URMA: Authenticated Records for Independent Data Recovery

Cristian Zmole
ROSINT

10 October 2026
Non-normative presentation of URMA V0, version byte `00`

Abstract

Data kept by a publisher can become unreachable when a device is lost or the publisher disappears. URMA defines records from which another party can verify and recover the original bytes without the publisher, using data that remains accessible and, for private content, a surviving secret. It has three constructions. A private object is cut into pieces, and each piece is sealed in a fixed-size encrypted record that carries enough authenticated metadata for the holder of the secret to find the object's records, notice a missing one and rebuild the original with no index and no final manifest. An atomic public record binds exact text, a fixed-format image or an opaque profile payload to the key of its author. A larger public object is split into parts that are listed by a bounded two-level inventory; each reference in it names both a transaction and the hash of the complete record expected there. When published on a selected chain, each record is carried by a Taproot commit and reveal transaction pair. Verification keeps separate questions apart: whether bytes are authentic, whether an object is complete, whether a transaction is in the chain, and how deeply it is confirmed. This paper explains the constructions and their rejection conditions; it introduces no consensus mechanism and assumes neither perpetual data availability nor a surviving publisher service.

1 The recovery problem

A device holds an original document. Its owner publishes recoverable records and keeps the necessary recovery secret elsewhere. Later the device is lost, and the software or service that published the document is unavailable. A replacement reader should be able to obtain those records, determine whether they are authentic and complete, and reconstruct the exact document. A saved filename, wallet backup or index entry is not a substitute for either the required bytes or the private recovery root.

The useful boundary is therefore a byte-level recovery contract. It specifies what to retain, what to authenticate, how to find the required positions, and when to refuse a result. A source may return corrupted bytes, omit records or report a stale chain view. These failures require different responses. Recovering one complete object should not depend on recovering every other object that once occupied the device.

URMA supplies three constructions: private encrypted objects, atomic public records, and generic public multipart objects. They use existing hash, symmetric encryption and signature primitives. A record is published by two transactions on a *selected chain*: a *commit*, which creates an output, and a *reveal*, which spends that output and carries the record in its witness (Section 4). The selected chain is one configured network. The specification treats Bitcoin and Litecoin as distinct chain contexts, and TXIDs, references and observations are scoped to the one selected [1, §8.4]. Bitcoin's use of public transaction history provides context [2]; URMA defines no mining rules, no rule for choosing among competing histories and no new consensus. Accessible retained history remains an external dependency.

This paper explains the construction. It does not replace the normative specification, revision 0.11 [1], to which it is subordinate. The specification owns the complete byte layouts and

acceptance rules, and an implementation is written from it, not from this paper. The equations and algorithms here explain that revision, whose version byte is `00`. The construction is independent of any particular application, library or payload interpreter.

A reader who wants to reimplement the construction needs the specification and the cited standards. The references identify the specification by revision and SHA-256 digest, so that any copy can be checked against the one described here. The specification makes conformance depend on public deterministic vectors with exact bytes and expected outcomes, rejections included, and on a reader that uses no URMA binary or SDK [1, §12]. This paper assesses no implementation.

2 Records, roles and retained material

Every record begins with the eight bytes

$$\texttt{55 52 4d 41} \parallel \texttt{00} \parallel \text{kind}[1] \parallel \texttt{00 00}.$$

These eight bytes are the *common prefix*: the marker, which is ASCII `URMA`, then the version, the kind and two reserved flag bytes. Kind numbers in this paper are hexadecimal. Unknown versions or kinds and nonzero reserved flags are unsupported, not invitations to infer a layout. Integers are unsigned little-endian, except for the big-endian counter of private encryption (Section 3.2). All size and offset arithmetic is checked before allocation or I/O.

The notation is as follows. $a \parallel b$ is the byte string a followed by b, and 0^8 is eight zero bytes. $|X|$ is the length of X in bytes, and $X[a : b]$ is the part of X from offset a up to but not including offset b. $\mathrm{LE}_{32}(v)$ and $\mathrm{LE}_{64}(v)$ are the unsigned integer v written in 32 or 64 bits, that is four or eight bytes, least significant byte first; $\mathrm{BE}_{64}(v)$ is v in 64 bits, most significant byte first. A field described as u8, u16, u32 or u64 holds an unsigned integer of that many bits in the little-endian form. A number in brackets after a field name, as in kind[1] above, is the length of that field in bytes.

The kinds fall under the three constructions:

Construction	Kinds	Record and recovery boundary
Private	`01`; wrapper `03`	32 768 original bytes per chunk; every record is 32 928 bytes. All authenticated positions recover one object.
Public atomic	`02`, `04`–`06`, `0A`–`0C`	At most 32 768 bytes per complete record; one record per reveal, without reassembly.
Public multipart	`07`–`09`	At most 262 144 bytes per complete record; a mandatory root manifest, leaf manifests and data parts recover one object.

Three roles appear. Private recovery rests on an *archive root* R, exactly 32 uniformly random bytes. It opens every private object sealed under it; those objects form one compartment, and independent compartments use independent roots. The archive root is not a wallet seed. Public authorship rests on a BIP340 x-only secp256k1 key A, the *author*. Funding is the third role: paying for a transaction does not establish the author of its content. The three are distinct as roles, not necessarily as keys: the author and the funder may use the same key, the specification requires no separation, and no field of a record detects whether one key serves more than one role [1, §3]. The archive root is the exception: it is generated on its own, and this permission does not extend to it.

A private record published on a chain also needs a key to sign its reveal (Section 4). The specification calls it a *transport key*: it authorizes the transaction, it is not an author proof for the private content, and it must not be derived from R. It may be reused across records and may be the writer's author or funding key; V0 requires no per-record separation [1, §8.1]. Reveals signed with one key are linked by it.

The receiver needs different material for each construction:

Construction	Locating the records	Authenticating them
Private	Archive root R; then either supplied records, individually or in the kind `03` wrapper, or a scan of the exact chain/network over an accessible history source from a start height no later than the first relevant reveal	The records alone: a discovery tag derived from R selects candidates and an HMAC under a key derived from R decides (Section 3.3); no manifest, journal or index
Public atomic	The reveal TXID on the selected chain, from a scan or a replaceable index	The reveal with its witness and the commit output it spends, amount and script included (Section 4)
Public multipart	The pair (chain/network, reveal TXID of the root manifest); no start height when the source returns a transaction by its TXID	The same witnessed proof for the root manifest, every leaf and every part, all under the root manifest's author A (Section 6.2)

The start height is a lower bound that the reader chooses for the scan. Any height no later than the first relevant reveal serves, down to the beginning of the chain, provided the history source reaches that far; an earlier one only lengthens the scan, so keeping a close one saves work [1, §6]. Private recovery never requires a wallet backup, publisher journal, original filename or implementation database [1, §3], and public multipart recovery starts from a chain and the reveal TXID of a root manifest [1, §7.8]. An index can supply candidates or TXIDs; it cannot replace verification or prove that it has disclosed every object.

3 Private objects

3.1 Canonical fragmentation

A private object is one original byte string. The writer cuts it into consecutive *chunks* of 32 768 original bytes; only the last chunk may be shorter. Each chunk is later sealed into one *record* of exactly 32 928 bytes (Section 3.2). The two sizes measure different things: 32 768 counts the original bytes in a chunk, and 32 928 counts the bytes of a finished record. The split is canonical, meaning that the original alone determines it and a reader accepts no other.

