Network Working Group
Internet-Draft
Intended status: Informational
Expires: 21 June 2026

F. Denis
Independent Contributor

18 December 2025

XET: Content-Addressable Storage Protocol for Efficient Data Transfer
draft-denis-xet-03

Abstract

This document specifies XET, a content-addressable storage (CAS) protocol designed for efficient storage and transfer of large files with chunk-level deduplication.

XET uses content-defined chunking to split files into variable-sized chunks, aggregates chunks into containers called xorbs, and enables deduplication across files and repositories through cryptographic hashing.

Discussion Venues

This note is to be removed before publishing as an RFC.

Source for this draft and an issue tracker can be found at https://github.com/jedisct1/draft-denis-xet.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 21 June 2026.

Copyright Notice

Table of Contents

1. Introduction 5
1.1. Use Cases 6
2. Terminology 6
2.1. Notational Conventions 8
2.1.1. Pseudo-Code Conventions 8
3. Protocol Overview 8
3.1. Upload Flow 9
3.2. Download Flow 9
4. Algorithm Suites 9
4.1. Suite Definition 10
4.2. Suite Requirements 10
4.3. Suite Negotiation 10
4.4. Defined Suites 10
5. Content-Defined Chunking 11
5.1. Gearhash Algorithm 11
5.2. Algorithm Parameters 11
5.3. Algorithm Description 11
5.4. Boundary Rules 12
5.5. Determinism Requirements 13
5.6. Performance Optimization 13
6. Hashing Methods 13
6.1. Chunk Hashes 13
6.2. Xorb Hashes 14
6.2.1. Internal Node Hash Function 14
6.2.2. Merkle Tree Construction 15
6.2.3. Xorb Hash Computation 17
6.3. File Hashes 17
6.4. Term Verification Hashes 18
6.5. Hash String Representation 19
6.5.1. Conversion Procedure 19
6.5.2. Example 20
7. Xorb Format 20
7.1. Size Constraints 20
7.2. Binary Format 20
7.3. Chunk Header Format 21
7.3.1. Version Field 21
7.3.2. Size Fields 21
7.3.3. Compression Type 22
7.4. Compression Schemes 22
7.4.1. None (Type 0) 22

7.4.2. LZ4 (Type 1) 22
7.4.3. ByteGrouping4LZ4 (Type 2) 22
7.4.4. Compression Selection 23
7.5. CasObjectInfo Footer 24
7.5.1. Main Header 24
7.5.2. Hash Section 24
7.5.3. Boundary Section 24
7.5.4. Trailer 25
8. File Reconstruction 25
8.1. Term Structure 25
8.2. Reconstruction Rules 25
8.3. Range Queries 26
9. Shard Format 26
9.1. Overall Structure 26
9.2. Data Types 26
9.3. Header 27
9.3.1. Magic Tag 27
9.4. File Info Section 28
9.4.1. File Block Structure 28
9.4.2. FileDataSequenceHeader 28
9.4.3. FileDataSequenceEntry 28
9.4.4. FileVerificationEntry 29
9.4.5. FileMetadataExt 29
9.4.6. Bookend Entry 29
9.5. CAS Info Section 29
9.5.1. CAS Block Structure 30
9.5.2. CASChunkSequenceHeader 30
9.5.3. CASChunkSequenceEntry 30
9.5.4. Bookend Entry 31
9.6. Footer 31
9.6.1. Lookup Tables 32
9.6.2. Chunk Hash Key Usage 33
10. Deduplication 33
10.1. Local Session Deduplication 33
10.2. Cached Metadata Deduplication 33
10.3. Global Deduplication 34
10.3.1. Eligibility Criteria 34
10.3.2. Query Process 34
10.3.3. Keyed Hash Security 34
10.4. Fragmentation Prevention 34
11. Upload Protocol 35
11.1. Step 1: Chunking 35
11.2. Step 2: Deduplication 35
11.3. Step 3: Xorb Formation 35
11.4. Step 4: Xorb Serialization and Upload 36
11.5. Step 5: Shard Formation 36
11.6. Step 6: Shard Upload 36
11.7. Ordering and Concurrency 36

12. Download Protocol 37
12.1. Step 1: Query Reconstruction 37
12.2. Step 2: Parse Response 37
12.3. Step 3: Download Xorb Data 37
12.4. Step 4: Extract Chunks 37
12.5. Step 5: Assemble File 38
12.6. Caching Recommendations 38
12.7. Error Handling 38
13. Caching Considerations 38
13.1. Content Immutability 38
13.2. Client-Side Chunk Caching 39
13.2.1. Cache Key Design 39
13.2.2. Cache Granularity 39
13.2.3. Eviction Strategies 39
13.3. Xorb Data Caching 40
13.3.1. Client-Side Xorb Cache 40
13.3.2. Byte Range Considerations 40
13.4. Shard Metadata Caching 40
13.4.1. Cache Lifetime 40
13.4.2. Cache Size 41
13.5. Pre-Signed URL Handling 41
13.6. HTTP Caching Headers 41
13.6.1. Server Recommendations 41
13.6.2. Client Recommendations 42
13.7. CDN Integration 42
13.7.1. CDN Cache Keys 42
13.7.2. Range Request Caching 43
13.8. Proxy and Intermediary Considerations 43
14. Security Considerations 44
14.1. Content Integrity 44
14.2. Authentication and Authorization 44
14.3. Global Deduplication Privacy 44
14.4. Access-Controlled Content 44
14.4.1. Repository-Level Access Control 45
14.4.2. CDN Considerations for Gated Content 45
14.4.3. Cross-Repository Deduplication 45
14.4.4. Privacy Implications 45
14.5. Denial of Service Considerations 46
IANA Considerations 46
References 46
Normative References 46
Informative References 46
Appendix A. Recommended HTTP API 47
A.1. Authentication 47
A.2. Common Headers 47
A.3. Get File Reconstruction 48
A.4. Query Chunk Deduplication 49
A.5. Upload Xorb 50

A.6. Upload Shard 50
Appendix B. Gearhash Lookup Table 51
Appendix C. Test Vectors 53
C.1. Chunk Hash Test Vector 53
C.2. Hash String Conversion Test Vector 53
C.3. Internal Node Hash Test Vector 53
C.4. Verification Range Hash Test Vector 54
C.5. Reference Files 54
Acknowledgments 54
Author's Address 54

1. Introduction

Large-scale data storage and transfer systems face fundamental challenges in efficiency: storing multiple versions of similar files wastes storage space, and transferring unchanged data wastes bandwidth. Traditional approaches such as file-level deduplication miss opportunities to share common content between different files, while fixed-size chunking fails to handle insertions and deletions gracefully.

XET addresses these challenges through a content-addressable storage protocol that operates at the chunk level. By using content-defined chunking with a rolling hash algorithm, XET creates stable chunk boundaries that remain consistent even when files are modified.

This enables efficient deduplication not only within a single file across versions, but also across entirely different files that happen to share common content.

The protocol is designed around several key principles:

* Determinism: Given the same input data, any conforming implementation MUST produce identical chunks, hashes, and serialized formats, ensuring interoperability.

* Content Addressing: All objects (chunks, xorbs, files) are identified by cryptographic hashes of their content, enabling integrity verification and natural deduplication.

* Efficient Transfer: The reconstruction-based download model allows clients to fetch only the data they need, supporting range queries and parallel downloads.

* Algorithm Agility: The chunking and hashing algorithms are encapsulated in algorithm suites, enabling future evolution while maintaining compatibility within a deployment.

* Provider Agnostic: While originally developed for machine learning model and dataset storage, XET is a generic protocol applicable to any large file storage scenario.

This specification provides the complete details necessary for implementing interoperable XET clients and servers. It defines the XET-BLAKE3-GEARHASH-LZ4 algorithm suite as the default, using BLAKE3 for cryptographic hashing, Gearhash for content-defined chunking, and LZ4 for compression.

1.1. Use Cases

XET is particularly well-suited for scenarios involving:

* Machine Learning: Model checkpoints often share common layers and parameters across versions, enabling significant storage savings through deduplication.

* Dataset Management: Large datasets with incremental updates benefit from chunk-level deduplication, where only changed portions need to be transferred.

* Container Images: OCI container images share common base layers across different applications and versions. Content-defined chunking enables deduplication not only across image layers but also across similar content in unrelated images.

* Version Control: Similar to Git LFS but with content-aware chunking that enables sharing across different files, not just versions of the same file.

* Content Distribution: The reconstruction-based model enables efficient range queries and partial downloads of large files.

2. Terminology

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “NOT RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

Throughout this document, the following terms apply:

Term	Definition
Algorithm Suite	A specification of the cryptographic hash function and content-defined chunking algorithm used by an XET deployment. All participants in an XET system MUST use the same algorithm suite for interoperability.
Chunk	A variable-sized unit of data derived from a file using content-defined chunking. Chunks are the fundamental unit of deduplication in XET.
Chunk Hash	A 32-byte cryptographic hash that uniquely identifies a chunk based on its content.
Xorb	A container object that aggregates multiple compressed chunks for efficient storage and transfer. The name derives from “XET orb.”
Xorb Hash	A 32-byte cryptographic hash computed from the chunk hashes within a xorb using a Merkle tree construction.
File Hash	A 32-byte cryptographic hash that uniquely identifies a file based on its chunk composition.
Shard	A binary metadata structure that describes file reconstructions and xorb contents, used for registering uploads and enabling deduplication.
Term	A reference to a contiguous range of chunks within a specific xorb, used to describe how to reconstruct a file.
File Reconstruction	An ordered list of terms that describes how to reassemble a file from chunks stored in xorbs.
Content-Defined Chunking (CDC)	An algorithm that determines chunk boundaries based on file content rather than fixed offsets, enabling stable boundaries across file modifications.

