CCNx (Pre 1.0) Access Control Specifications

Diana Smetters[1], Philippe Golle[1]

Abstract
This document describes the support for encryption and access control in CCNx 0.7x. It begins, for reference and convenience, with the documentation of signing and signature formats used by CCNx. It then presents a description of the content encryption and access control mechanisms provided by the CCNx Java library.

Keywords
Content Centric Networks, Security, Access Control

[1] *Palo Alto Research Center*
Point of Contact: Ersin Uzun <ersin.uzun@parc.com>

Contents

Introduction

This document describes the current support for encryption and access control in CCNx. It begins, for reference and convenience, with the documentation of signing and signature formats used by CCNx. It then presents a description of the content encryption and access control mechanisms provided by the CCNx Java library.

Content encryption can be performed using symmetric keys managed by an application in any way it chooses, or can be used together with a key distribution scheme to automatically manage encryption keys for content. The most general notion of a key distribution scheme for content protection is an encryption-based access control scheme; CCNx's general architecture for

defining and managing encryption-based access control schemes is described in section 3. Implementing any key distribution scheme requires the ability to wrap keys – encrypt keys in other keys. The default key wrapping mechanisms present in CCNx are described in section 4. We can also use key derivation to turn one key into many different keys, used to protect multiple pieces of content. CCNx's example access control scheme uses a generic key derivation algorithm, which is also available for more general use. This algorithm is defined in section 5. Finally, we can gather up these individual components – encryption, key wrapping, key derivation, and put them together to define new access control or key distribution schemes that are useful for various application scenarios. CCNx contains an example of one such scheme; a scheme for encryption-based access control providing (eventually) many of the features of standard file system access control, and implemented using the CCNx pluggable access control framework. This scheme is described in section 6.

1. Signing

A ContentObject in CCNx consists of:

```
ContentObject ::= Signature
                  Name
                  SignedInfo
                  Content
```

The `Signature`, described below, is computed over the concatenated ccnb binary encoding of the `Name`, `SignedInfo` and `Content` components of the `ContentObject`, with all of their start and end tags, but without the start or end tag of the `ContentObject` itself, or any component of the `Signature`. This makes possible a signing implementation that takes a packet in over the wire, selects the digest algorithm to use to verify the signature based on information in the `Signature` component, and then digests the bulk of the packet, exactly as it arrived on the wire, in order to verify its signature.

1.1 Signature

The `Signature` component of the `ContentObject` consists of:

```
Signature ::= DigestAlgorithm
              Witness
              SignatureBits
```

1.1.1 Digest Algorithm

The digest algorithm specifies the cryptographic digest algorithm used in signature generation. We need to specify either the digest algorithm used to generate the signature, or a combined signature algorithm which includes both the digest algorithm and public key algorithm used (for example, "SHA1withRSA"), so that verifiers know what digest to use to verify the signature. The X.509 digital signature standard uses a signature algorithm specified at the start of the certificate, as well as in the signature itself. The PKCS#7 standard for signed data, and the standard for XML signatures specify only a digest algorithm up front. Choosing to specify a digest algorithm, rather than a signature algorithm, at the start (and to only specify the digest algorithm, letting the signature algorithm be determined by the key rather than separately specified in the signature) forces us to assume that a given key can only be used for one algorithm type. However, assuming that a smaller number of digest algorithms are used than public key types, including only the digest algorithm in the specifier saves us the bytes for a separate specification of signature algorithm in the signature, and increases the chance that we will also be able to elide the digest algorithm identifier itself because the digest algorithm chosen will be one selected as the most common default (for now, SHA-256).

We place the digest algorithm identifier, along with the content of the signature itself, at the start of the `ContentObject`, so that devices that need to perform signature verification on the incoming data stream as it arrives may do so. (Though they will need to store the `Signature` itself for verification until both the data is processed and the public key needed has been retrieved.) For the moment, the digest algorithm is specified as a UTF-8 encoding of an Object Identifier, or OID. If it matches the default value (the OID for SHA-256, or 2.16.840.1.101.3.4.2.1) it is elided.

1.1.2 Witness

A `Witness` is additional information necessary to verify the signature, particularly in the case where signature generation is aggregated and performed over multiple `ContentObjects` at once. In such a case, the `Witness` allows an individual `ContentObject` to be verified as being part of that set. For example, for elements authenticated using a Merkle Hash Tree, the witness information would be the elements of the hash path through the tree.

The `Witness` is represented as a DER-encoded

PKCS#1 `DigestInfo`, which contains an `AlgorithmIdentifier` (an OID, together with any necessary parameters) and a byte array (OCTET STRING) containing the digest information to be interpreted according to that OID.

1.1.3 Signature Bits

The contents of the digital signature itself, computed as appropriate for the algorithm used (see section 1.2.1). For now, this is the bits of the signature itself, encoded as appropriate for the particular cryptographic algorithm used (in other words, no encapsulating specification of signature algorithm).

1.2 Signature Generation

Signature generation in CCNx takes one of two forms: either individual blocks are individually digitally signed with a standard public key signature algorithm, or multiple blocks are signed at once using an aggregated signature.

1.2.1 Algorithm Choice

The choice of signature algorithm and signature granularity (whether, or how much, to aggregate signing) is done by the publisher using a number of considerations: most importantly ease of implementation, computational constraints on the publisher and verifier, and bandwidth constraints. Signing each block individually is more computationally costly for the publisher than aggregating signature generation, but offers lower latency and requires less space for the signature. (Aggregated signatures require the use of per-block witness information to allow each block to be individually verified.) Verification cost is lower for aggregated signatures, as the consumer can cache and reuse the result of verifying the signature itself, and even parts of the witness, across multiple blocks.

The choice of the public key algorithm used to generate either the individual block or aggregated signature is determined by what keys the publisher has available, what algorithms they expect their consumers to support, and the relative cost of signature generation and verification for individual algorithms, as well as, of course, their security requirements. For example, the RSA algorithm offers a significant asymmetry in signing and verification times: signature generation is an order of magnitude (or more) slower than verification. It is therefore a good choice if signatures will be verified many more times than they will be generated (but it comes at the cost of relatively long signatures). Elliptic curve cryptography can be used to generate short signatures with high security, but verification is as computationally expensive, or more expensive, than signature generation.

1.2.2 Individual Block Signing

To sign an individual `ContentObject`, we generate a standard digital signature using PKCS#1 padding over the `Name`, `SignedInfo`, and `Content` portions of the encoded `ContentObject` described above with the specified digest algorithm and a signature algorithm determined by that and the key. We place the resulting signature in the `SignatureBits` portion of the `Signature`, omitting the `Witness`. Such a signature can be generated and verified using any number of standard cryptographic libraries.

1.2.3 Aggregated Signing

An aggregated signature takes a set of 2 or more `ContentObjects`, and generates a signature over their combination, together with a set of per-object witness data such that it is possible to verify for each `ContentObject` in the set that it is indeed a member of the set, that it was signed as part of the set by the designated public key, and that it has not been altered since (up to the security of the cryptographic algorithms used).

The CCNx standard library currently implements a single aggregated signature algorithm, using Merkle hash trees. However, the intent is to make the aggregated, or bulk signature implementation sufficiently parameterizable that other algorithms (for experimental or production use) can be included. At the same time, the expectation that all nodes in the network can verify any signature they choose suggests that the number of production algorithms eventually supported will be relatively limited. As of the initial release, aggregated signature generation is set up to allow for new implementations to be added, but verification is not.

While the typical use for aggregated signatures is to sign a set of related content objects (for example a set of segments from a single stream), there is no requirement that the objects aggregated be related at all. (However, when they are, it maximizes the likelihood that a verifier will be verifying them all together and so will be able to reuse cached verification data.)

1.2.4 Merkle Hash Tree Aggregated Signatures

We describe here the aggregated signature algorithm implemented in the CCNx library. Some of the design elements used in this algorithm were selected to maxi-

mize overlap with the standard signing implementation, and would be good common elements to use for any aggregated signature implementation. Additional details of the implementation can be seen in the Java library source code.

A Merkle hash tree is constructed most simply by taking a set of data elements, and arranging them as leaf nodes in a n-ary tree. Each leaf node is represented by its cryptographic digest, or hash. The parent of a set of n leaves is calculated by concatenating the digests of those n leaves and then computing a cryptographic digest, or hash, over the result. This process is iterated up the tree, until a single root digest is calculated at the top. That root digest is then digitally signed. To verify a single leaf, one needs the leaf itself, as well as a Merkle Path through the tree, i.e. the values (digests) in the tree of that leaf's $n-1$ siblings, and its parents' siblings, and so on up the tree, so as to be able to take the leaf and the path values, and recompute the root. A consumer verifying the leaf uses the leaf and path data to compute a root value, and then given a digital signature on the actual root, determines whether or not the computed root value matches the value that was originally computed and signed. Assuming the security of the cryptographic digest algorithm used to compute the tree, this verifies the content and position of the leaf in the tree.

CCNx uses binary Merkle hash trees, with a parameterizable digest algorithm used to compute the leaf and interior (node) digests. Given a set of 2 or more `ContentObjects`, the leaf digest of each of those `ContentObjects` is computed using the same method used to compute an individual signature over a single `ContentObject`. In other words, each leaf is represented by the cryptographic digest of the concatenated ccnb binary encodings of its contained `Name`, `SignedInfo` and `Content` fields. The node (interior) digests of the tree are computed as described above: as the digest of the concatenation of the two children of the node to be computed. If that node has only a left child (the tree formulation used ensures that no node will have only a right child), the digest of that node is computed as the digest of its left child alone (this simplifies implementation over skipping the digest computation).

To generate the signature on a Merkle hash tree (MHT), we sign the root node as follows. The root is already a digest, so in theory we could just wrap it up in some PKCS#1 padding and sign it with our private key. But there are basically no crypto software packages that provide signature primitives that take already-digested data and just do the padding and encryption, and so we would be asking anyone attempting to implement CCNx MHT signing (including ourselves) to re-implement a number of signature algorithms. We might also want to sign with a key that does not support the digest algorithm we used to compute the root (for example, DSA). So we take the slightly more computationally expensive, but vastly simpler (implementation-wise) approach of taking our digest and signing it with a standard signing API, which means digesting it one more time for the signature. So we sign (digest + encrypt) the root digest with a standard off-the-shelf signature algorithm. It is this signature that we place in the `SignatureBits` of the `Signature` of each `ContentObject` in the aggregated set, and the digest algorithm used for this signature that we place in the `DigestAlgorithm` field.

