Workgroup: Interpeer Project
Internet-Draft: draft-jfinkhaeuser-vessel-container-format-00
Published: 28 July 2023
Intended Status: Informational
Expires: 29 January 2024
Author: J. Finkhaeuser
Interpeer

Vessel Container Format

Abstract

This document describes a generic container file format suitable for authoring content collaboratively, both in real-time or with eventual merging. It is suitable for encapsulating an authoritative view of the resource, or managing multiple diverging versions. Confidentiality and authentication of content are both supported.

This document is currently not, in this form, submitted as an Internet-Draft. Any statements below that suggest this and assign copyright to the IETF are automatically added boilerplate and should be ignored. This notice will be removed if submission occurred.

The RFC Editor will remove this note

About This Document

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

The latest revision of this draft can be found at https://specs.interpeer.io/. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-jfinkhaeuser-vessel-container-format/.

Discussion of this document takes place on the interpeer Working Group mailing list (mailto:interpeer@lists.interpeer.io), which is archived at https://lists.interpeer.io/pipermail/interpeer/. Subscribe at https://lists.interpeer.io/mailman/listinfo/interpeer. Working Group information can be found at https://interpeer.io/.

Source for this draft and an issue tracker can be found at https://codeberg.org/interpeer/specs.

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 29 January 2024.

Copyright Notice

Copyright (c) 2023 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document.

This document may not be modified, and derivative works of it may not be created, except to format it for publication as an RFC or to translate it into languages other than English.

Table of Contents

1. Introduction

Human rights considerations increasingly and rightly influence technology design decisions. IRTF has issued [RFC8280] on this matter, with ongoing research intending to update this document with [I-D.draft-irtf-hrpc-guidelines-13].

This document is concerned with data storage and transfer under human rights considerations. To this end, it specifies a container resource format and associated operations for encapsulating arbitrary data in such a way that human rights are protected.

The above may require unpacking. There are two underlying assumptions to this:

1. Internet technologies exist to transfer data, and data transfer usually (but not necessarily) implies data storage.
2. Some human rights considerations in the digital sphere involve security concerns; when applied to data storage, these are predominantly data privacy concerns.

This suggests that some human rights concerns are best addressed at the data storage layer, prior to transfer. Note well that while this may lower the burden on data transfer, it does not imply that data transfer can be oblivious to such concerns.

The above view takes into account the position expressed in [RFC3365] that security **MUST** be implemented, even if end users decide to disable it. Furthermore, it focuses on the "content exfiltration" attack class described in [RFC7624].

Several solutions exist for cryptographically securing data at rest. But as [RFC8280] correctly points out, sometimes cryptographic solutions make assumptions that negatively impact other human rights. For example, strong authentication may impact the rights imparted by anonymous or pseudonymous operations. Care must be taken to balance such concerns.

The Vessel container format aims to provide a resource format and associated operations for securing arbitrary data in such a way that as many human rights concerns as possible are taken into account, which in practice means providing confidentiality through encryption ([UNHRC51] declares that "Encryption is a key enabler of privacy and security online and is essential for safeguarding rights, including the rights to freedom of opinion and expression, freedom of association and peaceful assembly, security, health and non- discrimination").

The aim of this format is to enable real-time and eventually consistent applications where content is created by multiple authors.

Encodings similar to Vessel already exist and are in use by applications and protocols such as GNUNet, BitTorrent, Freenet [FREENET], Gnutella, ERIS [ERIS] and others. However, none of them specifically target human rights considerations or fall short in addressing them, or are tied to their respective protocols and applications. Vessel defines an encoding independent of any specific protocol or application and decouples storage from transport (protocol considerations, however, are addressed in Section 4.2). Vessel may be seen as a modest step towards Information-Centric Networking [RFC7927].

1.1. Objectives

The objectives in the design of Vessel are:

Human Rights Preservation: As outlined above, this is the main objective; for an analysis on how this is achieved, see Section 4.1.

Availability: Content encoded with Vessel can be easily replicated and cached to increase its availability.

Middle-box Deniability: Intermediary peers, who transport or cache (store) Vessel extents and do not have access to content encryption keys cannot access the plain text content.

Deterministic Identifiers: Identifiers are always deterministic. In particular, this aids multi-authoring.

Multi-Authoring: Multiple authors should not interfere with each other or require coordination when adding to the same Vessel resource. Note that applications may need to implement additional precautions when implementing multi-authoring, but Vessel **MUST NOT** prevent this.

Storage Efficiency: Vessel can be used to encode small content as well as large content with reasonable storage overhead. Care is additionally taken to align Vessel resources well with file system blocks, as far as that can be predicted.

Simplicity: The encoding should be as simple as possible in order to allow correct implementation on various platforms and in various languages.

Completeness: Given a sequence of extents, it is possible to determine whether this sequence is complete (up to its latest entry).

Sequence Correctness: Given a set of extents, it is possible to determine the order in which they must be put into sequence.

Verifiability: We refer to the combination of completeness, sequence correctness, integrity and authenticity as verifiability; that is, an extent is verifiable if the above criteria can be ensured.

Real-Time Capabilities: Extents must be sized such that real-time applications such as for e.g. video streaming are feasible.

1.2. Scope

Vessel describes how arbitrary data can be encoded into a sequence of extents, in such a way that the sequence is deterministic even as multiple authors contribute to it in parallel. Each extent's authenticity can be proven, and its contents may remain confidential. Finally, content can also be deleted.

Vessel does not prescribe how extents should be stored or transported over a network. Some considerations on how transport protocols may be used are provided in Section 4.2.

The main aim is to preserve human rights in the storage and transport of data. This is partially reflected in the design of Vessel itself, but some considerations must be outside the scope of this specification. Section 4.1 provides an analysis, and Section 4.4 and Section 4.3 expand upon this with regards to privacy and security related concerns.

Vessel is an attempt to find a minimal common basis upon which higher functionality can be built.

Multi-authoring is supported by Vessel, but the specifics of distributing authorization tokens for this are out of scope, and need to be developed in a future specification.

Vessel is not intrinsically tied to the use of any specific cryptographic algorithms, but provides a simple means for specifying them. This specification defines a set of default algorithms that any implementation **MUST** provide, however.

1.3. Previous Work

Vessel draws on the general idea of chunking content into extents as practiced by many peer-to-peer protocols; however, the actual design has no specific basis.

In the course of designing this, some alternative approaches such as [ERIS] and [DMC] were analysed. Neither, however, adequately address content deletion for the purposes of human rights preservation. ERIS does not address deletion whatsoever; content blocks are immutable, as addressing occurs based on content hashes. DMC builds upon ERIS, and provides means by which blocks may be included in or excluded from the container index. Blocks excluded in this manner should be garbage collected by well-behaving nodes, but there is not much preventing a node from misbehaving. See Section 4.4 for some more details on this.

2. Conventions and Definitions

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.

In order to respect inclusive language guidelines from [NIST.IR.8366] and [I-D.draft-knodel-terminology-10], this document uses plural pronouns.

2.1. Terminology

Sector: Physical addressing of block storage devices often involves the use of so-called sectors ; a sector has a unique offset (within its enclosing scope) and addresses a sequence of octets.

Block: On the file system layer, sectors are typically mapped into blocks . That is, each logical block in a file system encompasses one or more physical sectors on the block storage device. Similar to sectors, blocks are addressed with an offset from the start of the file system's storage area.

Extent: An extent is a self-contained section of a Vessel resource; a single extent constitutes a fully-formed Vessel resource, but a resource may contain multiple extents.

Resource: In Vessel encoding, a resource consists of one or more extents.

Origin Extent: The origin extent is the first extent authored for a new resource.

Envelope: The leading and trailing metadata in an extent together form its envelope .

Payload: The extent content, distinct from its envelope, is termed the payload .

Section: Extent payloads are subdivided into sections , each of which may contain data for distinct application concerns.

(Sub-)Stream: A stream is defined as the sequence of sections, derived from the sequence of extents, that all pertain to the same application concern.

Key: A cryptographic key , permitting operations such as signing, verifying, encrypting or decrypting content.

Key Pair: A public and a private key used in asymmetric cryptography. The public key can be used to verify and encrypt content, the private to sign and decrypt content. Possession of the full key pair is equivalent of being in possession of an authoring identity.

Ownership (Key Pair): The key pair that is used to author the origin extent. This ownership key pair has the special property that it is authoritative with regards to authorizing other key pairs.

Authoring (Key Pair): The key pair(s) used to author extents following the origin extent.

2.1.1. Pseudo-Code Conventions

Character and String Literals:

We quote single characters in single quotes (') and character strings in double quotes ("). Characters and strings may be encoded in different encodings; this is noted in the surrounding text. Typically, we use ASCII for text used within the specification itself, and UTF-8 [RFC3629] for application provided text.

Concatenation: We use the pipe character (|) to denote concatenation.

Function Calls: Function calls are represented as the majority of programming languages implement them, with a function name, and parameters enclosed by an opening and closing bracket pair `()`. Parameters are delimited by commas (,), e.g. `foo(bar, baz)`.

Truncation: When we discuss truncating octet sequences in this document, we assume a pseudo-code function `truncate(input, size)` exists that returns the first `size` octets from the `input` octet sequence.

Size: Pseudo-code furthermore assumes the existence of a `length()` function that returns the length of a given field, or octet sequence, etc.

Names: Similar to the size function, a `name()` function is assumed that returns canonical names for algorithm choices.

3. Specification of Vessel

Vessel defines a container format for storing arbitrary data such that it may be encrypted at rest, but also transmitted in an efficient manner. Its properties are designed to help protect human rights concerns; see Section 4.1 for details.

To this end, Vessel chunks data into fixed-sized extents, each of which contain some meta information as well as content sections. Each section is scoped to a particular purpose. This permits interleaving sections dedicated to different concerns; in this manner, applications may leverage the facilities provided by Vessel to mix processing metadata with application data, or multiple related data types, such as audio and video. All sections relating to a single concern are called a stream.

The Vessel specification is somewhat generic with regards to the choice of cryptographic algorithms. We therefore provide version fields in the envelope for implementations to gracefully adjust to the specific implementation used in a resource.

3.1. Algorithms

The cryptographic algorithms used in this specification may change, but their use does not. Throughout the document, we will therefore use generic names, and only provide the choices implementations **MUST** provide and their defaults afterwards, in Section 3.8.

The following algorithms are required:

Version Tag Hash: A version tag hash function is required for generating version tags.

Extent Identifier Hash: Similarly, a extent identifier hash function is needed for creating extent identifiers.

Author Identifier Hash: Depending on the choice of authoring key pair, an author identifier hash function for effectively truncating longer keys into a fixed size may be necessary.

Nonce Hash: In order to provide fixed length nonces for other cryptographic operations, a nonce hash function is required.

Public/Private Key Pair: A public/private key pair enables asymmetric cryptographic operations. Such key pairs typically enable different algorithms, such as for signatures or key exchanges.

Asymmetric Signature Algorithm: Asymmetric signature algorithms are the main use of key pairs in this specification, though key exchange algorithms are likely required for additional sections not covered explicitly here.

Key Exchange Algorithm: Key pairs also enable two or more parties to securely exchange key material for symmetric encryption. This specification does not use key exchange algorithms, but needs to consider their use.

Symmetric Encryption: Encrypting extent payloads for confidentiality requires a symmetric block cipher algorithm .

Message Authentication Code: We may require a message authentication code function (MAC) to authenticate extent content.

Signature Algorithm: The extent signature algorithm may be a message authentication code based scheme, or a public key signature based scheme. If the former, we describe a MAC algorithm here. If the latter, it is identical to the asymmetric signature algorithm choice above.

Signature Hash: Some signature algorithms require the use of a specific hash function, while others permit some selection here. Vessel defines this as the signature hash function

Private Header: Some uses of this container format may make it desirable to hide some header information from in-path nodes. Such uses should set a private header flag.

Note that it is perfectly legitimate to use e.g. the same hash function for all of the different types of hash functions above.

Many of these algorithms are used in the calculation of a version tag (Section 3.2). This document refers to them in pseudocode as symbols.

Algorithm Name	Pseudocode Symbol
Version tag hash	version_tag_hash

Algorithm Name	Pseudocode Symbol
Extent identifier hash	extent_identifier_hash
Author identifier hash	author_identifier_hash
Nonce hash	nonce_hash
Signature Hash	signature_hash
Asymmetric Signature Algorithm	asymmetric_signature_algorithm
Message authentication code	message_authentication
Symmetric Encryption	symmetric_encryption
Signature Algorithm	signature_algorithm

Table 1: Pseudocode Symbols

3.2. Versioning of Extent Metadata

Vessel distinguishes between two versions, both of which **MUST** be present in the envelope.

