LTFS-VOF

April 24, 2023

1 LTFS Versioned Object Format

The LTFS Versioned Object Format (LTFS-VOF) is a format for saving object data to tapes. This document describes the format so that others may create compatible systems or tools. Python code is included which implements decoding of LTFS-VOF data and metadata. You can download and use this Jupyter notebook interactively to decode your own data.

This specification overlays the Linear Tape File System (LTFS). An implementation of the LTFS-VOF will want to refer to the LTFS specification or use a pre-built LTFS driver. LTFS provides a format for storing files on tape with a POSIX programming interface. The Versioned Object Format layers on top of one or more LTFS tapes to provide:

1. Efficient object and metadata packing. LTFS-VOF uses large files on LTFS, which both minimizes the overhead of LTFS metadata and tape load/unload time.

2. Support for very small and very large object sizes. Very small objects will be packed into large LTFS files, allowing rapid transfer of many small objects to/from tape. Very large objects may span tapes.

3. S3-compliant object versioning. Unlike POSIX, objects in LTFS-VOF may have multiple versions, including delete markers.

4. S3-compliant object names and metadata. POSIX and S3 have different naming restrictions and differences in the format of metadata. LTFS-VOF captures object metadata in LTFS files, similar to object data.

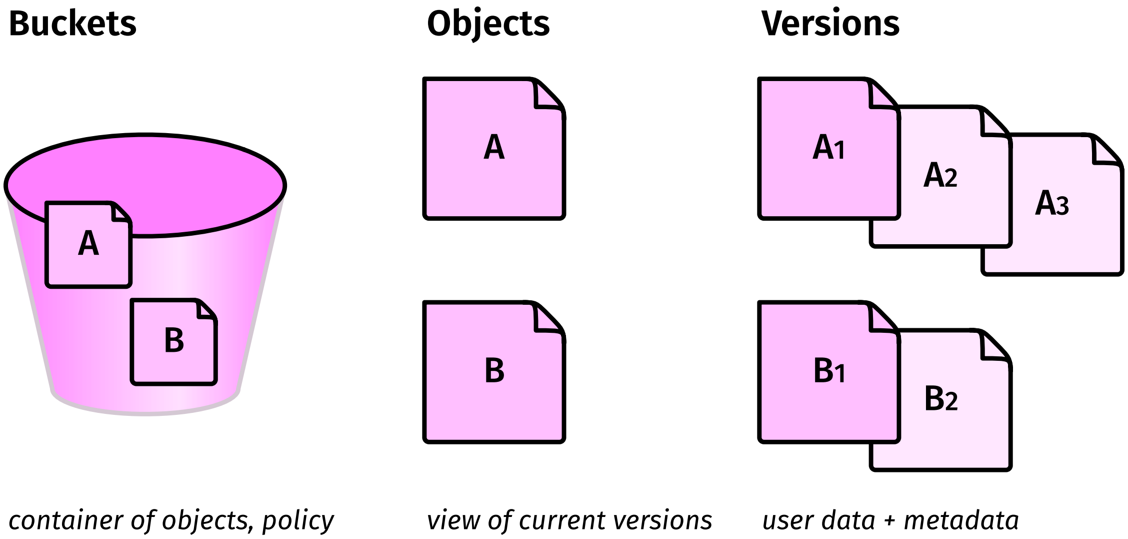
5. Modern compression, encryption, and hash codes. LTFS-VOF uses Zstandard compression, which allows users a great deal of flexibility to trade off speed vs. compression efficiency. AES-256 encryption is used, with flexible AES key identifiers that may reference various encryption key managers. Data integrity is assured with modern hash codes such as XXHash.

6. Support for tape-set parity. In configurations with multiple tape libraries, parity packs may be stored to maximize data availability in the event that a library is down or a set of tapes is damaged. (This feature is not documented in this version of the LTFS-VOF format, but will be included in a future version.)

In addition to LTFS, a LTFS-VOF implementation uses MessagePack to encode various structures. MessagePack implementations are available for most programming languages.

2 Object System Concepts

S3-compatible object stores have different concepts and terminology than file systems. This section provides an overview.



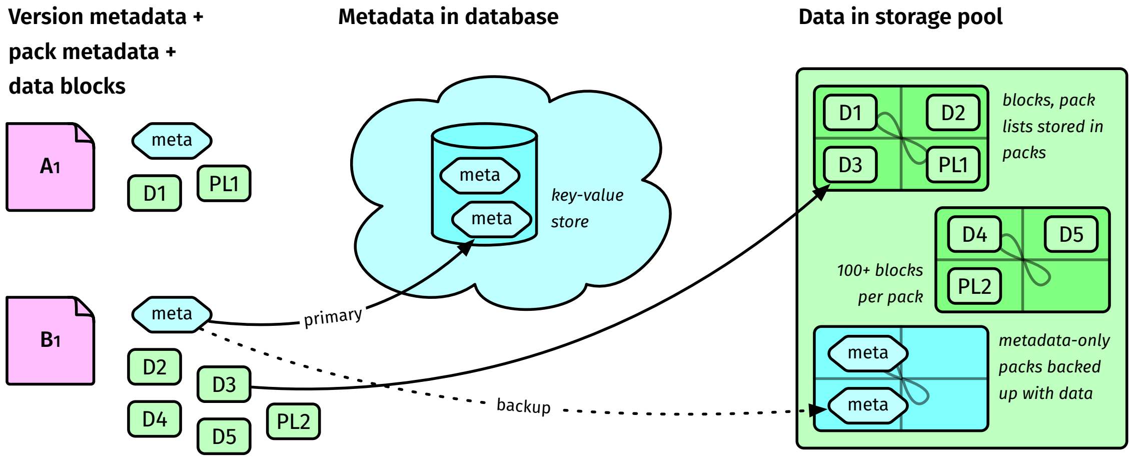
2.0.1 Buckets

A *bucket* is the outermost container for objects; it is most similar to a file system. Buckets tend to have high-level policies that apply to all objects within them, for example lifecycle policies that control where data is placed and when tape copies are mode.

2.0.2 Objects and Versions

Buckets contain *objects*. Each object has a name (or key) and other metadata associated with it, and a map of data blocks. Objects have one or more *versions*. The last version for a given name is called the *current version*. The object may be thought of as a pointer to the current version. To avoid confusion, this document will almost exclusively discuss versions instead of objects.

