a watchtower checks each step $state_{i-1} \rightarrow state_i$ against its own independent execution of the rollup’s deterministic state-transition function. If it finds one step i where the published $state_i$ does not match, it can isolate exactly that step with a short FlatMerkle inclusion proof (Section 3, batched per Section 7 as described above) against the batch’s committed digest, without needing any other watchtower to re-derive or re-publish the rest of the batch.

10.3 Enforcing the checks on Bitcoin

Both checks above are things any off-chain party can already do by downloading the annex; the open problem is making a *failed* check economically consequential – getting Bitcoin’s own consensus, which cannot evaluate SHA-256 compression-function chaining or an arbitrary state-transition function directly in Script, to adjudicate the dispute and pay out a slashing bond. Two current, concrete mechanisms exist for this, and either fits here depending on how the rollup structures its own state-transition check: **BitVMX**, a virtual CPU that runs an arbitrary program (RISC-V or MIPS) and resolves a dispute by an n -ary search over a hash chain of the execution trace down to one instruction, adjudicated by a small, fixed piece of Script [15]; and **BitVM3**, a newer, garbled-circuit-based protocol purpose-built to verify a single SNARK proof, where a challenger evaluates a garbled verifier circuit off-chain and, on a false claim, posts a compact, constant-size fraud proof on-chain [18]. We use BitVMX below since it fits an arbitrary re-execution check directly; a rollup

that instead produces one succinct SNARK proof per batch could use BitVM3 for the same role at lower on-chain dispute cost.

Concretely, a watchtower that found a violation of either kind opens a BitVMX dispute whose program takes as input the FlatMerkle inclusion proof for the disputed index (or, for a canonicity violation, the fraud proof produced by BUILDFRAUDPROOF, Algorithm 5.3), the revealed step data, `sha_annex`, and enough of the Bitcoin transaction and its Taproot signature to reconstruct *SigMsg* (Section 9.4). The disputed program itself checks, in order: that the inclusion (or fraud) proof folds correctly to the claimed `sha_annex` under `finalize_annex` (Section 9.5); that the supplied signature validates over *SigMsg* for the claimed `sha_annex`; and, for a transition dispute, that re-applying the state-transition function to the revealed `statei-1` and `txi` does *not* yield the revealed `statei` (for a canonicity dispute, that VERIFYFRAUDPROOF accepts the supplied fraud proof). If BitVMX’s bisection game resolves in the watchtower’s favor, the pre-arranged slashing path pays out. Authenticating a signature against *SigMsg* this way establishes only the first of the two sub-properties identified in Section 9.4 – that a validly signed witness variant with this `sha_annex` exists, not that it is the specific variant Bitcoin confirmed. Chaining successive batch-commitment transactions (each spending an output of the previous one) does not close that gap: a spend references the previous transaction’s `txid`, which excludes witness data, so it does not fix which annex variant was mined. For the dispute, however, sub-property 1 is enough, provided the slashing condition covers *every* witness variant the sequencer signs, not only the confirmed one: an invalid (non-canonical, or transition-violating) annex that carries a valid sequencer signature is slashable whether or not it was mined, so the dispute program never needs to learn which variant Bitcoin confirmed, and the watchtowers, who run full nodes, dispute the mined one. A light client that wants to learn the rollup’s *confirmed* state from a FlatMerkle proof still faces the trilemma of Section 9.4.

The linkage between an externally-signed Bitcoin transaction and a value consumed inside a BitVMX program is itself the subject of ESSPI [16], which replaces a direct in-Script Schnorr verification – reported there as producing a roughly 1.1 GB Script when attempted directly – with a Winternitz-committed hash of the signed data, checked by a secondary BitVMX instance that the commitment, the signature, and the claimed data are mutually consistent. ESSPI’s own construction is built around a transaction signed by a counterparty *as part of* the linking protocol, and its authors report no formal security analysis of the composed protocol and flag exposure to Bitcoin’s transaction-replacement behavior; whether it applies directly to authenticating an already-existing, independently signed batch-commitment transaction of the kind described here, or needs adaptation, is not something either ESSPI or this paper settles.

10.4 Scope

This section is a design sketch composing three separately published, separately analyzed mechanisms – FlatMerkle’s inclusion and fraud proofs, BitVMX’s dispute game, and ESSPI’s signature-linkage technique – and none of the composition itself has been implemented, benchmarked, or given a security proof here. Left unaddressed: the concrete slashing, sequencer- succession, and exit-mode game the surrounding rollup protocol would need; collateral sizing and the round/cost trade-offs BitVMX itself exposes as configurable [15]; the encoding of the step table’s entries; and, as already noted, embedding a structured file of this kind in `OP_RETURN` rather than the annex.

11 Application: an EVM rollup calldata commitment

FlatMerkle applies equally to an EVM-compatible rollup’s data-availability problem, though for a narrower reason than on Bitcoin: on an EVM chain, FlatMerkle is not the cheapest way to pub-

lish a byte, only the cheapest way we find to make an already-published, already-expensive byte cheaply disputable while keeping it *persistently* available. We work this out in full for Ethereum first, then redo the same four options against Rootstock [14], a Bitcoin-merge-mined, EVM-compatible sidechain, and against Arbitrum [22], an Ethereum rollup in its own right – not as further illustrations of the same result, but because the three chains turn out to disagree with each other on which option wins, for a reason worth understanding: the ranking depends on specific, independently adoptable protocol rules, not on anything intrinsic to FlatMerkle or to the EVM itself. The full, from-scratch derivation – every gas constant cited to its defining EIP, RSKIP, or ArbOS release note, every non-protocol approximation flagged as such – is given in Appendix A; this section states the setup, the four options being compared, and (§11.3) the result.

