Verifiable Data on Bitcoin: FlatMerkle Commitments, Refutable Proofs, and FlatChecks

Sergio Demian Lerner
Fairgate Labs Rootstock Labs
sergio@fairgate.io*

November 1, 2026

Abstract

We give a method to publish data on Bitcoin and to verify properties of that data on Bitcoin itself, against a publisher who may lie. The data is a FlatMerkle file – a structured byte string whose ordinary SHA-256 supports logarithmic inclusion proofs and logarithmic proofs of malformation – placed in a transaction's annex or, at four times the cost but relayable by Bitcoin Core, in an output. We show that the publisher's ordinary Taproot signature, with a suitably chosen signature-hash type, binds the publisher to the published data in a form that a dispute can authenticate with a single SHA-256 compression plus the signature check, or with three compressions when no annex is used. Claims about the data, and claims about such claims, are published as proof messages in two formats: a plain encoding, refuted by revealing it whole, and a traced encoding that stores the verifier's output after each level, so that any false message is refuted by revealing at most five of its elements. The format, chosen by a cost model that weighs challenge bytes against published bytes, is fixed by public context so that no message can be read in two formats; chains of claims about claims settle at four logical elements. A dispute thus reduces to a few dozen SHA-256 compressions and one Schnorr verification per commitment – too large for Bitcoin Script, but small enough to adjudicate with an interactive bisection game, a chunked script in the manner of BitVM2, or a garbled circuit. The same technique verifies any *locally refutable* verifier, one whose every rejection is witnessed by a single check over a few words of a canonical trace. In a *FlatCheck*, the prover publishes only the proof; the trace, the verifier's intermediate values, is published only if a challenger asks for it, and the dispute then examines the single check he names. When the verifier splits into groups that each read a few parts of the proof, a challenge forces the trace of one group only. We give two examples: a toy L2 of signed transfers, whose 512-transfer batches cost 12,200 vbytes to claim, and a Bitcoin header-chain light client verified on Bitcoin itself, which claims a retarget period of 2,016 headers in 45,200 vbytes; a dispute over a local check costs about 37,500 and 49,200 vbytes, and one over a malformed trace about 77,100 and 99,800, independently of the number of groups. Verifying STARK proofs is another application. We also remove FlatMerkle's restriction to power-of-two sizes without padding.

*This paper and its reference implementation were prepared with the assistance of Anthropic's Claude (Sonnet 5.5 and Opus 5.5) as an AI writing and coding assistant, and reviewed by GPT-6 Astra as an AI reviewer. All claims about the constructions were verified against the executable reference implementation described herein. *Competing interests:* the author is affiliated with Fairgate Labs and Rootstock Labs. BitVMX, its garbled-circuit module, OHMG and WISCH, discussed in Section 7, and FLEX and its formal analysis, used in Section 8, were developed by or with the support of these organizations, and the author co-authored BitVMX, OHMG, WISCH and FLEX.

Contents

1 Introduction **2**

2 Background: FlatMerkle in brief **4**

3 Publishing data with a one-compression commitment **5**
3.1 Reading the layout from the message length . . . 5
3.2 The commitment and its single compression . . . 6
3.3 An annex-free output profile . . . 6
3.4 Security and limits . . . 9
3.5 Size limits and relay . . . 11

4 FlatMerkle for any number of elements **12**

5 Proof messages **14**
5.1 Two message formats . . . 14
5.2 Choosing the format, and binding it . . . 16

6 Recursion and its limit **17**

7 Verifying claims on Bitcoin **19**
7.1 What a dispute computes . . . 19
7.2 Why not Script . . . 21
7.3 Interactive bisection . . . 21
7.4 Chunked Script, without garbled circuits . . . 21
7.5 Garbled circuits . . . 22

8 FlatCheck: verifying locally refutable proofs **25**
8.1 Definitions . . . 25
8.2 The trace published with the proof . . . 26
8.3 The trace on demand . . . 27
8.4 Grouped publication . . . 31
8.5 Costs of the components . . . 33
8.6 Two examples . . . 35
8.7 Several challengers . . . 40

9 Related work **47**

10 Conclusion **48**

1 Introduction

Many protocols built on Bitcoin need two things that Bitcoin does not provide directly: a place to publish a substantial amount of data so that everyone can see it, and a way to verify properties of that data on Bitcoin itself – to pay out, or to penalize, according to what the data says. A rollup must publish its transactions and let anyone dispute an incorrect state transition; a bridge or an oracle must commit to values that others can later prove wrong. In each case the publisher may be

the adversary, so the method must let an honest party respond to any false publication at a cost that Bitcoin can bear.

This paper assembles such a method from four parts.

Publishing. The data is placed in a transaction's *annex* as a FlatMerkle file [23]: a byte string whose ordinary SHA-256, the annex hash Bitcoin already computes, supports logarithmic inclusion proofs and logarithmic proofs that the file is malformed. The threat model is that of [23, Section 1]: for every byte string a publisher might publish, an honest opponent must be able to prove in logarithmic space either the inclusion of any element or the malformation, since a malicious publisher will pick whichever is harder. Here the guarantee holds for every publication of the size the context fixes (Section 3.3).

Binding the data to the publisher. The publisher's Taproot signature commits to the annex hash, but recomputing the signature hash costs several SHA-256 compressions. We show that, for a key-path signature whose signature-hash input is exactly 222 bytes long, the last 30 bytes of the published data's hash start on a block boundary, so the signature hash is one compression away from a 30-byte commitment and a fixed padding block. Exactly two hash types produce that length, one committing to the annex and one to an output, so the same check works whichever the publisher used.

Claims about the data, and claims about claims. A claim about published data – with its proof – can itself be published, and must then be refutable too. We give two proof-message formats: a plain one, refuted by revealing it whole, and a traced one that stores the verifier's output after each level, so that a false message is refuted by revealing at most five of its elements. A cost model chooses between them, the choice is bound to public context so that no message can be read in two ways, and nested claims shrink along the iterated logarithm.

Adjudicating on Bitcoin. Every dispute then reduces to a few dozen SHA-256 compressions and one Schnorr verification per commitment. That is far too large for Bitcoin Script, where a single compression takes about 300 kB of script, or 75,000 virtual bytes [17, 18], but small enough to adjudicate with an interactive bisection game, as in BitVMX, as a chunked script, as in BitVM2, or with a garbled circuit, as in BitVM3, Glock, BABE and OHMG-based designs.

Contributions

- **A one-compression publication commitment** (Section 3). A Taproot key-path signature whose signature-hash input is 222 bytes long, which two hash types produce, commits to a 30-byte truncated hash of data published in the annex or in an output, authenticated by one SHA-256 compression plus the signature check; an annex-free 190-byte variant, which Bitcoin Core relays, needs three. We show how to lay a FlatMerkle file out in either place, with the initial state after the prefix derived from public data, so that the same proofs and fraud proofs verify against the commitment.
- **FlatMerkle for any number of elements** (Section 4). An uneven, RFC 6962-style split removes the original restriction to powers of two without padding: the file keeps $2N-2$ items, proofs have the same worst-case number of reveals as padding, and the construction coincides with the original on powers of two.

- **Two proof-message formats, a cost model, and a binding format rule** (Section 5). A false traced message is refuted by revealing at most five elements in one multiproof, in $O(\log \log N)$ for a claim about an N-element file. Under a cost in which on-chain challenge bytes weigh about 200 times as much as published bytes, traced messages win for large files and plain ones for most small files; header binding, disjoint check sets and context checks prevent encoding attacks.

- **Recursion and its limit** (Section 6). Chains of claims about claims settle at four logical elements (448 published bytes); for any file a standard-weight transaction can carry, a proof of a proof already has that size.

- **Adjudication on Bitcoin** (Section 7). We state the dispute computation, measure its SHA-256 cost, and explain how interactive bisection, chunked Script and garbled-circuit protocols adjudicate it.

- **FlatCheck: verifying locally refutable proofs** (Section 8). We define *locally refutable* verifiers, whose every rejection is witnessed by one check over a few words of a canonical trace, a structured case of naysayer proofs [34], and *separable* ones, which split into groups that each read a few of the proof's parts. A FlatCheck commits the trace as a FlatMerkle file and adjudicates one check on Bitcoin, with the accused bound to answer and the challenger supplying the authentication of the confirmed publication, through publication slots that accept every signature hash type. The trace need not be published with the proof: the proof alone is published, and the trace only if challenged, bound to it by the hash chain of its annex hash, in at most six transactions with an on-demand bond. For a separable verifier, the proof's parts are published once each, one annex per input, a challenge forces only the trace of one group and the parts it reads, and each check authenticates at most one published part. With several independent challengers, the claim and each group's trace are shared, and only the adjudication is duplicated. We give two examples, both state machines made separable by publishing the state at every group boundary, in blocks, so that the cost of a dispute does not grow with the batch: a toy L2 of signed transfers, and a light client that verifies a chain of Bitcoin headers – proof of work, retargeting, timestamps and cumulative work – on Bitcoin itself. Verifying STARK proofs is another application.

The reference implementation is in the same repository as the original FlatMerkle paper's, `https://github.com/SergioDemianLerner/FlatMerkle`; the code that paper describes is preserved under the tag `v1.0`, and its current text under `v1.1`. Every number in this paper was measured with it, including signature messages built from BIP341 and signatures checked with a BIP340 reference implementation.

2 Background: FlatMerkle in brief

We recall only what later sections use; full definitions and proofs are in [23], cited below by item number.

The structured file. Let $\mathcal{C}$ be the compression function of a Merkle–Damgård hash with w-bit state and b-bit blocks (for SHA-256, 32 and 64 bytes), and pad_b a fixed injective encoding of a state into a block [23, Definition 2.1]. For $N = 2^L$ data elements, $\mathrm{block}(a, k)$ covers elements

$a, \ldots, a + 2^k - 1$:

$$\begin{aligned} \mathrm{block}(a, 1) &= D[a] \,\|\, D[a+1] \,\|\, \mathrm{MID}(a), \\ \mathrm{block}(a, k) &= \mathrm{block}(a, k-1) \,\|\, \mathrm{block}(a+2^{k-1}, k-1) \,\|\, \mathrm{MID}(a), \qquad k \geq 2, \end{aligned}$$

where $\mathrm{MID}(a) = \mathrm{pad}_b(\mathrm{CK}[a])$ re-injects the *checkpoint* $\mathrm{CK}[a]$, the chaining state when element a was first reached. The file is $\mathrm{block}(0, L-1) \| \mathrm{block}(2^{L-1}, L-1)$, with $2N - 2$ items, and its digest Δ is the ordinary hash of these bytes [23, Section 2].

Proofs. A proof for index t consists of the pair d_0, d_1 containing t and $L - 1$ reveals, one per level of the recursion above the pair [23, Section 3]. At a level where t is in the right child, the reveal is the left sibling's closing state, which becomes the checkpoint of the right child; where t is in the left child, it is the right sibling's state just before its closing MID, and the verifier completes that closing with the value it has just computed. VERIFY [23, Algorithm 3.3] folds these values bottom-up into the file's final chaining state and compares its finalization with Δ. The shape of this computation – which branch at each level, which value feeds which compression call – depends only on (N, t).

Security. Under free-start collision resistance of $\mathcal{C}$ [23, Assumption 5.1], an accepting proof binds: no digest accepts two different values for the same index [23, Theorem 5.9], even for a file that was not built canonically. A non-canonical file may, however, accept a value other than the bytes physically at that position [23, Remark 5.10], so FlatMerkle adds *fraud proofs*: anyone holding the whole file can find a malformed checkpoint in $O(N)$ and prove it in $O(\log N)$ [23, Section 5.5], with soundness requiring that mismatches count only at *anchored* levels [23, Definition 5.5, Theorem 5.12] and completeness following from targeting the violation with the largest start [23, Theorem 5.17]. A *multiproof* proves several elements at once, sharing common reveals [23, Section 7], and is as sound as single proofs [23, Lemma 7.1].

3 Publishing data with a one-compression commitment

To verify a property of published data on Bitcoin, a dispute must first establish *which* data was published, from something a Bitcoin verifier can check. In [23] the data is a FlatMerkle file placed in a transaction's annex, and its digest is the annex's own hash, sha_annex [23, Section 9]. That hash is bound to the publisher through the transaction's Taproot signature, which signs a message containing it [44]. The difficulty is cost: the signed message, the *signature hash*, is the SHA-256 of about 270 bytes, and a dispute that recomputes it from scratch needs several compressions before it even reaches the signature check. This section gives a way to authenticate a 30-byte commitment to data in the annex with a single SHA-256 compression, plus the signature check itself, and an annex-free variant, for data in a standard output, with three.

3.1 Reading the layout from the message length

For a key-path spend, BIP341 signs $\mathrm{SHA256}(M)$ with

$$M = \mathrm{SHA256}(\texttt{TapSighash}) \,\|\, \mathrm{SHA256}(\texttt{TapSighash}) \,\|\, \texttt{00} \,\|\, \mathit{SigMsg}(\mathit{hash_type}),$$

where *SigMsg* contains, in a fixed order, the fields selected by *hash_type*, an optional sha_annex, and, for `SIGHASH_SINGLE`, the hash of the corresponding output, sha_single_output [44]. Writing A, Nn and X for the indicators of `ANYONECANPAY`, `SIGHASH_NONE` and an annex, the length of M is

$$B = 239 - 49A - 32\,Nn + 32X \text{ bytes.}$$

The SHA-256 padding encodes $8B$ in the last block, so a verifier that sees only the end of the computation still learns B, and with it which fields end the message. In particular $B = 222$ happens for exactly two hash types, both with an annex: `ALL|ANYONECANPAY` (`0x81`), whose message ends with sha_annex, and `SINGLE|ANYONECANPAY` (`0x83`), whose message ends with sha_single_output. We checked the formula for every key-path hash type, with and without an annex, against a direct implementation of BIP341.

3.2 The commitment and its single compression

Let D be the last 32 bytes of M when $B = 222$: D = sha_annex for `0x81` and D = sha_single_output for `0x83`. D occupies bytes 190 to 221 of M, so its last 30 bytes start exactly at byte 192, a SHA-256 block boundary. We define the publication commitment as

$$C = D[2..31] = M[192..221],$$

and the last block of the SHA-256 computation is then built from C and constants alone:

$$F = C \,\|\, \texttt{80} \,\|\, \texttt{00}^{25} \,\|\, \texttt{00000000000006f0} \qquad (8B = 1776 = \texttt{0x6f0}).$$

Given the chaining state H_{192} after the first 192 bytes, the signed digest is

$$\text{sighash} = \mathcal{C}(H_{192}, F),$$

one compression call, and the publisher's BIP340 signature is checked against it [43]. A dispute therefore authenticates C with one SHA-256 compression and one Schnorr verification; the verifier never sees *hash_type* or the rest of M, and publisher and challenger agree on C whichever of the two hash types the publisher used.

Restricting the commitment to the annex. With `0x83`, C is the suffix of the hash of the output at the signed input's index, which the publisher may fill with an `OP_RETURN` of her choice. A protocol that accepts publications in the annex only can fix that output in its pre-signed graph, as a covenant: for instance an amount of zero and the script `OP_RETURN` alone, whose serialization is ten bytes – eight of amount, one of script length and the opcode. Its sha_single_output is then a known constant K, and the verifier, after its compression and the signature check, compares C with $K[2..31]$: if they are equal, the spend used `0x83` and C is not an annex hash; otherwise it used `0x81` and C is the suffix of sha_annex, unless an annex hashes to the last 240 bits of K, a preimage search of about 2^{240} work. The comparison costs about 240 AND gates in a garbled circuit. Any output that the graph fixes at that index serves as well as the `OP_RETURN`; in the slots of Section 3.4 the whole transaction is fixed anyway. A spend without an annex needs no such test: its message is not 222 bytes long, so its signature hash is not $\mathcal{C}(H_{192}, F)$ for any H_{192} and C, except through a free-start collision.

3.3 An annex-free output profile

Both 222-byte hash types require an annex, and Bitcoin Core does not relay annex-bearing spends (Section 3.5). A second profile avoids the annex: a key-path `SINGLE|ANYONECANPAY` (`0x83`) spend *without* an annex, whose message has $B = 190$ bytes and ends with D = sha_single_output, the hash of the output at the signed input's index, in bytes 158–189. The length alone no longer identifies the hash type: `ALL|ANYONECANPAY` without an annex also has $B = 190$, and ends with the spent input's `nSequence`. The verifier therefore starts from the public midstate after the two

tag hashes, which fill bytes 0–63, and receives bytes 64–157, 94 bytes from the epoch to the spent input's `nSequence`; it checks that the hash type byte is `0x83` and the spend type byte is 0 (key path, no annex), appends D, and finishes the hash:

$$\text{sighash} = \mathcal{C}\big(\mathcal{C}\big(\mathcal{C}(H_{64}, M[64..127]),\ M[128..157]\|D\|\texttt{80}\|\texttt{00}\big),\ \texttt{00}^{56}\|\texttt{00000000000005f0}\big),$$

three compressions, with H_{64} the public tag midstate and $8B = 1520 = \texttt{0x5f0}$. The whole message is authenticated from a public constant, so this step needs only ordinary collision resistance, and the commitment is all 32 bytes of D. The output costs four weight units per byte against the annex's one [30]; in exchange the transaction is standard (Section 3.5).

A FlatMerkle file in either place. The FlatMerkle file needs its items to start on SHA-256 block boundaries [23, Section 9.5], so in both places the hashed byte string is laid out as one 64-byte *prefix block* followed by the file:

- *Annex* (`0x81` with an annex, whose final field is sha_annex). The hashed string is CompactSize(ℓ), with ℓ the annex length, followed by the annex. The annex consists of the mandatory byte `0x50`, then $63 - p$ zero bytes, then the file, where $p \in \{1, 3, 5, 9\}$ is the width of CompactSize(ℓ): the p length bytes, `0x50` and the zero filler make up the prefix block.
- *Output* (`0x83`, whose final field is sha_single_output whether or not an annex is present: without one this is the profile of Section 3.3; with one, the 222-byte check of Section 3 applies to the output, not to the annex). The hashed string is the serialized output (a `CTxOut`): an 8-byte amount, fixed to zero by the profile, then CompactSize(ℓ) with ℓ the script length, then the script, which consists of
  - `OP_RETURN` (1 byte), which makes the output provably unspendable;
  - $f = 50 - p$ filler bytes `OP_0`, each a one-byte empty push;
  - `OP_PUSHDATA4` and the file's length in 4 little-endian bytes;
  - the file itself, as the data of that single push.

  The amount, the p length bytes and the first $56 - p$ script bytes make up the prefix block, and the script after `OP_RETURN` consists of pushes only.

In both cases p depends on ℓ, which depends on p; the publisher takes the smallest p for which the resulting length is encoded in p bytes. Because the file's length is a multiple of 64, such a p always exists: $p = 1$ for files of up to 128 bytes, $p = 3$ up to about 64 KB, and $p = 5$ up to about 4 GB. Beyond that the annex layout uses $p = 9$, which only the theoretical range $N \le 2^{54}$ reaches; the output layout cannot, because its single `OP_PUSHDATA4` push holds at most $2^{32} - 1$ bytes, so it is limited to $N \le 2^{25}$, and larger outputs would need a different layout and its own framing analysis. The file then starts at byte 64 of the hashed string, and the total length is $64(2N - 1)$ bytes.

The initial state is derived, never supplied. The prefix block is a public function of N and of the profile, so the verifier computes the chaining state after it, $\text{CK}[0] = \mathcal{C}(\text{IV}, \textit{prefix}(N))$, itself, or hard-codes it when N is fixed; it plays exactly the role of IV in [23, Section 5], and the finalization counts the 64 prefix bytes. A prover must never supply CK[0]. For inclusion proofs a supplied value would be harmless, since every compression that consumes it binds it; but the fraud proofs of [23, Section 5.5] treat the top-left root checkpoint as anchored precisely because it is the fixed IV [23,

Definition 5.5], and with a supplied value that argument fails: on an honestly built eight-element file, an accusation for target 2 against the closing block of block$(0, 2)$, with the real closing block and a false CK[0], folds to the true digest and is accepted. With CK[0] derived, the argument of [23, Section 5.5] applies unchanged, and the same accusation is rejected. A claim is checked against the commitment by folding the proof, in the derived frame, to a digest d and comparing $d[2..31]$ with C in the annex profile, or d with D in the output profile.