In addition to the object data, a version has metadata about the version record itself, for example the version's ID, ETag, and creation time. The version metadata is also saved to tape so that the tape set will contain both the data and metadata for all objects.



2.0.3 Blocks

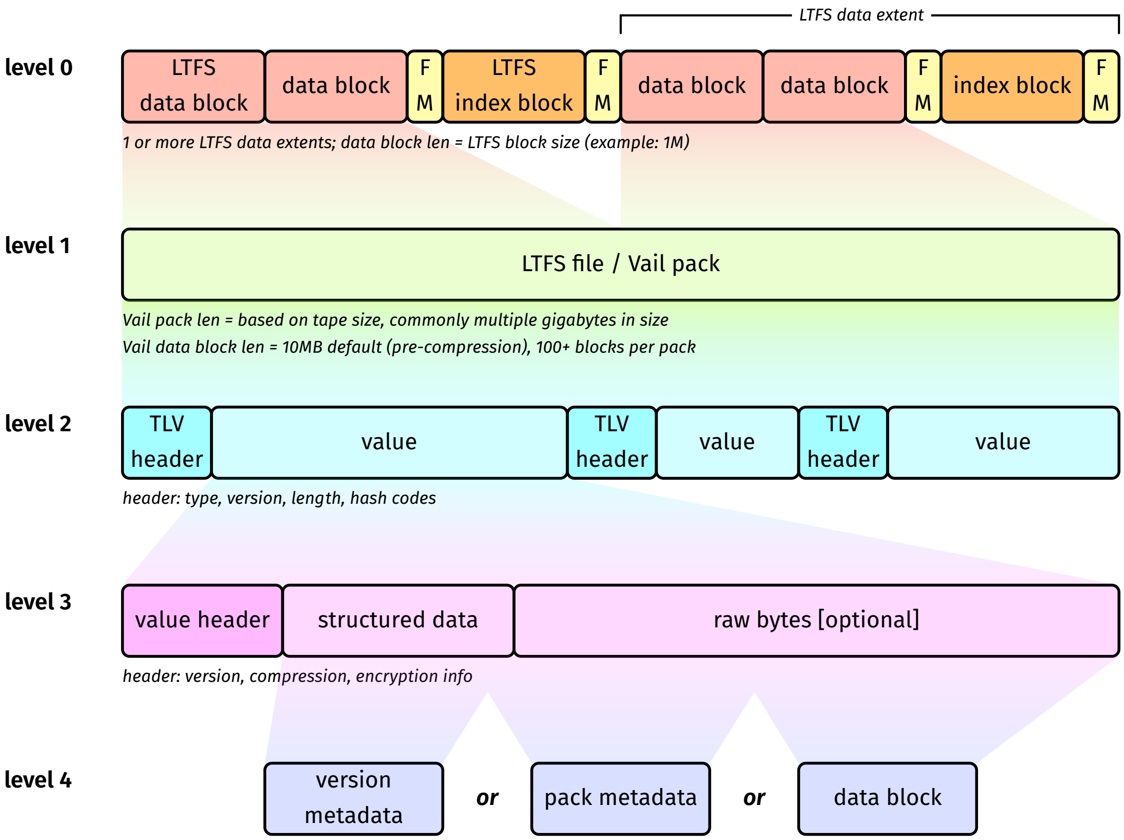
A version's data is stored in one or more *blocks*. Each block is a slice of object data, typically 10MB in size before compression. An object whose length is less than the block length will simply be composed of one short block. A larger object will be composed of multiple blocks with a short block at the end. Each block is compressed and encrypted individually so that a S3 client performing range reads may be answered by decoding only the blocks containing the range the client is asking for.

A set of blocks are stored together in *packs*. Packs are typically large files, where the optimal pack size is determined by the type of storage media. For tape, packs will be multiple gigabytes in size, and will contain hundreds of blocks stored end-to-end. Packs may contain blocks belonging to many versions.

Packs will also contain metadata in the form of pack lists and version records, not just blocks. This metadata allows a tape set to be fully self-describing.

3 Tape Format Overview

The LTFS Versioned Object Format is composed of several levels. At each level is an encoding scheme that is straightforward to implement, while providing high runtime efficiency. This section presents an overview of each level, with detailed descriptions in following sections.



1 or more LTFS data extents; data block len = LTFS block size (example: 1M)

Vail pack len = based on tape size, commonly multiple gigabytes in size
Vail data block len = 10MB default (pre-compression), 100+ blocks per pack

header: type, version, length, hash codes

header: version, compression, encryption info

3.0.1 Level 0: LTFS

The base (zero) level in the system is LTFS, as specified by the LTFS Format Specification version 2 or later. Any LTFS-compliant driver may be used. The LTFS-VOF should be implemented in a user-space process, using standard POSIX file system calls to manipulate the files on a LTFS tape. LTFS uses a combination of data blocks, index blocks, and file marks to lay out data on a tape. This is entirely transparent to the Vail Tape Format, however.

3.0.2 Level 1: Packs

The first level is *packs*, which are LTFS files that store encoded data or metadata. Each pack stores either object data in the form of *blocks*, or metadata in the form of *versions*. Both are described in later sections.

LTFS may use one or more extents to store a file, and each extent is a series of data blocks followed by an index block describing the extent. LTFS also stores the index block in a separate index partition which is read when a tape is mounted. Level one is also mostly transparent to a LTFS-VOF implementation.

3.0.3 Level 2: TLV

The second level is an end-to-end stacking of records within packs. Each record is encoded with a tag-length-value (TLV) format that uses a fixed-size header and variable-length value. This header contains the minimal information required to identify the type of data stored, its length, and hash codes to validate both the header's integrity and the value's integrity. Any TLV may be read and decoded by simply reading its range of bytes within the pack.

3.0.4 Level 3: Value Encoding

The third level uses a variable-size header that provides details on compression, encryption, and the application version used to encode the value. MessagePack is used to decode the header first, then the value contents must be decrypted and decompressed, then those block/version bytes are decoded.

3.0.5 Level 4: Data and Metadata

The final level is the LTFS-VOF data and metadata objects. These are the version records, pack lists, and data blocks which comprise all the data stored in the system.