11.1 Why calldata, not blobs

A rollup sequencer can publish its batch data to Ethereum in ordinary transaction *calldata*, costed at 16 gas per non-zero byte (4 gas per zero byte), a protocol constant unchanged since 2019 [1]. Since Dencun, it can instead publish the same bytes in a *blob* [5]: a separate, independently fee-market-priced resource (blob gas is charged 1:1 against a blob’s 131,072 bytes, `BLOB_GAS_PER_BLOB` = 2^{17}), historically far cheaper per byte than calldata when blob space demand is low – though this ratio floats with two independent, unrelated fee markets and is not a fixed protocol constant, so we do not rely on any particular multiple here. The catch, stated directly in EIP-4844’s own text, is that blobs are not meant to be kept: “[the consensus layer] blobs must be kept for at least a certain period[,] around 18 days” (`MIN_EPOCHS_FOR_BLOB_SIDECARS_REQUESTS` = 4096 epochs) [5], after which nodes are free to discard them. A rollup that wants Ethereum itself to remain a permanent record – so that a new L2 node can reconstruct the entire rollup state by replaying L1 history alone, indefinitely, the way this paper’s other applications assume the underlying chain’s data stays available – cannot rely on blobs for that role; ordinary calldata, once included in a block, has no comparable expiry. This section is about that persistent, expensive-by-design case: given a sequencer already pays calldata’s price to publish P bytes permanently, what is the cheapest way to also get a compact, on-chain, disputable commitment to those bytes? On Rootstock the question is not even a choice: there is no blob-carrying transaction type there at all, so calldata (or contract storage) is the only way to publish data on-chain in the first place, making a cheap on-chain commitment mechanism, if anything, more directly load-bearing there than on Ethereum, not less.

11.2 Four ways to commit to published calldata

- A. Copy the calldata into contract storage (`SSTORE`) and keep it there.
- B. Build an ordinary Keccak256 Merkle tree over the calldata inside the contract, store only the 32-byte root, and discard everything else.
- C. Build a FlatMerkle structured file over the data *off-chain*, submit the whole structured file as calldata, and hash it on-chain in one call to the SHA-256 precompile (address `0x02`), storing only the digest.
- D. Build the FlatMerkle structured file *inside* the contract, from ordinary (unstructured) calldata, hashing it incrementally via the BLAKE2 compression-function precompile (address `0x09`, [11]), storing only the final digest.

Option D uses BLAKE2b’s compression function $F(h, m, t, f)$ as $\mathcal{C}$. Besides the chaining value h and the message block m , F takes a byte counter t and a final-block flag f , and the precompile accepts any values for both. A FlatMerkle instantiation must fix them from the item’s position alone – for example, t the running byte count $128 \cdot (\text{item index} + 1)$ and $f = 0$ for every item, with any finalization applied as a separate fixed step – in the contract that builds the file and in any verifier that later checks a proof against its digest, and must never take them from a proof. Otherwise a proof could vary them, and Assumption 5.1, which is stated for a fixed function of (h, m) , would no longer cover the verifier’s computation (Remark 5.2 discusses BLAKE2b’s own security margin).

Options B and D need no fraud-proof machinery of the kind Section 5.5 gives for the Bitcoin annex application: the commitment is built by trusted contract code from data the contract itself received, so it cannot be non-canonical the way an externally supplied FlatMerkle file (option C, or the Bitcoin annex of Section 9) can be. A later dispute over the *data itself* (was this the right previous state? did this transaction really produce this result?) is a separate, application-level question this section does not address, matching Section 10’s treatment of the analogous Bitcoin case.

11.3 Result

Option	Calldata	Ethereum	Rootstock	Arbitrum
A. SSTORE, persistent	P	706.625	641	706.625
B. Keccak tree, discarded	P	40	17.3–18.4	17.3–18.4
C. FlatMerkle, off-chain	$2P$	80	32.75–43.5	32.75–34.1
D. FlatMerkle, on-chain	P	40	≈ 27.5	16.2–18.1

Table 8: Total transaction gas per original byte P (asymptotic), current protocol constants on three chains, derived in Appendix A. Ethereum: EIP-7623 calldata floor and EIP-2929 warm-call discount both present (floor dominates). Rootstock: neither. Arbitrum: EIP-2929 discount, no EIP-7623 floor – the one combination where D is not a clear loser against B.

The three chains disagree with each other, and the disagreement is the result. On Ethereum, option D does not beat option B: they tie, both at the EIP-7623 floor, which is indifferent to a more than tenfold spread in the two options’ real hashing cost. On Rootstock, option D loses to option B by a real margin (≈ 27.5 against ≈ 17.3 –18.4 gas/byte): with no floor to hide behind, raw efficiency matters again, and Rootstock’s missing warm-call discount taxes option D’s many separate BLAKE2F calls more than it costs option B’s zero calls. On Arbitrum – no floor, but *with* the warm-call discount – option D is, for the first time, at least competitive with option B and plausibly a little cheaper, though not decisively so given the uncertainty in our own “glue code” estimate (Appendix A). So the premise this section started from – that FlatMerkle built inside the contract should be strictly cheaper than a plain Merkle tree – is only borne out, and only weakly, on the one chain of the three that combines both relevant protocol properties (a missing calldata floor and a present warm-call discount); it does not hold in general, and which of B or D wins is a fact about a given chain’s own protocol rules, not about FlatMerkle or about “the EVM” as a single target. FlatMerkle-in-contract is, however, a large and consistent improvement over building the structured file off-chain (option C) and over persisting the raw data in storage (option A) on all three chains – roughly 1.5–2 $\times$ cheaper than C and well over an order of magnitude cheaper than A throughout, for reasons and with a fuller discussion of what this does and does not establish given in Appendix A. Ethereum’s pending Glamsterdam fork, as currently drafted, would raise the calldata

floor to 64 gas per byte; that raises the Ethereum column for B, C and D to 64, 128 and 64 but leaves the ranking unchanged (Appendix A).

12 Related work

12.1 Linking-based time-stamping

The closest structural antecedent we found is the *binary linking scheme* of Buldas, Laud, Lipmaa, and Villemson [2], which improves the linear hash-chain time-stamping of Haber and Stornetta [10]. There, the n -th linking item is $L_n = H(H_n, L_{n-1}, L_{f(n)})$: each item hashes the previous item and one earlier item at a power-of-two-structured distance $f(n)$, so that a verifying chain between any two items has length $O(\log n)$ (at most $3\lceil \log_2 n \rceil - 2$), and Buldas, Lipmaa, and Schoenmakers later gave optimally sized time-stamps for such schemes using threaded authentication trees [3]. FlatMerkle’s re-injection of $CK[a]$ through $MID(a)$ plays the same role as the back-link $L_{f(n)}$: it is what makes a distant point of the chain reachable in one step. The differences are in what is hashed and what is proved. In a linking scheme, each L_n is the output of a separate hash invocation over several inputs, and the object being authenticated is the one-way dependence between two items, for relative temporal ordering. In FlatMerkle, the back-links are compression-function chaining states folded into a single Merkle–Damgård stream, so the commitment is the ordinary hash of one byte string, and the statements proved are inclusion of an element in that string and, via fraud proofs, malformation of its layout.