1. The Envelope Version is a numeric value which describes the layout of the extent envelope. In this version, it is defined to be a NULL octet (all bits zero). Future specification versions **MUST** update this value if and only if the envelope layout changes.
2. The Version Tag is a octet sequence that describes both the remainder of the extent layout, as well as the cryptographic algorithms used.

3.2.1. Version Tag Algorithm

The algorithm for generating the version tag is relatively simple: concatenate all the choices of algorithms by their canonical identifier, separated by a semi-colon ; (U+003B) and apply the version tag hash function. Then concatenate the string "Vessel" with the output of the prior function invocation, and apply the version tag hash function again. Of the result, use as many octets from the start as the version tag requires.

The canonical identifiers are listed in Section 3.8. They are all ASCII text, such that no other character set considerations are necessary.

```
algorithms = name(version_tag_hash) |
  ';' | name(extent_identifier_hash) |
  ';' | name(author_identifier_hash) |
  ';' | name(nonce_hash) |
  ';' | name(signature_hash) |
  ';' | name(asymmetric_signature) |
  ';' | name(message_authentication) |
  ';' | name(symmetric_encryption) |
  ';' | name(signature_algorithm) |
  ';' | private_header_flag

pre_hash = version_tag_hash(algorithms)

hash = version_tag_hash("Vessel" | pre_hash)

version_tag = truncate(hash, length(version_tag))
```

Figure 1: Pseudocode for Version Tag

Given the relatively low number of possible inputs to this scheme, even a truncated hash has a high likelihood of remaining collision free. At the same time, the algorithm is simple enough to be implemented on many platforms. It also is strictly deterministic. The scheme is also sufficiently future proof. Addition of more algorithms is well supported by simply appending them to the concatenation list.

Note that implementations **MUST** treat all strings and individual characters in this algorithm as ASCII encoded.

- The `signature_algorithm` is a reference to one of the other algorithms only. Its possible values have special meaning, as described in Section 3.8.6.
- Conversely, the `private_header_flag` may only be one of two special words, `ph=0` or `ph=1` respectively, where `ph` stands for "private header", and a value of zero (`0`) indicates that private headers are not used, while a value of one (`1`) shows that they are.

3.2.1.1. Version Tag Example

An example version tag may help illustrate the above algorithm. Below, the hash sign (#) is used to indicate a code comment that the pseudo code would ignore.

```
algorithms = "sha3-512" | # name(version_tag_hash)
  ';' "sha3-512" |        # name(extent_identifier_hash)
  ';' "none" |            # name(author_identifier_hash)
  ';' "sha3-512" |        # name(nonce_hash)
  ';' "eddsa" |           # name(signature_hash)
  ';' "ed25519" |         # name(asymmetric_signature)
  ';' "kmac128" |         # name(message_authentication)
  ';' "chacha20" |        # name(symmetric_encryption)
  ';' "aead" |            # name(signature_algorithm)
  ';' "ph=1"              # private_header_flag
 # Results in:
 #
"sha3-512;sha3-512;none;sha3-512;eddsa;ed25519;kmac128;chacha20;ae
ad;ph=1"

pre_hash = version_tag_hash(algorithms)

hash = version_tag_hash("Vessel" | pre_hash)

version_tag = truncate(hash, length(version_tag))
 # Results in (hexadecimal): f8059ab6
```

Figure 2: Pseudocode Example for Version Tag

3.2.1.2. Version Tag Permutations

The number of permutations for all the supported combinations of functions in this document is not too large for a lookup table, but may be too large for the particular implementation's target platform. This is why only a subset of these algorithms are required, with the rest remaining optional.

The number of permutations resulting from only implementing the required algorithms is significantly lower, but is still large enough that lookup of the required functions may pose some overhead.

Being a container format and not a protocol, Vessel has no means for negotiating algorithms between the extent authoring and the extent consuming parties. The choice of algorithms therefore must be encoded into the extent by the author.

At the same time, leaving variable space in an envelope for algorithm specification complicates the overall specification, and makes parsing more brittle.

Using a truncated hash for the version tag is the right choice here, but excluding e.g. memory limited platforms is not desirable.

For this reason, this specification prvoides for three different compliance profiles:

Extended Compliance: Applications provide extended compliance if they implement all combinations of algorithms in this specification.

Full Compliance: Applications provide full compliance if they implement only the mandatory algorithms, i.e. those denoted by the key word **MUST**.

Limited Compliance: Applications provide limited compliance if they implement only a subset of the mandatory algorithms.

Limited compliance applications strictly speaking violate this specification, according to the use of key words in [RFC2119]. This specification disagrees, but only with respect to the supported algorithm choices. Here, the key words indicate what is required for full or extended compliance. The remainder of the document uses the key words according to [RFC2119] (and [RFC8174]).

Applications **SHOULD** aim for full compliance, but **MAY** provide only limited compliance if platform constraints or intended usage dicate such a choice.

Additionally, applications **SHOULD** produce only a strictly limited number of distinct version tags. The larger choice of algorithms is aimed more at fostering compatibility when reading extents.

Extended compliance **SHOULD NOT** be an aim. The version tag is intended for the purposes of having a compact representation of algorithm choices. Each application is likely to have need only of a strictly limited number of permutations. Requesting full compliance in this document is intended to foster some interoperability. But as a naive lookup table for mapping version tags to individual algorithm identifiers may consume several hundred KiB for full compliance already, it is clear enough that extended compliance will be out of reach for many applications.

3.3. Extents

Vessel resources are subdivided into extents . An extent consists of an envelope, a header, a footer, and a payload. The envelope's layout is defined by the envelope version, the remainder by the version tag.

A single extent is a complete Vessel resource. Resources may contain multiple extents. The order in which extents occur within a resource has no bearing on the logical order of extents, which is fully deterministic. Of course, implementations **SHOULD** write extents into a resource file in logical order, but **MAY** choose different ordering e.g. to optimize access or storage.

Implementations **MUST NOT** rely on resource hashing for resource identity. Instead, the origin extent's identifier **SHOULD** be used for the identify of the entire resource. Implementations **MAY** instead opt to treat resource identity separately, only referring to the origin extent in a resolution step.

Extents should align well with block boundaries provided by storage systems, as well as page sizes by operating system's memory allocation facilities. At the time of writing, the consensus here appears to be that blocks and pages of 4096 octets are in wide-spread use across different systems. At the same time, this is a small enough size that even embedded devices should be able to manage this amount of data at a time. Therefore, extents **MUST** be sized in multiples of 4096 octets.

3.3.1. Envelope

The envelope layout is as follows. Note that in the diagram below we list specific bit sizes for fields; these are not open to interpretation and **MUST** be implemented exactly as specified.

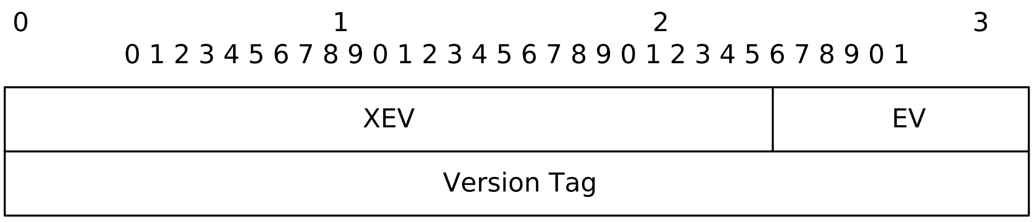

Figure 3: Envelope Diagram

1. A "magic" three octet sequence consisting of ASCII values 88 ('X'), 69 ('E') and 86 ('V'), together forming the string "XEV". Read backwards, "XEV" is "VEX", which may be considered shorthand for "Vessel EXtent".
2. A single octet containing the envelope version (see Section 3.2).
3. A four octet sequence with the version tag (see Section 3.2).

A fixed-sized envelope containing all relevant versioning information is sufficient for graceful adaptation to future specification changes. Furthermore, the envelope is long enough to serve as a synchronization boundary for detecting extents within a octet stream.

Note again that the envelope version relates only to the layout of this envelope. The first four octets **MUST** remain fixed in size and meaning (although it is possible to change the "XEV" string into a different one of the same size), otherwise the envelope version cannot be reliably read.

The envelope version exists primarily to permit graceful handling of future changes to the version tag, or addition of envelope metadata.

3.3.2. Header

Following the envelope information, the extent header provides metadata about the extent itself. The exact layout of the header depends on the version tag.

The primary reason for this is that the choice of different cryptographic algorithms implies different field lengths e.g. for identifiers specified in the header. Implementations **MUST** adjust to this, for all combinations of such algorithms they support. However, implementations **MUST NOT** deviate from the sequence of fields below.

Note that for this reason, the display sizes of 48 bits for extent and author identifiers given in this document should be considered examples only.

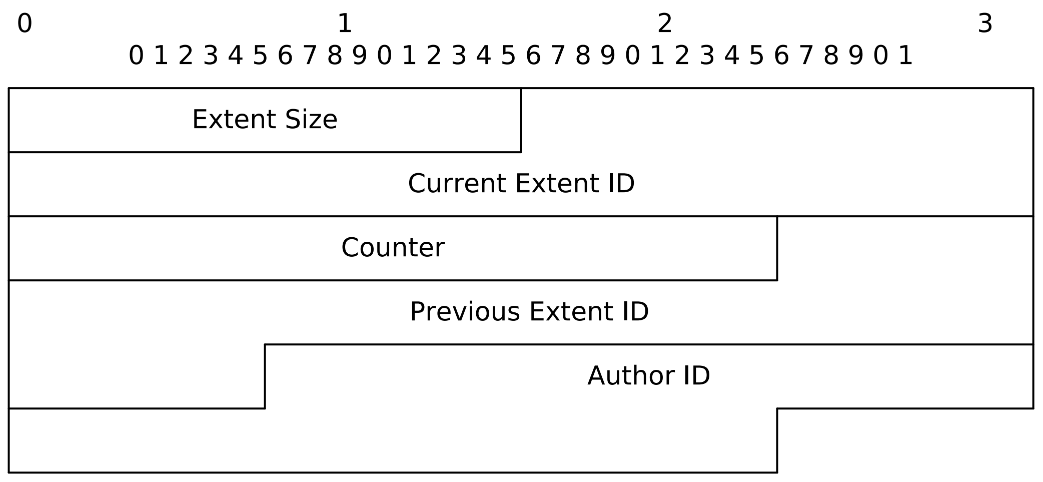

Figure 4: Extent Header Diagram

1. The extent size. This is a 16 bit unsigned integer value in big endian encoding. It specifies the size of the extent from the beginning of the envelope to the end of the footer. The value **MUST** be interpreted as a multiplier of a nominal block size of 4096 octets, yielding possible extent sizes from 4096 octets to 256 MiB.
2. The current extent identifier follows; see Section 3.5 for details.
3. An authoring counter follows. This is a 24 bit unsigned integer value in big endian encoding. Authors must increment this counter for every update to any extent they make; see Section 3.6.2 for details.
4. The next field specifies the previous extent's identifier. If the extent is the origin extent, its value **MUST** be a sequence of NULL octets.
5. Following this, is the author identifier. The identifier **SHOULD** be mappable to an authoring key pair, to permit authentication of the extent payload. It **MUST** be mappable to an authoring key pair if an asymmetric signature algorithm is chosen to validate the extent payload.

3.3.3. Payload

octets following the header, up to the beginning of the footer, are extent payload.

The payload contains the application data. Though the extent size is fixed, applications may provide arbitrary length data for encapsulation. For this reason, application data is encapsulated in sections within the payload; see Section 3.4 for details.

Different sections may have different length, but sections either include their own length specifier, or the version tag information provides an indication of the section length. Thus the payload consists of zero or more sections, each of which has a determinate length.

If the total length of all sections is less than the size of the payload area (the size of the extent minus the size of envelope, header and footer), then the remainder of the payload area **MUST** be filled with padding octets.

3.3.3.1. Padding

For padding, we use a variation of the PKCS#7 padding scheme defined in [RFC5652].

PKCS#7 is defined for block ciphers that typically encrypt blocks of no more than 256 octets at a time. The padding algorithm therefore fills padding octets with a value equal to the padding length. This also helps determine which parts of a decrypted plain text are padding, and which are payload.