3.1 Pack Files

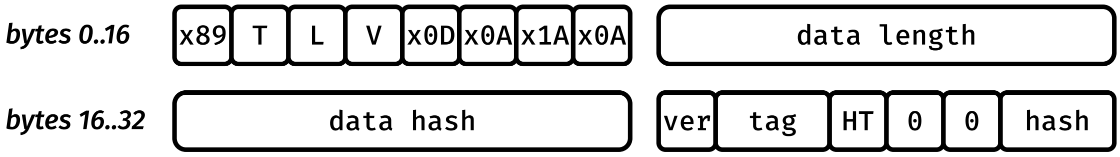
Packs are named with a ULID followed by the extension `.blk` for packs containing blocks, or `.ver` for packs containing versions. Blocks and versions are stored separately so that a system importing a set of tapes need only scan the `.ver` packs to build its database of metadata.

ULID is similar to a UUID, however it contains an embedded timestamp and sorts lexicographically from oldest time to newest time. Pack IDs should use the current time (in UTC) when the pack is first written to.

Pack files should be stored in the root directory of a LTFS tape.

It is important for read performance that pack files be stored as long runs of contiguous LTFS data extents. While LTFS supports interleaving of file data–which may happen if multiple files are written concurrently–this is not very efficient, as reading such a file will require LTFS to perform many seeks. Therefore it is strongly recommended that a LTFS-VOF writer sequentially write full packs to tape, one pack at a time, so that packs may be composed of large LTFS extents without on-tape interleaving.

4 TLV Encoding



The tag-length-value (TLV) format is used to store many records into a pack file. Its role is to provide just enough information for an application to scan through a pack file, hop from one TLV to the next, and ensure that records are valid before attempting to decode them.

TLV uses a 32 byte header and variable-length value. A TLV reader should read the header and validate it using these steps, as illustrated in the code which follows.

1. The header "magic" is correct. The sequence should match `0x89`, `T`, `L`, `V`, `0x0d`, `0x0a`, `0x1a`, `0x0a`. This sequence identifies the TLV and allows early detection of certain types of corruption, for example end-of-line mangling if the TLV was accidentally treated as text. (The header magic is borrowed from the PNG file format. Further detail on its rationale may be found in the PNG file format specification).

2. The version at byte 24 indicates the version of the TLV format used when this TLV was created. This is currently 0. Decoding should stop if an unknown version number is seen.

3. At this point the header hash should be calculated and verified. The hash type is stored at byte 27, and header hash value is stored at bytes 30..31. Hash type 8, XXHash64, is standard. This is validated by calculating a XXHash64, keeping only the lower 16 bits, and comparing those to the stored value. Decoding should stop if the header hash code does not match.

The following code demonstrates reading and unpacking the TLV header.

[1]:

```
# Install necessary Python modules for the code in this notebook.
# Uncomment following two lines if you have import failures:
# import sys
# !{sys.executable} -m pip install xxhash msgpack zstd cryptography

# Import modules needed for sample code below
from __future__ import annotations
import base64, io, msgpack, typing, unittest, zstd
from pprint import pprint
from ulid import ULID
from struct import unpack
from typing import Optional
from xxhash import xxh64
```

[2]:

```
class TlvHeader:
    def __init__(self, f: typing.BinaryIO):
        buf = f.read(32)

        if len(buf) == 0:
            raise EOFError

        if len(buf) < 32:
            raise RuntimeError(f'TLV header too short; need 32 bytes, got␣
↪{len(buf)}')

        self.magic, self.dlen, self.dhash, self.version, \
            self.tag, self.hashtype, self.hhash = unpack("!8sQQB2sBxxH", buf)

        if self.magic != b'\x89TLV\r\n\x1a\n':
            raise 'invalid TLV header magic'

        if self.version != 0:
            raise f'unknown version {self.version}; can only handle TLV version␣
↪0'

        if self.hashtype != 8:
            raise f'invalid hash type {self.hashtype}; can only handle 8␣
↪(xxhash64)'

        if self.hhash != (xxh64(buf[0:30]).intdigest() % 2 ** 16):
            raise 'TLV header hash mismatch'

    def __repr__(self):
        return f'TlvHeader(tag {self.tag}, dlen {self.dlen})'


# Small sample TLV, base64-encoded
with io.BytesIO(base64.
 ↪b64decode("iVRMVg0KGgoAAAAAAAAADuM9tfSfjss2AEMhCAAAuxRkYXRhIGRhdGEgZGF0YQ=="))␣
 ↪as f:
    header = TlvHeader(f)
    pprint(header)
```

```
TlvHeader(tag b'C!', dlen 14)
```

Once the header has been unpacked and verified, its remaining fields may be considered trustworthy. The tag at bytes 25..26 is used to identify the data type of the value. The data length at bytes 8..15 is stored in network byte order (big-endian). Finally, The data buffer integrity should then be validated using the hash code stored at bytes 16..23. This uses the same hash type as in step 3 above. (The full 64 bits are used this time, instead of 16, as used for header validation.)

To consume the data portion of the TLV, simply read the number of data bytes indicated in the header, and validate using the data hash code from the header.

The following code illustrates a simple TLV reader which reads and validates both the header and value. This code does no further decoding on the value; that is covered in the next section.

```
[3]: class TlvSimple:
         """
         Simple form of TLV that consumes a TLV from a stream, validates its␣
      ↪integrity, but
         does not do any decoding on the value.
         """

         def __init__(self, f: typing.BinaryIO):
             """
             Creates a TLV from input stream f, leaving the stream positioned at the␣
      ↪start of the next TLV.
             :param f: any file-like stream.
             """
             self.header = TlvHeader(f)
```

```
        self.data = f.read(self.header.dlen)

        if len(self.data) < self.header.dlen:
            raise f'short data read: expected {self.header.dlen} bytes, got␣
 ↪{len(self.data)}'

        if self.header.dhash != xxh64(self.data).intdigest():
            raise "TLV data hash mismatch"

    def __repr__(self):
        return f'TLV (tag {self.header.tag}, dlen {len(self.data)})'


# Small sample TLV, base64-encoded
with io.BytesIO(base64.
 ↪b64decode("iVRMVg0KGgoAAAAAAAAADuM9tfSfjss2AEMhCAAAuxRkYXRhIGRhdGEgZGF0YQ=="))␣
 ↪as f:
    tlv = TlvSimple(f)
    print(f'{tlv} data {tlv.data}')
```

```
TLV (tag b'C!', dlen 14) data b'data data data'
```

Multiple TLVs may be stored end-to-end within a file. The example file `3simple.tlv` has three TLVs with simple strings for values.