12.2 Authenticated append-only skip lists

Among authenticated data structures, the closest prior art we identified is Maniatis and Baker’s *authenticated append-only skip lists* (AASL) [20], itself building on the persistent authenticated dictionary of Goodrich, Tamassia, and Schwerin [8]. Both use the same deterministic, power-of-two skip-index structure that FlatMerkle’s position formula (Lemma 2.3) also produces, and both achieve $O(\log N)$ membership and append proofs.

The mechanisms differ, however: AASL is a genuine skip list, with distinct, separately tagged hash nodes at each level connected by explicit pointers, computed via multiple separate hash-function invocations per node. FlatMerkle achieves the equivalent skip structure without any separate node objects or multiple hash invocations – everything is a single, continuous Merkle–Damgård stream, and the “skip” capability comes entirely from re-injecting the same hash function’s own internal state back into that one stream. This is what allows FlatMerkle’s digest to be a literal, unmodified, single-pass hash of the raw structured bytes (Section 9) – a property we are not aware of AASL, its predecessors, or the linking schemes above offering, since their commitments are tied to their own multi-node hash structure rather than a flat byte stream.

12.3 Other related, but structurally distinct, work

- **Proofs of sequential work** [19] and related hash-chaining constructions also reuse intermediate hash states, but for a different purpose (proving elapsed sequential computation) and without the recursive, doubling checkpoint structure needed for logarithmic inclusion proofs.
- **Historical/tamper-evident logging** (e.g. Crosby and Wallach’s history trees [6]) targets a related but distinct problem – efficient proofs of consistency between two versions of an append-only log – using an explicitly tree-shaped structure. The underlying probabilistic skip list data structure itself traces back to Pugh [24].

- **Merkle mountain ranges** [34] generalize Merkle trees to efficiently-appendable forests of perfect binary trees, but remain tree-shaped structures, not a single linear hash stream.
- **Bitcoin’s own (unrelated) use of “skip lists”** for block header lookups (e.g. for efficient median-time-past or difficulty retargeting queries) skips over *block headers* via explicit stored pointers between blocks, not via hash-chain state re-injection, and solves a different problem (locating a specific historical block quickly) than proving inclusion inside a single committed dataset.

12.4 Scope of this comparison

This is a targeted literature search focused on the specific technique of achieving logarithmic proofs from a single linear hash stream via compression-state re-injection, not an exhaustive survey of authenticated data structures or of the time-stamping literature, and not a patent search. We are not aware of prior work using exactly this mechanism, but we note this as “apparently undocumented in what we found,” not as a claim of novelty suitable for a patent filing or a definitive related-work section without independent verification.

13 Discussion and conclusion

13.1 Beyond SHA-256: sponge-based hash functions

The construction as presented relies on Assumption 5.3: a Merkle–Damgård hash with an exposed, fixed-width compression function whose entire internal state is exposed at every step. This holds for SHA-256, but not for sponge constructions such as Keccak-256 (used, in its non-NIST-standardized form, by Ethereum), and adapting FlatMerkle to a sponge is not a drop-in change:

- Keccak-f[1600]’s internal state (1600 bits, 200 bytes) is larger than its absorbable rate per permutation call (1088 bits, 136 bytes for the 256-bit-output parameterization), so a full checkpoint cannot be re-injected in a single call the way a 32-byte SHA-256 state fits inside one 64-byte block; either a squeezed/truncated summary must be re-absorbed instead of the raw state, or a checkpoint must be spread across multiple absorb calls.
- The naive analog of $\mathcal{C}$ for a sponge – one absorption step, $\mathcal{C}(s, m) = P(s \oplus (m \parallel 0^c))$ for the Keccak-f[1600] permutation P , rate r bits, capacity c bits ($r + c = 1600$), and $m \in \{0, 1\}^r$ – is *not* a plausible candidate for an assumption analogous to Assumption 5.1: it is trivially breakable, with no computation at all. P is a permutation (injective), so it has no collisions of its own, but the absorption step built from it does: for any nonzero $\delta \in \{0, 1\}^r$ – *rate*-sized, matching m ’s own width, not the capacity – the pair $s' = s \oplus (\delta \parallel 0^c)$, $m' = m \oplus \delta$ satisfies

$$s' \oplus (m' \parallel 0^c) = s \oplus (\delta \parallel 0^c) \oplus ((m \oplus \delta) \parallel 0^c) = s \oplus (m \parallel 0^c)$$

since the two $(\delta \parallel 0^c)$ terms cancel under XOR – the *same* permutation input, hence $\mathcal{C}(s, m) = \mathcal{C}(s', m')$ with $(s, m) \neq (s', m')$: a free-start collision found by inspection, not cryptanalysis. (A capacity-sized δ , added only to s ’s capacity portion with no matching change to m , would *not* work: m ’s own XOR never reaches the capacity portion at all, so nothing would cancel it. The trivial collision instead exploits the *rate* portion, which s and m both act on.) This is not a defect specific to Keccak; it is a structural property of building a single-call “compression function” directly out of a sponge’s absorption step: free-start collision resistance (Assumption 5.1) requires the *entire* starting state s to be safely adversary-chosen, but the rate portion

of s and the message m act on the permutation’s input in exactly the same additive way, so any adversarial freedom to choose s ’s rate bits is redundant with – and trivially cancels against – the freedom to choose m . A sponge-based FlatMerkle variant therefore cannot simply substitute P for $\mathcal{C}$ and inherit Section 5’s security argument; it would need a fundamentally different re-injection mechanism – one that does not hand an adversary this redundant rate/message freedom, e.g. re-injecting only squeezed output through the rate portion while the capacity is never adversarially set and the rate is never simultaneously free on both sides of a single absorb call – and a correspondingly different security assumption and proof, not merely “its own independent justification” for the same one.

- Ethereum’s Keccak-256 uses the original Keccak padding (suffix `0x01`), not NIST SHA3-256’s domain-separated padding (suffix `0x06`); an EVM-targeted variant would need to pin down the exact finalization convention to match on-chain verification, in addition to resolving the free-start-assumption problem above.

We consider a sponge-based FlatMerkle variant a natural and promising direction for future work, but a genuinely separate construction requiring its own from-scratch mechanism, security analysis, and implementation, not a parameter change to the scheme presented here.

