ICNRG
Internet-Draft
Intended status: Experimental
Expires: 4 September 2025

C. Tschudin
University of Basel
C.A. Wood
Cloudflare
M.E. Mosko

D. Oran, Ed.
Network Systems Research & Design
3 March 2025

File-Like ICN Collections (FLIC)
draft-irtf-icnrg-flic-07

Abstract

This document describes how to encode an application data objet into a structured _manifest_ using Information Centric Networking (ICN) data objects, creating a File-Like ICN Collection (FLIC). The manifest is an "index table" of objects that make up the manifest itself and the application data. It records the hash value (content object hash) of each item so a consumer using the manifest may request each piece by a complete hash name. The manifest is hierarchical and may be encoded into realtively small ICN objects to fit within network MTU sizes. FLIC has several methods to guide a consumer in constructing appropriate Interest names based on the manifest. It also supports encryption of the manifest data. FLIC may be used in CCNx or Named Data Networking, or other ICNs.

Table of Contents

1. Introduction

ICN architectures, such as Content-Centric Networking (CCNx)[RFC8569] and Named Data Networking [NDN], are well suited for static content distribution. Each piece of (possibly immutable) static content is assigned a name by its producer. Consumers fetch this content using said name. Optionally, consumers may specify the full name of content, which includes its name and a cryptographic digest of said content, so the reques is self-verifying.

> Note: The reader is assumed to be familiar with general ICN concepts from CCNx or NDN. For general ICN terms, this document uses the terminology defined in [RFC7927]. Where more specificity is needed, we utilize CCNx [RFC8569] terminology where a Content Object is the data structure that holds application payload. Terms defined specifically for FLIC are enumerated below in Section 3.1.
>
> In this document, we prefer the CCNx terminology. We will use the CCNx term Content Object, where NDN would use the term Data packet. The term Content Objec Hash in CCNx corresponds to the NDN Implicit Digest.

To enable requests with full names, consumers need _a priori_ knowledge of content digests. A Manifest, a form of catalog, is a data structures commonly employed to store and transport this information. Typically, ICN manifests are signed content objects (data) which carry a collection of hash digests. As content objects, a manifest itself may be fetched by full name. A manifest may contain either hash digests of, or pointers to, either other manifests or content objects. A collection of manifests and content objects represents a large piece of application data, e.g., one that cannot otherwise fit in a single content object. Because a manifest contains a collection of hashes, it is by definition non-circular because one cannot hash the manifest before filling it in.

Structurally, this relationship between manifests and content objects is reminiscent of the UNIX inode concept with index tables and memory pointers. In this document, we specify a simple, yet extensible, manifest data structure called FLIC - _File-Like ICN Collection_. FLIC is suitable for ICN protocol suites such as CCNx and NDN. We describe the FLIC design, grammar, and various use cases, e.g., ordered fetch, seeking, de-duplication, extension, and variable-sized encoding. We also include FLIC encoding examples for CCNx and NDN.

The purpose of a manifest is to concisely name, and hence point to, the constituent pieces of a larger object. A FLIC manifest does this by using a _root_ manifest to name and cryptographically sign the data structure and then use concise lists of hash-based names to indicate the constituent pieces. This maintains strong security from a single signature. A Manifest entry gives one enough information to create an _Interest_ for that entry, so it must specify the name, the hash digest, and if needed, the locators.

FLIC is not an archive format, like _tar_ or _LTFS_ or _iso_. FLIC represents a single object that may be reconstructed by concatenating all the data pieces. Future extensions could define a new type of structure that aggregates multiple FLIC manifests into an archive.

FLIC is a distributed data structure illustrated by the following picture. The FLIC manifest is encoded in the Payload of a Content Object. A _root manifest_ has a name and a publisher signature. Subsequent manifests, because they hare in a cryptographic hash tree using SHA256 pointers, do not need to be signed; they have implicit trust. In the figure we call out "data pointer" and "manifest pointer", but in the actual encoding they are all just hash values and the consumer does not necessarily know what it is fetching until it has fetched it.

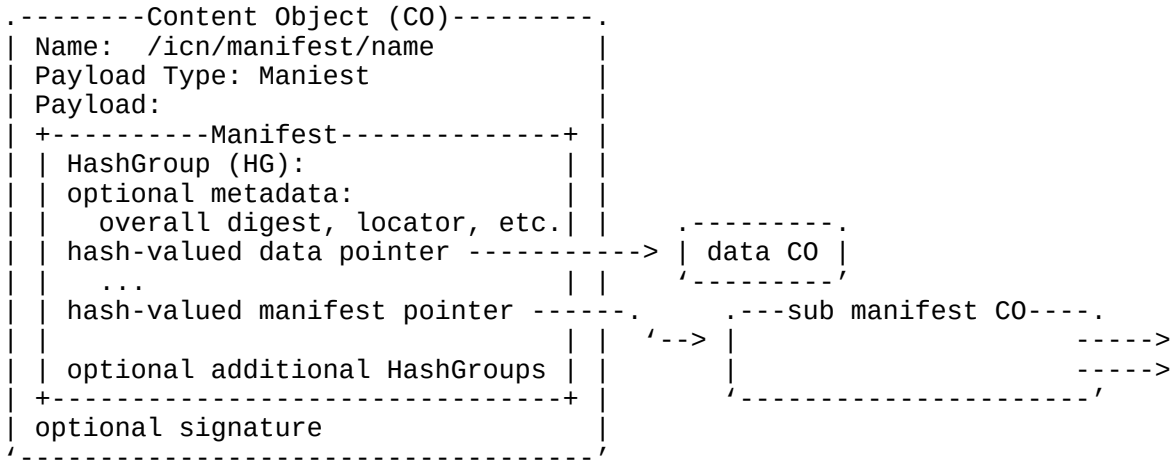


Figure 1: Simplified FLIC Content Object and acyclic graph example

A key design decision is how one names the root manifest, the application data, and subsidiary manifests. FLIC uses the concept of a Name Constructor. The root manifest (in fact, any FLIC manifest) may include a Name Constructor that instructs a manifest reader how to properly create Interests for the associated application data and subsidiary manifests. The Name Constructors allow interest construction using a well-known, application-independent set of rules. Some name constructor forms are tailored towards specific ICN protocols, such as CCNx or NDN; some are more general and could work with many protocols. We describe the allowed Name Constructor methods in Section 3.3. There are also particulars of how to encode the name schema in a given ICN protocol, which we describe in Section 3.9.

FLIC has encodings for CCNx (Section 3.9.1) as per [RFC8609] and for NDN (Section 3.9.2). For the implementor, the section Section 3.9.1.5 succintly defines the CCNx TLV structures and the format of each TLV value.

An example implementation in Python may be found at [ccnpy].

1.1. FLIC as an ICN experimental tool

FLIC enables experimentation with how to structure and retrieve large data objects in ICN. By having a common data structure applications can rely on, with a common library of code that can be used to create and parse manifest data structures, applications using ICN protocols can both avoid unnecessary reinvention and also have enhanced interoperability. Since the design attempts to balance simplicity,

universality, and extensibility, there are a number of important experimental goals to achieve that may wind up in conflict with one another. We provide a partial list of these experimental issues in Section 4. It is also important for users of FLIC to understand that some flexibility and extensions might be removed if use cases do not materialize to justify their inclusion in an eventual standard.

1.2. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD 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.

2. Design Overview

The FLIC design adopts the proven UNIX inode concept of direct and indirect pointers, but without the specific structural forms of direct versus indirect. FLIC is a collection of pointers, and when one de-references the pointer it could be an application object or another FLIC manifest. The pointers in FLIC use hash-based naming of Content Objects analogous to the function block numbers play in UNIX inodes.

Because FLIC uses hash-based pointers as names, FLIC graphs are inherently acyclic. Both CCNx and NDN support hash-based naming, though the details differ (see Section 3.9.1 and Section 3.9.2).

The FLIC data structure is an acyclic digraph of Content Objects. In this document, our examples are mostly trees, but that is not a requirement. For example, a de-duplication representation might have a common object or common sub-graph that is referenced from multiple parents in the tree. As another example, there could be a common sub-collection of objects organized in a Manifest, and that sub-manifest could be included in multiple places. Each node has a total order of all its children and they are visited in a specific traversal order. This leads to an unambiguous traversal even if a node has multiple in-edges.

A FLIC manifest graph does not need to have a unique root. When a publisher initially creates a FLIC manifest, it likely creates a unique root manifest. Over time, however, there may be additional root manifests. These could be used to update the root (e.g. replace an expired key or use a new certificate). Or the root might be renamed, or new locators used because the location of the data changed. An updated root might even point to the prior roots to maintain a history or access to prior cryptographic signatures or co-signatures.

In FLIC terms, a direct pointer links to application-level data, which is a Content Object with application data in the Payload. An indirect pointer links to a Content Object with a FLIC Manifest in the Payload. There is no distinction in the manifest between these two types of pointers.

> | Note: A substantial advantage of using hash-based naming is
> | that it permits block-level de-duplication of application data
> | because two blocks with the same payload may have the same hash
> | name if encoded that way.

A publisher creating a FLIC manifest may choose one of many permissible acyclic graph structures. Some examples are:

Chain: where each manifest has many direct pointers and at most only one indirect pointer.

Regular n-ary tree where there are m direct pointers and k indirect pointers at internal nodes (m+k=n) and n direct pointers at leaf manifests.

A leaf-only tree where interior nodes only have indirect pointers and leaf manifest nodes only have direct pointers.

As an example of choosing graph structure, the [ccnpy] implementation has a TreeOptimizer that calculates an optimal number of indirect and direct pointers per manifest node, given the limitations of a maximum object size and the overhead of necessary fields in each manifest object. The optimizer will find a set of tree parameters that minimizes the tree height and minimizes the number of wasted pointers due to the size of the application data.

A RECOMMENDED FLIC structure is to use a distinguished root manifest with exactly one pointer to the top manifest of a regular n-ary graph. The root manifest is signed by the publisher and contains links to certificates or public keys. It also specifies the Name Constructors used in the manifest. If using RSA-OAEP key wrapping, it contains the wrapped key. These one-time data structures take

significant space, so off-loading them to the root manifest is needed to use a regular n-ary graph for the rest of the manifest. In CCNx, the root manifest is the only one in the graph that needs to have a name if using nameless objects.

This method allows an intermediary service to respond to client requests with its own signed root Manifest that then points to the original root manifest. The client trusts the intermediary's response because of the intermediary's signature, and then trusts the content because of the Root manifest. In some cases, the intermediary could embed the original root manifest and avoid additional round trips before beginning download. This technique is used in a peer-to-peer sharing protocol [ProjectOrigin].

FLIC supports manifest encryption separate from application payload encryption (See Section 3.8). It has a flexible encryption envelope to support various encryption algorithms and key discovery mechanisms. The byte layout allows for in-place encryption and decryption.

A limitation of this approach is that one cannot construct a hash-based name for a child until one knows the payload of that child. In practical terms, this means that one must have the complete application payload available at the time of manifest creation. One workaround is to batch publication of a block of data, then extend that manifest tree as the top-left-child of a subsequent manifest tree. This results in multiple, signed, root manifests with two pointers (to the prior root manifest and the new manifest tree).

FLIC's design allows straightforward applications that just need to traverse a set of related objects. FLIC has two extensibility mechanisms that allow for more sophisticated uses: manifest metadata, and pointer annotations. These are described in Section 3.4 and Section 3.5 respectively.

FLIC goes to considerable lengths to allow creation and parsing by application-independent library code. Therefore, any options used by applications in the data structure or encryption capabilities MUST NOT require applications to have application-specific Manifest traversal algorithms to correctly re-construct the original data. This ensures that such application agnostic libraries can always successfully parse and traverse any FLIC Manifest by ignoring the optional capabilities. An application MAY include optimization or other hints to allow an application to use that data in different ways.

The reader may find it useful to refer to Section Example Usages (Appendix A) from time to time to see worked out examples.

3. FLIC Structure

3.1. Terminology

Data Object: a CCNx nameless Content Object that usually only has Payload. It might also have an ExpiryTime to limit the lifetime of the data.

Direct Pointer: borrowed from inode terminology, it is a CCNx link using a content object hash restriction and a locator name to point to a Data Object.

Hash Group: KA collection of pointers. A Manifest should have at least one Hash Group. A Hash Group may have its own associated meta data and Name Constructor.

Indirect Pointer: borrowed from inode terminology, it is a CCNx link using a content object hash restriction and a locator name to point to a manifest content object.

Internal Manifest: some or all pointers are indirect. The order and number of each is up to the manifest builder. By convention, all the direct manifests come first, then the indirect.

Leaf Manifest: all pointers are direct pointers.

Locator: a routing hint in an Interest used by forwarding to get the Interest to where it can be matched based on its Name Constructor-derived name.

Manifest: a CCNx ContentObject with PayloadType 'Manifest' and a Payload of the encoded manifest. A leaf manifest only has direct pointers. An internal manifest has a mixture of direct and indirect pointers. The outer TLV type of the payload identifies the manifest specifically as a FLIC manifest.

Manifest Waste: a metric used to measure the amount of waste in a manifest tree. Waste is the number of unused pointers. For example, a leaf manifest might be able to hold 40 direct pointers, but only 30 of them are used, so the waste of this node is 10. Manifest tree waste is the sum of waste over all manifests in a tree.

Name Constructor: The specification of how to construct an Interest for a Manifest entry.

Node The Node section of a manifest holds the manifest information.

It has node-wide metadata in a NodeData section, and then a series of HashGroups .

EncryptedNode A Node that is encrypted as specified in the Manifest's SecurityCtx. It is opaque octets.

Root Manifest: A signed, named, manifest that points to manifest nodes. The root manifest typically also defines the name constructors and encryption context data, if needed. The other manifest nodes may be CCNx nameless objects.

Top Manifest: One useful manifest structure is to use a Root manifest that points to a single Internal manifest called the Top Manifest. The Top manifest the begins the structure used to organize manifests. It is also possible to combine the two and use only a root manifest that also serves in the role of the top manifest.

3.2. Locators

Locators are routing hints used by forwarders to get an Interest to a node in the network that can resolve the Interest's name. In some naming conventions, the name might only be a hash-based name so the Locator is the only available routing information. Locators exist in both CCNx and NDN, though the specific protocol mechanisms differ. A FLIC manifest represents locators in the same way for both ICN protocols inside Name Constructors (Section 3.3), though they are encoded differently in the underlying protocol. See Section 3.9 for encoding differences.

CCNx only uses locators with nameless objects, whereas NDN may use multiple ForwardingHints per interest.

A name constructor definition may define one or more Locator prefixes that can be used in the construction of Interests from the pointers in the manifest. The Locators are inherited when walking a manifest tree, so they do not need to be defined everywhere. It is RECOMMENDED that only the Root manifest contain Locators so that a single operation can update the locators. One use case when storing application payloads at different replicas is to replace the Root manifest with a new one that contains locators for the current replicas.

3.3. Name Constructors

A manifest organizes pointers inside Hash Groups. Each Hash Group uses an NCID to indicate what Name Constructor to use to generate the name of an Interest for pointers inside that Hash Group.

A Manifest may define zero or more name constructors in Name Constructor Definitions (NCDs) located in the Manifest Node. An NCD associates a Name Constructor Id (NCID) to a Name Constructor definition. The NCID is used in other parts of the Manifest to refer to that specific definition.

A name constructor definition, NcDef, is placed in a NodeData field. An NcDef applies to that Node and all subsequent nodes along that manifest branch, until superceded by a new NcDef with the same NcId. A subsequent NcDef applies from that point onward in the manifest branch. It does not affect its ancestors.