To represent the witness, or Merkle Path, for each `ContentObject` in the aggregate, we list the leaf or node digests for the sibling of this leaf, and the sibling of its parent, and on up the tree, in that order. We do not include the digest of the leaf itself (that can be calculated from the content) or the root digest (which can be calculated from the calculated leaf digest and the path) in the path data (witness). In order to be able to verify the content with respect to the given path, the verifier needs to be able to determine whether this leaf represents the left or right leaf in a terminal pair, and which position (left or right child) each of the digests on the witness path takes (as the computation of the parent digest is order dependent). Because of the representation of trees used, the index of the leaf whose path this is determines the position of the remainder of the nodes on the path if they are presented in order (from top to bottom). We therefore represent our Merkle Paths as follows (noted in ASN.1):

```
MerklePath ::= SEQUENCE {
           nodeIndex INTEGER,
           nodes NodeList
}

NodeList ::= SEQUENCE OF OCTET STRING
```

We could probably save a few bytes by not encoding this as DER, and simply packing in the bytes to represent this data. But this encoding offers a fair amount of ease of parsing and clarity, at the cost of some $5+2*$ pathLength bytes of overhead, or 20 bytes in typical paths. At some point this may seem too much, and we will move to a more compact encoding.

The `Witness` for a Merkle hash tree-signed `ContentObject` contains, as noted above, a DER-encoded PKCS#1 `DigestInfo`. The `AlgorithmIdentifier` of that `DigestInfo` contains an OID that specifies both that this is a Merkle path, and the component digest algorithm used to compute the leaf and interior node digests. The OCTET STRING of that `DigestInfo` contains the DER-encoded `MerklePath` for this leaf. OIDs for initial MHT algorithms are below. The default for the CCNx library is SHA256MHT.

```
SHA-1-Merkle-Hash-Tree ::=
        1.2.840.113550.11.1.2.1
SHA-256-Merkle-Hash-Tree ::=
        1.2.840.113550.11.1.2.2
```

2. Encryption

This section describes the basic mechanics of content encryption for standard, segmented content, under a symmetric key. It relies on the `SegmentationProfile` for naming conventions. Content encryption can be performed using keys managed by an application in any way it chooses, or can be used together with a key distribution scheme to automatically manage encryption keys for content. Note that the current CCNx release supports only counter (CTR) mode; support for CBC mode is in progress.

2.1 Goals

The goals of this encryption specification are, in order of importance:

- simplicity: a programmer should need to specify as little as possible in order to successfully and securely encrypt content.
- speed: the encryption specification should be designed to allow as efficient operation as possible; in particular, it should enable (though not require) parallel operation whenever possible, and minimize the number of public key operations necessary in order to read content, modulo the ability to effectively manage access.
- data use flexibility: the encryption specification should provide for random access to the data, and allow reads to start from any data segment. This implies that the encryption specification should be segmentation-aware, i.e. responsive to the block boundaries of the content. Otherwise, encryption blocks may break across content segments, and require more complicated higher-level structures to manage decryption of that data. This does not preclude segmentation-opaque approaches to encryption, such as applications that encrypt their data completely at the application level and then add it to CCNx. Such approaches will just require more sophisticated application-level machinery to allow completely random access to the data.
- space efficiency: we use algorithms without length expansion whenever possible. (Though given the amount of space used for authentication data, this may seem a false economy; particularly when all we are saving is an explicit initialization vector or a block's worth of padding.)
- algorithm agility and future-proofing: we recognize that cryptographic algorithms come and go, and therefore it is necessary to allow algorithms to change, or different applications to plug in their own preferred algorithms when necessary. Ideally we want to enable this without requiring those applications to lose the benefits of the tight integration of encryption with efficient low-level networking code. The advantage of CCNx is that encryption algorithms must only be agreed on between content producers and consumers, removing the requirement that all potential communication partners agree on the universe of available algorithms.

2.2 Basic Approach

A content object $\mathbf{O}$ (possibly segmented according to the SegmentationProfile) is encrypted under a single key $\mathbf{K}_O$ (the derivation of $\mathbf{K}_O$ for a given object $\mathbf{O}$ is described in section 5). The basic encryption profile takes $\mathbf{K}_O$ and uses it to encrypt the $n = \text{length}(\mathbf{O})/\text{fragmentSize}$ segments of $\mathbf{O}$. If we use counter mode to do this, we need to ensure that both the publisher and the desired recipients can determine both the key and the initial counter value for each block, and that counter values are never reused for the same key.

Encryption is performed prior to packet authentication (signing). Callers can specify the desired final packet length, including headers, authentication information and encrypted data. The segmenter uses information about any message expansion performed by the cipher being used to determine the segment length to use to

achieve the desired output length, within the limits of same-length segments.

2.3 Supported Algorithms: AES-CTR and AES-CBC

The default algorithm for encryption of CCNx content is AES in counter mode (AES-CTR). Counter mode is in effect a stream cipher: it generates a stream of encrypted bytes (a key stream), which is then XORed with the data to produce the ciphertext. Its advantage is that it is length-preserving. The length of the ciphertext generated need not be a multiple of the block length of the underlying block cipher. Instead, the output ciphertext is only as long as the input plaintext (note that this leaks the length of the plaintext). AES-CTR is highly parallelizable. The keystream can be generated in parallel, potentially before the plaintext or ciphertext to be encrypted or decrypted has been received. Encryption and decryption use exactly the same operations. AES-CTR is a malleable mode of encryption (if you know the plaintext, you can easily alter unauthenticated ciphertext to contain any plaintext you want) and so it must be used with authentication. However, this is not a problem as all CCNx data is authenticated. The key stream is generated by encrypting sequential values of an incrementing counter using the underlying cipher. As this is a stream cipher, it is imperative that no key stream segment (i.e. no counter value) is ever used twice for a given key, otherwise all security is lost. To achieve full security, however, the counter value must be sufficiently unpredictable; e.g. must contain 64 bits of effectively random or at least rare data (e.g. a sender identifier). To achieve space compression, the initial counter value must be computable by the sender and receiver, rather than requiring that it be sent explicitly.

A secondary encryption mode uses AES in cipher-block-chaining mode (AES-CBC), generating the initialization vector using the algorithm described in section 5 to generate the random portion of the counter. AES-CBC is more complex, as the padding required may expand the size of the written data.

The key length and encryption algorithm (e.g. AES-CTR, AES-CBC) are specified inside the master key block (see section 5). This was determined to be more efficient and likely sufficiently general than marking each packet with the specific algorithm used. We begin with a simplifying assumption that all data keys derived from a single master key will use the same encryption mode, and specify the desired mode in the master key block. That allows the key derivation operation (see section 5) to derive not only the data key to be used, but any auxiliary data needed (counter for CTR mode, initialization vector for CBC mode). The low-level segmentation interface takes a Cipher parameter as well as a mechanism to construct a per-segment counter or initialization vector. This enables higher-level interfaces to begin to add their own choices for Cipher types.

2.4 Formal Specification

The segmenter is called with parameters identifying:

- the encryption algorithm and mode to use, if any
- the encryption key $\mathbf{K}_O$ to use for the data object $\mathbf{O}$ to be segmented
- an 8-byte value $\mathbf{IV}_{\text{seed}}$ used as an initialization vector seed for this item (in CBC mode) or a random counter component (in CTR mode). This derivation is described in section 5.
- the desired full segment (packet) length, including supporting data

2.4.1 Counter mode

In CTR mode, the counter for a given block B in segment S is constructed as follows:

$$\text{CTR} = \mathbf{IV}_{\text{seed}} || S_{\text{num}} || B_{\text{num}}$$

where the segment and block numbers are represented in unsigned, 1-based big endian format. The total width of the counter value is 16 bytes, where the first 8 bytes are the $\mathbf{IV}_{\text{seed}}$ value, the next 6 bytes are the segment number S_{num}, and the last 2 bytes are the block number B_{num}. Note that according to the SegmentationProfile, a single-segment object has a segment number component in its name, and follows the specification above for managing its encryption keys.

2.4.2 Cipher-block chaining mode

In CBC mode, the value $\mathbf{IV}_{\text{seed}}$ is used as a seed to generate an initialization vector $\mathbf{IV}_S$ for each segment S as follows:

$$\mathbf{IV}_S = E_{\mathbf{K}_O}(\mathbf{IV}_{\text{seed}} || S_{\text{num}} || B_0)$$

where the segment number is encoded in 1-based, unsigned, big-endian form, and represented in the $b - \ell$ rightmost bytes of the plaintext above, where b is the width of the block cipher in use, and ℓ is the length of the

numeric representation of the segment number. We let $B_0 = 1$ to maintain consistency with standard CTR mode use. The same initialization vector expansion function is used regardless of mode for simplicity.

If concerns were raised that key derivation (see section 5) might result in key collisions (e.g. if the wrong level of specificity was applied to key derivation, such as applying key derivation to a file instead of a version name), and if we were willing to require a segment to be retrieved prior to generation of its decryption keystream, than we could take the actual timestamp of a given segment, represented in its `SignedInfo` and combine it with the provided IV_{seed}.

2.5 Parameters

Counter mode can generate 2^{64} key stream blocks before an adversary gains any advantage in his attempt to distinguish the key stream from random, assuming that the counter contains sufficient entropy to thwart offline precomputation attacks (see [3]). For smallest AES key length (16 bytes), this will encrypt 2^{68} bytes, or 256 exabytes of raw data securely, if one can create 2^{64} different counter values.

AES in counter mode is typically used with 128 bit counters. As noted above, 64 bits of that counter must be relatively unpredictable, effectively derived by sender and receiver as part of the key. This leaves 64 bits (8 bytes) to be used for the counter itself.