13.2 Incremental growth

Doubling the number of elements from N to $2N$ turns out to be a pure append at the byte level:

$$F(2N) = F(N) \parallel \text{MID}(0) \parallel \text{block}(N, L) \quad (16)$$

– the old file’s bytes are untouched, followed by one new MID block (re-injecting the public IV, essentially free) and a fresh, self-contained block for the new elements. We verified this directly against the reference implementation for $N \in \{8, 16, 32\}$: the old file is always an exact byte-for-byte prefix of the new one. This is a narrower property than what the append-only structures of Section 12 offer, though, and the comparison should not be overstated: AASL [20] and Merkle mountain ranges [34] both support appending *one* new element at a time, with the structure remaining valid and provable after every single append; what we have shown here is only that *exact doubling* is a pure append, not that FlatMerkle supports incremental single-element growth in general – Section 13.3 below is exactly the missing piece that would be needed for that. What we have not yet worked out is the cheapest way to *upgrade* an existing proof for an index among the old N elements so it remains valid against the new digest – it appears to need only a small, index-dependent extension rather than a full rebuild, but we leave a precise statement and proof of that to future work.

13.3 Non-power-of-two N

The construction as presented requires N to be an exact power of two. Real datasets essentially never satisfy this exactly, and this paper does not currently specify a recommended handling: padding up to the next power of two is simplest but can waste up to nearly $2\times$ the space in padding elements alone, while an asymmetric “leftmost complete subtree” construction (in the style of RFC 6962’s Merkle tree [13]) would avoid the waste at the cost of a more complex, size-dependent position formula and branch rule. We leave working out and analyzing that variant to future work.

13.4 Summary

FlatMerkle shows that the tree-shaped structure of a Merkle tree is not actually necessary to obtain $O(\log N)$ inclusion proofs – only a disciplined way of periodically making the hash function’s

own internal state independently revealable is. This lets logarithmic-space, logarithmic-time inclusion proofs coexist with commitments that are, syntactically, ordinary flat hashes of ordinary byte strings – a property specifically useful wherever the commitment layer was fixed for other reasons and cannot be changed to accommodate a tree-shaped hash, as in the Bitcoin transaction annex application of Section 9. The construction’s security reduces cleanly to free-start collision resistance of the underlying compression function (Theorem 5.8), and its single-threaded construction speed is essentially tied with an equally-optimized Merkle tree and roughly $2\times$ faster than how Merkle trees are conventionally implemented in practice (Section 8) – though FlatMerkle’s chaining is inherently sequential in a way an ordinary Merkle tree’s independent subtrees are not, so a parallelized multi-core Merkle build can close or reverse that gap at large N (Section 8 discusses how much of FlatMerkle’s own work can still be precomputed in parallel). Soundness against equivocation – no digest can be made to accept two conflicting answers for the same index – holds regardless of whether the file was canonically built, under Assumption 5.1 (Theorem 5.9; “unconditional” here means unconditional with respect to *file canonicity*, not unconditional cryptographically), but soundness against a *non-canonically built* file does not come for free: a malicious publisher genuinely can construct a file that admits an accepting proof for data other than what a reader would find at that position (Remark 5.10). What restores a practical guarantee here is the fraud-proof mechanism of Section 5.5: any party willing to hold the whole file can detect such a deviation in $O(N)$ and prove it in $O(\log N)$, turning FlatMerkle’s trust model, for files whose canonicity is not otherwise attested to, into an optimistic one – safe by default, and falsifiable by a single honest watcher within a challenge window, rather than unconditionally safe with no watcher required at all.

A EVM rollup calldata-commitment costs: full derivation

This appendix gives the full, from-scratch derivation behind Table 8 and Section 11.3: every gas constant used, cited to its defining EIP, RSKIP, or ArbOS release note; the per-option computation on each of the three chains; and a fuller discussion of what the result does and does not establish. The numbers below are computed in gas units (not ETH/RBTC/USD, which float with market price and are not this paper’s concern) against current protocol constants as of this writing, with every non-protocol approximation (implementation-dependent “glue code”) flagged as such.

A.1 A protocol fact the naive comparison misses

A gas-cost comparison based only on calldata’s 16 gas/byte plus each option’s hashing cost misses a rule that binds precisely in this situation. EIP-7623, live on Ethereum mainnet since the Pectra upgrade (May 2025), adds a *floor* to every transaction’s gas cost [35]. Writing B for calldata bytes (worst case, all non-zero) and E for all other (“execution”) gas used – hashing, storage writes, memory operations – the total charged is

$$\text{gasUsed} = 21000 + \max(16B + E, 40B), \quad (17)$$

using EIP-7623’s own parameters (STANDARD_TOKEN_COST= 4, TOTAL_COST_FLOOR_PER_TOKEN= 10, and 4 “tokens” per non-zero byte, giving $16B$ and $40B$ respectively). The floor $40B$ *wins* whenever $E < 24B$: whenever a transaction’s real computation is cheap relative to how much calldata it carries – exactly the profile of a DA-posting transaction, whose entire purpose is to carry bytes, not compute on them. This means that for any of options B, C, D below, the specific hashing algorithm’s efficiency turns out not to matter once its execution gas is (as we verify it is, in every case) comfortably under the 24-gas/calldata-byte threshold: the transaction is billed at the floor

regardless, and what remains decisive is only how many calldata bytes had to be submitted in the first place. EIP-7623 is an Ethereum-mainnet protocol rule, not a property every EVM-compatible chain shares by virtue of being EVM-compatible – Section A.3 below returns to this once Rootstock is on the table, since Rootstock has no analogous floor.

A.2 The four options on Ethereum, computed

Constants used, each unchanged since its introducing EIP and (for the hash-related ones) not affected by any currently *live* repricing – two proposals that would raise them, EIP-2666 and EIP-7667, remain Stagnant, and the draft Glamsterdam repricing set leaves compute opcodes and precompiles unchanged (EIP-7904, listed in [31]); upcoming changes that *do* affect this comparison are discussed at the end of this appendix:

- KECCAK256 opcode: $30 + 6\lceil \text{len}/32 \rceil$ gas.
- SHA-256 precompile (0x02): $60 + 12\lceil \text{len}/32 \rceil$ gas.
- BLAKE2F precompile (0x09): 1 gas per round; the standard BLAKE2b compression uses 12 rounds, so 12 gas per call, each call processing one 128-byte message block against a 64-byte state [11].
- Cold SSTORE of a zero storage slot to a non-zero value: `SSTORE_SET_GAS` = 20000 [33] plus `COLD_SLOAD_COST` = 2100 [4], i.e. 22100 gas per new 32-byte slot – higher than a pre-2929 estimate of 20000 alone.
- A `STATICCALL` to a precompile is always “warm” post-Berlin, costing `WARM_STORAGE_READ_COST` = 100 gas [4] on top of the precompile’s own listed price – a cost the precompile gas figures above do *not* include, and which matters once an option needs many separate precompile calls rather than one.