In symbols, let X be the original and $L = |X| > 0$ its length in bytes. Let $C_p = 32\,768$ be the chunk capacity; the subscript marks the private construction, because multipart objects use a different capacity C_m (Section 6). Let N be the number of chunks and i a zero-based chunk index, also called the chunk's position, $0 \le i < N$. Chunk i is the byte string x_i, of length n_i:

$$N = \lceil L/C_p \rceil\,, \qquad n_i = \min(C_p, L - iC_p), \qquad x_i = X[iC_p : iC_p + n_i]. \tag{1}$$

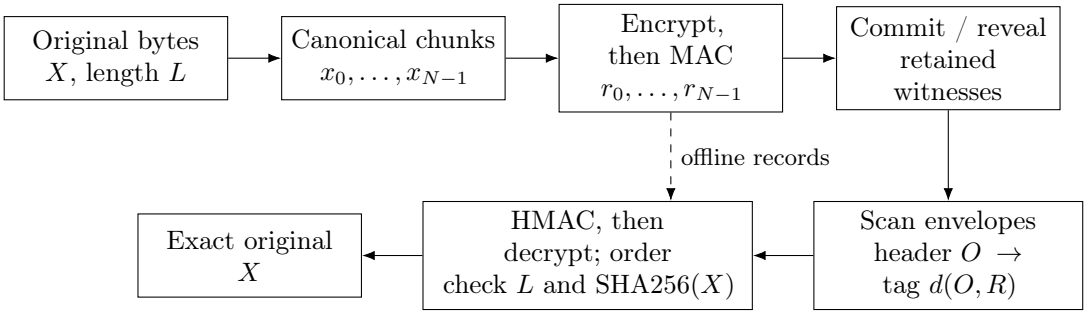
For example, $L = 32\,769$ gives $N = 2$: chunk 0 holds $n_0 = 32\,768$ bytes and chunk 1 holds the one remaining byte, $n_1 = 1$. Empty private objects are not encoded.

Every record stores i and N in 32-bit fields of its public header (Section 3.2), so $N \le 2^{32} - 1$. Because the chunk capacity is fixed, that field width is the only upper bound the format places on the size of a private object: $L \le (2^{32} - 1)\, C_p$. The length L has a 64-bit field of its own, but a reader requires $N = \lceil L/C_p \rceil$, so the count is what bounds it. The specification sets no smaller ceiling, and it requires a client's refusal for lack of capacity to be reported separately from invalid data [1, §5.1]. Table 1 gives the resulting figure and what limits an object in practice.

Each time an object is sealed, the writer draws one new independent OS-random 32-byte object identifier O, shared by all of that object's records. Before encryption, chunk i is placed in a *body* B_i of six fields: the SHA-256 digest of the whole original, its length L, a content-type hint t, four reserved flag bytes that are zero, the chunk itself, and random padding P_i that fills whatever capacity the chunk leaves unused:

$$B_i = \mathrm{SHA256}(X) \parallel \mathrm{LE}_{64}(L) \parallel \mathrm{LE}_{32}(t) \parallel \mathrm{LE}_{32}(0) \parallel x_i \parallel P_i, \tag{2}$$

where $|P_i| = C_p - n_i$ and the padding bytes are OS-random, so every body is exactly $48 + 32\,768 = 32\,816$ bytes. The body does not store n_i. A reader derives it from L and i once the record is authenticated (Section 3.2), keeps exactly the first n_i bytes that follow the four metadata fields, and discards the rest as padding [1, §5.4]. No field can express a different cut, which is why a reader accepts no other split. The type hint t is 0 (opaque), 1 (text), 2 (image), 3 (audio) or 4 (video); other values are unsupported. A hint neither transforms the bytes nor authorizes execution. Because the digest, length and hint repeat in every body, any single authentic record carries the metadata needed to recognize a missing position.



Independent receiver: surviving archive root R + accessible records.
Chain scanning also needs the exact network, a history source and a start height no later than the first reveal.

Figure 1: Private recovery follows the records, not a publisher database. A reader holding the archive root R selects candidate records by a keyed tag and authenticates each one before decrypting it (Sections 3.2 and 3.3). A chain source supplies candidates inside the transaction envelopes of Section 4; an offline source can supply the same canonical records directly.

3.2 Key derivation and authenticated encryption

Three values are derived from the archive root R and the object identifier O: an encryption key K_e, an authentication key K_m and a 16-byte discovery tag d. Use HKDF-SHA256 [3], HMAC-SHA256 [4, 5], and AES-256-CTR [6, 7]. Labels are exact ASCII without NUL or newline. Keys and identifiers are raw bytes. These primitives are chosen so that recovery is reproducible with common tools: HKDF-Extract and HKDF-Expand reduce to HMAC-SHA256 calls, and a general-purpose cryptographic toolkit such as the OpenSSL command line provides SHA-256, HMAC and AES-256-CTR, so a holder of R can authenticate and decrypt records without the software that wrote them. The specification requires that such an implementation be possible from the byte layouts and the cited standard primitives alone [1, §6].

$$\begin{aligned} \mathrm{PRK} &= \text{HMAC-SHA256}(O, R), \\ K_e &= \text{HKDF-Expand}(\mathrm{PRK}, \texttt{``URMA/V0/private/content''}, 32), \\ K_m &= \text{HKDF-Expand}(\mathrm{PRK}, \texttt{``URMA/V0/private/authentication''}, 32), \\ d &= \text{HKDF-Expand}(\mathrm{PRK}, \texttt{``URMA/V0/private/discovery''}, 16). \end{aligned} \tag{3}$$

The first line is HKDF-Extract with O as salt/HMAC key and R as input key material. Each requested expansion fits one SHA256 output block. It takes the required prefix of

$$\text{HMAC-SHA256}(\mathrm{PRK}, \text{label} \parallel \texttt{01}).$$

A record is assembled from four parts. Let h_i be the 64-byte public header: the common prefix, the object identifier O, the discovery tag d, $\mathrm{LE}_{32}(i)$ and $\mathrm{LE}_{32}(N)$. The initial counter value V_i

(the IV) is fixed by the index, E_i is the encrypted body (the ciphertext), T_i is the MAC, and r_i is the complete record:

$$V_i = \mathrm{BE}_{64}(i) \parallel 0^8, \tag{4}$$

$$E_i = \text{AES-256-CTR}(K_e, V_i, B_i), \tag{5}$$

$$T_i = \text{HMAC-SHA256}(K_m, h_i \parallel V_i \parallel E_i), \tag{6}$$

$$r_i = h_i \parallel V_i \parallel E_i \parallel T_i. \tag{7}$$

CTR encrypts the initial 128-bit big-endian counter V_i and increments the entire counter by one per block. The 32 816-byte body consumes exactly 2 051 AES blocks, while adjacent index ranges begin 2^{64} blocks apart. There is no cipher padding; P_i is authenticated plaintext padding internal to the record.

The sizes add directly: $64 + 16 + 32\,816 + 32 = 32\,928$. Encryption and authentication have separate derived keys. The discovery tag is a filter, not an authenticator (Section 3.3). A reader verifies the full HMAC over the first 32 896 bytes, and requires the stored IV to be the canonical one, V_i recomputed from the header's index, *before decrypting*. CTR alone would not detect modified ciphertext or metadata.