As we are encrypting segmented data, a portion of each counter needs to be used to identify the current segment, and the remainder to identify the cipher block within the segment. We assume likely segment sizes will range from a minimum of 1 KB (2^{10} bytes), to a maximum of 64 KB (2^{16} bytes). Segments typically represent the amount of data that can fit into a single packet (along with name and authentication information). The 1 KB size represents a packet meant to fit into a 1500 byte Ethernet frame with heavy overhead. At a minimum block size of 16 bytes (an AES key length of 128 bits), this corresponds to 2^6 blocks for the small block size, and 2^{10} blocks for the large block size. Allowing 2 bytes for a block counter would allow up to 2^{16} blocks in a single segment, equivalent to segments containing a maximum of 2^{20} bytes (1 MB) using 128 bit AES keys, or 2^{21} bytes (2 MB) using 256 bit AES keys.

Assuming 6 bytes remaining to represent the segment number, this scheme can represent up to 2^{48} segments. Using 1 KB segments, this scheme could index 2^{58} bytes, or 256 petabytes, in a single data stream. Using 64 KB segments, this scheme can index 2^{64}, or 16 exabytes. Using full-length 1 MB segments, we can achieve stream capacity of 256 exabytes.

This parametrization achieves a good balance between maximizing encryption capacity while supporting expected packet size ranges, using a fixed field width for segment and block numbers (which simplifies implementation).

2.6 Data Handling

Encrypted data objects are of type ENCR (typically, but not always, encapsulating type DATA; if the original type is not DATA, the consumer must be able to figure out how to handle the content post-decryption).

We would like to minimize the number of additional copies made of data during the process of outputting it to the network, whether it is encrypted or not. We can think of data as coming in, together with information about what name it should have and who should sign it, at the top of a stack, and signed `ContentObjects` exiting at the bottom of the stack. The `ContentObjects` may exit one by one, each individually signed, or they may be created as a group, where the signature process is amortized across the set.

Creating a `ContentObject` requires making a copy of the content that will go in that segment, because the content must be immutable from that point on. (For storage, the content can actually be stored separately from the `ContentObject`; for now we store a copy of the content inside the `ContentObject` object itself.) So that is the one copy we can't avoid.

The CCNSegmenter takes content stored in buffers specified by its caller (in general a stream with internal buffers, or a CCNWriter which takes a buffer from a higher-level user directly; network objects use streams for their output so reduce to a pass-through version of that case where there is no standing buffer, save perhaps one in the intervening stream). It does all its computation on those buffers, and then copies them segment-wise to `ContentObjects` at the end. There are two problems with this: first, it requires the caller of a segmenter to have buffers someplace, rather than just streaming data through. Second, if we want to add in encryption, e.g. by putting inputs to the segmenter into a CipherOutputStream, we then have to get it out to the `ContentObjects`, which currently expect to operate on buffers – hence requiring extra buffers.

A potential 2-stage plan would operate as follows. We keep callers dealing in terms of buffers for now.

We deal with encryption via CipherOutputStreams, and have those bottom out in a stream that writes directly to `ContentObjects`, and has the bulk signers work directly on those content objects, i.e. have the segmenter build content objects more incrementally. In stage 2, we allow callers to interface their output streams directly to the streams used by the segmenter, with no buffers required. Copies still happen, both at the bottom into the `ContentObject`, and as they pass through the cipher stream, but potentially that would be it. Once we move to stage 2, we can add support for AES-CBC and other block modes of operation that might incur length expansion.

3. Access Control in CCNx

CCNx is beginning to provide a general framework for "plugging" in access control schemes – schemes for automatically distributing encryption keys over content. In this section we describe the high-level functionality available to support such schemes, and in section 6 below we describe one such scheme currently supported by CCNx.

3.1 Access Control Model

Our initial approach to making it easy to implement new access control and/ or key distribution models in CCNx is to organize access control around the concepts of "namespaces" – subtrees of the content tree rooted at some particular node *n*, where the entire subtree's content is controlled according to some particular access control scheme.[1]

We provide some generic mechanisms to make it easy to associate an access control scheme with a namespace, and to use that "scheme" to provide keys to encrypt and decrypt content automatically as that content is read. This automatic key creation and retrieval functionality is currently supported by the CCNx Java library, for reading and writing content with the io streams and their subclasses, and the network objects. (Encryption/decryption keys can also be explicitly provided and this automated process avoided, when desired.)

[1] Note that by "access control" here we mean access control by encryption, and that when we say a subtree's content is controlled according to a particualar scheme, we mean to include the notion that some or all of that content may be deliberately left unencrypted. For example, the "default" access control scheme could be considered the null scheme, where no content is encrypted and no specific validation rules (which might be required, say, to implement write control) are required.

This framework of generic mechanisms is implemented by the abstract class `AccessControlManager`. Subclasses of `AccessControlManager` are used to implement particular access control schemes (e.g. `GroupBasedAccessControlManager`, described in section 6). But the `AccessControlManager` class (ACM, for short) itself provides the machinery for loading and using individual ACM subclass instances. We associate one such ACM subclass instance with each namespace *N*, and use that ACM instance to manage content in that namespace (subtree). We refer to this ability to aggregate access control policies across multiple, independently-managed subtrees as *Multi-Location Access Control*. These multiple subtrees can be managed using the same access control scheme, and have ACMs which are (parameterized) instances of the same ACM subclass, or they can be managed using a heterogeneous collection of ACM subclasses.

The core features of the ACM class used to implement this framework are as follows:

- `loadAccessControlManagerForNamespace`: a method which determines whether a given namespace is subject to an access control policy, and if it is, detects the policy in force and instantiates the appropriate ACM subclass to manage access control in that namespace for this user. These ACM instances are cached per-user (per-`KeyManager` instance), and so once one is loaded for a given namespace, it will be reused. See section 3.2.

- `keysForOutput`: a method which is called by the output streams (and via them, by the write methods of the network objects) to determine whether or not the content to be written is subject to access control, and if it is, to obtain from the ACM controlling that namespace, a content key to use to encrypt that content, created or obtained according to the access control scheme implemented by that ACM. (If a key is passed in to the constructor of the stream to use to encrypt the content, that key is used and the access control lookup is bypassed.) The default implementation of `keysForOutput` encrypts content under a randomly-generated nonce

key, and stores that in a place in the namespace defined in the `AccessControlProfile` class. That nonce key is then encrypted under a key obtained from the controlling ACM using the `getDataKeyWrappingKey` method, which needs to be implemented by ACM subclasses.

- `keysForInput`: when encrypted content is retrieved (content of type `ENCR`), and no key is provided directly to decrypt it, this method is called by the streams (and hence the network objects) in order to find out what ACM is managing the namespace from where the content is retrieved, and to request the decryption key from that ACM if the reader has rights to read the content. Eventually hooks will be put in place to allow readers to also potentially validate things like compliance with write access restrictions. The default implementation of `keysForInput` basically reverses the default implementation of `keysForOutput` – retrieving the nonce data key the content is encrypted under, and using the method `getDataKeyWrappingKey` to retrieve from the controlling ACM the wrapping key necessary to unwrap it. This means that a "new" ACM type that wishes to only implement a key distribution scheme need only overridde `getDataKeyWrappingKey`).

- `inProtectedNamespace`: a method implemented by ACM and potentially overridden by its subclasses, that is called by the ACM lookup machinery to determine whether this ACM is the one "controlling" a given piece of content.

- `isProtectedContent`: a method implemented by each ACM subclass to exempt particular types of content (e.g. content of type `KEY` or `LINK`) from encryption; it is called by keysForOutput to determine whether to encrypt this content or not. Subclass implementations should potentially call `super.isProtectedContent` and not ask to encrypt content that the superclass would not; otherwise some machinery may fail (e.g. the default `isProtectedContent` exempts public keys (type `KEY`), links (type `LINK`, though an encrypted link type may be needed in the future), and the metadata controlling access control itself from encryption).

This describes how an ACM instance, once retrieved, determines whether and how to encrypt or decrypt content. The last remaining component of our *Multi-Location Access Control* system is the mechanism for retrieving the ACM, if any, controlling a particular namespace.

3.2 Namespaces and Access Control Managers

There are two choices for how one could establish that access control (of a particular type) should be used for a particular namespace. One option would require application writers to always "know" what type of access control to use and set that up for any namespace they deal with. However, applications are usually designed to operate on content in many different namespaces, with many different policies, and the access control policy applied to a given namespace is usually determined by local IT policy – not something that a generic application should "know" about.

The second, more appealing, option, marks, for a given name tree, that we are using a particular access control profile to control access to content in that subtree. We don't want users to be able to decide to simply not encrypt things if they don't like the policy associated with that tree; similarly we don't want to offer users and developers too many opportunities for errors in the encryption of content. So we support the notion of marking a namespace with the profiles it is supposed to require, and having the library enforce those automatically throughout the content tree.

At the top of an access controlled namespace, we store a namespace marker in the metadata, indicating that access control is in force for this namespace, and what profile is in use (which is used, together with a registration/lookup mechanism in the `AccessControlManager` class, to determine what subclass of ACM to instantiate to manage this subtree). It can also list parameters necessary for the given ACM subclass to be instantiated, such as default prefixes for group and user information. This "policy marker" determines the root of the access control hierarchy, discussed below.

The `NamespaceManager` class is designed to store and retrieve policy markers demarcating functional namespaces in the overall name tree. The example currently implemented is this one – policy

markers for acces control – but the mechanism is intended to be general (though it is still very preliminary). The `NamespaceManager` class provides an efficient paralell search mechanism and negative caching feature to make it fast to lookup whether a policy of a given type exists at a particular point or along a particular path in the name hierarchy – this is used by `loadAccessControlManagerForNamespace`, discussed above, to find a policy marker if one is present, and instantiate the corresponding access control manager. To speed this process negative caching in the `NamespaceManager` is used to avoid re-searching previously checked locations in the namespace.[2]

There are two choices for how this automated ACM lookup functionality can be integrated with content writes. (Content reads handle encryption more simply – if content is encrypted but no keys are provided, a strong assumption can be made that an ACM should be searched for. Content writes are more ambiguous.) On every write, one could look for a controlling ACM – first in the cached set of ACMs then if none is found there, by searching for policy markers – to find any access control policy in force and obtain encryption keys prior to write. This has the advantage that it is completely transparent for developers. They don't have to do anything about access control – it just "happens" automatically. And if the "common" case is that access control is in use, the controlling ACM is searched for once and then usualy retrieved (rapidly) from cache, causing negligable impact on performance. However, if the "common" case is that access control is not used, every write incurs the penalty of searching the unsearched parts of the name path being written on the off chance that an access control policy might be found (previously searched components will be avoided by negative caching). This puts steep minimum performance bound on writes in the absence of access control, as they must time out looking for policy markers.