Let P be the total original (unstructured) payload size in bytes, large enough that fixed per-transaction terms (the 21000 base, the small constant terms in the hash formulas) amortize away.

A – persistent storage. $B = P$; $E = (P/32) \times 22100 = 690.625P$. Since $E/B \gg 24$, the floor does not bind: $\text{gasUsed} \approx 16P + 690.625P = 706.625P$.

B – Keccak256 tree, discarded. $B = P$. Using this paper’s own “same-convention” tree (leaf = H (32-byte chunk), internal = $H(\text{left}||\text{right})$, matching Section 8): $N = P/32$ leaves, $2N - 1 \approx 2N$ calls, $E \approx 36N + 42N = 78N = 2.4375P$ (or $\approx 1.3125P$ if raw chunks are used directly as leaves, skipping a separate leaf hash). Either way $E/B \ll 24$: the floor binds, $\text{gasUsed} = 40P$.

C – FlatMerkle, built off-chain, hashed via SHA-256. The structured file is $\approx 2P$ bytes ($2N - 2 \approx 2N$ items, $N = P/64$, 64 bytes each – the same $\approx 2\times$ blowup already measured in Section 8), all of it submitted as calldata: $B = 2P$. Hashing it in one SHA-256 precompile call costs $E \approx 0.375 \times 2P = 0.75P$. Using BLAKE2F instead does not need a one-shot hash at all: BLAKE2F only exposes the raw, one-block-at-a-time compression function (unlike SHA-256’s precompile, which hashes an arbitrary-length input in a single call), so verifying the same structured file this way needs $\approx 2N = P/64$ separate chained calls, each paying the 100-gas warm-call surcharge the one-shot SHA-256 call above never pays even once: $E \approx (P/64) \times (100 + 12 + \text{marshal}) \approx 2-3P$, comparable to or worse than the SHA-256 figure, not better. Either way $E \ll 24B = 48P$: the floor binds regardless of which precompile hashes the file, at $\text{gasUsed} = 40 \times 2P = 80P$ – exactly double option B, purely because of the doubled calldata, not because of anything about SHA-256 versus BLAKE2F.

D – FlatMerkle, built on-chain via BLAKE2F. Only the original P bytes are ever submitted as calldata; the structured file exists solely in EVM memory. Treating BLAKE2b’s block size the way Section 6.2 treats real SHA-256’s (block size fixed, every item – data or checkpoint – occupies one full block): $b = 128$ bytes, $N = P/128$, $2N-1 \approx 2N = P/64$ compression calls, each 12 gas of pure precompile time ($E_{\text{pure}} = 0.1875P$). Accounting for the warm-call surcharge and rough argument-marshaling cost each of the $P/64$ separate calls actually pays – an engineering approximation, not a protocol constant – $E \approx (P/64) \times 135 \approx 2.109P$. Either figure is far under the 24-gas/byte threshold (an implementation would need to be roughly *ten times* less efficient than this rough estimate before the floor stopped binding), so: $\text{gasUsed} = 40P$.

A.3 Rootstock: the same options, different protocol rules

Rootstock deliberately does not track Ethereum’s gas schedule change for change; RSKIP239 (Draft, unadopted, but authoritative on Rootstock’s *current*, pre-proposal costs, which is what we need here) says so directly: “it is not our goal to match Ethereum’s gas schedule. Indeed, the current gas schedule in Ethereum has evolved even further away from RSK’s [schedule], post EIP-2929” [28]. Concretely, as of this writing Rootstock has adopted EIP-2028’s calldata pricing (RSKIP400: 16 gas/non-zero byte, same as Ethereum [29]) and EIP-152’s BLAKE2F precompile (RSKIP153, same gas formula, 1 gas/round [27]), but – per RSKIP239’s own “current cost” column, **SLOAD** still at 200 gas – has *not* adopted EIP-2929’s Berlin repricing, and we found no Rootstock proposal resembling EIP-7623’s calldata floor at all. Two consequences follow, in opposite directions:

- No calldata floor means Rootstock’s transaction cost is the plain, unfloored sum $16B + E$ – there is no $\max(\cdot, 40B)$ waiting to absorb a cheap option’s hashing-cost advantage the way it does on Ethereum.
- No EIP-2929 means no warm-call discount for precompiles either: **STATICCALL** to a precompile still costs the flat, pre-Berlin **CALL** rate – confirmed directly against the reference client’s own gas-constants source, **CALL** = 700 in `GasCost.java` [30], with no cold/warm distinction defined anywhere in that file, cross-checking exactly against RSKIP239’s account. This is the same flat 700 that EIP-2929 itself replaced with the 100/2600 split on Ethereum [4]. This makes *repeated* precompile calls, not just hashing throughput, expensive on Rootstock in a way they are not on Ethereum.

SSTORE of a fresh 32-byte slot, with no cold-slot surcharge to add, stays at the plain **SET_SSTORE** = 20000 [30, 33] – 625 gas/byte, exactly the figure a pre-EIP-2929 estimate would give; the same source also confirms **SHA3** = 30, **SHA3_WORD** = 6 and **SHA256** = 60, **SHA256_WORD** = 12, identical to the Ethereum values Section A.2 uses.

Recomputing all four options against these rules (B , E , P as before, no floor – $\text{gasUsed} = 16B + E$ directly):

A. $E = (P/32) \times 20000 = 625P$; total = $16P + 625P = 641P$.

B. Unchanged hashing cost (1.3–2.4 P , Keccak256 is a native opcode on both chains, needing no call at all); total = $16P + (1.3\text{--}2.4)P \approx 17.3\text{--}18.4P$.

C. $B = 2P$ as before. One-shot SHA-256: $E \approx 0.75P$ (the flat per-call overhead is paid once, negligible for large P); total $\approx 32.75P$. Chained BLAKE2F: $E \approx (P/64) \times (700 + 12 + \text{marshal}) \approx 11.5P$; total $\approx 43.5P$ – here, without a floor to absorb it, using BLAKE2F instead of SHA-256 makes this option *more* expensive, not merely no cheaper, since Rootstock’s flat 700-gas call cost is paid $P/64$ times instead of once.