Malformed prefixes. Deriving CK[0] leaves one case to cover: a validly signed publication whose prefix block is not the canonical one. Its body can be entirely consistent with the prefix actually published; then inclusion proofs in the derived frame fail for targets in the left half, and no fraud proof of [23, Section 5.5] accuses anything, because the discrepancy lies in the initial state, which those proofs take as given. A *malformation proof* covers this case and the ordinary one. The challenger reveals the 64-byte prefix block P' actually published and a fraud-proof-shaped fold – a target, its reveals, and the actual bytes of the checkpoint blocks it substitutes – in the frame that P' defines, and the proof is accepted if either

- P' is the canonical prefix and the fold is an accepted fraud proof of [23, Section 5.5] in the derived frame; or
- P' is not canonical, the fold reaches the digest from $\mathcal{C}(\mathrm{IV}, P')$, and either some substituted level shows a mismatch at a level anchored by the rule of [23, Definition 5.5] *without* its fixed-IV exception, or the initial state $\mathcal{C}(\mathrm{IV}, P')$ is itself anchored by that rule – the target lies in the left half and the top-left root's checkpoint feeds a compression that is not substituted.

A malformation proof is $O(\log N)$, like a fraud proof, plus 64 bytes. Both results below hold for publications of *valid size*: a hashed string of exactly $64(2N-1)$ bytes for the N fixed by the context, as [23, Theorem 5.17] assumes.

Theorem 3.1 (Malformation-proof soundness)**.** *Under Assumption 5.1 of [23], no malformation proof is accepted against a canonical publication – canonical prefix and canonical body.*

Proof. If the revealed prefix is canonical, this is [23, Theorem 5.12]. Otherwise the fold starts from $\mathcal{C}(\mathrm{IV}, P') \neq \mathrm{CK}[0]$ (equality is a collision of $\mathcal{C}(\mathrm{IV}, \cdot)$) and reaches the canonical file's digest. A mismatch at a level anchored without the fixed-IV exception is bound to the digest exactly as in [23, Theorem 5.12], whose argument for such levels never uses the exception, and contradicts the canonical body. If instead $\mathcal{C}(\mathrm{IV}, P')$ is anchored, it feeds a non-substituted compression on the path to the digest; the honest computation feeds CK[0] to the same position, and the two computations meet with equal outputs and different inputs, which by [23, Lemma 5.6] yields a free-start collision. □

Acceptance therefore establishes that the publication is *not canonical*; it does not authenticate the revealed prefix, nor locate the defect, just as for fraud proofs [23, Remark 5.18]. For instance, take a canonical four-element publication, compute the left half's ending x from another prefix P'' and other data, and replace the last checkpoint block by $\mathrm{pad}_b(x)$: a malformation proof revealing P'' and that other data, with the actual prewrap state of the right half, is accepted, although the published prefix is canonical. The publication is genuinely malformed; only the localization is wrong.

Theorem 3.2 (Malformation-proof completeness)**.** *Every non-canonical publication of valid size admits an accepted malformation proof.*

Proof. Let P' be the published prefix, and scan the body in the frame P' defines. If the scan finds a checkpoint violation, take as target the largest starting index a of a violation and substitute every violated checkpoint on its path. By [23, Theorem 5.17] the fold reaches the digest and the violation starting at the target is anchored; that proof anchors it through successive left branches down to the base case, which always consumes its checkpoint, so it does not use the fixed-IV exception and the proof is accepted in either case of the definition. If the scan finds no violation but P' is not canonical, take target 0 with no substitution: the ordinary fold reaches the digest, and every level on target 0's path is a left branch whose checkpoint is $\mathcal{C}(\mathrm{IV}, P')$, consumed by the base pair, so the initial state is anchored and the proof is accepted. Otherwise the publication is canonical. □

The reference watcher constructs the proof exactly this way, and `annex_commitment.py` asserts that the construction is accepted for every malformed publication it builds.

Publications of invalid size. A published string whose length ℓ is not $64(2N-1)$ for the context's N – truncated, extended, or of an unsupported size – is outside both theorems, and we give no logarithmic proof of a length violation. It can still be excluded when the publication is admitted, by requiring the publisher to answer one opening request for the context's N. An opening ends with $\mathcal{C}(s, Pad_N)$, while the publication's digest ends with $\mathcal{C}(s', B_\ell)$, whose final block encodes the length ℓ in its last eight bytes, so $B_\ell \neq Pad_N$; an opening that verifies against a wrong-length publication is therefore a free-start collision in the output profile, which compares full digests, and a truncated one – the general form of Assumption 3.3 – in the annex profile. A successful opening thus establishes the length, under that assumption. Alternatively, a protocol may establish the length independently and use only the fixed-size form of the assumption. Where the accused must answer anyway, as in Section 8, a wrong length simply makes it lose.

3.4 Security and limits

The assumptions. Everything a dispute proves about the published data rests on FlatMerkle's assumption, free-start collision resistance of SHA-256's compression function [23, Assumption 5.1]: it is hard to find two different pairs of chaining state and block that compress to the same output, when the adversary may choose both states. This is stronger than ordinary collision resistance, and it is needed for every FlatMerkle proof the dispute checks, inclusion or fraud proof, message or challenge: they supply their own intermediate states. The annex profile needs it once more for the signature hash, because H_{192} is supplied by the prover: two pairs $(H, C) \neq (H', C')$ with $\mathcal{C}(H, F(C)) = \mathcal{C}(H', F(C'))$ are a full-output free-start collision. And the annex profile compares only 30 bytes of a FlatMerkle digest with C, which full-output collision resistance does not cover, since the extraction argument of [23, Lemma 5.6] starts from equal outputs. For a file of N elements in a given frame, let

$$T_N(s) = \mathrm{suffix}_{240}\big(\mathcal{C}(s, Pad_N)\big),$$

where Pad_N is the single padding block of a $64(2N-1)$-byte message and s the chaining state before it.

Assumption 3.3 (Truncated free-start collision resistance)**.** *It is hard to find pairs* $(s, B) \neq (s', B')$ *of chaining state and block with* $\mathrm{suffix}_{240}\,\mathcal{C}(s, B) = \mathrm{suffix}_{240}\,\mathcal{C}(s', B')$.

The case the fixed-size theorems use is $B = B' = Pad_N$: states $s \neq s'$ with $T_N(s) = T_N(s')$. Two openings accepted against the same C for a file of the context's N then have the same state before finalization, and from there the full-output free-start argument of [23, Theorem 5.9] applies backward. The general form is needed only to check a publication's length (below), where the two

final blocks differ. Generic attacks cost about 2^{120} in either form. Distinct proofs of the same file reaching the same digest, as inclusion proofs for different indices do, are not collisions of either kind: they meet at an equal state. In the fixed-size theorems N and the profile are fixed by the context of each comparison.

The best published attack on SHA-256 in the free-start model is a practical semi-free-start collision on 39 of its 64 steps [23, Remark 5.2], and no free-start collision is known for the full compression function. The output profile compares all 32 bytes and authenticates its signature message from a public constant, so it needs neither the truncated assumption nor a free-start assumption for the signature hash. Finally, the theoretical range is bounded by SHA-256's message length: $64(2N-1)$ bytes must stay below 2^{64} bits, so $N \leq 2^{54}$, which the message headers of Section 5 enforce.

What the signature binds. A signature check authenticates that the key holder signed a digest, nothing more: a malicious key holder can choose any (H_{192}, C), or any 94-byte message start and D, and sign the result without building a transaction. The dispute program therefore establishes *that* a commitment carries the publisher's signature; *which* publication is being disputed must be fixed by the protocol, explicitly: the commitment is a setup parameter of the circuit, or a mechanism equates the commitment the circuit receives with that of the confirmed publication. Having the challenge transaction spend an output of the publication does not do this by itself, since the circuit still has to be given that publication's commitment and check against it. Likewise a signature shows that the publisher signed this variant of the transaction, not that it was confirmed [23, Section 9.4]. In the rollup setting both are enough when the protocol penalizes any signed invalid publication [23, Section 10.3].

Publication slots. When the publisher must answer a challenge, as in the FlatChecks of Section 8, neither is enough: if she supplied her own authentication data, she could answer against a variant of her publication that she signed but never broadcast. A mechanism that closes this gap uses the dispute protocol's pre-signed transaction graph. A *slot* is an output of that graph, owned by the publisher, which she spends by key path to publish. The graph fixes the transaction that spends it up to its witnesses: either its inputs are slots and its outputs are fixed, so that any other transaction would void the graph that pays the publisher, or the slot is spent together with a connector whose pre-signature is `SIGHASH_ALL`, which fixes every input and output. What the publisher still chooses, and what no script can restrict on a key-path spend, is the signature hash type, any of the seven that BIP341 allows, and whether the spend carries an annex. For every hash type, the signature-hash input of the fixed transaction is then a constant except for the 32-byte annex hash; the dispute circuit holds, per hash type, the midstate before the block containing the annex hash and the constant bytes around it, and rebuilds the signature hash in at most two compressions from the full annex hash h. A spend without an annex has a constant signature hash. Three rules complete the mechanism. The authentication data are supplied by the challenger, who copies the hash type, h (or "no annex") and the signature σ from the confirmed transaction, and the publisher's opening is checked against them; a copy that does not verify convicts no one. A verifying copy without an annex shows that the publisher spent her slot without publishing, and convicts her. And a second verifying copy for the same slot exists only if the publisher signed two spends of it, in which case either may be used against her: signing two spends of a slot is itself what loses. A slot costs at most four compressions with BIP340's challenge hash and 97 input bytes; since it authenticates all 32 bytes of the annex hash from public constants, it needs neither Assumption 3.3 nor a free-start assumption for the signature hash.

Limits. Both profiles require key-path spends: a tapscript signature adds a 37-byte extension after *SigMsg*, which moves the commitment out of the positions used here.

Testing. `annex_commitment.py` builds BIP341 signature messages for random transactions and checks: the length formula for every key-path hash type with and without an annex; that $B = 222$ occurs only for `0x81` and `0x83` with an annex, and $B = 190$ without an annex only for `0x81` and `0x83`; that bytes 192–221 are the last 30 bytes of sha_annex or sha_single_output and one compression from H_{192} reproduces the signature hash; that, with the publishing input's output fixed to an amount of zero and `OP_RETURN` alone, `0x81` spends give $C \neq K[2..31]$ and `0x83` spends $C = K[2..31]$, no spend without an annex has a 222-byte message, and without the fixed output a `0x83` spend makes C whatever its output hashes to (`annex_only.py`); that three compressions from the tag midstate reproduce the 190-byte signature hash, that the hash type check rejects a `0x81` message of the same length, and that BIP340 signatures verify from the dispute's inputs and fail for any other commitment; that files published in both places, in their derived frames, have the expected digests, and that every inclusion proof and multiproof verifies in that frame while wrong data or reveals are rejected; that, for every file size from 64 bytes to about 5 MB, the output layout is self-consistent and parses as `OP_RETURN` followed only by pushes; and that on an honest prefixed file no single-level fraud accusation is accepted with the derived CK[0], while a verifier that took CK[0] from the prover accepts the false accusation above. For malformation proofs it checks a four-element publication whose prefix has one filler byte changed and whose body is consistent with it (inclusion in the derived frame fails for indices 0 and 1, no fraud proof of [23, Section 5.5] is accepted, and a malformation proof is); that 840 malformation proofs against honest publications of 4, 8 and 16 elements – every target, every set of substituted levels, canonical and random revealed prefixes – are all rejected; and that each of 577 malformed publications – random non-canonical prefixes and corrupted checkpoint blocks for 4, 8 and 16 elements, and every subset of corrupted checkpoint blocks for 8 elements, with canonical and non-canonical prefix – admits the malformation proof that the proof of Theorem 3.2 constructs, which is accepted; that, for publications of 4, 8 and 16 elements with a block or a byte added or removed, no opening for the context's N verifies or matches the 30-byte commitment; that the output layout accepts $N = 2^{25}$ and rejects $N = 2^{26}$; and that the localization counterexample above is accepted although the published prefix is canonical. Every verifier entry point rejects, without raising, an index outside $[0, N)$, an N that is not a power of two at least 4, and proof components of the wrong width or count; `domain_checks.py` relabels valid proofs of the first and last pair with indices -1, N and much larger ones, which a verifier that only follows the index's path would accept. `end_to_end.py` runs complete disputes through both profiles: publication, signature, a claim message published in its own signed annex, one challenge, and the dispute program, under the format rule and with every message traced. It refutes 48 false claims and rejects 240 challenges against honest ones, and 240 more in which the challenger altered either of the two leading bytes of F's digest, which the annex commitment does not cover; and it upholds a malformation proof against signed annex publications with a non-canonical prefix.

3.5 Size limits and relay

A FlatMerkle file of N elements is hashed as $64(2N - 1)$ bytes, and it must fit in one transaction. By consensus a block weighs at most 4,000,000 weight units, with witness bytes, the annex included, weighing one unit and output bytes four [30]: the annex can hold a file of $N \leq 2^{14}$ elements, about 2 MB, or $N \leq 31{,}242$ with the uneven split of Section 4, and an output $N \leq 2^{12}$, or $N \leq 7{,}811$. Relay policy is tighter. Bitcoin Core rejects any Taproot spend that carries an annex [7], so neither 222-byte profile is relayed by Bitcoin Core, whatever the size of its outputs. It relays transactions

of at most 400,000 weight units and, since version 30.0, `OP_RETURN` outputs whose scripts total up to 100,000 bytes [6], so the annex-free output profile is standard for $N \le 2^9$, or $N \le 779$. For the annex, a proposal to make short unstructured annexes standard – annex data starting with `0x00`, every input carrying an annex, at most 256 bytes – was closed without merging [16]; Libre Relay adopts the same rule without the size limit [41]; and a 2026 proposal would reserve the form `0x50 0x00` $\langle bytes \rangle$ for unstructured data at the consensus level [26]. The annex layout above has that form, since `0x50` is followed by at least 58 zero bytes; through Libre Relay, under the 400,000-unit limit, it allows $N \le 2^{11}$, or $N \le 3{,}117$. Otherwise an annex transaction must reach a miner directly. Larger data must be split across transactions, each with its own commitment.

4 FlatMerkle for any number of elements

The construction of [23] requires N to be a power of two [23, Section 2], and leaves other sizes open [23, Section 13.3]. Padding the data up to the next power of two works, but can nearly double the file. A better option needs no padding at all and keeps proofs as short as padding's, by letting a block's two children have different sizes, in the style of the Merkle trees of RFC 6962 [22]. For a block of n elements starting at a, let $h(n)$ be the largest power of two strictly less than n, and define

$$\begin{aligned}
\mathrm{block}(a,1) &= D[a],\\
\mathrm{block}(a,2) &= D[a] \,\|\, D[a+1] \,\|\, \mathrm{MID}(a),\\
\mathrm{block}(a,n) &= \mathrm{block}(a,h) \,\|\, \mathrm{block}(a+h,\, n-h) \,\|\, \mathrm{MID}(a), \qquad n>2,\ h=h(n),\\
F &= \mathrm{block}(0,h(N)) \,\|\, \mathrm{block}(h(N),\, N-h(N)),
\end{aligned}$$

with $\mathrm{MID}(a) = \mathrm{pad}_b(\mathrm{CK}[a])$ exactly as before. (Here the second argument of block is a number of elements, not a level.) When n is a power of two, $h = n/2$ and this is the construction of [23, Section 2], so power-of-two files are unchanged. Left children always have power-of-two size at least 2; a lone element – at most one per file, the last one when N is odd – only ever occurs as a right child and carries no MID. The file always has exactly $2N-2$ items. For example, $N = 13$ gives

$$F = \mathrm{block}(0,8) \,\|\, \big[\, \mathrm{block}(8,4) \,\|\, D[12] \,\|\, \mathrm{MID}(8) \,\big].$$

Proofs and verification follow [23, Section 3] level by level, with one new case. When the target is in a left child whose right sibling is a lone element, that sibling has no closing MID to re-inject the verified value, so the proof reveals the 64-byte element e itself instead of a 32-byte prewrap, and the verifier computes $\mathcal{C}(\mathit{left_end}, e)$. When the target is the lone element, the proof carries that single element, and the left sibling's revealed ending enters the computation as the state operand of $\mathcal{C}(\mathit{ckpt}, e)$. Both are compression calls whose inputs the extraction argument of [23, Lemma 5.6] already tracks, and in the second the checkpoint is consumed directly, so it counts as anchored [23, Definition 5.5]. Fraud proofs carry over unchanged: a lone element has no MID and so cannot be a violation, and the max-a choice of target in [23, Theorem 5.17] works for the same reason as before, since every sibling met by a left branch lies after the target.

Table 1 compares the options. In the number of reveals, the uneven split equals padding for every N we checked (every N up to 2,048, plus those listed), while its file has $2N-2$ items instead of up to nearly twice as many. In bytes there is one small difference: when N is odd, a proof that meets the lone last element reveals it as a 64-byte element rather than a 32-byte state, so some proofs are up to 32 bytes longer than with padding; for even N they never are.

An alternative is to lay out one complete block per 1-bit of N, largest first, each closing with the previous block's ending, as Merkle mountain ranges [40] do with peaks. It allows growing the file

	Worst-case reveals			Items in the file	
N	Uneven split	Padding	Chained blocks	Uneven split	Padding
13	3	3	4	24	30
100	6	6	7	198	254
255	7	7	13	508	510
1,000	9	9	13	1,998	2,046
1,025	10	10	10	2,048	4,094
2,047	10	10	19	4,092	4,094
10^6	19	19	24	1,999,998	2,097,150

Table 1: Arbitrary N: worst case over all indices of the number of state reveals in a proof, and the number of 64-byte items in the file, for the uneven split described here, for padding to the next power of two, and for the flat chain of blocks discussed below (computed from its structure; not implemented).

by pure appends (two equal blocks merge by appending one closing MID), which the uneven split does not, but a proof must then fold in every later block one at a time, which nearly doubles the worst case (the "Chained blocks" column). For data published once, the uneven split is the better choice.

The construction is implemented separately from the power-of-two code, in `flatmerkle_any.py`, which leaves the implementation behind [23] unchanged. For every power of two from 4 to 256 it produces byte-identical files, digests and proofs, the two implementations accept each other's proofs, and its scanner and fraud proofs agree with those of [23, Section 5.5]. For every N from 3 to 80, every proof for every index verifies, and wrong data, a corrupted pair element or reveal, a missing or extra reveal, relabeling to the pair partner, and a wrong N are all rejected. For fraud proofs, all 21,004 accusations against canonical files are rejected, and all 400 randomly corrupted files (N from 3 to 70, up to four violations) yield an accepting fraud proof. A tampered reveal in a genuine fraud proof is accepted only when it is the checkpoint of a disputed, unanchored level – a value that feeds no compression call [23, Remark 5.13] – which happened in 20 cases; the file is then genuinely non-canonical, so accepting is correct, and the power-of-two implementation behaves the same way. The position formula of [23, Lemma 2.3] applies only to powers of two; for other N an element's position follows from the recursion above.

Remark 4.1 (Pages, left for future work)**.** *The base case need not be a pair. A* page *of K consecutive 64-byte data blocks, closed by one checkpoint block that re-injects the page's starting checkpoint, can replace it, with the binary construction above the pages unchanged; $K = 2$ is FlatMerkle itself. For N and N/K powers of two the body has $N + 2N/K - 2$ blocks, an opening reveals the target's page and the path above it, $64K + 32\log_2(N/K)$ bytes, and verification takes at most $K + 2\log_2(N/K) + 1$ compressions, finalization included. The element-size analysis of [23, Section 6] varies the element together with the block width, so a checkpoint block costs as much as an element and the file stays about twice the data; keeping the checkpoint block at 64 bytes instead brings the expansion to about $1 + 2/K$. At $N = 2^{14}$, $K = 4$ publishes 1,572,800 bytes instead of 2,097,088 (25% less) with the same 29 compressions and 640-byte instead of 544-byte openings, and $K = 8$ saves 37.5% for 31 compressions and 864-byte openings. With $K = \Theta(\log N)$ the body is $N + O(N/\log N)$ blocks, while openings and verification stay $O(\log N)$. A prototype, `paged_prototype.py`, checks inclusion proofs only – 864 openings for K from 2 to 16 and N from 32 to 128 match ordinary SHA-256 and the formulas above, and $K = 2$ reproduces FlatMerkle. Multiproofs, fraud and malformation proofs,*

the message formats and arbitrary page counts are not built. The page still consumes its starting checkpoint in its first compression, which is what anchors the lowest level of a fraud proof, but the anchoring argument must be redone; and a challenge then reveals whole pages, so the challenge bounds of Sections 7 and 8 do not carry over.

5 Proof messages

A FlatMerkle proof is itself data, and can be published inside another FlatMerkle file. Suppose Alice publishes a structured file F with $N_F = 2^L$ elements and digest Δ_F, and Bob publishes a second structured file M, with n elements and digest Δ_M, whose content is a claim about F together with its proof – for example, "element i of F is x", with an inclusion proof – typically in the annex of another transaction (Section 3). Under the threat model of Section 1, Bob may be the adversary, so a third party holding M must be able to refute a false M in logarithmic space. This section shows how; Section 6 applies it recursively.