[4]:
```
with open('sample_data/3simple.tlv', 'rb') as f:
    for i in range(3):
        tlv = TlvSimple(f)
        print(f'{tlv} data {tlv.data}')
```

```
TLV (tag b'bk', dlen 6) data b'data 1'
TLV (tag b'bk', dlen 6) data b'data 2'
TLV (tag b'bk', dlen 6) data b'data 3'
```

5 Value Encoding

TLV values are encoded using a separate, second level called *value encoding*. This is separate from TLV because:

1. TLV allows many items to be stored together and validated without decoding the values. This allows copying TLVs from one system to another, or migrating from one storage medium to another, even if the encryption keys are not accessible.

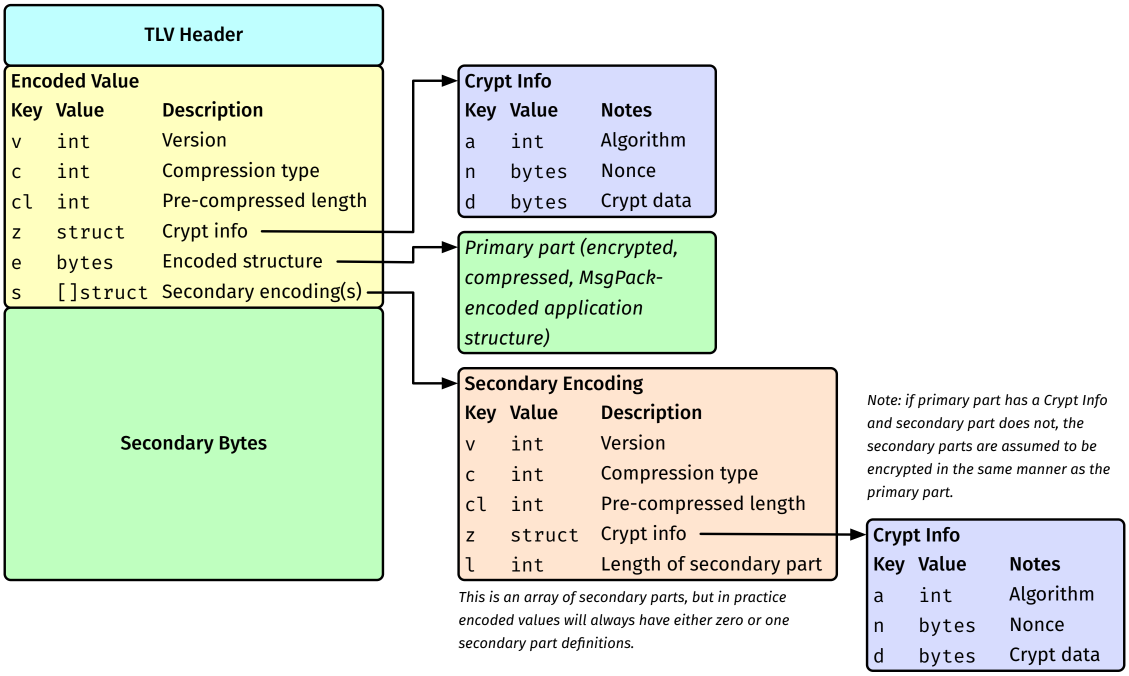
2. Applications may scan through many TLVs looking for specific types (identified by tag) or a specific instance (identified by data hash) and then decode only the TLVs it needs.

3. TLV requires a fixed-size header by design. The value encoder uses a variable-size header because the encoding may have several stages, and the parameters of each stage will be specified in the header. For example, if the value is encrypted, the header will include crypt-specific details. If not encrypted, these fields will not be present.

Value encoding provides the following features:

1. Compression, by default using the Zstandard algorithm. If the data is not compressible, then compression may be skipped.
2. Encryption with AES-256.
3. Data format versioning. If the application changes its saved data format, it will increment the version number for saved values. Decoding should inspect the version number and decode accordingly.
4. Flexibility for future features. Fields may be added to the header as necessary.

MessagePack is used to serialize both the header and the application-defined data. High-quality implementations of MessagePack are available for most programming languages. MessagePack is a fast, compact, binary encoding format.



TLV Header

Encoded Value

Key	Value	Description
v	int	Version
c	int	Compression type
cl	int	Pre-compressed length
z	struct	Crypt info
e	bytes	Encoded structure
s	[]struct	Secondary encoding(s)

Secondary Bytes

Crypt Info

Key	Value	Notes
a	int	Algorithm
n	bytes	Nonce
d	bytes	Crypt data

Primary part (encrypted, compressed, MsgPack-encoded application structure)

Secondary Encoding

Key	Value	Description
v	int	Version
c	int	Compression type
cl	int	Pre-compressed length
z	struct	Crypt info
l	int	Length of secondary part

This is an array of secondary parts, but in practice encoded values will always have either zero or one secondary part definitions.

Note: if primary part has a Crypt Info and secondary part does not, the secondary parts are assumed to be encrypted in the same manner as the primary part.

Crypt Info

Key	Value	Notes
a	int	Algorithm
n	bytes	Nonce
d	bytes	Crypt data

An encoded value has a manadatory first part describing the encoding and containing any structured data, called the primary part. It may also have an optional secondary part. The secondary part, if present, will be raw bytes which are usually compressed and/or encrypted in the same manner as the primary part.

5.1 Primary Part Decode

The following code shows how to decode a value. In a nutshell, the process is:

1. MessagePack decode the value, using the provided structure definitions below. This first pass decodes the header; the value's primary part remains encoded in the e field. In following steps, the term *primary part* will refer to bytes that are initially stored in e field and get passed through the various decoding steps below.

2. If key `z` (crypt info) is provided, then the primary part must be decrypted. Details on cipher setup and key identification follow in later sections.
3. If key `c` (compression type) is provided, then use the appropriate algorithm to decompress the primary part. Zstandard is the default algorithm.
4. The key `v` (version) is provided, it will indicate what version of structure is stored. Currently all LTFS-VOF structures are version zero so key `v` will not be present. If breaking changes are made to the format, this number will indicate which version is stored.
5. Now that the primary part's type is known (from TLV decode) and the version is known (from prior step), use MessagePack to decode the primary part into the appropriate data structure.