D. $B = P$, $E \approx (P/64) \times (700 + 12 + \text{marshal}) \approx 11.5P$; total $\approx 27.5P$.

Ranked cheapest to most expensive on Rootstock: $B (\approx 17.3\text{--}18.4) < D (\approx 27.5) < C (\approx 32.75\text{--}43.5) \ll A (641)$. This is not the ranking we expected going in, and it is worth stating plainly rather than smoothing over: on Rootstock, option D does *not* beat option B – removing the calldata floor does let raw hashing efficiency matter again, but Rootstock’s absence of a warm-call discount taxes option D’s $P/64$ separate BLAKE2F calls more than it costs option B’s zero calls (a native opcode, not a precompile) or even option C’s single call. What Rootstock’s numbers *do* show is that D is a clear, large improvement over C (off-chain construction) and over A (persistent storage) on any EVM chain we have checked, and that which of B or D is cheapest is not a fact about FlatMerkle or about the EVM in general – it is a fact about whether the specific chain discounts repeated precompile calls the way Ethereum has since Berlin, which a deployer needs to check for their actual target chain rather than assume.

A.4 Arbitrum: the missing combination

Ethereum has both a calldata floor and a warm-call discount (the discount just never gets to matter, since the floor already dominates); Rootstock has neither. The remaining combination – a warm-call discount *without* a calldata floor – is not hypothetical: it is Arbitrum’s current, documented configuration. Arbitrum One and Nova run Nitro, a modified Geth core that tracks Ethereum’s Istanbul/Berlin/London opcode and precompile pricing, including EIP-2929’s cold/warm split (2100/100 for **SLOAD**, the same split applied to precompile **STATICCALLS**); its own documentation, however, states plainly that EIP-7623 was *not* enabled as part of ArbOS 40 Callisto (Arbitrum’s Pectra-alignment upgrade, activated 2025) and remains unenabled as of ArbOS 51 Dia (activated January 8, 2026, current as of this writing), “because block size variance is less of a concern on Arbitrum chains” [22, 23]. This is precisely the case relevant to an Arbitrum Orbit L3 that wants to use Arbitrum One itself (rather than Ethereum L1) as its settlement/DA layer – an L2-native analogue of the Bitcoin rollup setting of Section 10, with Arbitrum playing the role of the underlying, published-to chain.

Recomputing with Ethereum’s own hashing/call constants (Arbitrum inherits them unchanged) but *no* EIP-7623 floor – $\text{gasUsed} = 16B + E$, as on Rootstock, but with 100-gas warm precompile calls, as on Ethereum:

A. Same as Ethereum: $706.625P$.

B. Unchanged: $16P + (1.3\text{--}2.4)P \approx 17.3\text{--}18.4P$.

C. $B = 2P$; one-shot SHA-256 $\approx 32.75P$; chained BLAKE2F $E \approx (P/64) \times (100 + 12 + \text{marshal}) \approx 2.1P$, total $\approx 34.1P$ – still no cheaper than one-shot SHA-256, for the same reason as before (many calls, even discounted ones, against one).

D. $B = P$, $E \approx (P/64) \times 135 \approx 2.1P$ (realistic) or $0.1875P$ (pure precompile); total $\approx 18.1P$, or as low as $16.1875P$ under the optimistic estimate.

Here, unlike Rootstock, option D is at least competitive with option B and plausibly a little cheaper ($16.2\text{--}18.1P$ against $17.3\text{--}18.4P$), rather than clearly worse – though the two ranges overlap enough, given how approximate the “glue code” estimate is, that we would not call this a decisive win either. What removing the floor while *keeping* the discount does establish is directional: it moves D from a clear loss (Rootstock) to a close contest (Arbitrum), consistent with the mechanism identified above – the discount is what lets many cheap calls compete with zero calls at all, and only shows up once nothing else swamps it.

A.5 Two narrower points, and the general lesson

FlatMerkle’s case here rests on two narrower points, not on beating a plain Merkle tree’s cost on every chain. First, the SHA-256 precompile’s one-shot, arbitrary-length interface cannot build a structured file incrementally with a custom starting state the way BLAKE2F’s raw, resumable compression-function interface can (this is exactly option C’s cost penalty above); BLAKE2F is, as far as we are aware, the only precompile available on any of the three chains that exposes what FlatMerkle’s construction actually needs, so a rollup that wants the structured file built entirely on-chain – with no dependence on an off-chain party supplying it correctly, and no fraud-proof step to check that they did – has option D as close to its only route to that property, not merely its cheapest one. Second, a rollup that also publishes a redundant or backup commitment via the Bitcoin annex application of Sections 9–10 can reuse a single flat, linear proof format across all four venues (Bitcoin, Ethereum, Rootstock, Arbitrum); a Keccak Merkle tree is not that format on any of them, and would need its own, separate verifier per venue.

More generally: whether option B or D is cheapest is not a fact about FlatMerkle, nor about “the EVM” as a single target – it is a fact about two *separable* protocol design choices a given chain may or may not have made: a calldata floor (EIP-7623-style, which erases the gap between B and D by making both floor-bound) and a discount for repeated precompile calls (EIP-2929-style, which is what lets D’s many small calls compete with B’s zero calls at all once there is no floor to erase the difference). Ethereum currently has the first, and the second without it mattering (it applies, but the floor already dominates); Rootstock has neither; Arbitrum has the second without the first, which is the one combination among the three where option D is not clearly worse than option B. A rollup deployer targeting a different EVM-compatible chain should check both properties directly rather than assume any of these three answers transfers.

Two further things this analysis does not cover, both flagged rather than resolved. First, the exact per-call “glue code” cost: D’s ≈ 135 gas/call on Ethereum and Arbitrum and ≈ 737 on Rootstock are estimates for a reasonably careful Solidity or Yul implementation, not measured or protocol-fixed numbers (on Ethereum the margin to the floor threshold is wide enough that this does not change the conclusion). Second, every constant used here is a protocol *parameter*, and several are about to move. Ethereum’s next fork, Glamsterdam, was activated on the Sepolia testnet in October 2026 and has no mainnet date as of this writing; its draft repricing set [31] raises the calldata floor from 10/40 to 64/64 gas per byte (EIP-7976), which would make the floor bind whenever $E < 48B$, putting options B and D at $64P$ and option C at $128P$ – the same ranking at higher absolute cost – and raises the cost of creating a storage slot to 97,920 state gas (EIP-8037), making option A several times more expensive still. The draft’s numbers are marked preliminary. Further out, EIP-7667, if ever adopted, would raise BLAKE2_GFROUND from 1 to 10 and the SHA-256/Keccak costs by roughly 5–10 $\times$, moving options B, C and D toward the floor threshold; RSKIP239, if adopted, would not touch the costs of Section A.3, but shows that Rootstock’s schedule is also a moving target. The tables are the correct computation against the rules in force at the time of writing, not a permanent ranking.