Each (O, i) is encrypted once. Resuming a publication reuses the exact encrypted bytes. Reencrypting, including after restoring a snapshot, requires a fresh O. This requirement prevents reuse of an object's counter range for another plaintext; RNG failure is outside the construction's security guarantee.

3.3 Discovery and recovery without a final manifest

Chain discovery starts from the envelope, not from a list of object identifiers. A scanner looks in the input witnesses of the scanned range for the exact envelope form of Section 4; the marker bytes appearing anywhere else mean nothing [1, §8.4]. From each such reveal the reader extracts the record, requires the exact 32 928-byte length and a kind `01` prefix, and reads O from the public header. O is public, still untrusted, and serves only as the HKDF salt. Once per distinct O, the reader derives PRK and d from (O, R)—two HMAC evaluations—and compares d with the discovery tag in the header. Because d is keyed by R, a match selects the record as a candidate for this compartment; the match authenticates nothing by itself, and the full HMAC decides. A record whose tag differs is *unrelated* to R; it may belong to another compartment, be damaged or be a forgery. The rule is unconditional: a record whose tag differs is not accepted for the archive, even when its MAC verifies [1, §5.4].

The reveal that carries an envelope also carries a proof for it (Section 4), and the specification has a scanner validate that proof for every record it extracts [1, §8.4]. For a private record the envelope and its proof are transport. The key that signs is a transport key, not an author proof (Section 2), and a record supplied offline has no transaction at all. In both cases what authenticates the content is the MAC. Supplied records skip the envelope step and meet the same checks. Records are grouped by O, and each group is reconstructed independently. The procedure below is written for discovery over a stream of candidates; the strict opening of a supplied set is described after it:

```
RecoverPrivate(R, records sharing one object ID O):
  derive d, Ke, Km from PRK(O, R); create bounded staging and an empty index map
  for each record r:
    require exact layout, kind 01, zero flags, N > 0 and i < N, else INVALID
    require header discovery tag = d, else UNRELATED
    require the HMAC over bytes [0, 32896) and the canonical IV, else INVALID
    only now decrypt the body
    require L > 0, ceil(L / 32768) = N, L within the format bound, else INVALID
    require a supported type and zero metadata flags, else INVALID
    require digest, L, N and type equal to those already retained, else CONFLICT
    if index i is retained with identical record bytes: continue
    if index i is retained with different record bytes: CONFLICT
    retain r; stage its first ni chunk bytes at position i
  if no record was retained: NO DISCOVERABLE OBJECT
  take L, N and the digest from the retained records, which agree
  if any position in [0, N) is absent: INCOMPLETE
  require staged length L and SHA256 equal to that digest
  release the exact original bytes; otherwise reject the reconstruction
```

The outcomes differ in reach. `INVALID` and `UNRELATED` concern one record. An unrelated record is set aside. An invalid one is counted and discarded, whether it fails before decryption (layout, MAC, IV) or after it (length, count relation, type, flags); in neither case does it poison retained data of this or any other object. `CONFLICT` concerns the object. It arises only between records that each passed every check of their own: two different records for one index, or records that disagree on digest, length, count or type. It invalidates this object's reconstruction, and first-wins or last-wins selection is forbidden [1, §5.2, 5.4]. Staging does not expose a partial plaintext file as a successful recovery.

Take the two-chunk example of Section 3.1, $L = 32\,769$. It yields two records. Record 0 carries 32 768 original bytes and no padding; record 1 carries one original byte and 32 767 padding bytes. The records total 65 856 bytes. Receiving only record 1 reveals, after authentication, the total length, the digest and the missing index 0. Receiving record 1 twice still leaves index 0 missing. Once both arrive, discarding only the derived padding recovers 32 769 bytes and allows the final hash check. No end-of-publication message is needed.

The records bound what a reader can conclude. Any authentic record names its object's L, digest and N, so a missing position is recognizable; no record names which other objects exist, and nothing here proves a complete archive inventory. With zero authentic records, a wrong archive root and absence of matching data are indistinguishable. A provider can omit a whole object without leaving a detectable hole in another. A scan started after the first relevant reveal cannot see the earlier records. It may find the object incomplete or not find it at all, and the records it does find cannot say where the missing ones are. A reader does not report a scan as complete when its source does not establish that, and a failed source is never an empty successful scan [1, §9]. Staging, retry and response limits are declared client capacity; refusing an object for capacity is not a validity result.

The same checks serve three modes with different consequences. *Discovery* runs over a stream of candidates, from a scan or any other source: a bad record is set aside and the rest continue, as in the procedure above. A *strict opening* takes a supplied set of records as one object: any invalid or unrelated record fails the whole opening, identical duplicates are accepted but never fill absent positions, and nothing is delivered but the complete validated original [1, §5.4, 6]. The third mode reads the offline container described next.

An optional offline wrapper (kind `03`) is a container for such a set. It stores one object's records as an eight-byte prefix, a u32 record count of at least one, and repeated u32 record lengths followed by record bytes. For q supplied records its exact length is $12 + 32\,932q$. Truncation, trailing bytes

and mixed objects are rejected, and no mode searches for a marker inside it, resynchronizes or repairs its framing. The wrapper adds no MAC because authentication resides in each record.

An *explicit recovery* from such a container is the third mode, which a reader may offer [1, §6.1]. It is selected explicitly, never entered as a silent fallback of a strict opening, and it recovers one object, not a valid container. It tolerates bad records but not bad framing: the container's framing must be intact, and the procedure above then runs over the slots, setting aside what is unrelated, a class that here also covers a slot without the marker or with the well-formed prefix of a public kind, counting and discarding what is invalid, and letting no unauthenticated identifier select the object. The records retained must name exactly one O. A second authenticated identifier, an authenticated conflict, a framing fault, a failed source or a lack of capacity refuses the recovery, even after a complete prefix, because the whole input is examined before anything is delivered. An absent position leaves the object incomplete, and nothing partial is delivered; with no authenticated record, nothing is recovered and nothing follows about the root. A success is reported as a recovered object, with counts of what was discarded and what was set aside as diagnostics rather than proof of origin or cause; it does not establish that the container was valid.

4 Commit, reveal and the author proof

Each on-chain record occupies one reveal; an offline container cannot be published as a record. A commit creates a native P2TR output. A reveal spends that outpoint with transaction version 2, exactly one input, an empty scriptSig, and one native SegWit *return output* (P2WPKH or P2TR): an ordinary spendable output that takes the remaining value and has nothing to do with `OP_RETURN`. Fees, output value, sequence and locktime are fixed before signing and satisfy the selected chain's rules. They are not content metadata. One commit may fund multiple reveals, with one output for each.

The internal key is A, and the Taproot tree has exactly one tapscript leaf, version `c0`, with no Merkle path. For a public record A is the author key. For a private record the same symbol stands for that record's transport key (Section 2): the mechanism below is identical, but what it proves is transport authorization, not authorship of the private content [1, §8.1]. The script S, the record's *envelope*, has this exact form:

```
PUSH32(A) OP_CHECKSIG
OP_0 OP_IF
  PUSH4("URMA")
  PUSH(record segment 0) ... PUSH(last record segment)
OP_ENDIF
```

The false branch carries the record without executing it. Split the complete record into consecutive 520-byte segments, 520 bytes being the largest element a script may push [10], and a final nonempty remainder if necessary. Minimal pushes and canonical segmentation are required; an empty trailing segment or extra opcode is rejected. In particular, one-byte values `01`–`10` and `81` use the corresponding numeric opcodes. A one-byte `00` segment uses `01 00`; it is distinct from the structural `OP_0`. The normative specification supplies the complete push encoding rules [1, §8]. The envelope uses BIP341 script-path spending and BIP342 tapscript exactly as those specifications define them [9, 10]; URMA defines the record layout, segmentation and validation rules, not the script mechanism.

