NDN, Technical Report NDN-0029, 2015. http://named-data.net/techreports.html
Revision 1: April 20, 2015

Public Key Management in Named Data Networking

Yingdi Yu
UCLA
yingdi@cs.ucla.edu

ABSTRACT

As every data is signed in Named Data Networking (NDN), public key management becomes critical. The public key management requires a well-defined certificate format and several systems and protocols to support certificate distribution and revocation. In this paper, we proposed the new NDN certificate format, discussed several approaches of serving certificates in NDN. We also discuss how to revoke certificates with the new certificate design.

1. MOTIVATION

In Named Data Networking (NDN), the content of a data packet is securely bound to the packet name through a signature. The validity of a data packet and its content depends on the validity of the signature. Since public key signatures are widely used in NDN, the public key management therefore becomes critical in NDN security.

From the perspective of a data consumer, the public key of the producer of a data packet must be authenticated before the signature verification. It is impractical to assume that a data consumer has pre-authenticated the public key of all the potential data producers. In many cases, a data producer may have to fetch a data producer's public key from the network on demand. Since the network is not assumed to be trustworthy in NDN, a data consumer must validate each public key that is fetched from the network. As a result, an assertion about the validity of a public key made by a trusted third party becomes important in NDN.

The most common assertion format is a public key certificate which securely associates a public key with a particular identity, or a name in the NDN context[1]. One of the purposes of this document is to introduce the format of the public key certificate in NDN and elaborate the reason of the design.

From the perspective of a data producer, it is very important to make its public key certificates available, otherwise its data will be rejected due to unverifiable signature. Although several certificate provisioning mechanisms have been proposed (such as NDNS [3], PIB [2]), they are all limited by the concept of publishing certificate explicitly through an application. Another purpose of this document is to propose an application-independent certificate provisioning mechanism.

Moreover, since data are directly signed, the data provisioning model of NDN becomes different from the one in traditional IP network and bring up several challenges in public key management in NDN. For example, how to check if signing key of a data has been not been compromised, how to check if a signature has not been revoked before the expiration timestamp how to maintain data after the signature expires, and etc. In this document, we will also discuss the possible solution to these challenges.

2. SECURITY MODEL

In IP network, the data security is based on the secure session through which data is transmitted. In order to start a secure session, an end point of the session needs to authenticate the other end point and negotiate a session key to encrypt data transmitted in the session. When public key cryptography is used in authentication, an end point must prove that it has a private key whose public key is bound to the name of the data producer.

Such a session based security model assumes that the host of data is also the producer of the data. As a result, when the data producer and data host are different, the data host must "pretend" to be the data producer. For example, in order to allow data consumer to establish a secure session to a Content Distribution Network (CDN) server, the data producer (a CDN customer in this example) either distributes its private keys on the data hosts (CDN servers in this example) or binds its own identity to the public key of the data hosts. Actually, such a session based security model has already caused some problems in distributing content in IP network [5].

Unlike IP network, NDN secures data directly. Data

[1]by identity, we mean an identifier of a data producer in the cyberspace rather than the producer's real world identity. The mapping from a cyberspace identifier to a real world identity is determined by applications or human users

is signed by its producer and can be directly verified using the producer's public key. As a result, the role of data producer and data host can be separated in data provisioning. If the credential about data producers are publicly available, the data hosts in NDN only need to serve the signed data rather than pretend to be the data producers.

3. CERTIFICATE IN NDN

A certificate usually associates a public key with an identity name through a signature. Similarly, the content of a NDN data packet is bound with the data name through a signature. If we assume that an identifier can be expressed as a data name, a data packet containing a public key bits is essentially a certificate. As a result, an NDN certificate is defined as a data packet with a special type of content.

3.1 Certificate Name

As a data packet, an NDN certificate must have a name. The certificate name plays at multiple roles in security and data fetching, thus it deserves further discussion.

3.1.1 What should a certificate name contain?

One major usage of certificate name is the `KeyLocator` of a data packet. The `KeyLocator` serves two purposes: first, as its name suggests, it can be used to retrieve the public key; second, it may also help data consumers to determine the legitimacy of the signer. Therefore, it would be necessary to encode the identity that is associated with a public key into a certificate name, so that data consumer can discard a data packet directly if the certificate name does not contains a correct identity. Moreover, if the prefix of a certificate name is the identity, certificate fetching can leverage the NDN packet forwarding system as fetching other normal data.