A false M can be false in two ways. Its *layout* can be non-canonical; that is refuted exactly as for any structured file, by a fraud proof against Δ_M [23, Section 5.5], in $O(\log n)$. Or its layout can be canonical while its *content* is not a valid proof. How that second case is refuted depends on how the proof is encoded in M, and we use two encodings.

5.1 Two message formats

Both formats consist of 64-byte elements and start the same way: element 0 is a *header* holding Δ_F, L (one byte, at most 54, the bound SHA-256's message length imposes, Section 3.4), *index*, and a format tag, with every remaining byte fixed; elements 1 and 2 hold the proof's base pair d_0, d_1. Figure 1 shows the same claim in both formats.

Plain format. After the base pair come the proof's $L-1$ reveals, two per element, and zero filler. The message has about $L/2+3$ elements. To refute an invalid plain message, a challenger reveals all of it: the verifier rebuilds the structured file from the n elements, checks that it hashes to Δ_M, decodes the proof and runs VERIFY, which fails. This costs the whole message, $O(n) = O(\log N_F)$.

Traced format. To refute a large message without revealing it, we also store VERIFY's intermediate results. VERIFY [23, Algorithm 3.3] proceeds level by level, from the base pair up, and the shape of each level – whether the target is in its left or right child, and which reveal or earlier value it uses – depends only on (N_F, index), never on the proof's contents. There are L levels:

- level 0, the base pair: $e_0 = \mathcal{C}(\mathcal{C}(\mathcal{C}(ck, d_0), d_1), \mathrm{pad}_b(ck))$;
- level k, $1 \le k \le L-2$, when the target is in the left child: $e_k = \mathcal{C}(\mathcal{C}(r_k, \mathrm{pad}_b(e_{k-1})), \mathrm{pad}_b(ck))$; when it is in the right child: $e_k = \mathcal{C}(e_{k-1}, \mathrm{pad}_b(ck))$, and r_k is instead the checkpoint of the levels below;
- level $L-1$, the top split, followed by the finalization, whose result must equal Δ_F.

Here ck, a level's checkpoint, is F's derived initial state CK[0] (Section 3; IV for a standalone file) or the reveal of the nearest higher level whose target was in its right child. After the base pair, the traced message holds one element per level, $[r_k \mid e_k]$ (with r_0 and e_{L-1} left zero), then zero filler: about $L+3$ elements, roughly twice the plain message.

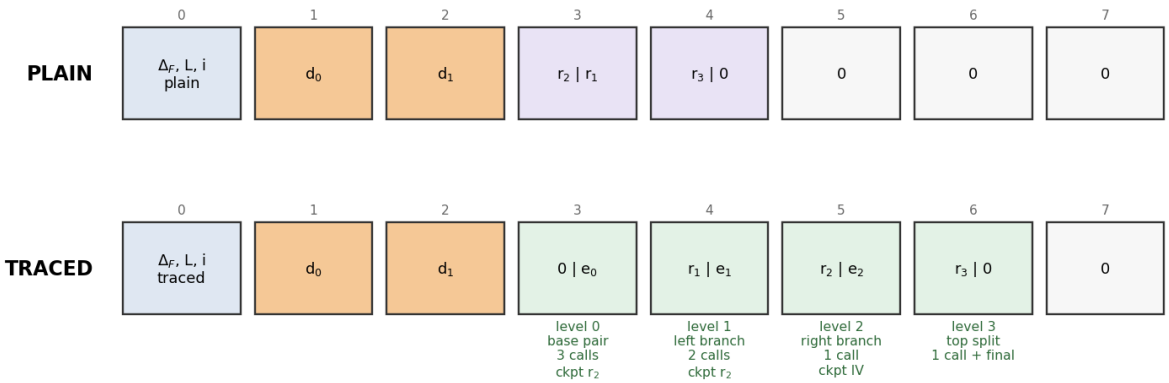


Figure 1: The same claim – element 5 of a 16-element file F, so $L = 4$ – in both message formats, drawn by the reference implementation. Each box is one 64-byte element; "$a \mid b$" holds two 32-byte values. r_k is the proof's reveal for level k of VERIFY, and e_k the value VERIFY has computed after level k. The plain format stores only the reveals, in the order the proof lists them. The traced format stores, for each level, that level's reveal and output, so each level can be checked on its own; the labels underneath give each level's shape, the number of compression calls it makes, and where its checkpoint comes from. The last level's output is not stored: the final check compares it, after finalization, with Δ_F.

The claim splits into *local checks*: the header (well formed and consistent with the context, Section 5.2); one check per level, which recomputes that level's one to three compression calls from the previous level's stored output, the level's own reveal, and its checkpoint, and compares the result with the stored e_k (or, for the last level, with Δ_F); and checks that every unused half-element and filler element is zero.

Lemma 5.1 (Local refutability)**.** *Let M be a traced message of the prescribed length n. All local checks pass if and only if (i) the header and all unused half-elements and filler elements satisfy the format rules, (ii) the proof $(d_0, d_1, \text{reveals})$ encoded in M is accepted by* VERIFY$(N_F, \Delta_F, \text{index}, \cdot)$ *in F's publication frame, and (iii) every stored e_k is the value* VERIFY *computes after level k on that proof. Consequently, every accepted proof has exactly one encoding that passes all checks, and if M is any other string of n elements, published as a canonical FlatMerkle file in its derived frame, some check fails; a challenger then refutes M by revealing that check's elements – at most five – in a single FlatMerkle multiproof against Δ_M [23, Section 7], in $O(\log n)$ space and verification time. A publication of M that is not such a file – a non-canonical prefix or checkpoint block – is refuted instead by a malformation proof (Section 3.3).*

Proof. The level checks replay VERIFY's compression calls, level by level, with every input read from the element that stores it, and with F's derived initial state and finalization (Section 3). If all checks pass, induction on k gives (iii); the last level check is VERIFY's own comparison with Δ_F, which gives (ii); and the header, half-zero and filler checks give (i), the number and order of reveals being fixed by the header. Conversely, (i)–(iii) make every check pass by construction. Condition (ii) alone is not enough: an accepted proof with one stored output altered is still accepted, yet fails that level's check, and is refuted. What refutation needs is the stated form: a message that passes every check is the canonical encoding of an accepted proof. A level check reads the header

(which fixes the level's shape), the element holding the previous output (or d_0 and d_1 for level 0), the level's own element, and the element holding its checkpoint. Binding of the revealed elements to Δ_M follows from [23, Theorem 5.9] and [23, Lemma 7.1], applied in M's derived frame. □

Figure 2 shows both refutations for the example of Figure 1.

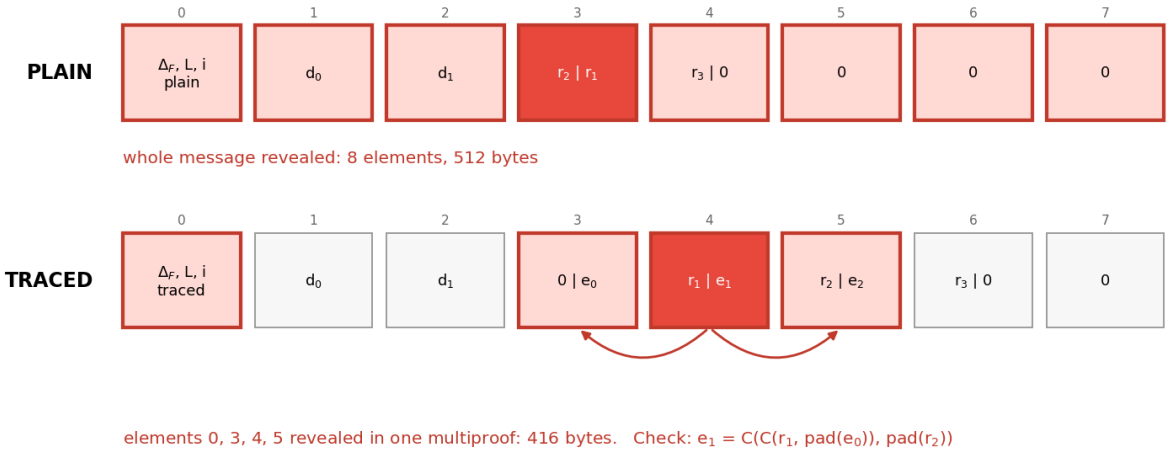


Figure 2: Refuting each format (example of Figure 1). Top: a plain message with a corrupted reveal is refuted by revealing all of it. Bottom: in a traced message, Bob's stored output for level 1 is wrong. The challenger reveals only the header, the previous level's output (element 3), level 1's own element (element 4), and the element holding level 1's checkpoint r_2 (element 5), in one multiproof; the verifier reads the level's shape from the header and recomputes its two compression calls. Byte counts are measured from the reference implementation.

The same construction applies to fraud-proof claims about F [23, Section 5.5]: VerifyFraudProof also has a fixed level structure for given (N, t, V), plus comparisons. Its acceptance condition, a mismatch at *some* anchored level, is existential; the header names the level, so that refuting it is again a single comparison. We implemented and tested the inclusion-proof case only.

5.2 Choosing the format, and binding it

Which format is better depends on what a refutation costs. When it is adjudicated on Bitcoin (Section 7), every byte of a refutation becomes an input of the dispute that must be committed on-chain with one-time signatures, which we take to cost about 200 times as much as a byte of the published message (Table 6; the rule below is the same for any factor from 40 to 856). Three sizes must be kept apart. A message of n logical 64-byte elements is its *payload*, $64n$ bytes; it is *published* as a FlatMerkle file of $2n-2$ blocks after the 64-byte prefix block, $64(2n-1)$ hashed bytes, since the checkpoint blocks must be on-chain; and its *refutation* is what a challenge reveals – all n logical elements for a plain message, from which the verifier rebuilds the checkpoint blocks, or a multiproof for a traced one. We compare formats by

$$\text{cost} = \text{published bytes} + 200 \times \text{refutation bytes},$$

with refutation sizes measured as the worst case over all checks *and all target indices* (Table 2): which elements a level check reads depends on the index through that level's branch and the position of its checkpoint, so the implementation maximizes over every index with at most two set bits,

which reaches the exhaustive maximum for every $L \leq 10$. The traced format is cheaper for every $12 \leq L \leq 54$; the plain format is cheaper for $6 \leq L \leq 11$, where a plain message has eight elements; the two tie at $L = 4$ and 5, where ties go to the plain format; and at $L = 2, 3$ the plain format is cheaper on both counts (256 refutation bytes against 416).

	Plain			Traced		
L	published	refutation	cost	published	refutation	cost
4	960	512	**103,360**	960	512	103,360
8	960	512	**103,360**	1,984	608	123,584
11	960	512	**103,360**	1,984	640	129,984
12	1,984	1,024	206,784	1,984	640	**129,984**
14	1,984	1,024	206,784	4,032	736	**151,232**

Table 2: Published size ($64(2n-1)$ hashed bytes for n logical elements, prefix included), worst-case refutation size (bytes) and cost for a claim about a 2^L-element file, worst case over target indices and checks (for example $L = 14$: index 8192, the level-5 check, elements 0, 7, 8 and 16), messages padded to a power of two ($n = 8$ or 16 plain, 8, 16 or 32 traced). At $L = 4$ the formats tie and the plain one is chosen. The cheaper format is in bold. $L = 14$ is the largest file that fits in a block (Section 3.5).

A protocol therefore fixes, as a public constant, the set of L for which claims use the traced format. Allowing two formats opens an *encoding attack*: a dishonest party could publish a message that one side reads as traced and the other as plain, so that a correct challenge in one reading is rejected in the other. Three rules close it. First, the format is a function of public context, not of the message: the verifier already knows which file the claim is about, (Δ_F, N_F), and so knows the required format before reading M. Second, the header check fails unless Δ_F and N_F match the context, the tag matches the required format, and n equals the size that format prescribes; such a header is refuted by revealing element 0 alone. Third, the two formats have disjoint sets of checks – level, half-zero and filler checks for traced, a single whole-message check for plain – and a verifier rejects a challenge that names a check of the other format.

Alternatives we compared. Storing one element per compression call, rather than one per level, makes the message about twice as large again (up to $2L + 1$ elements of trace instead of L) with equal or larger refutations, and is never cheaper than the traced format in our measurements (taken at index 0). Storing a level's output only every c levels, with outputs packed two per element, shrinks the message. With $c = 3$ a message for $6 \leq L \leq 9$ fits in 8 elements, but its worst-case refutation is 512 bytes, so it only ties with the plain format (103,360); at $L = 14$ the packed layout with $c = 3$ fits in 16 elements and is refuted with at most 704 bytes, cost 142,784 against 151,232 for the traced format. The protocol above uses $c = 1$ throughout; adopting the packed layout at $L = 14$ would be a change of the public rule only, and a refutation then recomputes up to three levels and reveals up to eight elements.

6 Recursion and its limit

Since M is itself a structured file, a claim about M – an inclusion proof for one of its elements, or a fraud proof against its layout – can in turn be encoded in a third file M_2, and so on. Along a chain $F \to M_1 \to M_2 \to \cdots$ each message has $n_k = O(\log n_{k-1})$ elements, so sizes follow the iterated

logarithm and reach a constant after $O(\log^* N_F)$ levels. Table 3 gives the concrete chain under the format rule above.

	$N_F = 2^9$	$N_F = 2^{11}$	$N_F = 2^{14}$
M_1 (claim about F)	plain, 8 / 960 / 512	plain, 8 / 960 / 512	traced, 32 / 4,032 / 736
M_2 (claim about M_1)	plain, 4 / 448 / 256	plain, 4 / 448 / 256	plain, 8 / 960 / 512
M_3 (claim about M_2)	plain, 4 / 448 / 256	plain, 4 / 448 / 256	plain, 4 / 448 / 256

Table 3: Recursive proof messages over SHA-256, worst case over target indices: for each message, its format, its logical size in 64-byte elements, its published size in hashed bytes ($64(2n-1)$, prefix included), and the bytes needed to refute it if false, for the largest files the annex-free output profile (2^9), Libre Relay's standard weight (2^{11}) and a block (2^{14}) can carry. In the first two columns a proof of a proof, M_2, is already the fixed point, 4 logical elements (256 bytes of payload, 448 published); at 2^{14} it is reached at M_3. The header admits N_F up to 2^{54}, for which the chain still settles within three levels.

The fixed point is the smallest file whose proof fits in itself: a plain message about a 4-element file needs the header, the pair and one reveal, which is exactly 4 logical elements; published with its prefix and checkpoint blocks it occupies 448 bytes. An end-to-end chain built from a real 1,024-element file gives messages of 8, 4 and 4 elements.

What this does *not* provide should be stated plainly. FlatMerkle gives succinct access to committed data, not succinct proofs of computation: each level is cheap only because the statement checked at that level is about a small object. A claim such as "$state_i$ is the correct result of executing tx_i" [23, Section 10] still needs that execution, or its own trace and dispute game. Each level's digest must also be authenticated to whoever checks it, which brings the considerations of [23, Section 9.4] to every level, and the verifier of a challenge must know n, just as the verifier of an ordinary proof must know N.

Testing. The implementation (`proof_messages.py`, using a real-SHA-256 port of the multiproof of [23, Section 7] in `loglink_annex.py`, tested on 756 target sets with every pair element and reveal tampered) checks the following, for every index at $N_F \in \{4, 8, 16, 32\}$ unless stated otherwise. Honest messages pass every check under three protocols (every claim traced, every claim plain, and the rule above), and no challenge naming any check of either format is accepted against them. All 540 encoding attacks tried are refuted: an honest message in the format the context does not prescribe, the prescribed tag over the other format's layout, the other layout resized to the prescribed n, a traced header with a different layout or stride byte, and a valid message about a different file. A challenge valid only under the other format's rule is rejected. For $N_F \in \{8, 16, 32\}$ and both formats, false data and a one-byte corruption of every element at three byte positions give 2,592 invalid messages, all refuted, the traced ones by revealing at most five elements; the remaining 112 corruptions still encode a true claim (for instance, flipping the lowest bit of the header's index turns the claim into one about the pair partner, which shares the same base pair and verification path). All of these tests are run twice: with standalone files, and with F and every message built in its derived annex frame, so that every multiproof, level check and plain re-verification uses the derived initial state and prefix-aware finalization of Section 3. The two alternative designs above are implemented in `recursive_proof.py` and `compact_trace.py`, with their own tests.

7 Verifying claims on Bitcoin

The previous sections reduce every dispute about published data to a short, fixed computation. This section describes that computation and how it can be adjudicated by Bitcoin itself.

7.1 What a dispute computes

A dispute concerns data that a publisher, Bob, has committed to in a transaction he signed (Section 3), and a claim about it. The dispute program receives Bob's public key, the commitment's authentication data – (H_{192}, C) in the annex profile, or the 94 message bytes of the output profile – Bob's signature, and the evidence, and accepts the challenge if and only if all of the following hold:

1. the signature hash rebuilt from the authentication data, by one compression (annex profile) or three (output profile), is a digest on which Bob's BIP340 signature verifies, two more compressions going to BIP340's tagged challenge hash of $R\|P\|m$, with its tag prefix precomputed, and the rest to the elliptic-curve check $s \cdot G = R + e \cdot P$ [43];

2. the evidence, folded as a FlatMerkle proof in the publication's derived frame, gives a digest whose last 30 bytes are C, or which equals D in the output profile; and

3. the evidence contradicts what Bob claimed: an inclusion proof for a value other than the one Bob asserted, a fraud proof showing that the file is malformed [23, Section 5.5], or a failing local check of a proof message (Section 5).

When the claim is a proof message about a file F that was itself published this way, the context the message's header is checked against (Section 5.2) is F's commitment and frame, and the header check compares the commitment with the stored Δ_F – *only* the authenticated value: in the annex profile the last 30 bytes of Δ_F against C. A dispute that took the full 32-byte Δ_F from the challenger would let him change its first two bytes, which no signature covers, make the header check of an honest message fail, and condemn it. The end-to-end test alters each of the two bytes against every check of every honest message and checks that no challenge is upheld. Every part except the elliptic-curve check is a short sequence of SHA-256 compressions. Which publication is disputed is fixed by the protocol, not by these checks (Section 3.4).

Who supplies the evidence. The predicate above is the right one when the *challenger* supplies the evidence: an invalid multiproof or proof only hurts the challenger, so the challenge is upheld when the evidence is valid *and* contradicts the claim. When the *accused* must answer – as in the FlatChecks of Section 8, where the circuit's inputs are posted by the party whose claim is in question – the predicate must be inverted: the accused loses unless she posts a valid opening *and* the check passes, and she also loses on a malformed response or by timeout. The authentication of her publication must then come from the challenger, through a publication slot (Section 3.4), not from her. Otherwise she could evade any challenge by posting a deliberately invalid opening.

What a dispute actually checks. A claim about F is never put to the dispute program directly. It is published as a proof message M_1 (Section 5), and what the program adjudicates is a *proof about that proof*: a challenge against M_1, which is a multiproof of at most five elements of M_1 and a local check, for the traced format, or all of M_1 followed by VERIFY of the proof it encodes, for the plain format. Item 2 above is therefore folded against M_1's commitment, item 3 is a failing check of M_1, and the program authenticates two commitments, F's, for the header check, and M_1's. The

same holds one level up (Section 6): a claim about M_1 is a message M_2, and a challenge against it is a proof about a file of at most 32 elements.

File sizes. A FlatMerkle file of N elements is hashed as $64(2N-1)$ bytes (Section 3), so the size limits of Section 3.5 bound L: $L \leq 9$ in the annex-free output profile, which Bitcoin Core relays; $L \leq 11$ in the annex for a transaction of standard weight, which only Libre Relay forwards; and $L \leq 14$ in the annex for any valid block. Table 4 gives the counts at these three sizes, measured by counting compression calls in the reference implementation.

	$L = 9$ output, Core	$L = 11$ annex, Libre Relay	$L = 14$ annex, block
M_1, claim about F: format, published	plain, 960 B	plain, 960 B	traced, 4,032 B
Refutation of M_1, bytes	512	512	736
Challenge against M_1	34	38	30
with two annex commitments (2×3)	40	**44**	**36**
with two output commitments (2×5)	**44**	48	40
M_2, claim about M_1: format, published	plain, 448 B	plain, 448 B	plain, 960 B
Refutation of M_2, bytes	256	256	512
Challenge against M_2	14	14	26
with two annex commitments	20	**20**	**32**
with two output commitments	**24**	24	36
Inclusion proof checked directly against F	19	23	29

Table 4: SHA-256 compressions in a dispute about a 2^L-element published file F, worst case over every target index and every check a challenger may name, finalization included. Message formats are those the rule of Section 5.2 prescribes; published sizes are hashed bytes, prefix and checkpoint blocks included. Files and messages are built in their derived frames; the derived initial state is a constant when N is a public parameter, and costs one more compression otherwise. Each authenticated commitment costs one compression (annex) or three (output) for the signature hash and two for BIP340's challenge hash; bold marks the profile each column assumes. The elliptic-curve part of each BIP340 check is not included. The last row is for comparison only: on Bitcoin, claims reach the dispute as messages. Reproduced by `dispute_costs.py`.