The commit output is what ties this script to the key A. BIP341 derives the Taproot output key Q from the internal key A and from the hash of the single leaf, which is a tagged hash over the leaf version and the exact script S [9]. Q therefore commits to A and to every byte of S, including the record inside it. The derivation uses BIP341's tagged hashes and rejection conditions exactly as that specification defines them, not ordinary untagged SHA256 operations. The commit script

is `OP_1 PUSH32(`$Q.x$`)`, where $Q.x$ is the 32-byte x coordinate of Q. The 33-byte control block in the reveal contains `c0` OR the parity of Q, followed by A. With it a verifier recomputes Q from the revealed script and compares the result with the key in the commit output being spent. A reveal whose key and script do not reproduce the key in the commit output is rejected, and a 33-byte control block has no room for the Merkle path that a second leaf would need.

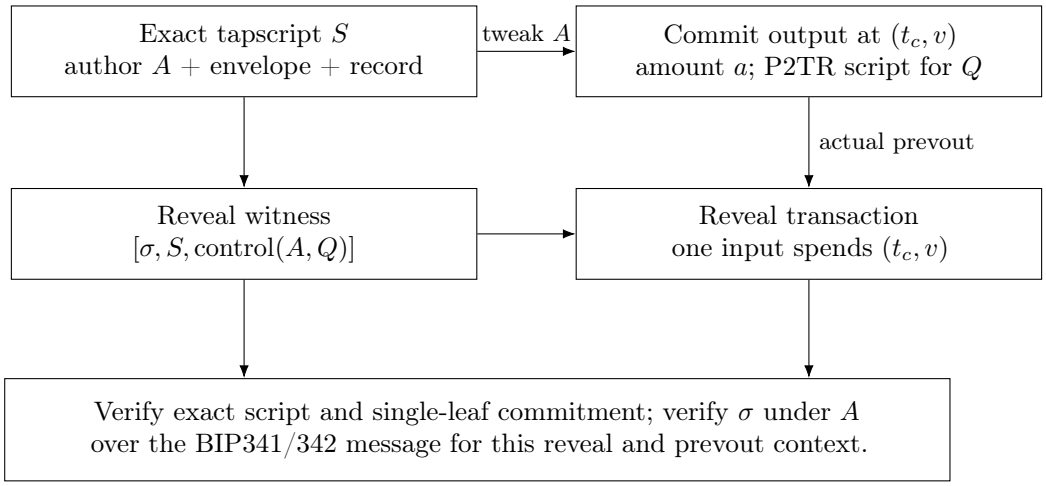


Figure 2: The parsed key becomes an authenticated public author only after checking both the commitment and the signature against the actual transaction context.

The witness is exactly $[\sigma, S, \text{control block}]$, where σ is a 64-byte BIP340 signature under A with the default sighash (`SIGHASH_DEFAULT`). There is no appended sighash byte, annex or extra witness item. The signed message is the BIP341 transaction message with the BIP342 script-path extension: `ext_flag=1`, `key_version=0`, the hash of this leaf, and `codesep_pos=0xffffffff` [8, 10]. These are the field names of those standards, kept so that the description can be matched against them. The message covers the reveal transaction, the amount and script of the output being spent and, through the leaf hash, the script and its record. The commitment alone requires no secret; the signature is made with the secret key that corresponds to A. It is therefore the authorization, under A, of this record in this transaction, which is why an implementation uses the complete standard message construction, not a signature over the record hash alone.

A verifier fetches the creating commit, recomputes its TXID, resolves the reveal's actual outpoint, and checks the amount and output script used in the signature message. It checks the reveal shape, exact script, leaf version, internal key, control block length and output commitment before accepting the signature. Copying A into another script, validating a funding signature or matching a TXID is insufficient. A public offline proof retains the reveal with its witness, and the commit with the output being spent (the prevout). This proves authorization by a key, not chain inclusion or a person's identity.

5 Atomic public records

Atomic records use the same envelope and proof under a stable author key. Their complete record cap is 32 768 bytes, including the prefix. The body layout follows directly from the kind [1, §7.1–7.1.3]:

Kind	Body after the eight-byte prefix	Complete bytes
`02`	Exact UTF-8 text	8–32 768
`04`	Target TXID (32 bytes), then exact UTF-8 text	40–32 768
`05`	Exact UTF-8 pseudonym, at most 128 bytes	8–136
`06`	An avatar: a 16×16 image at four bits per pixel, exactly 128 bytes	136
`0A`	Wire label, hashtags, then u16-length UTF-8 text	12–32 768
`0B`	Target TXID (32 bytes), then the `0A` fields	44–32 768
`0C`	Profile identifier (8 bytes), then opaque payload	16–32 768

Text is validated as UTF-8 but not normalized. No trimming, case folding, newline conversion or reserialization is allowed. Embedded U+0000 is data, not a terminator; empty UTF-8 is structurally valid. The binary avatar body does not undergo UTF-8 decoding. Display and identity-state resolution belong above record verification.

For example, the six text bytes `41 0d 0a c3 a9 00` form a 14-byte plain-post record. They encode an ASCII letter, CRLF, a two-byte UTF-8 character and U+0000. Replacing CRLF with LF or normalizing the character changes the authenticated bytes. A reply carrying the same text occupies 46 bytes because it adds the fixed 32-byte target. The plain text budget is 32 760 bytes, the reply budget 32 728 and the pseudonym budget 128.

A TXID field stores transaction-hash wire order: decode a displayed hexadecimal TXID and reverse the 32 bytes. Reverse again for display. Its chain context is external configuration. An unresolved reply target does not invalidate the reply's own bytes or author proof. Separate transactions containing identical text have distinct publication identities. No atomic record becomes multipart merely because its text exceeds the cap; multipart is a separate construction.

Kind `05` is the pseudonym, exact UTF-8 under a ceiling of its own [1, §7.1]. Its body holds at most 128 bytes, so a complete record is 8 to 136 bytes. The general rules apply unchanged: the bytes are validated as UTF-8 and never normalized, embedded U+0000 is data, and the empty pseudonym is valid. The ceiling counts encoded bytes, not Unicode scalar values, UTF-16 code units or grapheme clusters: 64 copies of U+03B1, two bytes each, fill the 128 bytes and fit, whereas 65 occupy 130 bytes and the record is invalid. A writer does not produce a longer body and a reader rejects one; neither truncates or normalizes it to fit. Nothing follows the pseudonym: no separator, terminator, timestamp, address or further field. The ceiling belongs to this kind alone; the text of kinds `02` and `04` and the payload of kind `0C` run to the record cap.