While a certificate associates a public key with an identity, the same identity may be associated with more than one public keys. For example, a user may deploy two keys associated to the same name on two devices, or a user may periodically replace its key while still keeping the same identity. As a result, a key id must be present in a certificate name in order to distinguish the certificates for different keys.

Moreover, the certificate of a public key may have more than one version due to the signature changes. For example, when the certificate issuer replaces its key, it should re-sign all the certificates which are signed with the replaced key. The re-signed certificate is essentially a new certificate due to the new signature. A version number in the certificate can effectively distinguish the two certificates with different signatures.

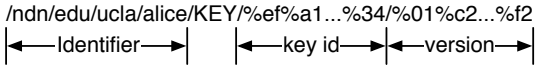

Figure 1: Certificate naming convention.

3.1.2 What should NOT be a part of a certificate name?

A certificate name should not include routing related information. Otherwise, one may have to generate multiple certificates for a single public key, if the public key is expected to be served from different networks. We will discuss how to route the interests for certificate in Section 4.

Similarly, a certificate name should not include the information about the serving application, in order to avoid generating multiple certificates if the public key is published by different applications. We will also discuss how to serve certificate from end hosts in Section 4.

Moreover, signer information should not be included in certificate name, for two reasons. First, a public key may only need one signer (or certifier) in most cases. For example, in a hierarchical trust model, the signer of a public key is usually pre-defined. In this case, it is redundant to include the signer name in the certificate name. Second, when more than one signers are required for a public key, it would be more efficient to retrieve all the signatures together rather than to retrieve the same public key with different signature separately (the multiple signatures will be discussed in Section 3.4.3). In this case, it is unnecessary to encode all signer's name in a certificate name.

3.1.3 Certificate naming convention

Based on the discussion above, we propose the naming convention for NDN certificates as shown in Figure 1. A certificate name starts with the associated identity name which is followed by a special name component KEY and another name component containing the key id. The last name component of a certificate name is the version number of the certificate.

There are two design choices for key id. One is a globally unique id, e.g., a SHA-256 hash of the public key bits. The other one is an relatively unique id (e.g., a timestamp), because the id has to be combined with the identity name to uniquely identify a public key.

The benefit of a relatively unique id is the size of the id. The key id may have only 8 bytes when it is expressed in terms of timestamp. In contrast, a key id in terms of SHA-256 hash will take 32 bytes.

The advantage of using key hash as the key id is that the certificate name implicitly fixes the associated public key. Such a property is more desirable, because it

```
MetaInfo:
  ContentType: key
  ContentFormat: X.509 Key
  FreshnessPeriod: 86400000 // ms
```

Figure 2: An example of MetaInfo of a public key certificate.

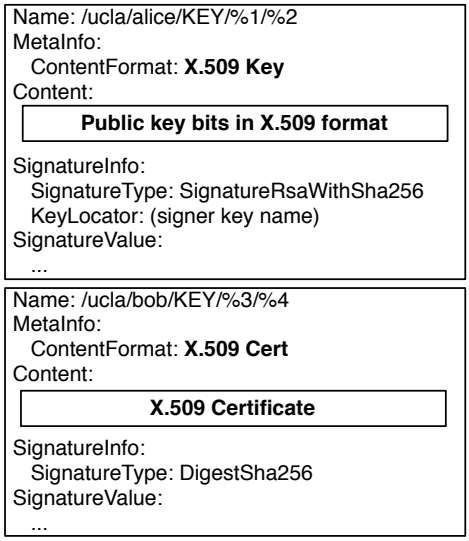

Figure 3: Example of certificate with different content format.

allows routers to check whether a retrieved public key matches the key hash in the interest name, thus partly avoiding cache poisoning attacks. Therefore, we recommend to use key hash as the key id when the length of the certificate name is less important.

3.2 MetaInfo

Public key is a special type of content in NDN. The value of `ContentType` in `MetaInfo` of a certificate is defined to be `Key`.

Besides `ContentType`, it would also be desirable to define another sub-field `ContentFormat` to describe the encoding format of the public key in the content. With such information, data consumer does not have to make implicit assumption about the public key encoding. The default public key format is `X.509 Key` (the public key format of X.509 standard [4]).