Three observations follow from the table. First, the dispute never touches F beyond the header check, so its cost depends on L only through the size of M_1: 34 to 38 compressions for a plain M_1, 30 for a traced one, and, one level up, 14 to 26 for a challenge against M_2, whose refutation is at most 512 bytes. Second, in the whole standard range, $6 \leq L \leq 11$, the rule prescribes the plain format, whose refutation reveals and re-verifies the whole message. A traced M_1 would need only 23 or 24 compressions, but its worst-case refutation is 608 to 640 bytes against the plain format's 512, and its published size twice as large, which is why the cost model, which counts only bytes, prefers the plain format. A protocol for which circuit size also matters can make the traced format the rule for these L as well; the rule is a public constant either way. Third, at these sizes a plain challenge costs more compressions than checking an inclusion proof directly against F (34 against 19 at $L = 9$), because it re-runs that check and also rebuilds M_1; what the message buys is not a cheaper circuit but a published, bindable claim that anyone can refute.

7.2 Why not Script

Bitcoin Script has no opcode that runs the SHA-256 compression function on a supplied chaining state, so the computation above must be implemented from bit or byte operations. The most optimized published implementation we know of, in the BitVM repository, works on 4-bit limbs with lookup tables; hashing a 32-byte message, a single compression, takes 344 kB of script in its first version and 297 kB after further optimization [17, 18]. In a tapscript, which is witness data at one weight unit per byte [30], that is about 75,000 virtual bytes per compression, and the dispute of Table 4 needs dozens. Verifying a Schnorr signature directly in Script is larger: a BIP340 verifier with the key fixed and the message and signature supplied in the witness, with field arithmetic certified by witness hints, measures 8,292,228 bytes of script and an 81,740-byte witness of 32,556 items [36], about 25 SHA-256 compressions' worth and roughly 1% of the Groth16 verifier that BitVM2 splits into 379 scripts of 0.5 to 4 MB each [28]. It exceeds both the 1,000-item stack limit and the transaction weight limit, so it too would have to be split. None of this fits in one transaction. There are three ways around it.

7.3 Interactive bisection

The dispute program can be executed off-chain and only its disagreement point brought on-chain. In BitVMX [25], the program runs on a virtual CPU; prover and challenger commit to hashes of the execution trace and bisect it, in a logarithmic number of rounds, down to a single instruction, which a small script adjudicates, the approach of Truebit and of interactive rollup fraud proofs [19, 37] adapted to Script. The FlatMerkle dispute fits this model directly, and only one instruction is ever executed on-chain. The price is interaction: every round is an on-chain transaction with its own timeout, so a dispute takes many rounds and their latency, in addition to the rounds of the FlatMerkle-level game itself when proof messages are challenged.

7.4 Chunked Script, without garbled circuits

BitVM2-CORE [28] adjudicates a program too large for one transaction by splitting it into chunks: when challenged, the operator, whose claim is in question, commits in an Assert transaction, with one-time signatures, to every chunk's inputs and outputs, and anyone who finds a wrong output re-executes that one chunk in a Disprove transaction, one Taproot leaf per chunk. The operator is fixed when the transaction graph is set up; whoever disproves is not.

Who asserts. The program here is the dispute program, whose input is the evidence and whose output is that the challenge is upheld, and the party who asserts it is the one who supplies the evidence. In the disputes of Section 7.1 that is the challenger. So it is for a FlatCheck (Section 8) adjudicated this way: the trace is public, so Bob, having found a failing check, opens its words himself rather than asking Alice to, and the predicate is the challenger's one, that the opening is valid and the check fails, which by binding no opening satisfies against an honest Alice's canonical publication; the reference implementation checks both directions on a traced verifier. The roles of BitVM2 are therefore inverted. When Alice publishes a proof and Bob challenges it, Bob is BitVM2's prover: he posts the Assert, committing to his evidence and every chunk boundary, and pays for it. Alice is its challenger: she disproves the Assert if any chunk is wrong, and if none is disproved before a timeout, Bob's challenge is upheld and Alice loses. A successful Disprove forfeits Bob's challenge bond.

Challengers are fixed in advance. The Disprove leaves check the Assert's one-time signatures against public keys written into their scripts, and the Assert and Disprove transactions are pre-signed, so the transaction graph that lets Bob challenge must be hard-wired for Bob, with his keys, when Alice's protocol is set up. Anyone can still disprove a wrong Assert, but that protects Alice, not the users of her claim; the right to challenge her is not open to anyone. A protocol with n potential challengers needs n such graphs, each with its own one-time-signature keys and Taproot leaves, and its security assumes that one of the n is honest and online. Letting the claimant answer instead, as in Section 8, restores BitVM2's orientation for the Assert – Alice asserts her opening and the chunk boundaries, and anyone may disprove – but not for the challenge: if any party could select the check, Alice could select a passing one first and exhaust the challenge, so the selection must either be reserved to designated challengers or be available once per check.

The FlatMerkle dispute fits this model with one SHA-256 compression per chunk – 332,830 bytes of script for a compression continued from a supplied midstate [36] – and the BIP340 check split by the stack limit: its 32,556 witness items need at least 37 chunks of at most 900 items, about 225 kB of script each. Chunk-internal hints certify deterministic values and can be supplied by the challenger, so the Assert commits only each compression's input block and output state, 96 bytes, and each signature chunk's boundary state, which we assume to be 100 bytes (mainly the running curve point). Table 5 gives the resulting estimates.

	M_1, $L \leq 11$	M_2, $L \leq 11$
SHA-256 chunks / BIP340 chunks	40 / 74	16 / 74
Total script, all leaves	29.9 MB	21.9 MB
Committed in the Assert	11.2 kB	8.9 kB
WISCH / Antichain Winternitz	323 / 752 kvB	257 / 598 kvB
Winternitz	1,574 kvB	1,251 kvB

Table 5: Estimated cost of adjudicating a dispute as chunked Script in the manner of BitVM2-CORE, two signatures per message challenge. Script sizes are measured per part [36]; chunk counts, the 100-byte signature boundary, and the Assert sizes (marginal, Table 6) are our estimates, computed by `dispute_costs.py`. The BIP340 script has not been split into stack-feasible chunks.

The estimated SHA-256 Disprove chunks fit the standard weight budget with Antichain Winternitz or WISCH: a chunk with its one-time-signature checks stays under 400,000 weight units, and is just above it with plain Winternitz. Standardness of the complete construction remains to be validated after implementing the BIP340 chunking and serializing all dispute transactions. The whole program, 22 to 30 MB, is about 2–3% of BitVM2's Groth16 verifier. Compared with a garbled circuit (Section 7.5), a dispute puts four to ten times more on-chain – the Assert, paid by the challenger and only when he challenges – but needs no garbled tables or cut-and-choose: its setup is, per authorized challenger, one-time-signature keys and Taproot leaves. The Assert re-commits data that is already published in the annex, because Script cannot read the annex; with `OP_CAT` this duplication would not be needed.

7.5 Garbled circuits

A newer family of protocols replaces the bisection game by a garbled circuit [45], used not for privacy but for the *authenticity* property of garbling [5]: an evaluator cannot produce an output label other than the one the circuit computes. The prover garbles a circuit for the dispute program in advance; when the prover's claim is later posted, a challenger evaluates the circuit off-chain on the posted

inputs and, if and only if the claim is false, learns a secret that serves as a fraud proof on-chain. This is the approach of BitVM3 [29], of Glock, which leaks a signature as the fraud proof [8], of BABE, which combines witness encryption with a garbled circuit for elliptic-curve scalar multiplication to verify Groth16 proofs [14], and of the garbled-circuit module of BitVMX [10]; OHMG gives an authenticity-oriented garbling of arithmetic gates, with each arithmetic gate costing at most one ciphertext [12]. The reported on-chain cost of a challenge in these protocols is small – for BitVM3, about $0.20 for the challenge transaction [29] – because the circuit itself never goes on-chain.

These protocols are designed around a SNARK verifier, a large circuit. The FlatMerkle dispute is much smaller and has a simpler structure. Its SHA-256 part is 20 to 44 compressions (Table 4), at 22,573 AND gates each in a standard boolean circuit [2]: about 0.45 million AND gates for a challenge against M_2 and at most about 1.0 million against M_1, for any file that a standard-weight transaction carries. With half-gates garbling, two 128-bit ciphertexts per AND gate and XOR gates free [46], that is 722 kB of garbled tables per compression, 14 to 32 MB for the SHA-256 part of a dispute, all of it off-chain. What remains is the elliptic-curve part of BIP340, two scalar multiplications on secp256k1, which is the expensive component of the circuit in boolean form; arithmetic garbling such as OHMG [12], or the elliptic-curve scalar multiplication circuit of BABE [14], or Argo MAC, which translates the bit decomposition of a curve point into a homomorphic MAC of it [9], targets exactly this kind of computation. The circuit can also be specialized: Bob's public key is fixed when the circuit is garbled, so $e \cdot P$ multiplies a known point.

What a challenge puts on-chain. The circuit's inputs must be committed on-chain so that they are bound to the party that supplies them and the evaluator can obtain the corresponding input labels. The garbled-circuit dispute of BitVMX posts the prover's circuit input in one transaction with Lamport signatures over SHA-256 [10, 21]; Winternitz signatures [31] can be translated off-chain into Lamport labels [35]; Antichain Winternitz commits to several bits per hash chain and guarantees that any accepted opening reveals the labels, through a public off-chain table [38]; Mosaic reveals the labels of each input digit by completing a Schnorr adaptor signature [20]; and WISCH replaces the per-byte hash chains by a single aggregate Schnorr signature in the optimistic case and a Winternitz opening only on challenge [13]. Table 6 gives the cost of one committed byte.

One byte of	weight units	vbytes	post-quantum
Annex (witness) data [30]	1	0.25	–
`OP_RETURN` output data [30]	4	1	–
Input, Schnorr adaptor (Mosaic) [20, 39]	≈ 68	≈ 17	no
Input, WISCH [13]	115	28.75	no
Input, Antichain Winternitz [38]	≈ 268	≈ 67	yes
Input, Winternitz [13]	≈ 560	≈ 140	yes
Input, Lamport, 20-byte hashes [38]	568	142	yes
Input, Lamport, 32-byte hashes, as in [10]	856	214	yes

Table 6: Marginal on-chain cost of one byte, script and witness included, fixed transaction overhead excluded. Mosaic: 2.13 vbytes per bit [39]. WISCH and Winternitz: 230 and about 1,120 weight units per 16-bit chunk with $w = 16$ (Table 2 of [13]). Antichain Winternitz with $(k, L) = (4, 16)$: 8.36 vbytes per bit, plus an off-chain table of 43.8 kB per bit. Lamport: 17.75 and 26.75 vbytes per bit, measured in [38]. "Post-quantum" refers to the commitment scheme alone; WISCH's optimistic path and adaptor signatures rely on discrete logarithms.

A challenge's circuit input is its refutation – the revealed message elements and their multiproof – plus, for each authenticated commitment, the midstate H_{192} and the BIP340 signature, 96 bytes; the 30-byte commitment itself is either recomputed from revealed data or read from a header. The output profile needs 158 bytes per commitment instead (94 message bytes and the signature), 124 more per dispute. Table 7 gives the totals for the disputes of Table 4, each with two annex commitments. Authentication can be omitted for commitments already independently authenticated and fixed as circuit parameters.

Challenge against	bytes	WISCH	ACW	Winternitz	Lamport-32
M_1, plain ($L \leq 11$)	704	20,240	47,084	98,560	150,656
M_1, traced ($L = 14$)	928	26,680	62,065	129,920	198,592
M_2, plain ($L \leq 11$)	448	12,880	29,962	62,720	95,872
M_2, plain ($L = 14$)	704	20,240	47,084	98,560	150,656

Table 7: Garbled-circuit input of a challenge, worst case over target indices and checks, in bytes, and its marginal on-chain cost in vbytes under the commitment schemes of Table 6. The packed $c = 3$ layout gives no saving at $6 \leq L \leq 9$ in the worst case. Reproduced by `dispute_costs.py`.

Two consequences follow. With Lamport or plain Winternitz commitments the input of a challenge against M_1 is close to or above the 100,000-vbyte limit of a standard transaction, and must be split; with Antichain Winternitz or WISCH every challenge fits in one, and a challenge against a proof of a proof, M_2, costs about 30,000 or 13,000 vbytes respectively for files of up to 2^{11} elements, and about 47,000 or 20,000 at 2^{14}. And the weight of a challenge byte relative to a published annex byte ranges from about 68 to 856. The factor of 200 used in Section 5.2 lies within that range, and the choice does not matter: recomputing the format rule, with published sizes counted as in Section 5.2, with any factor from 40 to 856 gives the same rule, traced exactly for $L \in \{12, 13, 14\}$, over every file size that fits in a block.

Post-quantum security. Nothing in the construction relies on a problem that a quantum computer solves efficiently, except the signatures Bitcoin itself uses. FlatMerkle, the publication commitment and the proof messages use only SHA-256, and garbling uses only symmetric primitives; against these, the known quantum attacks are generic searches that reduce security margins rather than break the schemes, and the practical relevance of those speedups at current parameters is not settled [39]. The quantum-vulnerable parts are the BIP340 signature that binds a publisher to a commitment and, if they are used, Schnorr-based input commitments such as WISCH's optimistic path or Mosaic's adaptor signatures [39]. The protocol is therefore post-quantum provided that Bitcoin adopts post-quantum signatures, the publisher signs with one, and circuit inputs are committed with a hash-based scheme such as Lamport, Winternitz or Antichain Winternitz. Two changes follow. The dispute circuit must verify the new signature instead of BIP340, a hash-based one costing many more compressions; and the one-compression commitment of Section 3 depends on the length and layout of BIP341's signature message, so it must be re-derived for whatever message a post-quantum spend signs. A post-quantum design must also rule out a key-path spend that bypasses the hash-based conditions [39], whereas our commitment, as specified, uses one. Without waiting for a soft fork, QSB makes a transaction's binding quantum-safe today, through hash-to-signature puzzles whose security rests on the preimage resistance of RIPEMD-160, at an off-chain cost of a few hundred dollars of GPU work per transaction [27]. It requires legacy (pre-SegWit) script execution and a bare output script, so it cannot carry an annex or use the commitment of

Section 3: a QSB publication would use the output layout, and a dispute would have to authenticate the file through the legacy signature hash rather than through one compression, which we have not worked out. The pre-signed transactions of the dispute game itself, which today use aggregated Schnorr keys, would need the same treatment.

What we have not done. We have not garbled the dispute circuit; the gate counts and table sizes above cover only its SHA-256 part, from a standard circuit, and the elliptic-curve part is not estimated. The binding of a signature to the confirmed transaction variant is outside the circuit (Section 3), as is the protocol that penalizes the publisher once a challenge succeeds.

8 FlatCheck: verifying locally refutable proofs

The traced proof messages of Section 5 compile one verifier, FlatMerkle's own, into a file of intermediate values in which a false claim is refuted by a single local check. Nothing in that idea is specific to FlatMerkle. Any deterministic verifier whose control flow is fixed by public parameters can be written out as a trace in which each step reads a few earlier values; if the trace is committed as a FlatMerkle file, a false proof is refuted by opening the few elements that one failing step reads, with one multiproof. This section names that property, gives three ways to publish such proofs on Bitcoin, and the transaction graph that adjudicates them. We call the construction a *FlatCheck*.

8.1 Definitions

Let V be a deterministic verifier for statements x and proofs P, byte strings. A *trace layout* for V assigns every pair (x, P) a sequence of 32-byte words $\tau(x, P)$, its *canonical trace*, whose length and layout depend only on public parameters, and which contains every word of P that the verifier reads. A *check* γ is a predicate on the words at a set of positions $R(\gamma)$, the check's *reads*; a *check system* is a finite set Γ of checks, also fixed by the public parameters. When the positions a step of V reads depend on a value in the trace – as those of a proof message's level depend on the target index in its header – the check system has one check per possible value: it reads the positions that value selects, and passes if the trace holds another value. The reads of every check thus remain fixed.

Three objects must be kept apart. The trace is a sequence of words, its *semantic* positions. A *physical layout* places those words in a FlatMerkle file, and may leave positions that hold no word of the trace: alignment words, which let reads that belong together share an element, and the padding to a power-of-two size. No check reads them, the honest prover fills them with zeros, and they constrain nothing; the definitions below concern the projection of a file on the semantic positions. Finally, whether that file is *canonical* is a property of its checkpoint blocks (Section 3.3), not of its contents.

Definition 8.1 (Locally refutable verifier)**.** (V, τ, Γ) *is* k-locally refutable *if*

1. (locality) *every check reads at most* k *words;*
2. (completeness) *if* $V(x, P)$ *accepts, every check of* Γ *passes on* $\tau(x, P)$*;*
3. (refutation soundness) *for every sequence of words* T *of the trace's length on which every check passes,* $T = \tau(x, P_T)$ *and* $V(x, P_T)$ *accepts, where* P_T *is the proof read from* T*'s proof positions.*

Refutation soundness is what makes a single check a refutation: if V rejects P, then *no* sequence of words that contains P passes every check, canonical or not, so whatever trace the prover stands behind, some check fails, and its index together with the k words it reads, and their opening against the trace's commitment, refutes the proof. "Local" counts positions, as for locally testable codes and probabilistically checkable proofs [3], not contiguity: the reads of a check may lie anywhere in the trace, and a FlatMerkle multiproof opens them together. The term *locally checkable proofs* already names a different notion, certificates checked within a constant radius of each node of a distributed network [15].

Definition 8.1 refines the *naysayer proofs* of Seres, Glaeser and Bonneau [34], in which a verifier is convinced that a false proof is invalid at a cost far below that of verifying a true one; they show that every NP language has constant-size naysayer proofs and give constructions for several proof systems. A locally refutable verifier yields naysayer proofs of k words and one multiproof against a commitment to the trace. What this section adds is how the trace is committed, published and adjudicated on Bitcoin without a smart contract, and how little of it needs to be published.

Definition 8.2 (Separable verifier)**.** *A locally refutable* (V, τ, Γ) *is* separable *into parts* $P_1, \dots, P_m$*, with* source sets $S_1, \dots, S_G \subseteq \{1, \dots, m\}$*, if every proof is* $P = (P_1, \dots, P_m)$ *and*

$$V(x, P) \;=\; \bigwedge_{g=1}^{G} V_g\big(x, (P_i)_{i \in S_g}\big),$$

where, for each g*, the restriction of* τ *and* Γ *to the positions of the parts in* S_g *and of the values* V_g *computes is a locally refutable verifier of* V_g*.*

Each part is published once, however many groups read it. Groups can be merged: the conjunction of several V_g reads the union of their source sets. In the *common-part* case a part C is read by every group, and each group has a part Q_g of its own: $S_g = \{C, Q_g\}$, and a check of C alone can be assigned to any group. Sources can also be sparse: a state machine whose state at every group boundary is published in blocks has groups that read their own transitions and the one or two blocks holding their boundary states (Section 8.6).

The proof messages of Section 5 are locally refutable, with $k \leq 10$: a check reads at most five 64-byte elements (Section 5.1), that is, at most ten words, and each level check exists once per target index, by the convention above. A STARK verifier is separable, in the common-part case: it derives every challenge and query position from the commitments it absorbs and then checks each query independently, so the common part is the Fiat–Shamir transcript and there is one part per query. We treat that application separately; here the verifier is arbitrary.

A *FlatCheck* is a locally refutable verifier whose trace is committed as a FlatMerkle file, published as in one of the three modes below, and adjudicated by the dispute program of Section 7: a challenge names a check, and the accused answers with the words it reads and their multiproof.