In a non-tree FLIC graph, a node may have multiple parents, with different NcDef for the same NcId (i.e. parent 1 has NcDef A for the NcID and parent 2 has NcDef B for the same NcID). This is allowed. A consumer traversing the tree performs a specific traversal order. It must apply its current branch's name constructor to its visited children. Of course, if a consumer has already visited a sub-tree, it likely already has the hash named objects fetched and does not need to re-fetch them.

A common practice is to have the manifest nodes in one namespace and the application data objects in a different namespace. For example, ccnx:/foo/photo.jpg/manifest/ManifestId=N for a manifest tree and ccnx:/foo/photo.jpg/Chunk=M for the application data. Or if using nameless objects, the manifest tree and data objects might have different locators. In these cases, a FLIC manifest usually has two name constructor definitions (e.g. NCID 0 for the data and NCID 1 for the manifest tree). This results in the direct pointers being in one hash group and then the indirect pointers being in a second hash group.

If a traversal of a FLIC manifest encounters an unknown NCID, the RECOMMENDED action is to report a malformed manifest to the user and abort the traversal.

It is RECOMMENDED to have NcDef fields only in the root or top manifest. It is RECOMMENDED to not re-define NcId values within the same manifest.

An explicit name constructor has an NcDef in a NodeData and a corresponding NcId in a HashGroup. NcId 0 is a special NCID that may be used implicitly.

3.3.1. Implicit Name Constructor

NCID 0 is an implicit name constructor. If a Hash Group does not have a GroupData with an NcId element, it is assumed to use NCID 0.

NCID 0 has an implicit definition as a Hash Schema. If NCID 0 has no NcDef, it is assumed to be a HashSchema with no locators. It is RECOMMENDED to define NCID with Locators to avoid repeatedly defining locators elsewhere.

It is more efficient, encoding-wise, to always have an NcDef for NCID 0 (if it is used) that enumerates its Locators once rather than having to put them in each NodeData or GroupData.

NCID 0 MAY be re-defined as any name constructor type. It does not need to remain a Hash Schema. In this case, a publisher may be able to omit a GroupData for a group that uses NCID 0, or at least omit the 5 bytes of an NcId field within a GroupData.

3.3.2. Name Constructor Schemas

FLIC defines the following types of Name Constructors. Additional name constructor types may be specified in a subsequent revision of the specification. The section Section 3.6 specifies the encoding of each name constructor.

Hash Schema: The children of a Hash Group using the Hash Schema are fetched using the Locators of the NcDef, or the HashGroup, or the NodeData, in that order if the more specific definition does not exist. CCNx can only use one locator per Interest (in the Name), so if there are more than one available Locator, it must choose one, such as in the order listed.

Prefix Schema: The NcDef> specifies a name, which will appear in every content object for that name constructor. The consumer uses it in an interest with the appropriate hash. The NCD may specify locators, which NDN uses as ForwardingHints. They are not used by CCNx, as these are not nameless objects. In Prefix Schema, all object have an identical name and are only differentiated by their hash.

Segmented Schema: Every content object has a different name, denoted by the last name component. FLIC allows the publisher to specify the TLV type of that last name component (e.g. ManifestId or ChunkNumber in CCNx). The publisher MUST include a mechanism to tell the consumer the ID of each content object. FLIC provides two methods: use a StartSegmentId field in the HashGroup or use a _SegmentIdAnnotation_ for each pointer. The consumer will use the Name in the NcDef as a prefix, append a TLV of the given type, and set the TLV value to the segment ID. The resulting name is used in the Interest. For NDN, Locators may also be used as Forwarding Hints.

3.3.3. Segmented Schema Details

The Segmented Prefix schema uses a different name in all Content Objects and distinguishes them via their ContentObjectHash. Note that in CCNx, using a SegmentedPrefixSchema means that only the Root Manifest has a Locator for the Segmented Prefix (minus the segment ID).

The consumer must determine the segment ID to use in the name. Typically, the application data would use a chunk number in the content name, which is a compact sequence number. These could be computed based on the in-order traversal, assuming one knows which manifests point to data versus other manifests. Manifests, however, may not use a compact sequence number and there is no simple way to predict the names.

The SegmentedSchema uses a name component suffix to distinguish each content object. This name component will be of type SuffixComponentType. In CCNx, this would be a ChunkNumber or ManifestId type (T_MANIFEST_ID is defined in this document's IANA section (Section 5)), though one may specify different types. The reason for specifying the SuffixComponentType in the name definition is that application data would typically use chunked names with sequential identifiers and the FLIC manifest tree would typically use the ManifestId.

A publisher may name the manifest tree and the application data as desired, as long as the consumer can generate the required Interest names. FLIC provides two mechanisms: AnnotatedPointers and StartSegmentId. One could use an annotated pointer to specify the exact name or name suffix for each pointer, so the consumer does not need to calculate anything; it simply uses the name specified. The downside is there is an annotation for each pointer, which reduces the number possible pointers in a given MTU. The StartSegmentId is a single integer per hash group that indicates the beginning ID for the pointer array. The consumer calculates each pointer's name suffix from the start segment ID plus pointer offset, and then wraps it in a name component of type SuffixComponentType.

Because FLIC uses a preorder tree traversal, the node IDs of a manifest's children in the traversal order are not compact. They will skip around based on the preorder recursion. This means one needs a different ID for each manifest if one wants to use the StartSegmentId hint. A publisher may use any form of integer ID in the ManifestId name component so long as it is locally (within a hash group) sequential and globally unique.

As an example, the [ccnpy] implementation uses the regular k-ary tree IDs per tree height as the manifest IDs (where k is the number of allowed indirect pointers per manifest). These are globally unique and locally sequential for all the children of a manifest.

There are two methods to guide a consumer in generating segment IDs for hash pointers:

* If fetching a pointer with a SegmentIdAnnotation (see Section 3.5), the consumer MUST use that segment ID for the pointer.

* If the GroupData has a StartSegmentId parameter, then the segment ID of each pointer is calculated the StaretSegmentId plus the offset of the pointer from the first pointer of the Hash Group. Pointers with a SegmentIdAnnotation still count against the offset even though the annotation takes precedence.

As an example, consider the manifest shown in Figure 2. The Interests generated, in order, have these names and hash restrictions. We have used short example hash values for the sake of clarity.

1. /foo/7=10, hash=0x0001
2. /foo/7=20, hash=0x0002
3. /foo/7=12, hash=0x0003
4. /bar/8=0, hash=0x0004
5. /bar/8=1, hash=0x0005
6. /bar/8=2, hash=0x0006

```
Meanifest
  Node
    NodeData
      NcDef NcId 1 SegmentedSchema Name ’/foo’ SuffixComponentType 7
      NcDef NcId 2 SegmentedSchema Name ’/bar’ SuffixComponentType 8
    HashGroup
      GroupData NcId 1 StartSegmentId 10
      AnnotatedPtrs
        PointerBlock
          Ptr HashValue(0x0001)
          Ptr HashValue(0x0002) SegmentIdAnnotation(20)
          Ptr HashValue(0x0003)
    HashGroup
      GroupData NcId 2 StartSegmentId 0
      Ptrs
          HashValue(0x0004)
          HashValue(0x0005)
          HashValue(0x0006)
```

Figure 2: Segment ID Example

Every group of a segmented NcId MUST have either a GroupData with a StartSegmentId, or use annotated pointers with SegmentIdAnnotation.

A producer MUST ensure that a segment ID names exactly one data item. That is, the producer MUST NOT duplicate the segment ID in an object name for a different object. The object hash MUST be the same for the same segment ID of a name, within the longest of the ExpiryTime of the manifest or the named object. In other words, there is a one-to-one correspondence between a segment ID and a hash for the lifetime of the manifest and the constituent objects. Note that a _consumer_ should not assume this is true, as a bad actor could violate this rule. A consumer should only assume the uniqueness of cryptographic hashes.

It is allowed to have multiple manifest entries with the same segment ID (see below).

While a producer MAY mix using GroupData StartSegmentId and SegmentIdAnnotation, we in general do not consider that a good idea. It is up to the manifest producer to ensure that every segment may be fetched, and fetch in the right order. Segments, when fetched in the *manifest order* reconstruct the original data.

Let us make this clear, the original data is constructed by the in-order manifest traversal, not the segment ID order. The consumer should make no assumptions about reconstructing the original data and the names of those data elements.

A consumer MAY skip fetching some segments. A consumer MAY> fetch segments in any order it chooses. It may skip around or omit segments.

A manifest MAY have multiple pointers to the same segment ID (and thus same hash). This can be used for data de-duplication, e.g. multiple occurrences of the same binary string within the reconstructed data object. If the producer uses this method, then the original data cannot be reconstructed by simply fetching the segment IDs in order.

3.4. Manifest Metadata

The FLIC Manifest may be extended by defining TLVs that apply to the Manifest as a whole, or alternatively, individually to every data object pointed to by the Manifest. This basic specification does not specify any, but metadata TLVs may be defined through additional RFCs or via Vendor TLVs. FLIC uses a Vendor TLV structure identical to [RFC8609] for vendor-specific annotations that require no standardization process.

For example, some applications may find it useful to allow specialized consumers such as _repositories_ (for example [repository]) or enhanced forwarder caches to pre-place, or adaptively pre-fetch data in order to improve robustness and/or retrieval latency. Metadata can supply hints to such entities about what subset of the compound object to fetch and in what order.

> | Note: FLICs ability to use separate namespaces for the Manifest
> | and the underlying Data allows different encryption keys to be
> | used, hence giving a network element like a cache or repository
> | access to the Manifest data does not as a side effect reveal
> | the contents of the application data itself.

3.4.1. SubtreeDigest and LeafDigest

It is RECOMMENDED that the root manifest have a SubtreeDigest field in the NodeData with the cryptographic hash of the entire application data. This allows the manifest consumer to validate that the final reconstructed data is correct.

A consumer SHOULD validate the outer-most SubtreeDigest, if any. If this digest does not validate, a consumer SHOULD validate nested SubtreeDigest or LeafDigest to help pinpoint where there is a discrepancy.

As an example, a publisher might include a SubtreeDigest in the root manifest NodeData and then a LeafDigest in every manifest with direct pointers. It is not necessary to validate every LeafDigest if the overall SubtreeDigest checks. If the SubtreeDigest does not check, then the consumer could locate where the problem is by validating the LeafDigests.

A consumer SHOULD inform the user when a digest does not validate.

A discrepancy in a digest should be considered an error condition.

3.4.2. SubtreeSize and LeafSize

The SubtreeSize and LeafSize fields are provided to help a consumer seek through a manifest to specific locations. They are not intended to validate the application data.

A consumer MAY compare the sizes in SubtreeSize and LeafSize to the true application payload sizes retrieved from the manifest nodes and report any discrepancies to the user.

3.5. Pointer Annotations

FLIC allows each manifest pointer to be annotated with extra data. Annotations allow applications to exploit metadata about each Data Object pointed to without having to first fetch the corresponding Content Object. This specification defines three annotations: SizeAnnotation, SegmentIdAnnotation, and LinkAnnotation.

The SizeAnnotation specifies the number of application layer octets covered by the pointer.

The SegmentIdAnnotation is a unique suffix for the pointer for use with the Segmented Schema name constructor. Using annotations allows non-sequential segment IDs, which could happen if manifest trees are named in the traversal order. See Section 3.3.3 details on using the annotation.

The LinkAnnotation is a link to be used to name the pointer (less the pointer itself). For NDN, the pointer would be appended to the name for a complete name. In CCNx, the name would be used in the Interest along with the pointer in the ContentObjectHashRestriction field. The link may include a KeyId restriction.

> | One may include a KeyId restriction in the generated interest
> | by also using the ProtocolFlags option of a name constructor.
> | This would be more space efficient than using annotated
> | pointers.

Other annotation may, for example, give hints about a desirable traversal order for fetching the data, or an importance/precedence indication to aid applications that do not require every content object pointed to in the manifest to be fetched. This can be very useful for real-time or streaming media applications that can perform error concealment when rendering the media.

Additional annotations may be defined through additional specifications or via Vendor TLVs.

3.6. Manifest Grammar (ABNF)

The manifest grammar is mostly, but not entirely independent of the ICN protocol used to encode and transport it. The TLV encoding therefore follows the corresponding ICN protocol, so for CCNx FLIC uses 2 octet length, 2 octet type and for NDN uses the 1/3/5 octet types and lengths (see [NDNTLV] for details). There are also some differences in how one structures and resolves links. [RFC8609] defines HashValue and Link for CCNx encodings. The NDN ImplicitSha256DigestComponent defines HashValue and NDN Delegation (from Link Object) defines Link for NDN. Section 3.9 below specifies these differences.

The basic structure of a FLIC manifest comprises a security context, a node, and an authentication tag. The security context and authentication tag are not needed if the node is unencrypted. A node is made up of a set of metadata, the NodeData, that applies to the entire node, and one or more HashGroups that contain pointers.