Due to subdividing payload into determinate length section, we do not need to distinguish between padding and plain text - the section headers provide that distinction for us. At the same time, a large extent that contains only a small section may contain padding that is significantly longer than 256 octets.

We define the padding size as follows:

```
payload_size = extent_size
   - envelope_size
   - header_size
   - footer_size

padding_size = payload_size - cumulative_section_sizes
```

Figure 5: Pseudocode for Padding Size Calculation

With this definition, the padding value is:

```
padding_value = padding_size modulo 256
```

Figure 6: Pseudocode for Padding Value Calculation

Finally and unlike PKCS#7, it is possible for extent payloads to contain no padding at all (zero length).

3.3.4. Footer

The footer contains only a signature, which verifies the entire extent. The size of the signature is dependent on the signature algorithm scheme which is defined by the version tag, and provided as 64 bits here for display purposes.

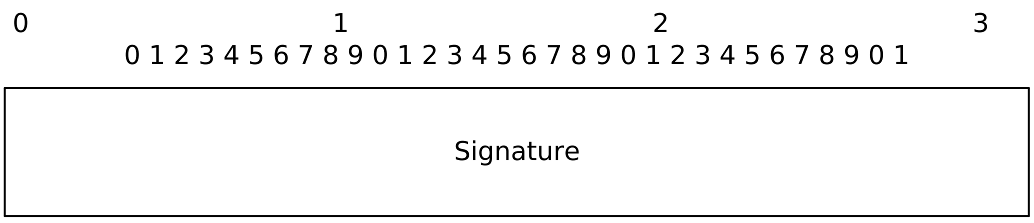

Figure 7: Extent Footer Diagram

1. The signature field. The range of data covered by the signature starts at the beginning of the envelope, and ends at the end of the payload.

Signatures are calculated *after* potential encryption of the payload.

3.3.5. Full Extent Layout

The full layout of an Extent is as follows; as with the identifier sizes, the payload size below is limited to 200 bits for display purposes only.

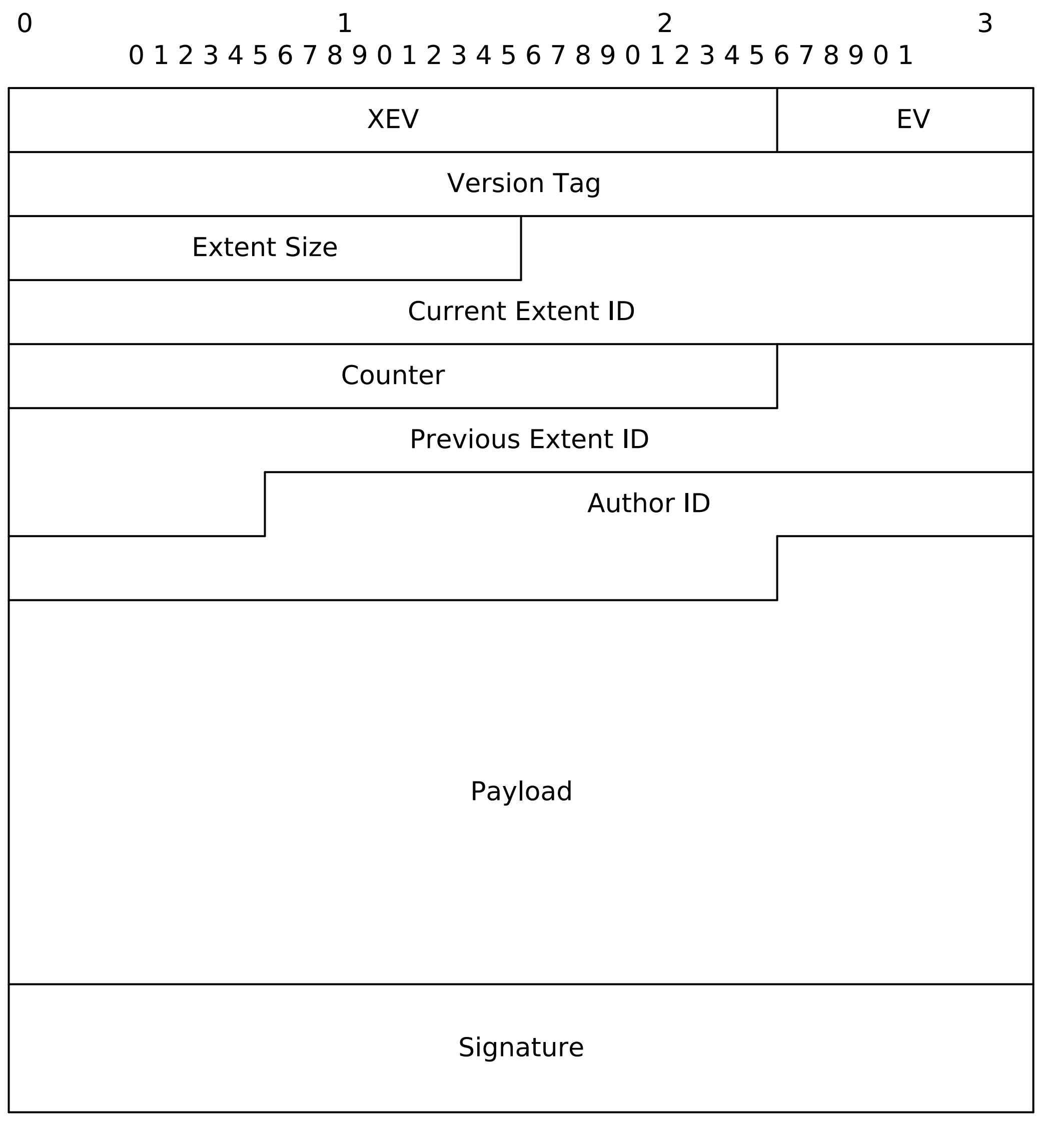

Figure 8: Combined Extent Diagram

3.4. Sections

Sections encode application data as well as other metadata interleaved with whatever the application requires. In order to help distinguish one kind of section from the other, all sections carry a type identifier.

Additionally, it is potentially necessary to logically group sections of different types. For example, a section type carrying audio data may be required for encoding a movie, but such a movie may be available in multiple languages. It should therefore be possible to organize related section into a single stream . For this purpose, sections carry a topic field, which identifies the stream the section relates to.

3.4.1. Fixed-Sized Sections

Some section types imply a fixed section size. For these sections, it is not necessary to carry additional size information, and the layout is as follows.

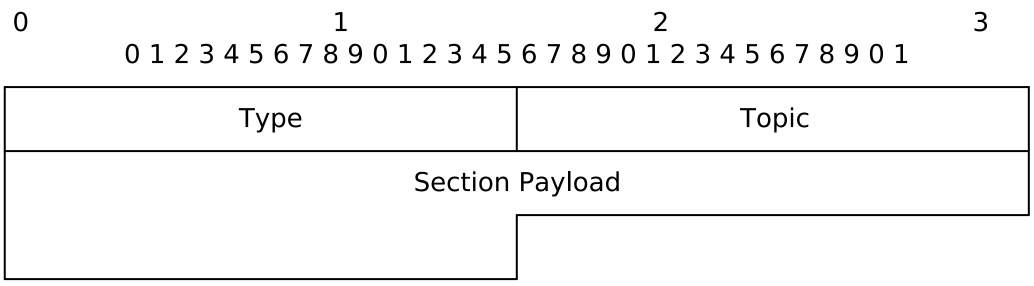

Figure 9: Fixed-Sized Section Header Diagram

3.4.2. Variable-Sized Sections

Other sections are variable sized; this is almost certainly the case for opaque application data.

Sections can never be larger than the extent within which they are encapsulated. However, sections need not be a multiple of 4096 octets in size as extents are.

It would be possible to encode the section size in 28 bits, but such an encoding is hard. Instead, we use a simple variable sized encoding scheme as described in Section 3.4.3.1.

To maintain compatibility with the fixed-size section header, the section size follows the topic.

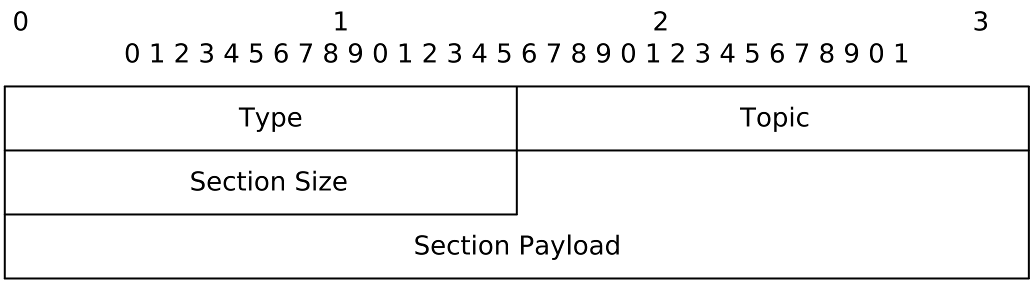

Figure 10: Variable-Sized Section Header Diagram

3.4.3. Section Fields

1. The section type is a 16 bit unsigned integer in big endian encoding. Types below 1024 are reserved; applications must specify the types they intend to use. In order for applications to determine compatibility with any such scheme, a special content type section is used (Section 3.4.5.4).

2. The topic is also a 16 bit unsigned integer in big endian encoding. Topics below 1024 are reserved. The same content type section as above is used to determine application compatibility.
3. If present, the section size is either 16 or 32 bits in length. The value of the first 16 bits determines if more bits follow (Section 3.4.3.1).
4. The remainder of the section is section payload.

Note that the section size determines the size of the entirety of the section, not the section payload size. This is largely to stay more consistent with the dealing with fixed sized sections; in either case, reading the section header gives you all the information to jump from the start of the section to its end.

3.4.3.1. Variable-Sized Length Encoding

The variable sized length encoding for section sizes is a simple, space efficient scheme. A section size consists of one or two words, each of which is a 16 bit, unsigned integer in big endian encoding. The algorithm for determining the section size is as follows:

1. Read the first word. If the most significant bit is unset, this is the entire section size and the algorithm terminates.
2. If the most significant bit of the first word is set,
    1. unset it.
    2. Shift the word into the most significant word of a 32 bit unsigned integer variable.
    3. Read the next word, and assign it to the least significant word of the 32 bit unsigned integer variable. The result is the section size.

Note that this scheme yields up to 31 bit sizes; any size larger than the extent payload are **MUST** be rejected. Since it is difficult to continue parsing an extent payload when the size of a section cannot be determined, implementations **SHOULD** reject the entire extent as malformed.

3.4.4. Topics

As described in Section 3.4.3, section headers carry a 16 bit topic value. Topic values below 1024 are reserved for use in this specification or later revisions or extensions; the remainder are application choices.

Topic	Decimal	Hexadecimal	Reference
Authentication, Authorization and Accounting	0	0x0000	Section 3.4.4.1
Metadata	10	0x00A0	Section 3.4.4.2

Table 2: Section Topics

3.4.4.1. Topic: Authentication, Authorization and Accounting

Decimal Topic	Hexadecimal Topic
0	0x0000

Table 3: Authentication Topic

The topic value for authentication, authorization and accounting (AAA) is reserved; an appropriate scheme of sections for use in this topic is out of scope for this specification.

3.4.4.2. Topic: Metadata

Decimal Topic	Hexadecimal Topic
10	0x00A0

Table 4: Metadata Topic

A topic value of ten (0x00A0) has a special meaning; it implies that the section carries extent metadata; for this reason, the remainder of this document refers to it as the metadata topic . Some sections only make sense relating the the extent as a whole; this topic is reserved for those sections.

Note that if applications wish to provide sections with extent-wide meaning, they **MUST NOT** re-use this metadata topic. Applications **MAY** define their own extent-wide topic, if they so wish.

Applications may provide any kind of metadata in this topic, such as from e.g. Dublin Core terms [DUBLIN-CORE].

3.4.5. Predefined Sections

The following describes sections defined in this specification.