Kind `06` is the avatar, an image in a fixed format [1, §7.1.3]. Its body is exactly 128 bytes and holds a 16×16 image at four bits per pixel. Each pixel is an index into a fixed palette of sixteen colours, none transparent, given as eight-bit red, green and blue in the sRGB colour space [12]: `000000`, `0000AA`, `00AA00`, `00AAAA`, `AA0000`, `AA00AA`, `AA5500`, `AAAAAA`, `555555`, `5555FF`, `55FF55`, `55FFFF`, `FF5555`, `FF55FF`, `FFFF55` and `FFFFFF`, for indices `0` to `F` in that order. The palette is part of the format: a reader uses these values and accepts no other. The body contains nothing else: no dimensions, length, palette, alpha channel, padding, compression or trailing data. Pixels are stored row by row, rows from top to bottom and pixels from left to right, so each row occupies eight consecutive bytes. For $0 \le x, y < 16$, pixel (x, y) has position $i = 16y + x$, lies in body byte $\lfloor i/2 \rfloor$, at record offset $8 + 8y + \lfloor x/2 \rfloor$, and occupies the upper four bits of that byte, its high nibble, when x is even and the lower four bits, its low nibble, when x is odd. A top row whose pixels hold the palette indices `0` to `F` in order is therefore the eight bytes `01 23 45 67 89 AB CD EF` at offsets 8–15, and the last byte of the record, offset 135, holds pixels $(14, 15)$ and $(15, 15)$. Every nibble is a valid palette index, so every 128-byte body is a valid image. A body whose bytes resemble a file signature or the URMA marker is still 256 pixels; a reader neither rejects it for the resemblance nor interprets it as a file, and infers no codec, dimensions or metadata from the body. The format fixes the resolution and the palette of an image, not what the image depicts: symbols, drawings and encoded information remain representable, harmful ones included; a verified record establishes its bytes and the key that authorized them, not what they depict.

Kinds `0A` and `0B` are structured forms of `02` and `04` under the same cap, envelope and proof [1, §7.1.1]. They add two kinds of label so that readers can group records deterministically: a *wire*, which is a flat channel name, and *hashtags*, which name subjects. After the prefix, and for `0B` after the same 32-byte target, the body holds in order: a u8 wire length (0–32) and that many ASCII bytes, zero meaning no wire; a u8 hashtag count (0–8), each hashtag a u8 length (1–32) followed by its ASCII bytes; a u16 text length and exactly that many UTF-8 bytes, after which the record ends. Every label that is present matches the pattern `[a-z0-9][a-z0-9_-]{0,31}`; hashtags are strictly ascending in byte order, which also excludes duplicates; noncanonical bytes are rejected, not normalized. The minimum complete lengths are 12 and 44 bytes, and the u16 field never authorizes text beyond the record cap. For example, wire `urma`, one hashtag `v0` and the text `hey` form the 14-byte body `04 75 72 6d 61 01 02 76 30 03 00 68 65 79` and a 22-byte `0A` record. Labels are open: they prove no registration, ownership or affiliation. A `0B` target may be a `02`, `04`, `0A` or `0B` record on the selected network, and a reply's labels are its own, not inherited. A `/` or `#` inside `02`/`04` text remains literal text, and a reader that does not support these kinds rejects them as unsupported rather than misreading their text.

Kind `0C` is the atomic profile record [1, §7.1.2]. A *profile* is a separate specification, outside this protocol, that gives a payload its layout and meaning. The URMA specification counts semantic families and lists this kind as a fourth [1, §2]; this paper counts constructions, and as a construction it is an atomic public record under the same cap, envelope and proof. After the prefix it holds an eight-byte profile identifier and a payload that consumes the rest of the record, so complete lengths run from 16 bytes, with an empty payload, to 32 768 bytes, with a 32 752-byte payload. The identifier is authenticated but not interpreted: eight zero bytes mean an unspecified profile, and an unknown identifier does not invalidate an otherwise valid record. The profile field of a multipart root manifest follows the same three rules (Section 6). The payload is opaque. It undergoes no UTF-8 decoding, and bytes resembling another URMA record, manifest or archive trigger no parsing. Payload layout, meaning and validation belong to the profile's own specification. Conformance to a profile is a separate claim from URMA conformance, and the protocol never depends on a profile interpreter. For example, the prefix `55 52 4d 41 00 0c 00 00` followed by eight zero bytes is a complete 16-byte record with an unspecified profile and an empty payload. A reader that does not support kind `0C` rejects it as unsupported.

6 Public multipart objects

6.1 Inventory and bounded geometry

Public multipart objects preserve arbitrary bytes without a decryption secret. The payload is cut into *data parts* (kind `07`). *Leaf manifests* (kind `08`) list the parts in order, and one *root manifest* (kind `09`) lists the leaf manifests. Below they are called parts, leaves and the root; these names have nothing to do with the archive root R of private objects or with the tapscript leaf of Section 4. The root also states the payload length, the digest of the complete payload, the part count, the leaf count and an opaque eight-byte application profile identifier. An unknown profile remains valid generic data; eight zero bytes mean unspecified application. The root's reveal TXID is the object's locator, and the root does not contain its own TXID.

In this section L is the length of the payload in bytes, N the number of parts and M the number of leaves. L and N play the roles they played for private objects but follow this construction's rules; in particular L may be zero. The index i and the length n_i are reused in the same way, for a part. Only this construction uses a 256 KiB complete-record ceiling: a part's record is at most 262 144 bytes, of which 12 are prefix and index, so a part carries at most C_m payload bytes. A

leaf lists at most 511 parts and the root at most 511 leaves; the specification names these two fanout limits separately, and both are 511 [1, §7.2]. Then

$$C_m = 262\,144 - 12 = 262\,132, \qquad N = \max(1, \lceil L/C_m \rceil), \qquad M = \lceil N/511 \rceil. \tag{8}$$

For a small example, $L = 262\,133$ uses two data parts, one leaf and one root. By the layouts given below, the data records are 262 144 and 13 bytes, the leaf is 144 bytes and the root 128. Their sum is 262 429 bytes before any scripts or transactions. An empty object still uses the same three levels: 12-byte data record, 80-byte leaf and 128-byte root, totaling 220 bytes.

The object ceiling is a separate, explicit rule: $0 \le L \le 8\,553\,279\,476$ bytes, including any framing that an application profile adds inside the payload. It yields $N \le 32\,630$ and $M \le 64$. The rule does not follow from the tree: a root with 511 leaves of 511 parts each could index $511 \cdot 511 \cdot 262\,132 = 68\,448\,169\,972$ bytes, about eight times more, and the specification states that this structural capacity authorizes nothing beyond the ceiling [1, §7.2].

The specification states the ceiling as a rule and does not derive it. Arithmetically, the fanout and the ceiling both match a record of 32 768 bytes, the cap of an atomic record. 511 is the largest fanout whose full root ($64 + 64 \cdot 511 = 32\,768$) and full leaf ($16 + 64 \cdot 511 = 32\,720$) fit in such a record. The ceiling equals $511 \cdot 511 \cdot 32\,756$, which is what the same tree would hold if a part's record were also limited to 32 768 bytes and so carried 32 756 payload bytes. A part's record may in fact reach 256 KiB, while the fanout and the ceiling stay as they are [1, §7.2, 7.5]. Manifests are not padded up to the larger record, and the smaller geometry is not an alternative encoding: a reader does not accept 32 756 bytes as the capacity of a full part, nor a root whose counts assume it.

Part i (kind `07`) is the common prefix, a u32 global index, and the consecutive original payload slice. Its complete length is $12 + n_i$, with $n_i = \min(C_m, L - iC_m)$ for a nonempty object. The empty object has exactly one zero-payload part at index 0. A part examined on its own, without its root, is already rejected if $i \ge 32\,630$, an index that no object under the ceiling contains; its exact payload length is checked once the root is known.