The NodeData element defines the namespaces used by the manifest. There may be multiple namespaces, depending on how one names subsequent manifests or data objects. Each HashGroup may reference a single namespace to control how one forms Interests from the HashGroup. If one is using separate namespaces for manifests and application data, one needs at least two hash groups. For a manifest structure of "MMMDDD," (where M means manifest (indirect pointer) and D means data (direct pointer)) for example, one would have a first HashGroup for the child manifests with its namespace and a second HashGroup for the data pointers with the other namespace. If one used a structure like "MMMDDDMMM," then one would need three hash groups.

```
TYPE = 2OCTET / {1,3,5}OCTET ; As per CCNx or NDN TLV
LENGTH = 2OCTET / {1,3,5}OCTET ; As per CCNx or NDN TLV

Manifest = T_FLIC_MANIFEST LENGTH
              [SecurityCtx] (EncryptedNode / Node) [AuthTag]

SecurityCtx = T_SECURITY_CTX LENGTH AlgorithmCtx
AlgorithmCtx = AEADCtx / RsaWrapCtx
AuthTag = T_AUTH_TAG LENGTH *OCTET ; e.g. AEAD authentication tag
EncryptedNode = T_ENCRYPTED_NODE LENGTH *OCTET ; Encrypted Node

Node = T_NODE LENGTH [NodeData] 1*HashGroup [Pad]
NodeData = T_NODE_DATA LENGTH [SubtreeSize] [SubtreeDigest]
              [Locators] 0*Vendor 0*NcDef
SubtreeSize = T_SUBTREE_SIZE LENGTH INTEGER
SubtreeDigest = T_SUBTREE_DIGEST LENGTH HashValue
```

```
   Pad = T_PAD LENGTH 0*OCTET ; as per RFC8609

   NcDef = T_NCDEF LENGTH NcId NcSchema
   NcId = T_NCID LENGTH INTEGER
   NcSchema = InterestDerivedSchema / DataDerivedSchema /
              PrefixSchema / SegmentedSchema / HashSchema

   InterestDerivedSchema = TYPE LENGTH [ProtocolFlags]
   DataDerivedSchema = TYPE LENGTH [ProtocolFlags]

   PrefixSchema = T_PrefixSchema LENGTH Name [Locators] [ProtocolFlags]
   SegmentedSchema = T_SegmentedSchema LENGTH Name SuffixComponentType
                        [Locators] [ProtocolFlags]
   SuffixComponentType = T_SUFFIX_TYPE LENGTH 1*OCTET
   HashSchema = T_HashSchema LENGTH Locators [ProtocolFlags]

   Locators = T_LOCATORS LENGTH 1*Locator
   HashValue = TYPE LENGTH *OCTET ; As per ICN Protocol
   Locator = TYPE LENGTH Link ; Link As per ICN protocol

   ProtocolFlags = T_PROTOCOL_FLAGS LENGTH *OCTET
                   ; ICN-specific flags, e.g. must be fresh

   HashGroup = T_HASH_GROUP LENGTH [GroupData] (Ptrs / AnnotatedPtrs)
   Ptrs = T_PTRS LENGTH *HashValue
   AnnotatedPtrs = T_ANNOTATED_PTRS LENGTH *PointerBlock
   PointerBlock = T_PTR_BLOCK LENGTH *Annotation Ptr
   Ptr = T_PTR LENGTH HashValue

   Annotation = SizeAnnotation / SegmentIdAnnotation /
                LinkAnnotation / Vendor
   SizeAnnotation = T_ANN_SIZE LENGTH Integer
   SegmentIdAnnotation = T_ANN_SEGMENT_ID LENGTH Integer
   LinkAnnotation = T_LINK LENGTH Link
   Vendor = T_ORG LENGTH PEN *OCTET
   PEN = 3OCTET ; IANA Private Enterprise Number

   GroupData = TYPE LENGTH [NcId] [LeafSize] [LeafDigest]
               [SubtreeSize] [SubtreeDigest] [StartSegmentId]
               [Locators] 0*Vendor
   LeafSize = T_LEAF_SIZE LENGTH INTEGER
   LeafDigest = T_LEAF_DIGEST LENGTH HashValue
   StartSegmentId = T_START_SEGMENT_ID LENGTH Integer

   AEADCtx = T_AEAD_CTX LENGTH AEADData
   AEADData = KeyNum AEADNonce AEADMode [KDFData]
   KeyNum = T_KEYNUM LENGTH INTEGER
   AEADNonce = T_NONCE LENGTH 1*OCTET
```

```
   AEADMode = T_AEADMode LENGTH (AEAD_AES_128_GCM / AEAD_AES_256_GCM /
              AEAD_AES_128_CCM / AEAD_AES_128_CCM)
              ; RFC5116 definitions
   KDFData = T_KDF_DATA LENGTH KDFAlg [KDFInfo]
   KDFAlg = T_KDF_ALG LENGTH INTEGER
       ; IANA "HPKE KDF Identifiers" [RFC9180]
   KDFInfo = T_KDF_INFO LENGTH 1*OCTET
       ; Passed to the KDF Info

   RsaOaepCtx = T_RSAOAEP_CTX LENGTH RsaOaepData
   RsaOaepData = AEADData [RsaOaepWrapper]
   RsaOaepWrapper = KeyId [KeyLink] HashAlg WrappedKey
       ; KeyId as pre RFC8609 for CCNx
       ; KeyLink as pre RFC8609 for CCNx
   HashAlg = T_HASH_ALG LENGTH alg_number
       ; alg_number from IANA "CCNx Hash Function Types"
   WrappedKey = T_WRAPPED_KEY LENGTH RsaOaepEnc{4OCTET 1*OCTET}
       ; Encrypted 4-byte salt plus AES key
```

Figure 3: FLIC Grammar

SecurityCtx: information about how to decrypt an EncryptedNode. The structure will depend on the specific encryption algorithm.

AEADMode "AEAD Algorithms" values from IANA "Authenticated Encryption with Associated Data (AEAD) Parameters." FLIC only specifies the use of four basic algorithms with full authentication tag.

AlgorithmId: The ID of the encryption method (e.g. preshared key, a broadcast encryption scheme, etc.)

AlgorithmData: The context for the encryption algorithm.

EncryptedNode: An opaque octet string with an optional authentication tag (i.e. for AEAD authentication tag)

Node: A plain-text manifest node. The structure allows for in-place encryption/decryption.

NodeData: the metadata about the Manifest node

SubtreeSize: The size of all application data at and below the Node or Group

SubtreeDigest: The cryptographic digest of all application data at and below the Node or Group

Locators: An array of routing hints to find the manifest components

HashGroup: A set of child pointers and associated metadata

Ptrs: A list of one or more Hash Values

Pad: The Node may include a Pad, as per Section 3.3.1 of [RFC8609]. This allows the publisher to hide the plaintext length in an encrypted manifest.

GroupData: Metadata that applies to a HashGroup

LeafSize: Size of all application data immediately under the Group (i.e. via direct pointers)

LeafDigest: Digest of all application data immediately under the Group

StartSegmentId: If using a name constructor that requires a chunk number (segment ID), this field indicates the starting value for the group. Using the StartSegmentId means that a consumer does not need to track the segment id between manifests and simplifies interest name generation.

Ptr: The ContentObjectHash of a child, which may be a data ContentObject (i.e. with Payload) or another Manifest Node.

3.7. Manifest Graphs

As mentioned in the introduction, FLIC uses a directed acyclic graphs that is not necessarily a tree. A tree requires a unique path to each node. FLIC allows multiple pointers to the same subtrees or application data.

The FLIC traversal rule for a given manifest is to visit each of its Hash Groups, in order, and within each Hash Group to visit each pointer, in order. It is a left-to-right reading of the pointers. We will use the term "child" for each pointer in a FLIC Node, where each child has an implicit order based on its position with a hash group, and the order of the hash groups within the node.

If the FLIC manifest is a tree, and all direct pointers occur before all indirect pointers, then the traversal is pre-order. This is the RECOMMENDED encoding order, as it minimizes the time to the first application data object (assuming one is not using leaf-only storage).

The pre-order encoding and decoding of a FLIC manifest works even if the manifest is not a tree. The [ccnpy] README file has examples of data de-duplication where both manifest nodes and application data nodes are de-duplicated and appear more than once in the traversal. There is no special encoding or decoding: it is all pre-order, but it does not build a tree.

3.7.1. Traversal Example

FLIC manifests use a pre-order traversal. This means pointers are read top to bottom, left to right. The algorithms in Figure 4 show the pre-order forward traversal code and the reverse-order traversal code, which we use below to construct such a graph. This document does not mandate how to build FLIC graphs. The function visit(node) means to enumerate the direct pointers of the node before the loop visits the indirect pointers. Appendix B provides a detailed example of building FLIC graphs.

```
 preorder(node)
    if (node = null)
       return
    visit(node)
    for (i = 0, i &lt; node.child.length, i++)
       preorder(node.child[i])

reverse_preorder(node)
   if (node = null)
       return
   for (i = node.child.length - 1, i &gt;= 0, i-- )
       reverse_preorder(node.child[i])
       visit(node)
```

Figure 4: Traversal Pseudocode

In terms of the FLIC grammar, one expands a node into its hash groups, visiting each hash group in order. In each hash group, one follows each pointer in order. Figure 5 shows how hash groups inside a manifest expand like virtual children in the tree. The in-order traversal is M0, HG1, M1, HG3, D0, D1, D2, HG2, D3, D4.

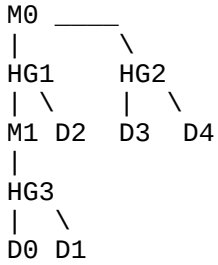


Figure 5: Node Expansio

Using the example manifest tree shown in Figure 25, the in-order traversal would be: Root, M0, M1, D0, D1, D2, M2, D3, D4, D5, M3, D6, D7, D8.

3.8. Manifest Encryption Modes

This document specifies two encryption modes. The first is a preshared key mode, where the parties are assumed to have the decryption keys already. It uses AES-GCM or AES-CCM. This is useful, for example, when using a key agreement protocol such as CCNxKE [I-D.wood-icnrg-ccnxkeyexchange]. The second is an RSA key wrapping mode using RSA-OAEP, similar to [RFC8017]. This may be used for group keying.

Additional modes may be defined in subsequent specifications. We expect that an RSA-KEM mode and Elliptic Curve mode should be specified.

To distribute a decryption key with the manifest itself, FLIC uses RSA-OAEP to wrap the WrappedKey element. This specification only details the pertinent aspects of the encryption. It describes how a consumer locates the appropriate keys in the ICN namespace. It does not specify aspects of a key manager which may be used as part of key distribution and management, nor does it specify the protocol between a key manager and a publisher. In its simplest form, the publisher could be the key manager, in which case there is no extra protocol needed between the publisher and key manager.

The manifest is always encrypted with an AEAD algorithm. The difference between the preshared key mode and the RSA wrapped mode is how the parties communicate the AES decryption key. The RSA OAEP mechanism facilitates the distribution of the shared key without an on-line key agreement protocol like (the expired draft) CCNxKE [I-D.wood-icnrg-ccnxkeyexchange].

3.8.1. AEAD Mode

This mechanism uses AES-GEM [NIST-800-38D] or AES-CCM [RFC3310] for manifest encryption. A publisher creating a SecurityCtx SHOULD use the mechanisms in [RFC6655] for AES-CCM IV generation and [RFC5288] for AES-GCM IV generation.

When the publisher and consumer establish the KeyNum, they SHOULD also establish a salt. This results in a 4-byte salt and 8-byte nonce. If no salt exists, the AEADNonce may be the entire IV.

The scope of the KeyNum is upto the publisher and consumer agreement. It may be for a specific manifest, or a specific session (e.g. a CCNxKE session), or some longer period. FLIC does not specify how the publisher and consumer agree on shared keys, their lifespan, or scope. See the Section 6 section for a discussion.

As these references specify, the publisher MUST use unique nonces on each Manifest that uses the same key or derived KDF key. A publisher MAY use the mechanisms of Section 3.2 of [RFC5116] for nonce generation (what we cal "salt" is termed "fixed-common" therein).

The AEAD Mode uses [RFC5116] defined symbols AEAD_AES_128_CCM, AEAD_AES_128_GCM, AEAD_AES_256_CCM and AEAH_AES_256_GCM to specify the key length and algorithm.

The KeyNum identifies a key on the receiver. The key MUST be exactly of the length specific by the Mode. Many receivers may have the same key with the same KeyNum.

The publisher MAY use a KDF for the manifest. If KDFData is present, the consumer MUST derive a manifest-specific key from the shared key identified by KeyNum. The KDFAlg is taken from the IANA registry for [RFC9180]. If KDFInfo is present, the consumer uses that string as in the KDF’s info parameter. If the KDFInfo is not present, then the consumer must use the Name TLV of the enclosing Content Object. This option is only possible for Segmented Schema name constructors, where each manifest has a unique name. If the publisher and consumer have agreed on a salt to be associated with the key for use in the AEAD algorithm, this salt should not be used in the KDF. The publisher and consumer may agree on a separate random salt as part of their key agreement on the KeyNum for use in the KDF.

Within a single manifest tree, a publisher MAY use the same KDFData and KDFInfo over multiple manifests, so long as it uses unique Nonce. This allows a consumer to cache a single derived key based on the (KeyNum, KDFInfo) pair. This is the normal use of the KDF mode: the publisher only needs to track a single unique value for each manifest

using KeyNum, not all the individual Nonces used with in a manifest tree. If the publisher uses a sequence number or timestamp, it does not even need to record previous KDFInfo, so long as they are unique.

When a Consumer reads a manifest that specifies a KeyNum, the consumer SHOULD verify that the Manifest's publisher is an expected one for the KeyNum's usage. This trust mechanism employed to ascertain whether the publisher is expected is beyond the scope of this document, but we provide an outline of one such possible trust mechanism. When a consumer learns a shared key and KeyNum, it associates that KeyNum with the publisher ID used in a public key signature. When the consumer receives a signed manifest (e.g. the root manifest of a manifest tree), the consumer matches the KeyNum's publisher with the Manifest's publisher.

Each encrypted manifest node has a full security context (KeyNum, Nonce, Mode). The AEAD decryption is independent for each manifest so Manifest objects can be fetched and decrypted in any order. This design also ensures that if a manifest tree points to the same subtree repeatedly, such as for deduplication, the decryptions are all idempotent.

If KDFData is present, one derives a manifest-specific key as:

1. If KDFInfo is present, the TLV (not just the value) is used as the Label below. Otherwise, the Content Object Name TLV (including type type and length fields) is used as the Label. If the content object does not have a name, it is an error.

2. FixedInfo = "FLIC" || KeyNum || AEADMode || Label, where KeyNum and AEADMode are the respective TLVs (including type and length).

3. If the publisher and consumer have agreed on a KDF-specific salt, it is used as the Salt. Otherwise, the KDF's default salt is used.

4. The derived key will use the left bits of the KDF output to satisfy the AEADMode key length (typically 128 or 256 bits).

5. DerivedKey = LEFT(KDF(Shared Key, FixedInfo, KDF Salt), bits).

To encrypt a Manifest, the publisher:

1. Removes any SecurityCtx or AuthTag from the Manifest.

2. Creates a SecurityCtx and adds it to the Manifest.

3. If KDFData is present, derives a key as above.

4. If the publisher and consumer shared a salt, it is used in the encryption.

5. Treats the Manifest TLV through the end of the Node TLV Length as unencrypted authenticated Header. That includes anything from the start of the Manifest up to but not including the start of the Node’s body.

6. Treats the body of the Node to the end of the Manifest as encrypted data.

7. Appends the AEAD AuthTag to the end of the Manifest, increasing the Manifest’s length

8. Changes the TLV type of the Node to EncryptedNode.

To decrypt a Manifest, the consumer:

1. Verifies that the KeyNum exists and the publisher is trusted for that KeyNum.

2. Saves the AuthTag and removes it from the Manifest, decreasing the Manifest length.

3. Changes the EncryptedNode type to Node.

4. Treats everything from the Manifest TLV through the end of the Node Length as unencrypted authenticated Header. That is, all bytes from the start of the Manifest up to but not including the start of the Node’s body.

5. Treats the body of the Node to the end of the Manifest as encrypted data.

6. If KDFData is present, derives a key as above.

7. If the publisher and consumer shared a salt, it is used in the decryption.

8. Verifies and decrypts the data using the key and saved AuthTag.

9. If the decryption fails, the consumer SHOULD notify the user and stop further processing of the manifest.

3.8.2. RSA-OAEP Key Transport Mode

The RsaOaepCtx> mode uses RSA-OAEP (see Section 7.1 of [RFC8017] and [NIST-800-56Br2]) to encrypt a symmetric key that is used to encrypt the Manifest. We call this RSA public key the Key Encryption Key (KEK) and the consumer has the private key. A separate key distribution system is responsible for distributing the KEK. The publisher MAY include a KeyLink to help a consumer find the corresponding private key, such as from a key distribution service. An ICN system may use the RSA-OAEP mode as part of a group keying protocol. Key distribution and group keying are both beyond the scope of FLIC.

This usage of RSA-OAEP is intended to be compliant with KTS-OAEP [NIST-800-56Br2]. See Section 6.4 for a discussion of compliance with NIST recommendations.

T_RSA_WRAP_CTX uses RSA-OAEP [RFC8017] with mask generation function MGF1 [RFC8017] and the specified hash algorithm. The hash algorithm number is taken from the IANA registry "CCNx Hash Function Types," which are all NIST approved hash functions.

The RsaOaepData includes an AEADData that indicates how to use the wrapped key to decrypt the manifest. It optionally includes the wrapped key itself. A publisher using RsaWrapCtx MUST include the optional wrapped key information in the first manifest node that uses it. The publisher MAY include the wrapped key in other manifest nodes. It MUST use distinct KeyNum for distinct WrappedKey.

The RSA-OAEP KeyNum is only significant within a given manifest.

The AEADData MAY include a KDFData element, in which case it is used the same was as with AEAD Mode.