Section	Decimal Type	Hexadecimal Type	Valid Topics	Reference
CRC32	1	0x0001	Metadata Section 3.4.4.2	Section 3.4.5.1
MAC	2	0x0002	Metadata Section 3.4.4.2	Section 3.4.5.2
Signature	3	0x0003	Metadata Section 3.4.4.2	Section 3.4.5.3
Content Type	10	0x00A0	any	Section 3.4.5.4

Section	Decimal Type	Hexadecimal Type	Valid Topics	Reference
BLOB	11	0x00A1	any	Section 3.4.5.5

Table 5: Predefined Sections

Sections may be valid only in combination with a specific topic. If that is the case, such valid topics are listed here or in the respective section descriptions. If no such restriction is listed, the section may appear in any topic.

Section values below 1024 are reserved for use in this specification or later revisions or extensions; the remainder are application choices.

If a section appears in an invalid topic, implementations **MUST** ignore it. They **SHOULD** emit warnings about malformed extents as far as they have provisions for doing so, and **MAY** reject the entire extent.

Note in particular that there is no generic "data" section defined here. This is done purposefully, such that applications **MUST** define at least one section type.

3.4.5.1. Section: CRC32

Decimal Type	Hexadecimal Type	Valid Topics	Size (octets)
1	0x0001	Metadata Section 3.4.4.2	2 + 2 + 4 = 8

Table 6: CRC32 Section

This section is only valid at the beginning of an extent payload. This is mostly to do with how such checksums are calculated. For the sake of keeping implementations simpler, implementations **MUST** ignore CRC32 sections if they are found anywhere in an extent payload other than at the start (offset zero).

The purpose of this of this section is twofold:

1. When used in an unencrypted extent, it provides simple integrity checking of the extent.
2. When used in an encrypted extent (see Section 3.7), then its presence at offset zero provides for a convenient verification mechanism that decryption succeeded.

The CRC32 section contains a checksum over the payload *following* this section, to the end of the payload (padding included).

This section is mutually exclusive with the MAC and signature sections.

This is a fixed sized section, so uses 2 + 2 octets for the section header. The payload is the 32 bit (= 4 octet) CRC32 sum.

3.4.5.2. Section: Message Authentication Code (MAC)

Decimal Type	Hexadecimal Type	Valid Topics	Size (octets)
2	0x0002	Metadata Section 3.4.4.2	2 + 2 + K

Table 7: MAC Section

A counterpart to the CRC32 section based on message authentication codes, this section calculates a MAC over the payload *following* itself, up to the end of the payload (padding included). Implementations **MUST** ignore it if they encounter it in any place other than at the start of the payload (offset zero).

The algorithm used in providing this MAC is specified in the `message_authentication` algorithm selection.

This section is mutually exclusive with the CRC32 and signature sections.

This is a fixed sized section, though the size depends on the `message_authentication` algorithm selected in the version tag. If the algorithm produces K octets of MAC, the section is 2 + 2 + K octets in size.

3.4.5.3. Section: Signature

Decimal Type	Hexadecimal Type	Valid Topics	Size (octets)
3	0x0003	Metadata Section 3.4.4.2	2 + 2 + K

Table 8: Signature Section

The public key cryptography counterpart to the MAC section is the signature section. It, too, is only valid at the beginning of the payload. Implementations also **MUST** ignore it if they are found other than at the start (offset zero).

The purpose of this section is the same as for the CRC32 section, except it aids in sign-encrypt-sign schemes (see Section 3.7).

The section contains a cryptographic signature from the authoring key pair, calculated over the payload *following* this section, to the end of the payload (padding included).

The algorithm used in providing this signature is specified in the `asymmetric_signature` and `signature_hash` algorithm selections.

This is a fixed sized section, though the size depends on the algorithm selection above. If the algorithms produce K octets of signature, the section is 2 + 2 + K octets in size.

This section is mutually exclusive with the MAC and CRC32 sections.

3.4.5.4. Section: Content-Type

Decimal Type	Hexadecimal Type	Valid Topics	Size (octets)
10	0x00A0	any	variable

Table 9: Content-Type Section

The content type section is used to signal to applications whether and how to consume a resource. In spirit, it can be treated as largely semantically equivalent to the collection of representation headers defined in [RFC7231], Section 3, most notably the "Content-Type" header.

Unlike the headers in HTTP, the section does not need to apply to an entire resource. Instead, the following logic applies:

1. If the section's topic is the metadata topic (Section 3.4.4.2), it specifies the default content type to assume if no others are provided.
2. If the section's topic is any other value, it specifies the content type only of sections in this topic. Such a topic content type therefore overrides the default.

Additionally, the content type only applies to application-defined sections. Sections described in this specification carry their own intrinsic meaning.

Much like HTTP headers, the section can carry multiple key-value pairs; the collection of key-value pairs therefore defines the content type as a whole. When a default content type and a topic specific content type are both provided, each key in the topic specific content type individually overrides the value provided in the default content type. Keys only provided in the default content type retain their meaning also for the topic. In this manner, both content types are merged into the final set of key-value-tuples applicable to a topic.

1. The section payload is encoded as a UTF-8 [RFC3629] character string.
2. Key-value pairs are separated from each other with a semicolon `; (U+003B)`.
3. Keys are separated from values with an equals sign `= (U+003D)`.
4. Either separator used as a key or value character must be escaped, i.e. preceded by a backslash `\ (U+005C)`.
5. Keys must be stripped of leading and trailing whitespace. In values, whitespace is considered part of the value. Whitespace is interpreted as equivalent to the `\p{Whitespace}` property in [UNICODE-TR18].

This is a variable sized section.

3.4.5.5. Section: BLOB

Decimal Type	Hexadecimal Type	Valid Topics	Size (octets)
11	0x00A1	any	variable

Table 10: BLOB Section

The BLOB section is a generic section for encoding binary large objects (BLOBs). It is variable sized.

BLOB sections are not interpretable unless more information about the data stored within is known. Applications **SHOULD** use a prior content type section (Section 3.4.5.4) for this purpose, but **MAY** decide that for their purposes, only a particular interpretation makes sense.

3.4.5.5.1. Content Type and HTTP

A number of HTTP headers have no semantic meaning to Vessel, but applications may find it interesting to use them. Unlike the Content-Type header, which is here represented as a few distinct keys, there is no such mapping provided for other headers in this specification. The only requirement is that if applications make use of HTTP header equivalents, their names **MUST** be the same as the HTTP header name in lower case, and they **MUST** additionally prefixed with the string "http-" to avoid collisions. For example, the HTTP "Content-Encoding" header would become "http-content-encoding", etc.

3.4.5.5.2. Content Type and Streaming

The purpose of breaking a Vessel resource into individual extents is to also facilitate streaming applications. In particular, this should permit decoding extents anywhere in the stream. As such, it is recommended that content type sections are repeated at regular intervals, otherwise they may remain unknown to decoding applications.

This, however, presents a uniqueness issue. If a subsequent content type section contains a different specification than a previous one, applications must decide which to honour.

To mitigate this, applications **MUST** only apply the content type to any sections following the content type section. If they encounter another content type section B later on, they **MUST** disregard the earlier section A. Subsequent sections **MUST** be interpreted according to the content type described in B, and so forth.

This scheme has the benefit of being simple and unambiguous, but does imply that it is wrong to assume a stream represents some kind of sub-resource. A better interpretation is that a stream represents a sequence of sub-resources delimited by content type sections.

3.4.5.5.3. Content Type Keys

This document defines only a few content type keys, some in reference to the HTTP standard, or the MIME standard HTTP is based upon.

Content Type Key	Equivalent to	Reference
media-type	Content-Type media-type	[RFC7231], Section 3.1.1.5
charset	Content-Type "charset" key	[RFC7231], Section 3.1.1.5
filename	Content-Disposition "filename" key	[RFC2183], Section 2.3

Table 11: HTTP/MIME Header Equivalency

Other keys are not defined at this point in time.

Note, however, that e.g. [RFC2183], Section 2.3 states that the "filename" key may contain only US-ASCII values. At the same time, this specification in Section 3.4.5.4 clearly states the use of UTF-8. In case of such conflicts, implementations **MUST** follow this specifications instead of the references.

3.5. Extent Identifiers

Extent identifiers uniquely identify each extent. As extent headers also encode the previous extent, an extent chain is effectively created. However, to ensure the result is effectively a chain and not some complex graph, the following must be fulfilled:

1. Individual authors must be able to create only a single new extent from a previous extent.
2. Multiple authors acting in parallel must not create the same extent identifier, effectively overriding each other's efforts.
3. There must be an absolute order to extents in order to create an unambiguous sequence. Any branches must eventually converge.

Origin extent identifiers are generated via a secure random function.

```
origin_extent = RNG(length(extent_identifier))
```

Figure 11: Pseudocode for Origin Extent Identifier Generation

Extent identifiers should be chosen as large enough that collisions are sufficiently unlikely, assuming the random function provides sufficient entropy. Given that UUID with its 128-bit labels is deemed sufficient in size for providing uniqueness ([RFC4122]), implementations **MUST** choose extent identifiers of at minimum 128 bits in length. Implementations **MAY** follow the UUIDv4 generation process to generate origin extent identifiers, but are not required to do so.

The recommendations expressed in the above paragraph may change in future.

Any identifiers for subsequent extents are generated by concatenating the author identifier and the previous extent identifier , as well as some random nonce, and then hashing the result. As this random nonce must be known in order to repeat this process, it must become part of the identifier itself. This may make it necessary to truncate the hash, but it is equally possible to choose the length of identifier such that it fits the hash as well as nonce.

Note that while the nonce discussed here is a salt, unlike in key stretching applications, the other inputs to the hash are not confidential. Some discussion on this is in Section 4.3. Introducing the salt is a low effort technique for making their discovery harder. For the most part, however, discussing a salt at all enables implementations to that do require these inputs to remain confidential able to provide such confidentiality with ease by providing a sufficiently larger salt and extent identifier.

For a similar reason, the hash function may be applied multiple times over the inputs. Implementations **MUST** apply it at least once, but **MAY** apply it more often. For confidentiality, implementations **SHOULD** fulfil the requirements of [RFC2898], Section 4.2.

Implementations **MUST** choose nonce sizes of 16 bits or more. For confidentiality, implementations **SHOULD** fulfil the requirements of [RFC2898], Section 4.1.

The choice of the number of iterations and nonce sizes is encoded as part of the `extent_identifier_hash` choice in the version tag.

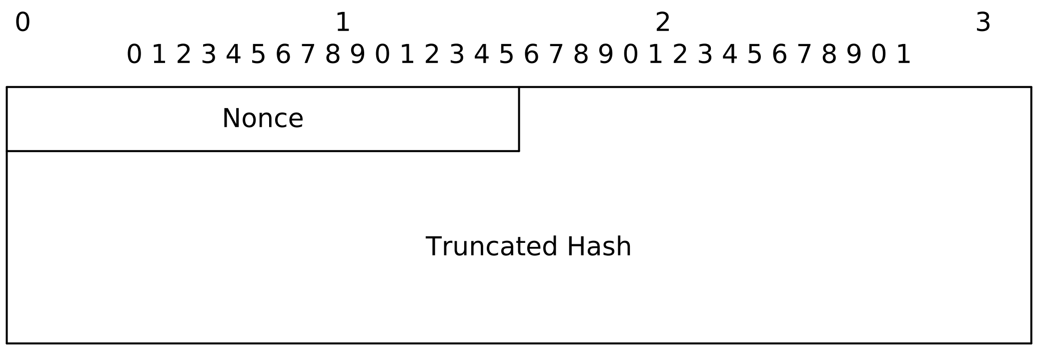

Figure 12: Nonce Diagram

Given the above layout, the extent identifier generation algorithm is as follows:

```
nonce = RNG(length(nonce))

hash_input = author_identifier | previous_extent_identifier

current_result = hash_input
foreach number_of_iteratons
  current_result = extent_identifier_hash(current_result | nonce)
done

truncated_length = length(extent_identifier) - length(nonce)
truncated_hash = truncate(current_result, truncated_length)

extent_identifier = nonce | truncated_hash
```

Figure 13: Pseudocode for Extent Identifier Generation

The two algorithms combined fulfil the first two requirements listed above. To also fulfil the last, we apply an ordering scheme to the resulting extent identifiers as described in Section 3.5.1.

3.5.1. Deterministic Ordering

Simply generating extent identifiers as outlined above will effectively yield a directed acyclic graph (DAG) of extents; more precisely, the result is a simple tree. In order to fulfil the last requirement above, we need a means by which authors deterministically choose the previous extent upon which they build.

As extents may not be always synchronized across all nodes, the approach provided here may still created branches. As extents become synchronized, however, all authors will converge on the same previous extent, thereby collapsing all branches again.