A manifest refers to a child by an *edge* of exactly 64 bytes: the 32-byte TXID of the child's reveal followed by the 32-byte SHA-256 digest of the complete child record,

$$\text{child reveal TXID} \parallel \text{SHA256(complete child record)}. \tag{9}$$

Neither value is truncated. The TXID locates a transaction. The hash binds the exact record, including its common prefix, independently of TXID's exclusion of witness bytes. The TXID is stored in the wire order of Section 5; the digest bytes are stored directly, without display reversal. Hashing only the payload, script or transaction would check a different value.

Leaf j (kind `08`) contains its u16 index, u16 entry count k, u32 first global part index, and the ordered data references. Its prefix and fields occupy 16 bytes. For leaf j of an object,

$$\text{first} = 511j, \quad k = \min(511, N - 511j), \quad |\text{leaf}_j| = 16 + 64k. \tag{10}$$

On its own a leaf must already satisfy $j < 511$, the root's fanout, $1 \le k \le 511$, first $= 511j$ and the exact length $16{+}64k$; once the root is known it must also satisfy $j < M$ and $k = \min(511, N{-}511j)$, and $M \le 64$ under the ceiling [1, §7.5]. The maximum leaf length is 32 720 bytes. A root (kind `09`) has 64 fixed bytes followed by M ordered leaf references, hence length $64 + 64M$, at most 4 160 bytes under the object ceiling. The fixed bytes are, in order, the common prefix, L as u64, the 32-byte digest of the complete payload, N as u32, M as u16, two reserved zero bytes and the eight-byte profile identifier [1, §7.6]. Reserved fields are zero and neither manifest has padding or trailing bytes. Root entry j names leaf j; leaf entry r names part $511j + r$.

The boundary example in Figure 3 makes the leaf's role explicit without enumerating hundreds of transactions. Part indices 0–510 fill leaf 0; part 511 requires leaf 1. The leaves are 32 720

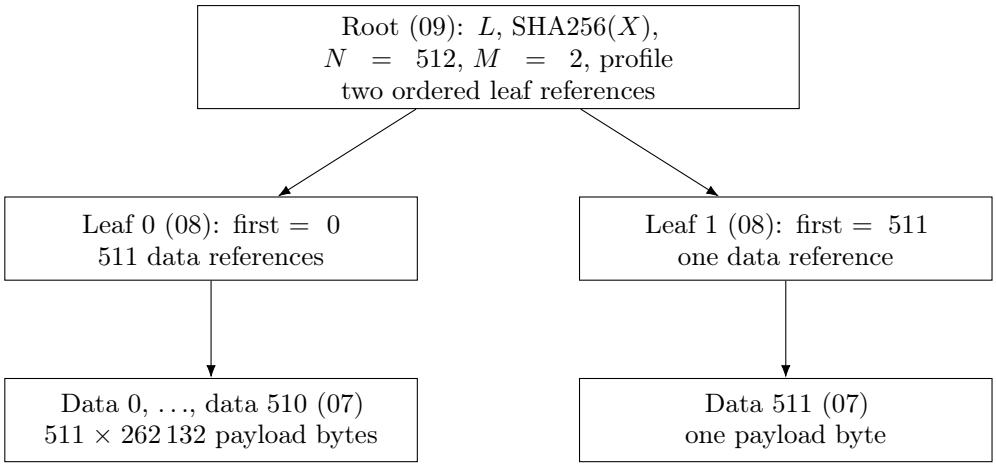


Figure 3: Crossing one leaf's capacity: $L = 511C_m + 1 = 133\,949\,453$ bytes. Leaves carry ordered inventory, not payload or repair data. The final leaf locates the one remaining part.

and 80 bytes and the root is 192 bytes. The root stays compact by locating two authenticated inventories instead of listing 512 data references directly. Even a one-part object uses a leaf; no root-to-data shortcut is accepted.

At the ceiling, the last part has 174 448 payload bytes, and the graph has $32\,630+64+1 = 32\,695$ records. These are arithmetic consequences, not maximum-capacity measurements or evidence that a network will relay the corresponding transactions (Table 1).

6.2 Construction and verification

A writer first builds and signs the data reveals; their TXIDs and complete-record hashes are then fixed, so the leaves can be built and signed, and finally the root. All reveals use the root's author A, and a reader requires every leaf and part to verify under that same author; a graph that violates this is not recovered [1, §7.7]. This dependency order fixes the reference bytes before any broadcast; submission order and confirmation policy remain separate. Computing the whole graph before anything confirms presumes that no TXID in it can change afterwards. One way to ensure this is for every unconfirmed transaction in the chain of dependencies to spend only SegWit inputs, with signatures that commit to all of its inputs and outputs: a TXID excludes witness data, so nothing a third party can alter then enters it [11]. Another is to wait for a commit to confirm before signing what depends on it. Either is a matter of the writer's method, not of what a reader accepts, and the specification leaves commit funding to the client [1, §8.1]. URMA provides no all-or-nothing chain transaction spanning the graph.

Recovery is a bounded two-level traversal. Define `FetchVerified(txid, hash?, A?)` to obtain a reveal together with the commit transaction it spends, recompute the requested TXID, verify the proof of Section 4 and the author where supplied, and only then compare the expected complete-record hash. A wrong TXID, or a proof that does not verify, makes what the source returned an invalid candidate; the specification directs the reader to discard it and try another source, and such a candidate does not poison authentic data already retained. The hash is judged only after the proof: once the proof at the requested TXID verifies, the record is fixed by the commitment, and a record hash that differs from the signed reference is a fault of the object, with no other source to try [1, §7.7]. Retries and raw response sizes are bounded. An unavailable source is not evidence that the object is empty.

Three named results are kept apart. An *invalid candidate* is a rejection at the source boundary: what a source returned has the wrong TXID or a proof that does not verify. *Invalid object structure* is a fault of the object as signed, not of the source: a verified manifest with impossible geometry, a duplicate reference or an impossible edge kind, or a child with a verified proof whose record hash differs from the manifest's reference; no other source is tried for that object. A record that matches its reference is also checked for the index, position and length that the manifest and the geometry assign it, and a completed traversal's payload for the length and digest stated in the root; when either check fails, the reader refuses the result and releases nothing [1, §7.8]. *Incomplete* recovery means that a required leaf or part is missing. These distinctions prevent a malicious provider from invalidating authentic retained data merely by returning a bad answer. Bounded inspection of parts that are already authenticated can be labelled partial diagnostics, never complete recovery or whole-profile acceptance [1, §7.7–7.8]. Figure 4 gives the complete procedure.

```
RecoverPublic(chain, root_txid):
  root = FetchVerified(root_txid)
  require kind 09, zero reserved fields, exact root length
  check L bound; derive N and M; require stored counts equal them
  A = root's verified author; seen = {root_txid}
  create bounded staging; initialize whole-payload SHA256
  for j = 0 .. M-1:
    require root.entry[j].txid not in seen, then add it
    leaf = FetchVerified(root.entry[j].txid, its hash, A)
    require kind 08, index j, first = 511*j
    require k = min(511, N-first), exact length 16+64*k
    for r = 0 .. k-1:
      require leaf.entry[r].txid not in seen, then add it
      part = FetchVerified(leaf.entry[r].txid, its hash, A)
      require kind 07 and index i = first+r
      require payload length min(262132, L-i*262132), or 0 if L=0
      append exact payload to staging; update hash and byte count
  require byte count = L and SHA256 = root's payload digest
  only now release the complete generic payload
```

Figure 4: Recovery of a public multipart object from the reveal TXID of its root manifest.