In a typical usage, the publisher would begin encryption at the root manifest. It must include the full WrapedData here, as it is the first usage of the KeyNum. In subsequent manifests, the publisher may only include the AEADData in the RsaOaepData to save space.

The KeyId identifies the KEK. If the consumer already has the private key corresponding to KeyId locally, it does not need to fetch it from the KeyLink. The KeyId is similar to the Content Object KeyId field [RFC8609].

The symmetric key MUST be one that is compatible with the AEAD Mode, i.e. a 128-bit or 256-bit random number. Further, the symmetric key MUST fit in the OAEP envelope. Note that for a 128-bit key (16 bytes) plus a 4-byte salt requires a minimum 688-bit RSA key and a

256-bit key requires a minimum 816-bit RSA key. So in practical terms, a 1024-bit RSA key is the minimum for the RSA-OAEP encryption with a SHA-256 hash algorithm (it is recommended to use at least 2048-bit RSA keys [NIST-800-131Ar2]).

Any group key protocol and system needed are outside the scope of this document. We assume there is a Key Manager (KM) and a Publisher (P) and a set of group members. Through some means, the Publisher therefore has at its disposal:

* A Content Encryption Key (CEK), i.e. the symmetric key.

* The RSA-OAEP wrapped CEK.

* The KeyId of the KEK used to wrap the CEK.

* The Locator of the KEK, which is shared under some group key protocol.

This specification requires that if a group member fetches the KEK key at Locator, it can decrypt the WrappedKey and retrieve the CEK. The WrappedKey includes a 4-byte salt plus an AES key. The 4-byte salt is used by the AEAD algorithm as part of the IV, and MUST remain the same for the given KeyNum.

In one example, a publisher could request a key for a group and the Key Manager could securely communicate (CEK, Wapped_CEK, KeyId, KeyLink) back to the publisher. The Key Manager is responsible for publishing the link. In another example, the publisher could be a group member and have a group private key in which case the publisher can create their own key encryption key, publish it under the Locator and proceed. The publisher generates CEK, Wrapped_CEK, KeyId, and a KeyLink on its own.

The RSA-OAEP "label" ("additional information" in [NIST-800-131Ar2]) is constructed as shown in Figure 6. It is the concatenation of several TLVs from the RsaOaepCtx. The AEADData, KeyId, KeyLink, HashAlg TLVs are from the RsaOaepWrapper field. The label is hashed via the HashAlg, so the length of the label does not affect the size of the encryption. To tightly bind the encryption to the manifest, the publisher SHOULD choose a unique KeyNum and KDFInfo pair.

```
AdditionalInfo = "FLIC RSA-OAEP" ||
                 KeyNum || AeadMode || [KDFData] ||
                 KeyID || HashAlg ||
                 KeyLink (if present in RsaOaepWrapper)
```

Figure 6: OAEP Label

To create the wrapped key using a Key Encryption Key:

1. Obtain the CEK in binary format (e.g. 32 bytes for 256 bits)

2. RSA encrypt the WrappedKey value using the KEK public key with OAEP padding and the OAEP label (additional information), following [NIST-800-131Ar2]. The encryption is not signed or authenticated because the root Manifest must have been signed by the publisher already. Or the encapsulating Content Object (Data Object) is signed via ICN mechanisms.

To decrypt the wrapped key using a Key Encryption Key:

1. RSA decrypt the WrappedKey value using the KEK private key with OAEP padding and OAEP label, following [NIST-800-131Ar2].

2. Verify the unwrapped key is a valid length for the AEADMode.

To encrypt a Manifest, the publisher:

1. Acquires the set of (CEK, salt, WrappedKey, KeyId, Locator). The publisher may generate these values itself, or acquire them from a key manager.

2. Creates a SecurityCtx and adds it to the Manifest. The SecurityCtx includes an AEADNonce and AEADMode, as per AEAD mode.

3. It is RECOMMENDED to also include the use of a KDF in the AEADData .

4. Encrypts the Manifest as per AEAD Mode using the RSA-OAEP SecurityCtx and CEK.

To decrypt a Manifest, the consumer:

1. Acquires the RSA private key corresponding to KeyId via the KeyLink or whatever means the publisher and consumers use to distribute keys.

2. It SHOULD verify that it trusts the Manifest publisher to use the provided wrapping key.

3. Decrypts the WrappedKey to get the CEK and salt. If the consumer has already decrypted the same exact WrappedKey TLV block, it may use that cached CEK and salt. The same wrapping key is determined by the publisher’s signing KeyId, the wrapping key KeyId, and the AEADData.

4. Using the CEK and AEADData, decrypt the Manifest as per AEAD Mode.

3.9. Protocol Encodings

3.9.1. CCNx Encoding

The CCNx type values are defined in Section 5. The section Section 3.9.1.5 goes over every TLV structure.

In CCNx, application data content objects use a PayloadType of T_PAYLOADTYPE_DATA. In order to clearly distinguish FLIC Manifests from application data, a different payload type is required. Therefore this specification defines a new payload type of T_PAYLOADTYPE_MANIFEST.

```
ManifestContentObject = TYPE LENGTH [Name] [ExpiryTime]
                        PayloadType Payload
Name = TYPE LENGTH *OCTET ; As per RFC8569
ExpiryTime = TYPE LENGTH *OCTET ; As per RFC8569
PayloadType = TYPE LENGTH T_PAYLOADTYPE_MANIFEST ; Value TBD
Payload : TYPE LENGTH *OCTET ; the serialized Manifest object
```

Figure 7: CCNx Embedding Grammar

3.9.1.1. CCNx Hash Naming Strategy

The Hash Naming Strategy uses CCNx nameless content objects and the HashSchema name constructor. This means that only the Root Manifest should have a name embedded in the Content object. All other are CCNx nameless objects. The Manifest should provide a set of Locators that the client may use to form the Interests.

It proceeds as follows:

* The Root Manifest content object bound to a name assigned by the publisher and signed by the publisher. It also may have a set of Locators used to fetch the remainder of the manifest. The root manifest has a single HashPointer that points to the Top Manifest. It may also have cache control directives, such as ExpiryTime.

* The Root Manifest has an NsDef that specifies HashSchema. Its GroupData uses that NsId. All internal and leaf manifests use the same GroupData NsId. A Manifest Tree MAY omit the NsDef and NsId elements and rely on the default being HashSchema.

* The Top Manifest is a nameless CCNx content object. It may have cache control directives.

* Internal and Leaf manifests are nameless CCNx content objects, possibly with cache control directives.

* The Data content objects are nameless CCNx content objects, possibly with cache control directives.

* To form an Interest for a direct or indirect pointer, use a Name from one of the Locators and put the pointer HashValue into the ContentObjectHashRestriction.

3.9.1.2. CCNx Single Prefix Strategy

The Single Prefix strategy uses a named Root manifest and then all other data and sub-manifest objects use the same Name. They are differentiated only by their hash. The single prefix name should be different than the root manifest name, to not confuse them, as one may wish to use a discovery mechanism on the root manifest and fetch it without a hash restriction.

It proceeds as follows:

* The Root Manifest content object has a name used to fetch the manifest. It is signed by the publisher. It has a single Locator used to fetch the remainder of the manifest using the common Single Prefix name. It has a single HashPointer that points to the Top Manifest. It may also have cache control directives, such as ExpiryTime.

* The Root Manifest has an NsDef that specifies PrefixSchema with the Locator for the single prefix.

* The Top Manifest has the name SinglePrefixName. It may have cache control directives. Its GroupData elements must have an NsId that references the NsDef.

* An Internal or Leaf manifest has the name SinglePrefixName, possibly with cache control directives. Its GroupData elements must have an NsId that references the NsDef.

* The Data content objects have the name SinglePrefixName, possibly with cache control directives.

* To form an Interest for a direct or indirect pointer, use SinglePrefixName as the Name and put the pointer HashValue into the ContentObjectHashRestriction.

3.9.1.3. CCNx Segmented Prefix Strategy

It proceeds as follows:

* The Root Manifest content object has a name used to fetch the manifest. It is signed by the publisher. It has a set of Locators used to fetch the remainder of the manifest. It has a single HashPointer that points to the Top Manifest. It may also have cache control directives, such as ExpiryTime.

* The Root Manifest has an NsDef that specifies SegmentedPrefix and the SegmentedPrefixSchema element specifies the SegmentedPrefixName.

* The publisher tracks the chunk number of each content object within the NsId. Objects are be numbered in their traversal order. Within each manifest, the name can be constructed from the SegmentedPrefixName plus a Chunk name component.

* The Top Manifest has the name SegmentedPrefixName plus chunk number. It may have cache control directives. It's GroupData elements must have an NsId that references the NsDef.

* An Internal or Leaf manifest has the name SegmentedPrefixName plus chunk number, possibly with cache control directives. Its GroupData elements must have an NsId that references the NsDef.

* The Data content objects have the name SegmentedPrefixName plus chunk number, possibly with cache control directives.

* To form an Interest for a direct or indirect pointer, use SegmentedPrefixName plus chunk number as the Name and put the pointer HashValue into the ContentObjectHashRestriction. A consumer must track the chunk number in traversal order for each SegmentedPrefixSchema NsId.

3.9.1.4. CCNx Hybrid Strategy

A manifest may use multiple schemas. For example, the application payload in data content objects might use SegmentedPrefix while the manifest content objects might use HashNaming.

The Root Manifest should specify an NsDef with a first NsId (say 1) as the HashNaming schema and a second NsDef with a second NsId (say 2) as the SegmentedPrefix schema along with the SegmentedPrefixName.

Each manifest (Top, Internal, Leaf) uses two or more HashGroups, where each HashGroup has only Direct (with the second NsId) or Indirect (with the first NsId). The number of hash groups will depend on how the publisher wishes to interleave direct and indirect pointers.

Manifests and data objects derive their names according to the application's naming schema.

3.9.1.5. CCNx TLV Encodings

This section defines each CCNx TLV encoding and the format of the TLV values.

FLIC uses variable length integer encodings in several places. We use the term varint to be a network byte order unsigned integer of length 1 - 8 octets. This applies to all fields defined as "INTEGER" in the grammar: T_SUBTREE_SIZE, T_LEAF_SIZE, T_NCID, T_KEYNUM.

FLIC encodes hash values as per Section 3.3.3 of [RFC8609]. That is, as a hash algorithm type, the length, and the hash output in the value. This applies to any field in the grammar defined as "HashValue": T_SUBTREE_DIGEST, T_LEAF_DIGEST, T_PTRS (an array of HashValue), T_PTR, T_KEYID.

The CCNx encoding allows for vendor extensions (T_ORG) in several places, as per Section 3.3.2 of [RFC8609]. A parser may skip over any T_ORG it does not implement. Where appropriate, we show an optional T_ORG tlv where allowed.

The CCNx encoding allows for experimental type identifiers (0x1000-0x1FFF) in most TLV contexts. A parser may skip over any experimental TLV it does not implement. We do not show experimental TLVs in the artwork below.

There is no required order of TLV blocks within a TLV context. We show the RECOMMENDED ordering of TLV blocks. TLV blocks MUST NOT repeat unless specifically allowed in the grammar (Section 3.6).

3.9.1.5.1. Manifest TLV

A Manifest has an outer TLV type that identifies type kind of manifest inside a Content Object with a payload type of T_PAYLOADTYPE_MANIFEST. FLIC is defined as type T_FLIC_MANIFEST.

A FLIC Manifest may be plaintext or encrypted. A plaintext manifest usually only has a Node. But it may also have a Security Context, Node, and Authentication Tag. This can happen when a manifest is in-

place decrypted. In place decryption operates on the body of the EncryptedNode TLV and then changes the type from T_ENCRYPTED_NODE to T_NODE.

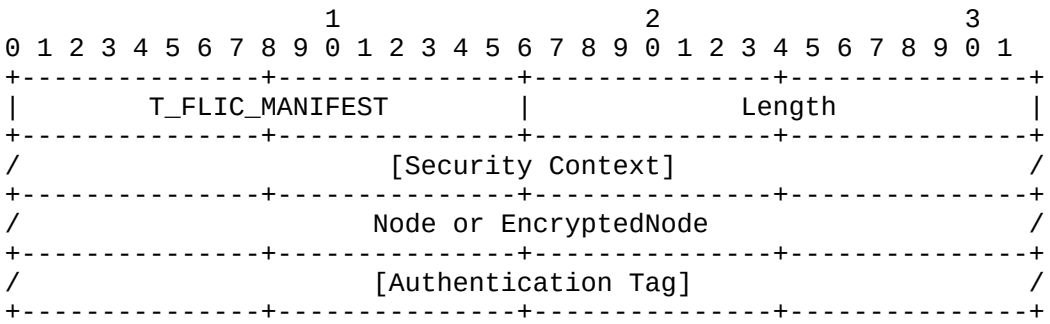


Figure 8: FLIC manifest TLV container

3.9.1.5.2. SecurityCtx TLV

The SecurityCtx TLV wraps an AlgroithmCtx, which may be either a symmetric key encryption context AEADCtx or an asymmetric key wrapping of the decryption key, RsaWrapCtx.

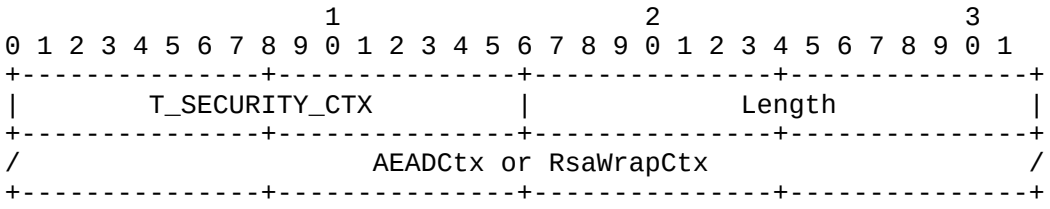


Figure 9: FLIC Security Context TLV container

3.9.1.5.3. AEAD Context TLV

The AEADCtx identifies a symmetric key shared by the publisher and consumer. It is identified by a KeyNum, which may be an small integer (e.g. within a TLS-like session) or a longer integer. CCNx uses up to a 64-bit integer.

The AEADNonce is used to generate the encryption IV, as described in Section 3.8.1. It is an octet string.

The AEADMode is an integer that identifies the encryption algorithm. All defined values fit within 1 octet. The IANA registry "AEAD Algorithms" allows for up to 2 octets.

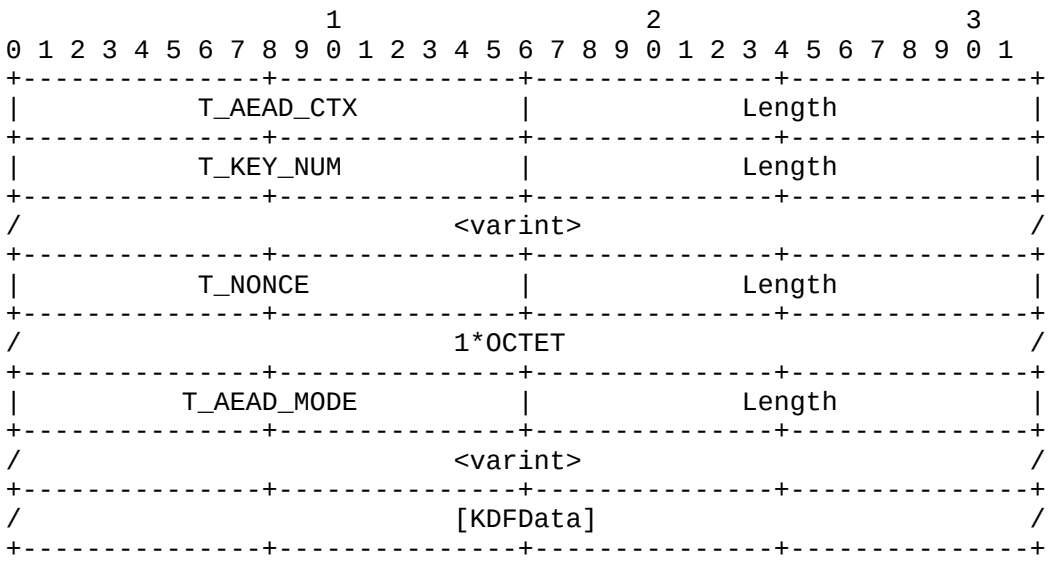


Figure 10: FLIC AEAD Context TLV container

3.9.1.5.4. RSA-OAEP Context TLV

The RsaOaepData has a mandatory AEADData section plus optional RsaOaepData. Note that in the grammar, those are not TLV containers, but simply data records. The figure below shows an RsaOaepCtx with all parameters.