As the access control support in CCN is still relatively early in its development, and the predominant use of CCN is for pilot applications without encryption-based access control, we have currently disabled the automatic search functionality. Applications wishing to use access control must arrange to explicitly load access control managers for the controlled namespaces they wish to work with. These ACMs must be loaded *per user* – ACM instances, and their caching mechanisms, are currently tied to a a given `KeyManager` holding the identity information for a particular user. Applications acting as more than one user (as is common, for instance, in access control test code) must currently be sure that they load the ACMs for each user that they are acting as prior to performing access control operations as that user.

[2]Negative caching can produce unexpected results, particularly in code responsible for initializing access control and writing policy markers – the `NamespaceManager` may have cached the non-existence of a policy marker at a node *n* which that code then writes a policy marker at. Careful use of `NamespaceManager.clearSearchedPathCache` can be used to get around such problems.

4. Key Wrapping

This section describes how symmetric and private keys are stored wrapped in other symmetric keys or encrypted under public keys.

For a given node in the name tree, one or more wrapped keys may be stored and used to either decrypt content at that node and below, (e.g. using the approach described in section 2), or used to unwrap (decrypt) keys stored lower in the name tree, or used to derive keys (e.g. using key derivation, see section 5) that are then used to encrypt content or wrap keys for nodes at or below this one in the name tree.

Such wrapped keys are stored in standard CCNx content blocks marked as type DATA. The absence of keys (signalling that they may be somewhere else) will be marked with blocks of type GONE, and blocks of type LINK may be used by higher level protocols to refer to wrapped keys at other points in the namespace.

4.1 Specification

Each wrapped key must specify:

1. The binary identifier (e.g. SHA256 digest) of the key used to wrap this key, which should be used to unwrap it (OPTIONAL, see below).

2. The algorithm of the wrapped key, which determines how to decode and reconstitute that key for use, represented as an OID. For public key algorithms (where the wrapped key is generally a private key), we use the standard OID (e.g. RSA or ElGamal). For symmetric keys we use an OID that represents the algorithm together with a mode of operation. For some symmetric ciphers, this also implies the bit length of the key (though that is obvious from the data). For AES-CTR mode,

which has no generally-accepted OID, we use the following OIDs: `aes128-ctr`, `aes192-ctr` and `aes256-ctr`.

3. The algorithm used to wrap the key, represented as an OID. (OPTIONAL, omitted if default). The defaults are:
   - RFC 3394 key wrapping (if AES symmetric keys are used as the wrapping key), augmented by RFC 5649 [2] to handle wrapping of public/private keys that may not be a multiple of 8 bytes in length. RFC3394 parallels a NIST standard, and the augmenting I-D has a NIST co-author and NIST OIDs, so will likely be accepted by NIST as well.
   - RSA-OAEP for wrapping with public keys (if RSA public keys are used as the wrapping key).
4. A UTF-8 label indicating the intended use for this key (OPTIONAL).
5. The (binary) wrapped key.
6. If the key to be wrapped is (encoded) longer than the longest object that the wrapping key is capable of encrypting (e.g. if we are wrapping a private key in a RSA public key), we interpose a random nonce key, which we select by default to be an AES key. We encrypt this nonce key in the wrapping key, and then wrap the key to be wrapped in the nonce key. To allow decryption, we include the encrypted (wrapped) nonce key.

Algorithms are represented using standard object identifiers (OIDs), from the existing public OID tree. We currently represent them as OID strings. If we need to reduce space (we expect these objects to be much less than a packet in size) we can move to a DER-encoded binary encoding, which is compact and easily handled by the necessary software.

The key used to wrap this key (and hence necessary to unwrap it) may be obvious from the name of the content, or identified inside the content if using it in the name would be somehow less optimal. Typical names for such wrapped key content would be the SHA256 digest of the key used for wrapping/unwrapping (in the case of public key wrapping, the digest of the public key). If this is not used as the last component of the name prior to any versioning information (if present) or segmentation information, then the key identifier should be stored inside the wrapped key data.

If a public key is used as the wrapping key, we expect most of the key components above to be optional. If they are not, a two-level wrapping may be needed, because the data to be wrapped will exceed the length of data that can be encrypted with RSA-OAEP given sensible RSA key lengths. Similarly, if we are wrapping a private key, we may need to wrap it under a symmetric key and that under a public key to deal with key length restrictions. In such cases, all key wrap material is stored at the same point in the name hierarchy, and the key identifiers/content names are used to chain them together.

To summarize, the representation of wrapped keys is:

```
WrappedKey ::=
        WrappingKeyIdentifier
        KeyAlgorithm WrapAlgorithm
        Label EncryptedNonceKey
        EncryptedKey
WrappingKeyIdentifier ::= BLOB
KeyAlgorithm ::= UTF-8
WrapAlgorithm ::= UTF-8
Label ::= UTF-8
EncryptedNonceKey ::= BLOB
EncryptedKey ::= BLOB
```

The intent is to pick wrapping algorithms which are secure, standardized, and easily available for wrapping and unwrapping in standard crypto libraries. We will need to specify additional default choices for public key algorithms other than RSA. (Though with public keys, key wrapping tends to need less special treatment compared to other types of encryption, so the sensible defaults are more obvious.) As such we wrap only the keys, which allows the key wrap algorithm to be specified in the clear if need be, and the standard key wrap routines (which tend not to take associated data to wrap, even if the specification allows it) to be used.

We represent wrapped keys as ccnb-encoded XML objects.

5. Key Derivation

This section describes the algorithms used to go from a content key, associated with a particular set of content items, to an individual object key used to encrypt one of those items. The details of how to then encrypt that (potentially segmented) item are described in section 2.

5.1 Basic Idea

The goal of the key derivation scheme is to efficiently generate a set of encryption keys and parameters for a set of closely related data that is likely to often be encrypted and decrypted together (e.g. a file and its associated intrinsic metadata), and to be subject to the same access control policy. It also serves the purpose of ensuring that a given combination of key and parameters is never used twice, both by providing deterministically-generatable (and hence not needing to be stored explicitly) but unpredictable IV/counter source material, and key diversity, protecting a non-security savvy programmer from shooting themselves in the foot by reusing keys.

Taken together with a high-level mechanism for mapping keys onto CCNx namespaces, such as CCNAccess-Control, this provides an efficient set of mechanisms for encryption-mediated access control of content.

5.2 Specification

A cohort of objects in this sense is usually defined as a namespace. For example, given a file or other piece of CCNx content, `/bigco/div/finance/report.xls`,

we can expect to see that high-level (user-oriented) content name act as the root of a content subtree. That subtree will contain a number of versions of the file, e.g.:

```
/bigco/div/finance/report.xls/%FD%
12%05%E6W%7De%DB
/bigco/div/finance/report.xls/%FD%12%
05%E6bE%E8%8B
```

Those versions will have associated metadata, e.g.:

```
/bigco/div/finance/report.xls/%FD%
12%05%E6bE%E8%8B/_meta_/STAT_INFO
/bigco/div/finance/report.xls/%FD%12%
05%E6bE%E8%8B/_meta_/AUTHOR
```

as defined in the MetadataProfile. Those versions and metadata items will have content associated with them (and the metadata items may themselves have versions). If that content is large enough, it will be broken into individual segments such as:

```
/bigco/div/finance/report.xls/%FD%
12%05%E6bE%E8%8B/%F8
/bigco/div/finance/report.xls/%FD%12%
05%E6bE%E8%8B/%F801
```

Let $\mathbf{K}_c$ denote a content key associated with a particular version of a file and its intrinsic metadata. Our goal is that the object key $\mathbf{K}_O$ for a given object can be constructed given its corresponding $\mathbf{K}_c$ and *any* single block of object **O** (or no block of **O** at all, given a-priori information about **O**). This allows decryption to start at any point in a large piece of content, without having to retrieve a special block with additional encryption information, other than the key blocks necessary to retrieve $\mathbf{K}_c$.

To achieve this, we use a standard key derivation function, or **KDF**, to derive the object keys $\mathbf{K}_O$ and initialization vector $IV_{\mathbf{O}}$ (or counter seed material) used to encrypt each object as follows:

$$\mathbf{K}_O||\mathbf{IV}_{\mathbf{O}} = \mathbf{KDF}(\text{Label}, \mathbf{K}_c, \text{Data}_{\mathbf{O}}, h),$$

where Label indicates the function of this key, $\text{Data}_{\mathbf{O}}$ is a set of auxiliary data specific to **O**, and h is the desired total length of $\mathbf{K}_O$ and $\mathbf{IV}_{\mathbf{O}}$ in bits.

We define $\text{Data}_{\mathbf{O}}$ as the concatenation of the ccnb encoding of the name of **O**, prior to segmentation (so the prefix of the name of any specific block of **O** prior to segmentation information and implicit content digest), and the PublisherID of the publisher of interest of **O**. This formulation has the advantage of being universal across all blocks of **O**, and can allow a querier to begin computing decryption keystream material while waiting for blocks to arrive. An alternative formulation would take the concatenation of the name of **O** and its entire encoded `SignedInfo`. This would seemingly have the advantage of including not only publisher information but timestamp and type as well. However, this alternative suffers from several problems. First, a block must be retrieved prior to keystream computation. Secondly, the segment-specific portions of the name of that block must be excised from the value presented to the **KDF**. (We want to use the **KDF** once per object **O**, and use the more efficient approach described in section 2 to use that key to encrypt the segments of **O**). Third, different segments of **O** may in fact have different timestamps. Rather than forcing all of the segments of **O** (which might be multitudinous and written over a long time period) to have exactly the same `SignedInfo` content, we adopt the simpler and more robust approach above.

As an initial default, we use the counter mode **KDF** defined in NIST special publication 800-108 [1], which can be described as follows:

Fixed values:

- **PRF**: a pseudo-random function,
- h: the length of the output of **PRF** in bits, and
- r: the length of the binary representation of the counter i.

Inputs: K_I, Label, Context, and ℓ, the desired output length of the **KDF** (keys and IVs) in bits.