Since `ContentFormat` enable public keys to be encoded in a variety of formats. As a result, it is possible to introduce new public key formats and also allow reusing non-NDN certificates, e.g., X.509 certificate by explicitly set `ContentFormat` to `X.509 Cert`.

3.3 Content

Depending on the `ContentFormat` in `MetaInfo`, the content of a data packet could be the raw bits of a public key (e.g., `X.509 Key`) or a certificate in other format (e.g., `X.509 Cert`), as shown in Figure 3. If the content is the raw key bits, the validity of the certificate is determined by the signature of the data packet. If the content is some other certificate, the validity is actually determined by the signature of the inner certificate. Data consumers who can recognize the content format should validate the public key using the format specific logic. Data consumers should discard a certificate if its content format cannot be recognized.

3.4 SignatureInfo & SignatureValue

The SignatureInfo and SignatureValue matters only when the content of certificate is raw key bits (e.g., content format is `X.509 Key`). In current NDN packet format specification, `SignatureInfo` is defined with two sub-fields: `SignatureType` and `KeyLocator`. However, some other information about the signature is still missing.

3.4.1 Signature Validity Period

The first missing information is the validity period of the signature. Therefore, we propose a new sub-field `ValidityPeriod` in `SignatureInfo`. The `ValidityPeriod` consists of two timestamps: `NotBefore` is the timestamp when the signature becomes valid, and `NotAfter` is the timestamp after which the signature should not be considered as valid. The actual validity period of a signature subjects to the validity period of its signing key, that is, a signature can only be valid during a period which is the intersection of the validity period of keys on the signing chain.

Both timestamps in `ValidityPeriod` are specified using absolute timestamp. It may inevitably introduce the problem of clock synchronization. The clock synchronization is a general problem for any certificate system, and how to solve this problem is beyond the scope of this document.

3.4.2 Signature Extensions

We also propose a pair of sub-fields: `CriticalExtensions` and `NonCriticalExtensions` in order to enable new signature features in the future. Each of these two subfields consists of a list of extensions. For any extension in the critical extension list, data consumer must process it if it can be recognized. Data consumer must treat the signature as invalid if a critical extension cannot be recognized. For any extension in the non-critical extension list, data consumer may process it if the extensions can be recognized, and may ignore it if the extensions cannot be recognized.

An example of signature extension is the signature status checking. In some cases, a signature may be re-

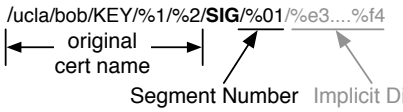

Figure 4: Signature bundle naming convention.

voked before its validity period expires. A data consumer may need to check the status of a signature even if it is still in the validity period. A signature extension for status checking can provide sufficient information for data consumer to collect the signature status, and it may allow signers to enforce signature status checking. We will discuss signature status checking in detail in Section 5.1.

3.4.3 Multiple Signature Extension

Another example of signature extension is the one for multiple signatures, `MultipleSig`. Note that multiple signatures are only needed by certificate. A normal data packet with multiple signatures is equivalent to a normal data packet signed with a key with multiple signatures. Therefore, for the discussion below, we focus on multiple signatures for public keys.

When an association between a public key and a identity can be certified by more than one signers, it is unnecessary to retrieve multiple data packets, each carrying the same public key and signed by a different signer. Instead, it would be more efficient to publish all the signatures in a single data packet, called *signature bundle*. Note that each signature in a key bundle is still independently verifiable, when combined with the `Name`, `MetaInfo`, and `Content` of the original packet.

The value of a multiple signature extension is the name of the signature bundle. The name of the signature bundle is an extension of the original data name as shown in Figure 4. It starts with the name of original data name as a prefix, so that data consumer can always fetch the signature bundle in the same way as the original data packet. After that, the signature bundle name has a special name component SIG indicating that the content is a signature bundle. After SIG component, there is a segment number in case the signature bundle is too big to fit in one data packet. A signature bundle name also has an implicit name component which is the digest of the bundle (or segment if a bundle is divided into multiple segments). When the digest is explicitly expressed (as the last name component of a signature bundle name) in the `MultipleSig` extension, it can fixes all the signatures of the certificate.