The HashAlg value is an unsigned 16-bit integer in network byte order.

The wrapped key value is the RSA-OAEP ciphertext of the WrappedKey record.

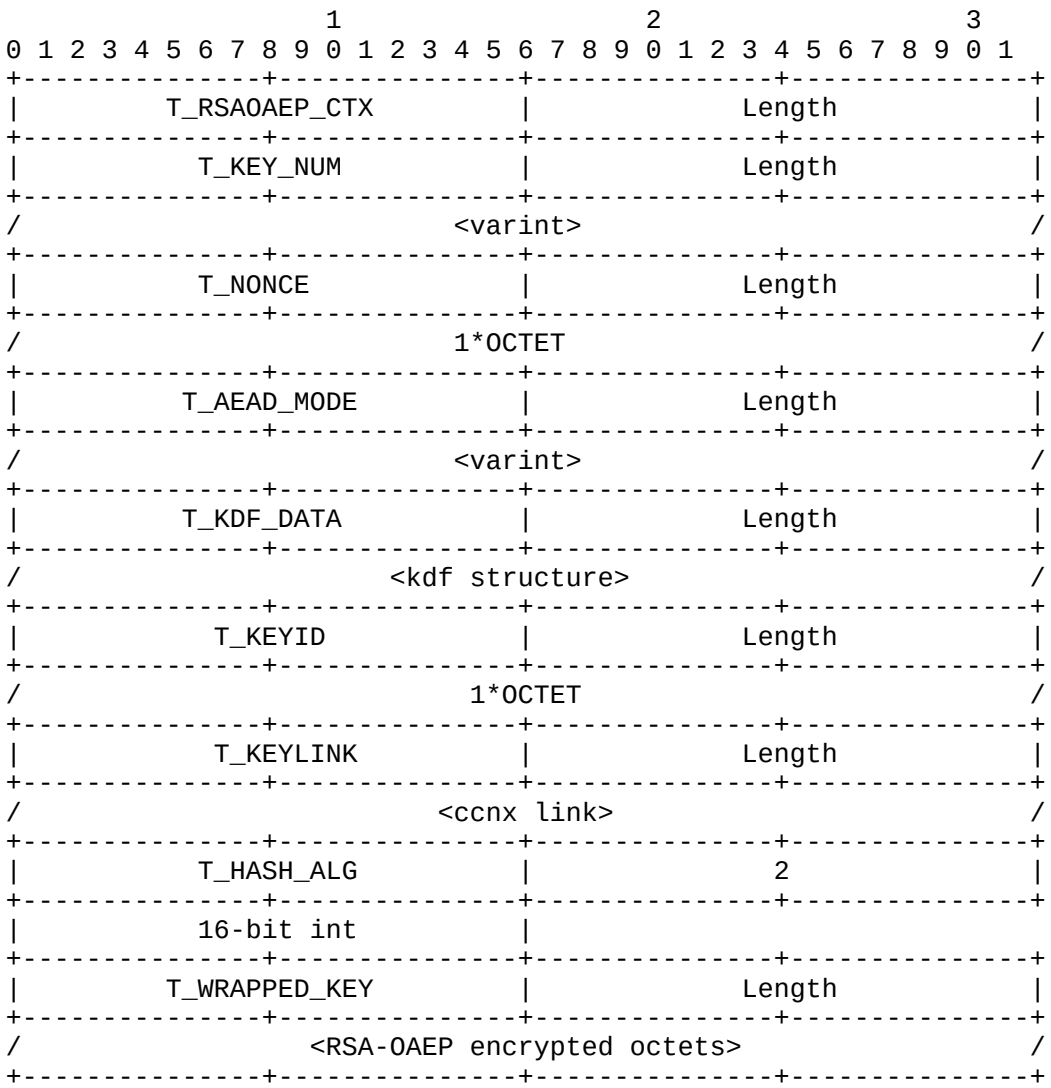


Figure 11: FLIC RSA-OAEP TLV container

3.9.1.5.5. Node TLV

The Node contains the manifest information. It has an optional NodeData, plus one or more hash groups.

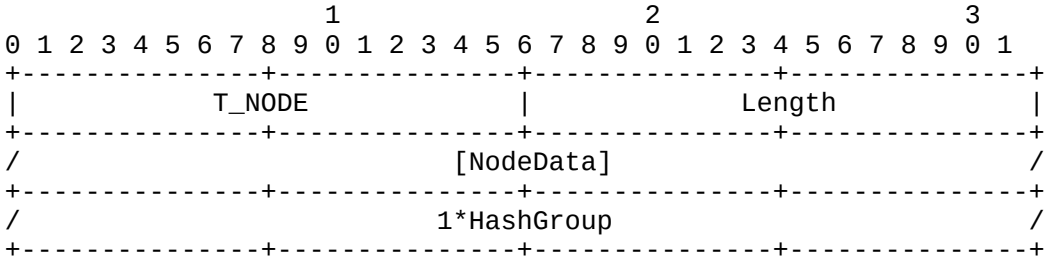


Figure 12: FLIC Node TLV container

3.9.1.5.6. Node Data TLV

NodeData has no mandatory fields, but several optional fields. If no fields are present, a publisher should omit the entire NodeData TLV container.

SubtreeSize, SubtreeDigest, and Vendor are encoded as previously specified (INTEGER, HashValue, and T_ORG, respectively).

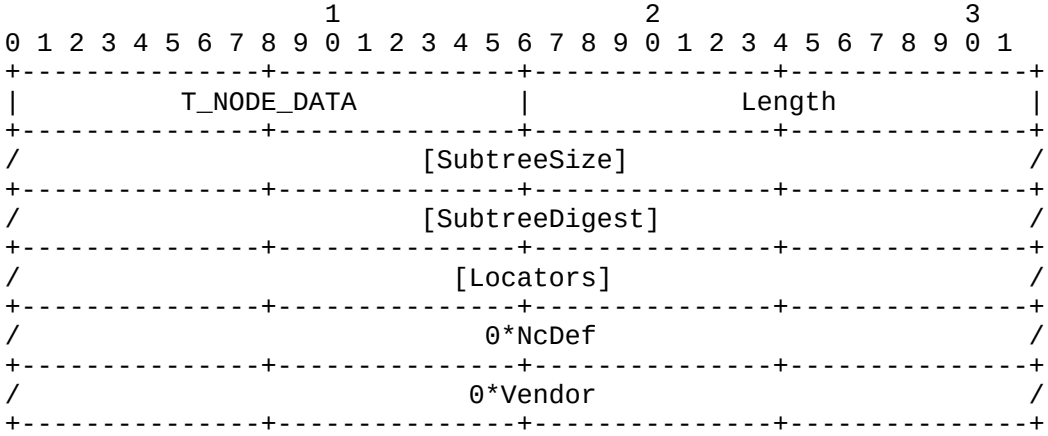


Figure 13: FLIC NodeData TLV container

3.9.1.5.7. Locators TLV

Locators is a TLV container around a list s Locator entries. Each Locator wraps a CCN x Link Section 3.3.4 of [RFC8609]. In the CCNx draft, "Link" does not have its own TLV type, but rather is embedded inside a type with specific meaning, such as KeyLink or PublicKeyLocator.

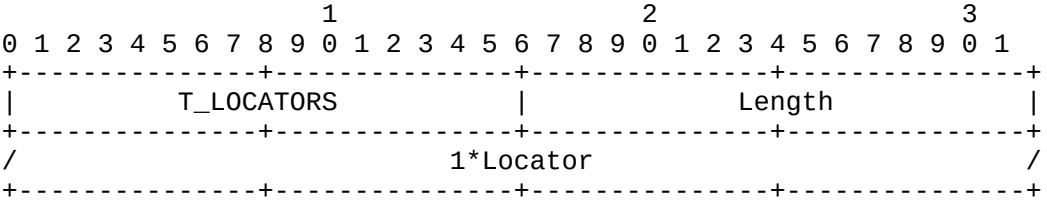


Figure 14: FLIC Locators TLV container

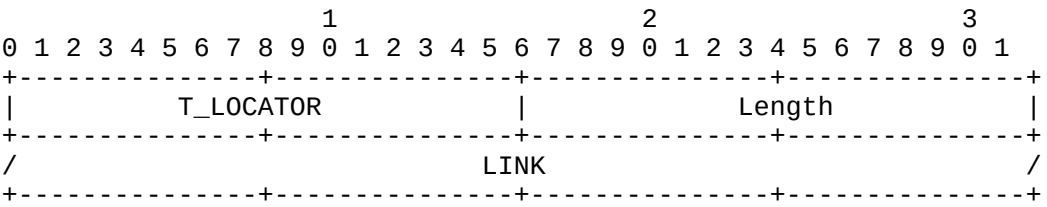


Figure 15: FLIC Locator TLV container

3.9.1.5.8. Name Constructor Definition TLV

NcDef provides the information needed for a consumer to construct an Interest from a manifest pointer. The information is in a Name Constructor Schema.

NcId is encoded as an INTEGER, as previously defined.

NcSchema may include ProtocolFlags, which are TLVs intended to be included in an Interest. These may control discovery or response freshness. The value of protocol flags is ICN specific.

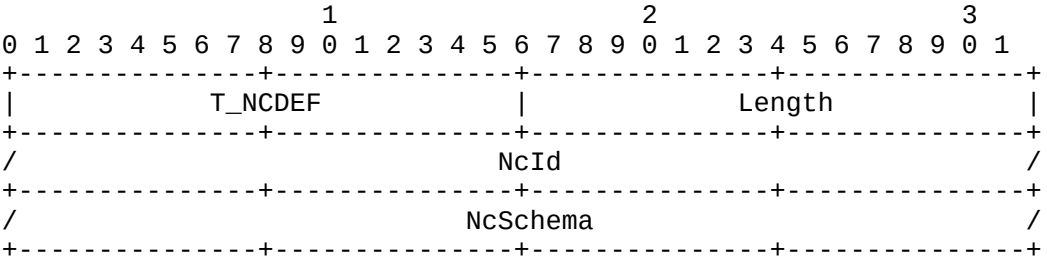


Figure 16: FLIC Name Constructor Definition TLV container

3.9.1.5.9. Hash Schema TLV

The HashSchema uses CCNx nameless objects. It must have one or more locators.

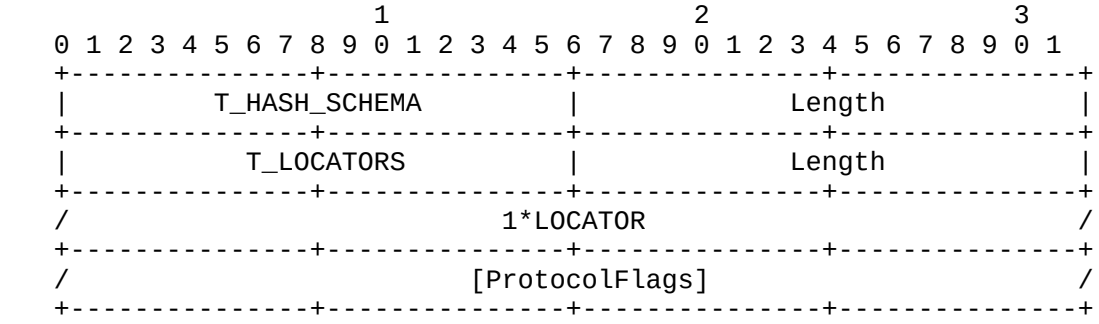


Figure 17: FLIC Hash Schema TLV container

3.9.1.5.10. Prefix Schema TLV

The PrefixSchema uses named content objects. CCNx uses the provided name and does not use locators.

The Name is encoded as Section 3.6.1 of [RFC8609].

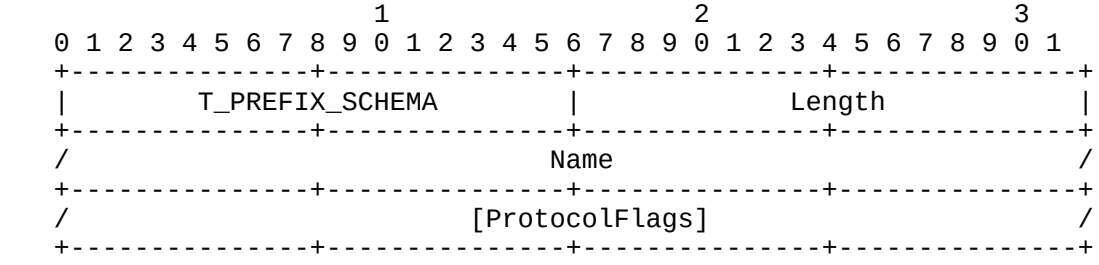


Figure 18: FLIC Prefix Schema TLV container

3.9.1.5.11. Segmented Schema TLV

The SegmentedSchema uses named content objects. Each object has a name prefix plus a unique suffix. The suffix is encoded with NameComponent type SuffixComponentType. CCNx uses the provided name and does not use locators.

The Name is encoded as Section 3.6.1 of [RFC8609].

In CCNx, the SuffixComponentType must be 2 octets, as per the CCNx TLV type size.

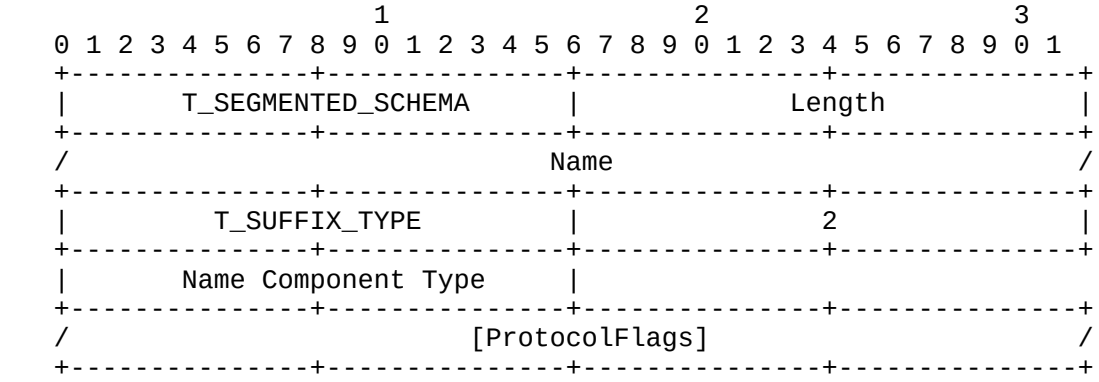


Figure 19: FLIC Segmented Schema TLV container

3.9.1.5.12. Hash Group TLV

A HashGroup encloses a series of hash values. It has an optional GroupData that applies to all the pointers in the group.