Key derivation process:

1. $n := \lceil \ell / h \rceil$
2. If $n > 2^r - 1$, then indicate an error and stop.
3. Result$(0) := \emptyset$.
4. For i = 1 to n, do
   - $K(i) = \mathbf{PRF}(K_I, [i]_2 || \text{Label} || 0x00 || \text{Context} || [L]_2)$
   - Result(i) = Result$(i-1) || K(i)$
5. Return $\mathbf{K_O} || \mathbf{IV_O}$ as the leftmost ℓ bits of Result(n).

In our CCNx implementation, the pseudo-random function **PRF** is specified to be HMAC-SHA256. An alternative would be to use CMAC, as it removes the dependency on a message digest algorithm and should be equivalent in speed, or even better the faster Rogaway-Shrimpton **KDF** defined in RFC5297. However, the choice of HMAC-SHA256 is conservative and matches the schemes available in all up-to-date crypto toolkits, while CMAC support is not yet common.

The value i is always represented by r bits. Label is represented in UTF-8, does not include a null terminator, and is defined by the particular key use context. If no label is specified, 0 bytes are used as label. Context refers to $\text{Data}_\mathbf{O}$, described above.

Those object keys and IV/counter values are then used to encrypt the actual blocks of content of each object in the cohort as described in section 2.

This **KDF** can also be used to derive hierarchical keys used only to wrap or derive other keys. In this case, the length of $\mathbf{IV_O}$ is 0, and only key material is derived.

6. Group Based Access Control

We describe a scheme for achieving group-based access control by encryption. Although our description is in the context of the CCNx naming hierarchy, our scheme, in fact, applies to general hierarchies of named content.

As this scheme is under active development, some of the implementation details are currently represented only in the code. We expect to fill in the details in the specification as they become clearer.

For background on, and examples of, encryption-based access control schemes, we refer the reader to [4, 5]. We believe, however, this is the first encryption-based access control scheme supporting a) a notion of Groups, and b) denying principals given read access at a higher level of a content hierarchy (e.g. /test) read access at a lower level of the content hiearchy (e.g. /test/foo).

6.1 Overview

We support a simplified access control model for nodes in a hierarchy of named content. This could be CCNx content or standard file system content. Our model supports a notion of named groups composed of users and other groups, and grants of access can be made to either such groups, or directly to individual users. Publicly readable data is unencrypted.

We allow only a simplified set of permissions, relatively to traditional file systems – users and groups can be granted read, write, or manage (change permissions) access to particular trees of content. We assume that write access implies read access, i.e., if you can write a document you are able to read it. This means that we don't support write-only access, but do support read-only access. Without loss of generality, management/ownership permission (ability to write access control lists) implies write access – if you are able to change the permission, you can always add yourself to the list of writers.

We specify access rights, or access control lists (ACLs) at nodes of the name tree. In the CCNx name tree, all nodes, including interior nodes, may contain content. In theory, we can support ACLs for a piece of content, or a specific version of the content. However, we do not support ACLs at the granularity of individual content segments, i.e. one is unable to specify different ACLs for different segments (fragments) of a single file. In practice, we may disallow specifying ACLs at levels in the hierarchy likely to cause significant confusion and unnecessary complication for users.

6.1.1 Access rights inheritance and overriding

We assume a modified hierarchical access control (HAC) model. If no specific rights are specified for a node, it inherits the settings associated with its parent. Unlike a

traditional encryption-based HAC model, however, we allow child nodes to set their associated rights arbitrarily – at any child node, you can attach an arbitrary ACL indicating a new set of permissions to be associated with that node and its children (until overridden lower in the tree by another ACL). A traditional HAC model nests permissions, and says that if you have rights to a node **N**, you have those rights over all of its children; you can grant additional principals rights to children of **N** but you cannot take away rights granted at higher points in the tree. We offer greater flexibility to specify arbitrary ACLs throughout the tree, supporting use cases much closer to what individuals are accustomed to with traditional file systems.

We assume a standard delegation model for rights inheritance in the tree. If node **N** has an associated ACL specifying rights for **N** and its children, and a user changes the contents of that ACL to add or remove rights, we assume that change should apply to all of **N**'s children whose rights were previously controlled by ACL. We do not offer the user a choice to apply that change to only **N** (and its associated contents), as some standard file systems do. These choices seem merely to cause confusion and error in most cases. If a user wishes to change the privileges associated with one or more of the descendants of **N**, they merely have to associate a new ACL to that descendant.

6.1.2 Lazy revocation of read rights

We support only lazy revocation of read rights. When an access specification changes, we try to ensure that any new content written to affected points of the content hierarchy is subject to the new access policy – i.e. is inaccessible to principals whose (read) rights have been removed. We also ensure that principals who have access rights added (e.g. a user newly added to a group) gain the ability to read old content written before they were in the group. We believe this behavior most closely mirrors what users expect from distributed file systems. Limited-time access, such as having new principals gain access to new but not old data is interesting for some applications, but should be supported outside this model in the context of those applications.

In distributed environments, content deletion or revoking access to old content is not always possible. In particular, previous rights holders may have cached a copy of the old content already. For this reason, in CCNx, we cannot guarantee clean up of old contents. However, in cooperative environments (e.g., within a corporation), we support cooperative deletion – where you can ask people to delete things for you, and hope that they will cooperate and do so. Using cooperative deletion, we can ask people to clean up old content keys, so that revoked users no longer have access to content that they did not choose to read while their rights were in effect, as the keys needed will no longer be available. We expect this model is actually quite sufficient for most cases (and the cooperative deletion functionality may only be necessary primarily for regulatory compliance). We can also combine encryption-based access control and token or capability-based access control mechanisms to enforce access revocation: e.g., former employees no longer have the ability to access data behind a firewall.

The choice to support only lazy revocation also means we never have to re-encrypt data; cooperative deletion is achieved by deletion of keys. Re-encryption (essentially re-publication) of CCNx data is possible, but changes the semantics of that data, as the publisher that signs the re-encryption may not be the same publisher that signed the original data.

6.1.3 Enforcing write/manage rights

Write control – determining who is allowed to write a particular piece of content is done by verifying that said content was signed by a publisher who has write privileges for that part of the namespace. Management rights are essentially equivalent to having write access to ACLs; those are similarly enforced through signature checks. See section 6.2.11 for details.

6.1.4 Resolving offline conflicts

One challenge we face is how to merge and resolve conflicts that arise from independent offline changes made to the same ACL or group specification. For example, one may change access control settings on shared content stored on your laptop while travelling on an airplane, but then need to sync with other copies of that data stored elsewhere, whose access control settings may have been changed independently. The scheme described in this section ensures that changes to separate parts of the content tree are independent – i.e. changes to access rights in non-overlapping subtrees can simply be merged. Changes to overlapping subtrees must go through a deterministic resolution protocol to resolve any conflicts and make sure that the result represents the net access policy intended by all sets of changes to be resolved. This part of our protocol is only loosely specified; it is likely to be implemented only as part of a larger resolution and synchronization infrastructure.

6.2 Basic Scheme

We consider two hierarchies: a top-down content hierarchy, and a bottom up group hierarchy. In the group hierarchy, users are leaf nodes, and in general can be considered as special, one-member groups. Groups can contain other groups as well as individual users.

We speak of the parent of a node as being above it in the hierarchy, and its children being below it; but the two hierarchies aggregate in different directions such that a content tree has its leaves (content) at the bottom (childward) and a group tree has its leaves (users) at the top (parentward).

6.2.1 Keys

We define the following types of keys:

- group key: a public/private key pair associated with a named group, denoted **GPK** and **GSK**
- user key: a public/private key pair associated with a named user, denoted **UPK** and **USK**. (For this initial treatment, we consider users to have one key pair each.)
- node key: a symmetric key associated with a content node, denoted **NK**.
- data key: a symmetric key used to derive keys to encrypt data, denoted **DK**.

We may also use a key derivation function, or **KDF** to derive additional keys deterministically from known keys, limiting the number of keys that must be stored to securely encrypt a content hierarchy (see section 5).

6.2.2 Representing Groups

Let **G** denote a group with group keys **GPK** and **GSK**. We encrypt the group secret key **GSK** under the public keys $\mathbf{PK}_i$ of its children nodes ($\mathbf{GPK}_i$ for groups or $\mathbf{UPK}_i$ for users).

$$\text{For all } \mathbf{PK}_i \in \text{Children}(\mathbf{G}) : \text{Encap}_{\mathbf{PK}_i}(\mathbf{GSK}_{\text{parent}})$$

where Encap is an encapsulation operation defined as follows. Let WK denote a random symmetric wrapping key:

$$\text{Encap}_{\mathbf{PK}_i}(\mathbf{GSK}_{\text{parent}}) = [E_{\mathbf{GPK}_i}(WK), E_{WK}(\mathbf{GSK}_{\text{parent}})]$$

For a child node which is a group belonging to the parent group, we encapsulate the group secret key **GSK** under the **GPK** for that child group. For a child node which is an individual user belonging to the parent group, we encapsulate **GSK** under that user's public key.

6.2.3 Representing ACLs on Nodes

For each subtree in the content hierarchy participating in this access control scheme, we associate an ACL with the root node of that subtree. When one marks a tree as access-controlled, we can initialize its root ACL either using a default ACL, or using an ACL representing public access (see below). All children nodes initially inherit the ACL of the root node.

To change the access rights associated with some subtree of this access-controlled tree from those specified for the root node, a user associates an ACL with that subtree. For example, if the root node / were marked as access controlled, with a default ACL, to attach different access rights to a subtree rooted at `/project/secret`, we would attach an ACL to `/project/secret` specifying those rights. The root node /, and all of its subtrees other than `/project/secret` would still be controlled by the root ACL associated with /.

To effect read access, we associate a node key, **NK** with each node **N** that has an ACL. That node key is used to derive the keys used to encrypt data keys (**DK**, the keys used to actually encrypt content) for nodes subject to that ACL (i.e where that ACL is the nearest ACL above that node in the naming hierarchy).

When a node **N** is given an ACL at node creation time (either the root node, at initialization, or a child node when an ACL is added to that node at node creation time), we do the following:

- Generate a new random node key **NK** for that node **N**.
- Encrypt that node key with the group public key of every group in its ACL: for all $\mathbf{GPK}_i \in ACL_{\mathbf{N}}$: $E_{\mathbf{GPK}_i}(\mathbf{NK}_{\mathbf{N}})$.