Content-Addressable Storage (CAS)	A storage system where objects are addressed by cryptographic hashes of their content rather than by location or name.
Global Deduplication	The process of identifying chunks that already exist in the storage system to avoid redundant uploads.

Table 1

2.1. Notational Conventions

All multi-byte integers in binary formats (xorb headers, shard structures) use little-endian byte order unless otherwise specified.

Hash values are 32 bytes (256 bits). When serialized, they are stored as raw bytes. When displayed as strings, they use a specific byte-swapped hexadecimal format (see Section 6.5).

Range specifications in this document use exclusive end: [start, end). Example: {"start": 0, "end": 4} means indices 0, 1, 2, 3.

2.1.1. Pseudo-Code Conventions

Pseudo-code in this document uses the following conventions:

* for i = a to b: iterates with i taking values a, a+1, ..., b (inclusive)
* for each x in list: iterates over each element in list
* // denotes integer division (truncating toward zero)
* % denotes the modulo operator
* array[start:end] slices from index start (inclusive) to end (exclusive)
* + on arrays denotes concatenation

3. Protocol Overview

XET operates as a client-server protocol. Clients perform content-defined chunking locally, query for deduplication opportunities, form xorbs from new chunks, and upload both xorbs and shards to the server. The CAS server provides APIs for reconstruction queries, global deduplication, and persistent storage.

3.1. Upload Flow

The upload process transforms files into content-addressed storage:

1. Chunking: Split files into variable-sized chunks using content-defined chunking (see Section 5).

2. Deduplication: Query for existing chunks to avoid redundant uploads (see Section 10).

3. Xorb Formation: Group new chunks into xorbs, applying compression (see Section 7).

4. Xorb Upload: Upload serialized xorbs to the CAS server.

5. Shard Formation: Create shard metadata describing file reconstructions.

6. Shard Upload: Upload the shard to register files in the system.

3.2. Download Flow

The download process reconstructs files from stored chunks:

1. Reconstruction Query: Request reconstruction information for a file hash.

2. Term Processing: Parse the ordered list of terms describing the file.

3. Data Fetching: Download required xorb ranges using provided URLs.

4. Chunk Extraction: Deserialize and decompress chunks from xorb data.

5. File Assembly: Concatenate chunks in term order to reconstruct the file.

4. Algorithm Suites

XET is designed as a generic framework where the specific chunking algorithm and cryptographic hash function are parameters defined by an algorithm suite. This enables future algorithm agility while maintaining full backward compatibility within a deployment.

4.1. Suite Definition

An algorithm suite specifies:

1. Cryptographic Hash Function: The hash algorithm used for all content addressing (chunk hashes, xorb hashes, file hashes, verification hashes).

2. Content-Defined Chunking Algorithm: The rolling hash function and boundary detection logic used to split files into chunks.

3. Compression Format: The compression algorithm used for chunk data within xorbs.

4. Keying Material: Domain separation keys for the hash function.

5. Algorithm Parameters: Chunk size bounds, mask values, lookup tables, and other constants.

4.2. Suite Requirements

Any conforming algorithm suite MUST satisfy:

* Determinism: Identical inputs MUST produce identical outputs across all implementations.

* Collision Resistance: The hash function MUST provide at least 128 bits of collision resistance.

* Preimage Resistance: The hash function MUST provide at least 128 bits of preimage resistance.

* Keyed Mode: The hash function MUST support keyed operation for domain separation.

4.3. Suite Negotiation

The algorithm suite used by an XET deployment is determined out-of-band, typically by the CAS server configuration. All clients interacting with a given server MUST use the same suite. Binary formats (xorbs, shards) do not contain suite identifiers; the suite is determined implicitly by the deployment context.

4.4. Defined Suites

This specification defines one algorithm suite:

* XET-BLAKE3-GEARHASH-LZ4: Uses BLAKE3 for all cryptographic hashing, Gearhash for content-defined chunking, and LZ4 for compression. This is the default and currently only defined suite.

Future specifications MAY define additional suites with different algorithms.

5. Content-Defined Chunking

Content-defined chunking (CDC) splits files into variable-sized chunks based on content rather than fixed offsets. This produces deterministic chunk boundaries that remain stable across file modifications, enabling efficient deduplication.

This section describes the chunking algorithm for the XET-BLAKE3-GEARHASH-LZ4 suite. Other algorithm suites MAY define different chunking algorithms with different parameters.

5.1. Gearhash Algorithm

The XET-BLAKE3-GEARHASH-LZ4 suite uses a Gearhash-based rolling hash algorithm [GEARHASH]. Gearhash maintains a 64-bit state that is updated with each input byte using a lookup table, providing fast and deterministic boundary detection.

5.2. Algorithm Parameters

The following constants define the chunking behavior for the XET-BLAKE3-GEARHASH-LZ4 suite:

```
TARGET_CHUNK_SIZE  = 65536      # 64 KiB (2^16 bytes)
MIN_CHUNK_SIZE     = 8192       # 8 KiB (TARGET / 8)
MAX_CHUNK_SIZE     = 131072     # 128 KiB (TARGET * 2)
MASK               = 0xFFFF000000000000  # 16 one-bits
```

The Gearhash algorithm uses a lookup table of 256 64-bit constants. Implementations of the XET-BLAKE3-GEARHASH-LZ4 suite MUST use the table defined in [GEARHASH] (see Appendix B for the complete lookup table).

5.3. Algorithm Description

The algorithm maintains a 64-bit rolling hash value and processes input bytes sequentially:

```
   function chunk_file(data):
       h = 0                    # 64-bit rolling hash
       start_offset = 0         # Start of current chunk
       chunks = []
       n = length(data)

       for i = 0 to n - 1:      # Inclusive range [0, n-1]
           b = data[i]
           h = ((h << 1) + TABLE[b]) & 0xFFFFFFFFFFFFFFFF  # 64-bit wrap

           chunk_size = i - start_offset + 1

           if chunk_size < MIN_CHUNK_SIZE:
               continue

           if chunk_size >= MAX_CHUNK_SIZE:
               chunks.append(data[start_offset : i + 1])
               start_offset = i + 1
               h = 0
               continue

           if (h & MASK) == 0:
               chunks.append(data[start_offset : i + 1])
               start_offset = i + 1
               h = 0

       if start_offset < n:
           chunks.append(data[start_offset : n])

       return chunks
```

5.4. Boundary Rules

The following rules govern chunk boundary placement:

1. Boundaries MUST NOT be placed before MIN_CHUNK_SIZE bytes have been processed in the current chunk.

2. Boundaries MUST be forced when MAX_CHUNK_SIZE bytes have been processed, regardless of hash value.

3. Between minimum and maximum sizes, boundaries are placed when (h & MASK) == 0.

4. The final chunk MAY be smaller than MIN_CHUNK_SIZE if it represents the end of the file.

5. Files smaller than MIN_CHUNK_SIZE produce a single chunk.

5.5. Determinism Requirements

Implementations MUST produce identical chunk boundaries for identical input data. For the XET-BLAKE3-GEARHASH-LZ4 suite, this requires:

* Using the exact lookup table values from Appendix B
* Using 64-bit wrapping arithmetic for hash updates
* Processing bytes in sequential order
* Applying boundary rules consistently

Other algorithm suites MUST specify their own determinism requirements.

5.6. Performance Optimization

Implementations MAY skip boundary checks until chunk_size reaches MIN_CHUNK_SIZE, since boundaries are forbidden before that point.

They MUST still update the rolling hash for every byte; skipping hash updates would change h and therefore alter boundary placement, violating determinism.

6. Hashing Methods

XET uses cryptographic hashing for content addressing, integrity verification, and deduplication. The specific hash function is determined by the algorithm suite. All hashes are 32 bytes (256 bits) in length.

This section describes the hashing methods for the XET-BLAKE3-GEARHASH-LZ4 suite, which uses BLAKE3 keyed hashing [BLAKE3] for all cryptographic hash computations. Different key values provide domain separation between hash types.

6.1. Chunk Hashes

Chunk hashes uniquely identify individual chunks based on their content. The algorithm suite determines how chunk hashes are computed.

For the XET-BLAKE3-GEARHASH-LZ4 suite, chunk hashes use BLAKE3 keyed hash with DATA_KEY as the key:

```
DATA_KEY = {
  0x66, 0x97, 0xf5, 0x77, 0x5b, 0x95, 0x50, 0xde,
  0x31, 0x35, 0xcb, 0xac, 0xa5, 0x97, 0x18, 0x1c,
  0x9d, 0xe4, 0x21, 0x10, 0x9b, 0xeb, 0x2b, 0x58,
  0xb4, 0xd0, 0xb0, 0x4b, 0x93, 0xad, 0xf2, 0x29
}

function compute_chunk_hash(chunk_data):
    return blake3_keyed_hash(DATA_KEY, chunk_data)
```

6.2. Xorb Hashes

Xorb hashes identify xorbs based on their constituent chunks. The hash is computed using a Merkle tree construction where leaf nodes are chunk hashes. The Merkle tree construction is defined separately from the hash function.

6.2.1. Internal Node Hash Function

Internal node hashes combine child hashes with their sizes. The hash function is determined by the algorithm suite.

For the XET-BLAKE3-GEARHASH-LZ4 suite, internal node hashes use BLAKE3 keyed hash with INTERNAL_NODE_KEY as the key:

```
INTERNAL_NODE_KEY = {
  0x01, 0x7e, 0xc5, 0xc7, 0xa5, 0x47, 0x29, 0x96,
  0xfd, 0x94, 0x66, 0x66, 0xb4, 0x8a, 0x02, 0xe6,
  0x5d, 0xdd, 0x53, 0x6f, 0x37, 0xc7, 0x6d, 0xd2,
  0xf8, 0x63, 0x52, 0xe6, 0x4a, 0x53, 0x71, 0x3f
}
```

The input to the hash function is a string formed by concatenating lines for each child:

```
{hash_hex} : {size}\n
```

Where:

* {hash_hex} is the 64-character lowercase hexadecimal representation of the child hash as defined in Section 6.5
* {size} is the decimal representation of the child's byte size
* Lines are separated by newline characters (\n)

6.2.2. Merkle Tree Construction

XET uses an aggregated hash tree construction with variable fan-out, not a traditional binary Merkle tree. This algorithm iteratively collapses a list of (hash, size) pairs until a single root hash remains.