The pointers are bundled in either a Ptrs list or an AnnotatedPtrs list.

Ptrs is an array of raw CCNx hash values entries (i.e. SHA256 TLVs), as per Section 3.3.3 of [RFC8609]. Each entry MAY be a different hash type and different TLV length, so a parser must use the hash TLV type and TLV length fields of each entry.

AnotatedPtrs is a list of annotations for each pointer.

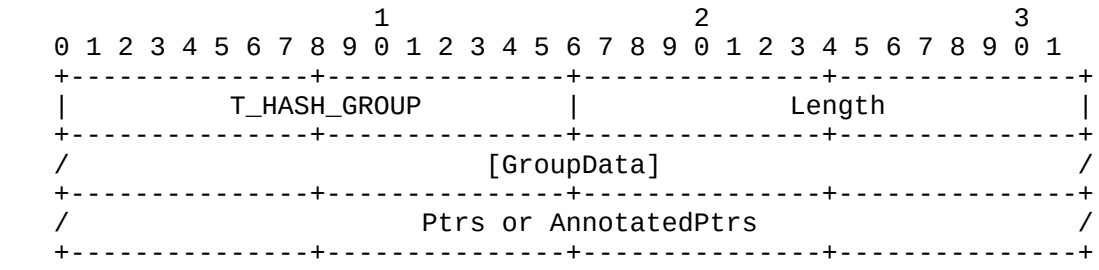


Figure 20: FLIC Hash Group TLV container

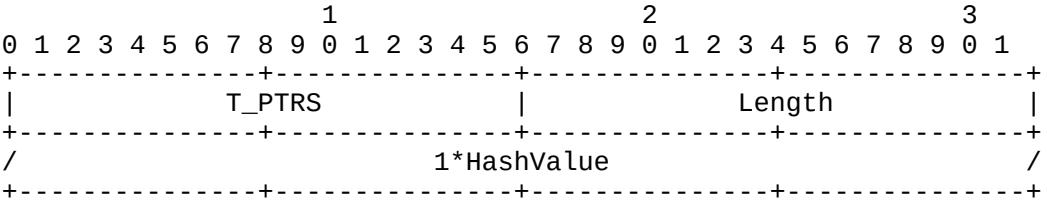


Figure 21: FLIC Ptrs TLV container

3.9.1.5.13. Group Data TLV

The GroupData is a set of optional fields. The encoding of these fields has been previously described (NcId, INTEGER, HashValue, T_ORG).

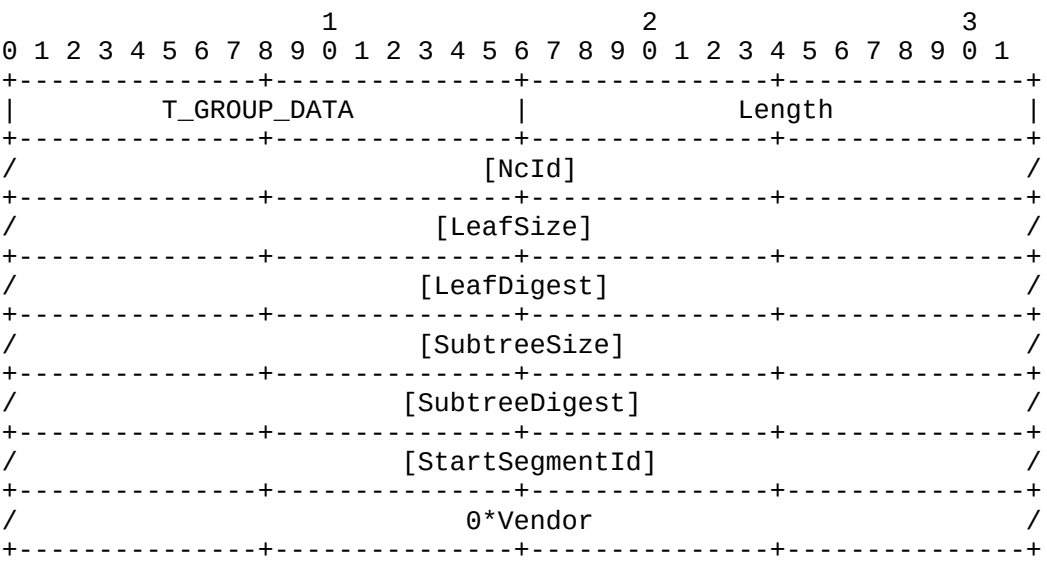


Figure 22: FLIC Group Data TLV container

3.9.1.5.14. Annotated Pointers TLV

AnnotatedPtrs is a list of one or more PointerBlocks. Each pointer block is one pointer plus zero or more annotations.

Annotations should not repeat in the same pointer block, unless specifically allowed by the annotation.

The defined annotations (SizeAnnotation, SegmentIdAnnotation, and LinkAnnotation) all use previously defined encodings (INTEGER or Name) and are not shown here.

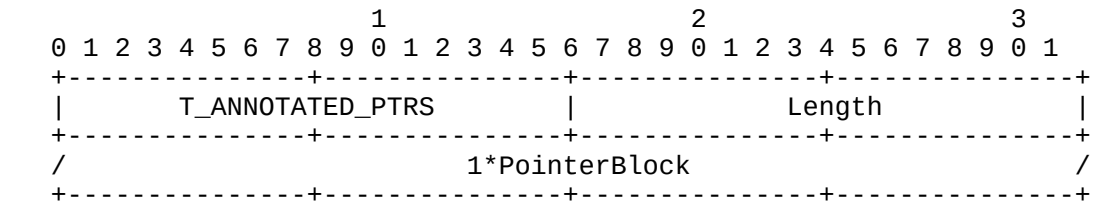


Figure 23: FLIC Annotated Pointers TLV container

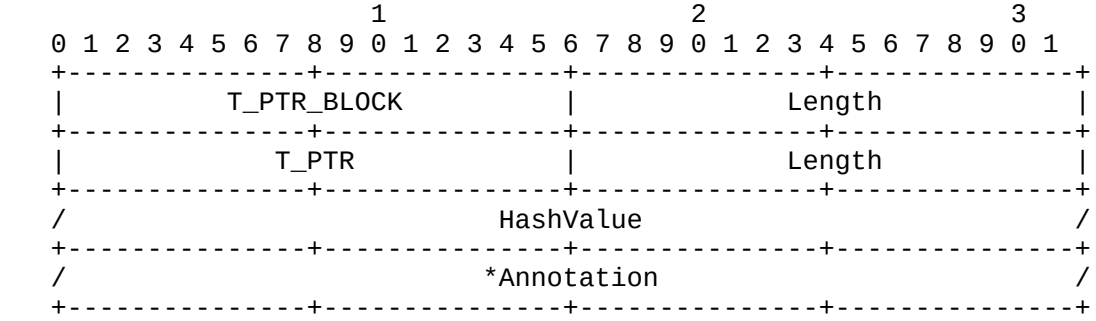


Figure 24: FLIC Pointer Block TLV container

3.9.2. NDN Encoding

In NDN, all Manifest Data objects use a ContentType of FLIC (1024), while all application data content objects use a PayloadType of Blob.

3.9.2.1. NDN Hash Naming

In NDN Hash Naming, a Data Object has a 0-length name. This means that an Interest will only have an ImplicitDigest name component in it. This method relies on using NDN Forwarding Hints.

It proceeds as follows:

* The Root Manifest Data has a name used to fetch the manifest. It is signed by the publisher. It has a set of Locators used to fetch the remainder of the manifest. It has a single HashPointer that points to the Top Manifest. It may also have cache control directives.

* The Root Manifest has an NsDef that specifies HashSchema. Its GroupData uses that NsId. All internal and leaf manifests use the same GroupData NsId. A Manifest Tree MAY omit the NsDef and NsId elements and rely on the default being HashSchema.

* The Top Manifest has a 0-length Name. It may have cache control directives.

* Internal and Leaf manifests has a 0-length Name, possibly with cache control directives.

* The application Data use a 0-length name, possibly with cache control directives.

* To form an Interest for a direct or indirect pointer, the name is only the Implicit Digest name component derived from a pointer's HashValue. The ForwardingHints come from the Locators. In NDN, one may use one or more locators within a single Interest.

3.9.2.2. NDN Single Prefix

In Single Prefix, the Data name is a common prefix used between all objects in that namespace, without a Segment or other counter. They are distinguished via the Implicit Digest name component. The FLIC Locators go in the ForwardingHints.

It proceeds as follows:

* The Root Manifest Data object has a name used to fetch the manifest. It is signed by the publisher. It has a set of Locators used to fetch the remainder of the manifest. It has a single HashPointer that points to the Top Manifest. It may also have cache control directives.

* The Root Manifest has an NsDef that specifies SinglePrefix and the SinglePrefixSchema element specifies the SinglePrefixName.

* The Top Manifest has the name SinglePrefixName. It may have cache control directives. Its GroupData elements must have an NsId that references the NsDef.

* An Internal or Leaf manifest has the name SinglePrefixName, possibly with cache control directives. Its GroupData elements must have an NsId that references the NsDef.

* The Data content objects have the name SinglePrefixName, possibly with cache control directives.

* To form an Interest for a direct or indirect pointer, use SinglePrefixName as the Name and append the pointer's HashValue into an ImplicitDigest name component. Set the ForwardingHints from the FLIC locators.

3.9.2.3. NDN Segmented Prefix

In Segmented Prefix, the Data name is a common prefix plus a segment number, so each manifest or application data object has a unique full name before the implicit digest. This means the consumer must maintain a counter for each SegmentedPrefix namespace.

> | *Optional*: Use AnnotatedPointers to indicate the segment number of each hash pointer to avoid needing to infer the segment numbers.
> |
> | *Optional*: Use StartSegmentId in GroupData to indicate the segment number of for each group. The producer must ensure that each subsequent GroupData starts at the correct offset.

It proceeds as follows:

* The Root Manifest Data object has a name used to fetch the manifest. It is signed by the publisher. It has a set of Locators used to fetch the remainder of the manifest. It has a single HashPointer that points to the Top Manifest. It may also have cache control directives.

* The Root Manifest has an NsDef that specifies SegmentedPrefix and the SegmentedPrefixSchema element specifies the SegmentedPrefixName.

* The publisher tracks the segment number of each Data object within a SegmentedPrefix NsId. Data is numbered in traversal order. Within each manifest, the name is constructed from the SegmentedPrefixName plus a Segment name component.

* The Top Manifest has the name SegmentedPrefixName plus segment number. It may have cache control directives. Its GroupData elements must have an NsId that references the NsDef.

* An Internal or Leaf manifest has the name SegmentedPrefixName plus segment number, possibly with cache control directives. Its GroupData elements must have an NsId that references the NsDef.

* The Data content objects have the name SegmentedPrefixName plus chunk number, possibly with cache control directives.

* To form an Interest for a direct or indirect pointer, use SegmentedPrefixName plus segment number as the Name and put the pointer HashValue into the ImplicitDigest name component. A consumer must track the segment number in traversal order for each SegmentedPrefixSchema NsId.

3.9.2.4. NDN Hybrid Schema

A manifest may use multiple schemas. For example, the application payload in data content objects might use SegmentedPrefix while the manifest content objects might use HashNaming.

The Root Manifest should specify an NsDef with a first NsId (say 1) as the HashNaming schema and a second NsDef with a second NsId (say 2) as the SegmentedPrefix schema along with the SegmentedPrefixName.

Each manifest (Top, Internal, Leaf) uses two or more HashGroups, where each HashGroup has only Direct (with the second NsId) or Indirect (with the first NsId). The number of hash groups will depend on how the publisher wishes to interleave direct and indirect pointers.

Manifests and data objects derive their names according to the application’s naming schema.

3.10. Example Structures

3.10.1. Leaf-only data

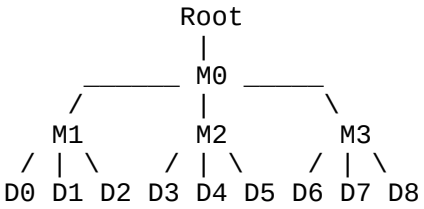


Figure 25: Leaf-only manifest tree

3.10.2. Linear (chain)

Of special interest are "skewed trees" where a pointer to a manifest may only appear as last pointer of (sub-) manifests. Such a tree becomes a sequential list of manifests with a maximum of data pointers per manifest packet. Beside the tree shape we also show this data structure in form of packet content where D stands for a data pointer and M is the hash of a manifest packet.

```
Root -> M0 ----> M1 ----> ...
|->DDDD  |->DDDD  |- ...
```

4. Experimenting with FLIC

FLIC is expected to enable a number of salient experiments in the use of ICN protocols by applications. These experiments will help not only to inform the desirable structure of ICN applications but reflect back to the features included in FLIC to evaluate their usefulness to those applications. While many interesting design aspects of FLIC remain to be discovered through experience, a number of important questions to be answered through experimentation include:

* use for just files or other collections like directories
* use for particular applications, like streaming media manifests
* utility of pointer annotations to optimize retrieval
* utility of the encryption options for use by repositories and forwarders
* need for application metadata in manifests

5. IANA Considerations

FLIC adds two values to existing registries and defines two new registry in the "Content-Centric Networking (CCNx)" group.

The FLIC registries follow the CCNx convention of reserving the namespace 0x1000 - 0x1FFF for experimental use. It also uses the T_ORG (0x0FFF) definition for organizational (vendor) extensions in the NodeData and GroupData fields.

IANA will assign value "0x0004" to "T_MANIFEST_ID" in the "CCNx Name Segment Types" registry.

IANA will assign value "0x0003" to "T_PAYLOADTYPE_MANIFEST" in the "CCNx Payload Types" registry.

IANA will add the following new registry to the "Content-Centric Networking (CCNx)" registry group at https://www.iana.org/assignments/ccnx):

* Registry Name: CCNx Manifest Type
* Registration Procedure: Specification Required
* This defines the TLV type of a Manifest inside a CCNx Payload.

