University of California
Los Angeles

Usable Security For Named Data Networking

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Yingdi Yu

2016

Abstract of the Dissertation

Usable Security For Named Data Networking

by

Yingdi Yu

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2016

Professor Lixia Zhang, Chair

Named Data Networking (NDN) is a proposed Internet architecture, which changes the network communication model from "speaking to a host" to "retrieving data from network". Such data-centric communication model requires a data-centric security model, which secures data directly rather than authenticating the host where data is retrieved from and securing the channel through which data is delivered, so that data can be safely distributed into arbitrary untrusted storage and retrieved over untrusted network.

The data-centric security model consists of two parts: data-centric authenticity and data-centric confidentiality. NDN achieves data-centric authenticity by mandating per packet signature, and data-centric confidentiality by data encryption. While the idea is straightforward, we observed that usability of data-centric security of NDN prevents developers from enabling security in their applications. This dissertation presents a security framework to automate data-centric security of NDN and reduce the enabling overhead. To achieve that, we designed NDN certificate system to facilitate public key distribution in NDN; we designed *Trust Schema*, a name-based policy language to specify trust model, in order to automate fine-grained data authentication; we designed a timestamp service *DeLorean* to address the authenticity problem of archival data; and we also designed an access control protocol *Name-based Access Control* to automate data-centric

confidentiality at fine granularities.

The dissertation of Yingdi Yu is approved.

Daniel Frank Massey

Songwu Lu

Mario Gerla

Lixia Zhang, Committee Chair

University of California, Los Angeles

2016

Table of Contents

List of Figures

Acknowledgments

I would like to sincerely thank my academic advisor Prof. Lixia Zhang for providing invaluable support throughout my Ph.D. program. I also want to thank my colleagues from UCLAs Internet Research Laboratory Alexander Afanasyev, Wentao Shang, and Zhenkai Zhu for their support and valuable discussions, without which this work would not have existed. I am also enormously grateful to our collaborators Prof. Van Jacobson, Prof. Alex. J. Halderman (University of Michigan), Prof. David Clark (MIT), Prof. Beichuan Zhang (University of Arizona) and many other members of the NDN team for insightful discussions that built up my understanding of the network security design in general and Named Data Network design in particular. At last, I would like to gratefully thank my wife Fei Lu for supporting me and encouraging me in the past four years.

Vita

2007	B.E. (Electrical Engineering), Shanghai Jiao Tong University, Shanghai, China.
2010	M.E. (Electrical Engineering), Shanghai Jiao Tong University, Shanghai, China.
2011–2012	Teaching Assistant, Computer Science Department, UCLA.
2012–2016	Graduate Research Assistant, Computer Science Department, UCLA.
2012, 2013	Research Intern, VeriSign, Reston, Virginia.

Publications

Y. Yingdi, A. Afanasyev, D. Clark, kc claffy, V. Jacobson, and L. Zhang, "Schematizing Trust in Named Data Networking," *Proc. of ACM ICN*, 2015.

A. Afanasyev, Z. Zhu, Y. Yu, L. Wang, and L. Zhang, "The Story of ChronoShare, or How NDN Brought Distributed Secure File Sharing Back," in *Proc. of IEEE MASS*, 2015.

Y. Yu, D. Wessels, M. Larson, and L. Zhang, "Check-R: A New Method of Measuring DNSSEC Validating Resolvers," in *Proc. of IEEE TMA Workshop*, 2013.

Y. Yu, D. Wessels, M. Larson, and L. Zhang, "Authoritative Name Server Selection of DNS Caching Resolvers," in *ACM Computer Communication Reviews*, 2012.

Y. Yu, A. Afanasyev, and L. Zhang, "NDN DeLorean: An Authentication System for Data Archives in Named Data Networking," *Technical Report NDN-0040*, 2016.

W. Shang, Y. Yu, R. Droms, and L. Zhang, "Challenges in IoT Networking via TCP/IP Architecture," *Technical Report NDN-0038*, 2016.

V. Lehman, A. Hoque, Y. Yu, L. Wang, B. Zhang, and L. Zhang, "A Secure Link State Routing Protocol for NDN", *Technical Report NDN-0037*, 2016.

W. Shang, Y. Yu, T. Liang, B. Zhang, and L. Zhang "NDN-ACE: Access Control for Constrained Environments over Named Data Networking", NDN, *Technical Report NDN-0036*, 2015

Y. Yu, A. Afanasyev, and L. Zhang "Name-Based Access Control", *Technical Report NDN-0034*, 2015

Y. Yu "Public Key Management in Named Data Networking", *Technical Report NDN-0029*, 2015

Y. Yu, A. Afanasyev, Z. Zhu, and L. Zhang "An Endorsement-based Key Management System for Decentralized NDN Chat Application", *Technical Report NDN-0023*, 2014

Y. Yu, J. Cai, E. Osterweil, and L. Zhang "Measuring the Placement of DNS Servers in Top-Level-Domain" *Technical Report*, May. 2011

CHAPTER 1

Introduction

The Internet has increasingly involved in our daily life by powering a wide range of applications including e-mail, web, online video, e-commerce, etc. It is critical to secure the communication over the Internet.

The original Interent architecture (also known as TCP/IP) provides a *point-to-point* communication primitive, i.e., one host sends packets to another host identified by IP address. As a result, a *channel-based* security model was proposed to secure the communication. More specifically, one end of communication first authenticates the other end and establishes an encrypted channel to receive subsequent communication data.

Today's application communication model however has shifted from "speaking to a host" to "retrieving data from wherever available". For example, Peer-to-Peer Networking (P2P) allows users to download data from any available peer. Content Distribution Networking (CDN) pushes data from its original host to edge servers that are closer to the end users. The mismatch between such *data-centric* communication model and the *channel-based* security model has caused several inconvenience and potential risks in network and security operation, e.g., CDN customers have to deploy their confidential private keys into all the CDN edge servers [LJD14].

Securing data directly seems to be a more appropriate security model for the data-centric communication model, since data may be retrieved from arbitrary hosts over arbitrary channels. Specifically speaking, a data recipient authenticates

data rather than the host from which data is received. Moreover, the original producer encrypts data so that only authorized consumers can read the data. We call such a security model *data-centric* security model.

Named Data Networking (NDN) [JST09, SJ09, ZAB14] is a proposed Internet architecture, which provides a general data-centric communication primitive. Communication entities either produce or consume named data. NDN architecture also mandates per packet digital signature as the first step of building data-centric security into the network layer. However, it is still non-trivial for developers to achieve complete data-centric security to protect the network communication. A common observation is that application developers usually disable security as their first step of implementation. We attribute the usability issue to three facts: lack of convenient data authentication solution, lack of adequate solution to the mismatch of data and signature lifetime, and lack of default confidentiality support. In this dissertation, we investigate the three problems above respectively and proposed corresponding solutions.

1.1 Data-Centric Authentication in NDN

NDN requires a data producer to attach a digital signature to the data at the time of production, so that consumers with the producer's public key can directly authenticate the data. Unfortunately, consumers on current Internet may not be able to pre-acquire the public key of each potential producer. They have to rely on some trust management mechanism to establish trust on producer public key dynamically. NDN however does not provide a systematic way for developers to specify the trust model of their application, thus leaving developers to hardcode data authentication logic in their applications.

We found that by treating producer public key as normal data, we can unify the trust management on public key as normal NDN data authentication. More-

over naming keys under the same naming system of data, we can express trust model in terms of the relationship between data name and key name. These two observations inspire us to design a name-based policy language *trust schema* and a data authentication library that can correctly interpret the trust model specified in a trust schema and automatically authenticate data according to the trusted model. As a result, developers can focus on using trust schema to describe trust model at a high level, without worrying about low level implementation details.

Additionally, trust schema allows security experts to define a set of templates of commonly reusable trust schema, which we call *security design patterns*, for popular network applications. Developers can simply choose an appropriate template from the pre-defined set and specialize it for their own applications.

1.2 Data Signature Lifetime Mismatch

The data-centric communication model waives the requirement that data producers and consumers of the same communication have to be online at the same time. This feature however exposes the problem of mismatch between long-lived data and short-lived digital signature. Current practice is to ask data producer to periodically re-sign its data, which introduces a heavy burden on data producers. In some cases, data may even outlive its producer, renderring periodical re-signning a non-viable solution.

We attributed this problem to current data validation model which requires a digital signature to be valid at the time of data consumption and solve this problem by proposing a *post-factum* validation model which allows a consumer to verify the validity of a digital signature at the time of data production. To achieve this goal, we designed *SigLogger* which can help a consumer to recover the production-time security context through three logging systems: a verifiable timestamp log, an append-only revocation log, and a publicly auditable context

log. The timestamp log provides consumers the existence proof of keys and trust schema. The revocation log help consumers to promptly acquire revocation information and exclude revoked keys and trust schema from the validation process. The context log allows consumers to faithfully bootstrap trust from a trust schema used during the production time.

The post-factum validation model effectively decouples data lifetime from signature lifetime. As a result, it can encouraging the use of short-live keys without worrying about the overhead of data re-signing. It can also significantly reduce the chance of revocation.

1.3 Data-Centric Confidentiality in NDN

NDN architecture per se does not specify any data confidentiality mechanism. It leaves developers to implement confidentiality support in their own applications. Although data encryption is the major approach to support confidentiality, a developer still has to address two practical issues: 1) how to properly encrypt data to achieve efficient data delivery and 2) distribute decryption key to authorized consumers.

To free developers from implementing their own confidentiality solution, we provide an encryption layer for NDN by designing a data-centric confidentiality protocol, called *name-based access control* (NAC). By leveraging NDN's hierarchical naming system, NAC allows data owner to control data encryption at fine granularities. NAC also leverages NDN's name-based data retrieval to distribute encryption instruction and decryption keys in an scalable way. We implemented NAC as a middle layer library between the upper layer applications and lower layer network, thus hiding all the implementation details of data encryption from application and preventing any intermediate network devices from seeing the plain text data.

The design of NAC emphasizes on distributed data production and dynamically changing access control. It allows multiple data producers to produce and encrypt data at the same time, and also allows the data owner to easily grant and revoke multiple consumers' access to data at real time.

1.4 Contributions of this work

Contributions of this dissertation can be summarized as follows:

- Design and implementation of NDN certificate and its management system, the fundamental building block of any NDN security solution (Chapter 3).
- Design of trust schema, the name-based policy language; and design and prototype of a security library that can automatically perform data authentication according to a trust model described in trust schema (Chapter 4).
- Introduce the post-factum validation model for long-lived data authenticity in NDN; Design and prototype of SigLogger which enables post-factum validation over NDN (Chapter 5).
- Design and prototype of Name-based Access Control (NAC), the first data-centric confidentiality solution in NDN (Chapter 6).

CHAPTER 2

Background

This chapter briefly introduces the security model of the Internet and Named Data Networking (NDN) architecture, which comprise the primary subject of this thesis. The following sections present a short description of the relevant parts of the architecture. A more detailed description can be found in specialized literature, such as [ZAB14, JST09, SJ09] for NDN and [CSF08, DR08] for the Internet security.

2.1 Internet Security Model

The original Internet design aimed at providing point-to-point communication. As a result, the corresponding security model is designed to protect a point-to-point communication channel. People had designed and implemented a set of security architecture to establish secure channels at different levels, from host-to-host channel (IPSec [KS05]) to application-to-application channel (TLS/SSL [FKK11, DR08], SSH [YL06]).

All these security architectures secure the communication through two steps. The first step is to authenticate communication end point. The second step is to establish an encrypted channel between two end points. Once the channel is established, both ends can securely exchange communication data through the channel.

2.1.1 Endpoint-Oriented Authentication

End point authentication is usually done through pre-shared secret keys or public/private key. More specifically, one end must prove to the other end its ownership of a particular pre-shared key or the ownership of a private key corresponding to a particular public key.

When public keys are used in authentication, a public key is usually associated to a subject, e.g., a domain name, IP address, user id, etc. The association between a public key and a subject therefore serves as the foundation of end point authentication and must be secured as well. Although it might be possible to store the key-subject association in each end host for a small scale system, the enormous amount of hosts over the Internet demand a more efficient system to secure the key-subject associations, which is usually called *Public Key Infrastructure* (PKI).

Users of a PKI system start with one or more pre-trusted keys and follow the PKI's trust model to gradually derive trust on each individual key-subject association. There are three major types of PKI running over current Internet: Certificate Authority (CA) [CSF08], DNSSEC [AAL05], and Web-of-Trust [CDF07]. Each system represents a different style of trust management. Web-of-Trust emulates the real-world identity-based trust management, within which people make trust decision based on the identity of association provider. DNSSEC represents the capability-based trust management, within which DNS validators check whether a key-subject association is made by someone who manages the corresponding DNS domain. The CA system defines two roles: certificate providers and certificate owners. Only certificate providers make key-subject association. All the users of the system trust a CA for any association it has ever made. A provider can also designate other providers.

Although DNSSEC-based PKI is getting more popular in recent years [HS12], the CA-based PKI is still the dominating one in the Internet. However, due to

lack of privilege regulation, a number of severe security accidents caused by CA happen every years [Bri11, Sec12, Wil15].

2.1.2 Channel-Based Confidentiality

After the end point authentication, two end points establish an encrypted channel. More specifically, two ends first negotiate a temporary session key. The sending end uses the session key to encrypt data sent through the channel. Only the receiving end can decrypt the data in cipher text. In other words, none of any intermediate network devices can see the data in plain text.

The session key is as ephemeral as its belonging communication channel. Exchanging the same data over a new channel between the same pair of end points implies re-negotiation of the session key and re-encryption of the data using the new session key.

2.2 Named Data Networking

Named Data Networking (NDN) is a proposed Internet architecture which provides data-centric communication primitives. NDN changes the Internet's communication model from *delivering packets to an end host* to *retrieving content for a given name*. Entities participating in communication either *produce* or *consume* named data. Data names in NDN are hierarchically structured. For example, the name of the first segment of an HTML page for the "`www.cs.ucla.edu`" website would look like "`/edu/ucla/cs/www/index.html/%00`".

NDN introduces two types of network level packets *Interest* and *Data* (Figure 2.1) to support the data-centric communication model. A consumer requests data by expressing an interest packet, which carries the name or name prefix of the requested data. NDN routers use the name to forward interest packets toward

the origin of requested data.[1] A data packet whose name matches the name in the interest is returned to the consumer by following the reverse path of the interest packet.

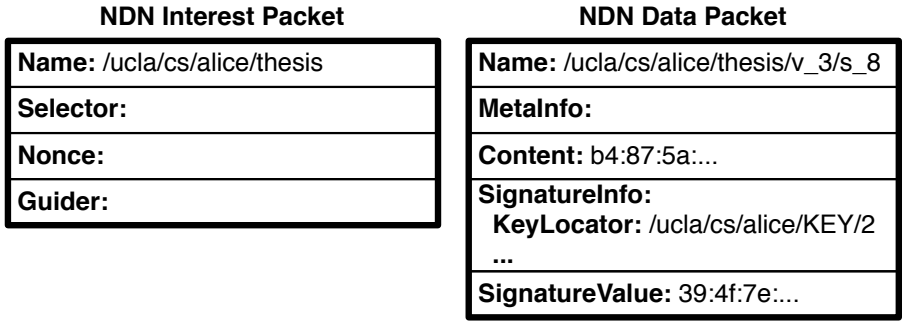

Figure 2.1: Packet format of NDN Interest and Data

Each NDN router maintains three major data structures:

- A *Forwarding Interest Base* (FIB) maps name prefixes to one or multiple physical network interfaces, specifying directions to which an interest can be forwarded.[2]

- A *Pending Interest Table* (PIT) holds all "not-yet-satisfied" interests that have been sent upstream toward potential data origins. Each PIT entry contains an interest packet and one or multiple incoming physical network interfaces, which indicate multiple downstream consumers. By maintaining this information, NDN routers can achieve the reverse path data forwarding and multicasting without requiring consumer to acquire a network address.

- A *Content Store* (CS) temporarily buffers data packets that pass through this router, allowing prompt response to different consumers requesting the same data.

[1] Ideally, any name is reachable in NDN. In practice, only a few namespaces are globally routable in NDN. NDN relies on Link Object [AYW15] and NDNS [Afa13] to scale up the routing.

[2] In this dissertation, we assume that any name is routable in NDN with existing routing scalability solution [AYW15], though the size limitation of FIB implies only a part of name prefix is globally routable.

The data-centric communication model facilitates data distribution by allowing data to be picked at any available place, including the original of data and in-network storage (i.e., content store). The data-centric communication model eliminates the requirement that both consumer and producer have to be online to achieve the communication. A producer can pre-create data packets and place the packets into a third party storage, so that consumers can still retrieve the data packets when the producer goes offline.

2.2.1 Data-Centric Security of NDN

NDN fosters a data-centric security model at the network layer by mandating a digital signature on each data packet [SJ09]. In other word, a producer attaches to a data packet a digital signature that binds data to its name at the time of production. A consumer can directly authenticate the origin of a data packet by verifying the signature using the producer's public key regardless where the data is retrieved from.

In case that a consumer does not have a producer's public key, a data producer also puts the name of its signing key into a specific field in the data packet, called `KeyLocator` (Figure 2.1). Consumers can follow this field to retrieve the public key in the same way as retrieving normal data packets.

The data packet carrying the public key is effectively a public key certificate. A consumer may recursively retrieve multiple keys until reaching a pre-trusted key, which we call *trust anchor*. Starting from a trust anchor, a consumer can verify the signature of each retrieved key and derive trust from the trust anchor to the target data. The list of keys between the target data and trust anchor is also called *trust chain*. Figure 2.2 shows an example of trust chain.

A consumer also needs a list of rules in order to correctly derive trust along the trust chain. The trust anchors and derivation rules constitute a trust model,

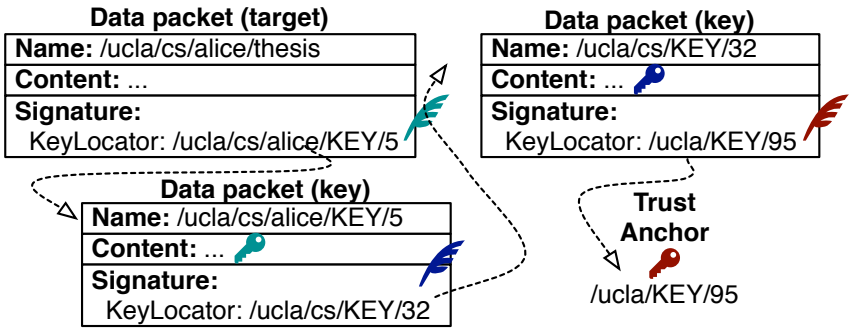

Figure 2.2: An example of trust chain consists of target data, intermediate keys, and trust anchor.

which we also call *security context*. Security context is application-specific. For example, the security context of UCLA thesis filing system cannot be the same one of a smart home application. Producers and consumers of the same NDN application must share the same security context in order to authenticate data correctly. However, the NDN architecture does not provide a concrete mechanism to specify a trust model and direct data authentication in NDN applications.

Additionally, NDN defined the data-centric authenticity into its architecture. However this makes only one part of the data-centric security model. The other part of the model, data-centric confidentiality must also be provided. In the rest of this dissertation, we will demonstrate how to provide those missing building blocks of the data-centric security in NDN.

CHAPTER 3

NDN Certificate

Certificate, which securely binds a key to a subject, is the most fundamental building block of network security. It serves as the foundation of trust management and authentication, upon which confidentiality is built. In Section 2.2.1, we briefly mentioned that NDN certificate is as simple as a data packet carrying a public key as its content. In this section, we first discuss the rationale of designing NDN specific certificate format (Section 3.1), then expand on the design detail of NDN certificate (Section 3.2), and how to properly manage certificate in NDN (Section 3.3).

3.1 Why NDN certificate?

A frequently asked question is: *why we need an NDN specific certificate format? rather than reusing the existing certificate format, such as X.509* [CSF08]? One reason of reusing X.509 certificates is that people have already obtained these certificates of which some are expensive to get and that the infrastructure of certificate management has already existed for many years. However, directly applying X.509 certificates in NDN may introduce several complexities.

First of all, NDN's data-oriented security model requires explicit trust relationship between data name and key name. Although X.509 has its own naming system, it is still different from NDN's strict hierarchical naming system. Additional name converting mechanism must be introduced in order to bridge the

gap between the two naming systems. Since the name in an X.509 certificate may not have a strict hierarchical structure (in most cases, the name is only a string, e.g., "`Google Internet Authority G2`"), the converting mechanism can be arbitrarily complicated.

Second, X.509 is not only a certificate format, but also involves a set of auxiliary protocols (e.g., CRL [CSF08], OCSP [SMA13]), which are built over IP. Reusing X.509 implies the additional dependency on IP network. On the other side, the overhead of porting these auxiliary protocols onto NDN may not be less than developing a new set of native auxiliary protocols over NDN.

Moreover, X.509 certificates are not directly retrievable but are delivered over an established point-to-point channel. This requires additional adaptation to convert an X.509 certificate to an NDN data packet.

With all the reasons above, we argue that it is still necessary to design an NDN specific certificate format, which 1) is natively compatible with NDN naming system, 2) has no dependency on IP network, and 3) can be retrieved as individual piece of data from the network.

3.2 Certificate Format

NDN data packets share several common properties with public key certificates. Figure 3.1 shows a comparison between an NDN data packet carrying a public key and an X.509 certificate. A public key certificate seals the binding between public key bits and a subject (or an identifier) through a digital signature generated by the certificate issuer. Similarly, an NDN data packet seals the binding between data and its name through a digital signature generated by the data producer. In fact, one can view NDN data packet as a general format of public key certificate, if the issuer can express a subject using NDN name and use the content to carry the public key bits.

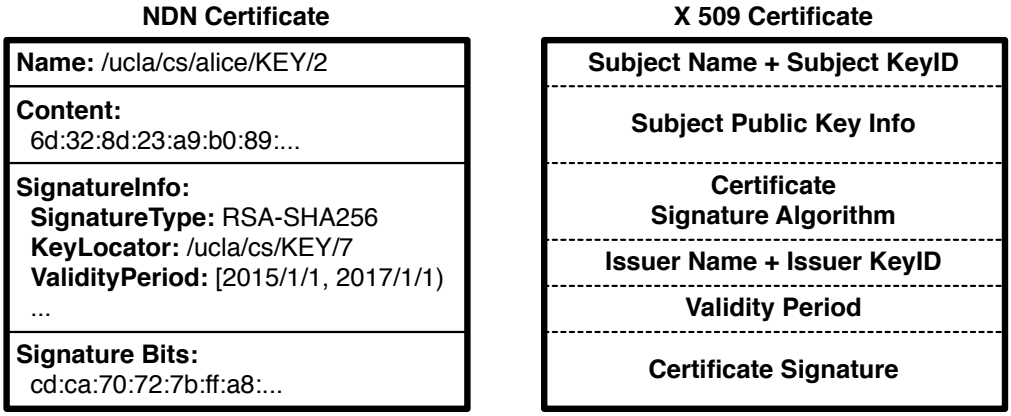

Figure 3.1: NDN certificate as specialized data packet

The most obvious advantage of using a data packet to represent a certificate is that a consumer can retrieve and validate a public key certificate as a normal data packet, thus allowing a common validation procedure for both data and keys. It also forces naming public key in the same hierarchical naming system as data, which enables us to explicitly express a trust model as will be discussed in Chapter 4.