Every parse also enforces the common prefix, allowed flags, exact layouts and checked arithmetic. The set `seen` spans the whole graph. Because every referenced TXID must be new to it, a leaf cannot appear twice, a part cannot appear under two leaves, and no reference can point back at the root manifest or at an ancestor. Fetching one reference again from another source is a retry, not a second reference. The kind required at each edge prevents leaf-to-leaf or root-to-data traversal, and payload bytes never trigger recursive parsing.

Two things remain allowed. Identical payload bytes may occur at different indices; their records differ by index, and so do their reveal TXIDs. And the duplicate rule is per root graph: another root manifest by the same author may reference the same otherwise valid records [1, §7.7].

The contrast with private recovery is structural. Private records repeat the object's digest, length and count and carry authenticated indices under keys derived from the archive root R, so each record's membership and position are authenticated by its own MAC. Public data parts carry only an index and payload and share no secret; their binding to an object is the record hash in a manifest signed by the root manifest's author (Section 4). The root manifest and the leaves therefore supply the inventory, and removing it removes information required for public multipart recovery. Under this structure the root manifest's reveal TXID is the object's locator:

recovery fetches named reveals directly on the selected chain, without a start height when the source returns a transaction by its TXID, and the same verified records may be referenced from more than one root manifest of the same author [1, §7.6–7.8].

7 What verification establishes

Content authentication, completeness and chain placement answer separate questions. A successful private MAC authenticates bytes under a key derived from the archive root; any holder of that root can produce such records. A successful public proof establishes authorization by A. Neither result establishes the truth of content, civil identity, uniqueness of a pseudonym or a real-world capture time.

Claim	Required basis
Accepted, authentic record	Private: matching discovery tag, then valid MAC and metadata checks; public: exact record and Taproot author proof.
Complete object	Every required position; exact final length and complete payload digest.
Chain inclusion	Evidence connecting the relevant transaction and witness to the selected chain, under a declared validation model.
Confirmations	A current chain observation, including its source and tip; it can change after a reorganization.
Profile acceptance	Complete generic recovery, or a verified atomic profile record, plus the separate profile's interpretation and checks.

TXIDs exclude witness data. Accordingly, a Bitcoin transaction Merkle proof alone does not authenticate the retained witness; raw-block verification includes the witness commitment [11]. The point is concrete: the holder of the secret key that corresponds to A can also spend a commit output by Taproot's key path, and with the same non-witness fields that transaction has the same TXID as the reveal and carries no record [9]. Full consensus and chain selection may be supplied by a validating node. Reading transaction JSON from an HTTPS provider does not itself validate consensus, inclusion, unspentness or complete history. A check that a validating node or another party performed, and that the reader relies on, is a delegated check, not an independent verification by the reader itself; a property that nothing establishes or refutes remains not established: the reader's result neither asserts nor denies it [1, §8.4, 9].

A reorganization changes placement observations without corrupting retained authentic bytes. Offline content verification can succeed without establishing current inclusion. A source failure must not be reported as a successful empty scan, and a provider's confirmation count must not silently become a locally proven claim. Network identity is checked before retrieval; fallback sources stay within the selected chain context.

8 Security, availability and cost

This section states what each construction protects, under which conditions, and what it leaves exposed.

8.1 Private records

Goal. A party without the archive root R should learn nothing from a record about the original bytes or the encrypted metadata beyond what its public header discloses, and should be unable to forge a record, that is, to produce a new record, or a modified version of an existing one, that a holder of R accepts as authentic. Presenting an existing record again, unchanged, is not forgery: identical authenticated duplicates are accepted idempotently by design, which lets several sources

supply the same object [1, §5.4]. What an observer learns from several records together, or from their transactions, is listed below under what stays visible.

Mechanism. The private record is the generic Encrypt-then-MAC composition: CTR encryption under K_e, then HMAC under the independent K_m over header, IV and ciphertext.

Published results. Bellare and Namprempre show that this composition of an IND-CPA encryption scheme with a strongly unforgeable MAC provides chosen-ciphertext privacy and ciphertext integrity, and Krawczyk's analysis of the composition orders reaches the same recommendation [13, 14]. CTR mode is IND-CPA when AES-256 is a pseudorandom permutation and no counter block is reused under a key [15]. HMAC-SHA256 with a secret key is a pseudorandom function under assumptions on the SHA-256 compression function [16]. K_e, K_m and d come from HKDF, with O as a public salt and R as the secret input [17]. Its expansion step is HMAC keyed by the secret PRK. Its extraction step has a different form: the HMAC key is the public O and the secret R is the message. That step therefore depends on the separate analysis of HKDF's extraction [17], not on the result for a secret HMAC key.

Conditions. The construction meets the conditions of these results only if all three of the following hold.

1. K_e and K_m are distinct HKDF outputs, and neither is reused for another purpose.
2. Every (O, i) counter range is used once. The fresh OS-random O, the canonical IV and the 2^{64}-block spacing provide this; reencryption under an old O would break it.
3. T_i is verified over the complete header, IV and ciphertext before any decryption, so that the object identifier, discovery tag, index, count and IV are bound and no unauthenticated byte reaches the decryptor.

Limits. These are the published results for the generic scheme. They are not a formal proof of this exact byte layout or an audit of any implementation, and they give no protection against RNG failure, side channels or a compromised endpoint. Two points are not examined here. Deriving the keys of many objects from one archive root runs HKDF's extraction with many public salts under one secret. A reader also runs it with an O taken from a header that nothing has authenticated yet, while RFC 5869 asks that salt values not be chosen by an attacker [3]. This paper does not check that the cited analysis covers either use.

What stays visible. Private encryption hides original bytes and encrypted metadata, not marker use, object grouping, real chunk count, publication timing or funding links. An object's records are grouped in the clear by their shared identifier O; the discovery tag d repeats with it and adds nothing to that link. The header has no field that names the compartment: O is drawn per object and d is derived per object. Observing N exposes $(N-1)C_p < L \leq NC_p$. There are no dummy chunks or size classes. On a chain the transactions show what the records do not: the key that signs each reveal, the funding of each commit and the time of publication. Reveals signed with one key are linked by it (Section 2).

Holders of the archive root. Compromise of the archive root exposes that compartment's objects; V0 defines no recipient wrapping, revocation or root rotation. A holder can also write. Any holder of R can produce records that authenticate (Section 7), and one added authentic record for an existing (O, i) that differs from the first makes that object a conflict for every reader whose record set contains both. A writer can do the same to itself by reencrypting under an old O: a record whose body differs, if only in its random padding, conflicts with the earlier one (Section 3.2). The original records are not destroyed, and a record set that leaves the added record out still reconstructs. A recipient may retain or redistribute plaintext. The originating device and OS see plaintext before encryption, and this construction does not protect it from a compromised endpoint.

Endpoint and tooling exposure. Practical readers also need careful secret and staging-file handling. Passing keys in process arguments, keeping temporary plaintext or relying on a tool without suitable constant-time behavior adds implementation exposure; the use of standard primitives does not remove it.

8.2 Public records

The author proof of Section 4 uses BIP340 signatures and the BIP341 commitment as those specifications define them [8, 9, 10]; this paper relies on them and adds no security analysis of its own. A successful proof establishes authorization by a key. It does not establish chain inclusion, the identity of a person, the uniqueness of a pseudonym, the truth of the content or its time of capture. Public signatures identify a pseudonymous key and reveal linkable publication activity, not anonymity.