In theory, ACLs may occur anywhere in the content hierarchy. In practice, we restrict this by disallowing ACLs and node key changes on individual segments of a segmented object (or even a single-segment object.) The last place where it is legal to hang a node key is the level in the hierarchy just above segmentation. We use the node key to derive the encryption keys to encrypt the contents of the corresponding node. To do this, we use the key derivation mechanism (described in section 5) to generate a key and initialization material from the node key, and basic encryption (described in section 2) to use that key and **IV** to encrypt individual content segments.

6.2.4 ACL Inheritance

There are two approaches to handling inheritance of ACLs on nodes which inherit their policies from their parents and do not override it.

Scheme 1: each such child node generates a new random node key **NK** and stores it encrypted under their parent's node key:

$$\mathbf{NK}_{\text{child}} \in_R \{0,1\}^* \ ; \ E_{\mathbf{NK}_{\text{parent}}}(\mathbf{NK}_{\text{child}}).$$

Scheme 2: each such child node does not explicitly select a new random node key **NK**, but derives a new node key **NK** from their parent's node key and information particular to that node, using a key derivation function, or **KDF** (see section 5):

$$\mathbf{NK}_{\text{child}} = \mathbf{KDF}(\mathbf{NK}_{\text{parent}}, \text{info}_{\text{child}}).$$

These two approaches result in different properties. In scheme 1, every node has an explicitly stored node key, and it is easy to find the key for a node without timeouts. In scheme 2, only nodes with ACLs have explicitly stored node keys, and the node keys for other nodes must be obtained through one of the following ways: 1) by asking for them and on time-out, asking for an ancestor key to derive the node key for the current node; or 2) by marking them as explicitly absent, either by publishing GONE blocks at those locations or, better, publishing a link to the parent node key in effect. However, if one has reason to believe that there are no ACL changes at this point in the hierarchy, one can go ahead and derive the current node key. The load on the publisher is effectively the same, as in order to avoid timeouts, they must publish something (a node key, a GONE block, or a link) at every node in the hierarchy to give information about the node key to use. (One could potentially publish a GONE block for an ACL and achieve the same effect.)

In scheme 1, one knows nothing about node keys one has not explicitly pulled. This has the advantage of giving reassurance that someone whose access is revoked cannot get access to old data for which they have not yet pulled key blocks. In scheme 2, one can generate large numbers of node keys one has not explicitly had to pull. This may offer increased efficiency in cases where one knows one doesn't have to incur a network round trip to pull them.

As the node keys in question are symmetric keys wrapped in symmetric keys, the computational cost of unwrapping a retrieved symmetric key and deriving one with a **KDF** is likely largely equivalent.

We discuss below the implications of the choice between these two schemes on the cost of key updates as access control settings change.

6.2.5 Encrypting Content with Data Keys

Data keys, **DK** are used to encrypt individual "files" or content. In a standard file system, they are used for file-level encryption. In CCNx, they are used to derive the keys for a family of data blocks. In general, data keys are stored at the level of a version of a file, and may cover metadata associated with that version as well as the file data itself. To do this securely, we use the algorithm specified in KeyDerivation to derive the counters/IVs and keys used for encrypting data blocks.

The purpose for having data keys rather than directly using node keys to encrypt content is three-fold: 1) The use of data keys can introduce more randomness into keys used to encrypt data, and reduce the number of times the same key is reused. 2) In some scenarios, this can allow more fine-grained access control – if data keys are used, I may give you access to a specific file by giving you the corresponding data key. If node keys are used to encrypt, I will have to give you the corresponding node key, which also gives you access to all children nodes whose ACLs inherit that of the current node. 3) It allows deterministic retrieval of the key used to encrypt each set of data; that key can then point to the node key **NK** used to encrypt it. Without this, we would need some way of figuring out what version of the node key **NK** was in use at the time a given piece of data was encrypted, in order to figure out how to decrypt it.

In essence, this is equivalent to using scheme 2 of section 6.2.4 for data keys, regardless of which scheme is used for node keys.

6.2.6 Adding groups or users to an ACL.

When a node's ACL changes, appropriate actions need to be taken depending on what changes happened. The easy case is when groups or users have been added to the ACL, but no user or group has had their access revoked. All we need to do in that case is encrypt the corresponding node key under the group or user keys of the added groups or users. The cost of adding n new users or groups to an ACL is proportional to the number of new users or groups (n encryptions) and does not depend on the size of the ACL prior to the addition.

6.2.7 Revoking access

If a group or user has write or management access revoked, but retains read access, we merely need to update

the ACL. (We may eventually need to tie the time of update to a representation of the state of content, to prevent revoked writers from forging content with dates indicating they were written before their write access was revoked, but that is under development).

If a group or user has their read access revoked, the node keys of all children in the subtree controlled by that ACL need to be changed (until overriden by another ACL). In our lazy revocation model, old files do not need to be re-encrypted, but we need to ensure that new files and folders added in the future are encrypted under the new node keys.

This work can potentially be deferred to when a writer needs to write data to a child node (denoted $\mathbf{N}'$). That writer will need to generate all node keys on the path from $\mathbf{N}$ to $\mathbf{N}'$.

If cooperative deletion is available, the old encrypted node key can be deleted. In the case of scheme 1 (see section 6.2.4), this gives some assurance that a revoked user who has not previously tried to read content is unlikely to have its key. Basically, a revoked user needs to cache n node keys to be able to decrypt all old contents in a subtree of size n. In scheme 2, a revoked user only needs to cache 1 node key, and he will be able to decrypt all old contents of that subtree even after revocation.

Scheme 1. If we are using explicitly stored node keys at each node (scheme 1), on ACL change we must regenerate all node keys for all of the descendant nodes whose access policy is determined by the ACL at node $\mathbf{N}$. This is difficult to do. Either the manager changing the ACL at $\mathbf{N}$ must attempt to enumerate all of the descendants of $\mathbf{N}$ and change their node keys, or we must implement a lazy update scheme where a writer needing to write data to a child node $\mathbf{N}'$ updates the node key at $\mathbf{N}'$ and its ancestors up to $\mathbf{N}$. The latter approach requires significant bookkeeping to track what nodes have had their node keys updated.

Scheme 2. If we are using hierarchically derived node keys (scheme 2, above), all we need to do is generate a new node key at the node whose ACL has changed (denoted $\mathbf{N}$). When writers need to write to a child node (denoted $\mathbf{N}'$) which does not have its own ACL but is instead subject to the ACL at $\mathbf{N}$, that writer can compute the derived node key on demand based on the new node key.

Other scheme. We could attempt an intermediate representation, where only nodes with ACLs have node keys, but each node stores a link to point back to the ancestor (or to itself) on whose node key it depends. This allows writers to locate the node key $\mathbf{N}$ to use with O(1) operation, as in scheme 1, but simplifies key update. The problem with such a scheme is finding (enumerating) all descendant nodes and updating their links when a new ACL is added to the tree, or an ACL is deleted. This approach has the potential to incur many timeouts as writers must check every node on the path above the point at which they are writing to see if it already exists and has a link to its node key.

Implementation. Given its simplicity and lower cost, we choose to implement scheme 2 for managing node keys. A manager revoking access to a group or user at node $\mathbf{N}$ merely updates the ACL for node $\mathbf{N}$, generates a new node key $\mathbf{NK}'$ for $\mathbf{N}$, and then encrypts $\mathbf{NK}'$ under the current version of the public keys for all the groups and users remaining on the ACL for $\mathbf{N}$. The manager also needs to encrypt the previous node key, $\mathbf{NK}$, under $\mathbf{NK}'$, so that newly added readers can access old content. The cost of this operation is the cost of generating a new node key, and the cost of encrypting this new node key under the public keys of all the entities in the ACL (this cost is linear in the size of the ACL).

6.2.8 Adding an ACL to a Node

Adding an ACL to a node that previously inherited its parent's ACL is effectively the same as the access revocation case above. We generate a new node key, encrypt that new node key under the public keys in its ACL, and encrypt its old node key (derived or stored) under the new node key. If the newly interposed ACL only extends the access rights granted by the parent ACL (but does not revoke any right), then the cost of interposing the ACL is the same as that of adding a user or a group to an ACL (see Section 6.2.6). If the interposed ACL revokes some access rights, then the cost of interposing the ACL is the same as that of revoking some ACL access (see Section 6.2.7).

If we use pointers to indicate for a node where its ACL lives, adding an ACL to an intermediate node requires fixing all of the pointers in children nodes. This suggests that either not having any such markers, and relying on name enumeration to efficiently determine where an ACL resides, or using simple GONE markers to indicate that there is no ACL here and that one should move up the tree without incurring a timeout (or both) may be the simplest option.

6.2.9 Adding a User or Group to an Existing Group

This only requires encrypting the group's secret key **GSK** under the new user or group's public key. The cost of adding *n* new members to a group is proportional to the number of new members (*n* encryptions) and does not depend on the size of the group prior to addition.

6.2.10 Revoking a User or Group from an Existing Group

1. Generate a new **GPK**/**GSK** pair, and encrypt the **GSK** under the public key of the remaining members of **G** (users or groups).

2. Encrypt the old **GSK** under the new **GPK**, so that new members can retrieve old data if necessary.

3. If we allow nested groups, we need to change the group keys of all the ancestor groups of **G** as well (all the groups of which **G** is a member).

4. We need to ensure that each node whose ACL contains **G** or its ancestors uses the new key pairs for **G** (or its ancestors) when encrypting new node keys. This is trivial, by having the ACL in effect reference the latest version of each group's public key.

5. Replace all of the node keys **NK** to which **G** has access, either directly or via membership in its ancestor groups, with new versions encrypted under the new group public keys; this ensures that new content is encrypted under new node keys and is not accessible to revoked members of **G**.

The cost of 1) and 2) is proportional to the number of group members. Similarly, the cost of 3) is proportional to the number of members of the parent groups of the group.