However, the basic NDN data packet format is too simple to become a certificate format. For example, one cannot simply use the subject in traditional certificate as certificate name, because multiple certificates may correspond to the same subject (due to different key bits, issuers or validity periods), but an NDN name must uniquely identify a data packet. Moreover, a certificate issuer may want to restrict the validity of a certificate within a certain period, or an issuer may want to specify a mechanism through which a consumer can actively check the status of a certificate. Original data packet format, however, does not include this additional information. In the rest of this section, we will explain how to extend current data packet format to accommodate the requirements of certificate.

3.2.1 Naming convention

We express the subject of a certificate in terms of NDN's hierarchical name. For example, a subject for UCLA can be expressed as "`/ucla`" and a subject for UCLA Computer Science Department can be expressed as "`/ucla/cs`". The hierarchical name space provides the context of different subjects. For example, two subjects "`/ucla/cs`" and "`/ucla/ee`" are all under the same context of "`/ucla`". The structured name also allows the certificate issuers to place subject attributes (e.g., "`ucla`", "`cs`") in an organized way, thus making it easier for others to derive the trust relationship between different subjects. Note that subject of NDN certificate may not only refers to an identity, but also to a capability or a role as we will expand in Chapter 4.

We mentioned earlier that subject cannot be directly used as NDN certificate name, as each NDN data packet must have a unique name, but multiple certificates may correspond to the same subject. There are three facts that will cause different certificates for the same subject: 1) the public key needs to be replaced over the time; 2) the attributes of a certificate (e.g., validity period) may change over the time; and 3) the signing key of a certificate may be different (e.g., key rollover, different issuers). Note that the last fact may happen when more than one issuers assert the same subject-key association or the same issuer replaces its signing key periodically. In order to generate a unique certificate name, we defined the naming convention for NDN certificates by appending a sequence of name components after the subject name, as shown in Figure 3.2.

Figure 3.2: NDN certificate naming convention

These name components include:

- **KeyID**: a name component that uniquely identifies a key that is associated with the subject. The concatenation of subject name and KeyID is also called *key name*. Since a certificate is considered as a data under the subject name space, it is the key owner's responsibility to assure the uniqueness of KeyID.[1] In other word, when a key owner requests a certificate from an issuer, the key owner should pre-specify the KeyID. A special name component "`KEY`" is appended after KeyID, indicating that the data packet is a certificate.

- **IssuerID**: a name component that distinguishes certificates issued by different issuers. The IssuerID component is appended after the "`Key`" component. Although an issuer can use arbitrary IssuerID, it is still beneficial to define a convention for IssuerID, so that one can deterministically construct an interest for a certificate issued by a particular issuer. For this purpose, we recommend using the hash of issuer's subject name as the IssuerID.

- **Version**: a name component that distinguishes certificates of the same subject issued by the same issuer. For each subject-key association, an issuer maintains an increasing version number. A new version of certificate is created whenever any certificate attribute changes or the issuer roll over its signing key. Note that the change of public key actually leads to a new subject-key binding, thus an issuer may start from version 0 for the new binding.

3.2.2 MetaInfo

The metainfo of NDN data packet describes additional information about the packet, including `ContentType`, `FinalBlockID`, and `FreshnessPeriod`. As the content of a certificate always public key bits, the `ContentType` of a certificate is

[1] One convenient option is to use the crypto hash of public key (e.g., SHA-256 digest) as KeyID.

defined to be: `KEY`. The `FinalBlockID` is used when the content of a packet cannot fit into a single data packet, thus there is no need to specialize it for certificate.

The `FreshnessPeriod` indicates how long a node should wait after the arrival of this data before marking it as stale. It is only used for data retrieval. For example, an interest packet with `MustBeFresh` flag set can only pick a data packet that is not marked as stale. Since a data packet without `FreshnessPeriod` will never be marked as stale, a certificate must have a specified `FreshnessPeriod`. However, `FreshnessPeriod` is unrelated to the validity of a certificate, which is determined by the `ValidityPeriod` as described later (Section 3.2.4).

3.2.3 Content

The content of a certificate contains the public key bits only, which is encoded in X.509 public key format (not X.509 certificate format). Since most crypto libraries can directly load or save a public key in X.509 format, this format can simplify any crypto-related implementation.

3.2.4 SignatureInfo

All the other attributes of a certificate are the attributes of issuer's assertion, thus should be placed in the `SignatureInfo` field. The `SignatureInfo` of a data packet has originally contained two subfields: `SignatureType` and `KeyLocator`, which specify the signature type (e.g., RSA with SHA-256) and the signer's key name (e.g., "`/ucla/cs/id=85/KEY/v=1`").

Apparently, these fields are not sufficient for a certificate. In order to make the certificate format extensible enough to accommodate more attributes. We extend `SignatureInfo` into a list of TLV blocks[2], with each block referring to an

[2]TLV is short for "Type-Length-Value". A TLV block is an encoding unit of NDN packet format, which usually refers to an individual field in a packet.

signature attribute. We also define two categories of attributes: *critical attributes* and *non-critical attributes*. If a certificate carries a critical attribute that the consumer does not recognize, the consumer must treat the certificate as invalid, while for non-critical attribute it is consumer's own decision to determine the validity of the certificate. We have introduced two attributes to enable basic certificate functionalities:

- **ValiditPeriod**: a critical attribute that restricts the lifetime of a signature, which also effectively restricts the lifetime of a certificate. A short-lived certificate can effectively mitigate the key revocation problem. Even if a key is compromised, the attack window is restricted by the limited lifetime. For this purpose, we introduce a new field `ValidityPeriod` which contains the starting timestamp (`NotBefore`) and the ending timestamps (`NotAfter`) of the certificate lifetime. The timestamps are UTC timestamp in ISO 8601 compact format (`yyyymmddTHHMMSS`, e.g., `20020131T235959`).

- **AdditionalDescription**: a non-critical attribute that provides additional information about the certificate. The information is expressed as a set of key-value pairs. Both key and value are UTF-8 strings, e.g., ("Organization", "UCLA"). The issuer of a certificate can specify arbitrary key-value pair to provide additional description about the certificate.

3.2.5 Multiple Signature

NDN data packet format allows only one signature per packet. Some trust models however may require multiple signatures for the same key-name binding. Although one can always represent multiple signatures using multiple certificates, it may not be trivial for a consumer to retrieve all these certificates. For example, in order to retrieve a certificate issued by a particular issuer, a consumer must specify the IssuerID component in the interest packet. In case that a consumer has no knowl-

edge about all the issuers, the consumer may have to enumerate the key namespace (e.g., "`/ucla/cs/id=85/KEY`") for certificates made by different issuers. Such certificate enumeration inevitably introduces large runtime complexity and does not guarantee that all the certificates are enumerated.

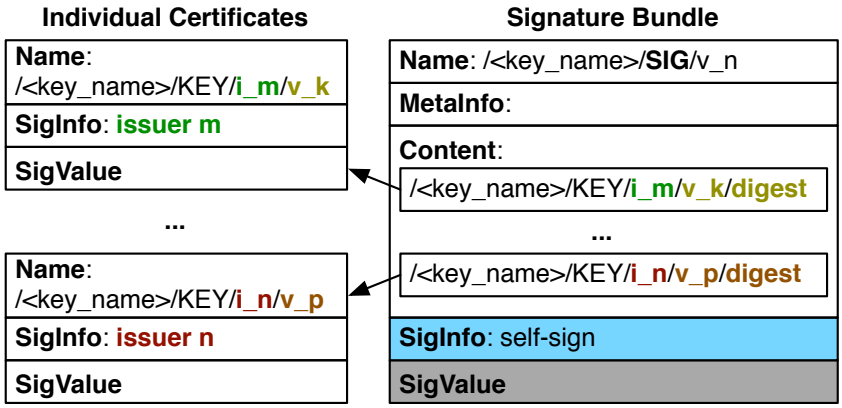

Figure 3.3: Signature bundle for multiple certificates

In order to efficiently retrieve multiple certificates for the same key, we introduce *signature bundle* as shown in Figure 3.3. A signature bundle is a data packet published by the owner of a key. It contains the full name[3] of all the latest version of certificates made by each issuer. A consumer with a signature bundle can follow the contained certificate names to retrieve the required certificates. Note that with all the certificate names available, a consumer can determine the best strategy for certificate retrieval.

Since publishing key bundle can significantly increase the chance of a key being authenticated by others, the key owner should have strong motivation of collecting the latest version of certificates made by each issuer, updating the signature bundle, and making it available (or publishing it in the network).

Signature bundle is the first packet to retrieve when a consumer starts to validate a key with multiple signatures. Therefore, a consumer must be able to

[3]The full name of a data packet is the concatenation of the data name and the data's implicit digest, which uniquely identify a particular data packet.

construct the signature bundle name directly based on the key name, without requiring any other information. For this purpose, we defined the naming convention as shown in Figure 3.4.

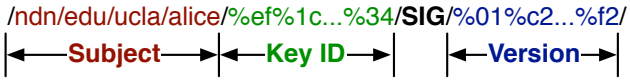

Figure 3.4: Signature bundle naming convention

A signature bundle name starts with the key name. In order to distinguish signature bundle from certificates, we append another special name component `SIG` to the key name. Since a signature bundle may be updated from time to time, a version number is appended as the last name component.

With this naming convention, a consumer can simply construct an interest packet with the name as the concatenation of key name and "`SIG`". The consumer can send the interest out with the assumption that the key owner will always keep the latest version of signature bundle available, and that the `FreshnessPeriod` of old version signature bundles are adequately specified to ensure an interest will always pick up the latest version of signature bundle.

Note that a key owner does not have to be online all the time for signature bundle publishing, it can self-sign the signature bundle (i.e., sign the signature bundle using the same key for which the bundle is made) and leaves the signature bundle data packets in a third party storage. The self-signed signature is sufficient enough for a consumer to authenticate the signature bundle data packet. Note that this signing rule does not provide any assurance about the certificates pointed by the names inside the bundle. A consumer still needs to authenticate each individual certificate.

3.3 Certificate Management

Since certificates are critical to NDN security, they must be properly managed. More specifically, certificate issuer must issue certificates to eligible key owner. A certificate issuer must also revoke a certificate properly once the key owner is no longer eligible. The key owner (also certificate owner) must make its certificates highly available, so that consumers (certificate users) can always retrieve them to perform data authentication. Figure 3.5 shows the interactions among certificate issuers, key owners, and certificate users.

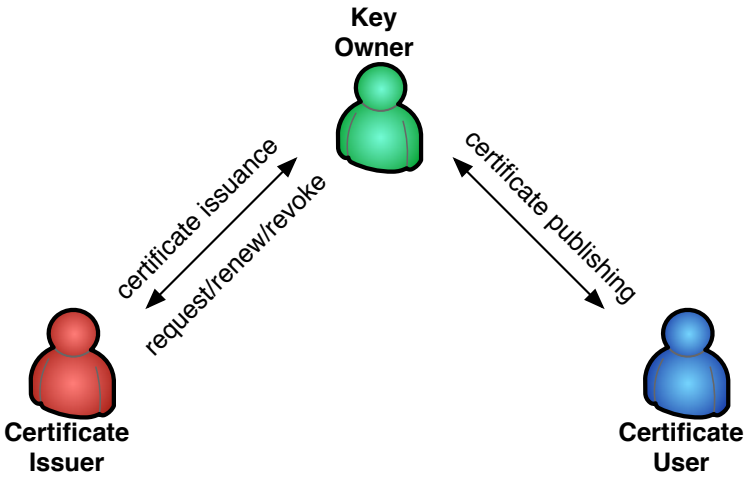

Figure 3.5: Operation model of certificate management.

3.3.1 Automated Certificate Issuance

Certificate issuance is the first step of enabling NDN security. The process of acquiring a certificate must be as convenient as possible in order to encourage enabling security in NDN application. To this purpose, we designed the NDN certificate issuing system by borrowing the concept of automated certificate management (ACME) framework [BHK15]. The issuing system, together with auto-signing feature of trust schema (Section 4.3.2), can automatically initialize the signing key configuration of NDN application. Next, we explain the working

mechanism of NDN certificate issuing system (Figure 3.6).

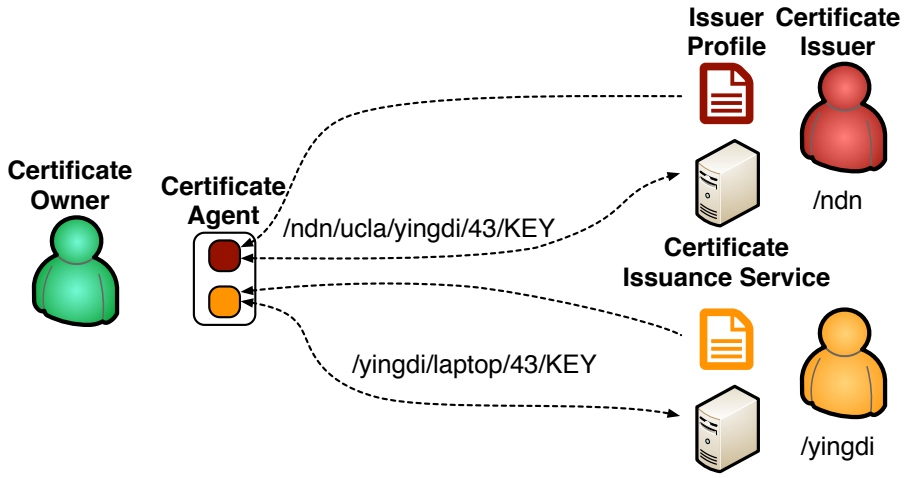

Figure 3.6: NDN certificate issuance system.

In the NDN certificate issuing system, each certificate issuer manages a namespace and runs a certificate issuance service for the namespace. Depending on the namespace a issuer is managing, the issuer could be an organization (e.g., UCLA may issue certificates to its enrolled students) or an individual (e.g., Yingdi may issue certificates for the keys in his personal devices). A key owner runs a certificate agent, which manage its keys and handles the certificate request on behalf of the key owner.

In order to automate certificate issuance process, a certificate issuer can specify a set of eligibility challenges. A certificate requester can obtain the requested certificate as long as it can accomplish the required challenges. Certificate issuers publish the challenge information as a part of the profile of issuance service. Key owners retrieve the profiles and load them into their certificate agent. The certificate agent can then follow the profile to automatically request certificates.

3.3.1.1 Issuer Profile

An issuer profile carries the information to direct certificate agent to automatically request a certificate from an issuer. It includes four pieces of information. The first part is the namespace of the issuer, with which the agent can help a key owner to select an appropriate issuer. The second part is the issuer's subject naming convention, with which the agent can construct a proper subject name of the requested certificate. The third part is the prefix under which the issuer publishes the result of certificate request, i.e., an issued certificate or a rejection.

The last part is a set of challenges that an issuer will use to validate the eligibility of a key owner. Such challenges may include proving the ownership of an email address, or a domain name, or a website. For example, if email is used as a challenge, the issuance system will send an challenge secret to the email address supplied by a key owner. Since only the owner of the email address can see the secret, a key owner can prove its ownership by sending the secret back (through the certificate agent) to the issuance service. If a domain name is used as challenge, the issuance system will send the secret directly to the certificate agent which will ask the certificate requester to create a special DNS record with the secret as value. The certificate issuance service can then make DNS query to validate the requester's ownership of the DNS domain.

Note that the issuer's subject naming convention restricts the certificate that a key owner can obtain by proving its ownership of a particular entity. For example, given a naming convention that can convert an email address to an NDN name (e.g., "`yingdi@ucla.edu`" to "`/edu/ucla/yingdi`"), a key owner can only acquire a certificate whose subject name corresponds to its email address.

3.3.1.2 Certificate Requesting Procedure

Whenever a key owner wants to request a certificate for its key, the key owner first selects an issuer whose profile has been loaded into the certificate agent (e.g., NDN testbed certificate issuer who is responsible for namespace "`/ndn`"). According to the profile, the certificate agent may ask the key owner to supply additional information to determine the requested certificate name. For example, the NDN testbed certificate issuer may require a key owner to supply an email address of any site that participates in the testbed. The certificate agent can follow the instructions in the issuer's profile to convert the email address (e.g., `yingdi@ucla.edu`) into a subject name in form of NDN name (e.g., "`/ndn/ucla/yingdi`"). With the knowledge of the requesting public key, the certificate agent can further construct the key name (e.g., "`/ndn/ucla/yingdi/43/KEY`").

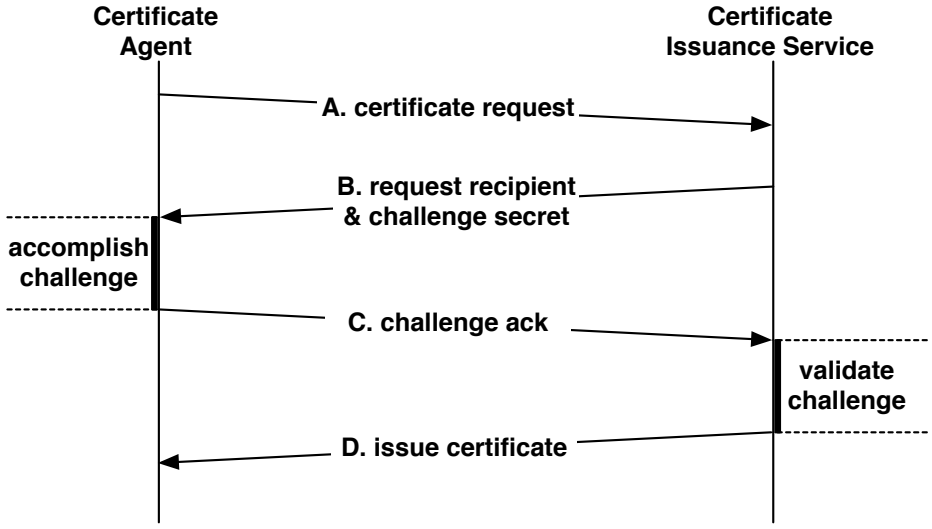

Figure 3.7: Workflow of automated certificate issuance system.

The certificate agent asks the key owner to select the challenges to accomplish and sends a certificate request to the corresponding certificate issuance service (Step A in Figure 3.7). A certificate request is expressed as an interest packet with all the information encoded in the interest name. As shown in Figure 3.8a, a

/[ServicePrefix]/Request/[KeyName]/[KeyBits]/[ChallengeSeletion]/[AgentKeyBits]

(a)

/[ServicePrefix]/Notification/[ReceiptNumber]/[OptionalResponse]

(b)

Figure 3.8: (a) Certificate request (in terms of interest name); (b) Challenge accomplish notification.

request name consists of five parts: 1) the name starts with the service prefix, so that the interest packet can be forwarded towards the service; 2) the requesting public key name and 3) the public key bits; 4) the set of challenges that the key owner has selected to answer; and 5) an agent specific account key used for maintenance later.

Upon receiving the certificate request, the issuance service generates a receipt number and the corresponding challenge secret, and keeps a record of the secret and the request for validation later. The service then replies with an receipt number of the request as a data packet(Step B). Depending on the type of challenges, the reply may also include challenge secrets as a part of the receipt (e.g., for DNS challenge and website challenge) or send challenge secrets through an out-of-band channel (e.g., sending a validation code through email). The secret is encrypted using the agent public key to ensure that the key owner (or its agent) is the only one that can see the secret in plaintext.

If the challenge secret is directly sent back to the certificate agent, the certificate agent decrypts the secret and inform the key owner to accomplish the challenge (e.g., adding the secret as a specific DNS record or publishing the secret with a specific URL). After the key owner accomplishes the challenge, the certificate agent can notify the issuance service (Step C) to validate the challenge by expressing another interest (Figure 3.8b). If a key owner receives the challenge secret through an out-of-band channel (e.g., email), the recipiency of the secret is already a eligibility proof. The key owner can input the secret into certifi-

cate agent, which decrypts the secret and sends it to the issuance service (in the `OptionalResponse` field of the notification interest). In all cases, the challenge secret is encrypted using the issuer's public key to ensure the secret is never leaked to anyone in the middle.

After validating a challenge, the issuance service can issue the certificate (Step D). When challenge secret is directly sent back as a part of validation notification, the validation is done immediately when the service receive the secret from certificate agent. The service can replay a data packet that encapsulates the issued certificate. In case that a challenge needs to be validated through an out-of-band channel (e.g., DNS query), the issuance service may not be able to issue the certificate immediately. Instead, the service replies to the certificate agent a name with which the agent can retrieve the validation result, i.e., an issued certificate or a rejection. A certificate agent can periodically express interest with the replied name until receiving the eventual result.

3.3.1.3 Certificate Revocation

Automated certificate issuance system provide a convenient way of obtaining a certificate, thus facilitating the use of short-lived certificates. In other words, instead of issuing a certificate with long validity period, a certificate issuance service can issue a sequence of certificates with short validity period and publishes these short-lived certificates one by one. Certificate owners can periodically retrieve a new version of the certificate in the same way as retrieving the first version of the certificate. Short-lived certificates can effectively eliminate the necessity of key revocation, because the issuance service can "revoke" a certificate by simply stopping publishing the new version of the certificate.

3.3.2 Certificate Bundle Publishing

Data producer must make its public key certificates available in the network, so that consumers can retrieve the certificate to authenticate data. Simply retrieving a producer certificate, however, may not be sufficient for a consumer to authenticate data. A consumer must also authenticate the retrieved certificate. The certificate authentication may require the consumer to retrieve the issuer's certificate. Although a consumer can follow the `KeyLocator` field to recursively retrieve all the required keys, the process may be time consuming when it involves a long trust chain. As a result, it may take a consumer application a long time before accepting the first data packet. To address this issue, we designed *certificate bundle* to speed up certificate retrieval.

A certificate bundle is a data packet that includes all the intermediate certificates on a trust chain to validate a producer public key. By retrieving a certificate bundle, one can immediately validate the producer public key as well as all the intermediate keys in the bundle. A data producer can prepare a certificate bundle for its key (e.g., a produce can build its own certificate bundle over its issuer's certificate bundle by simply appending the issuer certificate into the bundle) and publish it in the network. Instead of recursively retrieving each individual certificate in the trust chain respectively, consumers can extract the producer key name from the `KeyLocator` of data packets and retrieve the certificate bundle of the signing key directly.

Figure 3.9: Naming convention of certificate bundle.

We name certificate bundle after the target signing key as shown in Figure 3.9. In order to distinguish the certificate bundle with the certificate, we attach a

special name component "KEY-BUNDLE" after the key name. Since the intermediate certificates that a key needs for authentication may vary from one trust model to another trust model, an additional name component that describes the trust model is attached after "KEY-BUNDLE". Since certificates in a bundle may change over time, each certificate bundle also has its own version number.

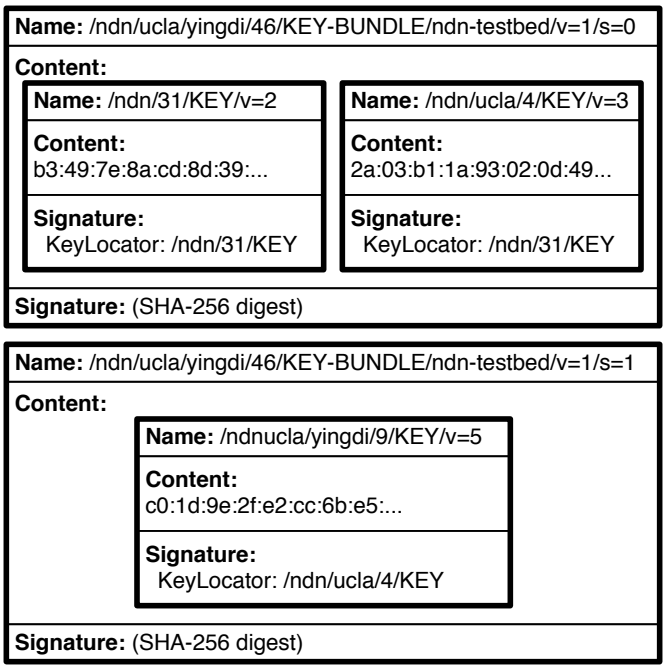

Figure 3.10: An example of certificate bundle.