Eventual synchronization can mean that the origin extent is not known to a node at any given time. This means an incomplete resource may be known as a collection of sub-trees. The algorithm here nevertheless provides a deterministic means for finding the ideal extent identifier to use as the basis for a newly created one.

For illustration purposes, consider the following example:

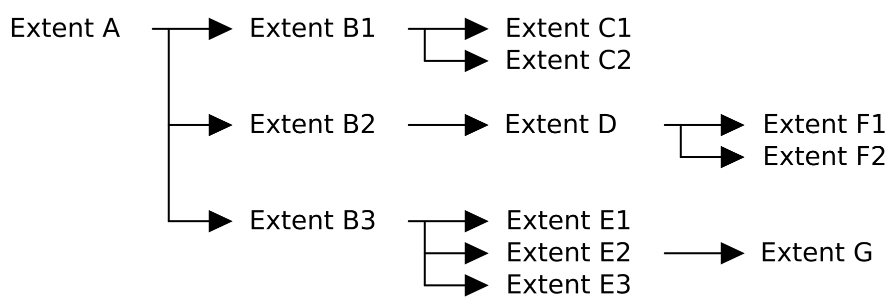

Figure 14: Example Extent Tree Diagram

First, we order extent identifiers by simple bitwise comparison. In the above example, this means that `B2` follows `B1`, and `B3` follows both, etc.

For the full algorithm, it is necessary to calculate the weight of each path from the root node at extent A, to all other nodes. The weight of each edge from a source to a target node is defined as one divided by the number of target nodes that lead from the source node. Alternatively, this is the number of siblings, including itself, that exist for the target node.

No.	Path	Edge Weights	Cumulative Path Weight
1	A -> B1	1/3	0.333
2	A -> B2	1/3	0.333
3	A -> B3	1/3	0.333

No.	Path	Edge Weights	Cumulative Path Weight
4	A -> B1 -> C1	1/3 + 1/2	0.833
5	A -> B1 -> C2	1/3 + 1/2	0.833
6	A -> B2 -> D	1/3 + 1	1.333
7	A -> B2 -> D -> F1	1/3 + 1 + 1/2	**1.833**
8	A -> B2 -> D -> F2	1/3 + 1 + 1/2	**1.833**
9	A -> B3 -> E1	1/3 + 1/3	0.666
10	A -> B3 -> E2	1/3 + 1/3	0.666
11	A -> B3 -> E2 -> G	1/3 + 1/3 + 1	1.666
12	A -> B3 -> E3	1/3 + 1/3	0.666

Table 12: Example Extent Path Weights

In this example, paths 7 and 8 have the highest weights. To tie break between paths of equal weight, the path wins whose leaf node comes last in bitwise ordering. In this example, `F2` comes after `F1`, so path number 8 wins.

Authors **MUST** select the leaf node of the winning path as the previous extent for creating a new extent.

Over time, this algorithm will converge on the longest and least ambiguous path. Nodes with fewer siblings have higher edge weights, and more nodes in the path will create a higher cumulative weight.

The algorithm will even lead to convergence after a major network split. Assume network A knows only a sub-tree TA, and network B knows only a sub-tree TB. Assume further that the winning paths in both sub-trees TA and TB have the same cumulative weight W. The two sub-trees will nonetheless have different leaf node extent identifiers, leading to a deterministic winner when both sub-trees are finally synchronized.

Note that the deterministic ordering scheme presented here is comparable to [TREEDOC]. However, instead of generating identifiers in some semantic order, this specification is only concerned with deterministic ordering. The disambiguator here is the inclusion of the authoring key, which is sufficient for the purposes of an abstract container.

3.6. Authoring

Authoring describes the act of creating an extent. To do so, one must be in possession of an authoring key pair. Authoring consists of several operations.

1. In order to author an extent, an author must first generate an extent identifier. The mechanisms for doing so are described in Section 3.5.
2. An extent must be created with the above extent identifier.
3. Data may be written to the extent payload (this is not strictly necessary; it is valid to create an extent where the payload consists solely of padding).
4. The extent must be cryptographically signed.

While the middle steps are certainly relevant in real-world scenarios, note that it is the combination of author identifier and signature that are the components that prove authorship.

The cryptographic signature here requires special attention. A signature here can come in two distinct forms:

1. An asymmetric signature. It **MUST** be possible to deduce a public key for verification from the author identifier of the extent. The signature **MUST** be created using the corresponding private key.
2. A message authentication code (MAC), if confidentiality is provided for the extent payload (see Section 3.7).

The choice of which signature to use is embedded in the version tag Section 3.2. This choice therefore applies to the entire extent. It is not possible to create extents with confidentiality provided only to partial payloads.

Note well that one implication of this is that a complete Vessel resource may consist of extents whose version tag differ. This is a deliberate choice made in order to support resources with different confidentiality requirements.

3.6.1. Author Identifiers

Author identifiers are derived from key pairs . Generally speaking, a public key serves as a sufficiently unique identifier in that a cryptographic challenge can be issued that only the party in possession of the corresponding private key can successfully respond to. Such a challenge protocol is outside of this document's scope.

Key fingerprints are often used as shorter, unique stand-ins for public keys. Unfortunately, each cryptosystem tends to define its own method for generating such fingerprints. The general approach, however, is to use a cryptographic hash function over a deterministic encoding of the key parameters.

For generating author identifiers, author public keys **MUST** be encoded using the Distinguished Encoding Rules (DER) of [X.690]. The author identifier hash function is an implementation choice. The resulting hash may be further truncated.

```
encoded = DER(author_public_key)

hash = author_identifier_hash(encoded)

author_identifier = truncate(hash, length(author_identifier))
```

Figure 15: Pseudocode for Author Identifier Generation

3.6.2. Authoring Counter

Extents are mutable; their content can change over time, while their identifier stays stable. This has benefits in addressing e.g. the right to be forgotten, but a downside is that means other than a content hash have to be found in order to signal that one version of an extent differs from another.

To provide this, we introduce the authoring counter . It is a counter value scoped to the tuple of author identifier and resource - that is, multiple authors may use the same value, and the same author may use the same value across multiple resources.

Whenever authors write to an extent, they also need to update the signature. Prior to doing so, they **MUST** increment the authoring counter they maintain for the resource, and write it to the header (i.e. the signature must extent also to this new counter value).

Doing so lets protocol implementations and receiving nodes quickly ascertain if an extent is a newer version (larger authoring counter than locally stored) or older version, and reject and/or delete older content. In order to maintain the right to be forgotten, implementations **MUST** delete extents thus outdated.

Nodes **SHOULD** implement means by which to detect other nodes that do not conform to this, as possible. Nodes **SHOULD** keep a record of versions sent to nodes, and if those nodes return older versions, blocklist them. Reasonable precautions **SHOULD** be taken to avoid feeding content to malicious or malfunctioning nodes, but such precautions are difficult to enforce remotely, and somewhat outside the scope of this specification.

The stable identifier and authoring counter method has benefits over the more common content hashing:

1. It is possible to detect outdated and updated versions, rather than merely detecting difference.
2. Approaches to creating resource identifiers from the sum of the extent signatures such as using merkle signature schemes (e.g. [RFC8391]) require the generation and propagation of a new resource identifier if a single component changes; this is less often the case if a similar tree scheme using stable extent identifiers was used.
3. Most importantly, the scheme does not break the deterministic ordering scheme described in Section 3.5.1.

Authoring counters do not have to be incremented strictly by one. The only requirement is that new counters are larger than previously written ones. Wrap-around will occur eventually, which is why implementations **SHOULD** use a reasonably large counter, and **MUST** use, at minimum, an unsigned integer of 24 bits or larger.

3.7. Confidentiality

Extents may provide payload confidentiality. Whether they do is encoded in the version tag (Section 3.2.1). If the version tag indicates an encryption algorithm is used to provide confidentiality, this algorithm should be chosen to decrypt parts of the extent prior to delivery to the application. The logic functions much the same in reverse: if the application requests confidentiality, the extent's version tag must encode this, and extents must be partially encrypted before storing or transmitting.

As the version tag is the leading information here, it follows that it may not be encrypted itself. It also follows that the choice of encryption spans the entire extent payload, not just parts of it. Authors wishing to mix encrypted with unencrypted content must write these to distinct extents.

Confidentiality may be provided by a variety of algorithms, such as standard authenticated encryption schemes with or without associated data, public key cryptography, and so forth. The present section is concerned with how to apply any of them in a generic fashion.

We must differentiate between symmetric and public key based schemes here, due to how public key schemes tend to be constructed. Because of the relative efficiency of symmetric schemes, large plain text - such as a Vessel extent - is usually encrypted symmetrically with an ephemeral key. The ephemeral key is then encrypted with the recipient(s) public key(s) before being transported, such that only they can decipher it, and therefore recover the content.

This specification is not concerned with transporting symmetric keys, though appropriately chosen AAA sections (Section 3.4.4.1) may be used for this purpose. A similar statement can be made for the exchange of public key information.

For the purposes of confidentiality, we can therefore conclude the following:

1. Confidentiality is provided via a symmetric encryption scheme.
2. The key used in this symmetric encryption scheme is not communicated in extent envelope, header or footer.
3. The signature algorithm may be provided by either of:
    - A message authentication code, using the same or a related key.
    - A public key pair based signature .

Requiring encryption and signing in this manner is equivalent to requiring authenticated encryption, though this term typically refers only to MAC based schemes. As the signature extends over unencrypted envelope and header data, it is practically an encrypted authentication with associated data scheme (AEAD).

In the interest of not creating new AEAD schemes, this specification permits only symmetric encryption algorithms that are already AEAD constructions. This has some implications.

1. AEAD schemes provide an authentication tag instead of a signature. Using confidentiality implies that the signature field in the extent footer contains this authentication tag. The choice of `signature_algorithm` then is limited (see Section 3.8.6).
2. Additionally, AEAD schemes are created for encrypting e.g. many network packets using the same secret key, a property that can be exploited to also produce many extents using the same secret key. To do so, each packet or extent uses a unique nonce. This nonce need not be confidential, as long as it is only used once per key. It is possible to derive this nonce from extent metadata (see Section 3.8.7).

3.7.1. Encrypt-then-Sign

Authenticated encryption schemes that first encrypt the plain text, and then sign the resulting cipher text do not provide an easy means for determining whether the decryption was successful.

When using such schemes, implementations **MUST** provide one of the CRC32 (Section 3.4.5.1), MAC (Section 3.4.5.2) or signature sections (Section 3.4.5.3). The presence of either already indicates that decryption was successful, and the section contents can extend this verification to the remaining payload.

Encrypt-then-MAC is recommended by [ISO-19772]. It is also conceptually the way AEAD schemes function.

3.7.2. Sign-then-Encrypt

When plain text is first signed, then encrypted, it is usually the case that the signature is concatenated to the plain text before encryption. In this manner, presence of the signature after decryption also indicates that decryption succeeded.

It is possible to emulate this by providing MAC (Section 3.4.5.2) or signature section (Section 3.4.5.3), and then using an otherwise unauthenticated stream cipher for encryption. The length of the signature in the footer would then be zero octets.

This mode is not supported by Vessel, and implementations **MUST NOT** use it.

3.7.3. Sign-then-Encrypt-then-Sign

In [SIGN-ENCRYPT], the author contests that neither sign-then-encrypt nor encrypt-then-sign provides sufficient guarantees for asymmetric encryption schemes. There is no guarantee in either case that the signing party and the encrypting party are identical, which can lead to some subtle forgery attacks.

It is possible to create that link by using a sign-then-encrypt-then-sign scheme in which the inner and outer signatures are verifiably made by the same party, with the implication that they at minimum agreed to the encryption.

As the symmetric ciphers permitted by this specification are already AEAD schemes of the sign-then-encrypt style, and based on a kind of MAC construct, a sign-then-encrypt-then-sign construction is reached if:

1. Encrypt-then-sign is used as in Section 3.7.1.
2. The checksumming section chosen is a MAC section (Section 3.4.5.2).
3. The key and nonce are identical for the MAC as for the outer AEAD scheme.

Implementations using confidentiality **SHOULD** use a MAC section in this manner. If the MAC section is chosen, implementations **MUST** use the same key and nonce as in the AEAD construction. See Section 3.8.4 for details.

3.7.4. Encrypted Extent Parts

Some extent fields must remain in plain text in order to identify and otherwise work with the extent, even if its contents remain encrypted.