8.2 The trace published with the proof

Protocol. Alice publishes the proof and its trace in the annex of a transaction that spends a publication slot (Section 3.4), in the annex's derived frame. If Bob believes the proof is false, he selects one check and commits, with one-time signatures, its index and the hash type, annex hash and signature copied from the confirmed publication. Alice must then post, committed with one-time signatures, the words the check reads and their FlatMerkle multiproof against that annex hash. Since Alice is the accused party answering a mandatory request, the circuit releases the fraud secret

– Alice loses – if Bob's copy verifies in the slot and her opening is not valid or the check fails, or if the copy shows a spend of the slot without an annex:

$$\text{Alice loses} \iff \text{SlotAuth} \wedge \big(\neg\,\text{ValidOpening} \vee \neg\,\text{CheckPasses}\big),$$

and a response that does not parse counts as an invalid opening. The opposite predicate for the opening, "valid and failing", would let her evade every challenge by posting a corrupted multiproof. The authentication comes from Bob, not from Alice: if Alice supplied it, she could answer against a variant of her publication that she signed but never broadcast. The circuit therefore takes inputs from both parties, as in FLEX [24]. If Alice does not respond, Bob wins by timeout; if Bob does not challenge, Alice does. Bob has a second branch: instead of selecting a check, he may prove that the publication is not canonical with a malformation proof (Section 3.3), which is challenger-supplied evidence; an accepted one makes Alice lose.

Why one check suffices. What is adjudicated is the values Alice can *open*, not the bytes physically published: on a non-canonical file an ordinary opening can accept a value other than the published element. The argument therefore has two cases.

Openable values. By binding [23, Theorem 5.9], which holds for every digest, each position has at most one value that Alice can open, whatever the publication; let T be these values at the positions of the trace. If Alice could answer every check, her responses would open the same values wherever they overlap, so every check would pass on T, and by refutation soundness T would be the canonical trace of a proof P_T that V accepts. This does not yet concern the published proof: P_T is what Alice can open, which need not be what she published.

Canonical publication. If the published file is canonical, every position can be opened to its published value, and by binding only to it, so T is the published trace and P_T the published proof. Hence, if V rejects the published proof, some check has no accepting response, up to a free-start collision. Bob finds it from the published bytes: he evaluates the checks on them and selects the first that fails.

Non-canonical publication. Then T may differ from the published words, and P_T may even be a proof that V accepts. But the publication has an accepted malformation proof, which Bob constructs from the published bytes (Theorem 3.2), and he uses the malformation branch.

Together:

$$V \text{ rejects the published proof, or the file is not canonical}$$
$$\Downarrow$$
$$\text{some check has no accepting answer, or a malformation proof is accepted.}$$

Conversely, an honest Alice publishes a canonical file, against which no malformation proof is accepted (Theorem 3.1), and by completeness every check can be answered. As in Section 5.2, the layout and the check system are fixed by the public parameters, so a trace cannot be read in two ways.

8.3 The trace on demand

Publishing the trace with the proof makes every proof pay for it, although the trace is read only in a dispute. It need not be published until then. Alice publishes only the proof, and publishes the trace if and only if she is challenged, bound to the proof she published first by the SHA-256 chain of its annex hash.

Protocol. Six transactions in the worst case:

1. *Proof* (Claim Tx in Figure 3). Alice publishes the proof A_1, unstructured, in the annex of a transaction that spends a publication slot (Section 3.4); its commitment is the annex hash h_{A_1}.

2. *Challenge.* Bob opens a dispute with a minimal transaction and posts a security bond. With FLEX [11, 24], the bond is attached to a pre-signed transaction only when the dispute starts, so challengers keep no capital locked while no dispute is open; as in FLEX, the challengers are a permissioned set (Section 7.4).

3. *Trace.* Alice publishes A_2, the trace as a FlatMerkle file, in the annex of a transaction that spends a second slot, or loses by timeout.

4. *Selection.* Bob selects one local check of A_2 and commits, with one-time signatures, the hash types, annex hashes h_{A_1} and h_{A_2} and Alice's signatures, copied from the two confirmed transactions; or he proves A_2 malformed (Section 3.3); or he forfeits his bond by timeout.

5. *Answer.* Alice posts the opening of the selected check against h_{A_2}, committed with one-time signatures, or loses by timeout.

6. *Settlement* (BobWins Tx or AliceWins Tx). Bob evaluates Alice's garbled circuit; if his copies verify in their slots and her answer is invalid or the check fails, or a copy shows a slot spent without an annex, he learns the secret and takes her stake. Otherwise Alice takes Bob's bond after a timeout.

An honest Alice publishes only A_1. Since a frivolous challenge makes her publish A_2, Bob's bond must cover at least its fee.

The transaction graph. Figure 3 gives the graph for one designated challenger, built from the components of FLEX [24]. We write "Claim Tx" for a transaction of the graph and the bare name for the step it carries out. $\langle A, B \rangle$ denotes a signature that Alice and Bob produce together at setup, so that a transaction carrying it is pre-signed and fixed. Alice's stake is an output of the setup that only pre-signed transactions can spend, except that Bob can take it with the secret P_b, which Alice's garbled circuit releases exactly when she loses. Claim Tx spends the first slot S_1 by key path and carries A_1 in its annex. Its single input and its outputs are fixed at setup, and the annex is witness data, so its transaction identifier is fixed although A_1 is chosen later; the graph can therefore pre-sign spends of its outputs. These are the second slot S_2, four connectors and K_w, which lets Alice win after 9Δ. A connector is spent either by the move it enables, at any time, or, after a relative timelock counted from Claim Tx, by a pre-signed timeout transaction of the other party, which also takes the stake: whoever does not move in time loses. The moves are:

- *Challenge.* Bob spends K_c with a pre-signed `ALL|ANYONECANPAY` signature, to which he attaches an input funding his bond; the bond goes to the pot Π_B, as in FLEX. Without a challenge, Alice's NoChallenge Tx spends K_c after 2Δ, together with K_t, which disables MissingTrace Tx, and the stake.

- *Trace.* Alice spends S_2 by key path with A_2 in its annex, together with K_t, whose pre-signature is `SIGHASH_ALL` and so fixes Trace Tx up to its witnesses, as a slot requires; otherwise Bob's MissingTrace Tx after 4Δ.

- *Selection.* Bob spends K_s, whose script verifies his one-time signatures on m_B: the selected check (or a malformation proof) and, for Claim Tx and Trace Tx, the hash type, annex hash and signature copied from the confirmed transaction, 197 bytes for a check; otherwise Alice's MissingSelection Tx after 5Δ.

- *Answer.* Alice spends K_a, committing her opening with one-time signatures whose preimages are her circuit's input labels. She also re-posts Bob's signatures on m_B and co-signs m_B with her own label keys; K_a's script checks that both sign the same message, as in FLEX's deposit transactions. The co-signature gives Bob the labels of m_B's bits and of no other value, so the check index can be a committed input here rather than a Taproot leaf. Otherwise Bob's MissingAnswer Tx after 7Δ.

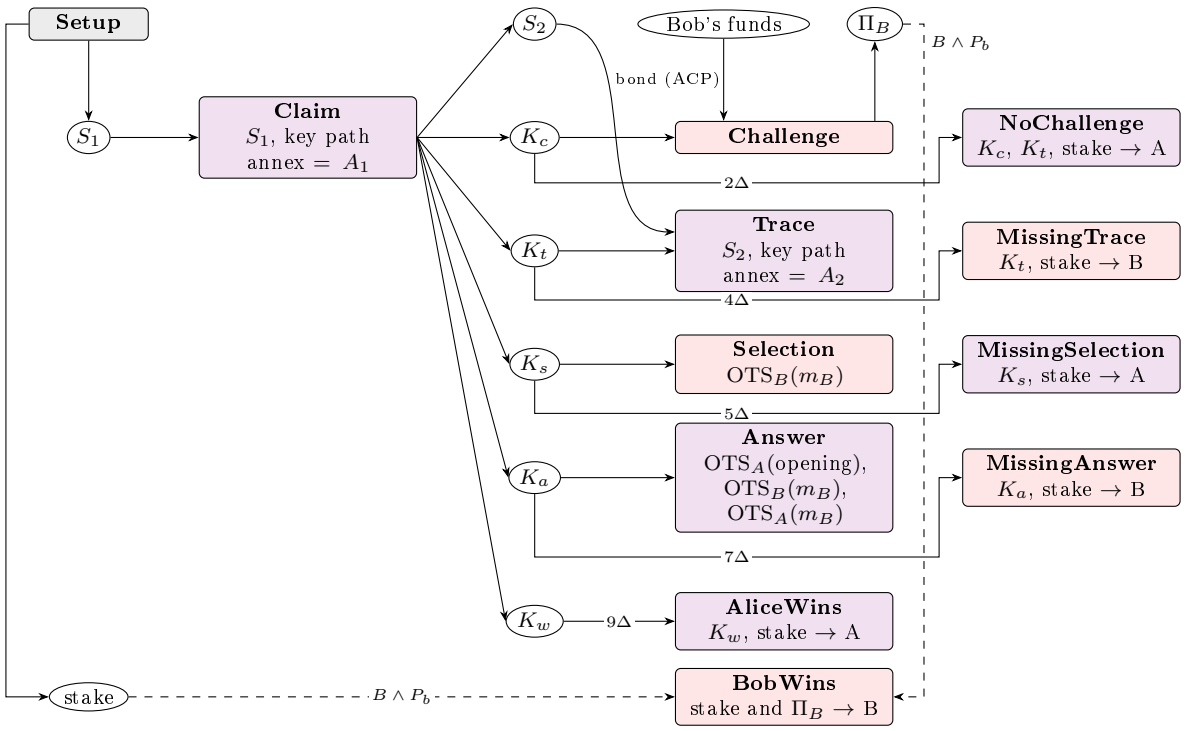
- *Settlement.* Bob evaluates the circuit. If Alice loses, he learns P_b and BobWins Tx takes the stake and his bond; otherwise Alice's AliceWins Tx takes the stake after 9Δ, and the bond after 9Δ from Challenge Tx.

Bob reclaims his bond only with P_b; otherwise it goes to Alice by the pot's timelock, which compensates the publication of A_2 when Bob stops or loses. When Alice stops instead, Bob's timeout transaction takes the stake, which must exceed the bond. Fees are paid through anchor outputs, which the figure omits. The protocol that publishes the trace with the proof uses the same graph without Trace Tx and S_2.

Timing. The deadlines assume that a transaction an online party broadcasts at time t confirms before $t + \Delta$, and that an honest party broadcasts as soon as its move is valid. A deadline must then cover the latest time at which the move it waits for can confirm, not the time at which the waiting party could first act. Where a cancellation and a move compete for the same connector – NoChallenge Tx and Challenge Tx for K_c, MissingSelection Tx and Selection Tx for K_s – the move can still confirm up to Δ after the cancellation becomes valid and is broadcast, and the response it calls for needs one Δ more; a response made of two dependent transactions needs two. Bob decides within Δ of Claim Tx and confirms within Δ more, so NoChallenge Tx is valid at 2Δ; a challenge can then confirm until 3Δ, and Trace Tx until 4Δ; Selection Tx confirms by 5Δ, or, racing MissingSelection Tx, until 6Δ, and Answer Tx by 7Δ; BobWins Tx by 8Δ, one Δ before AliceWins Tx. Shorter deadlines – one Δ per step – let a challenger who confirms just before a pending cancellation leave Alice without time to respond.

The file A_2. A_2 is the canonical trace of A_1 with two additions. The fields of A_1 that are not whole words of the trace are packed into *value words* exactly as they appear in A_1; every other 32-byte word of A_1 is already a word of the trace, which contains every word of the proof that the verifier reads. The chaining state of A_1's annex hash, SHA256(CompactSize$\|$*annex*), is stored after every r message blocks. The chaining states are *interleaved* with the words they hash: the words of A_1 appear in increasing trace position, and each chaining state is placed right after the last word its blocks read. A CHAIN check then reads a short run of consecutive elements, which one multiproof opens with shared siblings. Storing all chaining states at the end of the file would be equally sound, since locality counts positions, not contiguity, but each CHAIN check would open two distant regions, with two inclusion paths. Four kinds of local check are added:

- CHAIN: the next stored state is r compressions of the previous one over the next r message blocks, each block being two words of A_1 read from the trace; the state after the public prefix block and the final padding are constants;



Output	Spending condition (one Taproot leaf per row)	Transaction that uses it
stake	$\langle A, B\rangle$	the timeouts and AliceWins Tx
	$B \wedge H(P_b)$	BobWins Tx
S_1, S_2	Alice's key, key path only (publication slots)	Claim Tx; Trace Tx
K_c	$\langle A, B\rangle$	Challenge Tx
	$\langle A, B\rangle \wedge \mathrm{CSV}\, 2\Delta$	NoChallenge Tx
K_t	$\langle A, B\rangle$	Trace Tx, or NoChallenge Tx
	$\langle A, B\rangle \wedge \mathrm{CSV}\, 4\Delta$	MissingTrace Tx
K_s	$\langle A, B\rangle \wedge \mathrm{OTS}_B(m_B)$	Selection Tx
	$\langle A, B\rangle \wedge \mathrm{CSV}\, 5\Delta$	MissingSelection Tx
K_a	$\langle A, B\rangle \wedge \mathrm{OTS}_A(\text{opening}) \wedge \mathrm{OTS}_B(m_B) \wedge \mathrm{OTS}_A(m_B)$	Answer Tx
	$\langle A, B\rangle \wedge \mathrm{CSV}\, 7\Delta$	MissingAnswer Tx
K_w	$\langle A, B\rangle \wedge \mathrm{CSV}\, 9\Delta$	AliceWins Tx
Π_B	$B \wedge H(P_b)$	BobWins Tx
	$A \wedge \mathrm{CSV}\, 9\Delta$ (from Challenge Tx)	Alice, after a timeout

Figure 3: Transaction graph of the trace on demand, for one designated challenger, and the spending conditions of its outputs. In the table, $\langle A, B\rangle$ is a signature pre-signed by Alice and Bob at setup, A and B a fresh signature by Alice or Bob, $H(x)$ the revelation of a preimage x, $\mathrm{CSV}\, t$ a relative timelock of t, and $\mathrm{OTS}_X(v)$ one-time signatures by X on v; conditions in a row are joined by $\wedge$. Violet: Alice's transactions; red: Bob's. Ellipses are outputs, solid arrows spends, labels relative timelocks counted from Claim Tx (for Π_B, from Challenge Tx); dashed arrows are spends with P_b, which Alice's garbled circuit releases when she loses. Challenge Tx's pre-signature is `ALL|ANYONECANPAY` so that Bob can add his bond, and Trace Tx's, on K_t, is `SIGHASH_ALL`, which fixes Trace Tx up to its witnesses. m_B: the selected check (or a malformation proof) and Bob's copies of the authentication of Claim Tx and Trace Tx. Anchor outputs are omitted.

- FINAL: the final state equals h_{A_1}, as copied by Bob and authenticated in A_1's slot;
- BIND: a small proof field in the trace equals its slot in the value words;
- VPAD: the value words' unused bytes are zero.

Consider, as in Section 8.2, the values Alice can open from A_2. If all checks pass on them, the words that make up A_1 hash to h_{A_1}; by the free-start argument of [23, Theorem 5.9], applied backward along the chain, they *are* A_1, and the rest is the canonical trace of an accepting verification of A_1. Hence, if V rejects A_1, Alice cannot answer every check of any A_2 she publishes, including one built on a different, valid proof. The two cases of Section 8.2 then apply to A_2: if it is canonical, Bob finds an unanswerable check by evaluating the checks on the published bytes of A_1 and A_2; if it is not, he proves it malformed.

What it costs. The proof is paid always: $|A_1|$ annex bytes, a quarter of a vbyte each, plus the 64-byte prefix block. The trace is paid only in a dispute: A_2 is the trace with $\lceil b/32 \rceil$ value words for b bytes of small fields and $\lceil m/r \rceil$ chaining states for the m message blocks of A_1's annex hash, and when these fit in the power-of-two padding of the trace's FlatMerkle file, A_2 costs what the trace alone would. A challenge costs what Section 7 measures for one check, plus the slot authentications (Table 8).

8.4 Grouped publication

On demand, a challenge still makes Alice publish the trace of the whole verification, because A_1 is authenticated by a single annex hash, and the chain that binds A_2 to it runs over all of A_1. For a separable verifier (Definition 8.2) that is unnecessary.

Publication. Claim Tx spends one slot per part P_i and carries each part in the annex of its own input, once, however many groups read it. An input's signature message contains its own annex hash and no other input's [44], so each part is authenticated on its own by the slot mechanism of Section 3.4; with the input index as a circuit input, a template takes at most three compressions. Claim Tx's non-witness data stays fixed, so its identifier, and the pre-signed graph built on it, do not change.

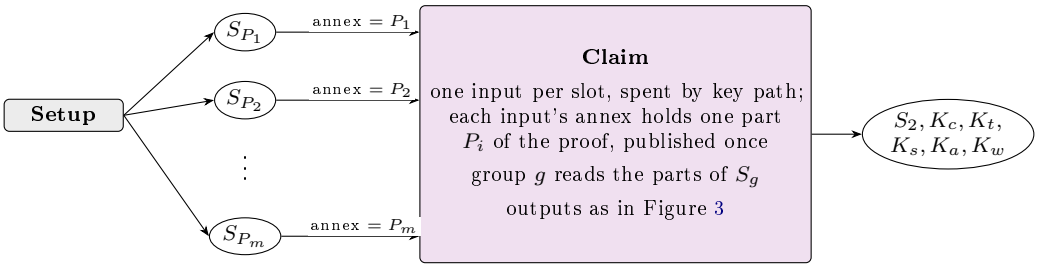
Dispute. Four steps of Section 8.3 change; Figure 4 shows what changes in the transaction graph:

- *Challenge.* Bob, who has verified the whole proof off-chain, names a group g whose verifier V_g rejects, and commits g with one-time signatures in Challenge Tx's witness.
- *Trace.* Alice publishes A_2^g, the trace of V_g alone, with the value words and chains of the parts of S_g, and one FINAL check for each.
- *Selection.* m_B carries the check index, Bob's copy of the authentication of Trace Tx and, for a FINAL check, his copy of the authentication of the part it names (see below).
- *Answer.* Alice re-posts Bob's signatures on g, from Challenge Tx, and on m_B, and co-signs both, so g enters the circuit as Bob committed it.

Neither party can change g after Challenge Tx: Bob signs it once, and Alice can only re-post that signature. Nor can Alice answer from another group's trace: adjudicated under g, its FINAL checks meet the annex hashes of g's sources, and checks that depend on the group, such as positions derived

from it, fail. If all checks of A_2^g pass, then, as in Section 8.3 for each part of S_g, the words of A_2^g that make up those parts are the published parts, and the rest of A_2^g is the canonical trace of an accepting V_g; if V rejects P, V_g rejects for the group Bob names. A part read by several groups is the same published annex for all of them, so they agree on it without any check between groups.

Selective authentication. Every check is adjudicated against Trace Tx's annex hash, since Alice's opening is a multiproof against it, and so needs Bob's copy of Trace Tx's authentication. Only a FINAL check compares with the annex hash of a published part, and only of the one part it names; the malformation branch concerns A_2^g alone. No check needs the authentication of two published parts. So m_B carries the check index (3 bytes), the group (1 byte), Bob's copy for Trace Tx and one more copy, which only a FINAL check reads: the circuit derives the slot it must verify, the input of Claim Tx that carries the named part, from the committed group and check, never from Alice. m_B is then $3+1+2\cdot 97 = 198$ bytes whatever the number of sources, against $3+1+(|S_g|+1)\cdot 97$ with a copy of every source, 295 for two. Each byte of m_B is committed three times, in Selection Tx and twice in Answer Tx, so each copy left out saves $3\cdot 97$ committed bytes, 8,366 vbytes with WISCH.



What changes	Trace on demand (Figure 3)	Grouped publication (this figure)
Claim Tx: inputs and annexes	one slot, S_1; one annex, A_1	$S_{P_1}, \ldots, S_{P_m}$; one annex per part P_i
K_c: leaf spent by Challenge Tx	$\langle A, B\rangle$	$\langle A, B\rangle \wedge \mathrm{OTS}_B(g)$ (Bob commits group g)
Trace Tx: annex	A_2, trace of the whole proof	A_2^g, trace of V_g only
m_B: committed in Selection Tx	check; copies for Claim Tx and Trace Tx	check; copies for Trace Tx and for the part a FINAL check names
K_a: leaf spent by Answer Tx	$\ldots \wedge \mathrm{OTS}_B(m_B) \wedge \mathrm{OTS}_A(m_B)$	$\ldots \wedge \mathrm{OTS}_B(g, m_B) \wedge \mathrm{OTS}_A(g, m_B)$

Figure 4: What grouped publication changes in the transaction graph of Figure 3: above, the publication; below, each element that differs, as it is in Figure 3 and as it becomes. Setup Tx creates one slot per part; Claim Tx spends them all, each by key path with its own part in its own annex, so each part is authenticated by its own input's signature. Bob commits the group g in Challenge Tx; Alice re-posts that signature in Answer Tx and co-signs it, together with m_B. All other transactions, connectors, timelocks and penalties are those of Figure 3.