Type	Name	Reference
0x0000	T_FLIC_MANIFEST	
0x0001-0x0FFF	Unassigned	
0x1000-0x1FFF	Reserved for experimental use	
0x2000-0xFFFF	Unassigned	

Table 1: Manifest Type

IANA will add the following new registry and sub-registries to the "Content-Centric Networking (CCNx)" registry group at https://www.iana.org/assignments/ccnx):

* Registry Name: FLIC Manifest
* Registration Procedure: Specification Required

Type	Name	Reference
0x0000	T_SECURITY_CTX	
0x0001	T_NODE	
0x0002	T_ENCRYPTED_NODE	
0x0003	T_AUTH_TAG	
0x0004-0x0FFF	Unassigned	
0x1000-0x1FFF	Reserved for experimental use	
0x2000-0xFFFF	Unassigned	

Table 2: FLIC Manifest

* Sub-registry Name: FLIC Manifest (SecurityCtx)
* Registration Procedure: Specification Required

Type	Name	Reference
0x0000	T_AEAD_CTX	
0x0001	T_RSAOAEP_CTX	
0x0002-0x0FFF	Unassigned	
0x1000-0x1FFF	Reserved for experimental use	
0x2000-0xFFFF	Unassigned	

Table 3: FLIC Manifest (SecurityCtx)

* Sub-registry Name: FLIC Manifest (Node)
* Registration Procedure: Specification Required

Type	Name	Reference
0x0000	T_NODE_DATA	
0x0001	T_HASH_GROUP	
0x0002	T_SUBTREE_SIZE	
0x0003	T_SUBTREE_DIGEST	Value is in CCNx Hash Format Section 3.3.3 of [RFC8609]
0x0004	T_NCDEF	
0x0005	T_NCID	
0x0006	T_LOCATORS	
0x0007	T_PTRS	Value is array of CCNx Hash Format Section 3.3.3 of [RFC8609]
0x0008	T_ANNOTATED_PTRS	
0x0009	T_PTR_BLOCK	
0x000A	T_PTR	Value is in CCNx Hash Format Section 3.3.3 of [RFC8609]

0x000B	T_GROUP_DATA	
0x000C-0x000F	Unassigned	
0x0010	T_HashSchema	
0x0011	T_PrefixSchema	
0x0012	T_SegmentedSchema	
0x0013-0x0FFD	Unassigned	
0x0FFE	T_PAD	Section 3.3.1 of [RFC8609]
0x0FFF	T_ORG	Section 3.3.2 of [RFC8609]
0x1000-0x1FFF	Reserved for experimental use	
0x2000-0xFFFF	Unassigned	

Table 4: FLIC Manifest (Node)

* Sub-registry Name: FLIC Manifest (NcSchema)
* Registration Procedure: Specification Required

Type	Name	Reference
0x0000	T_NAME	[RFC8609]
0x0001	T_PROTOCOL_FLAGS	
0x0002	T_SUFFIX_TYPE	
0x0003-0x0005	Unassigned	
0x0006	T_LOCATORS	
0x0007-0x000C	Unassigned	
0x000D	T_LINK	Value is a Link Section 3.3.4 of [RFC8609]
0x000E-0x0FFF	Unassigned	
0x1000-0x1FFF	Reserved for experimental use	
0x2000-0xFFFF	Unassigned	

Table 5: FLIC Manifest (NcSchema)

* Sub-registry Name: FLIC Manifest (Group Data)
* Registration Procedure: Expert Review

Type	Name	Reference
0x0000	T_LEAF_SIZE	
0x0001	T_LEAF_DIGEST	Value is a CCNx Hash Format Section 3.3.3 of [RFC8609]
0x0002	T_SUBTREE_SIZE	
0x0003	T_SUBTREE_DIGEST	Value is in CCNx Hash Format Section 3.3.3 of [RFC8609]
0x0004	T_START_SEGMENT_ID	
0x0005-0x0FFE	Unassigned	
0x0FFF	T_ORG	Section 3.3.2 of [RFC8609]
0x1000-0x1FFF	Reserved for experimental use	
0x2000-0xFFFF	Unassigned	

Table 6: FLIC Manifest (Group Data)

* Sub-registry Name: FLIC Manifest (SecurityCtx Parameters)
* Registration Procedure: Expert Review

Type	Name	Reference
0x0000	T_KEYNUM	
0x0001	T_NONCE	
0x0002	T_AEADMode	Value a per IANA "AEAD Algorithms" [RFC5116]
0x0003	T_HASH_ALG	
0x0004	T_WRAPPED_KEY	
0x0005	T_KDF_DATA	
0x0006	T_KDF_ALG	Value as per IANA "HPKE KDF Identifiers" [RFC9180]
0x0007	T_KDF_INFO	
0x0008	Unassigned	
0x0009	T_KEYID	As per Section 3.6.4.1.4 of [RFC8609]
0x000A-0x000D	Unassigned	
0x000E	T_KEYLINK	As per Section 3.6.4.1.4 of [RFC8609]
0x000F-0x0FFF	Unassigned	
0x1000-0x1FFF	Reserved for experimental use	
0x2000-0xFFFF	Unassigned	

Table 7: FLIC Manifest (SecurityCtx Parameters)

* Sub-registry Name: FLIC Manifest (Annotation)
* Registration Procedure: Expert Review

Type	Name	Reference
0x0000	T_ANN_SIZE	
0x0001	T_ANN_SEGMENT_ID	
0x0003-0x000C	Unassigned	
0x000D	T_LINK	Value is a Link Section 3.3.4 of [RFC8609]
0x000E-0x0FFE	Unassigned	
0x0FFF	T_ORG	Section 3.3.2 of [RFC8609]
0x1000-0x1FFF	Reserved for experimental use	
0x2000-0xFFFF	Unassigned	

Table 8: FLIC Manifest (Annotation)

6. Security Considerations

6.1. Trust, Security, and Privacy Goals

A FLIC manifest has it’s own security and privacy apart from the underlying application data that the manifest describes. The trust and privacy of the application data is not considered by FLIC, only the trust, security, and privacy of the manifest itself. The trust of a consumer on a manifest is based on the root manifest name, it’s publisher’s key, the signature on the root manifest, and the consumers belief that the given key is trusted to publish the name.

The security and privacy of a FLIC manifest is for the hash pointers and name constructors it contains. FLIC does not provide privacy for: the name of the root manifest or any named ICN packet in the tree, the publisher’s public key ID in any manifest or application data object, the key identifiers used to encrypt a FLIC manifest. The first two items are part of the Content Object and thus outside the FLIC security envelope (though they are authenticated by the root manifest signature). The FLIC SecurityCtx information is outside the encryption envelope, so elements like the symmetric KeyNumber or the RSA-OAEP wrapping KeyId are exposed.

An observer may be able to track, for example, all manifests from a publisher that use the same preshared symmetric key by the pair (publisher public key id, key number) in the Content Object and FLIC security context. With RSA-OAEP encapsulation, the wrapping key id and the wrapped key number are public information and could be used to correlate manifest objects. One could use a unique wrapped key (and thus key number) for each manifest. If using a KDF, the KDFInfo field is public and could be used to correlate manifests if kept constant for a single manifest tree or a sequential counter is used.

6.2. Integrity and Origin Authentication of FLIC Manifests

A FLIC Manifest is used to describe how to form Interests to access CCNx or NDN application data. The Manifest is itself either an individual content object, or a tree of content objects linked together via the corresponding content hashes. The NDN and CCnx protocol architectures directly provide both individual object integrity (using cryptographic strong hashes) and data origin authentication (using signatures). The protocol specifications, [NDN] and CCNx [RFC8609] respectively, provide the protocol machinery and keying to support strong integrity and authentication. Therefore, FLIC utilizes the existing protocol specifications for these functions, rather than providing its own. There are a few subtle differences in the handling of signatures and keys in NDN and CCNx worth recapitulating here:

* NDN in general adds a signature to every individual data packet rather than aggregating signatures via some object-level scheme. When employing FLIC Manifests to multi-packet NDN objects, it is expected that a consumer will use the FLIC Manifest signature to govern the trust of all the application data, rather than the signatures on the application data packets.

* In contrast, CCNx is biased to have primitive objects or pieces thereof be "nameless" in the sense they are identified only by their hashes rather than each having a name directly bound to the content through an individual signature. Therefore, CCNx depends heavily on FLIC (or an alternative method) to provide the name and the signed binding of the name to the content described in the Manifest

A FLIC Manifest therefore gets integrity of its individual pieces through the existing secure hashing procedures of the underlying protocols. Origin authentication of the entire Manifest is achieved through hash chaining and applying a signature *only* to the root Manifest of a manifest tree. It is important to note that the Name of the Manifest, which is what the signature is bound to, need not bear any particular relationship to the names of the application objects pointed to in the Manifest via Name Constructors. This has a number of important benefits described in Section 3.3.

6.3. Confidentiality of Manifest Data

ICN protocol architectures like CCNx and NDN, while providing integrity and origin authentication as described above, leaves confidentiality issues entirely in the domain of the ICN application. Therefore, since FLIC is an application-level construct in both NDN and CCNx, it is incumbent on this specification for FLIC to provide the desired confidentiality properties using encryption. One could leave the specification of Manifest encryption entirely in the hands of the individual application utilizing FLIC, but this would be undesirable for a number of reasons:

* The sensitivity of the information in a Manifest may be different from the sensitivity of the application data it describes. In some cases, it may not be necessary to encrypt manifests, or to encrypt them with a different keying scheme from that used for the application data

* One of the major capabilities enabled by FLIC is to allow repositories or forwarding caches to operate on Manifests (see in particular Section 3.4). In order to allow such intermediaries to interpret manifests without revealing the underlying application data, separate encryption and keying is necessary

* A strong design goal of FLIC is _universality_ such that it can be used transparently by many different ICN applications. This argues that FLIC should have a set of common encryption and keying capabilities that can be delegated to library code and not have to be re-worked by each individual application (see Section 2, Paragraph 15)

Therefore, this specification directly specifies two encryption encapsulations and associated links to key management, as described in Section 3.8. As more experience is gained with various use cases, additional encryption capabilities may be needed and hence we expect the encryption aspects of this specification to evolve over time.

6.4. RSA-OAEP Security

The RSA-OAEP encryption mechanism follows the recommendations of [NIST-800-56Br2], when not used for group keying. FLIC RSA-OAEP does not use integral key confirmation (MacKey), but rather signs the root manifest with the sender's private key. The RSA-OAEP encryption is of both the transported key, K, but also an AES salt, S.

RSA key generation is assumed to comply with [NIST-800-56Br2] Sec 6., but that is outside the control of FLIC.

The belief of the sender that it is using the correct wrapping public key for the receiver, and the sender's belief that the receiver has possession of the private key are functions of the key distribution system beyond the scope of FLIC. One possible mechanism to bring these beliefs closer to FLIC is to require that the manifest requester sign the Interest, such as using the CCNx mechanisms for signing an Interest, and the sender (publisher) check that the signature time is recent. This mechanism requires that the sender already have the public key and trust the owner of the private key to consume the manifest. This mechanism should not be used to distribute the public key.

The RSA-OAEP mechanism uses approved NIST hash functions (SHA256 or SHA512, as per the IANA CCNx hash function registry). It also uses additional information for the OAEP label, as recommended by [NIST-800-56Br2] Sec 9.1.

6.5. Symmetric key lifespan, scope, and use

The lifespan and scope of symmetric keys, as identified by a KeyNum, is different between an AeadCtx and a RsaOaepCtx. In the AeadCtx case, the lifespan and scope is agreed to by the publisher and consumer out of band. In the RsaOaepCtx the lifespan and scope is implicitly the current manifest. We will discuss the RsaOaepCtx case first, as it is the simpler of the two.

For RsaOaepCtx, the first manifest in the tree to use RSA-OAEP must include the wrapped key. The KeyNum is purely local within the manifest. The only requirement is that the KeyNum have a one-to-one association with a specific wrapped key within the manifest. That is, a publisher MUST NOT reuse a KeyNum within a single manifest tree for two different AES key and salt combinations. The key lifespan is strictly the ExpiryTime of the manifest and the scope is that single manifest.

A publisher should use new symmetric keys for each manifest encrypted with RSA-OAEP.

For AeadCtx, a publisher and one or more consumers agree on a shared symmetric key out-of-band and share it via some means. The publisher and consumer must agree on both the lifespan of the key and its scope. The lifespan of such a key should be no shorter than the greatest ExpiryTime of a manifest published with the key. The scope of use should be, at minimum, manifests signed by the publisher's KeyId. The publisher and consumer may also add name prefix limitations to a symmetric key's scope.

Because there is no requirement that an AeadCtx KeyNum be globally unique, one should store them based on their scopes, such as (Publisher KeyId, KeyNum) or (Publisher KeyId, Name Prefix, KeyNum).

If a publisher will use a single AES key across multiple manifests, it MUST use unique KDF derived keys or ensure that it never re-uses the same nonce.

6.6. Hiding Plaintext Length

A publisher MAY include a Pad element in the Node element to hide the true length of a manifest. It is RECOMMENDED that a publisher pad all encrypted Node to a multiple of 128 bytes, or up to the maximum that will fit in the MTU target, to reduce the number of possible manifest sizes. It is important that the sequence of manifest sizes within a manifest tree not have an identifying sequence of sizes. A publisher of an encrypted manifest SHOULD pad all manifests in a tree out to the same largest size.

7. References

7.1. Normative References

[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/info/rfc2119>.

[RFC3310] Niemi, A., Arkko, J., and V. Torvinen, "Hypertext Transfer Protocol (HTTP) Digest Authentication Using Authentication and Key Agreement (AKA)", RFC 3310, DOI 10.17487/RFC3310, September 2002, <https://www.rfc-editor.org/info/rfc3310>.

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

[RFC5288] Salowey, J., Choudhury, A., and D. McGrew, "AES Galois Counter Mode (GCM) Cipher Suites for TLS", RFC 5288, DOI 10.17487/RFC5288, August 2008, <https://www.rfc-editor.org/info/rfc5288>.

[RFC6655] McGrew, D. and D. Bailey, "AES-CCM Cipher Suites for Transport Layer Security (TLS)", RFC 6655, DOI 10.17487/RFC6655, July 2012, <https://www.rfc-editor.org/info/rfc6655>.

[RFC8017] Moriarty, K., Ed., Kaliski, B., Jonsson, J., and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2", RFC 8017, DOI 10.17487/RFC8017, November 2016, <https://www.rfc-editor.org/info/rfc8017>.

[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/info/rfc8174>.

[RFC8569] Mosko, M., Solis, I., and C. Wood, "Content-Centric Networking (CCNx) Semantics", RFC 8569, DOI 10.17487/RFC8569, July 2019, <https://www.rfc-editor.org/info/rfc8569>.

[RFC8609] Mosko, M., Solis, I., and C. Wood, "Content-Centric Networking (CCNx) Messages in TLV Format", RFC 8609, DOI 10.17487/RFC8609, July 2019, <https://www.rfc-editor.org/info/rfc8609>.

[RFC9180] Barnes, R., Bhargavan, K., Lipp, B., and C. Wood, "Hybrid Public Key Encryption", RFC 9180, DOI 10.17487/RFC9180, February 2022, <https://www.rfc-editor.org/info/rfc9180>.

7.2. Informative References

[ccnpy] Mosko, M., "ccnpy: A pure-python CCNx Client", 2024, <https://github.com/mmosko/ccnpy>.

[I-D.wood-icnrg-ccnxkeyexchange] Mosko, M., Uzun, E., and C. A. Wood, "CCNx Key Exchange Protocol Version 1.0", Work in Progress, Internet-Draft, draft-wood-icnrg-ccnxkeyexchange-02, 6 July 2017, <https://datatracker.ietf.org/doc/html/draft-wood-icnrg-ccnxkeyexchange-02>.

[NDN] "Named Data Networking", various, <https://named-data.net/project/execsummary/>.

[NDNTLV] "NDN Packet Format Specification.", 2016, <http://named-data.net/doc/ndn-tlv/>.

[NIST-800-131Ar2] Barker, E. and A. Roginsky, "Transitioning the Use of Cryptographic Algorithms and Key Lengths", National Institute of Standards and Technology SP 800-131A Rev. 2, 2019, <https://doi.org/10.6028/NIST.SP.800-131Ar2>.

[NIST-800-38D] Dworkin, M., "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC", National Institute of Standards and Technology SP 800-38D, 2007, <https://doi.org/10.6028/NIST.SP.800-38D>.

[NIST-800-56Br2] Barker, E., Chen, L., Roginsky, A., Vassilev, A., Davis, R., and S. Simon, "Recommendation for Pair-Wise Key-Establishment Using Integer Factorization Cryptography", National Institute of Standards and Technology SP 800-56B Rev. 2, 2019, <https://doi.org/10.6028/NIST.SP.800-56Br2>.

[ProjectOrigin] Mosko, M., "Peer-to-Peer Sharing with CCNx 1.0", 2014, <https://github.com/PARC/CCNxReports/blob/master/SelectedTopics/p2pshare.pdf>.