The content of certificate bundle is a list of intermediate certificates arranged with the order from the trust anchor to the certificate of the target key (Figure 3.10). In case that the size of certificate bundle may exceed the maximum transmission unit (MTU), the certificate bundle can be divided into multiple segments.[4] Note that each individual certificate always stays in one segment.

As the content of certificate bundle is a set of certificates along the trust chain within a particular trust model, these certificates can be authenticated with their

[4]A certificate user can speed up the retrieval of segmented certificate bundle by sending multiple interests (with specified segment number) in parallel in order to retrieve all the segment at the same time.

own signatures, thus eliminating the need of putting a strong signature on the certificate bundle packet. Therefore, we use digest as the signature of certificate bundle packet only for the purpose of integrity checking.

3.4 Certificate Discussion

3.4.1 Specifying KeyLocator

We define the `KeyLocator` of an NDN certificate as the issuer's key name. Since a key name consists of the subject name and the key ID, the `KeyLocator` of a certificate is equivalent to the combination of the issuer name and issuer key ID in X.509 certificate. Note that the `KeyLocator` does not specify a specific certificate name but a key name. Therefore it is the consumer's decision whether to retrieve multiple certificates for the key, or a certificate made by a specific issuer if the consumer knows the ID of the issuer. Also note that the `KeyLocator` does not include any version number, because the certificate of issuer may also be updated over time. A consumer should be prepared for retrieving some old version certificate and keeping retrieving newer version in this case.

3.4.2 Maintaining intermediate key for multiple signature

In order to support multiple signatures, we introduce an intermediate key to address the scalability issue. Since the owner of the intermediate key can technically sign any data on behalf of all the signers, a natural question related to the intermediate key is: who controls the intermediate key? This problem can be addressed using the secret sharing techniques similar to the one introduced by Adi Shamir [Sha79], which can break a private key into multiple pieces and requires the presence of certain number of pieces to recover the original private key. Using these techniques, all the signers of an intermediate key can obtain a

piece of private key to avoid single owner of the intermediate key.

3.4.3 Unsolicited Certificate Issuance

In Section 3.3, we focused on on-demand certificate issuance, i.e., a key owner explicitly requests a certificate from an issuer. In some other cases, an issuer may generate a certificate without being requested by the key owner. One example is the endorsement system. For example, one may endorse a name-key binding that has been validated through certain amount of interaction. In this case, the key owner may not be aware of the existence of the certificate.

Key owner's unawareness may cause some problem in key publishing. The key owner has no way to collect the unsolicited certificates, while the certificate issuer may not be able to keep the certificate available in the network. One possible solution to this problem is to build a repository equivalent to the key server in PGP [CDF07]. A certificate issuer can upload the unsolicited certificates to the repository, and a key owner may periodically query the repository to collect new certificates for its key.

CHAPTER 4

Schematizing Trust

Designing secure systems and network applications involves properly authenticating multiple entities in the system and granting these entities with the minimum set of privileges necessary to perform operations. In contrast to traditional IP networks where applications usually rely on an additional layer (e.g., Transport Layer Security [DR08], IPSec [KS05]) to authenticate connections, Named Data Networking (NDN) requires every application to name and sign the produced network-level data packets and to authenticate received packets. To utilize the data-centric security of NDN without requiring application developers and users to be security experts, system-level support is needed to automate the process of packet signing and authentication.

The power of the NDN architecture comes from naming data hierarchically with the granularity of network-level packets and sealing named data with public key signatures. Producers use key names to indicate which public key a consumer should retrieve to verify signatures of produced data packets. In addition to fetching the specified keys and performing signature verification, consumers also match data and key names to determine whether the key is authorized to sign each specific data packet.

To facilitate this matching process, we introduce the concepts of *trust rules* and *trust schemas*. A set of trust rules defines a trust schema that instantiates an overall trust model of an application, i.e., what is (are) legitimate key(s) for each data packet that the application produces or consumes. The fundamental

idea is that each trust rule defines a relationship between the name of each piece of data and its signing key, e.g., both must share the same prefix, share the same suffix, and/or have specific name components at certain position of the names. Given a trust schema that correctly reflects the trust model of the application, data producers can select (and if necessary generate) the right keys to sign the produced data automatically, and consumers can properly authenticate each retrieved data packet.

Threats to data authentication integrity in NDN include failed authentication, mis-authentication, and key compromise. Failed authentication of a legitimate key (false negative) can result in a consumer treating valid data as malicious, potentially leading to denial of service. Mis-authentication of a mis-configured or malicious key (false positive) can cause consumers to accept false data. These errors can occur when the trust schema (data-key relations) is incorrectly or unclearly defined, or when the authentication mechanism does not fully adhere to the defined schema. A set of commonly used trust schemas written by security experts not only can mitigate these threats, but also facilitate automation of both signing and authentication mechanisms.

When a legitimate key is compromised, an attacker can obtain privileges associated with this key. To mitigate this threat we enforce "the least privilege principle": each key must have a restricted non-elevating usage scope to limit the damage upon key compromise, and keys with broader privileges should be used as infrequently as possible.

In this chapter, we describe how NDN naming and the use of *trust schemas* enable automation of data signing and authentication in NDN applications with complex trust models. We have implemented a prototype of a trust schema in NDN application development libraries (ndn-cxx and NDN-CCL) which have been used to power the trust management of several NDN applications, including the NDN Forwarding Daemon (NFD), NDN Link State Routing Protocol (NLSR),

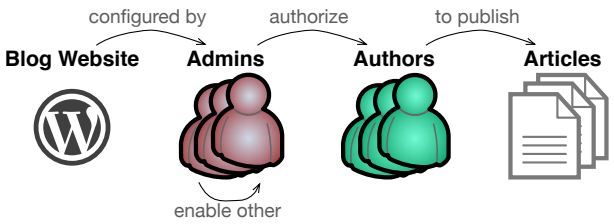

Figure 4.1: Entities of a simple blog website framework

NDN Domain Name System (NDNS), NDN Repository System (repo-ng), and ChronoChat applications [NDN15a].

4.1 Why We Need a Trust Schema

In general, the relationship between data and key names can be complex. Depending on an application's naming structure and trust model, data authentication may involve a trust chain (*authentication path*) across several different namespaces. We use a simplified blog website as an example throughout the chapter to illustrate a possible trust model and our proposed approach to schematize it. The framework includes four groups of entities (Figure 4.1): the website, website administrators, blog authors, and articles. The website may have a few administrators, who can authorize authors to publish articles on the website. Trust relations between these entities in NDN terms can be captured by signed data packets and chains of keys. When an administrator installs the website software, the installer generates a key[1] to act as the root of trust for the website. The installer process also creates a key for the initial administrator and signs it with the website's key. The initial administrator can further delegate management privileges to other administrators by signing their keys, and any administrator can add authors into the system by signing the authors' keys. Each author can publish on the website by signing the produced articles using a valid author key.

[1]This key may be self-signed or later secured using some trust model, e.g., PKI or web-of-trust.

When a reader retrieves an article, he or she can recursively follow the `KeyLocator` field in each data packet to retrieve the key of the author who wrote the article, the key of the administrator who authorized the author, and the key of the blog website where the article is published. If the reader accepts the website trust model and trusts the public key of the website (or uses PKI or web-of-trust mechanisms to verify authenticity of the key), the reader can reliably authenticate legitimate articles through a sequence of data packet signature verifications.

The example above illustrate the necessity of authentication across different namespaces, and highlight the need for the trust schema to concisely express complex trust model relations.

The blog website framework defines entities in the system and also their trust relationships. Since everything is explicitly named in NDN, the framework also needs to define a naming representation of the entities. Figure 4.2 shows a possible representation: assuming the website owns "`/a`" namespace and allocates "`/a/blog`" to blog publishing, articles are represented as data packets under the "`/a/blog/article`" namespace, with category, publication year, and unique article identifier; each author obtains a key under the "`/a/blog/author`" namespace with an author identifier;[2] each administrator obtains a key under the "`/a/blog/admin`" namespace with an administrator identifier; and the website itself has a configuration key with the name "`/a/blog`" (e.g., created during the installation of the blog). An implementation of this blog website framework must capture the trust relationship between all these entities in terms of the relationship between NDN namespaces. However, this comprehensive naming structure leads to the fact that an authentication path following the trust model may need to traverse three namespaces: "`/a/blog/article`", "`/a/blog/author`", and "`/a/blog/admin`" as shown in Figure 4.2.

[2]The last two components of each key name are "`KEY`" and a key identifier. This naming convention allows authors to change keys over time.

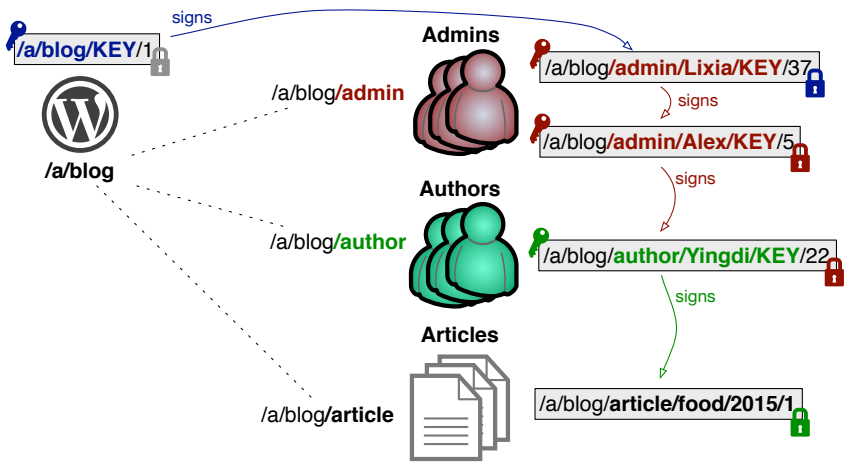

Figure 4.2: Example of namespaces and authentication paths in a blog website "`/a/blog`"

In theory, it is possible for application developers to hard-code all relationships in the trust model, i.e., relationships between articles and authors keys, between authors and administrators keys, between administrators keys and other administrators keys, and between administrators keys and the configuration key of the website. However in practice, even with a simple trust relationships as in our example, this process is non-trivial and error-prone. A small implementation error may compromise the security of the entire website. For example, a website implementation that accidentally associates author management with author keys rather than with administrator keys may allow authors to authorize another author without the permission from an administrator. Or, an article-publishing application that mistakenly uses an administrator key to directly sign an article violates the least privilege principle, and may also prevent browsers that comply with the trust model from authenticating articles.

In contrast, when the trust relationships are captured by a set of well-defined rules that match data and key names (*trust schema*), a system-level tool interpreting these rules can automatically execute authentication and signing procedures. This ability to automate unburdens developers from individually handling sophisticated data signing and authentication. A trust schema also makes it feasible for

security experts to define a set of generalized trust models (e.g., one for blog websites, one for mail services, etc.) that other application instances of the same type can reuse. Each reuse can continue to refine and debug the schema, improving it for future applications.

4.2 Trust Schema

In this section, we present the trust schema as a tool to define trust models in a generalized way. A trust schema comprises a set of linked trust rules and one or more trust anchors. As we will show later in this section, the trust schema mechanism can be used to automate both authentication and signing processes. To define trust schema rules, we will use a notation similar to regular expressions to express the *name pattern*. Table 4.1 gives a brief summary of the syntax elements we use in name patterns that are formally defined in [NDN15b].

Table 4.1: Elements of name patterns used in trust schema definitions

`<name>`	Match name component `name`
`<>`	Match any single name component, i.e., wildcard
`<name><>`	Match name component `name` followed by any single name component
`<>*`	Match any sequence of name components
`(...)`	Match pattern inside the brakets and assign it as an indexed sub-pattern
`\n`	Reference to the n-th indexed sub-pattern
`[func]`	Match (for authentication) or specialize (for signing) name component according to function `func` defined pattern, i.e., wildcard specializer
`rule(arg1,...)`	Derive a more specific name pattern from `rule`'s data name pattern with arguments `arg1`, ...

4.2.1 Trust Rule

A trust rule is an association of the data name with its corresponding signing key name. There are multiple ways to represent such association. For exam-

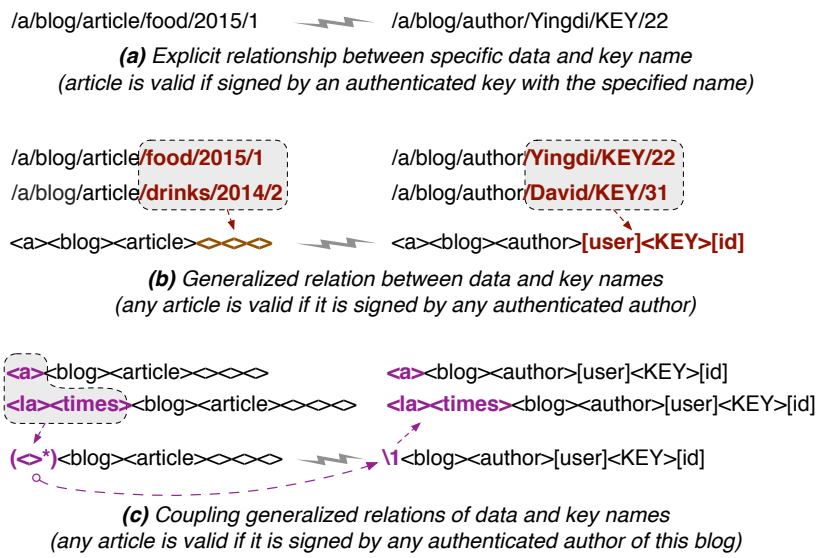

Figure 4.3: Trust rule generalization

ple, Figure 4.3(a) shows a simple direct association between an article name and its corresponding author name. This rule precisely captures that the article "`.../food/2015/1`" must be signed by author key "`.../Yingdi/KEY/22`", but says nothing about other articles or authors, even those that share the same naming patterns. If we can generalize the name relationships in trust rules, and reliably link rules to one another, we can construct concise, sophisticated, robust, and re-usable trust models.

4.2.1.1 Generalizing Trust Rules

A well-defined trust model usually associates the same type of data with the same type of keys, e.g., articles should always be signed by the authors. We can use the naming structure of a given application (or a set of applications that share the same naming structure) to create a set of rules to define the relationships between name patterns for data and keys in that application. This set of trust rules then captures the complete trust model for the application.

In the blog example, all articles share the same prefix "`/a/blog/article`",

but each article has its own category, year, and article identifier. One way to generalize this relationship is to use name patterns as shown in Figure 4.3(b). In Figure 4.3 and later examples, we use the wildcard "`<>`" to match any name component (i.e., the schema does not impose any restrictions on the content of the name component), "`[user]`" to match alphanumerical user identifiers, and "`[id]`" to match numerical key identifiers.

In general, trust models must explicitly associate a data name with its signing key name through matching of name components. In our example, both the article name and the author name must share the same website name ("`/a`"). To capture this constraint, we leverage sub-patterns and repetition syntax, as highlighted in Figure 4.3(c). We believe this syntax is sufficiently general to capture complex trust model frameworks, allowing reuse of trust models by different application instances. In other words, the trust schema for our blog example can be used by any other blog website that shares the same trust model.

4.2.1.2 Linking Trust Rules

A trust model should also properly associate keys with their signing keys, to ensure that a data consumer can reliably construct chains of keys to authenticate data and that a data producer can correctly choose or initialize its signing keys.

Figure 4.4(a) defines "`article`" and "`author`" trust rules. The key name pattern in the "`article`" rule will always match the data name pattern of the "`author`" rule, therefore both rules are implicitly linked. However, in order to ensure integrity of the trust model, the schema should unambiguously describe an authentication path (or paths) for each valid data packet. Therefore, each rule has to be explicitly linked to other rule(s) in the trust schema definition.

To explicitly link rules, we assign each rule a unique identifier to be used in a function-like way as part of the key name pattern, as shown on Figure 4.4(b).

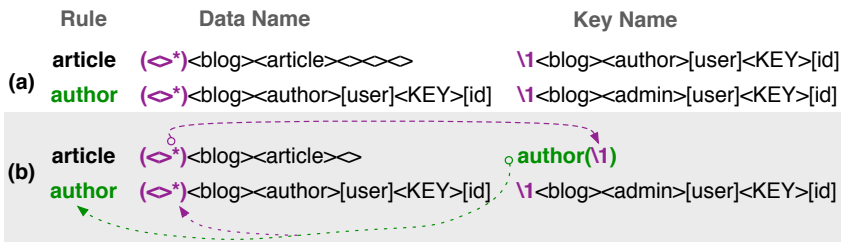

Figure 4.4: Generalization of trust rule linkage: (a) implicit linkage; (b) explicit linkage

In other words, invoking such rules is similar to invoking a function: invocation substitutes the key name pattern with the data name pattern from the invoked rule, specializing it with the supplied patterns or references to the indexed sub-patterns. In our example, the "`article`" rule invokes the "`author`" rule passing to it the first indexed sub-pattern. For the "`/a/blog/article/food/2015/1`" article, the sub-pattern will expand to "`/a`" and the invocation to the "`author`" rule will return "`<a><blog><author>[user]<KEY>[id]`" name pattern. This linkage imposes the restriction that only authorized authors of blog "`/a/blog`" can sign and publish articles of the blog.

4.2.2 Trust Anchor

To be complete, a trust schema must also include one or more trust anchors which serve as bootstrapping points for the trust model. A trust anchor is a key that is pre-authenticated using an out-of-band mechanism, e.g., manually installed or comes with software packages. In the trust schema we express trust anchors as special rules that include a key name pattern and a pre-authenticated key. Every successful authentication path must end at a trust anchor. Therefore, a trust schema must always include a way for trust rules to establish the link(s) from data or key names down to a trust anchor. Figure 4.5 shows an example of the trust rule "`admin`" linking to the trust anchor "`root`".

The trust anchor performs two important functions. First, it explicitly defines

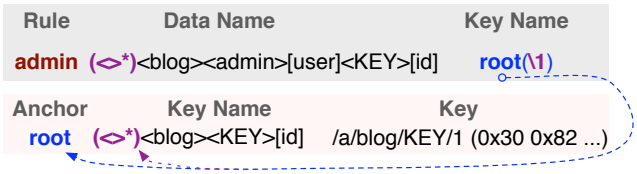

Figure 4.5: Example of linking trust rule and anchor

not only the name of the trust anchor, but also the key bits, i.e., if a packet is signed with a key that matches the name pattern in a trust anchor, this packet must be authenticated using the pre-specified key bits. Second, the anchor explicitly restricts the privilege of the pre-authenticated public key using name pattern, so that the key cannot be used to authorize anything else. For example, an administrator's key of another website "`/another/blog/admin/Carl`" will not be a valid administrator's key for "`/a/blog`": the expanded key pattern "`<another><blog><KEY>[id]`" will not match the blog's trust anchor "`/a/blog/KEY/1`". Note that the schema also prohibits another website's administrator key to be signed with the blog's trust anchor: the "`admin`" rule will rightfully reject such a key.

4.2.3 Crypto Requirements

In addition to providing a generalized formal definition of trust rules and trust anchors, a trust schema must also include cryptographic requirements on data signatures, such as the hash and signing algorithm and the minimum key size. These requirements are not directly related to naming, but can help prevent consumers from accepting data with easily compromised signatures. Therefore, a trust schema should clearly state these parameters as an essential part of a trust model.

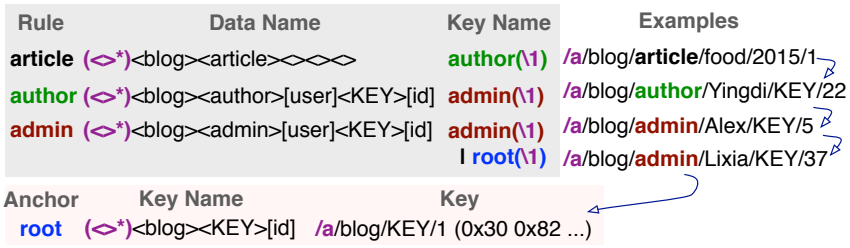

Figure 4.6: Trust schema the blog website framework with "`/a/KEY/1`" as the trust anchor

4.2.4 Trust Schema Examples

We now demonstrate how the trust schema we described so far can express two different trust models. The first trust model is for our blog website framework, and the second is an example of a model that resembles the trust model of DNSSEC and *strictly* follows the naming hierarchy to match data and key names.

4.2.4.1 Blog Website Framework

In the blog website example, the trust rules must capture the relationship between articles and authors, between authors and administrators, as well as between administrators and blog website configuration (the blog's trust anchor). An example of the trust schema that can achieve these goals is shown in Figure 4.6. Note that this schema assumes that the blog's configuration key "`/a/blog/KEY/1`" is pre-authenticated (i.e., a trust anchor). Depending on the specific usage scenario, a blog reader may further authenticate the configuration key using a hierarchical trust model similar to the example in Section 4.2.4.2, or using some other trust model, e.g., web-of-trust.

The first rule in the example schema, "`article`", captures the trust constraint that authors must sign their articles with their keys. Similarly, the "`author`" rule ensures that only blog administrators can sign authors' keys. The final "`admin`" rule defines two possible relations for administrators' keys in the security frame-

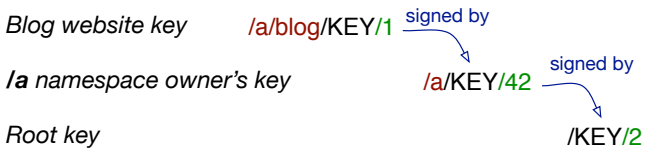

Figure 4.7: Example of naming in hierarchical trust model

work: (1) existing administrators may delegate administrator privileges to another person; and (2) authentication paths for the administrator keys must terminate at the blog website trust anchor.

Note that although every trust rule in the trust schema in Figure 4.6 uses the repeated wildcard "`<>*`" to match the website prefix, the prefix is always determined (specialized) at the moment when the "`article`" rule captures the original article data name. After the "`article`" rule captures "`/a/blog/article/food/2015/1`" data, prefix "`/a`" is propagated to the "`author`" rule as a reference to the first sub-pattern, then to the "`admin`" rule, and down to the "`root`" trust anchor.

4.2.4.2 Hierarchical Trust Model

In a linear hierarchical trust model, with DNSSEC [AAL05] as a prominent example, a single rule can capture the relationship between all the data and key names; in plain English, this rule is "the signing key name must be a prefix of the data name." Because key names should be unique and need to include additional suffix components as shown on Figure 4.7, the trust schema for the hierarchical relationship in NDN needs to consider these additional components.[3] The overall trust in this model can be bootstrapped using one or more trust anchors associated with the top level namespace(s).

Figure 4.8 shows an example of the trust schema that defines the hierarchical

[3]For simplicity, in this example we consider only authentication of DNS keys, but the trust model and schema can be easily extended to other DNS data, as shown with the blog website example.