6.2.2.1. Algorithm Parameters

```
MEAN_BRANCHING_FACTOR = 4
MIN_CHILDREN = 2
MAX_CHILDREN = 2 * MEAN_BRANCHING_FACTOR + 1  # 9
```

6.2.2.2. Cut Point Determination

The tree structure is determined by the hash values themselves. A cut point occurs when:

1. At least 3 children have been accumulated AND the current hash modulo MEAN_BRANCHING_FACTOR equals zero, OR

2. The maximum number of children (9) has been reached, OR

3. The end of the input list is reached

Note: When the input has 2 or fewer hashes, all are merged together. This ensures each internal node has at least 2 children.

```
function next_merge_cut(hashes):
    # hashes is a list of (hash, size) pairs
    # Returns the number of entries to merge (cut point)
    n = length(hashes)
    if n <= 2:
        return n

    end = min(MAX_CHILDREN, n)

    # Check indices 2 through end-1 (0-based indexing)
    # Minimum merge is 3 children when input has more than 2 hashes
    for i = 2 to end - 1:
        h = hashes[i].hash
        # Interpret last 8 bytes of hash as little-endian 64-bit unsigned int
        hash_value = u64_le(h[24:32])
        if hash_value % MEAN_BRANCHING_FACTOR == 0:
            return i + 1  # Cut after element i (include i+1 elements)

    return end
```

6.2.2.3. Merging Hash Sequences

```
function merged_hash_of_sequence(hash_pairs):
    # hash_pairs is a list of (hash, size) pairs
    buffer = ""
    total_size = 0

    for each (h, s) in hash_pairs:
        buffer += hash_to_string(h) + " : " + decimal_string(s) + "\n"
        total_size += s

    new_hash = blake3_keyed_hash(INTERNAL_NODE_KEY, utf8_encode(buffer))
    return (new_hash, total_size)
```

This produces lines like:

```
cfc5d07f6f03c29bbf424132963fe08d19a37d5757aaf520bf08119f05cd56d6 : 100
```

Each line contains:

* The hash as a fixed-length 64-character lowercase hexadecimal string
* A space, colon, space (:)
* The size as a decimal integer
* A newline character (\n)

6.2.2.4. Root Computation

```
function compute_merkle_root(entries):
    # entries is a list of (hash, size) pairs
    if length(entries) == 0:
        return ZERO_HASH  # 32 zero bytes

    hv = entries  # Working copy

    while length(hv) > 1:
        write_idx = 0
        read_idx = 0

        while read_idx < length(hv):
            cut = read_idx + next_merge_cut(hv[read_idx:])
            hv[write_idx] = merged_hash_of_sequence(hv[read_idx:cut])
            write_idx += 1
            read_idx = cut

        hv = hv[0:write_idx]

    return hv[0].hash
```

Where ZERO_HASH is 32 bytes of zeros, and hv[start:end] denotes slicing elements from index start (inclusive) to end (exclusive).

6.2.3. Xorb Hash Computation

The xorb hash is the root of a Merkle tree built from chunk hashes:

```
function compute_xorb_hash(chunk_hashes, chunk_sizes):
    n = length(chunk_hashes)
    entries = []
    for i = 0 to n - 1:
        entries.append((chunk_hashes[i], chunk_sizes[i]))
    return compute_merkle_root(entries)
```

6.3. File Hashes

File hashes identify files based on their complete chunk composition. The computation is similar to xorb hashes, but with an additional final keyed hash step for domain separation.

For the XET-BLAKE3-GEARHASH-LZ4 suite, file hashes use an all-zero key (ZERO_KEY) for the final hash:

```
ZERO_KEY = {
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
}

function compute_file_hash(chunk_hashes, chunk_sizes):
    n = length(chunk_hashes)
    entries = []
    for i = 0 to n - 1:
        entries.append((chunk_hashes[i], chunk_sizes[i]))
    merkle_root = compute_merkle_root(entries)
    return blake3_keyed_hash(ZERO_KEY, merkle_root)
```

For empty files (zero bytes), there are no chunks, so compute_merkle_root([]) returns ZERO_HASH (32 zero bytes). The file hash is therefore blake3_keyed_hash(ZERO_KEY, ZERO_HASH).

6.4. Term Verification Hashes

Term verification hashes are used in shards to prove that the uploader possesses the actual file data, not just metadata. The hash function is determined by the algorithm suite.

For the XET-BLAKE3-GEARHASH-LZ4 suite, verification hashes use BLAKE3 keyed hash with VERIFICATION_KEY as the key:

```
VERIFICATION_KEY = {
  0x7f, 0x18, 0x57, 0xd6, 0xce, 0x56, 0xed, 0x66,
  0x12, 0x7f, 0xf9, 0x13, 0xe7, 0xa5, 0xc3, 0xf3,
  0xa4, 0xcd, 0x26, 0xd5, 0xb5, 0xdb, 0x49, 0xe6,
  0x41, 0x24, 0x98, 0x7f, 0x28, 0xfb, 0x94, 0xc3
}
```

The input is the raw concatenation of chunk hashes (not hex-encoded) for the term's chunk range:

```
function compute_verification_hash(chunk_hashes, start_index, end_index):
    # Range is [start_index, end_index) - end is exclusive
    buffer = empty_byte_array()
    for i = start_index to end_index - 1:
        buffer += chunk_hashes[i]  # 32 bytes each
    return blake3_keyed_hash(VERIFICATION_KEY, buffer)
```

6.5. Hash String Representation

When representing hashes as strings, a specific byte reordering is applied before hexadecimal encoding.

6.5.1. Conversion Procedure

The 32-byte hash is interpreted as four little-endian 64-bit unsigned values, and each value is printed as 16 hexadecimal digits:

1. Divide the 32-byte hash into four 8-byte segments

2. Interpret each segment as a little-endian 64-bit unsigned value

3. Format each value as a zero-padded 16-character lowercase hexadecimal string

4. Concatenate the four strings (64 characters total)

```
function hash_to_string(hash):
    out = ""
    for segment = 0 to 3:
        offset = segment * 8
        value = u64_le(hash[offset : offset + 8])
        out += hex16(value)    # 16-digit lowercase hex
    return out

function string_to_hash(hex_string):
    hash = empty_byte_array()
    for segment = 0 to 3:
        start = segment * 16
        value = parse_hex_u64(hex_string[start : start + 16])
        hash += u64_le_bytes(value)
    return hash
```

Where:

* u64_le(bytes) interprets 8 bytes as a little-endian 64-bit unsigned integer

* u64_le_bytes(value) converts a 64-bit unsigned integer to 8 little-endian bytes

* hex16(value) formats a 64-bit value as a 16-character lowercase hexadecimal string

* parse_hex_u64(str) parses a 16-character hexadecimal string as a 64-bit unsigned integer

6.5.2. Example

```
Original hash bytes (indices 0-31):
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31]

Reordered bytes:
[7, 6, 5, 4, 3, 2, 1, 0, 15, 14, 13, 12, 11, 10, 9, 8,
 23, 22, 21, 20, 19, 18, 17, 16, 31, 30, 29, 28, 27, 26, 25, 24]

String representation:
070605040302010 00f0e0d0c0b0a09081716151413121110 1f1e1d1c1b1a1918
```

7. Xorb Format

A xorb is a container that aggregates multiple compressed chunks for efficient storage and transfer. Xorbs are identified by their xorb hash (see Section 6.2).

7.1. Size Constraints

```
MAX_XORB_SIZE   = 67108864  # 64 MiB maximum serialized size
MAX_XORB_CHUNKS = 8192      # Maximum chunks per xorb
```

Implementations MUST NOT exceed either limit. When collecting chunks:

1. Stop if adding the next chunk would exceed MAX_XORB_SIZE
2. Stop if the chunk count would exceed MAX_XORB_CHUNKS
3. Target approximately 1,024 chunks per xorb for typical workloads

7.2. Binary Format

Serialized xorbs have a footer so readers can locate metadata by seeking from the end:

```
+-------------------------------------------------------------+
|                 Chunk Data Region (variable)                |
|   [chunk header + compressed bytes repeated per chunk]      |
+-------------------------------------------------------------+
|                 CasObjectInfo Footer (variable)             |
+-------------------------------------------------------------+
|     Info Length (32-bit unsigned LE, footer length only)    |
+-------------------------------------------------------------+
```

The final 4-byte little-endian integer stores the length of the CasObjectInfo block immediately preceding it (the length does not include the 4-byte length field itself).

The chunk data region consists of consecutive chunk entries, each containing an 8-byte header followed by the compressed chunk data.

7.3. Chunk Header Format

Each chunk header is 8 bytes with the following layout:

Offset	Size	Field
0	1	Version (must be 0)
1	3	Compressed Size (little-endian, bytes)
4	1	Compression Type
5	3	Uncompressed Size (little-endian, bytes)

Table 2

7.3.1. Version Field

The version field MUST be 0 for this specification. Implementations MUST reject chunks with unknown version values.

7.3.2. Size Fields

Both size fields use 3-byte little-endian encoding, supporting values up to 16,777,215 bytes. Given the maximum chunk size of 128 KiB, this provides ample range.

Implementations MUST validate size fields before allocating buffers or invoking decompression:

* uncompressed_size MUST be greater than zero and MUST NOT exceed MAX_CHUNK_SIZE (128 KiB). Chunks that declare larger sizes MUST be rejected and the containing xorb considered invalid.