When multiple signature extension is used, the original certificate can be self-signed, as the multiple signature extension points to the actual signature set. Whenever the signature set changes, the owner of the public

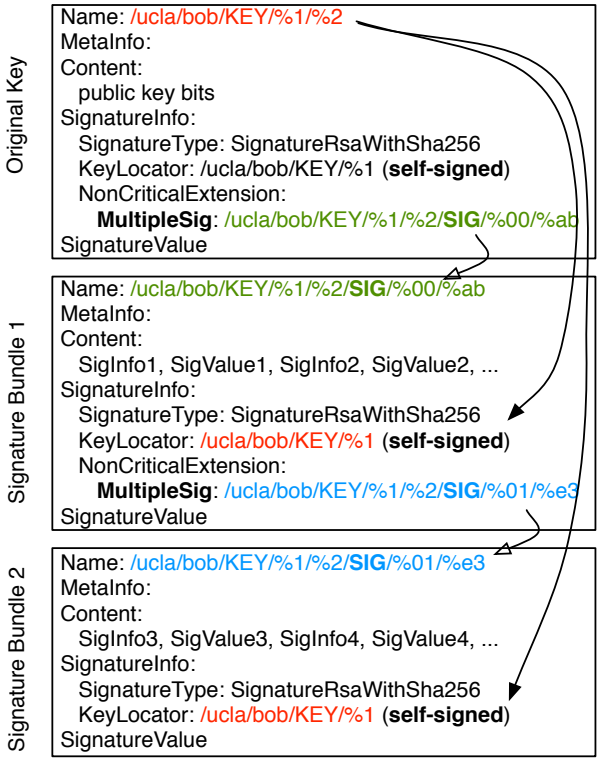

Figure 5: Example of multiple signature extension.

key may generate a new version of signature bundle, and also generate a new version of self-signed certificate whose multiple signature extension points to the new signature bundle.

As we mentioned earlier, it is possible that the signature bundle cannot be fit into a single data packet. In this case, the signature bundle can be divided into several data packets, or segments. Each segment is also self-signed. All the segments in the same signature bundle are chained together by putting the full name of the next segments (i.e., with implicit digest) into its own multiple signature extension, and a certificate user verify the integrity of the signature bundle. Figure 5 shows an example where a signature bundle is split into two segments.

3.5 Capability Certificate

The certificates discussed above associate a public key with an identity name. Some certificates may associate a public key with certain capabilities, e.g., the CA certificate in PKIX [4] associates the capability of certifying other's identity with a public key. For this type of certificates, we assume that the associated capability can be also expressed as a NDN name, i.e., named capability. With this assumption, both identity certificate and capability certificate can be generalized by the NDN certificate.

4. CERTIFICATE PROVISIONING

To facilitate the discussion, we introduce three roles in certificate provisioning: *private key owner*, *certificate host*, and *certificate user*.

A private key owner is the one who possesses a private key, thus it is also the owner of the corresponding public key certificate. Ideally, there should be only one owner for each certificate.

Certificate host is the one who has a copy of the certificate and listens to the prefix of the certificate in order to satisfy the interests for the certificate. Given a certificate, there could be multiple certificate hosts. Some certificate hosts are explicitly designated by the private key owners while all the other certificate hosts are non-designated hosts.

Certificate users are those who use a certificate to verify the signature of the private key owner. A certificate user may retrieve a certificate from one of the certificate hosts. Note that the interest for certificate may be satisfied by the copy cached in a router on the way to the certificate host.

Figure 6 shows an example of how the three roles interact with each other in the certificate provisioning. We will discuss them in detail in the rest of this section.

4.1 Private Key Owner

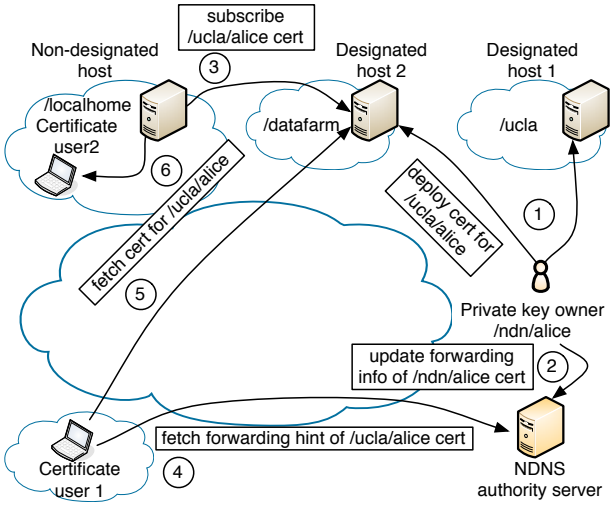

Figure 6: Example of certificate provisioning.