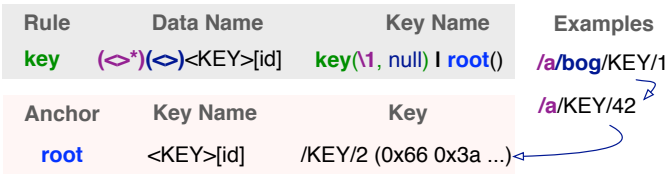

Figure 4.8: Trust schema for the hierarchical trust model with "`/KEY/2`" as the trust anchor

trust relationships, consisting of a single rule and a trust anchor. The rule "`key`" captures that keys at each level of the hierarchy must be signed by the keys from the parent namespace, i.e., the prefix before "`KEY`" of the signing key name must be one component shorter than the name of the key itself. The trust anchor ensures that the authentication path discovery terminates when it reaches the root namespace: when the prefix of the signing key before "`KEY`" is empty (just "`/`"), then it must be signed by the specified "`/KEY/2`" key.

The "`key`" rule is recursively linked to itself and to the trust anchor. In these cases, when matching data and key names, all specified patterns need to be considered, with anchor rules taking precedence. For a key "`/a/blog/KEY/1`", the rule "`key`" will extract the parent namespace of the key (i.e., "`/a`") and derive two name patterns: "`<a><KEY>[id]`" and "`<KEY><2>`". Given the signing key name matches the first pattern, the process recursively continues with the same rule, until there is a match with the trust anchor.

If the key's `KeyLocator` does not match any key name pattern, it implies that the key does not comply with the trust model and should be treated as an invalid key.

4.2.5 Schema for Authentication

For each data packet, the trust schema determines a valid authentication path(s) within the corresponding trust model. Given that the trust schema is expressed as formally defined rules, an *authentication interpreter* of the trust schema can auto-

mate the whole authentication process for any given trust model (Section 4.3.1).

For each received data packet, the authenticating interpreter finds the corresponding trust rule by matching the name of the packet against the specified name patterns in the rules. If the packet and its `KeyLocator` comply with constraints of the found trust rule, the interpreter can then retrieve the public key according to the data's `KeyLocator` and recursively inspect the retrieved key according to the trust schema, until reaching a trust anchor or a pre-defined limit on the number of recursive steps. In the former case, the interpreter has collected all the intermediate public keys on a valid authentication path, thus can verify signatures starting from the trust anchor up to the received data packet. When the interpreter cannot find a rule that matches the received data packet, or the constructed authentication path loops, or the path becomes overly long, the interpreter declares failure to discover the authentication path.

The received data packet is authenticated only if there is a valid authentication path according to the trust schema, and each signature on the path is verifiable and satisfies the cryptographic requirements of the schema. In other words, either failure to discover authentication path or failure to verify any signature on the authentication path implies that the received data packet cannot be authenticated with the interpreted trust model.

4.2.6 Schema for Signing

One can also view the trust schema as a collection of constraints on a data packet's signing key, with respect to its name, signature, key type and size, etc. Thus, the trust schema also specifies the required signing process, i.e., how to select or generate signing keys given the name of the data packet. Effectively, this allows automation of the signing process using a *signing interpreter* of the trust schema (Section 4.3.2).

The signing interpreter takes a data packet as an input and looks up the corresponding trust rule. Instead of checking for compliance of the data's name and `KeyLocator` to the trust rule, it infers the correct name of the key to be used to sign the data packet. If this key exists on the system, the interpreter will immediately sign and return the data packet. If the key does not exist, the interpreter will try to generate the key with the specified name and crypto requirements, and then sign this key by recursively re-interpreting the same schema again with the generated key as a new input. See further details in Section 4.3.2 on how the interpreter can generate key names based on rules in the trust schema.

Note that it is not always possible for the interpreter to automatically generate all necessary keys, without out-of-band verification mechanisms. For example, if a not-yet authorized author is trying to sign an article for publication, the interpreter will fail to sign it, as the author does not have a valid key to sign an article, nor a key to endorse an author on the blog, nor a key to configure a new administrator in the system. Even in this case, the interpreter can still generate useful diagnostic information, e.g., which keys are missing and how to obtain them.

4.3 Automating Trust

Now that we have introduced the concept of schema-based data authentication and signing, we will describe in detail how to automate these processes, using the blog website framework as an example.

4.3.1 Automating Authentication

Each step of the authentication path for data (key) packets is defined by the rules of the trust schema. Rules are linked together through a function-like invocation of rule names as part of the key name pattern definition, as shown in Figure 4.6.

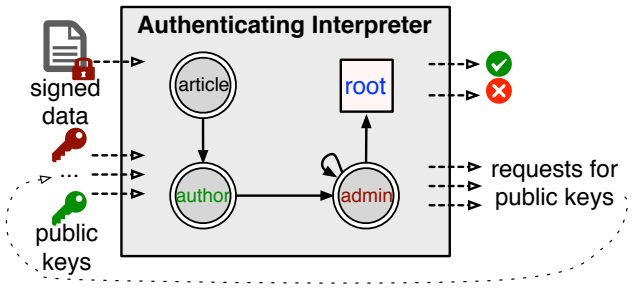

Figure 4.9: Finite state machine for the authentication interpreter of the blog website trust model schema

The authentication process moves forward (from one step to the next) only if the data (or key) satisfies the conditions of the rule. We can model this authentication process as a Finite State Machine (FSM), with each state representing a rule and state transitions representing function-like invocations. This way, once a data packet enters the FSM, the FSM's states define the packet's authentication path, and an automatic process can walk through these states until exiting the FSM with success or failure.

Execution of the FSM processing requires a trust schema interpreter. The interpreter used for data authentication, which we call *authenticating interpreter*, takes data packets as input, requests public keys when necessary, and outputs whether the received packet is authenticated or not. Given the trust schema for a trust model, an authenticating interpreter can effectively automate the process of data authentication for this trust model. Figure 4.9 shows the FSM of an authenticating interpreter for the blog website trust model discussed in Section 4.2.4.1.

4.3.1.1 Authentication State

Whenever a new data packet arrives at the FSM, the interpreter determines the corresponding initial state by checking the data name against the name patterns for each state. After that, the interpreter initiates the key name checking procedure, including steps to:

- extract components from the data name according to the defined sub-patterns;
- derive the key name pattern from the rule's key name functions with the extracted components;
- check if the data's `KeyLocator` matches the derived key name pattern.

If the data packet passes the key name checking, the authentication process transitions to the downstream state of the FSM: the interpreter requests the key identified by the `KeyLocator` field carried in this packet and pauses FSM processing until the key is retrieved. When the key is delivered to the interpreter, the interpreter initiates a new instance of the same checking procedure at the state on which the FSM processing previously paused. Whenever the FSM transitions to a trust anchor state, the interpreter immediately triggers verification of signatures, following the reverse path of transitions in the FSM.

4.3.1.2 Walking Through the State Machine

In this section we demonstrate how the authentication automation can work for the blog website trust model. We use an article data packet with name "`/a/blog/article/food/2015/1`" signed by an author key "`/a/blog/author/Yingdi/KEY/22`" as an example to show how the authentication process goes through the state machine shown in Figure 4.9.

Initial state Based on the trust schema, the article name "`/a/blog/article/food/2015/1`" will be captured by the "`article`" rule, thus the authentication process starts from the corresponding "`article`" state. When executing the key name checking procedure, the interpreter will extract "`<a>`" as the first sub-pattern and use it to derive a key name pattern through a function-like invocation of the "`author`" rule. The resulting pattern "`<a><blog><author>[user]<KEY>`

[id]" will successfully match the KeyLocator field of the data packet and the FSM will transition to the downstream "author" state.

State transition At this point, the interpreter makes a request for "/a/blog /author/Yingdi/KEY/22" key and pauses processing until the key is retrieved. After retrieving the requested key, the interpreter resumes operations at the "author" state with the retrieved key as an input. Similarly, the interpreter extracts "<a>" as the first sub-pattern from the author key name and derives through the "admin" rule a key name pattern "<a><blog><admin>[user]<KEY>[id]". Assuming that the retrieved key is signed with an admin key "/a/blog/admin /Alex/KEY/5", the FSM will transition to the corresponding "admin" state.

Self-loop transition The "admin" rule in the website trust schema links to two trust rules, of which one is the "admin" rule itself. This self-linked rule represents a management privilege delegation from one administrator to another administrator and is represented by a self-loop transition in the FSM. This transition can capture an administrator key "/a/blog/admin/Alex/KEY/5" signed with another administrator key "/a/blog/admin/Lixia/KEY/37". In this case, the FSM transitions to the same "admin" state over the loopback link and the interpreter requests for the other administrator key and pauses the FSM processing again.

Note that a self-loop transition can potentially accept authentication paths that contain loops or excessively long authentication paths. To prevent these loops, the interpreter can record names of every intermediate key that each state has observed during the authentication process, and abort processing when detecting a duplicate. To prevent excessively long authentication paths, e.g., from a carefully crafted key chains in attempts to cause denial of service attacks, the interpreter should set a limit on the number of state transitions.

Transitioning towards the trust anchor state When the interpreter retrieves the public key "`/a/blog/admin/Lixia/KEY/37`", it can repeat the key name checking procedure on the "`admin`" again, deriving two patterns for key name matching: "`<a><blog><admin>[user]<KEY>[id]`" (from the "`admin`" rule) and "`<a><blog><KEY>[id]`" (from the trust anchor "`root`"). If "`/a/blog/admin/Lixia/KEY/37`" key was signed by "`/a/blog/KEY/1`" (the specified trust anchor), the second name pattern would match the `KeyLocator`. In this case, the process immediately transitions to the trust anchor state, triggering initiation of the signature verification procedure.

Signature verification Once the signature verification procedure is triggered, the interpreter will follow the reverse path of FSM back to the original data packet, terminating with failure if at any step it cannot verify the signature. In the example, the process will start with validating "`/a/blog/admin/Lixia/KEY/37`" key using the trust anchor key, following checking signature of "`/a/blog/admin/Alex/KEY/5`" using the validated admin key, similarly for the author key "`/a/blog/author/Yingdi/KEY/22`", terminating with checking signature of the received article data packet using validated author key.

4.3.2 Automating Signing

Another version of the trust schema interpreter, a *signing interpreter*, can use a trust schema to automate selection of signing keys and generation of keys when necessary/possible. Similar to the authenticating interpreter, the signing interpreter compiles a trust schema to an FSM (Figure 4.10), but processes an *unsigned* data packet as input and outputs the data packet signed with a key that conforms to the trust model (or fails). During processing, the interpreter interacts with the private key store (e.g., Trusted Platform Module, TPM) to request data signing

and create signing keys when they are not yet available.[4]

The signing interpreter will fail to sign the supplied data packet if the required key is not available in the local TPM and when the key generation procedure has to cross security boundaries, e.g., a remote admin needs to sign new author's key. In such cases, the signing interpreter can resort to the automated certificate issuance system to acquire a desired certificate. While it is impossible for the signing interpeter to generate keys completely automatically, it can provide assistance in creating the required keys (e.g., generate signing requests) and simplify complex cryptographic operations.

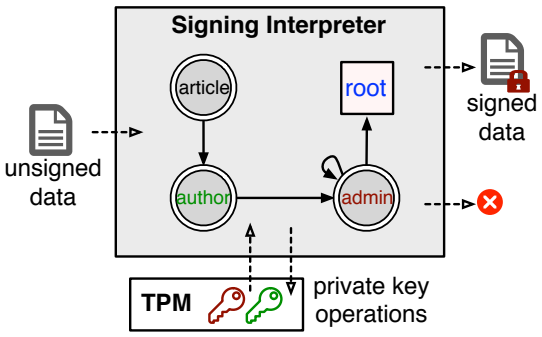

Figure 4.10: Signing interpreter for the blog website trust model schema

4.3.2.1 Key Selection

Given a data packet, the signing interpreter can derive the name pattern of a key that is allowed to sign this data according to the trust model. For this purpose, it finds the state in the FSM that corresponds to the data packet, and expands the corresponding signing key name pattern. For example, let us assume that an administrator of the blog wants to publish his article "`/a/blog/article/snacks/2015/3`". This data packet will enter the FSM from the "`article`" state, at which point the interpreter can derive the key name pattern "`<a><blog><author>[user]<KEY>[id]`", as shown in step 1 in Figure 4.11. With the derived

[4]Ideally, a signing interpreter should be implemented as a trusted service provided by operating system.

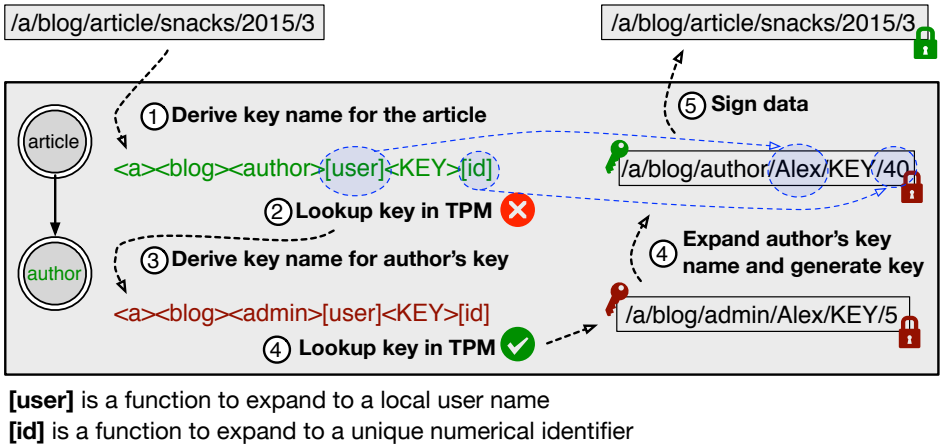

Figure 4.11: An interpreter processing the blog website trust schema directs the procedure of signing data "`/a/blog/article/snacks/2015/3`"

name pattern and the crypto requirements from the trust schema, the interpreter will search a qualified key in the TPM (step 2 in Figure 4.11). In our example, the admin is publishing a blog article for the first time, and is not yet authorized to do so, but the signing interpreter of the trust schema can automatically create such authorization, as we will show below.

4.3.2.2 Creating Keys

When the interpreter cannot find a signing key that corresponds to a state of the FSM (the result of step 2 in Figure 4.11), it transitions to a downstream state and repeats the key searching procedure. In our example, when the interpreter realizes that there are no author keys available, it will try to find out if there is any administrator key available. If not, the FSM will continue to transition downstream and repeat the search, until there are no more possible transitions available (note that self-loop transitions are skipped in the signing process when the signing key does not exist in the private key store). At this point the interpreter aborts the signing operation, as it will not be able to sign anything that will conform to the trust model.

In our example, the signing interpreter has access to the administrator's key (i.e., the author is also an administrator of the blog), and it will try to create a new author key. In order to create such key, the interpreter must derive a name for the key. In this case, the wildcard specializer `[func]` (Table 4.1) in a key name pattern can expand to specialize the key name. For example, `[user]` can specialize the name component for the author identifier using the local user name (e.g., "`Alex`"), and `[id]` can generate a unique identifier for the key. Therefore, at step 4 in Figure 4.11 (dotted blue lines), the interpreter can expand the author key pattern into "`/a/blog/author/Alex/KEY/40`". At this point, the interpreter is ready to generate an author key that satisfies the crypto requirements and overall trust model specified in the schema (step 4 on Figure 4.11), after which it will be ready to sign data packets of the article by this author (step 5 on Figure 4.11).

4.4 Discussion

Having described the trust schema and its applications, in this section we discuss the lessons learned, ongoing efforts, and remaining research issues.

4.4.1 Design Pattern for Security

A trust schema is more than just an approach to describe the relationships between data and key names, it also represents a design pattern to implement NDN security. Similar to design patterns in software engineering [GHJ94], which provide general reusable solutions to commonly occurring problems in software design, the trust schema provides a reusable solution of applying commonly used trust models in NDN applications. Security experts can define a set of trust schemas as the security patterns for frequently used data authentication models. An established set of trust schemas can greatly reduce the burden on NDN application developers, who can select an appropriate security pattern for their applications

during the design phase, to gain all the benefit of NDN's built-in security features.

4.4.2 Trust Schema Retrieval

A trust schema can be represented as NDN data packet(s), i.e., it can be named and signed. In this paper, we do not define a particular naming convention for trust schema. A meaningful name of a trust schema should be related to the name of the corresponding trust anchor, so that once a consumer learns a trust anchor, the consumer can retrieve and authenticate the trust schema.

Representing trust schema as NDN data packet allows multiple trust schemas to be combined (or chained) together: one can define a meta trust schema to authenticate other trust schemas. For example, an operating system manufacturer can use this feature to limit software installation, execution, and access to private key stores on the operating system only to applications with authenticated trust schemas. This is similar to the existing application sandboxing approaches (such as Apple's App Store and Google's Google Play), but gives operating system additional flexibility in controlling applications.

4.4.3 Key Caching & Bundling

In our examples, data authentication processes walked through the complete authentication paths defined by the trust schema. However, these processes can be optimized by utilizing cached keys that have been authenticated, given that a single key usually signs multiple data packets (e.g., an author uses the same key to sign multiple articles, an administrator uses his key to sign keys of authors, etc.). An interpreter can cache each intermediate key of an authentication process at the state where the key is checked and verified, so that a new authentication process may find one of its intermediate keys in those states before reaching a trust anchor.

Note that even with key caching, the first authentication process may still involve several round trips of retrieving intermediate keys. This process can be further optimized by having the data producer to maintain the chain of intermediate keys and making it available in form of a key bundle. By retrieving a key bundle, an authentication interpreter obtains all the required keys in a single retrieval. In fact, it has been a common practice in existing authentication systems (such as TLS [DR08]) to keep a complete chain of keys at the key owner side.

4.4.4 Multi-Path Authentication

Trust models that define single authentication path for data (e.g., PKI [CSF08], DNSSEC [AAL05]) are a common concern in the security research community. Having just one way to authenticate data creates a single point of failure, e.g., failing to timely renew certificate of any of the intermediate keys will result in data authentication failure. When multiple authentication paths are available, allowing any of the paths to authenticate data improves security resiliency of applications to maintenance failures. At the same time, by imposing a requirement that a key must be authenticated through a certain number of paths, applications can mitigate the damage of key compromise.

If/when a data packet can carry multiple signatures [Yu15], a trust model defined with a trust schema can associate the data name with key names across different namespaces. One of our ongoing directions is exploration of a variety of conditions on trust rules, such as "any valid", "all valid", etc.

4.4.5 Trust Bootstrapping

In describing the blog website example, we assume that data consumers have already obtained the trust anchor of the website. In general, a consumer may not always be able to obtain the trust anchor for each website it visits a priori. In

today's practice, a consumer may need to bootstrap trust from a limited number of pre-configured trusted keys and eventually establish trust on a particular website's trust anchor. We believe that bootstrapping trust remains as an important and challenging open issue, which is beyond the scope of this paper but included in our ongoing efforts. Besides the use of the existing Internet style PKI in NDN networks, more exciting directions to explore this open issue include realizations of web-of-trust and evidentiality-based trust bootstrapping models.

4.4.6 Signature Revocation

Signature revocation is an open research problem in NDN trust management. Although this problem is beyond the scope of this paper, our ongoing efforts explore the following approaches:

- constraining validity period of issued signatures, which may require mechanisms to certify validity of the signature at the time of creation (e.g., using secure timestamp);
- using trusted services to certify current validity of the signature, similar to revocation lists and OCSP in current PKI.

4.4.7 Formal Trust Schema Syntax

The syntax we used to describe the trust schema is still at an experimental stage. Trust schemas share many design philosophies with logic programming languages (such as Prolog [CM03]). It may be helpful to unify the trust schema syntax with formal syntax used by existing languages, and we would like to encourage researchers to apply techniques of programming language to enhance the trust schema design and improve the security of NDN applications.

CHAPTER 5

DeLorean: Long-Lived Data Authenticity

Named Data Networking (NDN) changes the network communication model from "delivering packets to an end host" to "retrieving (immutable) data by name," enabling and integrating many of the long sought-after functions into a unified network delivery framework, including efficient data distribution via multicast, delay-tolerant communication, ad hoc communication, and many more. This change in communication semantics relies on a data-centric security model, which is in part realized through digital signatures on every network-level data packet. Regardless from where a data packet is retrieved, it can always be authenticated directly, i.e., without trusting either the data storage or delivery channels.

Unlike physical signatures, digital signatures may not be considered trustworthy over prolonged time periods: given enough computation power and time, it is possible to reconstruct the corresponding private key and issue impersonated signatures.[1] In addition, each created signature "weakens" the privacy of the private key [BBD01], and there is also a chance that the keys get accidentally or maliciously leaked to adversaries. As a result, the current practices recommend the use of relatively short-lived signatures/certificates (from serval months to couple years) [BHK15]. This limited lifetime span works well for channel-based security model since communication channels have a limited duration, but not so well for data-centric security model of NDN. The lifetime of an NDN data packet can outlive the lifetime of its signature, especially in cases of historical data archives.

[1] Luckily, with the current computation technology and reasonably strong keys, it would take many years to do so [BBB15].

In this chapter, we propose an authentication system for NDN data archives, dubbed NDN DeLorean, which uses a *look back* data authentication model: data authentication is performed with the clock rolled back to the time of the data creation. In order to allow consumers to securely rollback the reference time for data authentication, we designed a publicly auditable timestamp service that issues proofs of data creation times by logging the fingerprints of archived data in the form of Merkle tree (Section 5.3). Given a data packet, the certificates that authenticate its signature (certification chain), and the proof of the creation time of data and certificates, one can always authenticate the data, regardless of the signature expiration and even the fact that the private key may have become known to everybody.

Our main contributions in this work include a) the look back validation model as the solution to long-lived data maintenance in NDN (Section 5.2), and b) the design of the first publicly auditable timestamp service over NDN (Section 5.3). They represent a significant step toward effective authentication of long-lived data. We also identified a number of remaining issues (Section 5.1) to be addressed in our future work to complete the construction of a fully functional validation system for long-lived data.

5.1 Threat Model

We focus on *long-lived* data, i.e., the data packets or data collections that need to be preserved for a long period of time. Typical examples of such data include newspaper articles, library archives, historical records, experimental results, etc. Although DeLorean could be used in all scenarios, it may be considered prohibitive expensive in terms of processing and storage overheads if one were to use DeLorean for unbounded volumes of data. The key security issue we address in this chapter is to ensure that the long-lived data can stay authenticatable, potentially many years

after the data producer ceased to exist. Note our focus is on the authentication aspect of the data; ensuring long-term secrecy of confidential data is outside the scope of this work.

We assume that the cryptographic keys in the the corresponding trust chains of long-lived data have limited validity periods in order to restrict key exposure and potential harm of the key compromise. We also assume that the consumers know which trust schema should be used to authenticate data and that the trust anchors do not change over time. We plan to address these assumptions in the next milestone of our research.

In order to authenticate the data with the above assumptions, NDN DeLorean implements a notary service that "certifies" the existence of data at particular points of time. To ensure that this third party service behaves correctly, there must be continuous audit of its consistency by either or both dedicated parties and volunteers (*auditors*). Potential misbehaviors of the notary service include timestamp *denial*, *repudiation*, *reordering*, and *injection*: the timestamp notary should not be able to deny access to the previously issued proofs, pretend that the previously issued proof is invalid, alter timestamp of the existing proof, or inject a new proof for a past timestamp. Any such misbehavior can be noticed by the public auditors, who can then take actions to remediate the issue: request immediate correction of the timestamp service behavior or switch to alternative timestamp service provider. The design described in this chapter assumes the existence of a single timestamp service; we briefly discuss how multiple alternative timestamp services can co-exist in Section 5.5.