1. The envelope is always in plain text. It is necessary for understanding anything at all about the extent layout. It also does not leak any sensitive meta information.
2. If encryption occurs, the payload is always encrypted in its entirety, including any padding.
3. The signature is always in plain text, as it verifies the encrypted payload. If a signature scheme with associated data is used, the signature **MUST** also verify the envelope and header.

Some difficulty is present in the question whether or not the header needs to be present in plain text, or may be encrypted. Encrypting would expose less meta information to observers, but limits the operations possible on the extent.

1. For streaming purposes, the extent size **MUST** be in plain text. At authoring time, it cannot be known which node uses which mode of data transfer, so mandating this field to always remain in plain text is only prudent. This information is also not particularly sensitive, as it provides few clues to the extent contents.
2. The current extent identifier **MUST** also be in plain text. It identifies the extent, and is necessary for any node to determine how to process this data. Extent identifiers may be generated using an algorithm that makes it infeasible to deduce its inputs (see Section 3.5).
3. The counter **MUST** be in plain text. The reason relates mostly to nonce generation and confidentiality (Section 3.8.7). But the tuple of current extent identifier and counter also uniquely identifies an extent version. This permits corretly handling multiple versions of the same extent, thereby permitting old extent payloads to be erased - this is required for the right to be forgotten to be implemented.

Given these fields, nodes can process streams of extents, and also make some decisions on whether and how to process them. The remaining header fields are not, strictly speaking, required unless a full resource is to be processed.

1. Knowledge of the previous extent identifier permits ordering individual extents into a single resource (Section 3.5.1).

2. Knowledge of the author identifier permits verifying that a public key based signature over the extent was created by the same public key as referenced via the author identifier.

If the private header flag is set in the version tag, these additional header fields **MUST** be encrypted. Otherwise they **MUST** remain in plain text.

3.8. Minimum Supported Algorithms

This section defines the mininum supported algorithms for specification conforming implementations.

3.8.1. Hash Functions

A cryptographically secure hash function is required for creating version tags, extent identifiers and author identifiers, and may be used in other constructs. Implementations Implementations **MUST** support the following algorithms:

Algorithm	Identifier	Use	Refer
SHA-3 512	sha3-512	version_tag_hash, extent_identifier_hash, author_identifier_hash, nonce_hash	[NIST 202]
SHA-3 384	sha3-384	version_tag_hash, extent_identifier_hash, author_identifier_hash, nonce_hash	[NIST 202]
SHA-3 256	sha3-256	version_tag_hash, extent_identifier_hash, author_identifier_hash, nonce_hash	[NIST 202]
SHA-3 224	sha3-224	version_tag_hash, extent_identifier_hash, author_identifier_hash, nonce_hash	[NIST 202]
SHA-2 512	sha-512	version_tag_hash, extent_identifier_hash, author_identifier_hash, nonce_hash	[NIST 180-4
SHA-2 384	sha-384	version_tag_hash, extent_identifier_hash, author_identifier_hash, nonce_hash	[NIST 180-4
SHA-2 256	sha-256	version_tag_hash, extent_identifier_hash, author_identifier_hash, nonce_hash	[NIST 180-4
SHA-2 224	sha-224	version_tag_hash, extent_identifier_hash, author_identifier_hash, nonce_hash	[NIST 180-4

Algorithm	Identifier	Use	Refer
No hash	none	author_identifier_hash	n/a

Table 13: Supported Hash Functions, their Uses and Definitions

Key pairs where the public key is small enough to reasonably fit into the author identifier field do not require hashing for creating collision free identifiers. Implementations **MUST** support this none hash function, which simply returns its input value.

Implementations **MUST** provide the SHA-3 family of hash functions, and the none function. The SHA-2 family of functions **MAY** be provided.

3.8.2. Public/Private Key Pairs

There is no particular reason for this specification to require the use of any specific public/private key pair type, as keys are only indirectly used here. However, the keys may be used for asymmetric signatures as described in Section 3.8.3, and for key exchanges building upon this specification or developed alongside it.

Implementations **MUST** only support key pairs for which asymmetric signature algorithms are defined. Furthermore, implementations **SHOULD** only support such key pairs for which key exchange algorithms are defined.

Only the asymmetric signature algorithm (Section 3.8.3) is included in the version tag, so specifying the key pair used directly is not required.

3.8.3. Asymmetric Signature Algorithm

As outlined above, Vessel requires support for asymmetric signature algorithms .

Algorithm	Identifier	Use	Reference
PureEdDSA for curve25519	ed25519	asymmetric_signature	[RFC8410], [RFC8032]
PureEdDSA for curve448	ed448	asymmetric_signature	[RFC8410], [RFC8032]
DSA	dsa	asymmetric_signature	[NIST.FIPS.186-4]
RSA	rsa	asymmetric_signature	[NIST.FIPS.186-4]

Table 14: Supported Asymmetric Key Algorithms, their Uses and Definitions

Implementations **MUST** provide the PureEdDSA functions based on edwards curves. DSA and RSA functions **SHOULD** be provided for compatibility with existing security standards.

3.8.3.1. Relation to Author Identifier Hash

- The EdDSA algorithms using edwards curve based keys produce small enough public keys that an author_identifier_hash may not be required, i.e. the none hash function **MAY** be used here (see Section 3.8.1).
- Both DSA and RSA may produce public keys large enough that using a fingerprint (i.e. author_identifier_hash) of the key is useful. Implementations **MUST** use a supported hash function here. They **SHOULD** use a version of SHA-3, but **MAY** use other hash functions. Keys **MUST** be written with DER encoding rules according to [X.690], with the hash function applied to the result.

3.8.3.2. Relation to Signature Hash

- The EdDSA algorithm defines what signature hash function to use in [RFC8032]. The signature_hash value **MUST** therefore be set to eddsa if this asymmetric signature algorithm is used.
- The DSA algorithm requires the selection of a hashing algorithm for signature_hash. Implementations **MUST** use one of the SHA-2 family of hash functions.
- The RSA algorithm also requires an signature_hash. Implementations **MUST** select one of the SHA-2 or SHA-3 family of hash functions.

3.8.4. Message Authentication Codes

The following message authentication codes are defined for use in MAC sections (Section 3.4.5.2).

Algorithm	Identifier	Use	Reference
Poly1305-AES	poly 1 3 0 5	message_authentication	[RFC8439], [POLY1305]
KMAC128	kmac 1 2 8	message_authentication	[NIST.SP.800-185], [NIST.FIPS.202]
KMAC256	kmac 2 5 6	message_authentication	[NIST.SP.800-185], [NIST.FIPS.202]

Table 15: Supported MAC Algorithms, their Uses and Definitions

Note that the section payload is the resulting code *only*, which also means the section size depends on the algorithm selection in the version tag. Any nonces or keys used in computing the MAC **MAY** be encapsulated in appropriate sections in the authentication topic (Section 3.4.4.1), but this is out of scope for this specification.

Some MAC algorithms require a nonce. Implementations **MUST** either provide a unique nonce per extent as required by the algorithm, or provide a unique key per extent if the algorithm has no nonce requirement. See Section 3.8.7 for details.

Implementations **MUST** provide the Poly1305 based algorithm, and **SHOULD** also provide the KMAC functions.

3.8.5. Symmetric Encryption

For encrypting the extent payload, we require symmetric stream ciphers that **SHOULD NOT** impose any expectation that the plain text is a multiple of some block size in length. This is because extent sizes are multiples of page sizes, and extent metadata occupies some of that space.

It is in principle possible to select this metadata size such that the payload size is divisible by some stream cipher's block size without remainder. Implementations **MAY** provide such a specific selection of algorithms, and thus permit additional stream ciphers. However, implementations **MUST NOT** permit the selection of such stream ciphers without taking precations with selecting the metadata size as above.

The list contains only authenticated encryption algorithms with associated data (AEAD). The rationale for this is in Section 3.7. These algorithms require a nonce per extent; see Section 3.8.7 for details on this.

Algorithm	Identifier	Use	Reference
ChaCha20	chacha20	symmetric_encryption	[RFC8439], [CHACHA]
AES-128 in GCM mode	aead_aes_128_gcm	symmetric_encryption	[RFC5116]
AES-256 in GCM mode	aead_aes_256_gcm	symmetric_encryption	[RFC5116]
AES-128 in CCM mode	aead_aes_128_ccm	symmetric_encryption	[RFC5116]
AES-256 in CCM mode	aead_aes_256_ccm	symmetric_encryption	[RFC5116]
No confidentiality	none	symmetric_encryption	n/a

Table 16: Supported Symmetric Encryption Algorithms, their Uses and Definitions

As these are AEAD modes, the extent signature is provided by the AEAD algorithm's authentication tag. See Section 3.8.6.

Implementations **MUST** provide the ChaCha20 implementation. AES-based AEAD **SHOULD** be implemented as well, but may be omitted.

If the special identifier none is chosen, there is no encryption and no confidentiality.

3.8.6. Signature Algorithm

The signature algorithm is a reference to other algorithms specified in the version tag hash; its identifiers therefore are not algorithm identifiers as such, but reserved words that refer to another algorithm identifier.

Extents can be authenticated via the signature, and additionally extent payloads may be confidential. If confidentiality is provided, the authentication is already part of the AEAD algorithm used. If confidentiality is not provided, signatures may either be provided by a message authentication code or an asymmetric signature. All extents **MUST** include a signature.

Algorithm	Identifier	Refers to	Use
Authentication tag	aead	The AEAD algorithm's authentication tag	**MUST** use when confidentiality is provided
Asymmetric signature	keypair	asymmetric_signature algorithm	**SHOULD** use when confidentiality is not provided
Message authentication code	mac	message_authentication algorithm	**MAY** use when confidentiality is not provided

Table 17: Signature Algorithm Uses

3.8.7. Nonce Generation

There are a number of attacks possible when keys - for message authentication or confidentiality - are re-used across multiple extents. In order to mitigate against these, some MAC algorithms and all AEAD algorithms listed in the previous sections require the use of a nonce. This permits sharing a key across multiple extents. The only requirement is that the nonces **MUST** be unique per extent.

Unlike the key, they do not themselves have to be confidential. That means we can derive a nonce from public metadata of the extent. The envelope is not particularly unique here, however, the current extent identifier and authoring counter in the public header together serve to identify a unique extent version.

[RFC8439] requires 96 bit nonces, which [RFC5116] also recommends. The latter also suggests using a fixed nonce prefix as well as a variable counter to construct the nonce. That is exactly what the current extent identifier and authoring counter provide.

Their length, however, depends on the algorithm choices. To make them more predictable, we will hash their concatenation, and truncate the result to 96 bits (12 octets).

```
inputs = current_extent_identifier | authoring_counter

hash = nonce_hash(inputs)

nonce = truncate(hash, 1 2 )
```

Figure 16: Pseudocode for Nonce Generation

3.8.7.1. Nonce Use

Some MAC algorithms may not require a nonce. However, the reasons for using a nonce still remain.

1. If the algorithm (MAC, AEAD, etc.) require a nonce, implementations **MUST** use the nonce as the algorithm requires.
2. If the algorithm does not require a nonce, implementations **MUST** still use it as outlined below.

The nonce is additional material that is used to create a unique key per extent from the key shared across extents as follows:

```
unique_key = nonce_hash(shared_key | nonce)
```

Figure 17: Pseudocode for Unique Key Generation from Nonce

4. Related Considerations

4.1. Human Rights Considerations

What follows is a list of objectives derived from [RFC8280], each with a brief statement how Vessel addresses each concern, or why it does not.

4.1.1. In Scope

Connectivity: By addressing storage independent of transport, Vessel observes the end-to-end principle, and does not require any specific functionality in middle-boxes.

Reliability: Vessel separates content into extents, each of which constitutes a fully defined Vessel document. Vessel organises extents in such a way that partial availability of extents does not negatively impact the ability to process those that are available (although encapsulated data may not have that same quality). Furthermore, extent authoring by multiple sources is conflict-free.

Content agnosticism: Vessel is fully content agnostic and treats data as opaque binary objects.

Heterogeneity Support: Part of the goal of subdividing data streams into extents was to permit heterogeneous devices to participate in manipulating them, as extents limit e.g. memory and bandwidth requirements. Similarly, no specific constraints are placed upon transfer protocols, allowing heterogeneity also here.

Integrity: Vessel contains provisions for ensuring the integrity of encapsulated data.