* compressed_size MUST be greater than zero and MUST NOT exceed the lesser of MAX_CHUNK_SIZE and the remaining bytes in the serialized xorb payload. Oversize or truncated compressed payloads MUST cause the xorb to be rejected.

7.3.3. Compression Type

Value	Name	Description
0	None	No compression; data stored as-is
1	LZ4	LZ4 Frame format compression
2	ByteGrouping4LZ4	Byte grouping preprocessing followed by LZ4

Table 3

7.4. Compression Schemes

7.4.1. None (Type 0)

Data is stored without modification. Used when compression would increase size or for already-compressed data.

7.4.2. LZ4 (Type 1)

LZ4 Frame format compression [LZ4] (not LZ4 block format). Each compressed chunk is a complete LZ4 frame. This is the default compression scheme for most data.

7.4.3. ByteGrouping4LZ4 (Type 2)

A two-stage compression optimized for structured data (e.g., floating-point arrays):

1. Byte Grouping Phase: Reorganize bytes by position within 4-byte groups

2. LZ4 Compression: Apply LZ4 to the reorganized data

Byte grouping transformation:

```
Original:  [A0 A1 A2 A3 | B0 B1 B2 B3 | C0 C1 C2 C3 | ...]
Grouped:   [A0 B0 C0 ... | A1 B1 C1 ... | A2 B2 C2 ... | A3 B3 C3 ...]
```

```
function byte_group_4(data):
    n = length(data)
    groups = [[], [], [], []]

    for i = 0 to n - 1:
        groups[i % 4].append(data[i])

    return groups[0] + groups[1] + groups[2] + groups[3]

function byte_ungroup_4(grouped_data, original_length):
    n = original_length
    base_size = n // 4         # Integer division
    remainder = n % 4

    # Group sizes: first 'remainder' groups get base_size + 1
    sizes = []
    for i = 0 to 3:
        if i < remainder:
            sizes.append(base_size + 1)
        else:
            sizes.append(base_size)

    # Extract groups from grouped_data
    groups = []
    offset = 0
    for i = 0 to 3:
        groups.append(grouped_data[offset : offset + sizes[i]])
        offset += sizes[i]

    # Interleave back to original order
    data = []
    for i = 0 to n - 1:
        group_idx = i % 4
        pos_in_group = i // 4  # Integer division
        data.append(groups[group_idx][pos_in_group])

    return data
```

When the data length is not a multiple of 4, the remainder bytes are distributed to the first groups. For example, with 10 bytes the group sizes are 3, 3, 2, 2 (first two groups get the extra bytes).

7.4.4. Compression Selection

Implementations MAY use any strategy to select compression schemes. If compression increases size, implementations SHOULD use compression type 0 (None).

ByteGrouping4LZ4 (Type 2) is typically beneficial for structured numerical data such as float32 or float16 tensors, where bytes at the same position within 4-byte groups tend to be similar.

7.5. CasObjectInfo Footer

The metadata footer sits immediately before the 4-byte length trailer. Implementations MUST serialize fields in this exact order and reject unknown idents or versions.

7.5.1. Main Header

* Ident: "XETBLOB" (7 ASCII bytes)
* Version: 8-bit unsigned, MUST be 1
* Xorb hash: 32-byte Merkle hash from Section 6.2

7.5.2. Hash Section

* Ident: "XBLBHSH" (7 bytes)
* Hashes version: 8-bit unsigned, MUST be 0
* num_chunks: 32-bit unsigned
* Chunk hashes: 32 bytes each, in chunk order

7.5.3. Boundary Section

* Ident: "XBLBBND" (7 bytes)
* Boundaries version: 8-bit unsigned, MUST be 1
* num_chunks: 32-bit unsigned
* Chunk boundary offsets: Array of num_chunks 32-bit unsigned values. Each value is the end offset (in bytes) of the corresponding chunk in the serialized chunk data region, including headers. Chunk 0 starts at offset 0; chunk i starts at chunk_boundary_offsets[i-1].
* Unpacked chunk offsets: Array of num_chunks 32-bit unsigned values. Each value is the end offset of the corresponding chunk in the concatenated uncompressed stream.

7.5.4. Trailer

* num_chunks: 32-bit unsigned (repeated for convenience)

* Hashes section offset from end: 32-bit unsigned byte offset from the end of the footer to the start of the hash section

* Boundary section offset from end: 32-bit unsigned byte offset from the end of the footer to the start of the boundary section

* Reserved: 16 bytes, zero

The 4-byte length trailer that follows the footer stores info_length (little-endian 32-bit unsigned) for the CasObjectInfo block only. This length field is not counted inside the footer itself.

8. File Reconstruction

A file reconstruction is an ordered list of terms that describes how to reassemble a file from chunks stored in xorbs.

8.1. Term Structure

Each term specifies:

* Xorb Hash: Identifies the xorb containing the chunks

* Chunk Range: Start (inclusive) and end (exclusive) indices within the xorb

* Unpacked Length: Expected byte count after decompression (for validation)

8.2. Reconstruction Rules

1. Terms MUST be processed in order.

2. For each term, extract chunks at indices [start, end) from the specified xorb.

3. Decompress chunks according to their compression headers.

4. Concatenate decompressed chunk data in order.

5. For range queries, apply offset_into_first_range to skip initial bytes.

6. Validate that the total reconstructed size matches expectations.

8.3. Range Queries

When downloading a byte range rather than the complete file:

1. The reconstruction API returns only terms overlapping the requested range.

2. The offset_into_first_range field indicates bytes to skip in the first term.

3. The client MUST truncate output to match the requested range length.

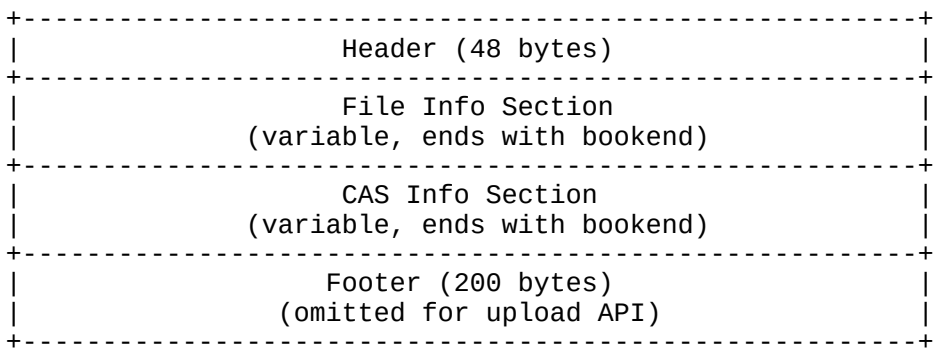
9. Shard Format

A shard is a binary metadata structure that describes file reconstructions and xorb contents. Shards serve two purposes:

1. Upload Registration: Describing newly uploaded files and xorbs to the CAS server

2. Deduplication Response: Providing information about existing chunks for deduplication

9.1. Overall Structure



9.2. Data Types

All multi-byte integers are little-endian. Field sizes are stated explicitly (e.g., “8-bit unsigned”, “32-bit unsigned”, “64-bit unsigned”). Hash denotes a 32-byte (256-bit) value.

9.3. Header

The header is 48 bytes at offset 0:

Offset	Size	Field
0	32	Tag (magic identifier)
32	8	Version (64-bit unsigned, MUST be 2)
40	8	Footer Size (64-bit unsigned, 0 if footer omitted)

The header version (2) and footer version (1) are independent version numbers that may evolve separately.

9.3.1. Magic Tag

The 32-byte magic tag identifies the shard format and the application deployment:

Offset	Size	Field
0	14	Application Identifier (ASCII, null-padded)
14	1	Null byte (0x00)
15	17	Magic sequence (fixed)

The magic sequence (bytes 15-31) MUST be exactly:

```
SHARD_MAGIC_SEQUENCE = {
  0x55, 0x69, 0x67, 0x45, 0x6a, 0x7b, 0x81, 0x57,
  0x83, 0xa5, 0xbd, 0xd9, 0x5c, 0xcd, 0xd1, 0x4a, 0xa9
}
```

The application identifier (bytes 0-13) is deployment-specific and identifies the XET application context. For Hugging Face deployments, the identifier MUST be "HFRepoMetaData" (ASCII):

```
HF_APPLICATION_ID = {
  0x48, 0x46, 0x52, 0x65, 0x70, 0x6f, 0x4d, 0x65,
  0x74, 0x61, 0x44, 0x61, 0x74, 0x61
}
```

Other deployments MAY define their own application identifiers. If the identifier is shorter than 14 bytes, it MUST be null-padded on the right.

Implementations MUST verify that bytes 15-31 match the expected magic sequence before processing. Implementations MAY additionally verify the application identifier to ensure compatibility with the expected deployment.

9.4. File Info Section

The file info section contains zero or more file blocks, each describing a file reconstruction. The section ends with a bookend entry.

9.4.1. File Block Structure

Each file block contains:

1. FileDataSequenceHeader (48 bytes)

2. FileDataSequenceEntry entries (48 bytes each, count from header)

3. FileVerificationEntry entries (48 bytes each, if flag set)

4. FileMetadataExt (48 bytes, if flag set)

9.4.2. FileDataSequenceHeader

Offset	Size	Field
0	32	File Hash
32	4	File Flags (32-bit unsigned)
36	4	Number of Entries (32-bit unsigned)
40	8	Reserved (zeros)

File Flags:

Bit	Name	Description
31	WITH_VERIFICATION	FileVerificationEntry present for each entry
30	WITH_METADATA_EXT	FileMetadataExt present at end

Table 4

9.4.3. FileDataSequenceEntry

Each entry describes a term in the file reconstruction:

Offset	Size	Field
0	32	CAS Hash (xorb hash)
32	4	CAS Flags (32-bit unsigned, reserved, MUST be set to 0)
36	4	Unpacked Segment Bytes (32-bit unsigned)
40	4	Chunk Index Start (32-bit unsigned)
44	4	Chunk Index End (32-bit unsigned, exclusive)

The chunk range is specified as [chunk_index_start, chunk_index_end) (end-exclusive).