The responsibility of a private key owner is to maintain the certificate. First, the owner of a private key should find at least one designated host for its certificate. Such a designated host could be managed by the private key owner or by some third party who provides data hosting service for the private key owner. For example, in Figure 6, the owner of a private key with the identifier (`/ucla/alice`) has two designated hosts: one in the `/ucla` network which is managed by the private key owner and the other one is provided by a data hosing service `/datafarm`. Having multiple designated certificate hosts may improve the availability of certificate.

Second, the owner of a private key should keep the consistency of the certificates on all its designated hosts. For example, when multiple signatures exist for a public key, the private key owner should manage the corresponding signature bundle as described in Section 3.4.3. Whenever the signature bundle is changed (e.g., new signature is added), the private key owner should assure that each designated host have a new version of the certificate. The owner may explicitly upload the new version of certificates to these designated hosts (as the step 1 in Figure 6, or specify a primary designated host and require all the other designated hosts to synchronize with the primary one. However, the details of these mechanisms are out of the scope of the document.

Moreover, a private key owner should make the forwarding information about its designated hosts available. This can be done with the help of NDNS [3]. For example, in step 2 in Figure 6, a private key owner creates NDNS forwarding hint records pointing to each designated certificate hosts.

4.2 Certificate Host

As we mentioned before, there are two types of certificate hosts. The designated hosts are supposed to be publicly available for general purpose, while the non-designated hosts are usually locally available for specific usage. For example, an end host could be a certificate host for prefix registration on a local hub. In this case, the end host can serve all the certificates that are required to validate a prefix registration command and do not expect receiving any interests for certificate from some certificate users other than the local hub.

Unlike designated certificate hosts, private key owners are not aware of the existence of non-designated certificate hosts. Therefore, it is the non-designated certificate hosts' own responsibility to obtain the latest changes on a certificate. As shown in step 3 in Figure 6, the non-designated certificate host may subscribe the certificate updates from one of the designated hosts.

4.3 Certificate User

Ideally, users of a certificate should not be aware of where and how the certificate is served. A certificate user may try to fetch the certificate within the local network (step 6 in Figure 6), and try to fetch the certificate from designated certificate hosts if the local attempt fails.

However, it would be beneficial if a certificate user has some knowledge about the certificate provisioning. For example, if a certificate user knows for sure that there is no local certificate host, it can skip the attempt to fetch certificate locally and look up the forwarding hint directly in NDNS (step 4 and 5 in Figure 6).

4.4 Application Independent Certificate Hosting

There are several certificate hosting mechanisms that have already been proposed, e.g., NDNS and PIB. However, all these mechanisms explicitly requires certificate users to send an interest for certificate as a query to an particular application. As a result, some application-specific name components (such as "NDNS" and "PIB") are explicitly encoded in the interest name. It might be problematic to encode hosting application specific components into a certificate name, because it implies that either a certificate has to be served using the same application, or a private key owner has to obtained a new certificate when the hosting application is changed.

If certificate name is kept as application-independent, then using existing certificate hosting mechanism to host certificate will inevitably involve packet encapsulation, that is, the requested certificate is encapsulated as a content of a data packet whose name is application-dependent. However, packet encapsulation implies that the same certificate may be cached as multiple packets due to different names, thus it may impair the efficiency of caching.

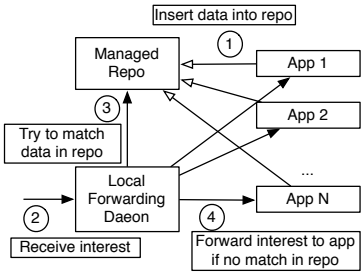

Figure 7: Application-independent publishing model

It would be worthwhile to notice that all the problems above are caused by a concept of publishing certificate explicitly through certain application (either NDNS or PIB), because the local interest forwarding daemon (e.g., NFD [1]) needs to explicitly forward the interests for certificate toward the publishing applications. It is reasonable to forward interests towards an application if the requested data should be produced by the application on demand, however, once data have been produced, serving the data from an end host should not involve the producing application. Certificate is a type of data that are rarely produced on demand, even if it is on demand, it can be done in an obscure way. Therefore, hosting certificate should be application independent.