What A_2^g contains. Figure 5 shows the file Alice publishes when Bob names group g, and what each kind of check opens. For each part of S_g, A_2^g holds a copy of every word of the part, interleaved with the chaining states of the part's annex hash; and it holds the values that V_g computes from them. As in Section 8.3, each chaining state is placed right after the last word its blocks read, and each word of a part appears in A_2^g once, as the trace word that V_g reads. The CHAIN and FINAL

checks of each copy tie it to the annex that Alice signed in Claim Tx: if a copy differs from the published part, its chain does not end at that part's annex hash. The checks of V_g then read only words of A_2^g, so they are local to one file, although the parts were published in different annexes. Parts that g does not read are neither copied nor authenticated.

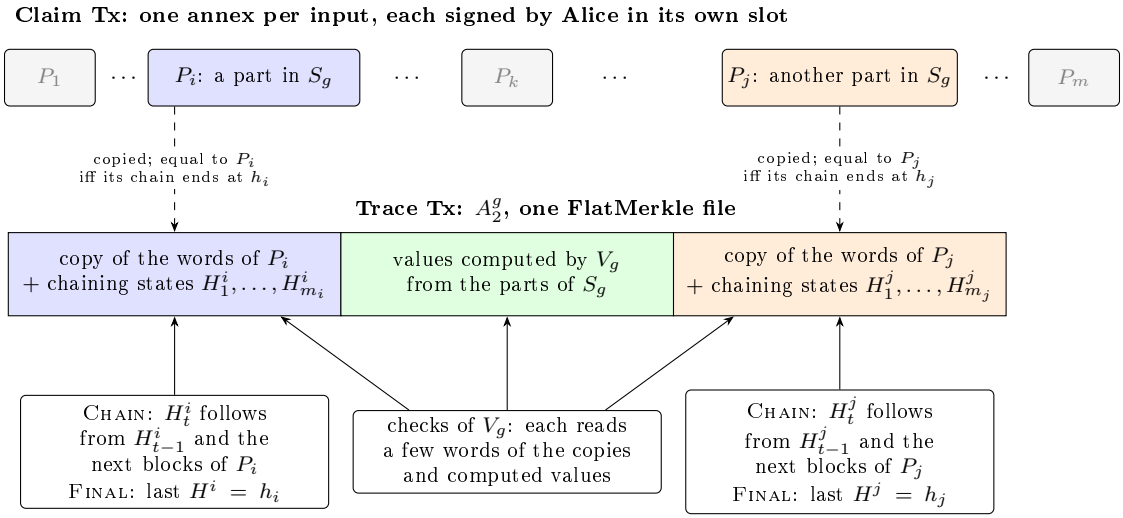


Figure 5: What Alice publishes in Trace Tx when Bob names group g, and what the checks open, drawn for a group that reads two parts; in general A_2^g has one copy, one chain and one Final check per part of S_g. Above, the parts published in Claim Tx; below, the file A_2^g, drawn grouped by role (in the layout, the chaining states are interleaved with the words they hash, each placed right after the last word its blocks read, and fields that are not whole words are held in value words, tied to the copies by Bind checks). Bob selects one check, and Alice opens the words it reads (solid arrows) with one multiproof against A_2^g's annex hash. The Chain and Final checks make each copy equal to the part published in Claim Tx, whose annex hash the circuit authenticates, for a Final check, with Bob's copy of Alice's signature in that part's slot; the checks of V_g then work on words of A_2^g alone. Parts outside S_g are neither copied nor read.

What it costs. The trace a challenge forces is that of V_g: the parts of S_g, their chains, and what V_g computes from them. It depends on the sizes of the parts g reads, not on the number of groups. In the common-part case every trace carries all of C, which is cheap when C is small, as a STARK's transcript is, and costly when C grows with the number of groups, as the boundary states of a state machine do (Section 8.6). The price of grouping is one input and one slot per part: about 58 vbytes per additional input of Claim Tx, for its outpoint, sequence and signature, and the 16 vbytes of each annex's prefix block, and one output per slot at setup, 43 vbytes for a P2TR output. The tables below count the input and the prefix block but not the slot's output, nor the fixed overhead of each transaction, about 11 vbytes: with them, a part costs about 117 vbytes instead of 74, which adds 1.4 to 2.1 kvB to the Claim Tx of the examples and changes none of the comparisons. Merging groups trades a larger trace on dispute for fewer inputs.

8.5 Costs of the components

Table 8 collects the costs that a FlatCheck adds to the dispute program of Section 7, independently of the verifier.

Component	SHA-256 compressions	Circuit input or bytes
Slot authentication, per publication	at most 4 (5 with the input index), plus the BIP340 curve check	97 bytes: hash type, annex hash, signature
m_B, committed in Selection Tx	–	check index (3 bytes), two copies: Trace Tx's and A_1's, 197 bytes on demand; Trace Tx's and that of the part a FINAL check names, with the group (1 byte), 198 grouped
CHAIN check, stride r	r	reads at most $2r + 2$ elements
FINAL check	authentication of its part	compares 32 bytes
BIND, VPAD checks	none	read at most three elements
Malformation branch, file of 2^L elements	$2L$ to verify the proof	$64 + 128 + 96(L - 1)$ bytes, plus m_B
Answer Tx, committed	–	opening, plus $2\|m_B\|$ (re-posted and co-signed)
Claim Tx, per additional part	–	about 58 vbytes for the input and 16 for the annex's prefix block

Table 8: Costs a FlatCheck adds to one check of the dispute program. The malformation branch is the worst accepted malformation proof, the revealed prefix and a fraud proof with every level of its path substituted; its compression count is the maximum the reference implementation measures on real malformed publications of 2^6 to 2^{14} elements. Committed bytes are before transaction overhead; their on-chain cost follows from Table 6.

Testing. The reference implementation realizes the three modes on the traced verifier of a toy STARK, the application we treat separately (`stark_ondemand.py`, `stark_grouped.py` and `slot_publication.py`), and grouped publication for any separable verifier given as its parts, a trace order and a check system (`flatcheck.py`), which Section 8.6 tests on two more verifiers. On the STARK:

- on demand, the stored chain reproduces BIP341's annex hash of A_1, an honest A_2 passes every check, and the accused's honest answers keep her stake; for each of several hundred false proofs obtained by changing one byte of an honest one, every A_2 built on the honest, valid proof fails a FINAL, CHAIN or BIND check; each one-byte corruption of a value word or chaining state fails a check; and the challenger's procedure on the published bytes finds a failing check or an accepted malformation proof;
- grouped, the parts reassemble A_1; on 265 proofs, the full verifier accepts exactly when every group's trace passes all its checks; another group's honest A_2, adjudicated under the challenged group, fails a check; and a part changed after publication fails FINAL;
- slot authentication, with real BIP341 signature messages and BIP340 signatures, on a fixed Claim Tx and a Trace Tx whose connector input fixes it: for all seven hash types, with and without an annex, the circuit's template reproduces the signature hash in at most two compressions and a signature verifies only for its own slot, hash type and annex hash; a false proof whose Trace Tx is signed with any hash type other than `0x81` escapes a check that accepts only `0x81` and loses under the full check; a slot spent without an annex loses; answers against a signed, unbroadcast variant of either publication escape under self-supplied authentication and lose under the challenger's copy; and 328 adjudications with forged copies convict no honest accused.

An alternative. Bob could instead prove A_1 invalid with a proof of his own, published as a FlatMerkle file and refutable by Alice in the same way. That removes A_2 and one transaction, but the large publication then falls on the challenger, and he needs a prover for a statement that hashes all of A_1 and runs V on it. We prefer to keep challenges cheap.

8.6 Two examples

Both examples verify a sequence of state transitions: transfers in a ledger, and blocks extending a chain. Such a verifier becomes separable once the state at every group boundary is published: the verifier of group g replays its transitions from the boundary state s_g and must end at s_{g+1}. We publish $s_0, \ldots, s_G$ in blocks of B consecutive states, each block a part of its own, and the transitions of group g as a part Q_g, so that V_g reads Q_g and the one or two blocks holding s_g and s_{g+1}; group 0 also checks s_0 against the public initial state. Consecutive groups need no check between them: the end of group g and the start of group $g+1$ are the same word of the same published block, which both traces must reproduce (Figure 5). If the claimed final state is false, then, replaying from s_0, some first group does not end at its claimed boundary, and Bob names it. The intermediate states are values the verifier computes: they are not published with the claim, and appear, interleaved with the words they follow from, only in the trace of a challenged group.

The boundary states could also form one common part, but every trace would then carry all of them: for n transitions in groups of s, a trace has $\Theta(s)$ words for the group and $\Theta(n/s)$ for the boundary states, at best $\Theta(\sqrt{n})$ in all. With blocks it has $\Theta(s+B)$, independent of n. Both examples are built on `flatcheck.py`, which implements grouped publication for any separable verifier, with either layout of the boundary states.

A toy L2. Eight accounts have public BIP340 keys, and the ledger state is their eight balances, uint32 each: exactly one word. A transfer (*src*, *dst*, *amount*) is signed by *src* over

$$m_i = \text{SHA256}(id \parallel batch \parallel i \parallel src \parallel dst \parallel amount),$$

where *id* names the L2 and i is the transfer's position in the batch, so that no signature can be replayed at another position or in another batch. Alice, the operator, claims that 512 transfers applied to the public state s_0 give s_{32}.

Claim Tx carries the boundary states $s_0, \ldots, s_{32}$ in blocks of eight, five parts of 1,056 bytes in all, and one part Q_g for each group of 16 transfers. A part Q_g is a sequence of blocks, one per four transfers. Each block starts with a record word, which holds four 8-byte records. A record is *src*, *dst* and *amount*, followed by two zero bytes. The four signatures follow the record word, two words each. A part Q_g therefore takes 1,152 bytes.

The trace of group g adds the state after each transfer. Its checks are:

- INIT, in group 0 only: s_0 is the public initial state;
- SIG(j): transfer j's signature verifies under its source's key, over m_i for its position i;
- STEP(j): the state after transfer j is the state before it – s_g, from the copy of its block, for the first transfer – with the amount moved, the accounts in range, the amount within the source's balance, no overflow, and the record's zero bytes zero;
- END: the state after the last transfer is s_{g+1}, from the copy of its block;

with CHAIN and FINAL for each part the group reads. Each check reads at most four words.

A Bitcoin light client in Bitcoin. Script cannot read block headers, so a contract that must react to an event on Bitcoin itself – a bridge paying out once a transaction is confirmed, for instance – has to be shown the headers and convinced they are valid. Alice claims that a list of headers extends a public checkpoint and is valid under Bitcoin's header rules, and thereby fixes the tip's hash, height and cumulative work: each header links to the previous one; its double-SHA-256 hash is at most the target its `nBits` encodes, which must be valid and at most the proof-of-work limit; `nBits` equals the previous one, or, at a retarget height, what Bitcoin Core's `CalculateNextWorkRequired` computes from the period's first and last timestamps; and the timestamp exceeds the median of the previous eleven. The version rules are not checked, nor the rule bounding a timestamp by the validating node's clock, which no proof can check. A valid chain is not necessarily the heaviest; choosing among competing claims, for instance by their work, is up to the protocol that uses them. The state after a header is four words: its hash; the cumulative work; the height, `nBits`, the start time of the retarget period and five timestamps; and six more timestamps, so that the last eleven timestamps, the median and the retarget timespan are all local. The boundary states are published in blocks of eight, and Q_g holds the 80-byte headers of group g, two per five words; the trace adds each header's state after the words that hold the header. For header j, with previous state P – the boundary state s_g, from the copy of its block, for the first header – and new state S, the checks are:

- HASH(j): S's hash is the header's double SHA-256, three compressions;
- POW(j): S's `nBits` encodes a valid target at most the limit, and the hash, as a little-endian integer, is at most that target;
- LINK(j): the header's previous-block field is P's hash;

- BITS(j): the header's `nBits` is S's, and equals P's or, at a retarget height, the retarget of P's over P's period;
- TIME(j): the header's timestamp exceeds the median of P's eleven;
- META(j): S's height, period start and timestamps follow from P's and the header's timestamp, and S's padding is zero;
- WORK(j): S's work is P's plus the block's work, $\lfloor 2^{256}/(target+1) \rfloor$;
- INIT(w), in group 0 only, and END(w), for each of the four words of a state, as in the L2;

with CHAIN and FINAL for each part the group reads. Each check reads at most five words. We measure one retarget period, 2,016 headers, in 42 groups of 48.

What they cost. Table 9 gives the sizes. An honest claim costs its data and the boundary states: 12,151 vbytes for 512 transfers and 45,178 for a retarget period of headers, both well under the standard transaction weight. A dispute adds the trace of one group and one check: about 37,500 and 49,200 vbytes with WISCH when Bob selects a local check, and about 77,100 and 99,800 when he proves A_2^g malformed, because Selection Tx and Answer Tx then carry the malformation proof. These totals span three transactions; the largest, the malformation branch's Answer Tx, is 50,000 and 61,100 vbytes, under the standard weight. Committing the same data as the input of a circuit that verifies everything would cost 1.09 and 4.80 million vbytes with WISCH, and the circuit would verify 512 signatures or 2,016 headers rather than one check.

Table 10 separates what each choice buys. Selective authentication removes one 97-byte copy from m_B, 8,366 vbytes from every dispute. Smaller groups halve the trace, at the price of more inputs in Claim Tx; with one common part they stop paying when the boundary states dominate the trace, as they already do for the headers, whose states are four words: groups of 48 with a common part leave the trace at 65 kB. Boundary blocks halve it again, to 33 kB, for 368 more vbytes in Claim Tx, and cut the dispute by 19% with respect to the common part with the same groups, 27% with respect to the layout with a copy of every source. For the L2, whose states are one word, a common part is as good at 512 transfers, but not as the batch grows: at 4,096 transfers it makes the trace 33 kB and the dispute 46,430 vbytes, while with blocks both stay where they were at 512 transfers, 8 kB and 37,526 vbytes. In both examples, the group index is one byte, so up to 256 groups.

Signatures in the circuit. Every check is adjudicated against Trace Tx's annex hash, so the circuit always verifies one Schnorr signature: Alice's, on Trace Tx, from Bob's copy. A FINAL check compares a chaining state with the annex hash of a part published in Claim Tx, and so also verifies Alice's signature on that part: two signatures. This is what binds A_2^g to the publications it must agree with, and it holds in both examples. The L2's SIG check also needs two: Trace Tx's and the transfer's. Since no check needs three, a circuit that accepts any check contains two BIP340 verifications, the second fed either by a part's slot authentication or by a transfer, in both examples. The header chain's checks add only SHA-256 and integer arithmetic.

Testing. `example_l2.py` and `example_headers.py` check, on small instances – 32 transfers in four groups of eight, and 64 headers of a mined chain on a toy network with Bitcoin's rules, a retarget interval of 16 and a proof-of-work limit of $2^{249}-1$, in eight groups of eight, with four retargets that hit both clamps of the timespan – and in both layouts, one common part and boundary blocks of two states:

	toy L2	header chain
Claim: transitions	512 transfers	2,016 headers
groups	32×16	42×48
boundary blocks of eight	5, 1,056 B	6, 5,504 B
one Q_g	1,152 B	3,840 B
all parts	37,920 B	166,784 B
annexes and inputs of Claim Tx	12,151 vB	45,178 vB
On dispute: A_2^g, words	99	506
FlatMerkle file, hashed	8,128 B	32,704 B
in the annex	2,032 vB	8,176 vB
Checks of a group, words read	65, ≤ 4	438, ≤ 5
Worst check, compressions	31	35
BIP340 verifications	2 (FINAL, SIG)	2 (FINAL)
256-bit products or quotients	–	2 (BITS)
circuit input	838 B	1,030 B
Malformation branch	16, 870 B	20, 1,062 B
Committed in Selection Tx, Answer Tx	198 B, 1,036 B	198 B, 1,228 B
Dispute with WISCH, local check	37.5 kvB	49.2 kvB
malformation branch	77.1 kvB	99.8 kvB
largest transaction	50.0 kvB	61.1 kvB
All parts as circuit input, WISCH	1.09 MvB	4.80 MvB

Table 9: The two examples with grouped publication and boundary states in blocks of eight. "Annexes and inputs": the parts in their annexes, with each annex's prefix block, plus 57.75 vbytes per input beyond the first, before the rest of Claim Tx; the slot outputs created at setup, 43 vbytes per part, and fixed transaction overhead are not included (Section 8.4). Every figure on dispute is the maximum over every check of every group's A_2^g, measured structurally as for the STARK; check costs include the slot authentication of Trace Tx, four compressions besides its BIP340 verification, the multiproof, and, for FINAL, the authentication of the part it names; circuit input includes m_B, 198 bytes. The dispute rows add the trace, Selection Tx and Answer Tx with WISCH, before transaction overhead: for a local check, Selection Tx commits m_B and Answer Tx the opening with m_B re-posted and co-signed; in the malformation branch, both also carry the malformation proof, so the dispute is costlier. The largest transaction is the malformation branch's Answer Tx. BIP340 verifications: every check authenticates Trace Tx's annex hash, and FINAL also the part it names; the L2's SIG verifies a transfer's signature instead. The malformation branch is given as compressions and circuit input. The last row is the alternative of committing every part as the input of a circuit that runs the whole verifier. Reproduced by `example_l2.py` and `example_headers.py`.

	Claim Tx	trace	dispute with WISCH	
Layout	vB	bytes	local check	malformation
Toy L2, 512 transfers				
one common part, 32 per group, a copy of every source	10,548	16,320	49,780	95,780
selective authentication	10,548	16,320	41,414	87,414
and 16 per group	11,856	8,128	37,526	77,086
boundary blocks of eight, 16 per group	12,151	8,128	37,526	77,086
Toy L2, 4,096 transfers, 16 per group, selective authentication				
one common part	94,680	32,704	46,430	99,790
boundary blocks of eight	97,040	8,128	37,526	77,086
Header chain, 2,016 headers				
one common part, 96 per group, a copy of every source	42,589	65,472	67,588	124,628
selective authentication	42,589	65,472	59,222	116,262
and 48 per group	44,810	65,472	61,062	116,262
boundary blocks of eight, 48 per group	45,178	32,704	49,190	99,790

Table 10: What each choice buys, maxima over every group, chain stride chosen per layout. "A copy of every source": m_B carries Bob's copies for C, Q_g and Trace Tx, 295 bytes; otherwise selective authentication, 198 bytes. With a common part, every trace carries all the boundary states, so the trace and the dispute grow with the batch; with boundary blocks they do not. Claim Tx excludes the slot outputs, 43 vbytes per part; counting them adds 0.7 to 2.1 kvB and changes no comparison. Reproduced by `example_l2.py` and `example_headers.py` with `--large`.

- every group's honest A_2^g reproduces the annex hashes of its sources and passes all its checks, and each of 132 and 130 (L2) and 768 and 650 (headers) honest answers keeps Alice's stake, while a missing answer loses;
- selective authentication, on a Claim Tx with one slot input per part and real BIP341 signature messages and BIP340 signatures: for every FINAL check of every group, the slot derived from the group and the check authenticates Bob's copy of exactly the part it names, and the copy of every other input is rejected there;
- for false claims, Bob names the first group whose verifier rejects, and the honest opening of the first failing check of Alice's A_2^g loses: for the L2, a forged signature, a signature by another account and a signature replayed at another position (SIG), a signed overspend with the boundary states restated to match (STEP), a false boundary state (END) and a false initial state (INIT); for the headers, insufficient work (POW), a broken link (LINK), a missing retarget and an `nBits` changed within a period (BITS), a timestamp not above the median (TIME), inflated cumulative work in a boundary state (END) and a false checkpoint (INIT);
- an A_2^g whose copy of a boundary part carries the true boundary state instead of the published false one fails FINAL for that part, or CHAIN if it keeps the published chaining states, and one built on a corrected Q_g fails FINAL for Q_g;
- each group's honest A_2^g, adjudicated under every other group, fails a check (12 and 56 cases per layout);
- no check reads a padding word of A_2^g; with every padding word set to a non-zero value and the file rebuilt, every check still passes, every honest answer keeps the stake, and the words

at the semantic positions are unchanged;

- on 87 and 120 one-byte changes of published words, the direct verifier accepts exactly when every group's trace passes all its checks, and the first group it rejects is the first whose trace fails;
- the header rules reproduce the genesis block's hash and the vectors of Bitcoin Core's `pow_tests.cpp` for the retarget, the block's work and invalid targets.