9.4.4. FileVerificationEntry

Present only when WITH_VERIFICATION flag is set:

Offset	Size	Field
0	32	Range Hash (verification hash)
32	16	Reserved (zeros)

The range hash is computed as described in Section 6.4.

9.4.5. FileMetadataExt

Present only when WITH_METADATA_EXT flag is set:

Offset	Size	Field
0	32	SHA-256 Hash of file contents
32	16	Reserved (zeros)

9.4.6. Bookend Entry

The file info section ends with a 48-byte bookend:

* Bytes 0-31: All 0xFF

* Bytes 32-47: All 0x00

9.5. CAS Info Section

The CAS info section contains zero or more CAS blocks, each describing a xorb and its chunks. The section ends with a bookend entry.

9.5.1. CAS Block Structure

Each CAS block contains:

1. CASChunkSequenceHeader (48 bytes)

2. CASChunkSequenceEntry entries (48 bytes each, count from header)

9.5.2. CASChunkSequenceHeader

Offset	Size	Field
0	32	CAS Hash (xorb hash)
32	4	CAS Flags (32-bit unsigned, reserved, MUST be set to 0)
36	4	Number of Entries (32-bit unsigned)
40	4	Num Bytes in CAS (32-bit unsigned, total uncompressed)
44	4	Num Bytes on Disk (32-bit unsigned, serialized xorb size)

9.5.3. CASChunkSequenceEntry

Offset	Size	Field
0	32	Chunk Hash
32	4	Chunk Byte Range Start (32-bit unsigned)
36	4	Unpacked Segment Bytes (32-bit unsigned)
40	4	Flags (32-bit unsigned)
44	4	Reserved (32-bit unsigned, zeros)

9.5.3.1. Chunk Byte Range Start Calculation

The chunk_byte_range_start field is the cumulative byte offset of this chunk within the uncompressed xorb data. It is calculated as the sum of unpacked_segment_bytes for all preceding chunks in the xorb:

```
function calculate_byte_range_starts(chunks):
    position = 0
    for each chunk in chunks:
        chunk.byte_range_start = position
        position += chunk.unpacked_segment_bytes
```

Example for a xorb with three chunks:

```
Chunk 0: unpacked_segment_bytes = 1000
         byte_range_start = 0

Chunk 1: unpacked_segment_bytes = 2000
         byte_range_start = 1000

Chunk 2: unpacked_segment_bytes = 500
         byte_range_start = 3000
```

This field enables efficient seeking within a xorb without decompressing all preceding chunks.

9.5.3.2. Chunk Flags

Bit	Name	Description
31	GLOBAL_DEDUP_ELIGIBLE	Chunk is eligible for global deduplication queries (see Section 10.3)
0-30	Reserved	MUST be zero

Table 5

9.5.4. Bookend Entry

The CAS info section ends with a 48-byte bookend (same format as file info bookend).

9.6. Footer

The footer is 200 bytes at the end of the shard. It is REQUIRED for stored shards but MUST be omitted when uploading shards via the upload API.

Offset	Size	Field
0	8	Version (64-bit unsigned, MUST be 1)
8	8	File Info Offset (64-bit unsigned)
16	8	CAS Info Offset (64-bit unsigned)
24	8	File Lookup Offset (64-bit unsigned)
32	8	File Lookup Num Entries (64-bit unsigned)
40	8	CAS Lookup Offset (64-bit unsigned)
48	8	CAS Lookup Num Entries (64-bit unsigned)
56	8	Chunk Lookup Offset (64-bit unsigned)
64	8	Chunk Lookup Num Entries (64-bit unsigned)
72	32	Chunk Hash Key
104	8	Shard Creation Timestamp (64-bit unsigned, Unix epoch seconds)
112	8	Shard Key Expiry (64-bit unsigned, Unix epoch seconds)
120	48	Reserved (zeros)
168	8	Stored Bytes on Disk (64-bit unsigned)
176	8	Materialized Bytes (64-bit unsigned)
184	8	Stored Bytes (64-bit unsigned)
192	8	Footer Offset (64-bit unsigned)

Total size: 200 bytes

9.6.1. Lookup Tables

Between the CAS info section and the footer, stored shards include lookup tables for efficient searching:

9.6.1.1. File Lookup Table

Located at file_lookup_offset, contains file_lookup_num_entries entries. Each entry is 12 bytes:

Offset	Size	Field
0	8	Truncated File Hash (64-bit unsigned, first 8 bytes of hash)
8	4	File Info Entry Index (32-bit unsigned)

Entries are sorted by truncated hash for binary search.

9.6.1.2. CAS Lookup Table

Located at cas_lookup_offset, contains cas_lookup_num_entries entries. Each entry is 12 bytes:

Offset	Size	Field
0	8	Truncated CAS Hash (64-bit unsigned, first 8 bytes of hash)
8	4	CAS Info Entry Index (32-bit unsigned)

Entries are sorted by truncated hash for binary search.

9.6.1.3. Chunk Lookup Table

Located at chunk_lookup_offset, contains chunk_lookup_num_entries entries. Each entry is 16 bytes:

Offset	Size	Field
0	8	Truncated Chunk Hash (64-bit unsigned, first 8 bytes of hash)
8	4	CAS Entry Index (32-bit unsigned)
12	4	Chunk Index within CAS (32-bit unsigned)

Entries are sorted by truncated hash for binary search. When keyed hash protection is enabled, the truncated hash is computed from the keyed chunk hash, not the original.

9.6.2. Chunk Hash Key Usage

In global deduplication responses, chunk hashes in the CAS info section are protected with a keyed hash. Clients MUST:

1. Compute keyed_hash(footer.chunk_hash_key, their_chunk_hash) for each local chunk
2. Search for matches in the shard's CAS info section using the keyed hashes
3. Use matched xorb references for deduplication

If chunk_hash_key is all zeros, chunk hashes are stored without keyed hash protection.

10. Deduplication

XET supports chunk-level deduplication at multiple levels to minimize storage and transfer overhead.

10.1. Local Session Deduplication

Within a single upload session, implementations SHOULD track chunk hashes to avoid processing identical chunks multiple times.

10.2. Cached Metadata Deduplication

Implementations MAY cache shard metadata locally to enable deduplication against recently uploaded content without network queries.

10.3. Global Deduplication

The global deduplication API enables discovering existing chunks across the entire storage system.

10.3.1. Eligibility Criteria

Not all chunks are eligible for global deduplication queries. A chunk is eligible if:

1. It is the first chunk of a file, OR
2. The last 8 bytes of its hash, interpreted as a little-endian 64-bit unsigned integer, satisfy: value % 1024 == 0

10.3.2. Query Process

1. For eligible chunks, query the global deduplication API.
2. On a match, the API returns a shard containing CAS info for xorbs containing the chunk.
3. Chunk hashes in the response are protected with a keyed hash; match by computing keyed hashes of local chunk hashes.
4. Record matched xorb references for use in file reconstruction terms.

10.3.3. Keyed Hash Security

The keyed hash protection ensures that clients can only identify chunks they already possess:

1. The server never reveals raw chunk hashes to clients.
2. Clients must compute keyed_hash(key, local_hash) to find matches.
3. A match confirms the client has the data, enabling reference to the existing xorb.

10.4. Fragmentation Prevention

Aggressive deduplication can fragment files across many xorbs, harming read performance. Implementations SHOULD:

* Prefer longer contiguous chunk ranges over maximum deduplication

* Target minimum run lengths (e.g., 8 chunks or 1 MiB) before accepting deduplicated references

11. Upload Protocol

This section describes the complete procedure for uploading files.

11.1. Step 1: Chunking

Split each file into chunks using the algorithm in Section 5.

For each chunk:

1. Compute the chunk hash (see Section 6.1)
2. Record the chunk data, hash, and size

11.2. Step 2: Deduplication

For each chunk, attempt deduplication in order:

1. Local Session: Check if chunk hash was seen earlier in this session
2. Cached Metadata: Check local shard cache for chunk hash
3. Global API: For eligible chunks, query the global deduplication API

Record deduplication results:

* New chunks: Will be included in xorbs
* Deduplicated chunks: Record existing xorb hash and chunk index

11.3. Step 3: Xorb Formation

Group new (non-deduplicated) chunks into xorbs:

1. Collect chunks maintaining their order within files
2. Form xorbs targeting ~64 MiB total size
3. Compute compression for each chunk
4. Compute xorb hash for each xorb (see Section 6.2)

11.4. Step 4: Xorb Serialization and Upload

For each new xorb:

1. Serialize using the format in Section 7
2. Upload to the CAS server
3. Verify successful response

All xorbs MUST be uploaded before proceeding to shard upload.

11.5. Step 5: Shard Formation

Build the shard structure:

1. For each file, construct file reconstruction terms
2. Compute verification hashes for each term (see Section 6.4)
3. Compute file hash (see Section 6.3)
4. Compute SHA-256 of raw file contents
5. Build CAS info blocks for new xorbs

11.6. Step 6: Shard Upload

1. Serialize the shard without footer
2. Upload to the CAS server
3. Verify successful response

11.7. Ordering and Concurrency

The following ordering constraints apply:

* All xorbs referenced by a shard MUST be uploaded before the shard
* Chunk computation for a file must complete before xorb formation
* Xorb hash computation must complete before shard formation

Within these constraints, operations MAY be parallelized:

* Multiple files can be chunked concurrently

* Multiple xorbs can be uploaded concurrently

* Deduplication queries can run in parallel

12. Download Protocol

This section describes the complete procedure for downloading files.