Authenticity: Vessel extent contents are always authenticated.

Confidentiality: Vessel extent contents may be encrypted; the format provides for facilities to achieve that.

Privacy: Vessel provides anonymous modes of operation and data confidentiality, which together specifically address [RFC6973], Section 5.1.2.

Censorship resistance: Censorship resistance is difficult to achieve within Vessel itself. Extents contain identifiers, which can be used to filter content either if the transport protocol itself does not provide confidentiality. Similarly, middle-boxes could be used to filter content. However, censorship resistance is one reason why the design of Vessel explicitly eschews content hashes as extent identifiers. As a result, censorship can be circumvented simply by generating a new identifier for the same content.

Outcome Transparency: If these specifications do not provide for outcome transparency, that **SHOULD** be considered reason for an amendment.

Adaptability: Adaptability is one of the major concerns of Vessel, being a content agnostic container format. A reason for subdividing content into self-contained extents was to make the format more easily usable in streaming applications, as well as persistent data storage.

Decentralization: The multi-authoring capabilities of Vessel aim at decentralized uses. Extents belonging to the same data stream can be created independently by multiple authors, synchronized, and brought into eventual consistency. If no synchronization occurs, multiple diverging versions can be maintained independently of each other.

Remedy: Remedy (new in [I-D.draft-irtf-hrpc-guidelines-13] often stands in stark contrast to anonymity and pseudonymity, both of which are out of scope. That is, it will be difficult to seek remedy against anonymous use that is abusive. However, another consideration in the design of Vessel is the [GDPR], specifically the "right to be forgotten". A leading consideration in generating extent identifiers independent of content is that content can also be replaced and deleted as a result. This stands in stark contrast to approaches using content hashes as identifiers, which favour immutability of content.

Open Standards: Care has been taken to define this specification in reference to other open standards (see the references section).

4.1.2. Out of Scope

Internationalization:

Vessel treats data as opaque binary objects, and thus permits internationalization of such data, but does not explicitly support it.

Localization The resource format is intended for processing by computers; localizing the name of each field is the concern of an application displaying this data.

Security: [BCP72] lists data transfer (i.e. protocol) considerations that are not in scope for Vessel. Furthermore, it enumerates concerns regarding the handling of cryptographic keys. Vessel itself is agnostic to such mechanisms, but may nonetheless be used to carry related information. Some suggestions on this are provided, but security as per [RFC8280] is, strictly speaking, out of scope.

Pseudonymity: Identifiers in Vessel are not tied to any personally identifiable information, and can be entirely ephemeral. It is up to applications using Vessel to ensure pseudonymity concerns.

Anonymity: As with pseudonymity, anonymity is an application concern. However, this document enumerates some in Section 4.4.

Accessibility: Being content agnostic, Vessel permits accessibility concerns being addressed in applications, but does not provide any facilities for it.

4.2. Protocol Considerations

Vessel does not require the use of any specific transport protocol. However, this section elaborates some considerations on what makes a transport protocol a good fit for Vessel.

Above other considerations, an extent is typically larger than the average network packet/ datagram. It is therefore necessary that transports provide a mode for chopping up the extent into packet payloads, and re-assembling the result at the receiving end. Applications that combine a Vessel implementation with transport **SHOULD** treat the entire extent as either available in full or not at all; Vessel has no provision for dealing with partial extents.

In a request/response oriented protocol, implementations **SHOULD** respond to the tuple of extent identifier and authoring counter. This tuple specifies a precise extent version. Implementations **SHOULD** additionally respond to just the extent identifier, and return the latest authoring counter for this identifier they are aware of. Implementations **MAY** refuse to return extent versions that are older than the latest they are aware of; this would help implement the right to be forgotten.

Implementations **MAY** treat a request for the origin extent of a resource as representing the entire resource. By example of a podcast encapsulated in Vessel, it may depend on context whether this means downloading a pre-recorded file, or joining a live stream at the current time stamp. The only suggestion here is that the origin extent identifier makes for a good resource identifier due to the logical tree structure that extent identifiers produce.

4.3. Security Considerations

Vessel is not a network protocol; a number of security considerations from [BCP72] do not apply. Predominantly, this specification attempts to protect against offline cryptographic attacks. From a certain point of view, each extent can be viewed as an individual message in a message stream. This means also active attacks from [BCP72], Section 3.3 are in scope of these considerations.

Vessel takes care not to invent new cryptographic constructs. It relies on well understood mechanisms such as authenticated encryption with associated data (AEAD), rather than producing a similar result via newly crafted encrypt-then-sign schemes or similar.

The use of the version tag is additionally a security consideration. While it makes sense for the longevity of this specification to make specific cryptographic algorithms an implementation choice, it also makes sense to limit such choices. According to [CVE-TYPE], vulnerabilities in cryptographic implementations are far from the most common, but regularly make the top ten list of issues. The report cannot measure impact, however. Often, cryptography is left to few base components upon which the majority of applications build.

It is prudent to limit the number of permutations of cryptographic algorithms, while still permitting the possibility for easily replacing outdated ones (Section 3.2.1.2).

Key negotiation is out of scope of this specification; therefore, attacks that relate to gaining access to key material, passwords, etc. are out of scope.

4.3.1. Confidentiality

Vessel extents are designed with confidentiality in mind, but may be used without such features. This is desirable in a container format, as information contained within may also be disseminated to the public.

The approach to confidentiality is based on current best practice AEAD algorithm choices from [RFC8439] and [RFC5116], and follow the suggestions in those documents.

4.3.2. Data Integrity

Data integrity is provided via a mandatory signature over the entire extent. This may be provided in the form of an AEAD authentication tag, in which case all unencrypted extent data is included in the associated data. Alternatively, MAC or asymmetric signature algorithms are provided for non-confidential extent payloads.

4.3.3. Peer Entity Authentication

In the context of Vessel, peer entity authentication may better be referred to as data origin authentication. In either case, the mandatory extent signatures are always keyed. This requires both origin and recipient to previously perform some kind of key exchange.

While the key exchange itself is not in scope of this document, on the assumption that it has occurred, it is possible to verify the data origin. Particular attention is given to subtler attacks when confidentiality is also used (see Section 3.7.3).

4.3.4. Non-Repudiation

Non-repudiation is desirable, but out of scope of this document largely because key exchange is out of scope. This specification presumes that the validity of keys is established before attempting to use them.

4.3.5. Unauthorized Usage

Authorization is generally outside of the scope of this specification, as it effectively relates to the problem space of key exchange, validation, etc.

However, some authorization concerns can be discussed. As the container format is specifically designed for multi-authoring purposes, authorization concerns must raise their head.

1. *Read authorization* is effectively granted by either
2. Providing the extent payload in plain text, or
3. Initiating a key exchange for the encrypted payload, which is itself out of scope.
4. *Write authorization* is less of a question of permitting authorship; any entity can generate a key pair and start authoring extents for a resource. Rather, it becomes a question of which authors are considered trustworthy enough to be included in the resource.

As the origin extent identifier is randomly generated, there is nothing stopping an attacker from re-using the same origin extent identifier for a competing resource. For a reader to verify the correct origin extent has been received is then a queston of accepting or rejecting the origin extent's author. This is out of scope of this specification. However, implementations using Vessel **MUST** take care to establish the origin extent author's trustworthiness.

With trust in the origin extent established, it is prudent to trust only those authors of additional extents that the origin extent author explicitly authorizes. Such a scheme is outside of the scope of this document; however, the AAA topic (Section 3.4.4.1) is specifically reserved to encapsulate such information.

Given an appropriate authorization scheme and a verified origin extent author, nodes can identify unauthorized extents, and **MUST** reject them as not belonging to the same resource.

4.3.6. Inappropriate Usage

Any scheme that prevents unauthorized use may also be extended to prevent inappropriate use. As above, these schemes are out of scope.

4.3.7. Denial of Service

It is feasible to treat the generation of extents that are not authored by authorized users as a kind of denial of service attack. It certainly consumes resources on the recipient's side.

Most such "denial of service" mitigation is out of scope for the reasons previously mentioned. However, an authorization scheme that is transported within Vessel extents itself may also permit nodes from understanding the authorization state of a resource without being privy to encrypted extent payloads. Such nodes **SHOULD** not relay extents to further recipients that they can already reject as unauthorized, thereby limiting the spread of the denial attack.

4.3.8. Replay Attacks

Replay attacks are not inherently damaging to Vessel; an extent may be received multiple times. The complete resource de-duplicates this via the extent ordering scheme described in Section 3.5.1.

However, appropriate countermeasures against a node transmitting an extent multiple times **SHOULD** be implemented near the transport layer.

4.3.9. Message Insertion

Message insertion is largely equivalent to the unauthorized use case described in Section 4.3.5, and mitigation against it therefore the same.

4.3.10. Message Deletion

The extent ordering scheme in Section 3.5.1 is deliberately designed to enable both handling of partial trees as well as deterministic merging of them afterwards. Message deletion in the sense of extent deletion therefore is mitigated against.

There is, however, some interaction with unauthorized use. Assume a reader wishes to join the live stream of a resource at the current time point. If this extent is not authored by the same entity as the origin extent, it will be difficult to establish trust in it; attackers may exploit this.

To exploit this, an attacker must be a man-in-the-middle. They must receive extents from upstream, and scan them for authentication information. They must then delete - i.e. not pass on - the extent containing such information. The downstream recipient now has two resource subtrees to deal with, one from the origin to this deleted extent, and one following the deleted extent. An attacker may use this deletion attack to then launch an insertion attack, and fake the entire subtree following the deleted extent.

Within Vessel itself, it is not possible to mitigate against such an attack. This is in scope of a transport protocol, however. Simple transport encryption via TLS and establishment of trust in the sending node mitigates the man-in-the-middle scenario, however.

If it is not possible to establish trust in the sending node in this manner, protocol implementations **MUST** take care to securely transmit authentication data out of band of the extent stream.

4.3.11. Message Modification

The mandatory signature algorithms should be sufficient mitigation against message modification.

4.3.12. Man-In-The-Middle

A conceivable man-in-the-middle scenario is discussed in Section 4.3.10; otherwise, as Vessel is not a networking protocol, the kind of attack does not easily apply.

4.3.13. Key Usage

Vessel relies on secure key exchange outside of its own specification. More precisely, it assumes that:

1. Public keys are exchanged, and mapped to author identifiers.
2. Encryption and MAC keys are exchanged as necessary.
3. The latter are updated regularly; see [I-D.draft-irtf-cfrg-aead-limits-05] for usage limits.

Each of these exchanges can be mapped into extent sections in the reserved AAA topic (Section 3.4.4.1), but such a mapping is not in scope of this document.

Nonces are, where they are used, derived from public extent metadata. This is consistent with the MAC and AEAD constructs referenced in this document, and should not pose any security risk.

4.4. Privacy Considerations

This section lists privacy considerations as covered by [RFC6973] and Vessel's relationship to them.

4.4.1. Surveillance

The surveillance concers outlined in [RFC6973] specifically relate to network protocols; Vessel is not such a protocol.

It is in the nature of a storage format that it applies to files, which may be duplicated and analysed at leisure. Vessel can only provide confidentiality.

However, it is worth stressing that authoring keys are in no way required to be identity keys, that is, keys intrinsically tied to a person. They may be ephemeral keys, and rotated at will in order to mitigate some surveillance measures. This interacts with key exchanges (which are also out of scope), as well as the attack described in {#sec:security-considerations-message-deletion}.

The consequence of this is that implementations adopting this format **SHOULD** take care to make authoring keys last only as long as necessary.

Besides efficiency in memory and storage mapping, the fixed sized extent was also chosen to mitigate against some kinds of "traffic" analysis, whereby the size of extent sequences can be matched against likely content candidates.

Using confidentiality and always padding to the same extent size obscures the size of the plain text payload, and makes such analysis harder.

4.4.2. Stored Data Compromise

Vessel mitigates against most stored data compromises by offering encrypted extents. Other data, such as key material, is outside of this specification's scope. Implementations **MUST** recognize that compromised keys may lead to data compromises.

4.4.3. Intrusion

In the context of Vessel, intrusion maps neatly onto message insertion ({#sec:security-considerations-message-insertion}). Vessel attempts to protect against such attacks, but is reliant on up-to-date authorization data at the receiving end.