With `--large`, both build the measured configurations with real data – 512 signed transfers, and 2,016 mined toy-network headers, with boundary blocks of eight – and answer every check of two groups, one of which reads two blocks.

8.7 Several challengers

The graph of Figure 3 has one designated challenger. A protocol that trusts that one of n designated challengers is honest and online needs all n of them to be able to challenge, and each must be able to win on his own, since none can rely on the others. The graph extends to n challengers by sharing what does not depend on the challenger and duplicating what does. Claim Tx is published once. The trace A_2^g depends only on the group and on what Alice published, so it is published at most once per group, however many challengers name it. The adjudication cannot be shared: Alice commits her opening with one-time signatures whose preimages are labels of a garbled circuit, and the construction does not safely release labels for several independent input assignments, so each challenger has his own circuit, one-time-signature keys, Selection Tx and Answer Tx.

Static connectors and a common calendar. Every connector is an output of Claim Tx, whose non-witness data is fixed at setup: for each challenger i, K_c^i, K_t^i, K_s^i and K_a^i; for each group g, the trace slot S_T^g; and one connector U. Every relative timelock of the graph is on one of these outputs, so every deadline counts from Claim Tx, for every challenger; the only other timelock is the pot's, which counts from Challenge Tx_i (Table 11). The deadlines follow the timing model of Section 8.3: each covers the latest time at which the move it waits for can confirm, including a move that confirms just before a pending cancellation, plus the time the response needs. Figure 6 shows the control graph of one challenger and one group, Figure 7 the stake and the bond, and Table 12 every transaction.

The bond and the challenge. Alice pre-signs Challenge Tx_i with `ALL|ANYONECANPAY`: the signature fixes its only output, the pot Π_i of value b, and lets B_i add an input, so the transaction is valid only if B_i funds it, with a UTXO of the exact value prepared in advance, since `ALL` leaves no room for change. Its txid therefore depends on B_i's UTXO and is not known at setup, and nothing pre-signed spends its output: the pot is spent with fresh signatures only. Challenge Tx_i and NoChallenge Tx_i both spend K_c^i, so a confirmed challenge cannot be cancelled, and a challenge that arrives after NoChallenge Tx_i is invalid and leaves B_i's funds where they were. B_i commits the group g_i with his one-time signatures in Challenge Tx_i's witness.

Forcing the right trace. Trace Tx_g spends S_T^g by key path with A_2^g in its annex, and its outputs are fixed: one *ticket* tk_g^i per challenger, so that it is fixed up to its witnesses, as a slot requires, although nobody knows at setup who will challenge g. Alice acknowledges the trace to challenger i with AckTrace $\mathrm{Tx}_{i,g}$, pre-signed in one version per group: version g spends K_t^i through a leaf that checks B_i's one-time signature on the value g, and the ticket tk_g^i. Before 5Δ this is the only

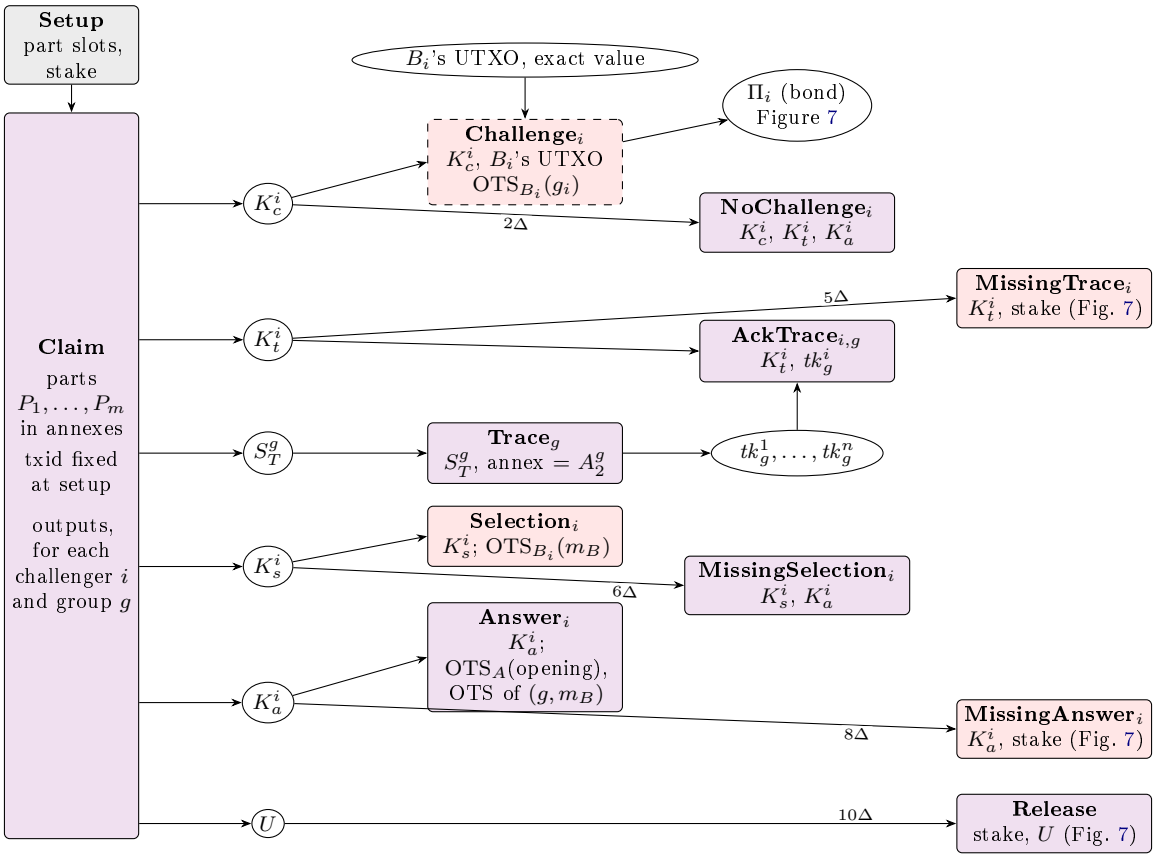
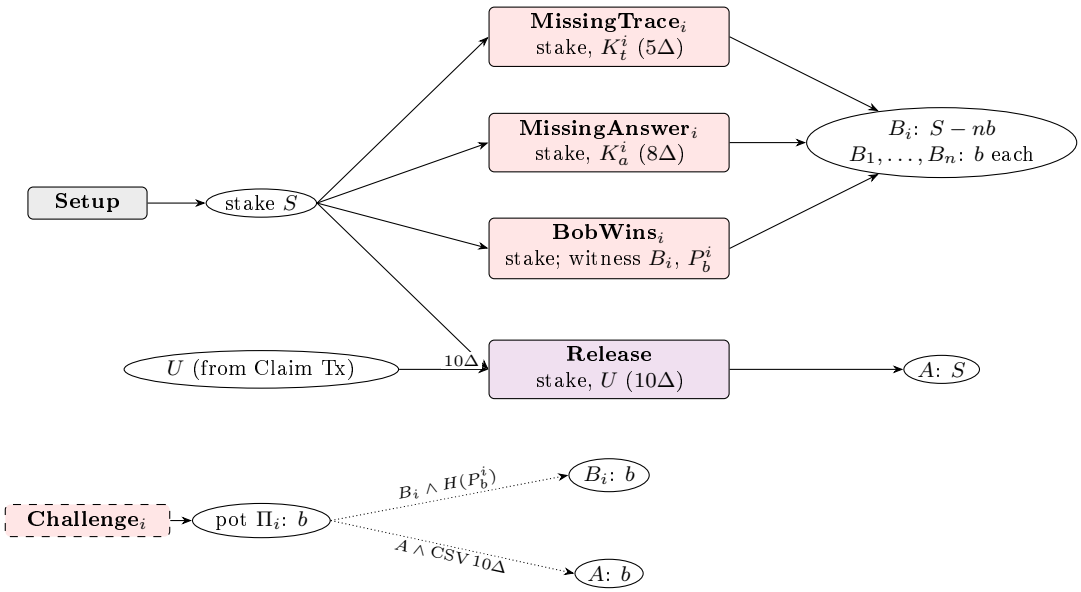


Figure 6: Control graph of challenger i and group g. Violet: Alice's transactions; red: the challenger's. Each box lists all its inputs; arrows are drawn from the input that names the move, labelled with its relative timelock, counted from Claim Tx. Challenge Tx_i (dashed) has a txid fixed only when B_i adds his funding, so it creates no output that a pre-signed transaction spends: its only output is the pot Π_i. NoChallenge Tx_i and MissingSelection Tx_i also consume K_a^i (and NoChallenge Tx_i K_t^i), which disables the punishments they would otherwise leave open. Trace Tx_g is published at most once and serves every challenger who names g, through his ticket.



Output	Spending condition (one Taproot leaf per row)	Transaction
stake	$\langle A, B_1, \ldots, B_n \rangle_{\text{MissingTrace}_i}$, for each i	MissingTrace Tx$_i$
	$\langle A, B_1, \ldots, B_n \rangle_{\text{MissingAnswer}_i}$, for each i	MissingAnswer Tx$_i$
	$\langle A, B_1, \ldots, B_n \rangle_{\text{BobWins}_i} \wedge B_i \wedge H(P_b^i)$, for each i	BobWins Tx$_i$
	$\langle A, B_1, \ldots, B_n \rangle_{\text{Release}}$	Release Tx, with U
U	$\langle A, B_1, \ldots, B_n \rangle_{\text{Release}} \wedge$ CSV 10Δ (from Claim Tx)	Release Tx
Π_i	$B_i \wedge H(P_b^i)$	B_i, any time
	$A \wedge$ CSV 10Δ (from Challenge Tx$_i$)	Alice

Figure 7: The stake and the bond. Every spend of the stake is pre-signed by Alice and all n challengers ($\langle\cdot\rangle$, subscript: the transaction signed, `SIGHASH_ALL`), so its outputs are fixed: the three transactions that take the stake pay the winner $S - nb$ and every challenger his bond b, and Release Tx pays Alice. Release Tx also spends U, an output of Claim Tx whose 10Δ timelock starts the clock at Claim Tx, whatever the stake's age. BobWins Tx$_i$ also needs B_i's fresh signature, so that no one else can broadcast the variant that pays B_i, and the secret P_b^i that Alice's circuit for B_i releases when she loses. The pot is spent with fresh signatures only (dotted), since Challenge Tx$_i$'s txid is not known at setup.

Deadline (from Claim Tx)	Who must have acted	Otherwise
2Δ	B_i: Challenge Tx_i confirmed	Alice: NoChallenge Tx_i
5Δ	Alice: Trace Tx_{g_i} and AckTrace Tx_{i,g_i} confirmed	B_i: MissingTrace Tx_i takes the stake
6Δ	B_i: Selection Tx_i confirmed	Alice: MissingSelection Tx_i
8Δ	Alice: Answer Tx_i confirmed	B_i: MissingAnswer Tx_i takes the stake
9Δ	B_i: BobWins Tx_i confirmed, if he learned P_b^i	–
10Δ	–	Alice: Release Tx; the pots' Alice leaves open (10Δ from Challenge Tx_i)

Table 11: The common calendar, with the timing model of Section 8.3: each deadline is the latest time at which the move it waits for can confirm. A challenge can confirm until 3Δ, racing NoChallenge Tx; Trace Tx and AckTrace Tx, two dependent transactions, need 2Δ more. A selection can confirm until 7Δ, racing MissingSelection Tx; Answer Tx needs Δ more. Between BobWins Tx and Release Tx there is one Δ of margin. Every deadline counts from Claim Tx, except the pot's, which counts from Challenge Tx_i.

way to spend K_t^i other than NoChallenge Tx_i; after it, MissingTrace Tx_i takes the stake. To use version g, Alice needs the ticket, which exists only once Trace Tx_g has confirmed, and B_i's signature on g, whose preimages she copies from Challenge Tx_i's witness. B_i signed g_i only, so to escape MissingTrace Tx_i she must publish $A_2^{g_i}$. That it is correct is the adjudication's business: the circuit takes g_i as B_i committed it and authenticates the trace against $S_T^{g_i}$, the slot it derives from g_i; a Selection Tx_i whose copy of that authentication does not verify – for instance because B_i selected before the trace was published – convicts no one. A trace published for an earlier challenger serves a later one: the later challenger's ticket is already there.

Disabling punishments. Every resolution in Alice's favour must consume the connectors of the punishments still open, otherwise a challenger could take the stake through a timeout that he provoked himself by omitting his own step. NoChallenge Tx_i spends K_c^i, K_t^i and K_a^i; MissingSelection Tx_i spends K_s^i and K_a^i; AckTrace Tx spends K_t^i and Answer Tx_i spends K_a^i (Table 13). An honest Alice therefore keeps her stake whatever any challenger does: after every resolution in her favour, neither MissingTrace Tx_i nor MissingAnswer Tx_i can be built, and no circuit releases its secret P_b^i, since every answer of an honest Alice passes.

Every spend of the stake is pre-signed by everyone. The stake's leaves require a pre-signature of Alice and all n challengers, with `SIGHASH_ALL`: an honest challenger signs only the transactions of the graph, so no coalition of Alice and other challengers can redirect the stake. The three transactions that take it – MissingTrace Tx_i, MissingAnswer Tx_i and BobWins Tx_i – have fixed outputs: the winner's prize, $S - nb$, and one output of value b for every challenger; BobWins Tx_i also needs B_i's fresh signature, so that no one else can broadcast the variant that pays him, and P_b^i. Alice recovers the stake only with Release Tx, which spends it together with U, whose timelock of 10Δ counts from Claim Tx: one transaction, valid one Δ after the last winning spend is due.

Transaction	Inputs (leaf)	Witness	Lock	Outputs	Pre-signed by, sighash
Claim Tx	part slots (key path)	Alice's signatures, annexes $P_1, \ldots, P_m$	–	$K_c^i, K_t^i, K_s^i, K_a^i$ $(\forall i)$, S_T^g $(\forall g)$, U, anchor	fixed by the graph (slots)
Challenge Tx_i	K_c^i $(\langle A, B_i\rangle \wedge \text{OTS}_{B_i}(g))$, B_i's UTXO	B_i's OTS on g, B_i's signatures	–	Π_i (b)	A: `ALL|ANYONECANPAY`; B_i fresh
NoChallenge Tx_i	K_c^i, K_t^i, K_a^i	–	2Δ	anchor	A, B_i: `ALL`
Trace Tx_g	S_T^g (key path)	Alice's signature, annex A_2^g	–	$tk_g^1, \ldots, tk_g^n$	fixed by the graph (slot)
AckTrace $\text{Tx}_{i,g}$	K_t^i $(\langle A, B_i\rangle \wedge \text{OTS}_{B_i}(g))$, tk_g^i	B_i's OTS on g, copied	–	anchor	A, B_i: `ALL`; G versions
MissingTrace Tx_i	K_t^i, stake	B_i	5Δ	B_i: $S - nb$; B_j: b $(\forall j)$	$A, B_1..B_n$: `ALL`
Selection Tx_i	K_s^i $(\langle A, B_i\rangle \wedge \text{OTS}_{B_i}(m_B))$	B_i's OTS on m_B	–	anchor	A, B_i: `ALL`
MissingSelection Tx_i	K_s^i, K_a^i	–	6Δ	anchor	A, B_i: `ALL`
Answer Tx_i	K_a^i $(\langle A, B_i\rangle \wedge \text{OTS}_A(\cdot) \wedge \text{OTS}_{B_i}(g, m_B) \wedge \text{OTS}_A(g, m_B))$	Alice's OTS on the opening; B_i's on (g, m_B), copied; Alice's on (g, m_B)	–	anchor	A, B_i: `ALL`
MissingAnswer Tx_i	K_a^i, stake	B_i	8Δ	as MissingTrace Tx_i	$A, B_1..B_n$: `ALL`
BobWins Tx_i	stake	B_i's signature, P_b^i	–	as MissingTrace Tx_i	$A, B_1..B_n$: `ALL`; B_i fresh
Release Tx	stake, U	–	10Δ	A: S	$A, B_1..B_n$: `ALL`
pot Π_i spends	Π_i	B_i and P_b^i; or A	–; 10Δ	free	fresh signatures only

Table 12: Every transaction of the graph. "Lock": relative timelock on the input that names the move, counted from Claim Tx (for the pot's Alice leaf, from Challenge Tx_i). Outputs of fixed transactions also carry a fee anchor. Every input of a pre-signed transaction is an output of Setup Tx, Claim Tx or Trace Tx_g, whose txids are fixed at setup; only Challenge Tx_i's txid is not, and no pre-signed transaction spends its output.

What B_i does	Alice's move, and when	What it consumes
does not challenge	NoChallenge Tx_i, at 2Δ	K_c^i, K_t^i, K_a^i
challenges	Trace Tx_{g_i} (if not yet published) and AckTrace Tx_{i,g_i}, before 5Δ	K_t^i
challenges, does not select	MissingSelection Tx_i, at 6Δ	K_s^i, K_a^i
challenges and selects (even late, before K_s^i is spent)	Answer Tx_i, before 8Δ	K_a^i
selects without challenging	NoChallenge Tx_i, at 2Δ	K_c^i, K_t^i, K_a^i

Table 13: Every resolution in Alice's favour consumes the connectors of the punishments still open: after it, neither K_t^i nor K_a^i remains, so neither MissingTrace Tx_i nor MissingAnswer Tx_i can be built. In particular, a challenger who challenges and then neither selects nor lets the dispute end cannot take the stake with MissingAnswer Tx_i after Alice's MissingSelection Tx_i, which spends K_a^i.

An honest challenger wins on his own. With a false claim, B_i challenges by 2Δ, naming a group whose verifier rejects. Alice must acknowledge its trace by 5Δ or lose to MissingTrace Tx_i; he selects a failing check, or a malformation proof, by 6Δ; Alice must answer by 8Δ or lose to MissingAnswer Tx_i; if she answers, his circuit releases P_b^i and BobWins Tx_i confirms by 9Δ, before Release Tx is valid. Other challengers cannot interfere: their connectors are their own, and every spend of the stake needs his pre-signature. Alice can still hand P_b^k to a colluding challenger B_k, who then takes the stake: the claim is not accepted, which is what the challengers protect, but the coalition loses little, a limit of any construction in which the accused can choose who wins.

Bonds. The bond compensates Alice for the trace and the answer that a frivolous challenge forces; she collects the pot after 10Δ if the challenger has not reclaimed it with his secret. A challenger whose dispute is pending when another takes the stake might never get that secret, since Alice can stop answering him; the compensation outputs of the stake-taking transactions return his bond from the stake itself, which must therefore be at least the prize plus nb (Table 14).

Case, for challenger B_j	Pot Π_j	Net for B_j
loses against an honest Alice	Alice, at 10Δ	$-b$
abandons (no selection)	Alice, at 10Δ	$-b$
wins (BobWins Tx_j or a timeout)	B_j with P_b^j, or Alice at 10Δ after a timeout	+prize; his b back from Π_j or from the stake
another takes the stake, B_j's dispute still wins	B_j with P_b^j	$+b$ (pot and compensation)
another takes the stake, Alice stops answering B_j	Alice, at 10Δ	0 (compensation b from the stake)
challenge not admitted (NoChallenge Tx_j first)	never created	0

Table 14: Where challenger B_j's bond ends. "Compensation": the output of value b that every stake-taking transaction pays each challenger. A challenger who never challenged also receives it when a cheating Alice is slashed; that is the cheater's money.

Several instances. The stake is created at setup and the graph's clock starts at Claim Tx, through U and the connectors, so the stake's age does not matter. What the graph does not provide is reuse: the stake is locked for one instance, and has no exit if Claim Tx is never published. A protocol with many instances, such as a bridge, can chain them: Release Tx of instance k pays to the stake of instance $k+1$, whose graph pre-signs spends of that output, known since Release Tx is itself pre-signed; any transaction that takes the stake breaks the chain. This is the *stake chain* of some BitVM bridge designs.

Costs. A worst-case dispute, in which n challengers reach Answer Tx and name D distinct groups, publishes the payloads

$$|Claim\ Tx| + D \cdot |A_2^g| + n \cdot \big(|Selection\ Tx| + |Answer\ Tx|\big),$$

plus a one-byte group commitment per challenger: for the header chain of Section 8.6, with WISCH, $45{,}178 + 8{,}192\,D + 40{,}998\,n$ vbytes, or $91{,}598\,n$ per challenger with malformation proofs; about 94,000 vbytes for one challenger, 177,000 for three on one group and 193,000 for three on three groups. The graph adds outputs of 43 vbytes each: $4n + G + 1$ in Claim Tx, n tickets in each published trace

and $n+1$ in the transaction that takes the stake, 2,700 to 2,900 vbytes for three challengers on the header chain, before the inputs and anchors of the small transactions. Per challenger, the setup pre-signs $G+8$ transactions – Challenge Tx, NoChallenge Tx, G versions of AckTrace Tx, Selection Tx, MissingSelection Tx, Answer Tx, MissingTrace Tx, MissingAnswer Tx and BobWins Tx – and one Release Tx; the $3n+1$ spends of the stake are signed by all $n+1$ parties, $O(n^2)$ signatures, for instance with MuSig2. Alice garbles one circuit per challenger, which dominates the setup.