12.1. Step 1: Query Reconstruction

Request file reconstruction information from the CAS server by providing the file hash. For partial downloads (range queries), specify the desired byte range.

12.2. Step 2: Parse Response

The reconstruction response provides:

* Bytes to skip in the first term (for range queries)

* An ordered list of terms to process

* URLs and byte ranges for downloading xorb data

12.3. Step 3: Download Xorb Data

For each term:

1. Identify the xorb and byte range needed for the term's chunk range

2. Download the xorb data from the provided URL

3. Use HTTP range requests when only a portion of the xorb is needed

Multiple terms may reference the same xorb; implementations SHOULD avoid redundant downloads.

12.4. Step 4: Extract Chunks

For each downloaded xorb range:

1. Parse chunk headers sequentially

2. Decompress chunk data according to compression type

3. Extract chunks for the term's index range

12.5. Step 5: Assemble File

1. For the first term, skip offset_into_first_range bytes
2. Concatenate extracted chunks in term order
3. For range queries, truncate to requested length
4. Write to output file or buffer

12.6. Caching Recommendations

See Section 13 for comprehensive caching guidance. Key recommendations:

* Cache decompressed chunks by hash for reuse across files and sessions
* Avoid caching reconstruction API responses (pre-signed URLs expire quickly)
* Cache shard metadata for local deduplication during uploads

12.7. Error Handling

Implementations SHOULD implement:

* Retry logic with exponential backoff for transient failures
* Validation of decompressed chunk sizes against headers
* Hash verification of reconstructed files when possible

13. Caching Considerations

XET' s content-addressable design enables effective caching at multiple levels. This section provides guidance for implementers on caching strategies and considerations.

13.1. Content Immutability

Objects in XET are identified by cryptographic hashes of their content. This content-addressable design provides a fundamental property: content at a given hash never changes. A xorb with hash H will always contain the same bytes, and a chunk with hash C will always decompress to the same data.

This immutability enables aggressive caching:

* Cached xorb data never becomes stale

* Cached chunk data can be reused indefinitely

* Cache invalidation is never required for content objects

The only time-sensitive elements are authentication tokens and pre-signed URLs, which are discussed separately below.

13.2. Client-Side Chunk Caching

Implementations SHOULD cache decompressed chunk data to avoid redundant decompression and network requests. The chunk hash provides a natural cache key.

13.2.1. Cache Key Design

Chunk caches SHOULD use the chunk hash (32 bytes or its string representation) as the cache key. Since hashes uniquely identify content, there is no risk of cache collisions or stale data.

13.2.2. Cache Granularity

Implementations MAY cache at different granularities:

* Individual chunks: Fine-grained, maximizes deduplication benefit

* Chunk ranges: Coarser-grained, reduces metadata overhead

* Complete xorbs: Simplest, but may cache unused chunks

For most workloads, caching individual chunks by hash provides the best balance of storage efficiency and hit rate.

13.2.3. Eviction Strategies

Since all cached content remains valid indefinitely, eviction is based purely on resource constraints:

* LRU (Least Recently Used): Effective for workloads with temporal locality

* LFU (Least Frequently Used): Effective for workloads with stable hot sets

* Size-aware LRU: Prioritizes keeping smaller chunks that are cheaper to re-fetch

Implementations SHOULD track cache size and implement eviction when storage limits are reached.

13.3. Xorb Data Caching

Raw xorb data (compressed chunks with headers) MAY be cached by clients or intermediaries.

13.3.1. Client-Side Xorb Cache

Caching raw xorb byte ranges avoids repeated downloads but requires decompression on each use. This uses local storage to reduce bandwidth consumption. Implementations SHOULD prefer caching decompressed chunks unless bandwidth is severely constrained.

13.3.2. Byte Range Considerations

When caching partial xorb downloads (byte ranges), implementations SHOULD:

1. Cache at chunk-header-aligned boundaries to enable independent chunk extraction

2. Track which byte ranges are cached for each xorb hash

3. Coalesce adjacent cached ranges when possible

13.4. Shard Metadata Caching

Shard metadata enables deduplication without network queries. Implementations SHOULD cache shards from recent uploads for local deduplication.

13.4.1. Cache Lifetime

Unlike content objects, shard metadata has implicit lifetime constraints:

* Global deduplication responses include a chunk_hash_key that rotates periodically

* The shard_key_expiry field in the footer indicates when the key expires

* After expiry, keyed hash matches will fail

Implementations SHOULD evict cached deduplication shards when their keys expire.

13.4.2. Cache Size

Shard metadata is relatively compact (typically under 1 MiB per upload session). Implementations MAY cache several hundred recent shards without significant storage impact.

13.5. Pre-Signed URL Handling

The reconstruction API returns pre-signed URLs for downloading xorb data. These URLs have short expiration times (typically minutes to hours) and MUST NOT be cached beyond their validity period.

Implementations MUST:

* Use URLs promptly after receiving them
* Re-query the reconstruction API if URLs have expired
* Never persist URLs to disk for later sessions

Reconstruction responses SHOULD be treated as ephemeral and re-fetched when needed rather than cached.

13.6. HTTP Caching Headers

13.6.1. Server Recommendations

CAS servers SHOULD return appropriate HTTP caching headers for xorb downloads:

For xorb content (immutable):

```
Cache-Control: public, immutable, max-age=<url_ttl_seconds>
ETag: "<xorb_hash>"
```

* max-age MUST be set to a value no greater than the remaining validity window of the pre-signed URL used to serve the object (e.g., a URL that expires in 900 seconds MUST NOT be served with max-age larger than 900).
* Servers SHOULD also emit an Expires header aligned to the URL expiry time.
* Shared caches MUST NOT serve the response after either header indicates expiry, even if the content is immutable.

The immutable directive still applies within that bounded window, allowing caches to skip revalidation until the signature expires.

For reconstruction API responses (ephemeral):

```
Cache-Control: private, no-store
```

Reconstruction responses contain pre-signed URLs that expire and MUST NOT be cached by intermediaries.

For global deduplication responses:

```
Cache-Control: private, max-age=3600
Vary: Authorization
```

Deduplication responses are user-specific and may be cached briefly by the client.

13.6.2. Client Recommendations

Clients SHOULD respect Cache-Control headers from servers. When downloading xorb data, clients MAY cache responses locally even if no caching headers are present, since content-addressed data is inherently immutable.

13.7. CDN Integration

XET deployments typically serve xorb data through CDNs. The content-addressable design is well-suited for CDN caching:

* Hash-based URLs enable cache key stability
* Immutable content eliminates cache invalidation complexity
* Range requests enable partial caching of large xorbs

13.7.1. CDN Cache Keys

Effective cache key design determines whether multiple users can share cached xorb data. Since xorb content is immutable and identified by hash, the ideal cache key includes only the xorb hash and byte range, maximizing cache reuse. However, access control requirements constrain this choice.

Two URL authorization strategies are applicable to XET deployments:

Edge-Authenticated URLs. The URL path contains the xorb hash with no signature parameters. Authorization is enforced at the CDN edge via signed cookies or tokens validated on every request. The cache key is derived from the xorb hash and byte range only, excluding any authorization tokens. This allows all authorized users to share the

same cache entries. This pattern requires CDNs capable of per-request authorization; generic shared caches without edge auth MUST NOT be used.

Query-Signed URLs. The URL includes signature parameters in the query string (similar to pre-signed cloud storage URLs). Cache keys MUST include all signature-bearing query parameters. Each unique signature produces a separate cache entry, resulting in lower hit rates. This approach works with any CDN but sacrifices cache efficiency for simplicity.

For both strategies:

* Cache keys SHOULD include the byte range when Range headers are present

* Cache keys SHOULD NOT include Authorization headers, since different users have different tokens but request identical content

For deployments with access-controlled content (e.g., gated models requiring user agreement), see Section 14.4 for additional CDN considerations.

13.7.2. Range Request Caching

CDNs SHOULD cache partial responses (206 Partial Content) by byte range. When a subsequent request covers a cached range, the CDN can serve from cache without contacting the origin.

Some CDNs support range coalescing, where multiple partial caches are combined to serve larger requests. This is particularly effective for XET where different users may request different chunk ranges from the same xorb.

13.8. Proxy and Intermediary Considerations

Corporate proxies and other intermediaries MAY cache XET traffic.

Pre-signed URLs include authentication in the URL itself, allowing unauthenticated intermediaries to cache responses.

However, reconstruction API requests include authentication tokens and SHOULD NOT be cached by intermediaries.

14. Security Considerations

14.1. Content Integrity

XET provides content integrity through cryptographic hashing:

* Chunk hashes verify individual chunk integrity
* Xorb hashes verify complete xorb contents
* File hashes verify complete file reconstruction

Implementations SHOULD verify hashes when possible, particularly for downloaded content.

14.2. Authentication and Authorization

Token-based authentication controls access to storage operations. Implementations MUST:

* Transmit tokens only over TLS-protected connections
* Avoid logging tokens
* Implement token refresh before expiration
* Use minimum required scope (prefer read over write)

14.3. Global Deduplication Privacy

The keyed hash protection in global deduplication prevents enumeration attacks:

* Servers never reveal raw chunk hashes
* Clients can only match chunks they possess
* The chunk hash key rotates periodically, and shard expiry limits the reuse window

14.4. Access-Controlled Content

XET deployments may support access-controlled or “gated” content, where users must be authorized (e.g., by accepting terms of service or requesting access) before downloading certain files. This has several implications for XET implementations.

14.4.1. Repository-Level Access Control

Access control in XET is typically enforced at the repository or file level, not at the xorb or chunk level. The reconstruction API MUST verify that the requesting user has access to the file before returning pre-signed URLs. Unauthorized requests MUST return 401 Unauthorized or 403 Forbidden.