5.2 Secondary Part Decode

The secondary part has its own encoding parameters. Note that this field is a list to allow for multiple secondary parts. In its current form LTFS-VOF will only use one secondary part in a value.

If key `s` is provided for secondary encoding parameters, the mandatory sub-key `l` will specify the encoded length of the secondary part. For all other secondary encoding parameters, they should be assumed to match the encoding of the primary part unless explicitly overridden in the secondary encoding structure.

Whereas the primary part is always encoded with MessagePack, the secondary part will be raw bytes. In LTFS-VOF only data blocks will have a secondary part.

5.3 Encryption

Values may be encrypted. If key `z` is present the parameters needed for decryption will be provided. These will include the algorithm (generally AES-256), nonce, and a data field which is used to identify the key. The format of the data field will vary depending on the key manager in use, but it will generally provide whatever metadata is required to retreive the key needed for decryption.

5.4 Compression

Either or both parts may be compressed. If key `c` is present and non-zero, use Zstandard to decompress.

[5]:

```
class TLV:
    """
    TLV reader that reads a TLV from a stream, validates its integrity, and␣
 ↪decodes the value.
    """

    def __init__(self, f: typing.BinaryIO):
        """
        Creates a TLV from input stream f, leaving the stream positioned at the␣
 ↪start of the next TLV.
        :param f: any file-like stream.
        """
        self.header = TlvHeader(f)
        data = f.read(self.header.dlen)

        if len(data) < self.header.dlen:
            raise f'short data read: expected {self.header.dlen} bytes, got {len(data)}'

        if self.header.dhash != xxh64(data).intdigest():
            raise "TLV data hash mismatch"

        self.__decode_value(data)

    def __decode_value(self, data: bytes):
        """
        Decode a value from self.data, setting the primary part as self.value
        and the secondary part (if present) as self.secondary. self.value
        will be a dict and self.secondary will be bytes or None.
        """
        # Use streaming unpacker because extra data may be present
        unpacker = msgpack.Unpacker(io.BytesIO(data), use_list=False)
        val: dict = unpacker.unpack()

        if 'z' in val:
            # TODO: take apart z to obtain key properties, consult KMS for key, decrypt
            raise NotImplementedError('encrypted values not supported')

        # Decompress encoded value if compressed
        if val.get('c') == 1:
            val['e'] = zstd.decompress(val['e'])

        self.value: dict = msgpack.unpackb(val['e'], use_list=False)
        self.secondary: bytes = bytes(0)

        try:
            sec_enc = val['s'][0]  # Encoding specifier of secondary part
            sec_len = sec_enc['l']  # Length of secondary part
            self.secondary = data[len(data) - sec_len:]

            # If secondary encoding specifies compression, or that key is missing and primary encoding specifies
            # compression, then decompress the secondary value.
            if sec_enc.get('c', val.get('c')) == 1:
                self.secondary = zstd.decompress(self.secondary)
        except IndexError:
            pass  # no secondary part
        except KeyError:
            pass  # no secondary part

    def __repr__(self):
        return f'TLV (tag {self.header.tag})'


with open('sample_data/3values.tlv', 'rb') as f:
    for i in range(3):
        tlv = TLV(f)
        pprint(tlv.value)
        pprint(tlv.secondary)
```

```
b'value 1 header'
b'value 1 data'
b'value 2 header'
b'value 2 data'
b'value 3 header'
b'value 3 data'
```

6 Blocks and Pack Lists

The values in LTFS Versioned Object Format may be of one of three types:

1. Blocks, which store data.
2. Pack list metadata, which describe how blocks are arranged into versions.
3. Version metadata, which are described in a later section.

This section describes blocks and pack lists. As data enters a LTFS-VOF system, it is split into slices (default 10MB in size) and each slice becomes a block. A version will have one or more blocks. In addition, a pack list is created which shows which ranges of a version are mapped to which ranges of each block.

6.1 Blocks

Each stored block will have a primary part and secondary part. The primary part will only have key `I` which specifies the composite version ID this block belongs to. The secondary part will contain the raw bytes for the block.

6.1.1 Composite Version IDs

A composite Version ID is a combination of the bucket name, object name, and a unique ULID for the version. The format is:

`[26 byte version ULID]:bucketname/objectname`

The `bucketname` will be a bucket name that complies with AWS S3 bucket naming conventions. The `objectname` will be any S3 compliant object name. Note, when parsing, that an object name may contain slashes.

The following sample code demonstrates parsing a composite version ID.

[6]:

```
class VersionID:
    """
    VersionID represents a LTFS-VOF composite version identifier.
    """

    def __init__(self, bucket: str, object: str, version: str):
        self.bucket: str = bucket
        self.object: str = object
        self.version: str = version

    def __repr__(self):
        return f'VersionID({self.bucket}, {self.object}, {self.version})'

    @classmethod
    def from_str(cls, v_str: str) -> VersionID:
        """
        Parse a version string into a VersionID.
        """
        # First 26 characters is a ULID specifying the version
        version = ULID.from_str(v_str[:26])
        # Remaining characters (after a ':' separator) are bucket/object name
        bucket, object = v_str[27:].split('/', maxsplit=1)
        # Together these form the complete version identifier
        return cls(bucket=bucket, object=object, version=version)

    @classmethod
    def from_dict(cls, v_dict: dict) -> VersionID:
        """
        Convert dict form of version to VersionID.
        """
        return cls(v_dict['b'], v_dict['o'], ULID.from_str(v_dict['v']))

VersionID.from_str('7YGGZJ4YSFMYW6BQVHFKD5KKTV:bucket/object/name.txt')
```

[6]:

```
VersionID(bucket, object/name.txt, 7YGGZJ4YSFMYW6BQVHFKD5KKTV)
```

6.1.2 Block Parsing

The rest of the block parsing is straightforward:

[7]:

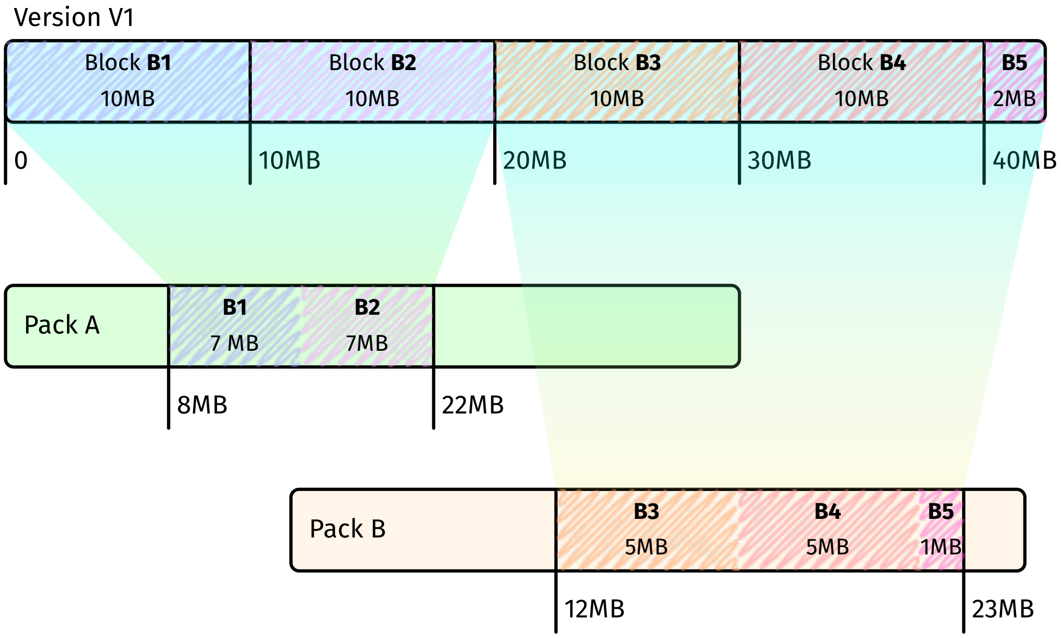
```
class Block:
    """
    Block represents a single block of object data.
    """

    def __init__(self, tlv: TLV):
        self.versionid: VersionID = VersionID.from_str(tlv.value['I'])
        self.data: bytes = tlv.secondary

    def __repr__(self):
        return f'Block({self.versionid}, {len(self.data)} bytes)'
```

6.2 Pack Lists

In the following figure, consider a version which is split into five blocks, and those blocks are stored across two packs. Version `V1` would require a pack list with two entries, the first referring to a range of `Pack A` which contains its first two blocks, the second referring to `Pack B` which contains the other three blocks.



Version is stored as 5 blocks across 2 packs. Pack list entries:

Pack	**SourceRange**		**PackRange**		**BlockLens**	**SourceLens**
	Start	**Len**	**Start**	**Len**		
`Pack A`	`0`	`20MB`	`8MB`	`14MB`	`[7MB]`	
`Pack B`	`20MB`	`22MB`	`12MB`	`13MB`	`[5MB, 5MB]`	

6.2.1 Source and Pack Ranges

Each entry contains a source range which refers to the version's data without any compression or encryption. It also contains a pack range which refers to where that data is stored, including compression and encryption. When reassembling a version from pack files, the pack range will include the TLV and value header. Thus the decode process should seek to the start of the pack range and expect to decode one or more blocks.

6.2.2 Block and Source Length Lists

Each pack list entry also contains two optional lists for block lengths and source lengths. The block lengths list specifies the stored length of each block except the last one; the length of the last block may be inferred from the pack length range minus the other stored block lengths. The source lengths list will usually be empty. If the version was created with a regular stride (e.g. 10MB in this example) then all blocks will have the same source length, except the last which may be shorter. If blocks were created on an irregular stride, then the source lengths list will contain deltas from the version's normal stride. (Again, this is usually zero.)

When reassembling whole versions, the block and source lengths lists do not need to be used. They are only required for efficient recall of partial objects.

6.2.3 Pack List Parsing

The following code shows how to handle encoded pack lists:

[8]:

```
class Range:
    """
    Range simply stores a start offset and length.
    """

    def __init__(self, r_dict: dict):
        self.start: int = r_dict.get('s', 0)
        self.len: int = r_dict.get('l', 0)

    def __repr__(self):
        return f'Range(start {self.start}, len {self.len})'


class PackEntry:
    def __init__(self, pe_dict: dict):
        self.packid: ULID = ULID.from_str(pe_dict['p'])
        self.sourcerange: Range = Range(pe_dict['o'])
        self.packrange: Range = Range(pe_dict['t'])
        self.blocklens: list[int] = pe_dict.get('E', [])
        self.sourcelens: list[int] = pe_dict.get('N', [])

    def __repr__(self):
        return f'PackEntry({self.packid}, src {self.sourcerange}, pack {self.
 ↪packrange})'


class Packs(list):
    """
    List type defined so that can easily tell the difference between a pack␣
 ↪reference and a list[PackEntry].
    """
    def __repr__(self):
        return f'Packs({super().__repr__()})'


class PackList:
    """
    PackList is a stored map of one version's list of packs storing that␣
 ↪version's data.
    """

    def __init__(self, tlv: TLV):
        self.versionid: VersionID = VersionID.from_str(tlv.value['I'])
        self.uploadid: str = tlv.value.get('U', '')
        self.packs: Packs = Packs([PackEntry(pe_dict) for pe_dict in tlv.value.
 ↪get('P', [])])

    def __repr__(self):
        return f'PackList({self.versionid}, {len(self.packs)} PackEntry)'


def data_pack_reader(f: typing.BinaryIO):
    """
    Scan a data pack file, yielding each entry as a Block or PackList tuple.
    :param f: TLV-encoded data pack file
    """
    handlers = {b'bk': Block, b'ol': PackList}

    while True:
        try:
            tlv = TLV(f)
            if tlv.header.tag not in handlers:
                raise RuntimeError(f'unknown tag {tlv.header.tag}; no handler␣
 ↪registered')

            yield handlers[tlv.header.tag](tlv)
        except EOFError:
            break


with open('sample_data/3blocks.blk', 'rb') as f:
    for entry in data_pack_reader(f):
        pprint(entry)
```

```
Block(VersionID(bucket, object, 7YF1QJW74PNYV552JB3YPAJJX1), 12 bytes)
Block(VersionID(bucket, object, 7YF1QJW74PNYV552JB3YPAJJX1), 12 bytes)
Block(VersionID(bucket, object, 7YF1QJW74PNYV552JB3YPAJJX1), 12 bytes)
PackList(VersionID(bucket, object, 7YF1QJW74PNYV552JB3YPAJJX1), 1 PackEntry)
```

7 Versions

The last top-level structure in LTFS-VOF is the version record. As mentioned earlier, a version ID is a composite of a ULID, the bucket name, and object name. The version ULIDs sort in order of their creation time with millisecond resolution. Version records will be stored in their own pack files, so the complete history of a bucket can be assembled by reading all the version packs.