In this chapter we assume that the key used to sign data is valid during timestamp certification and is not leaked during its validity period, i.e., there is no *producer impersonation* while the trust chains are within their validity. For example, if a USA Today article is timestamped at time t, we assume this article has a valid signature at the time, and only USA Today can create the signature

during the validity period of the corresponding trust chain. As part of our future work we plan to extend the design to incorporate revocation of the archived data to address potential producer impersonation problem during validity periods of the keys.

However, an attacker may launch impersonation attack after the timestamp creation. In this case, an attacker may pre-produce data signed by an uncertified key and record the data in DeLorean chronicle (described below). The attacker may recover the key of the victim's certificate issuer after certain amount of time (through key leaking or brute force computation). At this point, the attacker can create a certificate for its previously uncertified key and claim that the data was valid when it was produced. Our counter measure to this is to timestamp both data and their signing keys, and ask consumers to verify the existence of both data and keys. In this way, consumers can reject the falsified certificate because it cannot proof its existence before the data production.

5.2 DeLorean Overview

In this section, we present a high-level overview of DeLorean, a verifiable and publicly audited timestamp service, as the solution to the threats described above.

DeLorean is an always-on service that publishes a data "chronicle" (Figure 5.1). The chronicle consists of a sequence of volumes, each containing fingerprints of the witnessed data packets, such as specific USA Today articles, within a fixed timeslot, e.g., 10 minutes. The existence of a data packet (its fingerprint) in a particular volume is a *timestamp proof* that the data packet has existed before the end of the corresponding time slot. Each volume is finalized at the end of each time slot and published as a set of data packets, given the volume's information may not fit into a single data packet. After the volume is finalized, it cannot be changed without invalidating consistency with any future volumes.

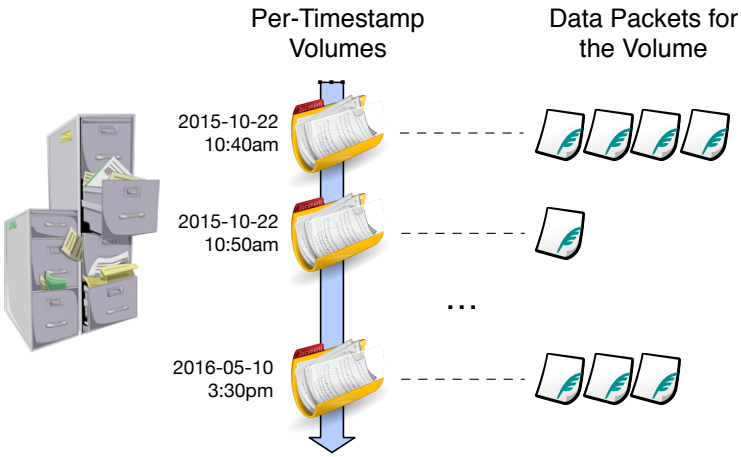

Figure 5.1: DeLorean's data chronicle

At any time, a data producer (article's author) or an archive service on the producer's behalf (USA Today publisher) can request a timestamp proof for data (articles) from DeLorean (Flow P.1 in Figure 5.2), supplying a fingerpint of the archived data in form of a hash digest of an individual data packet or a digest of the manifest that represents a data collection. The response to this request is a name of the chronicle volume that will be published by DeLorean at the end of the current cycle (Flow D) and the index of the fingerprint in the volume. After waiting until the volume is ready (on average a 5 minute wait in our example), the producer can retrieve the volume to verify whether DeLorean has included the data fingerprint in the volume (Flow P.2). In the end, the producer can publish the timestamp proof, which includes the full name of the volume and the index of data fingerprint, alongside the data (Flows P.3).

To verify data independently of its signature validity, consumers need to "look back" to the timepoint when data was produced (or time stamped). A consumer first obtains the corresponding timestamp proof, which can be stored alongside the data (Flow C.1), and verifies the data existence by retrieving several additional DeLorean volumes (see Section 5.3). Similarly, the consumer verifies the existence of the data's signing key certificates.[2] With all certificates proving their existences,

[2]Certificate issuers request timestamp proofs for the issued certificates. Alternatively, a

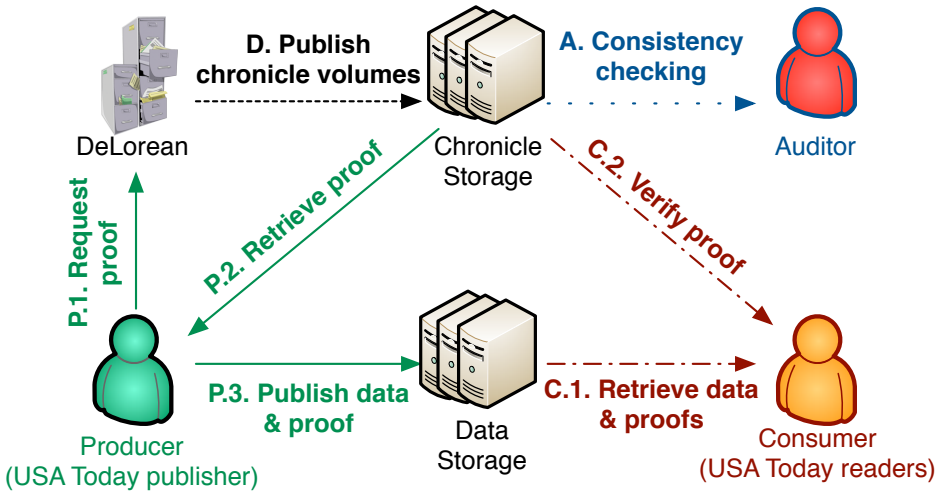

Figure 5.2: DeLorean workflow

the consumer can verify the data signature as if it was at the time of production or time stampping.

In order to ensure the correct and truthful operations of DeLorean, a set of third-party auditors continuously check the consistency of the chronicle (Flow A), i.e., checking that DeLorean has not modified the previously published volumes. If auditors detect that DeLorean has modified the chronicle, the users of the service (auditors, data producers, and consumers) will take immediate actions to either fix the issue or abandon the specific instance of DeLorean service. In order to guarantee consistency, each DeLorean volume has to be retrieved at least by one auditor around the time it is published. The more auditors are involved in the process, the less frequently each individual auditor needs to perform consistency checking. Note that although consumer and producer roles are separated from the auditor role in Figure 5.2, they can be (and, from security perspective, should be) combined.

---

data producer can request and publish the timestamp proofs of the data and the corresponding certificates as a bundle, similar to our previous certificate bundle proposal (Section 3.3.2).

5.2.1 Design Objectives

The first DeLorean's design objective is to minimize the storage and verification overheads, as overwhelming overheads would prohibit any use of system. For example, a naïve solution to consistence verification is to ask each auditor to maintain a local copy of the whole chronicle by retrieving volumes at the end of each time slot. Such solution not only requires impractical storage at the auditor side, but also requires each auditor to timely retrieve each volume. On the other hand, if consistency verification cost is trivial, it can encourage more users to audit DeLorean, improving fidelity and overall usefulness of the system.

The second design objective is to prevent DeLorean from knowing identities of auditors and knowing if audit requests are coming from the same auditor(s) or not. If DeLorean could do that, it would be able to present one consistent chronicle to a group of auditors while presenting a completely different consistent chronicle to another. In this case, none of auditors in the two groups can detect any modification, while the timestamp service would be obviously inconsistent.

The third design objective is to minimize the maintenance overhead. Given the number of volumes increases over the time, the amount of data stored in the volume should be sufficiently concise yet faithfully record all evidence in the corresponding time periods. Moreover, consistence and existence verifications involve continuous retrieval of the published volumes, desiring as low overhead as possible.

For the these design objectives, we will present the solutions in detail in Section 5.3. However, there are several design objectives that we have not addressed yet. For example, how to prevent a single party from monopolizing the recording of chronicle; how to increase the robustness of DeLorean; and how to reboot DeLorean in case of inevitable failures. We will discuss the potential solutions to these objectives in Section 5.5.

5.3 DeLorean Design

In this section, we present the design details of DeLorean. At the end of every time slot, DeLorean publishes a volume in the chronicle in a form of one or multiple data packets, to archive data packets recorded during the time slot. For the sake of simplicity, we first assume that DeLorean publishes a volume as a single data packet and explain how to expand the capacity of a single volume with multiple data packets in In Section 5.3.3.

The volumes per se however are also archive data. A simple solution to ensure authenticity of old volumes would be inclusion of crypto hash digest of the previous volume in a new volume, effectively constructing a hash chain of chronicle volumes. In this case, to authenticate any historical volume, one needs to authenticate the latest volume which should have a valid signature, and then verify the authenticity of previous volumes by checking their hashes one by one.

This hash chain based chronicle however results in large overhead, as authentication time would require $O(n)$ volume retrievals, where n is the number of time slots between the current time slot and the time slot of the volume under verification. For example, with 10-minute time slot, one has to retrieve a prohibitive amount of records (about a million) to authenticate a volume 20 years ago.

Inspired by Certificate Transparency [Lau14], we design DeLorean chronicle as a Merkle tree [Mer80] to minimize the authentication overhead. As we explain in detail in Section 5.3.1, the state of the chronicle is represented as a Merkle tree with volumes represented as leaves of the tree, where the root node effectively stores a hash of all volumes published so far. With this structure, the verification overhead of an old volume can be reduced to a much smaller number of operations $O(\log n)$. For example, the same 20 year old record can be authenticated by DeLorean with the binary Merkle tree using just several dozens of retrievals, which can be even further reduced by selecting a more optimal tree structure.

Note that the chronicle volume authentication by itself does not provide guarantee of the DeLorean chronicle consistency, i.e., DeLorean can still arbitrarily inject data into past volumes and then re-create subsequent volumes. To provide the guarantee, a set of auditors (dedicated entities, consumers, and producers) is required to periodically retrieve and authenticate the current volume (current state) and check consistency with previously fetched volumes (past state). With the trivial verification overhead, made possible by the use of Merkle trees, the consistency check between the current and any previously verified state is a trivial task (Section 5.3.2), which can be performed by a large number of auditors. With chronicle under public audit, consumers can simply assume all the historical volumes are consistently covered by the latest chronile state.

5.3.1 Chronicle Construction

Next, we describe how to construct a chronicle using Merkle tree, and how to efficiently verify the existence of a volume.

Merkle tree is a k-ary tree, where the value of each node is the hash of the concatenation of the value of its children. Similar to hash chains in which the last node fixes all the previous nodes, root node of the Merkle tree fixes all the leaves in the tree. Any change of any leaf leads to the change of the root hash. Figure 5.3a shows a binary Merkle tree with three leaves.

To construct a chronicle using a Merkle tree, we align volumes as leaf nodes in a Merkle tree, adding a new leaf to the tree whenever a new volume is published. This addition leads to change of hash values in all of the ancestors up to the root of the tree. If the tree is full, it can grow one level up to accomodate more leaves (or volumes). For example, the three-level tree in Figure 5.3c grew from the two-level tree in Figure 5.3a and can cover at most 8 leaves.

Whenever a new volume is added, in addition to updating the Merkle tree,

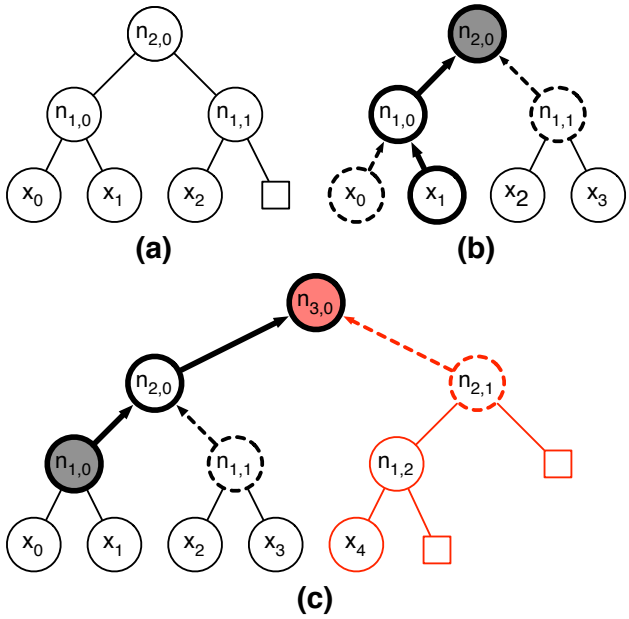

Figure 5.3: Merkle tree examples: (a) a Merkle tree with three leaves; (b) the evidence proof for leaf x_1 in a four-leaf Merkle tree; (c) the consistence proof between a tree with two leaves (x_0 and x_1) and a tree with five leaves (x_1 to x_4).

DeLorean also signs the root hash and publishes it as a separate data packet, described in the next section. The signature of this data packet can be used to transitively authenticate all previously published volumes.

In order to verify existence of a particular volume in the chronicle (e.g., that the volume x'_1 exists), one needs to reconstruct a part of the tree along the path from the corresponding leaf node to the current root node of the tree ($x'_1 \rightarrow n'_{1,0} = \text{hash}(x_0, x'_1) \rightarrow n'_{2,0} = \text{hash}(n'_{1,0}, n_{1,1})$ in Figure 5.3b). The volume x_1 can be proved to exist in the chronicle if the reconstructed value of root node ($n'_{2,0}$) matches ($n_{2,0}$), the one recorded in the tree. Therefore, if we use a k-ary Merkle tree that has n leaves, a single verification only requires $O(\log_k n)$ hash computations in total.

To verify consistency of the Merkle tree evolution, one needs to know the root digest represented some old state and the most recent root digest. For example, to verify consistency between states x_1 and x_4 in Figure 5.3c, one needs

to know past root digest $n_{1,0}$ and the current digest $n_{3,0}$. Similar to the existence verification, one can reconstruct nodes along the path from old to new root: $n'_{2,0} = \text{hash}(\text{n}_{1,0}, \text{n}_{1,1}) \rightarrow \text{n}'_{3,0} = \text{hash}(\text{n}'_{2,0}, \text{n}_{2,1})$. The tree evolution can be declared consistent if all reconstructed values match the one stored in the tree.

5.3.1.1 Proof Publishing

To verify existence of the volume in DeLorean requires knowledge of all sibling nodes along the path to the root (nodes circled by dashed line in Figure 5.3b). In order to allow it, DeLorean publishes each node of the Merkle tree as an individual data packet, including the hash values of all its children (Figure 5.5). Note that with a 1500-byte MTU and SHA-256 hash algorithm being, a single data packet can safely carry 32 SHA-256 hashes (1024 bytes in total), leaving enough space for other fields in the data packet. For that reason, we chose 32-ary Merkle tree to

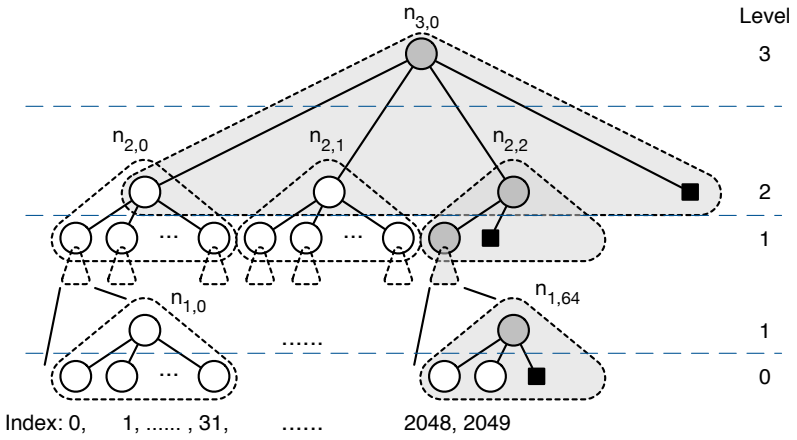

Figure 5.4: 32-ary Merkle tree example

construct DeLorean's chronicle, exemplified in Figure 5.4. In this case, a chronicle with volumes for each 10 minute time interval will require only four levels of the Merkle tree to record 20 years worth of state.

Note that retrieving nodes individually leverages efficient data distribution of NDN: requests from multiple auditors can be efficiently joined or served from in-

network caches. Because nodes at higher layers are involved in more verifications, they are more frequently requested and have a high chance of being universally cached in the network.

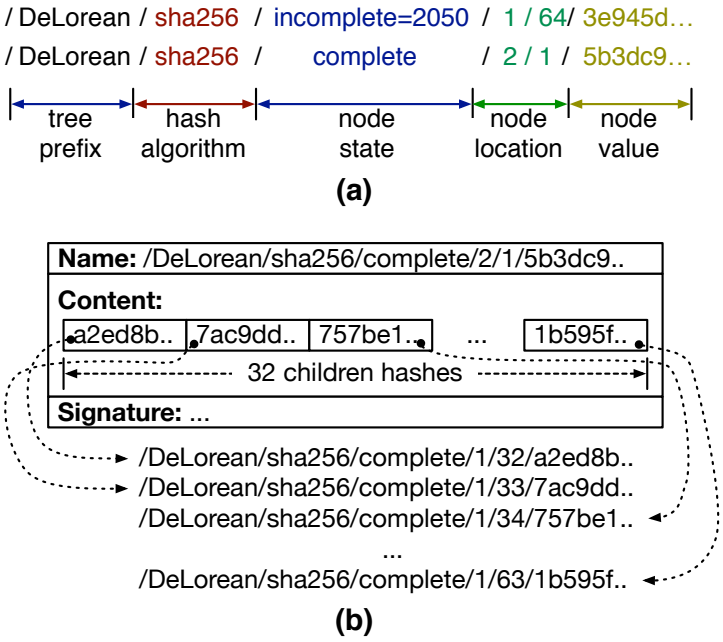

Figure 5.5: (a) Naming convention of 32-ary chronicle tree node; (b) An example of 32-ary chronicle tree node data.

5.3.1.2 Node Naming Convention

For the Merkle tree nodes we defined the naming convention as shown in Figure 5.5a, which consist of five parts. The first two parts specifies the tree prefix and the hash algorithm used to construct the tree.

The third part of the name is a component indicating a state of the node: "complete" when a node has the full set of descendents (e.g., white nodes $n_{1,0}$, ..., $n_{1,63}$, $n_{2,0}$, and $n_{2,1}$ in Figure 5.4), or "incomplete" when one more descendents do not yet exist (gray nodes in Figure 5.4). The hash values of the incomplete nodes are keep changing until all leaves added to the corresponding subtree, after which the node becomes complete with the perpetually fixed hash value. Given a

node can have as many incomplete states as the half of the leaf nodes it covers, to disambiguate the name for different states, we included the sequence number of the next leaf node that can be added to subtree. In Figure 5.4 example, all incomplete nodes would have "`incomplete-2050`" as a node state component (e.g., the first name in Figure 5.5a).

Note that all nodes in all states are represented as immutable data packets, and can be easily replicated in the network. At the same time, only the latest state of the node is needed to perform the volume existence or chronicle consistency verifications: the content of the data packet that represents a new state includes all information that existed in previous data packets. Therefore, as soon as the new state of the node is created, the old state data can be safely removed from the system.

The fourth part of a node name includes two parts: the level of the node and the index of the node at the specified level. Given the total number of nodes in the Merkle tree n, the sequence number s of the leaf node, and the level l ($0 \leq l \leq \lceil \log_{32} n \rceil$) of the intermediate node, its index in the level can be calculated as $i_l = \lfloor s \times 32^{-l} \rfloor$. Using these simple conversions, consumers and auditors can request intermediate nodes for any desired level, e.g., requesting them simultaneously.

The last part is the hash value of the node, which is also the digest of data content. During existence and consistency verifications, consumers and auditors can explicitly request a node using the expected hash digest value, calculated from the pre-verified parts of the tree.

5.3.2 Public Audit

A chronicle, once being detected as inconsistent (i.e., a previous volume being modified), immediately loses its trustworthiness. As we explained before, by build-

ing DeLorean chronicle over Merkle tree, the consistence verification overhead is on the order of $O(\log n)$. With this trivial overhead, a large number of auditors can occasionally and effortlessly check the consistence of the DeLorean chronicle. The collective behavior of the auditors can ensure that at each time slot the consistence of DeLorean chronicle is checked by at least one auditor. Therefore, it is difficult for the chronicle publisher to modify previous volumes without being caught, thus effectively deterring the chronicle publisher from modifying the history. For example, it would be impossible for DeLorean to modify the record for October 22, 2015 issue of USA Today or deny its existence without actually using the time machine and altering the reality. Moreover, since the producers and consumers of data recorded by chronicle rely on it to provide the existence proofs, they have strong motivation to audit the consistence of the chronicle.

5.3.2.1 Consistence Verification

To audit the consistence of the Merkle tree based chronicle, auditors occasionally retrieve the root hash of the tree and check whether it "covers" a root hash that the auditor has retrieved before. Once an auditor verifies the consistence between the two hashes, the auditor can keep the new root hash and discard the old one. Therefore, the auditor's storage overhead is constant.

Similar to existence verification, the consistence verification is to re-compute the new root hash from the old one, other nodes retrieving nodes along the way. With all the nodes of the chronicle tree being published, an auditor can retrieve the nodes that are necessary for consistence verification in at most $O(\log n)$ number of iterations.

Incomplete Node Issue Note that the previously recorded root hash is most cases would be represented as an incomplete node, whose status will change to complete or a different incomplete state. For example, an incomplete node $n_{2,0}$

in Figure 5.3a captures state of volumes x_0, x_1, x_2, while the same node becomes complete when capturing state for volumes $x_0, \ldots, x_3$ in Figure 5.3b. This fact, while complicating the process, does not impact the ability to perform the tree reconstruction. The auditor will need to retrieve the latest state of the node and check that it is a superset of the old state. The exact current state of any node in the Merkle tree is determined by the number of leaves. For a 32-ary tree with the largest volume sequence number s, for a node at the level l,

$$\text{node state is } \begin{cases} \text{“}\texttt{complete}\text{”}, & \text{if } s \geq 32^l \\ \text{“}\texttt{incomplete-(s+1)}\text{”}, & \text{otherwise} \end{cases}$$

Multiple History Issue The only way that the chronicle publisher can modify the history without being detected is to present a different chronicle consistently to the same group of auditors, which is usually called a multiple history issue. The stateful data retrieval and in-network caching of NDN architecture, however, intrinsically eliminate the possibility of creating multiple histories that target different auditors.

Interests that request node data do not reveal any information about the requesters, or in this case auditors. Therefore, it is impossible for DeLorean to craft the auditor-specific responses.

In addition to that, DeLorean will not receive all interests for node data, as they can be aggregated (when multiple auditors request state at the same time) or served from in-network caches. The higher-level nodes of the DeLorean chronicle can be used to verify many different individual states, Therefore, we expect that the data packets that correspond to these states will be universally cached throughout the NDN network, further reducing any possibility of DeLorean to crafting individual responses.

5.3.3 Volume Construction

In the previous description we focused on verification of the state of the chronicle volumes. Consumers, however, would want also to verify the existence of a data packet in the volume, which represent a collection of data packet fingerpints submitted within the corresponding time interval. In the simplest case, the volume is represented as a single data packet which records all submitted fingerprints for the corresponding time period. However, a volume may need to record a large number of data packet fingerprints, exceeding capacity of a single data packet. Therefore, a volume needs to be constructed as a set of data packets, but in a way to minimize overhead for data existence verification.