As illustrative, non-binding capacity figures: since Fusaka (December 2025), EIP-7825 caps every Ethereum transaction at $2^{24} = 16,777,216$ gas [25], so a single floor-bound transaction (options B or D) carries at most $(2^{24} - 21,000)/40 \approx 419,000$ bytes, and filling a ≈ 60 -million-gas block, about 1.5 million bytes, takes several transactions. At Rootstock’s 6.8 million gas block limit [26], option B’s ≈ 17.3 – 18.4 gas/byte allows roughly 370–390 thousand bytes. At Arbitrum’s 32 million gas per-transaction cap (ArbOS 51 Dia, child-chain execution gas only, excluding any separate parent-chain data fee) [23], option D’s ≈ 16.2 – 18.1 gas/byte allows roughly 1.8–2.0 million bytes in a single transaction. All three figures ignore every other use of the same block’s or transaction’s gas.

References

- [1] Alexey Akhunov, Eli Ben Sasson, Tom Brand, Louis Guthmann, and Avihu Levy. EIP-2028: Transaction data gas cost reduction. Ethereum Improvement Proposals, no. 2028, 2019. <https://eips.ethereum.org/EIPS/eip-2028>. Reduces the calldata cost of a non-zero byte from 68 to 16 gas; unchanged since.
- [2] Ahto Buldas, Peeter Laud, Helger Lipmaa, and Jan Villemson. Time-stamping with binary linking schemes. In *Advances in Cryptology – CRYPTO ’98*, volume 1462 of *Lecture Notes in Computer Science*, pages 486–501. Springer, 1998. doi: 10.1007/BFb0055749.
- [3] Ahto Buldas, Helger Lipmaa, and Berry Schoenmakers. Optimally efficient accountable time-stamping. In *Public Key Cryptography – PKC 2000*, volume 1751 of *Lecture Notes in Computer Science*, pages 293–305. Springer, 2000. doi: 10.1007/978-3-540-46588-1_20.
- [4] Vitalik Buterin and Martin Swende. EIP-2929: Gas cost increases for state access op-codes. Ethereum Improvement Proposals, no. 2929, 2020. <https://eips.ethereum.org/EIPS/eip-2929>. Live since the Berlin upgrade (April 2021); introduces the cold/warm access-cost distinction used for `SSTORE` and precompile calls in Section 11.
- [5] Vitalik Buterin, Dankrad Feist, Diederik Loerakker, George Kadianakis, Matt Garnett, Mofi Taiwo, and Ansgar Dietrichs. EIP-4844: Shard blob transactions. Ethereum Improvement Proposals, no. 4844, 2022. <https://eips.ethereum.org/EIPS/eip-4844>. Live since the Dencun upgrade (March 2024); defines blob transactions and their ~ 18 -day (4096-epoch) data retention window.
- [6] Scott A. Crosby and Dan S. Wallach. Efficient data structures for tamper-evident logging. In *Proceedings of the 18th USENIX Security Symposium*, 2009.
- [7] Hans Dobbertin. Cryptanalysis of MD5 compress, 1996. Announced at the rump session of Eurocrypt ’96; exhibits free-start collisions for the MD5 compression function years before any collision was known for MD5 itself.
- [8] Michael T. Goodrich, Roberto Tamassia, and Andrew Schwerin. Implementation of an authenticated dictionary with skip lists and commutative hashing. In *Proceedings DARPA Information Survivability Conference and Exposition II (DISCEX ’01)*, volume 2, pages 68–82. IEEE, 2001.
- [9] Jian Guo, Pierre Karpman, Ivica Nikolić, Lei Wang, and Shuang Wu. Analysis of BLAKE2. Cryptology ePrint Archive, Paper 2013/467 (CT-RSA 2014), 2013. URL <https://eprint.iacr.org/2013/467>.
- [10] Stuart Haber and W. Scott Stornetta. How to time-stamp a digital document. *Journal of Cryptology*, 3(2):99–111, 1991.
- [11] Tjaden Hess, Matt Luongo, Piotr Dyraga, and James Hancock. EIP-152: Add BLAKE2 compression function F precompile. Ethereum Improvement Proposals, no. 152, 2016. <https://eips.ethereum.org/EIPS/eip-152>. Live since the Istanbul upgrade (December 2019); 1 gas per round, 12 rounds for standard BLAKE2b.
- [12] Harry Kalodner, Steven Goldfeder, Xiaoqi Chen, S. Matthew Weinberg, and Edward W. Felten. Arbitrum: Scalable, private smart contracts. In *27th USENIX Security Symposium*, 2018.