The most expensive operation in this set is the last one – updating all of the "dirty" node keys to new versions. Rather than incur the significant cost of enumerating the full tree to find the nodes to which **G** has access, we defer rekeying operations to when writers add new files or folders. This ensures that new content is protected by new keys, preventing access by revoked members, but only does the work of updating keys in places where it is needed - if no new content is ever added to node **N**, we do not incur the cost of updating its key.

In this approach, each writer adding data at a node $\mathbf{N}'$ must find the node key **NK** corresponding to the ACL in force at $\mathbf{N}'$, located at a node **N**. The writer must then determine whether that node key is "dirty" – whether the public keys of the groups (or potentially users) that node key was encrypted for have changed since it was written. If we represent at **N** what version of each group **G**'s public key its ACL is encrypted for, and have each writer check to see whether or not that is the latest version of **G**'s public key. If so, that writer makes a new node key $\mathbf{NK}'$ at **N**, and encrypts it under the latest versions of the keys listed in the latest version of **N**'s ACL. (This requires that writers, not just managers, be allowed to update node keys.) The writer also encrypts the previous version of **NK** under the new $\mathbf{NK}'$, so that if new readers are added to the ACL, they will have access to old content.

An additional advantage of this approach – having writers, not the group managers that change a group's membership, update node keys is that writers are guaranteed to have access to the node key they need to update, while group managers very likely do not.

6.2.11 Enforcing Write/Manage Rights

Enforcing access controls on writes requires determining that the key that signed a given piece of content (or an ACL, for management rights) is allowed to do so – either on the list of allowed writers or the allowed managers for the ACL in force for the node being updated at the time of update.

Making this verification requires first verifying that the ACL itself is valid - that it was signed by someone with management rights according to either the parent ACL that it was created under, or the previous version of the same ACL. This can incur work linear in the number of ACL versions, and increases if the manager is specified as part of the group as one must verify the integrity of group membership. We can reduce the cost by caching validated data or signatures, or by re-signing intermediate steps with a root of trust for the user (either a trusted authority or the user themselves) to mark that intermediate ACL versions can be directly trusted as valid, or that their validity has been previously determined.

We also need to verify that contents and node keys have been written by legitimate writers. This involves verifying the ACL itself, and checking that the content publisher is authorized to write by the ACL.

Prevention of "write rollback" attacks – where a user or group that has had their write access to a piece of content revoked writes new content that claims to have been written at a previous time (before their write access was revoked), in the absence of certified time, requires a

method for validating overall temporal state. For example, by constructing the notion of a version of a content tree, that represents the state of all the content in that tree at a given point in time (much like the notion of a version in the source control system git), one can mark the "time" a piece of content was written by incorporating this cryptographic representation of this point in the write stream into the content. That content is then itself included in the future representation of time used in other writes. This makes it impossible to go back and insert content from previous times into the write stream. Making this work in practice requires a) a notion of aggregated versioning, and b) an approach to reconciling writes made to multiple disconnected repositories who have a different notion of time. These are areas of active research, and will be incorporated into future versions of access control mechanisms.

Enforcement of write/manage rights is not yet implemented.

6.2.12 Group-Related Rights

In CCNx, groups are stored as special nodes in the name tree. Therefore, the ability to modify group membership translates to the write right for the content node corresponding to the group. Similarly, "group management right" – the ability to modify the set of writers, translates to the management right of the content node corresponding to the group.

Enforcement of write control over group membership is not yet implemented.

7. Implementation

We need to generate efficient data representations for access control-related items that allow retrieval and update of necessary data in a minimum number of round-trips and packets. We have implemented a prototype of this access control scheme in the CCNx Java library (it will be ported to C when stable). We describe in this section the implementation of both the overall access control framework (section 3.1) and the particular group-based access control scheme we have implemented using it (section 6). Code can be found in the `org.ccnx.ccn.profiles.security.access` package. Most operations are mediated by `GroupAccessControlManager`, the group-based access control-specific subclass of `AccessControlManager` (see section 3).

As noted in section 3.2 each namespace under access control may be marked with the desired access control scheme to apply to that namespace, the subclass of `AccessControlManager` to instantiate to manage access there, and any additional parameters necessary to specify to allow that type of access control manager to operate in that namespace.

Each namespace participating in group-based access control has the root of that namespace marked with a policy marker indicating that `GroupBasedAccessControlManager` or a subclass thereof should be instantiated to manage access control for names in that namespace (under that node in the name tree). Such a policy marker also provides a set of `ParameterizedNames` – names indicating where the keys for Groups and Users participating in access control and defined under this namespace, can be found. Such a `ParameterizedName` has two components – the "prefix" or fixed portion of the name between the namespace root and the variable group or user name, and the "suffix" – the portion of the name (if any) between the individual group or user name and the actual encryption public key of that group or user. For example, if we define that the namespace `/bigco/finance/` is to be protected using group-based access control, we might specify prefix of a `ParameterizedName` for group definition of "Groups", meaning that groups are defined under the prefix `/bigco/finance/Groups` – e.g. `/bigco/finance/Groups/Managers`, `/bigco/finance/Groups/Accountants`, and so on. If the policy also specifies a suffix for that group namespace of "EncryptionKey", then the actual public key of those groups would be stored at `/bigco/finance/Groups/Managers/EncryptionKey` and `/bigco/finance/Groups/Accountants/EncryptionKey`, each followed by a version. If no such parameterized names are given to control the locations at which group and user keys are expected to be published, they are placed at the default locations mentioned below.

Note that a given namespace may have more than one location at which Group and User keys can be published. Also note that in multi-location access control, where more than one namespace is subject to access control at once, each namespace under access control will likely have its own location for defining groups and users (as each namespace usually represents a separate sphere of control, managed on its own). An access control policy (ACL) specified under namespace N may refer to principals (groups or users) defined under namespace N', and

managed by a separate `AccessControlManager` instance.

Operating in such a circumstance requires that the `AccessControlManager` for each protected namespace be loaded so that each can be queried to resolve references to principals defined in its namespace, and to retrieve the appropriate name parameters necessary to obtain their public keys. As groups and users are referred to by string "friendly" names, which might collide, they are separated by a "distinguishing hash" which separates friendly prefixes defined in namespace N from those defined in namespace N' (see references to `distinguishingHash` in `GroupManager` and `KeyDirectory`). Use of this feature allows ACLs defined in namespace N to refer to principals (groups or users) defined not only in N but also N' and so on.

When a user wishes to define a new Group, they must specify in what namespace it should be defined in; by defining it using the `GroupBasedAccessControlManager` controlling that namespace (and its corresponding `GroupManager`). (While each protected namespace can have more than one location for defining groups, they typically won't, and the ability to have more than one may be removed in future design iterations.) Software acting on that user's behalf – e.g. a GUI designed to make the group management process easier – is likely to choose for the user the appropriate location in which to define their group. For example, a group first defined to protect a subtree of content in namespace N is likely to be best defined in the group namespace of N; a group defined to contain members from namespaces A, B and C all of which are in some sense "members" of a shared namespace D is likely best defined in D. However, such policies are specific to the set of use cases operating for particular classes of content, and as such are not prescribed by the CCNx library.

7.1 User Keys

We identify users in our access control system by public keys. These public keys may evolve over time and can be versioned. They can be associated with a friendly name either by being published under a CCNx from which software constructs friendly name, or through a trust management infrastructure that allows users to associate names with keys, as in a Simple Distributed Security Infrastructure (SDSI).

In our implementation, we support an open universe of users – you can refer to any user's public key (as a CCNx Name) in a group membership specification. For convenience, we provide a **user namespace** – a location under which users can write (or link, but we'll start out with writing) their public keys. This namespace can be a corporation's existing directory of users (and will be, in most corporate contexts), but could also be a user's own address book or merely a peer-to-peer list of registered users. The point of having a list is not to rule out inclusion of other users, but to have an easy place for GUI tools to generate a list of available users for creation of groups, etc.

It is a trivial policy addition to require that users' keys be signed by a particular authority for use – for example, to use this scheme to represent access control in a corporation, where corporate employees (who have keys signed as such) are considered valid users to include in groups and access control lists, and others are not.

A **user namespace** is given to an **AccessControlManager** as a parameter; software wishing to offer the ability to list available users will publish a link to each participating user's key in this namespace. If unspecified, it defaults to:

```
<protected_namespace_root>/Users/
```

Where `<protected_namespace_root>` is the root of the namespace which is access controlled. If available, a namespace policy marker stored at this root can point to an alternate user (and group) namespace.

The user's public key is stored under the following. Its latest version is the up-to-date public key of that user.

```
<protected_namespace_root>/Users/
<username>
```

For now, we link to user keys using fully-demarcated names or publisher identifiers.

There may be an optional suffix separating the user friendly name and the actual key data (the default is empty); this is parameterizable and set in the policy marker for the namespace along with the prefix (default of "Users").

7.2 Group Keys and Membership Lists

In contrast to user keys, which are assumed to be by default managed by individual users, or by policy managed by users and signed by a trusted entity (e.g. in corporate access control), the access control system assumes that it owns the representation of Groups and their public keys, and manages those keys on behalf of the group

managers.

7.3 Group namespace

Groups are stored under a specified Group subnamespace for this access-controlled namespace. This can be a standardized group namespace used by a corporation for many applications, or a specific namespace targeted for this particular set of protected content. This group namespace is given to an AccessControlManager as a parameter. If unspecified, it defaults to

GROUP_PREFIX:
=<protected_namespace_root>/Groups/

It can be specified in the policy marker marking the namespace root. It is meant to be publicly visible, and enumerated by software.

7.4 Friendly names

Each group is associated with a human-readable "friendly name". Information associated with that group is stored under:

GROUP_PREFIX:
=<protected_namespace_root>/Groups/
<GroupFriendlyName(UTF-8)>

Friendly names of group members are obtained by looking at the keys pointed to by the group membership list (usually the last component of the name of that key, minus the version or segmentation information, or a user-specified name associated with that key).

7.5 Group membership lists

A group membership list is represented as a versioned collection, where the latest version indicates the current set of members. The collection contains links to the keys of the members of the group, authenticated using any of the means available to the CCNx links. Child groups are specified in the collection using the unversioned name of the group (under the same group namespace). Without active freshness protection, this is vulnerable to a rollback attack, but peer-to-peer techniques may be sufficient to defend against that well enough for standard levels of threat.