To accommodate a large number of data fingerprints and yet provide efficient verification, we construct the volumes with the help of Merkle trees. Leaves of the volume-specific Merkle tree represet hashes of data (the recorded fingerprints), and root hash of the tree "fixes" all fingerprints in the volume (Figure 5.6). Recoding the volume tree's root hash as a leaf node of the chronicle's Merkle tree, effectively "fixes" this volume in time.

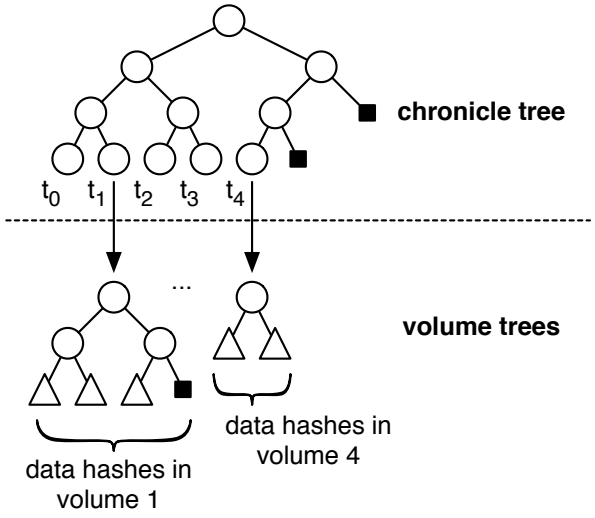

Figure 5.6: Two-level Merkle tree hierarchy of the timestamp service.

Given a volume tree, a consumer can quickly locate a record according to the

local record index. Based on the local index, a consumer can compute a verification path from the record back to the volume tree root and verify the existence of the record in a volume, same way volume existence verification described in Section 5.3.1.

In summary, to assure that a specific data packet existed at a specific time point, the consumer needs to have: (1) volume hash, (2) volume index, and (3) local record index within a data volume. The first two are used to reconstruct the relevant portions of the chronicle tree; and the last one along the fingerprint of the data (obtained from data directly) can reconstruct and verify consistency of the volume tree.

5.3.4 Hash Rollover

For the sake of simplicity, we used only one hash algorithm (SHA-256) in the description above. However, no hash algorithm can be secure forever. Once a hash algorithm is broken (though not very often), all the records in the chronicle are no longer secure.

A proper hash algorithm rollover is the key for DeLorean to prevent hash breaking. More specifically, before the hash algorithm in use is broken,[3] DeLorean can publish another chronicle tree with the same volume sequence but using a stronger hash algorithm. Auditors can verify the new chronicle tree against the existing one to ensure the correctness. Note that hash algorithm breaking happens rarely, the overhead of verifying a new tree though expensive is still affordable.

In some rare case, a hash algorithm may break unexpectedly. In order to survive from the "hash crisis", DeLorean can always publish two sets of chronicle tree, of which each is constructed using a hash algorithms with different crypto strength. Since it is really rare (if not impossible) that the two hash algorithms

[3]In most cases, a hash algorithm does not completely break immediately.

will be broken at the same time, the stronger hash algorithm offers the publisher to find another hash algorithm with more crypto strength and rebuild the chronicle trees.

Note that hash rollover cannot address the broken hash issue completely. Although it can secure the chronicle tree and volume tree, it cannot prevent an attacker to utilize hash collision to modify data content if the hash algorithm of the original data signature is broken. However, given the chance of falsifying a meaning content with the same digest is really rare, we will address this issue in our future work.

5.4 Storage Requirements

Since DeLorean chronicle is a permanent record of archived data and is growing over time, it is important to evaluate the storage requirements. We consider a 20-year chronicle with 10-minute time slots, with both chronicle and volumes represented as 32-ary Merkle trees.

The storage overhead of DeLorean consists of two parts: the chronicle tree and volume trees. A 20-year chronicle involves about 1 million (i.e., $\simeq 32^4$) volumes. Therefore, the chronicle tree has four levels and 32259 intermediate nodes.[4] Assume the size of each node packet is 1500 bytes, the total storage capacity required to save 20 years of chronicle tree is about 48 MB.

If chronicle volumes contain on average 1024 fingerpints of data packets, the corresponding volume trees would have two levels, with the leaf nodes as 32-byte hashes. Therefore, a volume tree involves 33 intermediate nodes (32 nodes at level 2 and one root node). A single volume tree would take about 50 KB, and the total storage requirement for 1 million volumes would be about 50 GB. If we

[4]The leaf nodes of chronicle tree is also the root node of volume tree, so they are included in the volume tree storage calculation.

increase the average capacity of a volume to 32768 (i.e., 32^3) data packets, the storage overhead of a single volume would increase to 1.6 MB, renderring the total storage overhead for all volumes to 1.6 TB. And for volumes with 1 million data packet capacity (i.e., 32^4), the total storage would be only 50 TB, which is still modest compared to the current commercial servers.

5.5 Discussion

5.5.1 Scaling DeLorean Storage

Although our analysis above suggested that DeLorean may potentially record a large amount of data over the time, the storage however will still be overwhelmed if all the produced packets in the world would require timestamp certification. Therefore, there has to be a limit on the number or frequency of the certifications. This limit can be enforced, for example, by business relationships between data producers and DeLorean provider where producers pay for each certification using real money or time working as system auditor (e.g., auditing DeLorean for an hour gives a credit for one timestamp certification).

The limit on number of certifications does not mean that only a limited number of data packets can be timestampped. Using manifest-based aggregation techniques [BDN12, TW16, Moi14] producers can request a single certification for a large collection of data packets. For example, USA Today publisher does not need to request timestamp proofs for every single article published on October 22, 2015, instead it can create a manifest linking all October 22 articles (in a simple list or a tree of manifests) and request proof just for that manifest.

As part of our future work we will investigate how to support safe deletion or compression volume records that are no longer needed [CW09]. This will allow further reduction of the storage overhead and relaxing of the enforced certification

limit.

5.5.2 Recovery from Audit Failures

Whenever the auditors detect that the DeLorean provider is not consistent with the previously recorded states, they need to take actions to remediate the issue. The first course of actions would be to publicly contact the provider and attempt to correct the problem, which highlights necessity of an open channel to the provider through which problems can be reported. This way, the issue can be confirmed by multiple auditors, forcing the provider to immediately address the problem.

In an unlikely case when the DeLorean provider does not respond to the reported issues or no longer wishes to provide the service, it is possible to transition to a new provider. Recall that the authenticity of previous volumes can be implicitly verified through Merkle tree state. The new provider can pick up the service from a state under the consensus of the auditors and obtain copies of the existing volumes that represent the last consistent state. After updating the publisher public key, users of the timestamp service can keep using the same historical volumes.

5.5.3 Resiliency & Multiple DeLorean Providers

The example presented in this dissertation only involves a single instance of DeLorean. However, it is important to avoid a single point of failure, which can be implemented in part using a set of hot backup instances. As soon as there is an issue with a primary DeLorean instance, another timestamp service instance can immediately resume from any mirror. Note that all the data structures (e.g., Merkle trees and hash chains) of DeLorean are publicly audited, thus it is trivial to keep them in-sync among backup instances.

It is also possible to run multiple independent DeLorean instances: it would be

producer's decision on which instance to use. The only change to the described protocol would be in the naming: instead of a single "`/DeLorean`" prefix, the chronicle volumes will be published under "`/google/DeLorean`", "`/apple/DeLorean`", and similar prefixes. However note that consistency guarantee of a single DeLorean instance depends on the quality of the public audit. In case a consumer does not have much confidence about public audit, it can still audit a particular instance by itself at a frequency that satisfies the consumer's own need.

5.5.4 Impact Timestamping on Data Production

Data aggregation and timestampping does not block data production and consumption. It is an additional procedure to ensure data can still be authenticated after the signature expires. In other words, before the original signature on data packets expires, consumers can directly verify the data without needing the timestamp. For example, "Youth Jailed" article of USA Today can be directly authenticated on October 22, 2015 or several days after, until the original signature is still valid. Only when readers access that article year or so later, they may need to use the timestamp proof in order to ensure authenticity of the article at the time it was originally published.

CHAPTER 6

Name-Based Access Control

Data sharing has become increasingly popular in today's Internet applications. For example, people may share their daily activities with their friends; family members may join a private chat group to share chat messages. Sharing private data among multiple parties requires strict access control to prevent authorized parties from seeing the shared content.

Due to a number of reasons (scalability, availability, economy, etc.), today's data sharing applications, by and large, place the shared data on a managed storage and rely on the storage to enforce the data access control (both read and write operations). Such container-oriented access control leaves the data owners no choice but have to completely trust on the data storage. Moreover, this model assumes that an end-to-end secure channel must exist between the data storage and a data consumer. This assumption however imposes stringent requirements on today's content distribution systems, which involves more and more middle boxes to improve the efficiency of data delivery.

One promising solution to secure private data sharing is *data-centric confidentiality*, which secures data directly. More specifically, a data producer encrypts and signs data properly at the time of data production, and distributes the decryption keys and verification keys to authorized data consumers. As a result, data access control is decoupled from the storage and delivery channel.

The data-centric confidentiality model implies that a producer must have acquired the latest access control policy at the time of data production, otherwise

the producer may improperly grant data access. However, when data sharing involves multiple distributed producers and dynamically changed consumers (e.g., automated building management, wellness data sharing), it is difficult (if not impossible) to update the access control policy on all the producers.

To address this issue, we designed Name-based Access Control (NAC), an implementation of data-oriented access control model over Named Data Networking (NDN). NAC leverages NDN's hierarchical naming structure to explicitly express and enforce access privilege. NAC also leverage NDN's data-centric communication primitives to facilitate access control credential verification and distribution. Although the design presented in this dissertation is for use by NDN applications, we believe that the solution can be generally applicable to other applications which use a data-centric communication model, e.g., web and file sharing.

6.1 Access Control Model

To facilitate explanation and discussion in the rest of this paper, we first introduce a simple wellness application example (Figure 6.1). A user Alice uses a wellness application to collect her heart rate data and activity data, which is produced by two sensors respectively. The activity sensor can produce two types of data every minute: the number of steps and the user location, while the pulse sensor only produces Alice's heart rate data.

Alice may share her data with different people at different granularities. For example, Alice may share the daily activity data with her husband Bob and occasionally share the location data with her friend Cathy when she is doing outdoor running exercise. Alice also wants to share her heart rate data with her personal physician David.

In order to facilitate data sharing, Alice may upload all the wellness data to a data storage (e.g., cloud), so that the authorized consumers can retrieve data

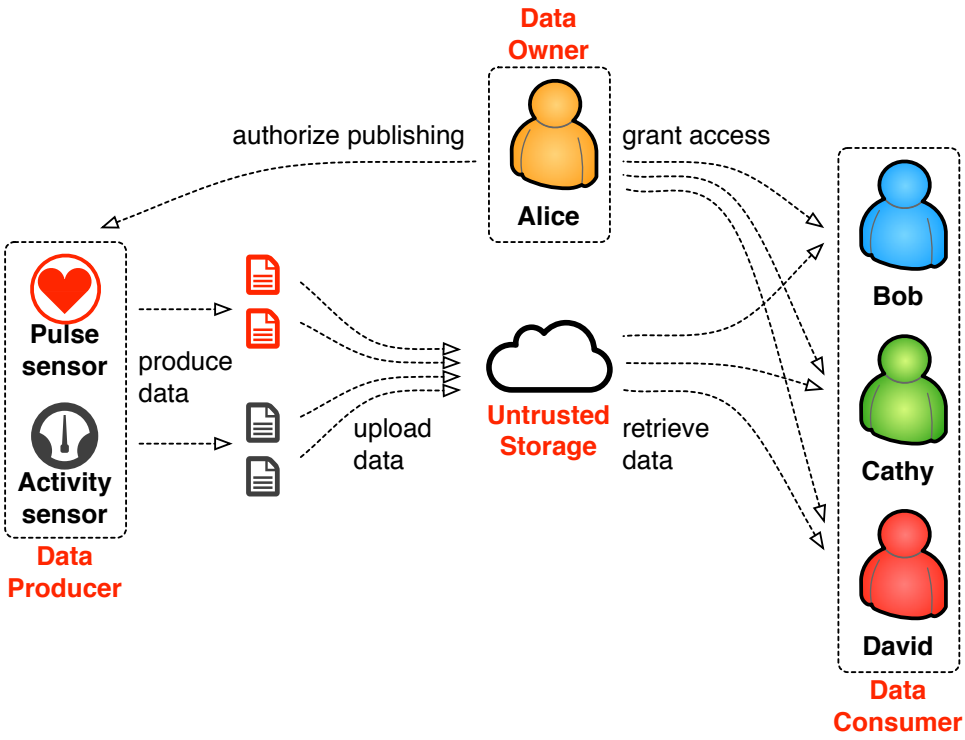

Figure 6.1: Example of health data sharing

at anytime. Alice assumes that the data storage will guarantee the availability of her wellness data, but does not rely on the data storage to enforce access control.

6.1.1 Data-Centric Confidentiality

Next, we will demonstrate how to apply the *data-centric confidentiality* to secure the data sharing in the example above. As shown in Figure 6.2, this model involves three types of entity: *data owner*, *data producer*, and *data consumers*. A data owner (e.g., Alice) can grant a data producer (e.g., activity sensor) the write access by allowing the producer to produce data on behalf of the owner. Unlike traditional container-oriented access control in which the data owner and consumers rely on the data storage to authenticate a data producer, the data owner can issue a public key certificate for an authorized data producer, so that consumers can directly verify whether a data packet is produced by an authorized producer. This certificate is a *production credential* in which the data owner

explicitly specifies the producer's privilege, i.e., the data set that the producer is authorized to produce.

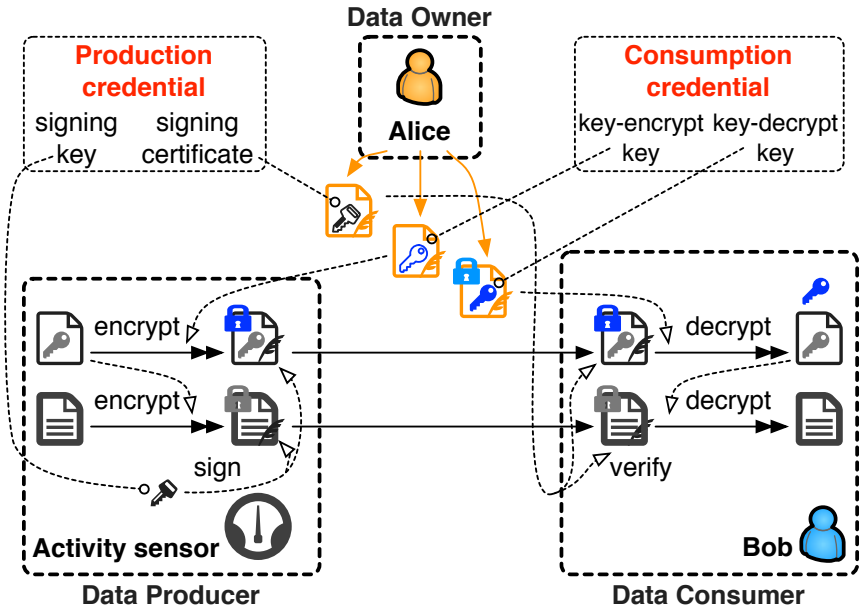

Figure 6.2: Production credential and consumption credential in data-oriented access control.

In order to enforce read access control, a data owner requires a data producer to encrypt the data at the time of production. A data owner can grant a data consumer (e.g., Bob) the read access by distributing the decryption key to the consumer. More specifically, a producer encrypts content using a symmetric key (*content key*), which is generated by the producer. A data owner enforces read access control by controlling the delivery of content keys. A data owner generates a pair of public/private keys, which we call *consumption credential*. As shown in Figure 6.2, all authorized consumers will obtain the private key (key-decrypt key, or KDK), while data producers retrieves the public key (key-encrypt key, or KEK) and use it to encrypt content key. The data owner explicitly specifies the privilege of consumption credential, i.e., the data set that a consumer is authorized to read, so that producers know which content keys should be encrypted using a particular KEK.[1]

[1]Note that public/private key pair is only one of possible implementations for consump-

6.1.2 Design Issues

To achieve practically usable content-based access control, we must address the following design issues.

A data owner must be able to explicitly specify privilege in both production credential and consumption credential. Expressing privilege explicitly in credentials is an important premise to provide fine-grained access control. For example, when Alice can explicitly specify in the activity sensor certificate that the sensor can only produce activity data, data consumers or the data storage can reject any non-activity data produced by the activity sensor. When Alice can explicitly specify the readable data set for each consumption credential, a consumer with the KDK can read data only within the data set, because data producers will not use the corresponding KEK to encrypt content key whose corresponding content is beyond the data set.

A data owner must be able to deliver the credentials to the corresponding entities. None of data owner, data producers, and data consumer will be online all the time. The only always-online entity in the system is the data storage. Similar to normal data, credentials will also be stored in the untrusted storage and delivered through the untrusted network. This implies that producers and consumers must directly authenticate credentials, independent from any retrieval mechanism. Sensitive credentials, such as decryption keys, must be properly encrypted, so that they are only visible to authorized consumers.

A data owner must be able to revoke the access of producer and consumer. A data owner must retain the ability of preventing a producer (or a consumer) at any time from further producing (or reading) data.

In the next section, we demonstrate how to leverage named data and keys, together with *naming conventions*, to solve the above three problems.

tion credential. Other implementation may include identity-based encryption, attribute-based encryption, and etc.

6.2 Naming Access

In this section, we explain how to name production credential (signing/verification keys) and consumption credential (key-decrypt/decrypt keys) to specify different access privileges. We will also show how to distribute keys in a distributed system to achieve the content-based access control and discuss how to revoke the access.

6.2.1 Naming Data

In NDN, data is named under a hierarchical namespace. This allows us to group data with the same property into the same namespace. As an illustrative example, Figure 6.3 shows the naming hierarchy for Alice's health data. Alice can put all her health related data (including keys as we will show later) under a namespace "`/alice/health`". Under this namespace, Alice allocates a sub-namespace "`/alice/health/samples`" for the data produced by sensors. Alice can further sort her health data into two categories: "`activity`" and "`medical`", and give each of them an individual namespace: "`/alice/health/samples/activity`" and "`/alice/health/samples/medical`". The activity namespace covers two types of data: steps ("`alice/health/samples/activity/steps`") and location ("`/alice/health/samples/activity/location`"). Each piece of data is named under the namespace for its own type, with a suffix that can describe additional information of the data. For example, the data under name "`/alice/health/samples/activity/steps/2015/08/27/16/30`" refers to the step data that is produced during 16:30 to 16:31 on August 27th, 2015 (assuming activities are measured in the time unit of one minute).

6.2.2 Naming Production Credential

Since data is organized under the hierarchical naming structure, a data owner can express the privilege of a production credential as the namespace under

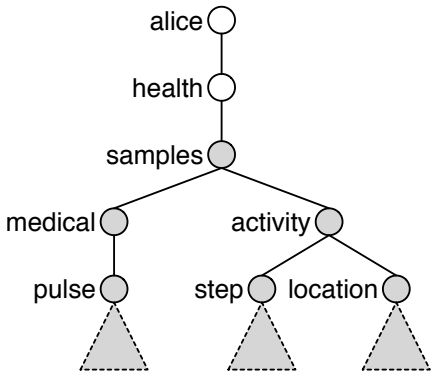

Figure 6.3: An example of naming hierarchy for Alice's health data

which the producer is authorized to produce data. For example, a namespace "`/alice/health/samples/activity`" represents the privilege of producing Alice's activity data, including both step and location.

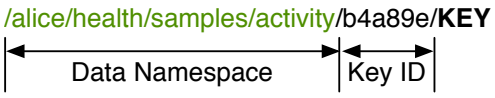

Figure 6.4: The naming convention of signing key

To authorize a data producer to produce data under a given data namespace, a data owner can issue a signing key certificate which associates the producer's signing key with the authorized namespace. Figure 6.4 shows the naming convention of signing key.

Figure 6.5 shows an example trust model for Alice's production credential authentication. This hierarchical trust model has the root key of Alice's own namespace "`/alice`" as the trust anchor. A separate key, "`/alice/health/5fdf51/KEY`", is created to manage the wellness sub-namespace ("`/alice/health`"), which is used to sign the certificate of each authorized data producer (pulse sensor and activity sensor).

Note that this signing key naming convention complies with the NDN certificate format (Section 3.2) and the trust model can be easily described in a trust schema (Section 4). As a result, a data owner can publish the trust schema of

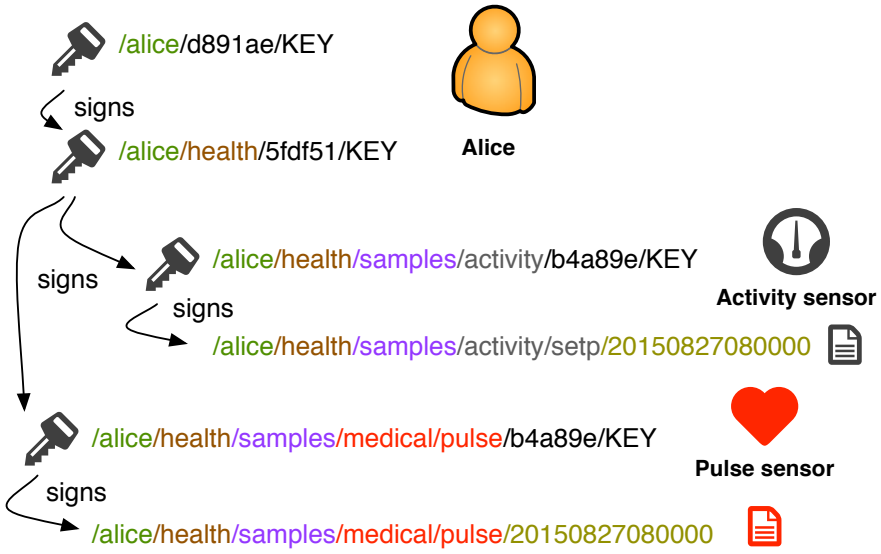

Figure 6.5: Signing hierarchy of Alice's health data

its own namespace, so that consumers can retrieve the trust schema and use it to automatically authenticate data producers.

6.2.3 Naming Consumption Credential

In our design consumption credential is another public key pair (KEK/KDK) for content key encryption. We use well defined naming convention to help a data owner explicitly specify the privilege of a consumption credential. We mentioned earlier that the privilege of a consumption credential is a data set that a consumer with the key-decrypt key (KDK) can access (indirectly through decrypted content keys). With the privilege encoded in the corresponding key-encrypt key (KEK) name, a data producer can tell which content key should or should not be encrypted through the key-encrypt key.

The naming of key-encrypt/decrypt key (KEK/KDK) must accommodate four facts. First, key-encrypt/decrypt keys have different usage than the signing/verification keys introduced above. A signing key is possessed by an authorized producer while a KDK is held by an authorized consumer. The roles of

these two pairs of keys are different, and the key name must explicitly reflect such differences.

Second, consumption credential is created and managed by data owner. The naming convention of consumption credential should prevent other entities (e.g., producers or consumers) from issuing valid consumption credential.