- [13] Ben Laurie, Adam Langley, and Emilia Kasper. Certificate transparency. RFC 6962, IETF, 2013. Defines the Merkle tree convention (distinct leaf/internal hash prefixes, leftmost-complete-subtree handling of non-power-of-two leaf counts) referenced in Section 11.
- [14] Sergio Demian Lerner, Javier Álvarez Cid-Fuentes, Julian Len, Ramsès Fernández-València, Patricio Gallardo, Nicolás Vescovo, Raúl Laprida, Shreemoy Mishra, Federico Jinich, and Diego Masini. RSK: A Bitcoin sidechain with stateful smart-contract functionality. Cryptology ePrint Archive, Paper 2022/684, 2022. <https://eprint.iacr.org/2022/684>.
- [15] Sergio Demian Lerner, Ramon Amela, Shreemoy Mishra, Martin Jonas, and Javier Álvarez Cid-Fuentes. BitVMX: A CPU for universal computation on Bitcoin. arXiv:2405.06842, 2024. <https://arxiv.org/abs/2405.06842>.
- [16] Sergio Demian Lerner, Martin Jonas, and Ariel Futoransky. ESSPI: ECDSA/Schnorr signed program input for BitVMX. arXiv:2503.02772, 2025. <https://arxiv.org/abs/2503.02772>.
- [17] Yingxin Li, Fukang Liu, and Gaoli Wang. New records in collision attacks on SHA-2. Cryptology ePrint Archive, Paper 2024/349, 2024. URL <https://eprint.iacr.org/2024/349>. First practical semi-free-start collision for 39-step SHA-256 (of 64).
- [18] Robin Linus Woll, Ioannis Alexopoulos, Lukas Aumayr, Zeta Avarikioti, Matteo Maffei, and David Tse. BitVM3: Efficient Bitcoin bridges via garbled circuits. Cryptology ePrint Archive, Paper 2026/933, 2026. <https://eprint.iacr.org/2026/933>. Supersedes an earlier RSA-based BitVM3 construction retracted after a security break (documented separately in ePrint 2025/1291); this citation refers to the current garbled-circuit-based version.
- [19] Mohammad Mahmoody, Tal Moran, and Salil Vadhan. Publicly verifiable proofs of sequential work. In *Proceedings of the 4th Conference on Innovations in Theoretical Computer Science (ITCS)*, 2013.
- [20] Petros Maniatis and Mary Baker. Authenticated append-only skip lists. *arXiv preprint arXiv:cs/0302010*, 2003.
- [21] Ralph C. Merkle. Protocols for public key cryptosystems. In *Proceedings of the 1980 IEEE Symposium on Security and Privacy*, pages 122–134. IEEE, 1980. Introduces the Merkle tree / hash tree construction.
- [22] Offchain Labs. ArbOS 40 Callisto. Arbitrum Docs, ArbOS release notes, 2025. <https://docs.arbitrum.io/run-arbitrum-node/arbos-releases/arbos40>. Activated 2025 (Arbitrum’s Pectra-alignment upgrade); states EIP-7623 is intentionally not enabled on Arbitrum chains, “because block size variance is less of a concern on Arbitrum chains.”.
- [23] Offchain Labs. ArbOS 51 Dia. Arbitrum Docs, ArbOS release notes, 2026. <https://docs.arbitrum.io/run-arbitrum-node/arbos-releases/arbos51>. Activated January 8, 2026; confirms EIP-7623 remains unenabled on Arbitrum One/Nova, introduces `gasFloorPerToken` only for Arbitrum chains posting to an EIP-7623-enabled parent chain, and sets a 32,000,000 child-chain-execution-gas per-transaction cap.
- [24] William Pugh. Skip lists: A probabilistic alternative to balanced trees, 1990. *Communications of the ACM*, 33(6), 668–676.

- [25] Giulio Rebuffo and Toni Wahrstätter. EIP-7825: Transaction gas limit cap. Ethereum Improvement Proposals, 2024. URL <https://eips.ethereum.org/EIPS/eip-7825>. Caps each transaction at 2^{24} gas; activated on mainnet with Fusaka, 3 December 2025.
- [26] Rootstock Developers Portal. rBTC gas fees: Optimizing transaction costs. Web page, last updated June 2026, 2026. <https://dev.rootstock.io/concepts/rbtc/gas/>. States the current transaction gas limit per block is 6,800,000 units.
- [27] Rootstock Improvement Proposals. RSKIP153: Add BLAKE2 compression function F precompile. Status: Adopted, 2020. <https://ips.rootstock.io/IPs/RSKIP153.html>. Adds the EIP-152-compatible BLAKE2F precompile (same 1-gas-per-round pricing) to Rootstock, active since the Iris hardfork.
- [28] Rootstock Improvement Proposals. RSKIP239: Reprice trie read opcodes. Status: Draft (unadopted), 2021. <https://ips.rootstock.io/IPs/RSKIP239.html>. Authored by S. D. Lerner and S. Mishra; its “current cost” column and rationale document Rootstock’s live, pre-EIP-2929 gas schedule (as of writing, and since the proposal itself remains unadopted, presumably still), stating directly that matching Ethereum’s post-Berlin schedule is not Rootstock’s goal.
- [29] Rootstock Improvement Proposals. RSKIP400: Calldata gas cost reduction. Status: Adopted, 2023. <https://ips.rootstock.io/IPs/RSKIP400.html>. Implements EIP-2028’s 16-gas/non-zero-byte calldata pricing on Rootstock.
- [30] RSK Labs / RootstockLabs. GasCost.java, RSKj reference client. Source file, `rsksmart/rskj`, 2026. <https://github.com/rsksmart/rskj/blob/master/rskj-core/src/main/java/org/ethereum/vm/GasCost.java>. Authoritative current opcode/precompile gas constants: CALL=700, SET_SSTORE=20000, SLOAD=200, SHA3=30/SHA3_WORD=6, SHA256=60/SHA256_WORD=12, no cold/warm access-cost constants defined – consistent with RSKIP239’s account.
- [31] Maria Silva and Ansgar Dietrichs. EIP-8007: Glamsterdam gas repricings. Ethereum Improvement Proposals, Draft (Meta), 2025. URL <https://eips.ethereum.org/EIPS/eip-8007>. Directory of Glamsterdam repricing EIPs, incl. EIP-7976 (calldata floor 10/40 $\rightarrow$ 64/64 gas per byte), EIP-8037 (state-creation gas), EIP-7904 (no compute/precompile repricing). Numbers marked preliminary. Accessed October 2026.
- [32] Marc Stevens, Pierre Karpman, and Thomas Peyrin. Freestart collision for full SHA-1. IACR Cryptology ePrint Archive, Report 2015/967 (“The SHAppening”), 2015. Exhibits a free-start collision for the full SHA-1 compression function in 2015, two years before the first ordinary SHA-1 collision (SHAttered, 2017).
- [33] Wei Tang. EIP-2200: Structured definitions for net gas metering. Ethereum Improvement Proposals, no. 2200, 2019. <https://eips.ethereum.org/EIPS/eip-2200>. Source of SSTORE_SET_GAS= 20000, unchanged by later repricings.
- [34] Peter Todd. Merkle mountain ranges. OpenTimestamps project documentation / mailing-list writeup; no formal peer-reviewed publication, 2016. Generalizes Merkle trees to an efficiently-appendable forest of perfect binary trees.
- [35] Toni Wahrstätter and Vitalik Buterin. EIP-7623: Increase calldata cost. Ethereum Improvement Proposals, no. 7623, 2024. <https://eips.ethereum.org/EIPS/eip-7623>. Live since

the Pectra upgrade (May 2025); introduces the 40-gas/byte calldata floor for data-heavy, low-execution transactions used in Section 11.

- [36] Pieter Wuille, Jonas Nick, and Anthony Towns. BIP 341: Taproot: SegWit version 1 spending rules. <https://github.com/bitcoin/bips/blob/master/bip-0341.mediawiki>, 2020. Defines the annex field and its sha₂₅₆ commitment.