The group membership list is stored under the following name:

<GROUP_PREFIX>/<Group Friendly Name
(UTF-8)>/Members/<version marker>

which contains a collection containing a list of members of this group, as links pointing to either groups (as /, with optional authentication information) or users (as links to their user keys, for now)

7.6 Group public key

We store the group public key as a versioned named content item, whose segment(s) are of type KEY. This is stored under the friendly name prefix of the group. We write keys using PublicKeyObjects, which handle storing a versioned representation of a DER-encoded (X.509) SubjectPublicKeyInfo.

A group public key is stored under the following name:

<GROUP_PREFIX>/<Group Friendly Name
(UTF-8)>/KEY/<version
marker>/<segmentation>

which contains the DER-encoded public key for this version of the group membership, as a KEY block (or blocks)

There may be an optional suffix separating the friendly name and the actual key data (here KEY is the suffix); this is parameterizable and set in the policy marker for the namespace along with the prefix.

7.7 Group private key

We store the current group private key wrapped under a symmetric nonce (random) key (the private key wrapping key). For each member of the group, we store that nonce key wrapped under the member's latest public key (where the member can be a group or a user). Each key is stored in a WrappedKey structure, named with the cryptographic digest (key ID) of the key used to wrap it. We store these wrapped key blocks under the version of the public key whose private key is being wrapped. We also store a set of indexing links helping users find the keys associated with particular principals, and the versions of those principals keys being used.

We assume the universe of groups is small, so that each user can scan the available groups to determine whether they are a member of any of them. For each group they are a member of, they remember the group name to look for updates, and retrieve the private key associated with the group and cache it. Using CCNx's Interest mechanism, they can generate interests that will let them know if the group public key is ever updated.

<GROUP_PREFIX>/<Group Friendly Name
(UTF-8)>/KEY/<version

marker>/PrivateKey
contains the private key for this version of the group membership, as a WrappedKey, wrapped under a nonce symmetric key – the private key wrapping key.

<GROUP_PREFIX>/<Group Friendly Name (UTF-8)>/KEY/<version marker>/keyid:<KeyID>
contains a set of WrappedKey blocks containing the private key wrapping key for this group encrypted under the current public key of its members, identified by the key ID of the target public key.

<GROUP_PREFIX>/<Group Friendly Name (UTF-8)>/KEY/<version marker>/p:<PrincipalName>:<PrincipalVersion>
contains a link from a friendly identifier of the target principal (group friendly name or user friendly name as listed in the User or Group namespace), and the version identifier (as 12-bit timestamp) of their current key (at the time of wrapping), to the wrapped key block containing that principal's version of the group wrapping key.

<GROUP_PREFIX>/<Group Friendly Name (UTF-8)>/KEY/<version marker>/PreviousKey (optional)
contains a WrappedKey block containing the previous key for this group encrypted under the current key for this group. The label of this WrappedKey block provides a reference to the version of the group where the previous key is stored (to allow chaining backwards through multiple previous keys).

<GROUP_PREFIX>/<Group Friendly Name (UTF-8)>/KEY/<version marker>/SupersedingKey (optional)
contains a link pointing to the next version of this key (e.g. for this group); in a small number of instances this contains a WrappedKey block containing this key wrapped in the superseding key.

7.8 ACL and Node Keys

At a node **N** where an ACL is set, we represent the ACL in an access subnamespace. This is intended to be a namespace we expect only programs, not people, to look into.

<N>/_access_/ACL/<version marker>
points to a collection containing a version of the ACL information for this node. If there was an ACL at a given level but it has been deleted, we may store a block at:

<N>/_access_/ACL/<version marker>
for the latest version of the ACL.

An ACL is stored as a collection, a list of links where the links point to the keys of the principals listed on that ACL. The difference between read, write and manage privileges is represented as tagged roles for each link:

```
READ ::= r
WRITE ::= rw (implies read)
MANAGE ::= rw+ (implies read, write)
```

(An alternative would be to use 3 ACLs per node, one for each type of access.)

Each principal is listed on the ACL only once – as MANAGE rights imply WRITE access, and WRITE access implies READ access, this allows total specification of rights for each named principal.

<N>/_access_/DK
points to the data key used to encrypt the content at this node (the data key is stored in the repository).

<N>/_access_/NK/<version marker>
points to a collection containing a version of the node key information for this node. If there was a node key (and ACL) at a given level but it has been deleted, we store a block at /_access_/NK/ for the latest version of the node key. Under that version, we store a Superseded link pointing to the new node key in force at this node.

For each public key on the reader list, under the node key version we store the node key encrypted under that public key in a WrappedKey block. We represent these in the same way that group private keys are stored for each group member, as above (with links corresponding to principals and so on). Because the node key may change without a change to the ACL (e.g if a group membership changes), we need to evolve the stored version of the group key in some sensible way. Recall that under a given group key, we represent version n of the group as the group member list for that version, the group public key, and the list of WrappedKey blocks for the group private key. Similarly, we represent version n of the node key as a name:

NK/<version marker>
under which we store the wrapped keys for that version of the node key. When we update the access control list and node key in parallel (e.g. when we remove someone

from the access control list), we give the updated node key and access control list the same version identifier.

`<N>/_access_/NK/<version marker>/p:<PrincipalName>:<PrincipalVersion>`
contains a link from a friendly identifier of the target principal (group friendly name or user friendly name as listed in the User or Group namespace), and the version identifier (as 12-bit timestamp) of their current key (at the time of wrapping) to the wrapped key block containing that principal's encrypted copy of the node key.

`<N>/_access_/NK/<version marker>/keyid:<key_id>`
contains blocks all making up this version of the node key, each of which contains a WrappedKey object targeted for a particular entity on the ACL. For now we name them using the key digest under which the **NK** is wrapped.

7.9 Reader Operation

To use this access control scheme, readers must perform a series of steps (embodied in the CCNx library, and largely unseen by users).

1. A reader retrieves a piece of content, and discovers that it is encrypted (marked as type ENCR).

2. The reader consults their namespace manager to determine what profiles are in effect for the namespace in which the content resides, and is handed an AccessControlManager handling this namespace. That AccessControlManager (ACM) implements group-based access control.

3. The reader uses the ACM to retrieve the wrapped data key **DK** used to encrypt this content. The ACM, using its AccessControlProfile knows the naming convention where the **DK** will be stored, as a WrappedKey. This data key is wrapped in an Effective Node Key, the key derived from the node key in force at this node at the time the data was written, iterated using KeyDerivation to result in the effective node key for this node.

4. The reader looks in the wrapped data key to determine what node key was used to derive the key used to wrap it. If it has this key in its cache, it unwraps the data key and moves on to verifying write validity for the content. This is the expected "fast path" for normal operation – the remainder of the algorithm pursued by the reader is somewhat heuristic, and focused on building up a cache of node keys used to access content. This cache can be shared across applications and retained over time, maximizing the speed of decryption. (Another advantage of scheme 2 of section 6.2.4, where one node key is used to derive keys for many data items.) This does make it more difficult to revoke access for a user that retains control of their cache, but in practice access is revoked only rarely, and the win in efficiency is in many cases worth the risk. (In others, key caching may be disallowed.)

5. The user's AccessControlManager pursues a search strategy to attempt to unwrap the desired node key, or if that node key is superseded, to unwrap a later version of that node key (which can be used to unwrap previous versions of that node key, including the desired key). There are two large classes of algorithms that can be used to retrieve and decrypt the node key. All retrieved keys are stored in the user's key cache for reuse in further steps of the algorithm.

   - Forward algorithms look at the ACL associated with the node key, and attempt to find a principal on that list that is either the user themselves, or a group the user is in. The user builds up a cache over time of information about their own group memberships and the group private keys for those groups. Given that knowledge, the algorithm tries to pull a wrapped key block for the desired node key encrypted with that cached private key, and decrypt it.
   - Reverse algorithms look only at wrapped key blocks and the associated principal information, always attempting first to find a block wrapped with a key that is already in the user's cache (using the keyid information), and then using the principal information stored in links to try to find a group or user key that the user might have access to.

6. The user attempts to verify whether the first content block of the desired content was signed by someone that is both acceptable to the user, and allowed by the write privileges granted for this content. This requires looking at the ACL for

this node, unless we put an indexed representation of write privileges among the key information. Details of implementing write control are still in development.

If the node key cannot be retrieved, signals an access denied error.

7.10 Writer operation

The writer wishing to write a piece of content performs the following steps:

1. Generates a random data key for this content, and uses it to encrypt the content.
2. Stores the encrypted content at the desired name.
3. Retrieves the node key in force at this node using the search algorithm described above for readers. As write access implies read access, a user without read access will not be able to retrieve the correct key to encrypt the data. Determining that a user has write access and not just read access has to be done as a separate step, but the ability to correctly encrypt the data demonstrates that it was written by at least someone with the ability to read the data. If the node key cannot be retrieved, signals an access denied error.
4. Checks the versions on the keys of the principals that node key is encrypted for against the current versions of those keys that are available. If later keys are available (e.g. if a group has had a member revoked and updated its public key), generates a new node key version, and encrypts the new (now current) node key according to the latest version of the access control list for this content, and encrypts the old node key under the new node key as described above.
5. Derives the effective node key from the current node key using the algorithm in KeyDerivation. Uses that to encrypt the data key.
6. Stores the encrypted data key block at a point designated by the AccessControlProfile.

References

[1] Lily Chen. Recommendation for Key Derivation Using Pseudorandom Functions (Revised). NIST Special Publication 800-108. Available on the web at csrc.nist.gov/publications/nistpubs/800-108/sp800-108.pdf

[2] R. Housley, M. Dworkin. Advanced Encryption Standard (AES) Key Wrap with Padding Algorithm (RFC 5649). Available on the web at http://tools.ietf.org/html/rfc5649

[3] David McGrew. Counter Mode Security: Analysis and Recommendations. Available on the web at http://cr.yp.to/bib/2002/mcgrew.pdf

[4] M. Atallah, K. Frikken, M. Blanton. Dynamic and Efficient Key Management for Access Hierarchies. ACM CCS, 2005.

[5] Dominink Grolimund, Luzius Meisser, Stefan Schmid and Roger Wattenhofer. Cryptree: A Folder Tree Structure for Cryptographic File Systems. SRDS '06: Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems, 2006.