Third, a data owner may want to delegate the consumption credential management to a third party. The consumption credential naming convention should facilitate such management delegation, at the same time, a data owner must be able to restrict the privilege of this third party to consumption credential management only. In other word, the third-party entity should not be able to produce wellness data on behalf of the data owner.

Last, the data set that a consumption credential covers may need additional description that cannot be explicitly encoded as the data namespace. For example, a data owner may want to create a consumption credential that allows consumers to access data produced during certain time period, e.g., from 6pm to 10pm on every workday. Therefore, the consumption credential name must be able to carry additional information to enforce a variety of access restrictions beyond the data naming hierarchy.

Next, we present a naming design that can address the four issues above.

6.2.3.1 Consumption credential namespace

To distinguish consumption credentials apart from production credentials, we allocate a separate namespace for consumption credential, which is parallel to the data namespace as shown in Figure 6.6. Take Alice's health data as an example, Alice can create a namespace "`/alice/health/read`" for consumption credentials. The naming hierarchy of the consumption credential namespace mirrors that of the data namespace, except that data under this hierarchy are all consumption

credentials.

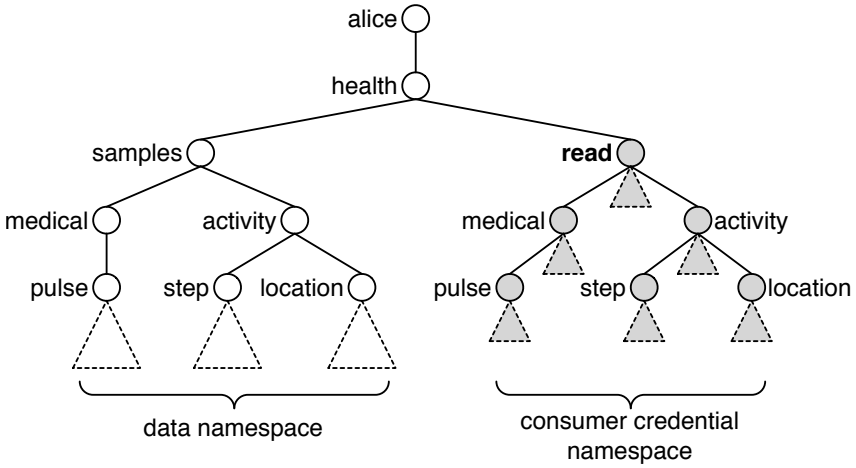

Figure 6.6: An example of consumption credential namespace along with data namespace

With a separate consumption credential namespace, a data owner can delegate the whole or part of the consumption credential management to a third party. The data owner can publish the delegation as a certificate which binds the third party's public key to the delegated consumption credential namespace or sub-namespace. Figure 6.7 shows an example of consumption credential delegation in which Alice delegated her physician to control the read access to her medical data. As restricted by the certificate name, the delegated entity (e.g., Alice's physician) can only issue consumption credential for certain type of data (e.g., medical data), and cannot issue any production credential nor produce any data, including medical data.

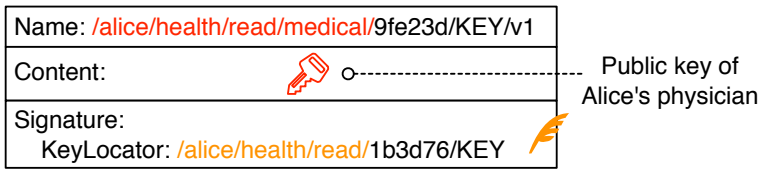

Figure 6.7: An example of consumption credential delegation.

The naming hierarchy under the consumption credential namespace also enables multi-level delegation. For example, Alice's physician can further delegate

a cardiology expert to manage the read access to Alice's heart rate data.

6.2.3.2 Consumption credential name convention

Under the consumption credential namespace, a data owner can name a consumption credential at any level of the naming hierarchy (e.g., "`/alice/health/read/medical`", and "`/alice/health/read/medical/pulse`") with the meaning that consumers with the credential can only access data under the corresponding data namespace. We mentioned earlier that our design uses a public key pair (KEK/KDK) to construct a consumption credential. Both keys need to be named properly to convey the privilege of a consumption credential.

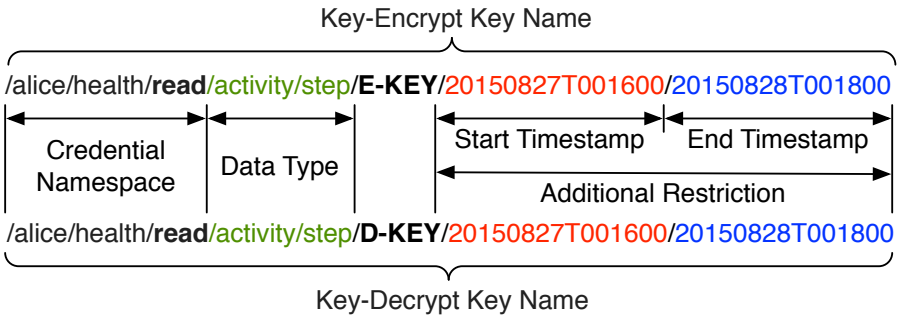

Figure 6.8: The key naming convention of consumption credential

Figure 6.8 shows the naming convention for the keys of a consumption credential. Both key-encrypt key (KEK) and key-decrypt key (KDK) share the same naming structure. They all start with a particular prefix under the consumption credential naming hierarchy. After the prefix, each type of keys has a key-tag component that distinguishes the usage of these keys: "`E-KEY`" for key-encrypt key and "`D-KEY`" for key-decrypt key. After the key-tag, a data owner can append other additional restrictions that have not been explicitly encoded in the data namespace. For example, the key names in Figure 6.8 says that a consumer with this corresponding credential can access Alice's step data produced between 4pm to 6pm on August 27, 2015.

6.2.4 Credential Delivery

For producer credential, we assume that a producer creates and retains signing key and data owner issues signing key certificate using conventional certificate issuing mechanism. Only consumers need to retrieve signing key certificates for data authentication. Since a signing key certificate is an NDN data packet, data owner can simply upload the issued certificate to a data storage. Potential data consumer can follow "`KeyLocator`" in data packet to retrieve the certificate later.

The key-encrypt/decrypt key of a consumption credential, however, are created by data owner and should be delivered to related data producers and authorized consumers respectively. We will explain how to deliver these keys to related entities.

6.2.4.1 Key-encrypt key delivery

As mentioned earlier, the key-encrypt key (KEK) in this design is a public key. A data owner can name a KEK as we mentioned above (Figure 6.8), and publish the key as a data packet. Since each encryption key has the data owner's signature, they can be safely uploaded to the data storage, retrieved and verified by data producers and consumers. As long as a data producer knows the key-encrypt key naming convention, it can infer the name of the key-encrypt key to retrieve. The naming convention can be tailored for specific applications to facilitate key retrieval.

Let's consider the wellness application as an example. In this application, producers (e.g., sensors) produce wellness data continuously. With the naming convention defined in Figure 6.8, a data owner can specialize the "additional restriction" as a time interval, and create a sequence of KEKs. The time interval of these KEKs can be concatenated together to cover a continuous time period as shown in Figure 6.9. Note that this naming convention implies that the ending

timestamp of a KEK is the starting timestamp of the next KEK.

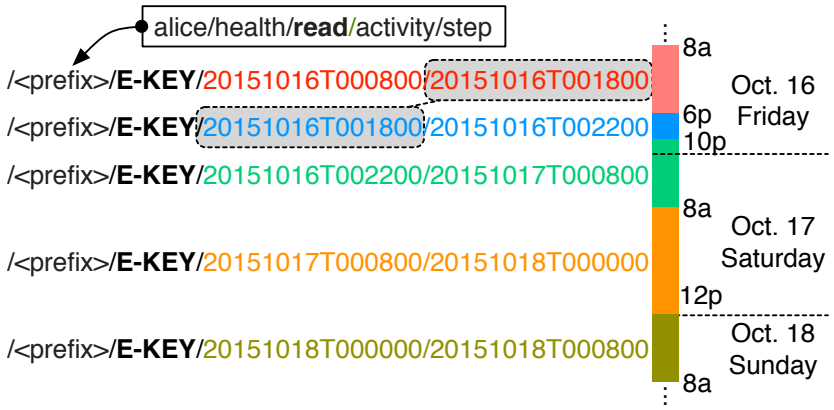

Figure 6.9: A sequence of key-encrypt keys cover a continuous time period.

To construct an interest to retrieve a KEK, a data producer must first determine the credential prefix. Note that there could be multiple prefixes which a data producer can infer from the name of produced data. For example, given Alice's step data "`/alice/health/samples/activity/step`", the activity sensor can derive the most specific credential prefix "`/alice/health/read/activity/step`" corresponding to the step data namespace. Since every parent credential prefix of the most specific prefix also covers the step data, the data producer can determine all the possible credential prefix by tracing back to the root of the credential namespace (e.g., "`/alice/health/read`"). In the example above, the activity sensor can deterministically derive three prefixes: "`/alice/health/read`", "`/alice/health/read/activity`", and "`/alice/health/read/activity/step`".

For each derived credential prefix, a data producer needs to determine the starting timestamp for the KEK to retrieve. We mentioned earlier that the ending timestamp of a KEK is the starting timestamp of the next KEK. When a data producer has already obtained a KEK, it can construct an interest for the next KEK by specifying the starting timestamp using the ending timestamp of the obtained key. Routers and data storage can apply the longest prefix match to pick the next KEK and satisfy the interest.

If a data producer has not received any KEK before, the data producer can express an interest with the credential prefix (e.g., "`/alice/health/read/activity/E-KEY`"). The interest can bring back a KEK which can serve as a starting point for the KEK enumeration as described above.

When a retrieved KEK's ending timestamp is much earlier than current timestamp, KEK enumeration may become inefficient. In this case, a data producer can use "`Selectors`" to speed up the key enumeration process. More specifically, a data producer may use "`Exclude`" filter to exclude any KEK whose starting timestamp is earlier than the latest one among all the received KEKs.[2] A data producer may also specify "`ChildSelector`" to select the "rightmost" KEK under the credential prefix.

6.2.4.2 Key-decrypt key delivery

Key-decrypt key (KDK) should be visible only to authorized consumers. Note that a data owner may not be online all the time, it would be desirable for the data owner to leave the KDKs in a data storage for authorized consumers to retrieve it whenever needed. Since the data storage is untrusted, a data owner can encrypt a KDK using the public key of each authorized consumers. Each encrypted copy makes an individual data packet.

Figure 6.10 shows the format of encrypted data. The data content consists of two components: "`EncryptionAlgorithm`" which the meta-information about the encryption scheme and "`EncryptedContent`" which contains the cipher text of content. Note that the format is general enough to carry any content which is not restricted to KDKs but also include content key and normal content.

Since each consumer has its own encrypted copy of KDK, each copy must have a unique name. To distinguish different copies, we define the naming con-

[2]In case clock is not synchronized, one may also set "`Exclude`" filter to exclude any KEK whose starting timestamp is later than current timestamp.

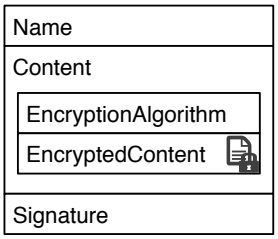

Figure 6.10: Data packets carrying encrypted data and keys

vention for encrypted data as shown in Figure 6.11. For each encrypted copy, we append a special name component "FOR" and the encrypting key name after the content name. For example, a decryption key for Alice's activity data that is encrypted using Bob's public key is named as "/alice/health/read /activity/D-KEY/20151016T000800/20151016T001800/FOR/bob/health/access /E-KEY".

/<ContentName>/**FOR**/<EncryptionKeyName>

Figure 6.11: Naming convention of encrypted data

The name of encrypting key in each data packet can help a data consumer to construct a decryption chain to access the original content as shown in Figure 6.12. When a consumer retrieves an encrypted data packet, it can extract the content key name from the data name. We assume that an authorized consumer should know its authorized credential prefix[3]. With the content key name, a consumer can construct an interest by appending the consumption credential prefix to the content key name. With longest prefix match, routers and data storage can satisfy the interest with the encrypted content key. After receiving encrypted content key, the consumer can extract the key-encrypt key (KEK) name and construct an interest for the corresponding key-decrypt key (KDK) by appending the consumer's own name to the KDK name. In the end, the consumer can retrieve the encrypted KDK and recursively decrypt all the intermediate keys and the

[3]An authorized consumer does not have to know the complete credential name, i.e., the full name of each decryption key.

original content.

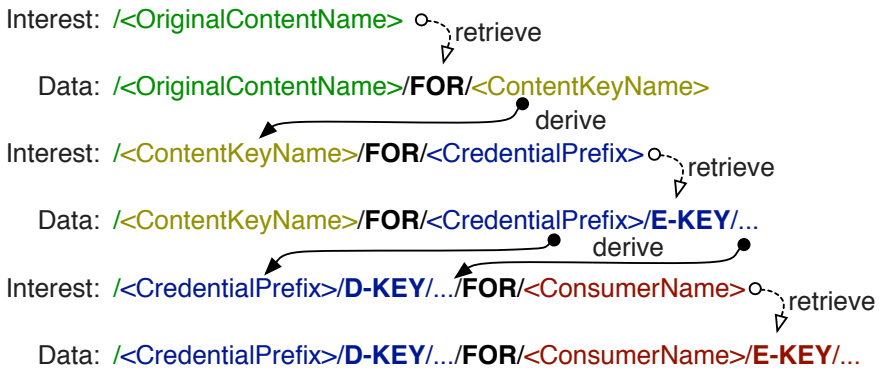

Figure 6.12: A chain of keys to decrypt wellness data

6.2.5 Fine-Grained Access Control

With consumption credential, a data owner can control the read access to content from two dimensions: specifying the privilege of individual consumption credential and restricting the set of credentials that a consumer can obtain. For example, Alice may want to share her location information with her husband Bob all the time, but with her colleague Cathy only during working hours. In this case, Alice can specify a sequence of consumption credentials which cover her location data for 9am-5pm and 5pm-9am every day. Alice can encrypt the KDKs for 9am-5pm from Monday to Friday for both Bob and Cathy, and encrypt all the other decryption keys for Bob only, as shown in Figure 6.13.

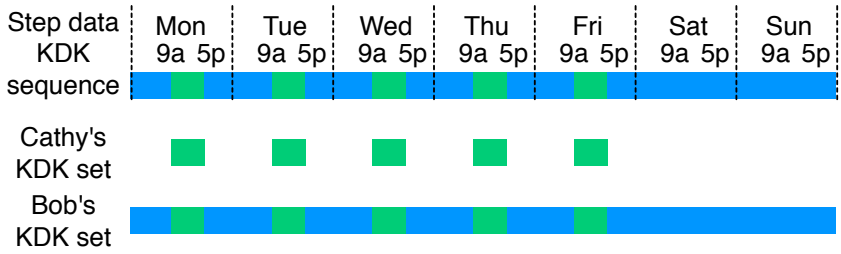

Figure 6.13: Different read privilege in terms of KDK set.

In fact, a data owner can divide the data set arbitrarily into multiple consumption credentials (KEK/KDKs), so that the data owner can create different

combination of KDKs to represent different privilege of individual consumers. When the read privilege can be pre-defined, consumption credentials can be automatically created and published in the network.

6.2.5.1 After-Fact Access Granting

The model we discussed so far focus on controlling the access to data as they are being produced. The name-based access control also allows a data owner to grant a new consumer the access to the data that is produced long time ago. When the granted access is covered by one or more KDKs that were generated earlier, the data owner can simply encrypt KDKs directly using the new consumer's public key.

Note that a data owner can always create a top-level consumption credential (e.g., "`/alice/health/read`") and retain the KDKs to itself only. Since every producer will publish a copy of content key encrypted using the KEK of the top-level credential, the data owner can obtain all the content keys. As a result, when the granted access cannot be expressed as a combination of existing KDKs, the data owner can re-encrypt the granted content key directly for the new consumer.

6.2.6 Access Revocation

With content-based access control, revoking write access is equivalent to revoking the producer's public key certificate, so that neither data storage nor end consumers will accept data of the revoked producer. A data owner can also easily prevent a previously authorized consumer from reading any new data by stopping publishing consumption credentials for the consumer. However, revoking data access that has been granted requires strict control on the availability of KDKs, i.e., preventing a revoked consumer from accessing these KDKs. For example, a data owner may delete from a data storage the KDKs encrypted for a revoked

consumer.

A more effective solution can be provided at the application layer. For example, to control the access to video with copy right, a video provider (e.g., Netflix, Hulu) may ship its own video player as a blackbox to user. The video player not only decrypts the video stream, but also can prevents users from retaining a copy of the decrypted video. We assume the same techniques can be applied here to control the access to the KDKs. More specifically, the blackbox can negotiate an ephemeral keys for KDK distribution and throw the ephemeral keys and KDKs away after data decryption. For a revoked consumer, the blackbox will fail to obtain an ephemeral key, thus preventing the consumer from accessing content.

6.3 Evaluation

We perform a comparative evaluation on NAC. We first compare NAC with CCN-AC [KWU15], another encryption-based access control scheme using regular public key cryptography, about the overhead of crypto computation and key retrieval. We examine the difference between NAC and Attribute-Based Encryption [GPS06], another encryption scheme which is often mentioned as an easy-to-use solution for encryption-based access control.

6.3.1 NAC vs. CCN Access Control

CCN-AC [KWU15] is an encryption-based access control scheme where each data producer has complete knowledge about the access control policy, i.e., who are the authorized consumers and what the consumers are allowed to access. We first consider the total number of encryption/decryption operations that both NAC and CCN-AC must perform to distribute content keys. We compare the two schemes under two scenarios.

The first scenario includes m producers under the same namespace and n

consumers that are authorized to access data under the namespace for one day. For both schemes, we assume that each producer will generate per-hour content key to encrypt data it produces within each hour.

In CCN-AC, each producer needs to explicitly encrypt the content key for each authorized consumer. Therefore, the total number of encryption operations in one day can be calculated as:

$$N_{ccn-ac} = 24mn \tag{6.1}$$

For NAC, a data owner can create a consumption credential for all the data produced in one day and encrypt the credential KDK for each authorized consumer. Each producer only needs to encrypt each content key with the credential KEK.

$$N_{nac} = 24m + n \tag{6.2}$$

Clearly, the number of encryption operations that NAC has to perform is much less than CCN-AC, especially in cases of a large number of producers or consumers.

In the second scenario, we also consider m producers but 24 consumer groups, and each group has n consumers. Each group of consumers can access data produced in a particular hour of a given day. In CCN-AC, each producer will encrypt a content key for all the authorized consumers:

$$N_{ccn-ac} = 24m + 24mn \tag{6.3}$$

For NAC, a data owner needs to create 24 credentials. A data owner will encrypt each KDK for authorized consumers, while each data producer encrypts the content key using the corresponding credential KEKs:

$$N_{nac} = 24m + 24n \tag{6.4}$$

The result suggests that NAC scales better than CCN-AC when there is more than

one producers in the system. Note that the scalability comes from the one-level indirection of data owner, which aggregates multiple consumers into a group, so that data producers only need to be aware of the group's KEK rather than the key of each group member.

Our second evaluation metric is the overhead in key retrieval. CCN-AC puts all encrypted content keys into a single data blob, called *manifest*. A consumer must retrieve the whole manifest to extract the content key encrypted for it. When the size of the data blob is larger than the network's maximum transmission unit (MTU), the data blob must be segmented. Assume that MTU is 1500 bytes and that consumers use RSA keys, one data segment can carry about 4 encrypted content keys. Assuming that a content key is granted to n consumers, the average number of data packets that a consumer must retrieve is:

$$N_p = \lceil \lceil n/4 \rceil \cdot 2 \rceil \tag{6.5}$$

This analysis suggests that a CCN-AC consumer only needs to retrieve one key packet when there are fewer than 4 authorized consumers for a content key, but will need to retrieve multiple key packets when the number of authorized consumers increases, and the number increases linearly with the number of authorized consumers. In contrast, NAC requires a consumer to retrieve a content key and a key-decrypt key, of which each is carried by a separate data packet, therefore only two data packets are needed to retrieve these two keys.

6.3.2 NAC vs. Attribute-Based Encryption

Before we make a brief comparison between NAC and Attribute-Based Encryption (ABE) [GPS06], it is helpful to understand the working mechanism of ABE (Figure 6.14(b)). Unlike traditional encryption techniques, ABE encrypts data using a set of predefined, descriptive attributes instead of crypto keys, eliminat-

ing the need for data producers to fetch encryption keys. To enable such a scheme, ABE requires a key authority that knows the attributes of all the receivers and can generate a *master key* and its corresponding *public params*. An ABE receiver (data consumer) must obtain its private key from this key authority. The key authority derive a user's private key from the master key together with the user's attributes. Users with the identical attribute set will obtain the same private key. An ABE sender (data producer) generates ciphertext using the public params together a set of attributes. A receiver can decrypt a ciphertext only if the receiver's attributes match the attributes with which the ciphertext is generated.

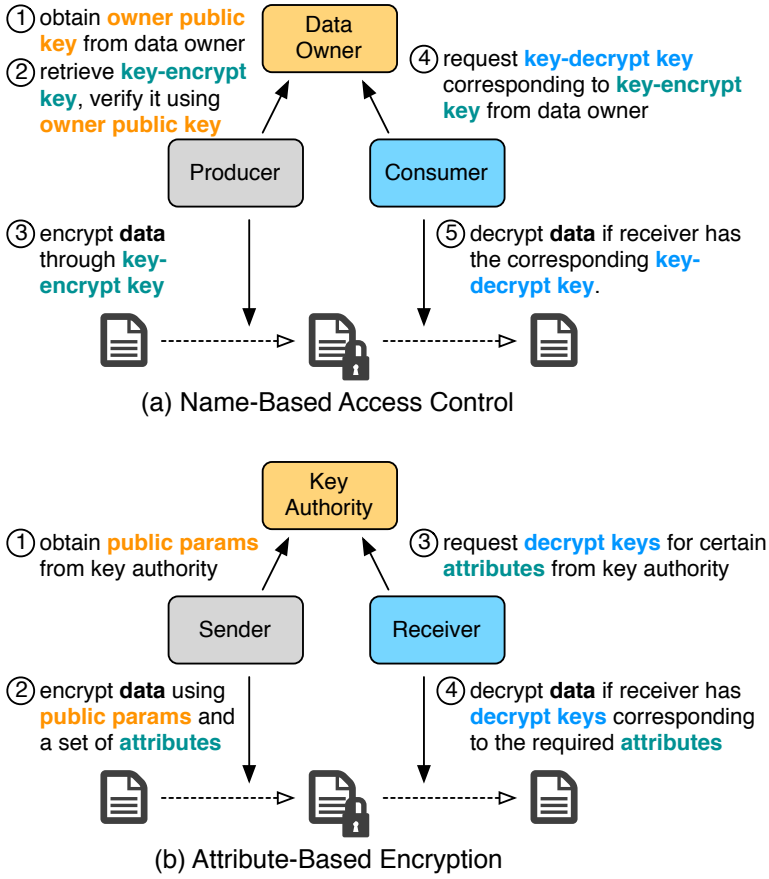

Figure 6.14: Comparison between attribute-based encryption and name-based access control

Figure 6.14 shows the working flow of both NAC and ABE. Next, we will compare ABE based access control and NAC on four aspects: setup, encryption,

decryption, and revocation.