Testing. `multi_challenger.py` models the graph at the level of outpoints, pre-signed transactions, witnesses and relative timelocks: a transaction is available only if its inputs are unspent, their timelocks have expired and the data its witness needs – one-time signatures on chain, a secret – exists. Its scheduler follows the timing model: a party broadcasts a move once it is valid, and the transaction confirms after a delay below Δ chosen adversarially – an honest party's move is usually delayed to the maximum – unless a conflicting transaction confirms first. Whether Alice's answer to a check passes is abstracted to the predicate of Section 8.2, with the slot authentication in m_B, which the examples test on real traces. It checks:

- implementability: every input of a pre-signed transaction is an output of Setup Tx, Claim Tx or a Trace Tx_g, whose txids are fixed at setup, and nothing pre-signed spends Challenge Tx_i's output;
- deterministic schedules: no challenge without the bond's UTXO; a confirmed challenge cannot be cancelled, and a cancelled one leaves the bond with its owner; after each resolution in Alice's favour, no punishment remains available, including MissingAnswer Tx_i after MissingSelection Tx_i; a colluding challenger on a passing group does not stop an honest one on a failing group; Alice answering late, not answering, or publishing another group's trace loses the stake; when another challenger takes the stake, a pending honest challenger is compensated; a Selection Tx that confirms just before a pending MissingSelection Tx, or a Challenge Tx just before a pending NoChallenge Tx, still leaves Alice the time to answer, or to publish and acknowledge the trace; and with every move at its latest, BobWins Tx confirms before Release Tx is valid;
- 6,000 randomized schedules, with up to four challengers and four groups, honest and random strategies for every party and arbitrary outcomes for every race: a true claim and an honest Alice keep the stake and collect every bond; a false claim never returns the stake to Alice and leaves every honest challenger with at least his bond; value is conserved, and no honest party's funds are left behind;
- removing a component breaks an invariant on some schedule: MissingSelection Tx or NoChallenge Tx not consuming K_a^i lets a challenger take an honest Alice's stake; Release Tx not waiting on U lets Alice keep the stake of a false claim; without the compensation outputs, an honest challenger loses his bond; and with one Δ per step, $\Delta, 2\Delta, 3\Delta, 4\Delta, 6\Delta$, an honest Alice loses her stake;
- the single-challenger graph of Figure 3, on the same scheduler: the same two races leave Alice time to respond, and 3,000 randomized schedules keep the invariants with its deadlines $2\Delta, 4\Delta, 5\Delta, 7\Delta, 9\Delta$, while one Δ per step breaks them.

The model does not build serialized transactions or garbled circuits.

9 Related work

FlatMerkle. This paper builds directly on [23], which introduces the structured file, its inclusion proofs, multiproofs and fraud proofs, and their applications to the Bitcoin annex and to a rollup data-availability layer; its related-work section covers authenticated skip lists, linking-based time-stamping and Merkle mountain ranges [23, Section 12].

Uneven splits. The split of Section 4 is the one RFC 6962 uses for Certificate Transparency's Merkle trees [22], applied here to the recursion of a single hash stream rather than to a tree of separate hashes. The append-friendly alternative discussed there corresponds to Merkle mountain ranges [40].

Dispute games over traces. Storing intermediate states so that a long computation can be disputed at a single step is the idea behind Truebit [37], interactive fraud proofs [19], BitVM [4] and BitVMX, which bisects a hash chain over a program's execution trace down to one instruction [25]. A traced proof message applies the same idea non-interactively to a verifier whose trace is short enough to publish in full: the dispute needs no rounds, because every intermediate value is already committed. State-transition fraud proofs for blockchains [1] likewise commit to intermediate state roots so that a light client can be shown a single invalid step.

Refutation instead of verification. Naysayer proofs [34] make the observation of Section 8 a definition: for many proof systems it is far cheaper to convince a verifier that a false proof is invalid than to check a true one, every NP language has constant-size naysayer proofs, and an optimistic verification mode results for constrained verifiers such as smart contracts. Locally refutable verifiers (Definition 8.1) are a structured case of that idea, a refutation being one check and its reads; FlatCheck adds the commitment, publication and adjudication on Bitcoin. The notion of locality is that of probabilistically checkable proofs and locally testable codes [3], a few positions anywhere, but refutation here is deterministic and the challenger knows where to look; *locally checkable proofs* [15] are a different notion, from distributed computing.

Garbled circuits on Bitcoin. Garbled circuits [45], here used for their authenticity rather than their privacy [5], move the disputed computation off-chain. BitVM3 [29], Glock [8] and BABE [14] are designed around a SNARK verifier; BitVMX has its own garbled-circuit module [10]; and OHMG [12] garbles arithmetic gates for authenticity rather than privacy. A large part of what these protocols put on-chain is the commitment to circuit inputs, with Lamport or Winternitz one-time signatures [21, 31, 35]; WISCH replaces them with correlated Schnorr signatures at a few times lower cost [13]. Antichain Winternitz reduces Lamport's cost by up to about half while guaranteeing label revelation [38]; Mosaic obtains malicious security by cut-and-choose with an on-chain footprint independent of the number of garbled copies [20]; Argo MAC makes garbled elliptic-curve arithmetic far cheaper [9]; and the post-quantum cost of these components is assessed in [39]. These designs verify the succinct proof inside the circuit; a FlatCheck (Section 8) instead adjudicates one local check of its verifier. Section 7 uses them unchanged; what this paper contributes on that side is a dispute statement small enough to garble directly, without a SNARK, and a message format that keeps its inputs few.

Committing to data in Bitcoin transactions. The annex is reserved by BIP341 for future extensions [44]; using it to publish a FlatMerkle file, and the alternatives within a linearly hashed

annex, are discussed in [23, Sections 9.1–9.3]. Making unstructured annex data standard has been proposed in Bitcoin Core [16], implemented in Libre Relay [41], proposed for inscriptions [33], and in 2026 proposed as a reserved consensus-level form [26]; our layout already has that form. Inscriptions today place data in a tapscript envelope [32], which standard nodes relay, but the envelope splits data into pushes of at most 520 bytes, each preceded by an opcode, which breaks the 64-byte alignment a FlatMerkle file needs. `OP_RETURN` outputs of up to 100,000 bytes are standard since Bitcoin Core 30.0 [6], which makes the annex-free output profile of Section 3.3 relayable for files of up to 2^9 elements; the 222-byte profiles carry an annex and are not. The one-compression commitment of Section 3 and its annex-free variant are, as far as we know, new.

Recursive proofs. Recursive composition of succinct arguments, from incrementally verifiable computation [42] onwards, lets a short proof attest to the correct verification of other proofs, and so compresses *computation*. The recursion here is of a different kind: it compresses only *access to committed data*. Each level is cheap because the statement it makes is about a small object, and checking it still means recomputing the relevant part of a verification, not checking a succinct certificate of it.

10 Conclusion

Publishing data on Bitcoin and verifying properties of it on Bitcoin, against a publisher who may lie, needs four things to fit together. The data must be committed with a hash that Bitcoin already computes and that supports logarithmic proofs of both inclusion and malformation, which a FlatMerkle file in the annex provides. The commitment must be bound to the publisher cheaply, which a 222-byte key-path signature hash does with one SHA-256 compression for data in the annex, and a 190-byte annex-free one with three for data in a standard `OP_RETURN` output, the initial state after the prefix block being derived, never supplied. Claims about the data must themselves be cheaply refutable, which the plain and traced proof messages provide, with a format rule that cannot be gamed and nested claims that settle at four logical elements, 448 published bytes. And the final dispute – a few dozen compressions and one Schnorr verification per commitment – must be adjudicated by Bitcoin, which an interactive bisection game, a chunked script in the manner of BitVM2 with challengers fixed in advance, or a garbled circuit can do, although a single script cannot.

The same pieces verify any locally refutable verifier. Compiled into a trace of local checks and committed as a FlatMerkle file, a proof is refuted, if false, by one failing check, provided the accused must answer, loses on an invalid answer, and is held to the publication that was confirmed, which publication slots ensure under every signature hash type (Section 8). The trace need not be published unless the proof is challenged: the proof alone is paid always, and the trace, bound to it by the hash chain of its annex hash, only in a dispute; for a separable verifier, only the trace of the challenged group and the parts it reads. A state machine becomes separable when its state at every group boundary is published, and, published in blocks, its boundary states keep the dispute independent of the batch: a toy L2 claims 512 signed transfers in 12,200 vbytes, and a Bitcoin header chain of one retarget period, verified on Bitcoin itself, in 45,200, with disputes of about 37,500 and 49,200 vbytes over a local check, and 77,100 and 99,800 over a malformed trace. Verifying STARK proofs this way is the application we treat separately.

Every component but the signatures is hash-based or symmetric, so the protocol becomes post-quantum once Bitcoin adopts post-quantum signatures and inputs are committed with a hash-based scheme, at the price of re-deriving the publication commitment for the new signature message

(Section 7).

Several pieces remain open. Pages (Remark 4.1) would bring the published size down to about $1+2/K$ times the data, and to $N+O(N/\log N)$ blocks with logarithmic proofs, but their multiproofs, fraud and malformation proofs are not built. We have not garbled the dispute circuit or measured its full size, which we expect to be dominated by the elliptic-curve part of BIP340. Fraud-proof claims have the same structure as the inclusion claims we implemented, but are not implemented. The messages measured here are padded to a power of two: the uneven split of Section 4 removes that need for FlatMerkle files in general, but its multiproofs are not yet implemented. A circuit that accepts whichever check is selected, and the serialized transactions of the FlatCheck graph, remain to be built. Finally, as for any use of the annex, BIP341 reserves it for future soft forks and current Bitcoin Core does not relay any spend that carries one [23, Section 9.6]; the annex-free output profile of Section 3.3 is standard, but limited to files of 2^9 elements and four times as expensive per byte.

References

[1] Mustafa Al-Bassam, Alberto Sonnino, and Vitalik Buterin. Fraud and data availability proofs: Maximising light client security and scaling blockchains with dishonest majorities. arXiv:1809.09044, 2018. URL https://arxiv.org/abs/1809.09044.

[2] David Archer, Victor Arribas Abril, Steve Lu, Pieter Maene, Nele Mertens, Danilo Sijacic, and Nigel Smart. 'Bristol fashion' MPC circuits. https://nigelsmart.github.io/MPC-Circuits/, accessed October 2026, 2026. SHA-256 compression circuit: 22,573 AND gates.

[3] Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy. Proof verification and the hardness of approximation problems. *Journal of the ACM*, 45(3):501–555, 1998.

[4] Lukas Aumayr, Zeta Avarikioti, Robin Linus, Matteo Maffei, Andrea Pelosi, Christos Stefo, and Alexei Zamyatin. BitVM: Quasi-Turing complete computation on Bitcoin. Cryptology ePrint Archive, Paper 2024/1995; AFT 2026, 2024. URL https://eprint.iacr.org/2024/1995.

[5] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In *ACM Conference on Computer and Communications Security (CCS)*, pages 784–796, 2012.

[6] Bitcoin Core developers. Bitcoin Core 30.0 release notes. October 2025, 2025. URL https://bitcoincore.org/en/releases/30.0/. `-datacarriersize` default raised to 100,000 bytes, applied to the aggregate of all `OP_RETURN` outputs.

[7] Bitcoin Core developers. Transaction relay policy: `MAX_STANDARD_TX_WEIGHT` = 400,000; taproot spends with an annex are non-standard. Bitcoin Core source code, `src/policy/policy.h` and `src/policy/policy.cpp`, 2025. URL https://github.com/bitcoin/bitcoin/blob/master/src/policy/policy.cpp.

[8] Liam Eagen. Glock: Garbled locks for Bitcoin. Cryptology ePrint Archive, Paper 2025/1485, 2025. URL https://eprint.iacr.org/2025/1485.

[9] Liam Eagen and Ying Tong Lai. Argo MAC: Garbling with elliptic curve MACs. Cryptology ePrint Archive, Paper 2026/049, 2026. URL https://eprint.iacr.org/2026/049.

[10] Fairgate Labs. BitVMX garbled-circuit dispute resolution protocol. Source code, `rust-bitvmx-client`, `src/program/protocols/gc_drp.rs` and `src/program/setup/steps/garbler_step.rs`, 2026. URL https://github.com/FairgateLabs/rust-bitvmx-client/blob/dev/src/program/protocols/gc_drp.rs.

[11] Ramses Fernandez. A formal analysis of FLEX and FLEX2. Cryptology ePrint Archive, Paper 2026/983, 2026. URL https://eprint.iacr.org/2026/983.

[12] Ariel Futoransky, Fadi Barbàra, Ramses Fernandez, Gabriel Larotonda, and Sergio Demian Lerner. OHMG: One hot modular garbling. Cryptology ePrint Archive, Paper 2025/2338, 2025. URL https://eprint.iacr.org/2025/2338.

[13] Ariel Futoransky, Ramses Fernandez, Emilio Garcia, Gabriel Larotonda, and Sergio Demian Lerner. WISCH: Efficient data signing via correlated signatures. Cryptology ePrint Archive, Paper 2025/1650, 2025. URL https://eprint.iacr.org/2025/1650.

[14] Sanjam Garg, Dimitris Kolonelos, Mikhail Sergeevitch, Srivatsan Sridhar, and David Tse. BABE: Verifying proofs on Bitcoin made 1000x cheaper. Cryptology ePrint Archive, Paper 2026/065; ACM CCS 2026, 2026. URL https://eprint.iacr.org/2026/065.

[15] Mika Göös and Jukka Suomela. Locally checkable proofs in distributed computing. *Theory of Computing*, 12(19):1–33, 2016. Conference version: PODC 2011.

[16] Joost Jager. policy: make unstructured annex standard. Bitcoin Core pull request 27926, 2023. URL https://github.com/bitcoin/bitcoin/pull/27926.

[17] Martin Jonas. Optimized SHA256 version working with u4. BitVM repository, pull request 65, 2024. URL https://github.com/BitVM/BitVM/pull/65. Implementation: `bitvm/src/hash/sha256_u4_stack.rs`.

[18] Martin Jonas. Optimizing algorithms for Bitcoin Script. BitVMX Knowledge Hub, 27 June 2024; part 3, 20 September 2024, 2024. URL https://bitvmx.org/knowledge/optimizing-algorithms-for-bitcoin-script.

[19] Harry Kalodner, Steven Goldfeder, Xiaoqi Chen, S. Matthew Weinberg, and Edward W. Felten. Arbitrum: Scalable, private smart contracts. In *27th USENIX Security Symposium*, 2018.

[20] Nakul Khambhati, Mukesh Tiwari, Azz, Sapin Bajracharya, Manish Bista, Liam Eagen, Christian Lewe, and Aaron Feickert. Mosaic: Practical malicious security for garbled circuits on Bitcoin. Cryptology ePrint Archive, Paper 2026/812, 2026. URL https://eprint.iacr.org/2026/812.

[21] Leslie Lamport. Constructing digital signatures from a one way function. Technical Report CSL-98, SRI International, 1979.

[22] 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).

[23] Sergio Demian Lerner. FlatMerkle: Logarithmic-space inclusion proofs over a single linear hash stream. Manuscript, published 14 October 2026; version v1.1 of https://github.com/SergioDemianLerner/FlatMerkle, 2026. URL https://github.com/SergioDemianLerner/FlatMerkle/tree/v1.1.

[24] Sergio Demian Lerner and Ariel Futoransky. FLEX – capital-efficient optimistic bridges with on-demand security bonds for Bitcoin. Cryptology ePrint Archive, Paper 2025/1392, 2025. URL https://eprint.iacr.org/2025/1392.

[25] 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.

[26] LesLie. Defining 0x50 0x00 as unstructured taproot annex data. Delving Bitcoin, 16 June 2026, 2026. URL https://delvingbitcoin.org/t/defining-0x50-0x00-as-unstructured-taproot-annex-data/2620.

[27] Avihu Mordechai Levy. Quantum-safe Bitcoin transactions without soft-forks. StarkWare, 9 April 2026, 2026. URL https://github.com/avihu28/Quantum-Safe-Bitcoin-Transactions.

[28] Robin Linus Woll, Lukas Aumayr, Zeta Avarikioti, Matteo Maffei, Andrea Pelosi, Orfeas Stefanos Thyfronitis Litos, Christos Stefo, David Tse, and Alexei Zamyatin. Bridging Bitcoin to second layers via BitVM2. Cryptology ePrint Archive, Paper 2025/1158, 2025. URL https://eprint.iacr.org/2025/1158.

[29] 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.

[30] Eric Lombrozo, Johnson Lau, and Pieter Wuille. BIP 141: Segregated witness (consensus layer). https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki, 2015. Defines transaction weight: base bytes count 4 weight units, witness bytes 1.

[31] Ralph C. Merkle. A certified digital signature. In *Advances in Cryptology – CRYPTO '89 Proceedings*, volume 435 of *LNCS*, pages 218–238. Springer, 1990. doi: 10.1007/0-387-34805-0_21. Describes Winternitz's one-time signature.

[32] Casey Rodarmor. Inscriptions. Ordinal Theory Handbook, 2023. URL https://docs.ordinals.com/inscriptions.html.

[33] Casey Rodarmor. Store inscriptions in taproot signature annex. ord issue 2405, 2023. URL https://github.com/ordinals/ord/issues/2405.

[34] István András Seres, Noemi Glaeser, and Joseph Bonneau. Naysayer proofs. In *Financial Cryptography and Data Security (FC 2024)*, 2024. doi: 10.1007/978-3-031-78679-2_2. URL https://eprint.iacr.org/2023/1472. Cryptology ePrint Archive, Paper 2023/1472.

[35] Mikhail Sergeevitch, Konrad Staniec, David Tse, Nikhil Vanjani, and Robin Linus Woll. Non-interactive translation of Winternitz signatures to Lamport signatures via secret sharing. Cryptology ePrint Archive, Paper 2026/1684, 2026. URL https://eprint.iacr.org/2026/1684.

[36] Solving Bitcoin. bitcoin-scripts: a library of Bitcoin Script arithmetic and cryptographic primitives. Source code and measurements; `knowledge/primitives/secp256k1-schnorr.md`, `sha256-u4.md`, 2026. URL https://github.com/solving-bitcoin/bitcoin-scripts.

[37] Jason Teutsch and Christian Reitwießner. A scalable verification solution for blockchains. arXiv:1908.04756; first circulated as the Truebit whitepaper, 2017, 2019. URL https://arxiv.org/abs/1908.04756.

[38] Mukesh Tiwari and Aaron Feickert. Antichain Winternitz: Guaranteed garbled-circuit label revelation on Bitcoin with permissionless recovery. Cryptology ePrint Archive, Paper 2026/1568, 2026. URL https://eprint.iacr.org/2026/1568.

[39] Mukesh Tiwari and Aaron Feickert. The post-quantum cost of garbled-circuit Bitcoin bridges. Cryptology ePrint Archive, Paper 2026/2383, 2026. URL https://eprint.iacr.org/2026/2383.

[40] Peter Todd. Merkle mountain ranges. OpenTimestamps server documentation, `doc/merkle-mountain-range.md`, first committed 2012; cited at commit `4f5a3c6`, 2012. URL https://github.com/opentimestamps/opentimestamps-server/blob/4f5a3c6ae56be766cc6d83e31fb5341f78ecad7c/doc/merkle-mountain-range.md. Generalizes Merkle trees to an efficiently-appendable forest of perfect binary trees.

[41] Peter Todd. Standard unstructured annex. Post to the bitcoindev mailing list, 20 March 2025, 2025. URL https://mirror.b10c.me/lists/bitcoindev/Z9tg-NbTNnYciSOh@petertodd.org.

[42] Paul Valiant. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In *Theory of Cryptography Conference (TCC 2008)*, volume 4948 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2008.

[43] Pieter Wuille, Jonas Nick, and Tim Ruffing. BIP 340: Schnorr signatures for secp256k1. https://github.com/bitcoin/bips/blob/master/bip-0340.mediawiki, 2020.

[44] 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 sha_annex commitment.

[45] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 162–167, 1986.

[46] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole: Reducing data transfer in garbled circuits using half gates. In *Advances in Cryptology – EUROCRYPT 2015*, volume 9057 of *LNCS*, pages 220–250, 2015.