8.3 Availability, cost and capacity

Availability. Recovery requires that bytes leave the device before its loss and remain accessible elsewhere. Private recovery additionally requires the surviving archive root. A chain, archive or offline copy can supply records, but the record format cannot compel retention or connectivity. Every record of an object is needed: if the bytes of one private record, or of one multipart part, leaf or root manifest, can be obtained neither from a copy nor by reconstruction, that object cannot be completed. V0 specifies no normative erasure-code recovery step. Optional reconstructed candidates still have to pass ordinary record and object validation. A URL preserved in public text preserves the URL bytes, not the external resource.

Format bounds. Each construction has a bound on the size of one object (Table 1). The bounds follow from field widths and fixed rules. They do not show by themselves that objects of those sizes have been published or recovered.

Construction	Format bound on one object	Where the bound comes from	Reveals at the bound
Private object	140 737 488 322 560 bytes, about 128 TiB	$(2^{32} - 1) \times 32\,768$, from the 32-bit chunk count; the format sets no policy ceiling [1, §5.1]	4 294 967 295
Atomic public record	32 768 bytes	Fixed cap on one complete record [1, §7.1]	1
Public multipart object	8 553 279 476 bytes, about 7.97 GiB	Explicit ceiling, below what the 511-by-511 tree could hold [1, §7.2]	32 695

Table 1: Format bounds by construction. A bound states what the format can represent or permits. It is not by itself evidence that an object of that size has been published or recovered. The last column counts the reveals an object at the bound would need if it were published on a chain; private records kept offline need none.

What limits an object in practice. The practical limits are the number of reveal transactions an object needs, their fees, relay and mining policy, and the capacity a client declares. Every record published on a chain needs its own reveal. A private object of L bytes therefore needs $\lceil L/32\,768 \rceil$ reveals, which is 32 for an object of exactly one mebibyte. Its commit transactions come in addition, and private records kept offline need no transaction at all.

Cost of a record. Publication consumes transactions and fees. Record bytes, serialized transaction bytes and transaction weight are different quantities. Under the BIP141 accounting model, weight is $3 \times$ base size + total size [11]; script and witness framing still count. As a worked case, take one private record. Its 32 928 bytes split as $63 \times 520 + 168$. Each 520-byte segment needs a three-byte push prefix and the last segment two, and with the key check, the envelope marker and the branch opcodes the script is 33 161 bytes. Signature, script and control block make a 33 264-byte witness. With a P2WPKH return output the non-witness part is 82 bytes, so the reveal totals 33 348 bytes, counting the two-byte SegWit marker and flag, and weighs $3 \times 82 + 33\,348 = 33\,594$ weight units: about 8.4 kvB for one 32 KiB chunk. Under the BIP141 limit of 4 000 000 weight units per block [11], at most 119 such reveals fit in one block. That is an arithmetic ceiling on reveals alone, before any commit transaction, and not an observed rate. Commit batching can change costs without changing record validity. Relay policy, mining policy and selected-network rules may constrain publication more tightly than the record format. A 256 KiB record ceiling does not assert universal relay acceptance, a fixed fee or resistance to censorship.

Recovery resources. Recovery also consumes bandwidth, CPU and storage. A reader bounds source responses, retry work, counters and staging before trusting advertised sizes. It can process one record at a time, incrementally hash output and keep its index inventory in bounded storage. Private staging remains unreleased until the object validates; multipart traversal need not load a complete object into memory. Declared local capacity may be smaller than a protocol maximum. Refusing work for capacity reasons is neither an invalid-object judgment nor evidence of missing bytes.

9 Conclusion

URMA makes recovery depend on explicit records and verification rules. Private objects are discovered by a tag keyed with the archive root and repeat enough authenticated metadata to recover without a final manifest; public multipart objects expose a bounded inventory of signed, hash-bound records. Atomic public records preserve a single author's exact bytes in one reveal. Across these constructions, the receiver checks the data independently of the publisher while reporting availability, completeness and chain observations separately. The result is a construction for recoverable authenticated bytes under stated dependencies, not a promise that every published object will remain available.

References

[1] *URMA V0—Normative protocol specification.* Document revision 0.11, 10 October 2026; version byte `00`. SHA-256 of the file: `a6ef4162 4d180bc5 eb459839 02d6dbf0 b8e1309d dfd695e5 f1242f21 b1ca0bcb`.

[2] S. Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System.* 2008. `https://bitcoin.org/bitcoin.pdf`.

[3] H. Krawczyk and P. Eronen. *HMAC-based Extract-and-Expand Key Derivation Function (HKDF).* RFC 5869, 2010. `https://www.rfc-editor.org/rfc/rfc5869`.

[4] H. Krawczyk, M. Bellare and R. Canetti. *HMAC: Keyed-Hashing for Message Authentication.* RFC 2104, 1997. `https://www.rfc-editor.org/rfc/rfc2104`.

[5] NIST. *Secure Hash Standard (SHS).* FIPS 180-4, 2015. `https://csrc.nist.gov/pubs/fips/180-4/upd1/final`.

[6] NIST. *Advanced Encryption Standard (AES)*. FIPS 197, updated 2023. https://csrc.nist.gov/pubs/fips/197/final.

[7] M. Dworkin. *Recommendation for Block Cipher Modes of Operation: Methods and Techniques*. NIST SP 800-38A, 2001. https://csrc.nist.gov/pubs/sp/800/38/a/final.

[8] P. Wuille, J. Nick and T. Ruffing. *Schnorr Signatures for secp256k1*. BIP340. https://bips.dev/340/.

[9] P. Wuille, J. Nick and A. Towns. *Taproot: SegWit version 1 spending rules*. BIP341. https://bips.dev/341/.

[10] P. Wuille, J. Nick and A. Towns. *Validation of Taproot Scripts*. BIP342. https://bips.dev/342/.

[11] E. Lombrozo, J. Lau and P. Wuille. *Segregated Witness (Consensus layer)*. BIP141. https://bips.dev/141/.

[12] IEC. *Multimedia systems and equipment—Colour measurement and management—Part 2-1: Colour management—Default RGB colour space—sRGB*. IEC 61966-2-1:1999. https://webstore.iec.ch/en/publication/6169.

[13] M. Bellare and C. Namprempre. *Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm*. ASIACRYPT 2000; Cryptology ePrint Archive, Paper 2000/025. https://eprint.iacr.org/2000/025.

[14] H. Krawczyk. *The Order of Encryption and Authentication for Protecting Communications (or: How Secure Is SSL?)*. CRYPTO 2001; Cryptology ePrint Archive, Paper 2001/045. https://eprint.iacr.org/2001/045.

[15] M. Bellare, A. Desai, E. Jokipii and P. Rogaway. *A Concrete Security Treatment of Symmetric Encryption*. FOCS 1997. https://web.cs.ucdavis.edu/~rogaway/papers/sym-enc.pdf.

[16] M. Bellare. *New Proofs for NMAC and HMAC: Security without Collision-Resistance*. CRYPTO 2006; Cryptology ePrint Archive, Paper 2006/043. https://eprint.iacr.org/2006/043.

[17] H. Krawczyk. *Cryptographic Extraction and Key Derivation: The HKDF Scheme*. CRYPTO 2010; Cryptology ePrint Archive, Paper 2010/264. https://eprint.iacr.org/2010/264.