We propose an application-independent mechanism to host certificate. Note that, such an mechanism should also apply to any other data that is not produced on demand. Figure 7 illustrates the data publishing mechanism. When an application needs to publish some data, the application simply store the data in a managed repository (Step 1). A write access control whose trust model is equivalent to prefix registration can be applied to the managed repository, so that only legitimate application can add/remove data under the its registered prefixes.

When the local forwarding daemon receives an interest for the data (Step 2), the daemon will look for the matched data in the managed repository (Step 3) before forwarding the interest to the applications that can produce the data (Step 4). When certificate publishing is concerned, step 4 can be omitted since certificate is never produced on demand.

5. PUBLIC KEY MANAGEMENT

In this section, we discuss how to manage public key from perspectives of private key owner. A key owner should consider three things: 1) how to make public key certificate available; 2) how to revoke a signature; and 3) how to revoke a public key. The first question has been discussed in Section 4. In this section, we focus

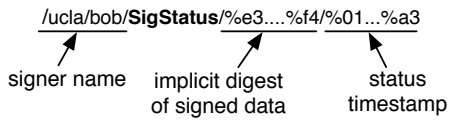

Figure 8: Signature status naming convention.

on the rest two questions.

It would be worthwhile to clarify the difference between signature revocation and key revocation first. Signature revocation usually affects only one signature. Revoking a signature usually implies that the binding between the data name and content is no longer valid even if the signature is still within its validity period. Signature revocation does not necessarily imply the signing key is compromised. However, key revocation is needed when the a key is compromised. Therefore, all the certificate of the compromised key becomes invalid, and the validity of all signatures generated using the compromised key becomes questionable if the time point when the key is compromised is unknown.

5.1 Signature Revocation

There are two signature revocation approaches. The first one is to ask the private key owner to publish a negative statement about the signature, and the second one is to keep publishing a positive statement until the signature is revoked or expires. Both approaches have their own limitations when the statements are temporarily unavailable, (e.g., due to packet loss or operation failure). In the worst case, the absence of a negative statement may cause false authentication, thus privilege may be given out to unauthorized parties. In contrast, the absence of a positive statement may cause denial of service, but do not grant any privilege. It is not easy to permanently keep data unavailable in a distributed system such as NDN, but once a privilege is granted it is really difficult to revert all the effects. Therefore, temporary denial of service is more tolerable than inappropriately granted privilege in NDN public key management.

In NDN, the statement about the validity of a signature is expressed as a data packet, called *signature status*. Since a signature represents an assertion made by the private key owner, it is the owner's decision whether to publish its signature status data. If the private key owner decides to publish its signature status, the owner should assure the same availability of its signature status as its certificate.

The naming convention of signature status data is shown in Figure 8. Its name starts with the name of the key owner (or signer) as its prefix. That implies that signature status data shares the same name prefix as the signer's certificate, so it can be fetched in the same way as the signer's certificate. After the signer name, there is a special name component SigStatus which indicates that the content is the status of one of the signer's signatures. The digest of the data packet to which the signature belongs is appended after SigStatus. This digest can uniquely identify the corresponding signature. The last component in the name of a signature status data is the timestamp of the status. Although ideally a signer should publish the status of its signature on demand for every query, it is still the signer's decision how to publish the signature status. For example, a signer may choose to publish its signature periodically and can decide the period of the signature status publishing.

The content of a signature status data consists of two pieces of information: the signature status ("good" or "revoked") and the reason if the signature is revoked.

As we mentioned in Section 3.4.2, a signature extension `StatusChecking` can be defined to support signature status checking. The presence of `StatusChecking` implies the signature status data is available. By making `StatusChecking` as a critical signature extension, the private key owner can explicitly require certificate users to check the status of its signature. If a status checking extension is marked as non-critical extension, it is determined by the certificate user's policy to check the signature status.

The `StatusChecking` extension also provides sufficient information that can help a certificate user to fetch the signature status. For example, a private key owner can specify the period of the signature status publishing. With this information, a certificate user can determine the timestamp of the requested signature status.