[repository] "Repo Protocol Specification", Various, <https://redmine.named-data.net/projects/repo-ng/wiki/Repo_Protocol_Specification>.

[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/info/rfc7927>.

[venti] "Venti: a new approach to archival storage", Bell Labs Document Archive /sts/doc, 2002, <http://doc.cat-v.org/plan_9/4th_edition/papers/venti/>.

Appendix A. Usage Examples

A.1. Locating FLIC leaf and manifest nodes

The names of manifest and data objects are often missing or not unique, unless using specific naming conventions. In this example, we show how using manifest locators is used to generate Interests. Take for example the figure below where the root manifest is named by hash h0. It has nameless children with hashes with hashes h1 ... hN.

```
Objects:
manifest(name=/a/b/c, ptr=h1, ptr=hN)  - has hash h0
nameless(data1)                        - has hash h1
...
nameless(dataN)                        - has hash hN

Query for the manifest:
interest(name=/a/b/c, implicitDigest=h0)
```

Figure 26: Data Organization

After obtaining the manifest, the client fetches the contents. In this first instance, the manifest does not provide any Locators data structure, so the client must continue using the name it used for the manifest.

```
interest(name=/a/b/c, implicitDigest=h1)
...
interest(name=/a/b/c, implicitDigest=hN)
```

Figure 27: Data Interests

Using the locator metadata entry, this behavior can be changed:

```
Objects:
manifest(name=/a/b/c,
hashgroup(loc=/x/y/z, ptr=h1)
hashgroup(ptr=h2)              - has hash h0
nameless(data1)                - has hash h1
nameless(data2)                - has hash h2

Queries:
interest(name=/a/b/c, implicitDigest=h0)
interest(name=/x/y/z, implicitDigest=h1)
interest(name=/a/b/c, implicitDigest=h2)
```

Figure 28: Using Locators

A.2. Seeking

Fast seeking (without having to sequentially fetch all content) works by skipping over entries for which we know their size. The following expression shows how to compute the byte offset of the data pointed at by pointer P_i, call it offset_i. In this formula, let P_i.size represent the Size value of the _i_-th pointer.

offset_i = \sum_{k=1}^{i-1} > P_k.size

With this offset, seeking is done as follows:

```
Input: seek_pos P, a FLIC manifest with a hash group having N entries
Output: pointer index i and byte offset o, or out-of-range error
Algorithm:
offset = 0
for i in 1..N do
   if (P &gt; offset + P_i.size)
      return (i, P - offset)
   offset += P_i.size
return out-of-range
```

Figure 29: Seeking Algorithm

Seeking in a BlockHashGroup is different since offsets can be quickly computed. This is because the size of each pointer P_i except the last is equal to the SizePerPtr value. For a BlockHashGroup with N pointers, OverallByteCount D, and SizePerPointer L, the size of P_N is equal to the following:

D - ((N - 1) * L)

In a BlockHashGroup with k pointers, the size of P_k is equal to:

D - L * (k - 1)

Using these, the seeking algorithm can be thus simplified to the following:

```
Input: seek_pos P, a FLIC manifest with a hash group having
                   OverallByteCount S and SizePerPointer L.
Output: pointer index i and byte offset o, or out-of-range error
Algorithm:
if (P > S)
   return out-of-range
i = floor(P / L)
if (i > N)
   return out-of-range # bad FLIC encoding
o = P mod L
return (i, o)
```

Figure 30: Seeking Algorithm

> | *Note*: In both cases, if the pointer at position i is a
> | manifest pointer, this algorithm has to be called once more,
> | seeking to seek_pos o inside that manifest.

A.3. Block-level de-duplication

Consider a huge file, e.g. an ISO image of a DVD or program in binary be patched. In this case, all existing encoded ICN chunks can remain in the repository while only the chunks for the patch itself is added to a new manifest data structure, as is shown in the diagram below. For example, the venti archival file system of Plan9 [venti] uses this technique.

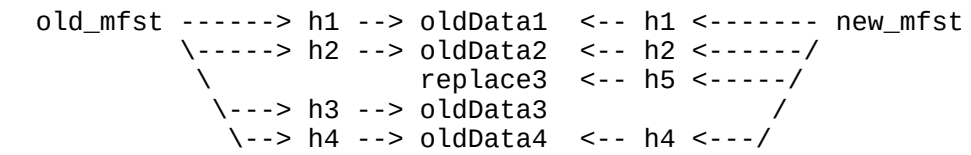


Figure 31: De-duplications

A.4. Growing ICN collections

A log file, for example, grows over time. Instead of having to re-FLIC the grown file it suffices to construct a new manifest with a manifest pointer to the old root manifest plus the sequence of data hash pointers for the new data (or additional sub-manifests if necessary).

In the example shown in Figure 32, there is a first root manifest, Root 1, that has a name /logfile/v1 with an RsaOaepCtx encryption context #1 and a CCNx content object RSA signature #2. In this example, there is only room for 4 pointers per manifest (for the sake of a simple drawing), the manifest Top 1 has two direct pointers and one indirect pointer. Manifest M2 has four direct pointers. Thus, the original log file is the concatendation of D0 ... D5 payloads.

At a later time, the log files is extended with root manifest Root 2. This has its own name, /logfile/v2, with a new encryption content RsaOaepCtx #2. It has its own Content Object signature, Signature #2. Root 2 has one pointer to the previous root manifest, Root 1, and one pointer to the new FLIC tree, Top 2. The previous manifest stays encrypted with the RsaOaepCtx #1 key and the new manifest is encrypted with the new key in RsaOaepCtx #2.

If the same encryption key were used, Root 2 could point to Top 1 instead of Root 1.

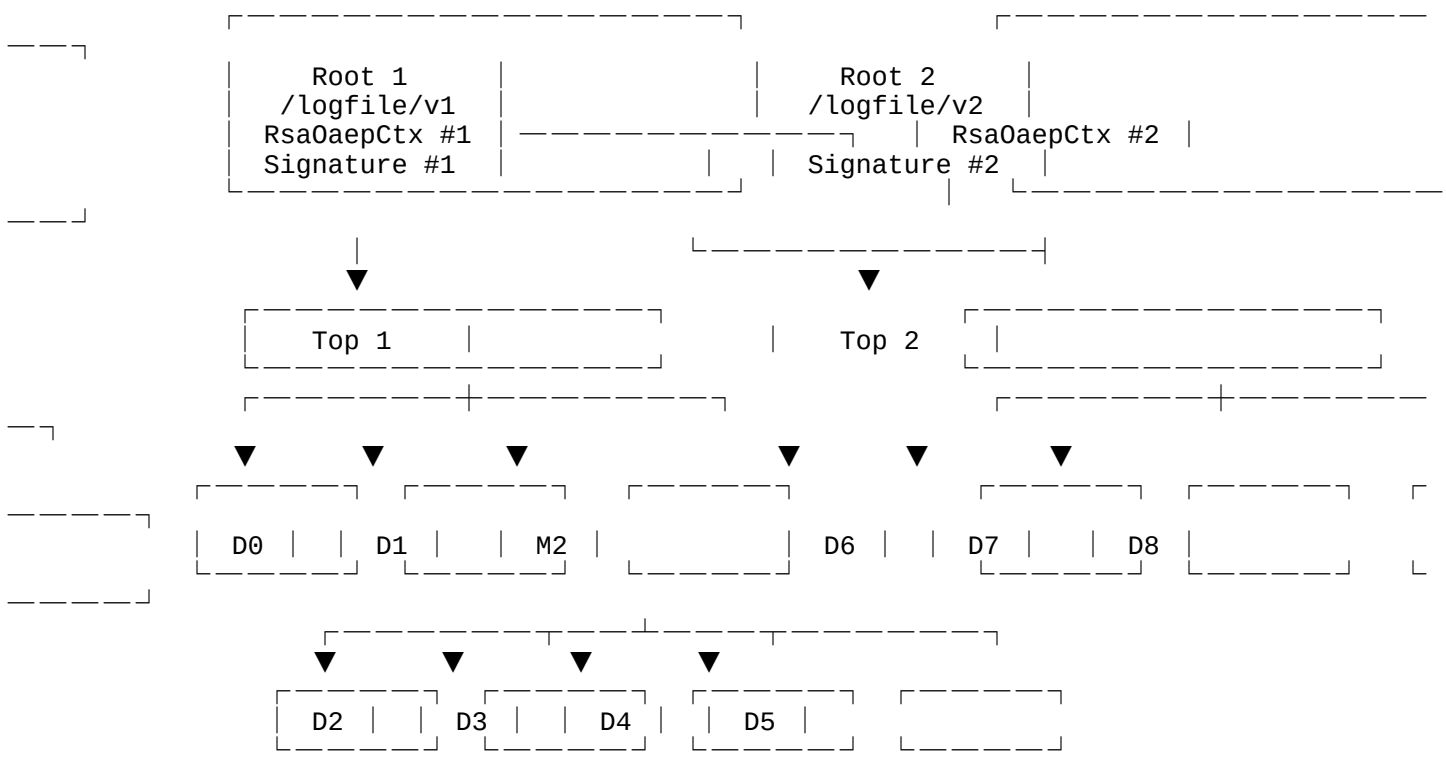


Figure 32: Growing A Collection

Note that the grown manifest does not need to specifically continue the previous naming. The old manifest (Root 1) might use one set of name constructors for the old manifests and data, and the new manifest (Root 2) might use different name constructors. If the manifests used HashedSchema, then naming is not an issue except for the root manfiests.

A.5. Re-publishing a FLIC under a new name

There are several use cases for republishing a collection under a new namespace, or having one collection exist under several namespaces:

* It can happen that a publisher’s namespace is part of a service provider’s prefix. When switching provider, the publisher may want to republish the old data under a new name.

* A publishes wishes to distribute its content to several repositories and would like a result to be delivered from the repository for consumers who have good connectivity to that repository. For example, the publisher /alpha wishes to place content at /beta and /gamma, but routing only to /alpha would not send a request to either /beta or /gamma. The operators of of /beta and /gamma could create a named and signed version of the root manifest with appropriate keys (or delegate that to /alpha) so the results are always delivered by the corresponding repository without having to change the bulk of the manifest tree.

This can easily be achieved with a single nameless root manifest for the large FLIC plus arbitrarily many per-name manifests (which are signed by whomever wants to publish this data):

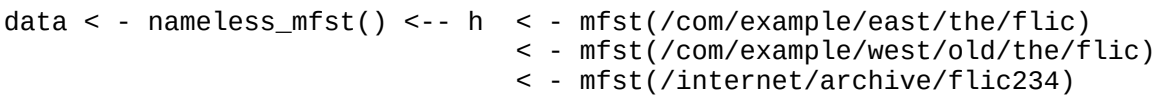


Figure 33: Relocating A Collection

| Note that the hash computation (of h) only requires reading the
| nameless root manifest, not the entire FLIC.

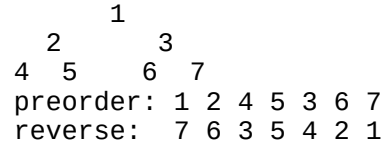
This example points out the problem of HashGroups having their own locator metadata elements: A retriever would be urged to follow these hints which are "hardcoded" deep inside the FLIC but might have become outdated. We therefore recommend to name FLIC manifests only at the highest level (where these names have no locator function). Child nodes in a FLIC manifest should not be named as these names serve no purpose except retrieving a sub-tree’s manifest by name, if would be required.

Appendix B. Building FLIC Graphs

This appendix describes one method to build FLIC graphs. We will use the term "tree" here, as all the construction and traversal mechanisms work as for trees, even if there are duplicate pointers in the graph that cause a node to have multiple parents.

We constructs a pre-order tree in a single pass of the application data, going from the tail to the beginning. This allows us to work up the right side of the tree in a single pass, then work down each left branch until we exhaust the data. Using the reverse-order traversal, we create the right-most-child manifest, then its parent, then the indirect pointers of that parent, then the parent's direct pointers,then the parent of the parent (repeating). This process uses recursion, as it is the clearest way to show the code. A more optimized approach could do it in a true single pass.

Because we build from the bottom up, we use the term 'level' to be the distance from the right-most child up. Level 0 is the bottom-most level of the tree, such as where node 7 is:



The [ccnpy] Python code has more complex algorithms to build a FLIC graph is one pass while optimizing the tree structure. Its default is to minimize the tree height while fitting within a given packet MTU. It will use m-indirect and k-direct pointers for interior manifest nodes and n-direct pointers (n=m+k) for leaf manifest noides.

The Python-like pseudocode build_tree(data, n, k, m) algorithm creates a tree of n data objects. The data[] array is an array of Content Objects that hold application payload; the application data has already been packetized into n Content Object packets.An interior manifest node has k direct pointers and m indirect pointers.

```
build_tree(data[0..n-1], n, k, m):
  # data is an array of Content Objects (Data in NDN) with app data.
  # n is the number of data items
  # k is the number of direct pointers per internal node
  # m is the number of indirect pointers per internal node

  segment = namedtuple('Segment', 'head tail')(0, n)
  level = 0

  # This bootstraps the process by creating the right most child
  # manifest. A leaf manifest has no indirect pointers, so k+m
  # are direct pointers
  root = leaf_manifest(data, segment, k + m)

  # Keep building subtrees until we're out of direct pointers
  while not segment.empty():
    level += 1
    root = bottom_up_preorder(data, segment, level, k, m, root)

  return root

bottom_up_preorder(data, segment, level, k, m, right_most_child=None):
  manifest = None
  if level == 0:
    assert right_most_child is None
    # build a leaf manifest with only direct pointers
    manifest = leaf_manifest(data, segment, k + m)
  else:
    # If the number of remaining direct pointers will fit
    # in a leaf node, make one of those. Otherwise, we need to be
    # an interior node
  if right_most_child is None and segment.length() &lt;= k + m:
    manifest = leaf_manifest(data, segment, k+m)
  else:
    manifest = interior_manifest(data, segment, level,
                                 k, m, right_most_child)
  return manifest

leaf_manifest(data, segment, count):
  # At most count items, but never go before the head
  start = max(segment.head(), segment.tail() - count)
  manifest = Manifest(data[start:segment.tail])
  segment.tail -= segment.tail() - start
  return manifest
```

```
   interior_manifest(data, segment, level, k, m, right_most_child)
     children = []
     if right_most_child is not None:
       children.append(right_most_child)

     interior_indirect(data, segment, level, k, m, children)
     interior_direct(data, segment, level, k, m, children)

     manifest = Manifest(children)
     return manifest, tail

   interior_indirect(data, segment, level, k, m, children):
     # Reserve space at the head of the segment for this node's
     # direct pointers before descending to children.  We want
     # the top of the tree packed.
     reserve_count = min(k, segment.tail - segment.head)
     segment.head += reserve_count

     while len(children) &lt; m and not segment.head == segment.tail:
       child = bottom_up_preorder(data, segment, level - 1, k, m)
       # prepend
       children.insert(0, child)

       # Pull back our reservation and put those pointers in
       # our direct children
       segment.head -= reserve_count

   interior_direct(data, segment, level, k, m, children):
     while len(children) &lt; k+m and not segment.head == segment.tail:
       pointer = data[segment.tail() - 1]
       children.insert(0, pointer)
       segment.tail -= 1
```

Authors' Addresses

Christian Tschudin
University of Basel
Email: christian.tschudin@unibas.ch

Christopher A. Wood
Cloudflare
Email: caw@heapingbits.net

Marc Mosko
Email: marc@mosko.org

David Oran (editor)
Network Systems Research & Design
Email: daveoran@orandom.net