4.4.4. Misattribution

Misattribution in [RFC6973] refers to misattribution to individuals; Vessel has no understanding of individuals. It is possible to create identity key pairs for individuals and use them directly in authoring extent. Such use is strongly discouraged. Implementations **SHOULD** rather generate authoring key separately, and use a secure channel to get them authorized. Such authorization may require linking them to the identity key, but this is not required by, and not known to Vessel.

Authorization and key generation are out of scope for this specification.

4.4.5. Correlation

Vessel cannot protect against correlation of any data sent in plain text. When confidentiality is used, it is difficult to glean any information from the plain text that would permit correlation.

The main information that can be correlated is in the use of the authoring key. Implementations **SHOULD** not only regularly generate new authoring keys and undergo re-authorization, but also **SHOULD** use different authoring keys for different resources.

As a result, it will be difficult to correlate any information over longer term usage. Only within the window in which a single authoring key is used can an attacker establish how many extents this author created.

Authorization and key generation are out of scope for this specification.

4.4.6. Identification

The authoring key is the only public information that can be used to identify a user. As explained in the previous sections, authoring keys should not be used over a long term. This also thwarts identification by providing less data to link to the same user.

As far as personal identifiable information (PII) is concerned, the [GDPR] requires that such information can be deleted (right to be forgotten). This is the reason why extent identifiers are not based on content, but on position in the resource stream. This permits modifying the content in a later version of the same extent, which includes deleting content.

Note, however, that the extent identifier and authoring key are strongly linked. An updated extent must be authored by the same authoring key. If applications wish to support this right to be forgotten, they **MUST** store outdated keys until all of an extent payload has been deleted, and the extent is empty.

4.4.7. Secondary Use

Secondary use concerns are not in scope of this document.

4.4.8. Disclosure

As Vessel is a storage format rather than a network protocol, the threat model assumes attackers have easy access to extents. Any disclosure related concerns are authorization concerns, which are largely out of scope for this document except as discussed in {#sec:security-considerations-unauthorized-usage}.

4.4.9. Exclusion

Exclusion is a network protocol consideration, and does not apply to Vessel. It is strongly recommended that transport protocols consider this issue.

Consider a transport protocol that permits subscribing to a resource. As new extents get created, the node creating them may advertise their creation to subscribers immediately. It is feasible for an attacker to deduce from such advertisments when an author is both actively creating and/or attached to the internet in some fashion.

Transport protocols **SHOULD** consider batching advertisments or delaying them to thwart such analysis.

4.5. IANA Considerations

This document has no IANA actions.

5. References

5.1. Normative References

[CHACHA] Daniel J Bernstein, "ChaCha, a variant of Salsa20", January 2008, <http://cr.yp.to/chacha/chacha-20080128.pdf>.

[GDPR] Council of the European Union, "General Data Protection Regulation (GDPR)", EU Regulation 2016/679, 27 April 2016, <https://eur-lex.europa.eu/legal-content/EN/ALL/?uri=CELEX:32016R0679&qid=1661512309950>.

[ISO-19772] Technical Committee ISO/IEC JTC 1/SC 27 Information security, cybersecurity and privacy protection, "ISO/IEC 19772:2020: Information security - Authenticated encryption", ISO 19772.202, November 2020, <https://www.iso.org/standard/81550.html>.

[NIST.FIPS.180-4] Dang, Q., "Secure Hash Standard", National Institute of Standards and Technology report, DOI 10.6028/nist.fips.180-4, July 2015, <https://doi.org/10.6028/nist.fips.180-4>.

[NIST.FIPS.186-4] "Digital Signature Standard (DSS)", National Institute of Standards and Technology report, DOI 10.6028/nist.fips.186-4, July 2013, <https://doi.org/10.6028/nist.fips.186-4>.

[NIST.FIPS.202] Dworkin, M., "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions", National Institute of Standards and Technology report, DOI 10.6028/nist.fips.202, July 2015, <https://doi.org/10.6028/nist.fips.202>.

[NIST.IR.8366] Miller, K., Alderman, D., Carnahan, L., Chen, L., Foti, J., Goldstein, B., Hogan, M., Marshall, J., Reczek, K., Rioux, N., Theofanos, M., and D. Wollman, "Guidance for NIST staff on using inclusive language in documentary standards", National Institute of Standards and Technology (U.S.) report, DOI 10.6028/nist.ir.8366, April 2021, <https://doi.org/10.6028/nist.ir.8366>.

[NIST.SP.800-185] Kelsey, J., Chang, S.-j., and R. Perlner, "SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash and ParallelHash", National Institute of Standards and Technology special publication, DOI 10.3030/NIST.SP.800-185, n.d., <https://doi.org/10.6028/nist.sp.800-185>.

[POLY1305] Daniel J Bernstein, "The Poly1305-AES message-authentication code", March 2005, <http://cr.yp.to/mac/poly1305-20050329.pdf>.

[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>.

[RFC2183] Troost, R., Dorner, S., and K. Moore, Ed., "Communicating Presentation Information in Internet Messages: The Content-Disposition Header Field", RFC 2183, DOI 10.17487/RFC2183, August 1997, <https://www.rfc-editor.org/rfc/rfc2183>.

[RFC3365] Schiller, J., "Strong Security Requirements for Internet Engineering Task Force Standard Protocols", BCP 61, RFC 3365, DOI 10.17487/RFC3365, August 2002, <https://www.rfc-editor.org/rfc/rfc3365>.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <https://www.rfc-editor.org/rfc/rfc3629>.

[RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, DOI 10.17487/RFC5116, January 2008, <https://www.rfc-editor.org/rfc/rfc5116>.

[RFC5652] Housley, R., "Cryptographic Message Syntax (CMS)", STD 70, RFC 5652, DOI 10.17487/RFC5652, September 2009, <https://www.rfc-editor.org/rfc/rfc5652>.

[RFC6973] Cooper, A., Tschofenig, H., Aboba, B., Peterson, J., Morris, J., Hansen, M., and R. Smith, "Privacy Considerations for Internet Protocols", RFC 6973, DOI 10.17487/RFC6973, July 2013, <https://www.rfc-editor.org/rfc/rfc6973>.

[RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, DOI 10.17487/RFC7231, June 2014, <https://www.rfc-editor.org/rfc/rfc7231>.

[RFC7624] Barnes, R., Schneier, B., Jennings, C., Hardie, T., Trammell, B., Huitema, C., and D. Borkmann, "Confidentiality in the Face of Pervasive Surveillance: A Threat Model and Problem Statement", RFC 7624, DOI 10.17487/RFC7624, August 2015, <https://www.rfc-editor.org/rfc/rfc7624>.

[RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <https://www.rfc-editor.org/rfc/rfc8032>.

[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>.

[RFC8280] ten Oever, N. and C. Cath, "Research into Human Rights Protocol Considerations", RFC 8280, DOI 10.17487/RFC8280, October 2017, <https://www.rfc-editor.org/rfc/rfc8280>.

[RFC8410] Josefsson, S. and J. Schaad, "Algorithm Identifiers for Ed25519, Ed448, X25519, and X448 for Use in the Internet X.509 Public Key Infrastructure", RFC 8410, DOI 10.17487/RFC8410, August 2018, <https://www.rfc-editor.org/rfc/rfc8410>.

[RFC8439] Nir, Y. and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols", RFC 8439, DOI 10.17487/RFC8439, June 2018, <https://www.rfc-editor.org/rfc/rfc8439>.

[UNICODE-TR18] Mark Davis, "Unicode® Technical Standard #18: Unicode Regular Expressions", 8 February 2022, <http://www.unicode.org/reports/tr18/>.

[X.690] International Telecommunications Union, "Information Technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)", ITU-T Recommendation X.690, 1994.

5.2. Informative References

[BCP72] Rescorla, E. and B. Korver, "Guidelines for Writing RFC Text on Security Considerations", BCP 72, RFC 3552, DOI 10.17487/RFC3552, July 2003, <https://www.rfc-editor.org/rfc/rfc3552>.

[CVE-TYPE] MITRE Corporation, "Vulnerability Type Distributions in CVE", 22 May 2007, <https://cve.mitre.org/docs/vuln-trends/index.html>.

[DMC] "Distributed Mutable Containers (DMC)", n.d., <http://purl.org/dmc>.

[DUBLIN-CORE] DCMI Usage Board, "DCMI Metadata Terms", 20 January 2020, <https://www.dublincore.org/specifications/dublin-core/dcmi-terms/>.

[ERIS] Renberg, E., "Encoding for Robust Immutable Storage (ERIS)", n.d., <https://eris.codeberg.page/spec>.

[FREENET] Clarke, I., Sandberg, O., Wiley, B., and T. Hong, "Freenet: A Distributed Anonymous Information Storage and Retrieval System", Designing Privacy Enhancing Technologies pp. 46-66, DOI 10.1007/3-540-44702-4_4, 2001, <https://doi.org/10.1007/3-540-44702-4_4>.

[I-D.draft-irtf-cfrg-aead-limits-05] Günther, F., Thomson, M., and C. A. Wood, "Usage Limits on AEAD Algorithms", Work in Progress, Internet-Draft, draft-irtf-cfrg-aead-limits-05, 11 July 2022, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-aead-limits-05>.

[I-D.draft-irtf-hrpc-guidelines-13] Grover, G. and N. ten Oever, "Guidelines for Human Rights Protocol and Architecture Considerations", Work in Progress, Internet-Draft, draft-irtf-hrpc-guidelines-13, 28 March 2022, <https://datatracker.ietf.org/doc/html/draft-irtf-hrpc-guidelines-13>.

[I-D.draft-knodel-terminology-10] Knodel, M. and N. ten Oever, "Terminology, Power, and Inclusive Language in Internet-Drafts and RFCs", Work in Progress, Internet-Draft, draft-knodel-terminology-10, 11 July 2022, <https://datatracker.ietf.org/doc/html/draft-knodel-terminology-10>.

[NGI0-Discovery] Stichting NLNet, "NGI Zero Discovery", DOI 10.3030/825322, 1 November 2018, <https://doi.org/10.3030/825322>.

[NLNET] Stichting NLNet, "NGI Zero Discovery", n.d., <https://nlnet.nl/discovery/>.

[RFC2898] Kaliski, B., "PKCS #5: Password-Based Cryptography Specification Version 2.0", RFC 2898, DOI 10.17487/RFC2898, September 2000, <https://www.rfc-editor.org/rfc/rfc2898>.

[RFC4122] Leach, P., Mealling, M., and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace", RFC 4122, DOI 10.17487/RFC4122, July 2005, <https://www.rfc-editor.org/rfc/rfc4122>.

[RFC7927] Kutscher, D., Ed., Eum, S., Pentikousis, K., Psaras, I., Corujo, D., Saucez, D., Schmidt, T., and M. Waehlisch, "Information-Centric Networking (ICN) Research Challenges", RFC 7927, DOI 10.17487/RFC7927, July 2016, <https://www.rfc-editor.org/rfc/rfc7927>.

[RFC8391] Huelsing, A., Butin, D., Gazdag, S., Rijneveld, J., and A. Mohaisen, "XMSS: eXtended Merkle Signature Scheme", RFC 8391, DOI 10.17487/RFC8391, May 2018, <https://www.rfc-editor.org/rfc/rfc8391>.

[SIGN-ENCRYPT] Don Davis, "Defective Sign & Encrypt in S/MIME, PKCS#7, MOSS, PEM, PGP, and XML", Proceedings of USENIX Technical Conference 2001 , 5 May 2001, <http://world.std.com/~dtd/sign_encrypt/sign_encrypt7.PDF>.

[TREEDOC] Preguica, N., Marques, J., Shapiro, M., and M. Letia, "A Commutative Replicated Data Type for Cooperative Editing", 2009 29th IEEE International Conference on Distributed Computing Systems, DOI 10.1109/icdcs.2009.20, June 2009, <https://doi.org/10.1109/icdcs.2009.20>.

[UNHRC51] United Nations Human Rights Council, "The right to privacy in the digital age", Annual reports of the United Nations High Commissioner for Human Rights and reports of the Office of the High Commissioner and the Secretary-General, report A/HRC/51/17, 4 August 2022, <https://undocs.org/A/HRC/51/17>.

Acknowledgments

Development of this work was undertaken under a grant agreement with NLNet Foundation [NLNET], which manages the NGI Zero Discovery fund [NGI0-Discovery].

Index

Author's Address

Jens Finkhäuser
Interpeer gUG (haftungsbeschraenkt)
Email: ietf@interpeer.io
URI: https://interpeer.io/