In case that the private key owner may not be able to periodically publish the signature status data, the key owner may delegate the task of publishing signature status to a trusted third party by specifying the name of the third party in the signature status extension. As a result, the signature status data will be published under a name with the third party name prepended.

5.2 Key Revocation

If a key is compromised, it is ineffective to revoke each signature generated using the key. A more straightforward solution is to ask the private key owner to suicide the key by making a self-signed "suicide" certificate. Such a suicide certificate is almost the same as a self-signed certificate, except that the certificate name has an special name component REVOKED appended.

Note that such a suicide certificate is a negative statement, therefore it is important to make the revocation certificate publicly available for certificate users. The private key owner, of course, can publish the suicide certificate in the same way as normal certificate. However, it requires certificate users to explicitly fetch the sui-

cide certificate on demand and also requires key owner to keep generating a NACK for the suicide certificate before the suicide is committed.

Another solution is to build a global suicide directory for keys which is synchronized by all the certificate users, so that certificate users already know which keys suicide before verifying the signature. Since the design details of the suicide directory is beyond the scope of this document, we briefly describe the idea of the suicide directory here.

A suicide directory can be implemented as a tamper-evident log. Suicide certificates are appended to the log one-by-one. The appending order is managed by only a limited number of entities who receive and verify the suicide report from the owner of compromised private keys.

Since the log is tamper-evident, if a managing entity tries to modify the log, it can be detected immediately, thus forcing each managing entity to behave correctly. Moreover, unless all the managing entities collude together, it is really difficult to prevent a private key owner to append a suicide certificate to the log.

6. DATA PROVISIONING

According to the discussion above, a data consumer (or a certificate user) may need to fetch a lot of public key related data to validate a data packet. These data may include certificates that can be used to derive a chain of trust from a trust anchor to the data, the signature status of data or certificates on the chain, and the directory of suicide keys. We generally call these public key related data as *authentication data*.

Although ideally a data host only needs to serve data, it would be more useful to ask data hosts to serve all the authentication data as well. These authentication data are usually spread at different places in the network, it is not guaranteed that a data consumer will be able to fetch all the required data. Failing to fetch one of them may prevent the data consumer from validating the data. If data hosts can pre-collect all the information, such single point failures can be effectively avoided. Moreover, it can also save the efforts of data consumers to individually fetch these data.

Furthermore, data host can publish all the public key related data as a bundle, called *key bundle*. As all these data can be combined in one or a few packets, several round trip time for data consumer can be avoided in data fetching, thus reducing the latency of data validation.

The naming convention of key bundle is shown in Figure 9. It starts with the original data name in order to guarantee that the key bundle can be retrieved in the same way as the data. A special name component PROOF is appended after the original data name,

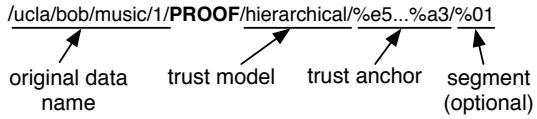

Figure 9: Key bundle naming convention.

indicating the content type. After PROOF, there are three more components. The first two components indicate the trust model and trust anchor of the key bundle. That is, if a data consumer is using the same trust model and trust anchor, it should be able to validate the original data with this key bundle. The last component is segment number. It is only needed when all the authentication data cannot be fit into one data packet.

Since the content of a key bundle is authentication data which are all verifiable, the key bundle does not have to be signed. The authentication data are arranged in a specific order: the authentication data for a key should be placed before those keys downstream the signing chain, so that each intermediate key can be immediately validated with the authentication data that have been validated.

7. REFERENCES

[1] NFD - Named Data Networking Forwarding Daemon. http://named-data.net/doc/NFD/.

[2] Public-key Info Base service. http://redmine.named-data.net/projects/ndn-cxx/wiki/PublicKey_Info_Base.

[3] A. Afanasyev. *Addressing Operational Challenges in Named Data Networking Through NDNS Distributed Database*. PhD thesis, UCLA, 2013.

[4] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. RFC 5280: Internet x. 509 public key infrastructure certificate and certificate revocation list (crl) profile, May 2008.

[5] J. Liang, J. Jiang, H. Duan, K. Li, T. Wan, and J. Wu. When https meets cdn: A case of authentication in delegated service. In *Security and Privacy (SP), 2014 IEEE Symposium on*, 2014.