6.3.2.1 Setup

Both NAC and ABE requires an authority (the data owner or key authority) that determine how encryption and decryption keys match up. In NAC, the encryption and decryption keys are generated by the data owner. The data owner names the encryption/decryption keys by their encryption scope and signs the keys, both producers and consumers must learn the *owner public key* at the setup phase, so that they can authenticate the encryption/decryption keys and use them properly.

In ABE, an encryption key can be constructed by any sender, and the corresponding decryption key is derived by the key authority. In order to pair up encryption and decryption keys, a sender must learn the *public parameters* of the key authority at the setup phase, and use them to generate encryption key correctly. Receivers must learn certain information (e.g., key authority's public key) so that they can securely request decryption keys from the key authority.

The ABE scheme, however, also requires each sender to be configured with the knowledge about all the supported attributes, i.e., which attributes should be used to encrypt which data. In contrast, NAC does not require such configuration, but relies on the encryption naming convention to determine which keys should be used to encrypt a particular piece of data.

6.3.2.2 Encryption

NAC and ABE differ most significantly in encryption. NAC requires producers to periodically retrieve KEKs from data owners, thus introducing the overhead of key retrieval. While failure in retrieving a KEK does not block data production, it may prevent consumers from accessing the data. In contrast, ABE waives the need of key retrieval. A sender can directly encrypt data with a set of descriptive

attributes. In order to control the time granularity of encryption, a sender may add time as one of encrypting attributes.

The computational overhead of two schemes are determined by different factors. In NAC, a producer encrypt data using a content key, which in turn is encrypted by one or more key-encrypt keys. In the worst cases where there is a key-encrypt key at each level of the key hierarchy, the computational overhead is proportional to the depth of the key hierarchy. However, in ABE, a producer may need to process each attribute used in encryption, the computational overhead will linearly increase with the number of attributes used in the encryption.

6.3.2.3 Decryption

Both NAC and ABE relies on the authority (data owner or key authority) to generate decryption keys and distribute the keys to corresponding consumers if they have authenticated themselves to the authority. In both scheme, the generated decryption keys must be securely delivered to the consumers.

NAC creates a decryption key hierarchy and assign consumers into the different levels in the hierarchy according to the consumer's privileges. ABE expresses a consumer's privilege in terms of the consumer's attributes and crafts decryption keys according to the consumer's attributes. While ABE waives the need of maintaining the decryption key hierarchy, the process of key generating and decryption is non-trivial, and may introduce significant computation overhead.

6.3.2.4 Revocation

Both NAC and ABE handle revocation through restricting the livetime of encryption keys. In both schemes, consumers have to periodically "renew" the corresponding decryption keys. In NAC, data owner explicitly specifies the lifetime of KEKs in the key name. As a result, the data owner controls the temporal

granularity of keys. In contrast, senders in ABE specify the validity period of the encryption key as an attribute involved in the encryption. When the validity period of a receiver's decryption key does not satisfy the required attributes, the receiver must request a new decryption key with appropriate validity period from the key authority. Therefore, the lifetime of encryption keys is usually determined by the senders in ABE.

6.4 Discussion

6.4.1 Consumption Credential Implementation

Besides public/private key, there are several other design options of implementing key-encrypt/decrypt key in consumption credential. The first one is symmetric key. In this case, a data owner not only needs to encrypt the symmetric key for each authorized consumer, but also needs to encrypt the symmetric key for each related producers. Therefore, this design introduces more encryption overhead.

Another option is attribute-based encryption. In this case, a data owner does not need to publish key-encrypt key for data producers. Instead, a data producer can encrypt content key with well-known attributes. This option requires delicate design of attribute set. We plan to compare the complexity of attribute-based encryption against NAC, and investigate the feasibility of implementing consumer credential using attribute-based encryption.

6.4.2 Emergent Revocation

A data owner needs to pre-specify the effective time interval of each consumption credential, but it is possible that the data owner may want to revoke a consumer's access before the end of the time interval. A data owner can publish a new consumption credential with a new starting timestamp as an indication of

revoking the previous KEK/KDK, so that producers will use the new KEK/KDK to encrypt content key. However, this solution requires producers to pro-actively retrieve KEK all the time. Another solution is to specify short-lived consumption credentials. This solution alleviates the burden of data producers, but it requires data owners to publish credential more frequently.

6.4.3 Forward Secrecy

Forward secrecy requires past communication to be free from compromise of a long lived key. Since our design directly encrypt KDKs using a consumer's public key, compromise of a consumer's private key may allow an attacker to access all the data that the consumer has accessed before.

A possible solution is to encrypt KDKs using an eph-emeral key which is negotiated through a plain-text key exchange protocol, such as Diffie-Hellman key exchange [DH76]. Since the ephemeral key will be thrown away once the KDK is decrypted, compromise of a consumer's private key cannot help an attacker to recover ephemeral keys.

An online key distribution service, however, must exist to run the key exchange protocol and negotiate eph-emeral keys with consumers. In this paper, we assume the only always-online entity is an untrusted data storage. For applications or systems that requires forward secrecy, we may relax the restriction on the online entity and enable key exchange service on it.

6.4.4 Content-Based v.s. Perimeter-Based Access Control

Both models have their own advantages and disadvantages. Content-based access model eliminates the trust over data storage and middle boxes. However, it requires additional mechanism to control the content availability. As a results, revocation in content-based access control cannot prevent a consumer from read-

ing data whose read access was granted to the consumer previously, as long as both the data and keys are available. In contrast, perimeter-based access model directly controls data availability, but it requires the enforcement of access control policy in every device on the perimeter.

6.4.5 Key-Encrypt Key Distribution

In Section 6.2, we explained how to leverage naming conventions to facilitate the key-encrypt key distribution when the additional restriction has only one dimension (e.g., time). However, when there are more than one dimension of additional restriction (e.g., geo-fencing), data producers need a name-independent mechanism to retrieve key-encrypt keys (KEKs).

We consider data synchronization as a promising approach to distribute KEKs with various additional requirement. For example, a data owner can create several data sets (one for a particular consumption credential prefix) and publish new KEKs in the corresponding data set. Producers can synchronize the KEK set that is related to their interest, so that they can get notification at first time when a new KEK is published.

6.4.6 Automated Consumer Authorization

In this paper, we assume that data owner manually authorizes each consumer. It is also possible to automate the consumer authorization. For example, a data owner may accept read requests from consumers. After a consumer's key is authenticated, the consumer's public key can be automatically used to deliver the key-decrypt key (KDKs) for the authorized data set. Different data owner may apply different trust model to authenticate consumers. Trust schema [YAC15] can help data owner to customize their trust model and automate consumer authorization.

6.4.7 User Key Management

The NAC design requires participants to have their own public/private key pairs. How to educate users to correctly manage their keys remains a challenging security problem. In NDN, we assume key management has become a normal practice for users.

CHAPTER 7

Related work

7.1 Trust Management

The focus of this paper is trust management automation. We are aware of similar efforts for Public Key Infrastructure (PKI), including a standardized path validation algorithm for X.509 certificate authentication [CSF08], certificate chain discovery methods for SPKI certificate system [CEE01, LWM01], and general chain discovery mechanisms [BGR07]. However, these studies assume a specific trust model. Automation based on trust schemas is a general trust management solution for NDN applications which can have different trust models. Moreover, it not only allows automation of authentication process, but also enables (at least partial) automation of the data signing process.

The design of trust schema leverages NDN naming to enforce name-based trust policies for data packets. DNSSEC [AAL05], a security extension of DNS, adopts a similar mechanism to authenticate DNS resource records: a key bound to a DNS domain name is globally trusted to sign only DNS resource records under this domain. DANE [HS12] extends the name-based mechanism of DNSSEC to authenticate a TLS public keys. At the same time, both DNSSEC and DANE assume a specific hierarchical trust model, while our trust schema can capture many different trust models that NDN applications may need.

The trust schema is basically a policy language, where rules define policies on which keys are trusted to authenticate data. Compared to previous work on pol-

icy languages for access control and authorization, such as PolicyMaker [BFL96], SD3 [Jim01], RT [LMW02], and Cassandra [BS04], our work focuses on data authentication and integrates data authentication into the NDN network architecture.

7.2 Data-Centric Confidentiality

There are several existing works on enforcing access control over information-centric network. Ghali [GST15] proposed an interest-based access control solution. However, the solution requires every router in the network to enforce the data producer's access control policy. Our work made a weaker assumption that network is not trustworthy, and we aimed at minimize dependency on intermediate devices to enforce access control.

Kurihara [KWU15] proposed an encryption-based access control framework. The framework enforce access control by encrypting content directly. However, the framework assumes that each producer has full knowledge about the access control policy. In contrast, our work consider a more general scenario in which multiple producers may collectively produce content under the same namespace. In this scenario, it may be infeasible to notify each producer of any change in the access control policy, e.g., adding a new consumer or removing an existing one. Our work introduces one-level indirection by explicitly dividing the information about access control policy into two parts, thus having better scalability.

Misra [MTM13] proposed another encryption-based access control scheme which used broadcast encryption to achieve large scale content delivery.

7.3 Archive Authenticity

To the best of our knowledge, Haber and Stornetta [HS90] were the first to propose the use of the timestamp service to secure digital documents. They built the service by linking documents in a time order using a crypto hash function, allowing users to check the existence of a document by checking against a set of documents along the timeline. Buldasi et al. [BLL98] later proposed a binary linking timestamp that simplified implementation of the timestamp service. Additional information and history of the timestamp service designs is available in the survey by Vigil et al. [VBC15].

The timestamp service work that is most related to the DeLorean design is KASTS [MB02]. KASTS not only timestamps signed documents, but also keeps a secure storage of verification keys. Compared to KASTS that builds the timestamp service over hash chains, DeLorean uses Merkle tree hierarchy to allow efficient public auditing. Moreover, KASTS is focused on a single trust model, i.e., the PKI model, while DeLorean supports data signing under arbitrary trust models, as long as consumers know the corresponding trust schema.

The foundation of DeLorean design is a work of Crosby and Wallach [CW09] that proposes the use of Merkle tree to implement a tamper-evident logging system. They conducted the detailed performance analysis to prove efficiency of the Merkle tree based logging system. They also proposed a scheme to safely delete log entries that are no longer needed, which we plan to investigate in the next revisions of the DeLorean design.

Certificate Transparency (CT) [Lau14, CSP15] offers one of the most important use cases of Merkle tree based logging system and inspired the design of DeLorean. CT is designed to mitigate the certificate mis-issuance problem through a "security through publicity" approach. CT uses Merkle tree to build a public board, on which certificate authorities are required to post all the issued certifi-

cates. Using this board, the legitimate owners of the domain names can easily detect the mis-issued certificates. DeLorean borrows the same "security through publicity" concept, but applies it to verification of absolute time of the chronicle volumes: any attempt to "back-publish" or modify volume will be detected by a set of public auditors. Because of the nature of IP protocol, CT instance will always know the source address of the requester. To avoid problem of multiple consistent views to different users, CT design includes an additional gossip protocol [CSP15]. DeLorean intrinsically avoids this problem by being an NDN-based system: data retrieval in NDN does rely on source addresses, but uses states set up by the incoming requests.

BitCoin [Nak08] represents another example of "security through publicity" approach to support a consistent append-only log based on hash chain. However, BitCoin requires an efficient peer-to-peer overlay multicast network and also requires each peer in the system to keep a copy of the history, thus making it unsuitable for maintenance of a large amount of long-lived data.

CHAPTER 8

Conclusion

Usability is a fundamental requirement for any security solution. The Named Data Networking (NDN) design mandates that each network-layer data packet carries a digital signature for authentication. Although this requirement on the packet format represents a significant first step toward securing networking system within the a data-centric security model, its actual effectiveness depends on the implementation. At the same time, the other piece of data-centric security model, data-centric confidentiality is still missing in current architecture.

Our observations during the first few years of NDN application development suggest that it is a non-trivial task for developers to properly define trust relationships between data and keys, to handle proper key chain construction, and to enforce authentication of data according to the defined rules. We also noticed that, due to lack of efficient data encryption and decryption key distribution mechanism, developers have to handle data encryption and decryption by themselves. It happens too often that developers use shortcuts to get around security (e.g., using hard-coded keys, or simply turning verification off "temporarily" when it blocks development progress, or simply deliver data packet in plain text).

In response to the above important and urgent issue, we invented the idea of a trust schema to formally define application trust models, and to automate the signing and verification processes. We developed prototypes of two trust schema interpreters that can convert trust schemas into finite state machines and help applications to rigorously sign and authenticate data automatically. We applied

our prototypes to secure a range of NDN applications, and our experience so far gives us confidence in the solutions general applicability to most, if not all, NDN applications.

We also introduce post-factum validation model to address the authenticity problem caused by the mismatch between data lifetime and signature lifetime. By designing SigLogger, we free application developers and data producers from periodically re-signing data. Besides reducing the overhead of long-lived data maintenance, SigLogger also decouples the lifetime of data and signature, thus encouraging the use of short-lived keys and significantly reducing the chance of key revocation.

We also designed the name-based access control (NAC) as the first data-centric confidentiality solution in NDN. We developed the NAC prototype that can enforce automated data encryption at fine granularity in a scalable way. We applied our prototype to control the data access in two network environments (health data sharing and building management system), our experience so far gives us confidence in the efficiency of solution in large scale distributed data production systems with dynamically updated access control policy.

We believe we have contributed a meaningful step toward a reusable approach to data-centric security. We plan to apply the schematized trust management in more NDN applications and integrate the schematized trust management with operating system support. We would also like to see interested parties, especially people in security research community, to identify and define other commonly reusable trust schemas ("security design patterns") for popular network applications, to be used to secure more applications across the Internet.

We plan to extend SigLogger into more distributed logging system to increase the robustness of the system. We would also like to extend the post-factum validation into more general usage beyond the Internet communication.

We also plan to extend NAC into a full-fledged confidentiality solution by integrating it with existing name confidentiality solutions [DGT12], and further integrate trust schema and SigLogger to provide an NDN security layer with fully supported data-centric security.

References

[AAL05] Roy Arends, Rob Austein, Matt Larson, Dan Massey, and Scott Rose. "DNS security introduction and requirements." RFC 4033, 2005.

[Afa13] Alexander Afanasyev. *Addressing Operational Challenges in Named Data Networking Through NDNS Distributed Database*. PhD thesis, UCLA, 2013.

[AYW15] Alexander Afanasyev, Cheng Yi, Lan Wang, Beichuan Zhang, and Lixia Zhang. "SNAMP: Secure Namespace Mapping to Scale NDN Forwarding." In *Proc. of Global Internet Symposium*, 2015.

[BBB15] Elaine Barker, William Barker, William Burr, William Polk, and Miles Smid. "Recommendation for Key Management." NIST Special Publication 800-57, July 2015.

[BBD01] Mihir Bellare, Alexandra Boldyreva, Anand Desai, and David Pointcheval. "Key-privacy in public-key encryption." In *Advances in Cryptology—ASIACRYPT 2001*, 2001.

[BDN12] Mark Baugher, Bruce Davie, Ashok Narayanan, and Dave Oran. "Self-verifying names for read-only named data." In *Computer Communications Workshops (INFOCOM WKSHPS), 2012 IEEE Conference on*. IEEE, 2012.

[BFL96] Matt Blaze, Joan Feigenbaum, and Jack Lacy. "Decentralized trust management." In *Proc. of IEEE Symposium on Security and Privacy*, 1996.

[BGR07] Lujo Bauer, Scott Garriss, and Michael K Reiter. "Efficient proving for practical distributed access-control systems." In *ESORICS*, 2007.

[BHK15] Richard Barnes, Jacob Hoffman-Andrews, and James Kasten. "Automatic Certificate Management Environment (ACME)." IETF Draft, October 2015.

[BLL98] Ahto Buldasi, Peeter Laud, Helger Lipmaai, and Jan Villemson. "Timestamping with Binary Linking Schemes." In *CRYPTO'98*, 1998.

[Bri11] Peter Bright. "Independent Iranian hacker claims responsibility for Comodo hack." http://arstechnica.com/security/2011/03/independent-iranian-hacker-claims-responsibility-for-comodo-hack/, March 2011.

[BS04] Moritz Y Becker and Peter Sewell. "Cassandra: Distributed access control policies with tunable expressiveness." In *Proc. of International Workshop on Policies for Distributed Systems and Networks (POLICY)*, 2004.

[CDF07] J Callas, L Donnerhacke, H Finney, D Shaw, and R Thayer. "OpenPGP Message Format." RFC 4880, November 2007.

[CEE01] Dwaine Clarke, Jean-Emile Elien, Carl Ellison, Matt Fredette, Alexander Morcos, and Ronald L Rivest. "Certificate chain discovery in SPKI/SDSI." *Journal of Computer Security*, 2001.

[CM03] William Clocksin and Christopher S Mellish. *Programming in PROLOG*. Springer Science & Business Media, 2003.

[CSF08] D Cooper, S Santesson, S Farrell, S Boeyen, R Housley, and W Polk. "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile." RFC 5280, 2008.

[CSP15] Laurent Chuat, Pawel Szalachowski, Adrian Perrig, Ben Laurie, and Eran Messeri. "Efficient gossip protocols for verifying the consistency of certificate logs." In *Communications and Network Security (CNS), 2015 IEEE Conference on*, pp. 415–423. IEEE, 2015.

[CW09] Scott A Crosby and Dan S Wallach. "Efficient Data Structures For Tamper-Evident Logging." In *Security Symposium*, pp. 317–334. Usenix, 2009.

[DGT12] Steven DiBenedetto, Paolo Gasti, Gene Tsudik, and Ersin Uzun. "ANDaNA: Anonymous Named Data Networking Application." In *Network & Distributed System Security Symposium (NDSS)*, 2012.

[DH76] Whitfield Diffie and Martin E Hellman. "New directions in cryptography." *Information Theory, IEEE Transactions on*, 1976.

[DR08] T. Dierks and E. Rescorla. "The transport layer security (TLS) protocol version 1.2." RFC 5246, 2008.

[FKK11] Alan Freier, Philip Karlton, and Paul Kocher. "The Secure Sockets Layer (SSL) Protocol Version 3.0." RFC 6101, August 2011.

[GHJ94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[GPS06] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. "Attribute-based encryption for fine-grained access control of encrypted data." In *ACM CCS*, 2006.

[GST15] Cesar Ghali, Marc A Schlosberg, Gene Tsudik, and Christopher A Wood. "Interest-Based Access Control for Content Centric Networks." In *Proceedings of the 2nd International Conference on Information-Centric Networking*. ACM, 2015.

[HS90] Stuart Haber and W Stornetta. "How to Time-Stamp a Digital Document." In *CRYPTO'90*, 1990.

[HS12] Paul Hoffman and Jakob Schlyter. "The DNS-based authentication of named entities (DANE) transport layer security (TLS) protocol: TLSA." RFC 6698, 2012.

[Jim01] Trevor Jim. "SD3: A trust management system with certified evaluation." In *Proc. of IEEE Symposium on Security and Privacy*, 2001.

[JST09] Van Jacobson, Diana K. Smetters, James D. Thornton, Michael F. Plass, Nicholas H. Briggs, and Rebecca L. Braynard. "Networking Named Content." In *Proceedings of CoNEXT*, 2009.

[KS05] Stephen Kent and Karen Seo. "Security Architecture for the Internet Protocol." RFC 4301, December 2005.

[KWU15] Jun Kurihara, C Wood, and Ersin Uzun. "An Encryption-Based Access Control Framework for Content-Centric Networking." In *IFIP Networking Conference*, 2015.

[Lau14] Ben Laurie. "Certificate transparency." *Queue*, 2014.

[LJD14] Jinjin Liang, Jian Jiang, Haixin Duan, Kang Li, Tao Wan, and Jianping Wu. "When HTTPS meets CDN: A case of authentication in delegated service." In *Security and Privacy, 2014 IEEE Symposium on*. IEEE, 2014.

[LMW02] Ninghui Li, John C Mitchell, and William H Winsborough. "Design of a role-based trust-management framework." In *Proc. of IEEE Symposium on Security and Privacy*, 2002.

[LWM01] Ninghui Li, William H Winsborough, and John C Mitchell. "Distributed credential chain discovery in trust management." In *Proc. of Conf. on Comp. and Comm. Security (CCS-8)*, 2001.

[MB02] Petros Maniatis and Mary Baker. "Enabling the Archival Storage of Signed Documents." In *the USENIX Conference on File and Storage Technologies (FAST) 2002*. Usenix, 2002.

[Mer80] Ralph C Merkle. "Protocols for Public Key Cryptosystems." In *Security and Privacy, 1980 IEEE Symposium on*. IEEE, 1980.

[Moi14] Ilya Moiseenko. "Fetching content in Named Data Networking with embedded manifests." Technical Report NDN-0025, NDN, 2014.

[MTM13] Satyajayant Misra, Reza Tourani, and Nahid Ebrahimi Majd. "Secure content delivery in information-centric networks: design, implementation, and analyses." In *Proceedings of the 3rd ACM SIGCOMM workshop on Information-centric networking*. ACM, 2013.

[Nak08] Satoshi Nakamoto. "Bitcoin: A Peer-to-Peer Electronic Cash System.", 2008.

[NDN15a] NDN Team. "Libraries / NDN Platform." `http://named-data.net/codebase/platform/`, 2015.

[NDN15b] NDN Team. "NDN Regular Expression." `http://named-data.net/doc/ndn-cxx/current/tutorials/utils-ndn-regex.html`, 2015.

[Sec12] The H Security. "Trustwave issued a man-in-the-middle certificate." http://www.h-online.com/security/news/item/Trustwave-issued-a-man-in-the-middle-certificate-1429982.html, February 2012.

[Sha79] Adi Shamir. "How to share a secret." *Communications of the ACM*, 1979.

[SJ09] Diana Smetters and Van Jacobson. "Securing network content." Technical report, PARC, 2009.

[SMA13] S Santesson, M Myers, R Ankney, A Malpani, S Galperin, and C Adams. "X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP." RFC 6960, 2013.

[TW16] C. Tschudin and C. Wood. "File-Like ICN Collection (FLIC)." Internet-Draft, draft-tschudin-icnrg-flic-00, 2016.

[VBC15] Martín Vigil, Johannes Buchmann, Daniel Cabarcas, Christian Weinert, and Alexander Wiesmaier. "Integrity, authenticity, non-repudiation, and proof of existence for long-term archiving: a survey." In *Elsevier Computers & Security*, 2015.

[Wil15] Kathleen Wilson. "Distrusting New CNNIC Certificates." https://blog.mozilla.org/security/2015/04/02/distrusting-new-cnnic-certificates/, April 2015.

[YAC15] Yingdi Yu, Alexander Afanasyev, David Clark, kc claffy, Van Jacobson, and Lixia Zhang. "Schematizing Trust in Named Data Networking." In *Proceedings of the 2nd International Conference on Information-Centric Networking*. ACM, 2015.

[YL06] Tatu Ylonen and Chris Lonvick. "The secure shell (SSH) protocol architecture." RFC 4251, January 2006.

[Yu15] Yingdi Yu. "Public Key Management in Named Data Networking." Tech. Rep. NDN-0029, NDN, 2015.

[ZAB14] Lixia Zhang, Alexander Afanasyev, Jeffrey Burke, Van Jacobson, kc claffy, Patric Crowley, Christos Papadopoulos, Lan Wang, and Beichuan Zhang. "Named Data Networking." *ACM Computer Communication Reviews*, 2014.