When reading the version packs, each unique combination of bucket name and object name (that is, each object) will have one or more version records. They should be sorted by version ID to get the correct order of events. The following S3 event types will be represented:

1. *Put object* and *complete multipart upload* will both create version records with most of their fields filled in. The *delete* flag will be absent.
2. *Delete object* will create a version with the *delete* flag set. This is called a delete marker.
3. *Delete object* with a version ID specified will create a special *version delete* object. This is different from a delete marker, which indicates that the object has been removed. A version delete indicates deletion of a specific version only. The version delete object is required because pack files are append-only and immutable once finalized.

The TLV tag for a version record is `vr`. The tag for a version delete record is `vd`.

7.1 Version Clones & Data References

A copy of a version's data is called a clone. The version records will contain a list of clones under key `p`. The clone structure includes the pool identifier–one of which will be the tape pool–and an encoded data field. The data field encoding may take multiple forms depending on the type of pool. For a tape-based pool, MessagePack is used to encode a structure with the pack list (if small) or a reference to where the pack list is stored (if large). The included code handles both scenarios.

7.2 Embedded Version Data

If the data of a version is very small (hundreds of bytes) it will be embedded in the version record directly and no data packs or pack lists will be created. The clones list may be empty in this scenario.

7.3 Version Decoding Code

Code for parsing version records follows. In addition, the generic `ltfsvof_reader` function can be used with any stream of LTFS-VOF encoded data, both data and version packs.

[9]:

```python
class ACL:
    """
    ACL represents a single ACL entry.
    """

    def __init__(self, acl_dict: dict):
        self.idtype: int = acl_dict['t']  # 0: user, 1: group
        self.id: str = acl_dict['i']  # user/group ID
        self.permissions: int = acl_dict['p']  # 1: read, 2: write, 4: read acl, 8: write acl

    def __repr__(self):
        return f'ACL({self.idtype}, {self.id}, {self.permissions})'


class CryptData:
    """
    CryptData represents the encryption metadata for an object.
    """

    def __init__(self, cd_dict: dict):
        self.type: int = cd_dict['x']  # 0: none, 1: customer managed key, 2: S3 managed key
        self.datakey: bytes = cd_dict['k']  # encrypted data key or MD5 of customer key
        self.extra: bytes = cd_dict['e']  # extra string data

    def __repr__(self):
        return f'CryptData({self.type}, {self.datakey}, {self.extra})'


class PackReference:
    """
    PackReference represents a reference to a pack.
    """

    def __init__(self, pr_dict: dict):
        self.pack: str = pr_dict['k']
        self.packrange: Range = Range(pr_dict['r'])

    def __repr__(self):
        return f'PackReference({self.pack}, {self.packrange})'


class Clone:
    """
    Clone represents a single clone of an object.
    """

    def __init__(self, clone_dict: dict):
        data = clone_dict['l']
        try:
            # see if this is a pack list
            ref = msgpack.unpackb(data)
            if 'p' in ref:
                data = Packs([PackEntry(p) for p in ref['p']])
            elif 'R' in ref:
                data = PackReference(ref['R'])
        except msgpack.FormatError:
            # must not be msgpack, so leave data field as-is
            pass

        self.pool: str = clone_dict['p']
        self.data: Packs | PackReference | bytes = data
        self.flags: int = clone_dict.get('f', 0)
        self.blocklen: int = clone_dict['B']
        self.len: int = clone_dict['s']

    def __repr__(self):
        return f'Clone({self.pool}, {self.data}, {self.flags}, {self.blocklen},␣
 ↪{self.len})'


class Version:
    """
    Version represents a single version of an object.
    """

    def __init__(self, tlv: TLV):
        val = tlv.value
        self.versionid: VersionID = VersionID.from_dict(val)
        self.owner: str = val.get('w', '')
        self.acls: list[ACL] = [ACL(a) for a in val.get('A', [])]
        self.len: int = val.get('l', 0)
        self.etag: str = val.get('e', '')
        self.deletemarker: bool = val.get('d', False)
        self.nullversion: bool = val.get('N', False)
        self.crypt: Optional[CryptData] = CryptData(val['c']) if 'c' in val␣
 ↪else None
        self.clones: list[Clone] = [Clone(c) for c in val.get('p', [])]
        self.metadata: dict[str, str] = val.get('s', {})
        self.usermetadata: dict[str, str] = val.get('m', {})
        self.legalhold: bool = val.get('h', False)
        self.data: bytes = val.get('D', b'')

    def __repr__(self):
        return f'Version(id {self.versionid})'


class VersionDelete:
    """
    VersionDelete represents the deletion of a single verison.
    """

    def __init__(self, tlv: TLV):
        # TODO: update for version delete structure once tags are known
        self.versionid: VersionID = VersionID.from_dict(tlv.value)
        self.deleteid: VersionID = VersionID.from_str(tlv.value['???'])

    def __repr__(self):
        return f'VersionDelete(id {self.versionid}, deleteid {self.deleteid})'


def ltfsvof_reader(f: typing.BinaryIO):
    """
    Scan any LTFS-VOF file, yielding each entry as tuple of the appropriate␣
  ↪type.
    :param f: file-like stream with TLV-encoded blocks or versions
    """
    handlers = {b'bk': Block, b'ol': PackList, b'vm': Version, b'vd':␣
  ↪VersionDelete}

    while True:
        try:
            tlv = TLV(f)
            if tlv.header.tag not in handlers:
                raise RuntimeError(f'unknown tag {tlv.header.tag}; no handler␣
  ↪registered')

            yield handlers[tlv.header.tag](tlv)
        except EOFError:
            break
```

8 Appendix: Tests

This section is implements basic tests for the code in this notebook.