14.4.2. CDN Considerations for Gated Content

Since the same xorb may be referenced by both public and access-controlled files, CDN caching requires careful design:

Edge-Authenticated Deployments. When using edge authentication (cookies or tokens validated per-request), the CDN enforces access control on every request. Xorbs referenced only by access-controlled files remain protected even when cached. This is the recommended approach for deployments with gated content.

Query-Signed URL Deployments. When using query-signed URLs, each authorized user receives unique signatures. Cache efficiency is reduced, but access control is enforced by signature validity. Deployments MAY choose to exclude xorbs from access-controlled repositories from CDN caching entirely.

14.4.3. Cross-Repository Deduplication

The same chunk may exist in both access-controlled and public repositories. XET's content-addressable design allows storage deduplication across access boundaries:

* When a user uploads to a public repository, chunks matching access-controlled content may be deduplicated
* The user does not gain access to the access-controlled repository; they simply avoid re-uploading data they already possess
* The keyed hash protection in global deduplication (Section 10.3) ensures users can only match chunks they possess

This is a storage optimization, not an access control bypass. Implementations MUST still enforce repository-level access control for all download operations.

14.4.4. Privacy Implications

Deployments with access-controlled content SHOULD consider:

* Global deduplication queries reveal chunk existence (via 200/404 responses), though not which repositories contain the chunk

* Keyed hash protection in responses ensures clients can only identify chunks they already possess; key rotation limits temporal correlation

* For highly sensitive content, deployments MAY exclude chunks from the global deduplication index entirely

14.5. Denial of Service Considerations

Large file uploads could exhaust server resources. Servers SHOULD implement:

* Rate limiting on API endpoints

* Maximum shard size limits

* Maximum xorb size limits (MAX_XORB_SIZE, 64 MiB)

IANA Considerations

This document does not require any IANA actions.

References

Normative References

[BLAKE3] Aumasson, J., Neves, S., O'Connor, J., and Z. Wilcox-O'Hearn, "BLAKE3: One function, fast everywhere", 9 January 2020, <https://github.com/BLAKE3-team/BLAKE3-specs/blob/master/blake3.pdf>.

[LZ4] Collet, Y., "LZ4 Frame Format Description", 2015, <https://github.com/lz4/lz4/blob/dev/doc/lz4_Frame_format.md>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

Informative References

[FASTCDC] Feng, D., Hu, Y., Hua, Y., Jiang, H., Liu, Q., Xia, W., Zhang, Y., and Y. Zhou, "FastCDC: A Fast and Efficient Content-Defined Chunking Approach for Data Deduplication", USENIX ATC 2016 , 2016, <https://www.usenix.org/conference/atc16/technical-sessions/presentation/xia>.

[GEARHASH] Rijsdijk, S., "rust-gearhash: Fast, SIMD-accelerated GEAR hashing", 2020, <https://github.com/srijs/rust-gearhash>.

[MERKLE] Merkle, R. C., "A Digital Signature Based on a Conventional Encryption Function", CRYPTO 1987, LNCS 293, pp. 369-378 , 1987.

Appendix A. Recommended HTTP API

This appendix defines a recommended HTTP API for CAS servers implementing the XET protocol. This is informative guidance; deployments MAY use different URL structures, authentication mechanisms, or transport protocols entirely.

A.1. Authentication

API requests requiring authorization use a Bearer token in the Authorization header:

Authorization: Bearer <access_token>

Token format, acquisition and refresh mechanisms are deployment-specific.

A.2. Common Headers

Request headers:

* Authorization: Bearer token (when authentication is required)

* Content-Type: application/octet-stream for binary uploads

* Range: Byte range for partial downloads (optional)

Response headers:

* Content-Type: application/json or application/octet-stream

A.3. Get File Reconstruction

Retrieves reconstruction information for downloading a file.

GET /api/v1/reconstructions/{file_hash}

Path parameters:

* file_hash: File hash as hex string (see Section 6.5)

Optional request headers:

* Range: bytes={start}-{end}: Request reconstruction for a specific byte range

Response (200 OK):

```
{
  "offset_into_first_range": 0,
  "terms": [
    {
      "hash": "<xorb_hash_hex>",
      "unpacked_length": 263873,
      "range": {
        "start": 0,
        "end": 4
      }
    }
  ],
  "fetch_info": {
    "<xorb_hash_hex>": [
      {
        "range": {
          "start": 0,
          "end": 4
        },
        "url": "https://...",
        "url_range": {
          "start": 0,
          "end": 131071
        }
      }
    ]
  }
}
```

Response fields:

* offset_into_first_range: Bytes to skip in first term (for range queries)

* terms: Ordered list of reconstruction terms

* fetch_info: Map from xorb hash to fetch information

Fetch info fields:

* range: Chunk index range this entry covers

* url: Pre-signed URL for downloading xorb data

* url_range: Byte range within the xorb (end inclusive), directly usable as HTTP Range header values

Chunk index ranges (range fields) continue to use the document-wide [start, end) convention (exclusive end; see Section 2.1), while url_range follows HTTP Range semantics and is therefore inclusive.

Error responses:

* 400 Bad Request: Invalid file hash format

* 401 Unauthorized: Missing or invalid token

* 404 Not Found: File does not exist

* 416 Range Not Satisfiable: Invalid byte range

A.4. Query Chunk Deduplication

Checks if a chunk exists in the global deduplication index.

GET /api/v1/chunks/{namespace}/{chunk_hash}

Path parameters:

* namespace: Deduplication namespace (e.g., default-merkledb)

* chunk_hash: Chunk hash as hex string (see Section 6.5)

Response (200 OK): Shard format binary (see Section 9)

The returned shard contains CAS info for xorbs that include the queried chunk. Chunk hashes in the response are protected with a keyed hash (see Section 10.3).

Response (404 Not Found): Chunk is not tracked by global deduplication.

A.5. Upload Xorb

Uploads a serialized xorb to storage.

```
POST /api/v1/xorbs/{namespace}/{xorb_hash}
Content-Type: application/octet-stream
```

Path parameters:

* namespace: Storage namespace (e.g., default)

* xorb_hash: Xorb hash as hex string (see Section 6.5)

Request body: Serialized xorb binary (see Section 7)

Response (200 OK):

```
{
  "was_inserted": true
}
```

The was_inserted field is false if the xorb already existed; this is not an error.

Error responses:

* 400 Bad Request: Hash mismatch or invalid xorb format

* 401 Unauthorized: Missing or invalid token

* 403 Forbidden: Insufficient token scope

A.6. Upload Shard

Uploads a shard to register files in the system.

```
POST /api/v1/shards
Content-Type: application/octet-stream
```

Request body: Serialized shard without footer (see Section 9)

Response (200 OK):

```
{
  "result": 0
}
```

Result values:

* 0: Shard already exists

* 1: Shard was registered

Error responses:

* 400 Bad Request: Invalid shard format or referenced xorb missing

* 401 Unauthorized: Missing or invalid token

* 403 Forbidden: Insufficient token scope

Appendix B. Gearhash Lookup Table

The XET-BLAKE3-GEARHASH-LZ4 content-defined chunking algorithm requires a lookup table of 256 64-bit constants. Implementations of this suite MUST use the exact values below for determinism.

```
TABLE = [
    0xb088d3a9e840f559, 0x5652c7f739ed20d6, 0x45b28969898972ab, 0x6b0a89d5b68ec777,
    0x368f573e8b7a31b7, 0x1dc636dce936d94b, 0x207a4c4e5554d5b6, 0xa474b34628239acb,
    0x3b06a83e1ca3b912, 0x90e78d6c2f02baf7, 0xe1c92df7150d9a8a, 0x8e95053a1086d3ad,
    0x5a2ef4f1b83a0722, 0xa50fac949f807fae, 0x0e7303eb80d8d681, 0x99b07edc1570ad0f,
    0x689d2fb555fd3076, 0x00005082119ea468, 0xc4b08306a88fcc28, 0x3eb0678af6374afd,
    0xf19f87ab86ad7436, 0xf2129fbfbe6bc736, 0x481149575c98a4ed, 0x0000010695477bc5,
    0x1fba37801a9ceacc, 0x3bf06fd663a49b6d, 0x99687e9782e3874b, 0x79a10673aa50d8e3,
    0xe4accf9e6211f420, 0x2520e71f87579071, 0x2bd5d3fd781a8a9b, 0x00de4dcddd11c873,
    0xeaa9311c5a87392f, 0xdb748eb617bc40ff, 0xaf579a8df620bf6f, 0x86a6e5da1b09c2b1,
    0xcc2fc30ac322a12e, 0x355e2afec1f74267, 0x2d99c8f4c021a47b, 0xbade4b4a9404cfc3,
    0xf7b518721d707d69, 0x3286b6587bf32c20, 0x0000b68886af270c, 0xa115d6e4db8a9079,
    0x484f7e9c97b2e199, 0xccca7bb75713e301, 0xbf2584a62bb0f160, 0xade7e813625dbcc8,
    0x000070940d87955a, 0x8ae69108139e626f, 0xbd776ad72fde38a2, 0xfb6b001fc2fcc0cf,
    0xc7a474b8e67bc427, 0xbaf6f11610eb5d58, 0x09cb1f5b6de770d1, 0xb0b219e6977d4c47,
    0x00ccbc386ea7ad4a, 0xcc849d0adf973f01, 0x73a3ef7d016af770, 0xc807d2d386bdbdfe,
    0x7f2ac9966c791730, 0xd037a86bc6c504da, 0xf3f17c661eaa609d, 0xaca626b04daae687,
    0x755a99374f4a5b07, 0x90837ee65b2caede, 0x6ee8ad93fd560785, 0x0000d9e11053edd8,
    0x9e063bb2d21cdbd7, 0x07ab77f12a01d2b2, 0xec550255e6641b44, 0x78fb94a8449c14c6,
    0xc7510e1bc6c0f5f5, 0x0000320b36e4cae3, 0x827c33262c8b1a2d, 0x14675f0b48ea4144,
    0x267bd3a6498deceb, 0xf1916ff982f5035e, 0x86221b7ff434fb88, 0x9dbecee7386f49d8,
    0xea58f8cac80f8f4a, 0x008d198692fc64d8, 0x6d38704fbabf9a36, 0xe032cb07d1e7be4c,
    0x228d21f6ad450890, 0x635cb1bfc02589a5, 0x4620a1739ca2ce71, 0xa7e7dfe3aae5fb58,
    0x0c10ca932b3c0deb, 0x2727fee884afed7b, 0xa2df1c6df9e2ab1f, 0x4dcdd1ac0774f523,
```

```
    0x000070ffad33e24e, 0xa2ace87bc5977816, 0x9892275ab4286049, 0xc2861181ddf18959,
    0xbb9972a042483e19, 0xef70cd3766513078, 0x00000513abfc9864, 0xc058b61858c94083,
    0x09e850859725e0de, 0x9197fb3bf83e7d94, 0x7e1e626d12b64bce, 0x520c54507f7b57d1,
    0xbee1797174e22416, 0x6fd9ac3222e95587, 0x0023957c9adfbf3e, 0xa01c7d7e234bbe15,
    0xaba2c758b8a38cbb, 0x0d1fa0ceec3e2b30, 0x0bb6a58b7e60b991, 0x4333dd5b9fa26635,
    0xc2fd3b7d4001c1a3, 0xfb41802454731127, 0x65a56185a50d18cb, 0xf67a02bd8784b54f,
    0x696f11dd67e65063, 0x00002022fca814ab, 0x8cd6be912db9d852, 0x695189b6e9ae8a57,
    0xee9453b50ada0c28, 0xd8fc5ea91a78845e, 0xab86bf191a4aa767, 0x0000c6b5c86415e5,
    0x267310178e08a22e, 0xed2d101b078bca25, 0x3b41ed84b226a8fb, 0x13e622120f28dc06,
    0xa315f5ebfb706d26, 0x8816c34e3301bace, 0xe9395b9cbb71fdae, 0x002ce9202e721648,
    0x4283db1d2bb3c91c, 0xd77d461ad2b1a6a5, 0xe2ec17e46eeb866b, 0xb8e0be4039fbc47c,
    0xdea160c4d5299d04, 0x7eec86c8d28c3634, 0x2119ad129f98a399, 0xa6ccf46b61a283ef,
    0x2c52cedef658c617, 0x2db4871169acdd83, 0x0000f0d6f39ecbe9, 0x3dd5d8c98d2f9489,
    0x8a1872a22b01f584, 0xf282a4c40e7b3cf2, 0x8020ec2ccb1ba196, 0x6693b6e09e59e313,
    0x0000ce19cc7c83eb, 0x20cb5735f6479c3b, 0x762ebf3759d75a5b, 0x207bfe823d693975,
    0xd77dc112339cd9d5, 0x9ba7834284627d03, 0x217dc513e95f51e9, 0xb27b1a29fc5e7816,
    0x00d5cd9831bb662d, 0x71e39b806d75734c, 0x7e572af006fb1a23, 0xa2734f2f6ae91f85,
    0xbf82c6b5022cddf2, 0x5c3beac60761a0de, 0xcdc893bb47416998, 0x6d1085615c187e01,
    0x77f8ae30ac277c5d, 0x917c6b81122a2c91, 0x5b75b699add16967, 0x0000cf6ae79a069b,
    0xf3c40afa60de1104, 0x2063127aa59167c3, 0x621de62269d1894d, 0xd188ac1de62b4726,
    0x107036e2154b673c, 0x0000b85f28553a1d, 0xf2ef4e4c18236f3d, 0xd9d6de6611b9f602,
    0xa1fc7955fb47911c, 0xeb85fd032f298dbd, 0xbe27502fb3befae1, 0xe3034251c4cd661e,
    0x441364d354071836, 0x0082b36c75f2983e, 0xb145910316fa66f0, 0x021c069c9847caf7,
    0x2910dfc75a4b5221, 0x735b353e1c57a8b5, 0xce44312ce98ed96c, 0xbc942e4506bdfa65,
    0xf05086a71257941b, 0xfec3b215d351cead, 0x00ae1055e0144202, 0xf54b40846f42e454,
    0x00007fd9c8bcbcc8, 0xbfbd9ef317de9bfe, 0xa804302ff2854e12, 0x39ce4957a5e5d8d4,
    0xffb9e2a45637ba84, 0x55b9ad1d9ea0818b, 0x00008acbf319178a, 0x48e2bfc8d0fbfb38,
    0x8be39841e848b5e8, 0x0e2712160696a08b, 0xd51096e84b44242a, 0x1101ba176792e13a,
    0xc22e770f4531689d, 0x1689eff272bbc56c, 0x00a92a197f5650ec, 0xbc765990bda1784e,
    0xc61441e392fcb8ae, 0x07e13a2ced31e4a0, 0x92cbe984234e9d4d, 0x8f4ff572bb7d8ac5,
    0x0b9670c00b963bd0, 0x62955a581a03eb01, 0x645f83e5ea000254, 0x41fce516cd88f299,
    0xbbda9748da7a98cf, 0x0000aab2fe4845fa, 0x19761b069bf56555, 0x8b8f5e8343b6ad56,
    0x3e5d1cfd144821d9, 0xec5c1e2ca2b0cd8f, 0xfaf7e0fea7fbb57f, 0x000000d3ba12961b,
    0xda3f90178401b18e, 0x70ff906de33a5feb, 0x0527d5a7c06970e7, 0x22d8e773607c13e9,
    0xc9ab70df643c3bac, 0xeda4c6dc8abe12e3, 0xecef1f410033e78a, 0x0024c2b274ac72cb,
    0x06740d954fa900b4, 0x1d7a299b323d6304, 0xb3c37cb298cbead5, 0xc986e3c76178739b,
    0x9fabea364b46f58a, 0x6da214c5af85cc56, 0x17a43ed8b7a38f84, 0x6eccec511d9adbeb,
    0xf9cab30913335afb, 0x4a5e60c5f415eed2, 0x00006967503672b4, 0x9da51d121454bb87,
    0x84321e13b9bbc816, 0xfb3d6fb6ab2fdd8d, 0x60305eed8e160a8d, 0xcbbf4b14e9946ce8,
    0x00004f63381b10c3, 0x07d5b7816fcc4e10, 0xe5a536726a6a8155, 0x57afb23447a07fdd,
    0x18f346f7abc9d394, 0x636dc655d61ad33d, 0xcc8bab4939f7f3f6, 0x63c7a906c1dd187b
]
```

This table is from the rust-gearhash crate [GEARHASH].

Appendix C. Test Vectors

The following test vectors are for the XET-BLAKE3-GEARHASH-LZ4 algorithm suite.

C.1. Chunk Hash Test Vector

Input (ASCII): Hello World!
Input (hex): 48656c6c6f20576f726c6421

Hash (raw hex, bytes 0-31):
a29cfb08e608d4d8726dd8659a90b9134b3240d5d8e42d5fcb28e2a6e763a3e8

Hash (XET string representation):
d8d408e608fb9ca213b9909a65d86d725f2de4d8d540324be8a363e7a6e228cb

C.2. Hash String Conversion Test Vector

The XET hash string format interprets the 32-byte hash as four little-endian 64-bit unsigned values and prints each as 16 hexadecimal digits.

Hash bytes [0..31]:
00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f

Expected XET string:
0706050403020100f0e0d0c0b0a090817161514131211101f1e1d1c1b1a1918

See the hash_to_string function in Section 6.5 for the conversion algorithm.

C.3. Internal Node Hash Test Vector

Child 1:
hash (XET string): c28f58387a60d4aa200c311cda7c7f77f686614864f5869eadebf765d0a14a69
size: 100

Child 2:
hash (XET string): 6e4e3263e073ce2c0e78cc770c361e2778db3b054b98ab65e277fc084fa70f22
size: 200

Buffer being hashed (ASCII, with literal \n newlines):
c28f58387a60d4aa200c311cda7c7f77f686614864f5869eadebf765d0a14a69 : 100\n
6e4e3263e073ce2c0e78cc770c361e2778db3b054b98ab65e277fc084fa70f22 : 200\n

Result (XET string):
be64c7003ccd3cf4357364750e04c9592b3c36705dee76a71590c011766b6c14

C.4. Verification Range Hash Test Vector

Input: Two chunk hashes from the Internal Node Hash Test Vector above, concatenated as raw bytes (not XET string format).

Chunk hash 1 (raw hex):
aad4607a38588fc2777f7cda1c310c209e86f564486186f6694aa1d065f7ebad

Chunk hash 2 (raw hex):
2cce73e063324e6e271e360c77cc780e65ab984b053bdb78220fa74f08fc77e2

Concatenated input (64 bytes, raw hex):
aad4607a38588fc2777f7cda1c310c209e86f564486186f6694aa1d065f7ebad
2cce73e063324e6e271e360c77cc780e65ab984b053bdb78220fa74f08fc77e2

Verification hash (XET string):
eb06a8ad81d588ac05d1d9a079232d9c1e7d0b07232fa58091caa7bf333a2768

C.5. Reference Files

Complete reference files including sample chunks, xorbs, and shards are available at: https://huggingface.co/datasets/xet-team/xet-spec-reference-files

Acknowledgments

The XET protocol was invented by Hailey Johnson and Yucheng Low at Hugging Face. This specification is based on the reference implementation and documentation developed by the Hugging Face team.

Author’s Address

Frank Denis
Independent Contributor
Email: fde@00f.net