[10]:

```
class TlvTests(unittest.TestCase):
    def setUp(self) -> None:
        self.tlv_data = base64.
  ↪b64decode("iVRMVg0KGgoAAAAAAAAADuM9tfSfjss2AEMhCAAAuxRkYXRhIGRhdGEgZGF0YQ==")

    def test_read_tlv_header(self):
        with io.BytesIO(self.tlv_data) as f:
            header = TlvHeader(f)
            self.assertEqual(header.magic, b'\x89TLV\x0d\x0a\x1a\x0a')
            self.assertEqual(header.dlen, 14)
            self.assertEqual(header.hashtype, 8)
            self.assertEqual(header.version, 0)
            self.assertEqual(header.tag, b'C!')

    def test_read_tlv(self):
        with io.BytesIO(self.tlv_data) as f:
            tlv = TlvSimple(f)
            self.assertEqual(tlv.data, b'data data data')

    def test_3tlv(self):
        with open('sample_data/3simple.tlv', 'rb') as f:
            tlv = TlvSimple(f)
            self.assertEqual(tlv.header.tag, b'bk')
            self.assertEqual(tlv.data, b'data 1')
            tlv = TlvSimple(f)
            self.assertEqual(tlv.header.tag, b'bk')
            self.assertEqual(tlv.data, b'data 2')
            tlv = TlvSimple(f)
            self.assertEqual(tlv.header.tag, b'bk')
            self.assertEqual(tlv.data, b'data 3')
```

[11]:

```
class ValueTests(unittest.TestCase):
    def test_value_decode(self):
        with open('sample_data/3values.tlv', 'rb') as f:
            for i in range(3):
                tlv = TLV(f)
                self.assertEqual(tlv.value, bytes(f'value {i + 1} header',␣
 ↪'utf-8'))
                self.assertEqual(tlv.secondary, bytes(f'value {i + 1} data',␣
 ↪'utf-8'))

    def test_compressed_value_decode(self):
        with open('sample_data/compressed_value.tlv', 'rb') as f:
            tlv = TLV(f)
            self.assertEqual(tlv.value, b'header header header header header␣
 ↪header header header')
            self.assertEqual(tlv.secondary, b'data data data data data data␣
 ↪data data data data data')
```

[12]:

```
class BlockTests(unittest.TestCase):
    def test_read_block(self):
        blocks = []
        packlist = None

        # Sample file contains 3 simple blocks and a packlist
        with open('sample_data/3blocks.blk', 'rb') as f:
            for entry in data_pack_reader(f):
                pprint(entry)
                if isinstance(entry, Block):
                    blocks.append(entry)
                else:
                    packlist = entry

        self.assertEqual(len(blocks), 3)
        self.assertEqual(blocks[0].data, b'block 1 data')
        self.assertEqual(blocks[1].data, b'block 2 data')
        self.assertEqual(blocks[2].data, b'block 3 data')
        self.assertEqual(1, len(packlist.packs))
        self.assertEqual(0, packlist.packs[0].sourcerange.start)
        self.assertEqual(3 * len(b'block x data'), packlist.packs[0].
↪sourcerange.len)

        # Furthermore, we can read individual blocks based on the packlist
        with open('sample_data/3blocks.blk', 'rb') as f:
            packentry = packlist.packs[0]

            # Block 1 will be at the start of the pack range
            f.seek(packentry.sourcerange.start)
            block1 = Block(TLV(f))
            self.assertEqual(block1.data, b'block 1 data')

            # Block 2 will be start of pack range plus the first blocklen
            f.seek(packentry.sourcerange.start + packentry.blocklens[0])
            block2 = Block(TLV(f))
            self.assertEqual(block2.data, b'block 2 data')

            # Block 3 will be start of pack range plus the first two blocklens
            f.seek(packentry.sourcerange.start + packentry.blocklens[0] +␣
↪packentry.blocklens[1])
            block3 = Block(TLV(f))
            self.assertEqual(block3.data, b'block 3 data')
```

[13]:

```
class VersionTests(unittest.TestCase):
    def test_read_version(self):
        with open('sample_data/7YF1JH4PP45BYWK21Y7H0YHFYN.ver', 'rb') as f:
            for entry in ltfsvof_reader(f):
                # pprint(entry)

                if isinstance(entry, Version):
                    v: Version = entry
                    if isinstance(v.clones[0].data, Packs):
                        print('version has embedded packlist')
                        pprint(v.clones[0].data)
                    elif isinstance(v.clones[0].data, PackReference):
                        pr: PackReference = v.clones[0].data
                        print(f'need to load packlist from {pr.pack}')
                        with open(f'sample_data/{pr.pack}.blk', 'rb') as f2:
                            f2.seek(pr.packrange.start)
                            packlist = PackList(TLV(f2))
                            pprint(packlist.packs)
```

[14]:

```
unittest.main(argv=[''], verbosity=2, exit=False)
```

```
test_read_block (__main__.BlockTests) … ok
test_3tlv (__main__.TlvTests) … ok
test_read_tlv (__main__.TlvTests) … ok
test_read_tlv_header (__main__.TlvTests) … ok
test_compressed_value_decode (__main__.ValueTests) … ok
test_value_decode (__main__.ValueTests) … ok
test_read_version (__main__.VersionTests) … ok

----------------------------------------------------------------------
Ran 7 tests in 0.005s

OK

Block(VersionID(bucket, object, 7YF1QJW74PNYV552JB3YPAJJX1), 12 bytes)
Block(VersionID(bucket, object, 7YF1QJW74PNYV552JB3YPAJJX1), 12 bytes)
Block(VersionID(bucket, object, 7YF1QJW74PNYV552JB3YPAJJX1), 12 bytes)
PackList(VersionID(bucket, object, 7YF1QJW74PNYV552JB3YPAJJX1), 1 PackEntry)
version has embedded packlist
Packs([PackEntry(7YF1JH4PP45BYWK21Y7H4QPHAT, src Range(start 0, len 36), pack
Range(start 0, len 303))])
need to load packlist from 7YF1JH4PP45BYWK21Y7H4QPHAT
Packs([PackEntry(7YF1JH4PP45BYWK21Y7H4QPHAT, src Range(start 0, len 36), pack
Range(start 0, len 303))])
```

[14]:

```
<unittest.main.TestProgram at 0x103b8ae60>
```

[ ]: