UNIVERSITY OF CALIFORNIA
Los Angeles

Developing a Trust Plane for Interconnected Networks

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

by

Tianyuan Yu

2025

© Copyright by
Tianyuan Yu
2025

ABSTRACT OF THE DISSERTATION

Developing a Trust Plane for Interconnected Networks

by

Tianyuan Yu
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2025
Professor Lixia Zhang, Chair

The Internet connects entities in trust relationships in cyberspace. However, the original Internet protocol stack lacked built-in security protection. As new threats emerged, security features were added as patches, resulting in a patchwork of incompatible solutions. This accumulation of retrofits created a fragmented security landscape, in which each application must develop its own protection mechanisms. Consequently, implementations of modern defense strategies, such as Zero Trust, often depend on practitioners assembling multiple independently developed or isolated solutions.

This dissertation aims to enable systematic development of security solutions by introducing a framework, called a *trust plane*, that represents and formalizes existing trust relationships among entities in cyberspace. Trust plane uses *semantic naming* and provides structured support for core security functions—authentication, authorization, and auditing (the 3As).

This dissertation formulates the trust plane framework, identifying and designing the system components and functions needed to implement it. These include a functional module for managing security parameters and APIs that applications utilize to secure data commu-

nications; mechanisms for securely bootstrapping new entities with the necessary parameters for authenticated and authorized communication; mechanisms for maintaining trust relationships over time; and an immutable ledger that supports data auditing. The utility of the trust plane is demonstrated through a case study in the cellular core, illustrating how it enables control plane functions within the cellular core to communicate securely while significantly reducing system complexity and security overhead compared to existing implementations.

As the first work to establish and implement a trust plane, this dissertation outlines a path toward an Internet where applications are provided with systematic support for authentication, authorization, and auditing.

The dissertation of Tianyuan Yu is approved.

Sam Kumar

Omid Abari

George Varghese

Songwu Lu

Lixia Zhang, Committee Chair

University of California, Los Angeles

2025

Our knowledge can only be finite, while our ignorance must necessarily be infinite.

— *Karl Popper*

TABLE OF CONTENTS

1 Introduction **1**

2 Background **6**

2.1 Internet Security Solutions: What We Have 7

2.1.1 For Applications 7

2.1.2 For Infrastructures 8

2.2 The Core Objectives: the 3As on Identifier Spaces 9

2.3 Challenges in Zero Trust 10

2.4 Requirements to Systematically Supporting the 3As 12

2.5 Named Data Networking 13

2.5.1 Named Secured Data 13

2.5.2 Stateful Forwarding 14

2.5.3 Transport: Dataset Synchronization (Sync) 14

3 Trust Plane **16**

3.1 Identifying Trust Domains 17

3.2 Developing A Trust Plane 18

3.3 Semantic Naming is the Key 19

3.3.1 Named Secured Data Is All You Need 19

3.3.2 Leveraging the DNS Namespace 20

3.4 Authentication Design 21

3.4.1 Trust Domains and Intra-Domain Communications 21

3.4.2 Inter-Domain Communications 22
3.5 Authorization Design 23
3.5.1 LightVerSec Syntax 24
3.5.2 Sample Trust Schema for NDNFit 24
3.5.3 Name Schema Tree 26
3.6 Trust Information Base (TIB) Design and Communication API 27

4 Security Bootstrapping 31
4.1 A Four-Step Process 31
4.2 Hydra Bootstrapping 32
4.2.1 Hydra Overview 32
4.2.2 Exploring the Design Space 33
4.2.3 Cornerstone Overview 34
4.2.4 Installing Trust Anchor and Trust Schema 35
4.2.5 Authenticating New Hydra Entity 36
4.2.6 Naming New Hydra Entity 39
4.2.7 Certification 41
4.3 Ownly Bootrapping 42
4.3.1 Ownly Overview 42
4.3.2 Addressing Technical Challenges 44
4.3.3 Workspace Initialization 45
4.3.4 Invitation New Users 46
4.3.5 A Summary of Operational Workflow 47
4.4 Related Work 49

4.4.1 Leveraging Direct Connection 49
4.4.2 Leveraging Physical Proximity 50
4.4.3 Limitations 51

5 CertRevoke 52
5.1 When We Need Revocation 53
5.1.1 Invalidating Certificates 53
5.1.2 An Example Scenario 54
5.1.3 Revoking Name-Key Endorsements 56
5.2 The CertRevoke Framework 56
5.2.1 Problem Statements 57
5.2.2 Revocation Records 59
5.2.3 Revocation Submission Protocol 60
5.2.4 Certificate Revocation Checking 62
5.2.5 Putting Together A Case Study 63
5.3 Evaluation 64
5.3.1 Implementation 64
5.3.2 Evaluation Scenario 64
5.3.3 Revocation Submission 65
5.3.4 Revocation Record Checking 66
5.3.5 Benefit from In-network Caching 66
5.4 Discussions 70

6 CLedger: A Secure Distributed Certificate Ledger 73

6.1 Background 73
6.2 System Model 74
6.3 Design Goals 75
6.4 CLedger Design 75
6.4.1 Certificate Submissions 76
6.4.2 Interlocking Records 76
6.4.3 System Operations 78
6.5 Evaluation 79
6.5.1 Evaluation Setup 80
6.5.2 Submission Latency and Fetching Latency 80
6.5.3 Immutability Latency 81
6.5.4 Certificate/Total Record Ratio 82
6.6 Discussion 83

7 TrustNG 85
7.1 Background 85
7.2 Problem Statement 87
7.3 TrustNG Design 88
7.3.1 Semantic Naming 89
7.3.2 Authentication and Authorization 90
7.3.3 Policy Execution 92
7.4 Implementation 94
7.5 Evaluation 97
7.5.1 Latency and Load Resilience 97

7.5.2 Resource Overhead 99

7.5.3 Bandwidth Utilization 100

8 Related Work 102

8.1 Trust Domains 102

8.2 SPKI/SDSI 104

9 Discussion 105

9.1 Infrastructure Security 105

9.2 Cost of Signing Data 105

9.3 Independence from Networking Architecture 106

9.4 Evolution of Several Works 106

10 Other Contribution: In-Network Data Repository 108

10.1 Repo Design 108

10.2 Repo in Operation 110

10.3 Becoming Application Oblivious 113

10.4 Routing and Data Security: 115

10.5 Supporting Fully Asynchronous Applications 115

10.6 Implementation and Deployment 117

11 Conclusion and Future Work 118

References 120

.1 NAAP Protocol Details 127

LIST OF FIGURES

2.1 Forwarding pipelines at an NDN Router: upstream indicates the direction of the data producer and downstream is towards the data consumer. The consumer expresses Interest to network to named and secured Data. The Interest lifetime indicates for how long an Interest should stay in PIT before being satisfied. . . . 15

3.1 An example of trust relationships among cyberspace entities 16

3.5 An example of TIB usage and the Produce/Consume API for NDNFit. 27

4.1 The entities in the Hydra trust domain. Not shown here are the trust relationships among the entities. 33

4.2 Overall workflow of Cornerstone. The magnifier focuses on the mutual authentication process and the upper part of this figure reveals more details of this two-way authentication. 36

4.3 NAAP Overview 37

4.4 Example of name assignment based on domain names 40

4.7 Certificate chains involved in Ownly bootstrapping. The initiator and invitee have already mutually authenticated by their own names. 46

5.1 A routing prefix registration scenario where certificate revocation happens. . . . 55

5.2 An example of multi-path authentication. 57

5.3 CertRevoke System Model 58

5.4 Site administrator Alice revokes the certificate of CS department 60

5.5 Interest-Data exchanges between Revoker and Ledger 61

5.6 Evaluation scenario where checker and ledger, revoker and ledger are multiple hops away. 65

5.7 The latency of Revoker submitting revocation records to the Local Ledger . . . 65

5.8 The latency of checking revocation records from the Local Ledger 66

5.9 Evaluation when checked records follow Uniform distribution 67

5.10 Evaluation when checked records follow Zipf distribution 68

5.11 Normalized latency with popularity rank 69

5.12 Example of Revocation Impact 71

6.1 A scenario where CLedger has Loggers deployed in the network, and CertOwners would like to use CLedger to provide their certificate availability, while CertFetchers want to obtain certificates from CLedger in order to validate application data. 74

6.2 A DAG example where $K = 3$ and each record is published by an unique Logger. 77

6.3 An CLedger example where Alice is the CertOwner while Bob is the CertFetcher. 79

6.4 Availability Latency 80

6.5 Immutability Latency 82

6.6 Certificate/Total Record Ratio 83

7.1 5G network slicing architecture with distributed VNF deployment, highlighting session setup, registration, and discovery processes. 86

7.2 naming convention for trust management in next-generation cellular networks with multi-domain, -tenant, -slice deployments. 89

7.5 TrustNG system architecture: A Kubernetes-based 5G core deployment with NDN integration. Each VNF Pod includes an NDN-SCP sidecar container for NDN communication. NFD pods enable routing and forwarding. 94

7.6 Internal architecture of the NDN-SCP. The proxy transparently intercepts HTTP traffic and translates it into semantically meaningful NDN Interest/Data exchanges. 96

7.7 HTTP-to-NDN translation example. An HTTP request URI is semantically mapped to an NDN name. 97

7.8 Comparison of TrustNG with traditional 5G core operational variants 98

7.9 Average CPU and Memory Utilization 100

10.1 Alice sends Object Insertion Commands to Repo 110

10.2 ADU Insertion Request and Insertion Request Parameters 111

10.3 Checking ADU Availability Status 112

10.4 In network P, three Ownly workspaces users are running NDN Sync over group G. Repo is an oblivious participant in G, as requested by U_1. 114

10.5 An example where three Ownly users fully asynchronously collaborate by Repo updating Interest cache and replaying them upon updated Sync Interests. 116

10.6 The topology map of global NDN Testbed. Repo is deployed on every Testbed router. 117

.1 Hydra server authentication workflow 127

.2 Hydra user authentication workflow 128

LIST OF TABLES

7.1 Bandwidth usage during 600 seconds of UE registration at 1 req/sec. C: Caching Access Token, NC: No Caching 101

ACKNOWLEDGMENTS

I am deeply grateful to my advisor and committee chair, Prof. Lixia Zhang, for her patience, guidance, and insight throughout my doctoral journey. Her mentorship has shaped the course of this dissertation and has been instrumental to my development as a researcher.

I would also like to express my sincere appreciation to my committee members—Prof. Songwu Lu, Prof. George Varghese, and Prof. Omid Abari—for their time, constructive feedback, and valuable suggestions, all of which have strengthened the quality of this work. I am especially grateful to Prof. Sam Kumar for serving as the security representative on my dissertation committee. His careful evaluation has played an important role in refining and clarifying the contributions of this thesis.

I have had the privilege of working with many colleagues and coauthors whose contributions have enriched this research. I especially thank Zhiyi Zhang, Xinyu Ma, Varun Patil, Philipp Moll, Hongcheng Xie, Adam Thieme, Amirreza Ghafoori, Tolga Ataly, Alexander Afanasyev, Jeffrey Burke, Davide Pesavento, Siqi Liu, and Susmit Shannigrahi, as well as members of the NDN community. Their collaboration, discussions, and technical insights have been invaluable to the completion of this dissertation.

Chapter 7 is based on joint work with Tolga Ataly and Amirreza Ghafoori. My contributions include the system design and library support, and my collaborators contributed the implementation and evaluation.

VITA

2013–2017	B.S. (Electronics and Information Science and Technology), Sichuan University.
2020-2025	Teaching Assistant, Computer Science Department, UCLA. Taught sections of Computer Science 118 (computer networking) under direction of Professor Lixia Zhang.
2019–2025	Graduate Student Researcher, Internet Research Lab, Computer Science Department, UCLA.

CHAPTER 1

Introduction

The original Internet protocol stack lacked built-in security protections [Cla88]. As security threats emerged in reality, various specific solutions were patched into the deployed system, each addressing a specific threat. This patchwork approach diminishes the overall effectiveness of all deployed solutions and complicates the development of secure, autonomous, and resilient systems that span administrative boundaries domains. Over time, the Internet has evolved into a diverse ecosystem of applications running on top of communication supports with fragmented and often incompatible security mechanisms. These include isolated or orthogonal solutions such as DNSSEC for authenticating DNS lookups, RPKI for routing announcement authentication, and the Web PKI for web-based applications; occasionally, they even conflict with one another, such as between VPNs and firewalls. The drawbacks of deploying a collection of piecemeal solutions have become clearer when one attempts to support *Zero-Trust* as the latest security strategy. Zero-Trust emphasizes that one should "never trust, always verify," all security operations adhere to "least privilege," and organizations should be prepared for "assume breach." To meet these requirements, Zero-Trust practitioners must assemble and integrate a complex set of independently developed security solutions.

To address the issue mentioned above, this dissertation introduces a *trust plane* framework that provides a systematic approach to developing Internet security solutions. The trust plane employs the 3As – authentication, authorization, and auditing – as its core security mechanisms.

The first requirement for authentication is that cyberspace entities be identified *semantically.* The trust plane uses semantically meaningful DNS names to identify all cyberspace entities and the data they generate, and installs trust anchors to issue certificates to semantically identified entities. Authorization decisions are based on the trust relationship between semantically identified entities. The trust plane captures trust relationships by defining corresponding security policies that specify who can do what, with both who and what identified by semantic names. Finally, auditing requires immutable logs to record all actions taken in cyberspace, including certificate issuance and changes, security policy updates, and all activities performed. The trust plane encodes all of the above as semantically *named, secured data* items and records them in an immutable ledger. Semantic naming offers a rich source of information for comprehensive and fine-grained auditing. Audit logs can directly record meaningful events at the data communication level, such as which named entity accessed or produced which named data.

We further note that trust relationships between entities can be broadly classified into two categories: within the same administrative domain or between different administrative domains. Correspondingly, we formulate a concept of *trust domain*[1]. Within each trust domain, the system administrator (or domain controller) owns the trust anchor, defines trust policies, and installs them into all entities in the domain, enabling them to authenticate one another via the shared trust anchor. Authentication between trust domains requires mutual verification, whereas authorization – such as determining which actions an external entity may perform – remains the decision of each domain's local controller.

One could envision the trust plane building a network of named entities in cyberspace, with links between them representing their trust relationships rather than physical connectivity. The trust plane uses semantically named, secured data as the basic building block. The security of data is preserved regardless of the transport protocols or intermediaries: compro-

[1]We recognize that the term trust domain has been used in various previous works, and we compare these differing uses in Chapter 8.

mising a connection does not compromise the data being transported. System auditing can also be configured to the desired granularity. Each entity E maintains a *Trust Information Base* (TIB) that contains its own trust anchor, certificates, and security policies, as well as a set of verified certificates for known other entities. E's TIB provides the information for E to produce signed data items and verify received data items from others, as well as carry out actions authorized by its policies.

Developing such a trust plane requires a functional module to store the trust anchor, certificates, and trust policies, along with application library support for its use, practical bootstrapping solutions to configure entities with these security parameters, usable mechanisms to maintain trust relationships (such as certificate revocation), and an immutable ledger to provide an audit log.

This dissertation describes work on all four components. Specifically, it makes the following contributions.

1. **Trust Plane Formulation:** We identify trust domains that represent the administration domains in cyberspace, and entities within and between these trust domains have different trust relationships with one another. We then define the trust plane concept, outline the trust plane framework, and identify the essential system functions and components needed to support the 3As.

2. **Trust Information Base and Secure Communication API:** We develop the functional module *Trust Information Base (TIB)* to record trust relationships and store security parameters at each entity, as well as the secure communication API for application use. Applications use it to securely produce and consume named secured data, with built-in data authentication and authorization support, and the library is implemented on top of Named Data Networking (NDN) to deliver secured data based on their semantic names efficiently.

3. **Security Bootstrapping:** We develop a four-step bootstrapping process that includes

mutual authentication, trust anchor and trust schema installation, and certificate issuance. We then apply this process to create bootstrapping solutions for a federated file storage system and a decentralized collaborative editor.

4. **Certificate Revocation:** We design *CertRevoke*, a lightweight certificate revocation mechanism that leverages NDN's data-centric forwarding plane to distribute revocation information efficiently.

5. **Distributed Certificate Ledger:** We design *CLedger*, a distributed certificate (and could be generic data) ledger that keeps a verifiable and immutable history of certificate issuance and revocation as a Directed Acyclic Graph (DAG) and can be used for generic data logging to support auditing.

6. **Applying the Trust Plane in the Cellular Core Network:** We develop *TrustNG*, a next-generation cellular core architecture that uses the trust plane to provide systematic support for control-plane communications. It captures the trust relationships among the 5G core network operators, tenants, network slices, and Virtual Network Functions (VNFs) using semantic names in the DNS namespace. It enables bootstrapped VNFs to communicate with the named secure service data. The preliminary evaluation results show that, compared to current practice, TrustNG has significantly lower security complexity and operating overhead.

The remainder of this dissertation is organized as follows. Chapter 2 reviews existing Internet security solutions and summarizes the NDN primitives used in this work. Chapter 3 formalizes trust domains and explains how intra- and inter-domain trust relationships create the trust plane, followed by a description of the TIB and the communication API we developed. Chapter 4 systematizes security bootstrapping and provides specific solutions for two practical applications. Chapter 5 presents the CertRevoke framework, and Chapter 6 introduces CLedger as a first step towards auditing support. Chapter 7 applies the trust plane to cellular cores, enabling secure and resilient inter-VNF interactions. Chapter 8

reviews related work and Chapter 9 addresses common points of clarification. Chapter 10 presents other contributions that the author has made that are relevant to the presented work. Chapter 11 concludes this dissertation.

CHAPTER 2

Background

Butler W. Lampson, in *Computer Security in the Real World* [Lam04], states that

> "... security is not about perfect defenses against determined attackers. Instead, it's about value, locks, and punishment. The bad guy balances the value of what he gains against the risk of punishment, which is the cost of punishment times the probability of getting punished. The main thing that makes real world systems sufficiently secure is that bad guys who do break in are caught and punished often enough to make a life of crime unattractive. The purpose of locks is not to provide absolute security, but to prevent casual intrusion by raising the threshold for a break-in."

Based on matters that are important in practice, Lampson identifies the three fundamental mechanisms for implementing effective security: **A**uthentication, **A**uthorization, and **A**uditing (the 3As). Authentication and authorization prevent unauthorized access to resources. By recording "who did what and when," authentication and auditing together establish system accountability, which is essential for enabling real-world consequences.

In the rest of this section, we begin by examining the current security solutions. We show that they can all be sorted into supporting some parts of the 3As. Then, we discuss the technical challenges involved in deploying these solutions collectively to support Zero-Trust. Finally, we provide a brief overview of the design of Named Data Networking (NDN), which we use to implement the trust plane.

2.1 Internet Security Solutions: What We Have

2.1.1 For Applications

Transport Layer Security (TLS) [DA99] was introduced to encrypt TCP connections between browsers and web servers, which in turn created the need to authenticate the public keys of the servers. As a result, trust in server identities was delegated to third parties: Certificate Authorities (CAs) [BSP08], which endorse domain owners' public keys by issuing X.509 certificates. The system of certificate issuance and revocation became known as the Web PKI. Root certificates in this ecosystem are owned by CAs and distributed out-of-band through software update channels, such as operating system or browser releases. As HTTPS became the de facto standard for securing the Web, incidents of CA misbehavior through certificate misissuance [Pri11, Moz18, Goo15] gained attention, leading to the introduction of Certificate Transparency (CT) [LLK13]. Initiated by Google, CT provides public append-only logs of certificates so that the community can audit issuance and detect misbehaving CAs. However, operating a global log infrastructure requires significant resources, which only hyperscalers and large CAs can support.

At the application layer, modern web services typically follow the Single Sign-On (SSO) model [Har12, OAS05], where applications act as Service Providers (SP) and rely on external Identity Providers (IdP) to authenticate users. When a user logs in, the SP invokes the IdP's HTTPS API, and the IdP issues a signed identity assertion (or token) that can be verified through the IdP's public key, published at a well-known URL in its domain. In effect, each IdP operates its own application-layer PKI, layered on top of the Web PKI for secure transport. However, this still requires trust relationships between different parties, SPs, IdPs, and CAs, making PKI indispensable for user authentication at scale. OAuth 2.0 [Har12] has become the de facto standard for web authorization. Resource owners grant limited access to their protected resources by authorizing an OAuth client to obtain an access token. This token is issued by an authorization server and conveys the scope and

duration of the granted access. Because these tokens must be protected in transit and at rest, OAuth deployments rely on HTTPS, and thus inherit the trust relationships and certificate validation dependencies of the Web PKI.

2.1.2 For Infrastructures

Firewalls act as traffic filters by inspecting packet headers and payloads to block undesired or malicious traffic before it reaches its destination. To augment these coarse filtering rules, network operators often deploy Intrusion Detection and Prevention Systems (IDS/IPS) [Ope24, Zee18], which monitor traffic for signatures of known attacks or anomalous behavior and, in the case of IPS, actively block suspicious flows based on IP addresses and port numbers. Although these mechanisms reduce the immediate attack surface, their restrictive policies can degrade connectivity, while their operational complexity and false positives introduce significant friction in managing distributed applications.

DNSSEC [AAL05] introduced a cryptographic hierarchy in which each domain controls a Zone Signing Key (ZSK) to sign records and a Key Signing Key (KSK) to sign the ZSK. These KSKs are chained upward through parent zones until the DNSSEC root, forming a single global trust root. The DNSSEC root key, like the Web PKI root, is installed out of band via software distribution. However, this design requires global coordination, and its complex signing relationships (ZSK/KSK rotation, TTL planning) make key management difficult. As a result, DNSSEC adoption has remained limited, while Web PKI continued to dominate web security.

To secure the global routing system so that packets are delivered to the correct destination, the Resource Public Key Infrastructure (RPKI) [LK12b] was introduced to verify which Autonomous Systems (ASes) are authorized to originate specific IP prefixes, addressing long-standing vulnerabilities in the Border Gateway Protocol (BGP). RPKI does so by binding Internet number resources, IP address blocks, and AS numbers to certificates issued by Internet registries. These certificates are then used to generate Route Origin Authoriza-

tions (ROAs), which BGP routers can use to validate incoming route announcements and detect hijacks or configuration errors. However, like DNSSEC, RPKI requires coordinated deployment across multiple independent administrative domains. As a result, adoption has been uneven, with significant variation between providers and regions.

Virtual Private Networks (VPNs) were introduced to establish encrypted tunnels between communication endpoints. Each endpoint maintains a cryptographic key pair, and only packets authenticated with the corresponding public key are accepted. However, this point-to-point model introduces a scalability challenge: As the number of tunnels increases, the number of keys to manage grows linearly. The manual distribution and rotation of keys quickly become infeasible in large-scale deployments. To address this, many modern VPN systems adopted a retrofit approach analogous to the Web PKI: instead of provisioning shared keys with each client, the server side of the VPN tunnel is authenticated using X.509 certificates. In commercial VPN services, these certificates are often issued by trusted Certificate Authorities (CAs) on the Web PKI, while on the client side, VPN providers increasingly rely on external authentication systems, such as Identity Providers (IdPs) using Single Sign-On (SSO) protocols [Har12, OAS05], to verify user identities.

2.2 The Core Objectives: the 3As on Identifier Spaces

Across today's Internet security mechanisms, regardless of which layer they are deployed deployed at, the core objectives remain the same: identify legitimate entities (**A**uthentication), determine whether they are allowed to perform an action (**A**uthorization), and generate evidence of activity for later inspection (**A**uditing). All existing mechanisms perform these 3As, and each uses the identifier space that is visible or natural to its layer or functionality.

For Applications: Application mechanisms implement the 3As on top of DNS names and application-level identifier spaces. The Web PKI authenticates servers by binding DNS

names to public keys, and CT provides an auditable record of certificate issuance indexed by DNS names. IdPs authenticate users and client applications through identifiers native to their own domains—such as opaque user IDs or subject claims that are not part of the DNS namespace. Authorization is expressed through policies and resource identifiers defined within application-specific namespaces.

For Infrastructures: Infrastructure mechanisms implement the 3As on top of network-level identifiers, such as addresses, ports, protocol fields. IPsec authenticates endpoints by binding cryptographic keys to IP addresses; firewalls authorize traffic according to rules expressed over address and port tuples; and IDS/IPS detect or block suspicious flows based on signatures or anomaly scores computed over packet headers and flow metadata, while RPKI authenticates the origin of BGP route announcements using IP prefixes and AS numbers.

Infrastructure mechanisms rely on IP addresses and port numbers for historical reasons, consistent with the origins of the Internet as a point-to-point packet delivery fabric. In contrast, *Semantic* namespaces such as DNS encode information about the principals, resources, and application context, making them inherently more suitable for expressing policies.

2.3 Challenges in Zero Trust

Zero Trust Architecture (ZTA) has emerged as a strategic response to the growing inadequacy of perimeter-based security models that rely on firewalls and traditional VPNs to enforce trust boundaries. It rests upon three foundational principles: *Never trust, always verify*, *Least privilege*, and *Assume breach* [Mic24, RBM20]. These principles aim to continuously validate the identity, minimize the scope of the authorization, and contain potential compromises. However, practical ZTA deployments today are largely constructed atop the fragmented and retrofitted security mechanisms aforementioned, inheriting their inherent fragmentation and complexity.

Never Trust, Always Verify: This principle requires explicit and continuous verification of every request, regardless of network location or any previously established authentication state. However, in practice, such verification is not implemented through a unified architectural mechanism. Instead, organizations rely on a patchwork of existing systems: federated IdPs authenticate users through application-layer tokens, while server-to-server authentication is enforced through mTLS and X.509-based mechanisms such as SPIFFE [SPI25].

Least Privilege: The second principle mandates that authenticated entities be granted only the minimum permissions necessary for a specific task and for the shortest feasible duration. In principle, this requires a unified mechanism for expressing and enforcing fine-grained authorization policies across users, devices, and workloads. In practice, however, Least Privilege is implemented through a collection of techniques—Role-Based or Attribute-Based Access Control (RBAC/ABAC) policies in application platforms, Just-in-Time (JIT) access provisioning in cloud environments, and micro-segmentation in network infrastructures. Each enforcement point—API gateways, service meshes, cloud control planes, and application runtimes—operates its own policy language, configuration format, and identifier space. As a result, the semantics of "least privilege" must be repeatedly translated across incompatible systems (*e.g.,* token scopes for OAuth clients, SPIFFE IDs for server workloads, or subject names in X.509 certificates), fragmenting policy enforcement.

Assume Breach: The third principle, that breaches should be assumed rather than treated as exceptional events, drives continuous monitoring, rapid containment, and post-incident accountability. In practice, this principle is implemented through a collection of specialized platforms, each targeting different layers of the system. Security Information and Event Management (SIEM) systems such as [IBM25] ingest logs from servers, applications, and network devices and provide correlation and alerting, while Security Orchestration, Automation, and Response (SOAR) platforms [Pal25, Spl25] execute automated playbooks that may revoke credentials, isolate workloads, or update firewall rules when a SIEM raises an alert. However,

SIEMs rely on collected system logs with IP addresses, hostnames, HTTP paths, or session cookies; and SOAR platforms have to work with this mix of identifiers when performing automated response actions.

2.4 Requirements to Systematically Supporting the 3As

The challenges described in Zero Trust highlight the need for systematic support for the 3As.

A unified and semantic identifier space: Systematic support of authentication and authorization requires a unified identifier space that can name cyberspace entities and data objects in a way that reflects their roles and relationships. Such identifiers must have meaningful semantics for the applications, thus enabling policies to be written and enforced at a level that matches the way applications reason about participants and resources.

Trust anchor for authentication: Security systems require a root of trust from which authentication begins. A trust anchor serves as the ultimate authority that signs or endorses the semantic identifiers within its scope. Given a trust anchor, the entities participating in the communication can have certificates that link their names to public keys. These certificates must be traceable back to the trust anchor through verifiable chains of signatures.

Trust policies for authorization: Beyond authenticating identities, systems must specify and enforce policies that describe which entities are allowed to perform what actions on which resources. Trust policies express these rules in the unified semantic identifier space of entities and data objects. Given configured trust policies, an entity can detect if the ongoing communication attempt is legitimate.

Immutable Ledger for auditing: Systematic audit support requires logging of all system events, including certificate issuance/revocation, policy updates, and application data

production. Rather than maintaining separate logs for different components, layers, or security solutions, a comprehensive data ledger is needed. Lessons from CT further underscore that such a ledger must be append-only and ensures that previously recorded events cannot be modified or removed. Given such a comprehensive and immutable data ledger, auditors can independently verify the correctness of system behavior and identify inconsistencies or policy violations that would otherwise remain hidden.

2.5 Named Data Networking

Named Data Networking (NDN) [ZAB14, ARW18] is a newly developed network protocol architecture that shifts the network communication paradigm from TCP/IP's point-to-point node-centric model to a data-centric model, where applications communicate via fetching named, secured data. Networking is no longer about delivering packets between addressed endpoints but also moving semantically named data between named entities. The rest of the section first describes the basic building block of NDN – named, secured data, then describes how NDN utilize semantic naming in forwarding, followed by the new transport function Sync.

2.5.1 Named Secured Data

Internet applications generally use unique and semantically meaningful DNS names. The NDN design follows the application practice and extends it. Applications assign unique, hierarchically structured, and semantically meaningful names to individual pieces of data. For example, an encrypted chat message that Bob produces in a group chat application "`group-chat.com`" may carry the name "`group-chat.com/bob/seq=123`" and is signed by Bob's private key. Data names are used across the NDN protocol stack by applications, transport protocols, and network delivery.

2.5.2 Stateful Forwarding

To fetch data, NDN defines two types of packet: *Data packet* that is named and secured, and *Interest* packet that is used to fetch Data. Producers publish secured data packets and make them available. Consumers retrieve data by sending Interests that carry the names of desired data but no address. Network routers forward the Interests according to their names, and return the matching Data when found.

Figure 2.1 shows the overview of how a packet is processed within an NDN forwarder. When an Interest packet arrives, its name will be used to look up three tables in the order of Content Store (CS), Pending Interest Table (PIT), and Forwarding Information Base (FIB). The CS is essentially a cache of previously received Data, and if a match is found, the Data will be returned. The PIT records the Interests that have been forwarded to the nexthop but have not received the returning Data yet. If a match is found, the arriving Interest is aggregated to the same PIT entry by adding the incoming face of the Interest to the PIT, so that when the Data return, it will be sent to that face. The FIB is a name-based routing table, and the Interest will be forwarded to the nexthop(s) supplied by the FIB based on router's per prefix forwarding strategy and create a new entry in the PIT. On a FIB look up miss, the router may return a NACK indicating there is no route to forward.

2.5.3 Transport: Dataset Synchronization (Sync)

To fetch data promptly, NDN applications must be notified whenever new data is generated. Assuming multi-party applications as the general case, where all entities join a multicast group and share the same set of data, Sync [MPW22] provides prompt and reliable synchronization of the data names in the shared dataset among all entities participating in the same application. Sync encodes the data of each producer using a 2-tuple [producer, sequence

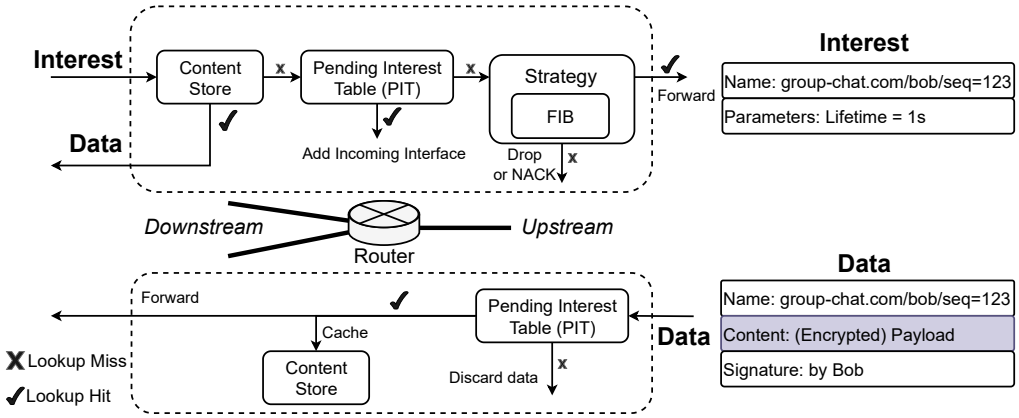


Figure 2.1: Forwarding pipelines at an NDN Router: upstream indicates the direction of the data producer and downstream is towards the data consumer. The consumer expresses Interest to network to named and secured Data. The Interest lifetime indicates for how long an Interest should stay in PIT before being satisfied.

number] and encodes the group state as a vector of such tuples [1]. Every time a producer generates new data, it immediately multicasts a Sync Interest that carries the state vector, which is signed by the sender. The entities receiving the notification have the option to decide whether and when to fetch newly produced data based on their local conditions, *e.g.,* whether the data are considered urgent, or they are well connected, or running out of battery.

[1]The sequence number is used here as an efficient way to represent one's data production. Sync also provides the mapping from the sequence number to the semantic data name.

CHAPTER 3

Trust Plane

In this chapter, we begin with identifying trust domains that represent the administration domains in cyberspace, and entities within and between these trust domains have different trust relationships with one another. We then define the trust plane concept, outline the trust plane framework, and identify the essential system functions and components needed to support the 3As. Afterwards, we move on to demonstrate why semantic naming enables the trust plane to provide structured support for them and how the trust plane exactly uses it for authentication and authorization. Finally, this chapter focuses on the first of the identified components. We present the design of the Trust Information Base (TIB) as the functional module that records the security parameters used by the trust plane, then illustrate how applications leverage the secure communication API built atop the TIB and NDN.

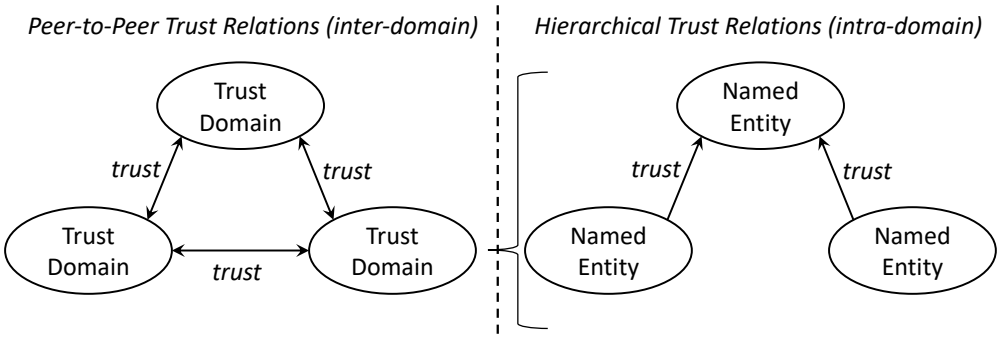


Figure 3.1: An example of trust relationships among cyberspace entities

3.1 Identifying Trust Domains

Cyberspace is a reflection of human space, and cyberspace security is regulated by trust relationships among different parties in human space. In the real world, trust relationships between named entities can follow different models. Two typical ones are hierarchical trust relationships and peer-to-peer trust relationships, as illustrated in Figure 3.1. The named entities belonging to the same administrative domain generally follow a hierarchical trust model, where all entities in the domain share a single trust root. Independent entities, on the other hand, can use a Web-of-Trust model to establish their trust relations [Gar95], where they endorse the cryptographic keys of each other in a peer-to-peer fashion. These two models complement, rather than conflict, each other, and may co-exist. For example, a smart home composed of networked IoT devices can use the hierarchical trust model for all IoT devices in a user's home, while neighboring homes may establish peer-to-peer trust relations if they desire to communicate with each other (*e.g.,* when home-A's ISP has an outage, neighbor home-B may instruct its WiFi router to help forward traffic for home-A, while prohibiting all its other devices from interacting with home-A's traffic).

The trust relationships in a hierarchical trust model ending with a single trust root. The realization of a hierarchical trust model requires that each entity E under the same administrative control establishes a local *trust anchor* that cryptographically identifies the trusted root and installs the trust anchor in E. We also refer to an administration domain with this realization as a *trust domain*. A trust domain consists of a collection of named entities in the DNS namespace under the same administrator's control. The entity that controls the trust relationships of the domain is the trust domain *controller*. It owns the trust anchor T of the domain and bootstraps the new entity E_{new} in the domain. In a trust domain, entities are authenticated by certificates issued to their names, and communications are authorized by *trust policies* defined by the controller. In addition, there is an immutable data ledger to record all system operations. As a result, all intra-domain communications

are authenticated and authorized locally with accountability support. When inter-domain communications are needed, the two controllers mutually authenticate, so that each domain's entity can be authenticated through its certificate chain, and inter-domain communications are authorized by local trust policies and logged to the local ledger.

3.2 Developing A Trust Plane

One can envision the *trust plane* building a network of named entities in cyberspace, where links represent trust relationships rather than physical connectivity. Within this network, a named entity E can securely communicate with any other entity with which it shares an intra- or inter-domain trust relationship. These trust relationships are realized through trust anchors, certificates, and trust policies, all of which encode how entities authenticate each other and what actions they are authorized to perform. Before E can participate in secure communication, it must be bootstrapped with the appropriate security parameters. Because trust relationships evolve over time, these parameters must also be continuously maintained and updated. Finally, all communications validated and secured under these parameters are recorded in the data ledger, providing a verifiable history for the system.

Based on the vision above, we identify four system functions or components that are necessary for a trust plane implementation.

- **Trust Information Base (TIB) and API:** A functional module on each entity that records all trust relationships among entities within and across trust domains. The TIB maintains the identities, certificates, keys, trust anchors, and trust policies needed to evaluate authentication and authorization decisions. It also exposes a communication API through which applications and other system components (such as an immutable data ledger) securing their data communications.

- **Security Bootstrapping:** Mechanisms that provision the initial security parameters

required for entities to participate in a trust domain. This includes installing domain trust anchors, issuing certificates, and trust policies into TIB.

- **Maintenance Mechanisms:** Mechanisms that update the security parameters as the trust plane evolves. These include certificate renewal, key rotation, revocation handling, policy updates. Maintenance ensures that the trust plane reflects the current trust relationships and continues to enforce correct authentication and authorization decisions.

- **Immutable Ledger:** An append-only audit log that records the evolution of trust relationships as well as all application-level events. By ensuring that the entire system history is durable and tamper-evident, the ledger provides a verifiable timeline that supports accountability, anomaly detection, and forensic analysis, and enables external parties to validate both the correctness and the integrity of the system over time.

3.3 Semantic Naming is the Key

3.3.1 Named Secured Data Is All You Need

To provide structured support for the 3As, we must first identify the *basic unit* on which these mechanisms are applied. In today's Internet, security solutions are largely tied to network channels—authenticating endpoints, securing connections, or filtering flows. They authenticate IP addresses, authorize ports or connections, and audit packets or flows, none of which correspond to the entities or resources that applications actually reason about. Data often outlive the connections that deliver it, are cached or replicated by intermediaries, and are consumed by entities that may not coincide in time or directly communicate. Once data leaves a connection, any security properties associated with that connection no longer persist. Therefore, these realities suggest that the 3As should apply to the *data* themselves. Section 2.4 further suggests that these data, together with the trust domains and entities

that produce and consume them, should be *semantically named*.

If data is signed directly by the producer, a consumer can verify who created it, regardless of how it was transported. If authorization policies refer to the names of data and entities, a domain controller can specify which entities may read particular classes of data and which entities may produce or modify them, regardless of how and when data are stored or transported. Likewise, audit trails tied to data names allow systems to record who produced or accessed a specific object, rather than inferring behavior from packet or flow logs.

Not only application data, but also the security parameters (keys, certificates, and trust policies) that govern them are *named secured data*. The mechanisms used to authenticate ordinary application data naturally extend to the authentication of keys and certificates. Name-based authorization rules can also apply to both data access and key usage. An immutable data ledger can also record certificates, keys, and trust policies in the exact same format, greatly easing the auditing process.

3.3.2 Leveraging the DNS Namespace

The trust plane uses the DNS namespace to semantically name entities and data for two reasons. First, DNS domains usually reflect administrative control on the Internet today. Organizations and services commonly anchor their online presence under DNS domains whose ownership is broadly understood, making these names a reasonable starting point for identifying trust domains and the entities within them. Second, DNS names are hierarchical and semantically expressive to encode application semantics. Therefore, although DNS was not originally designed for this role, it provides a reasonable foundation on which to build semantic naming for a trust plane.

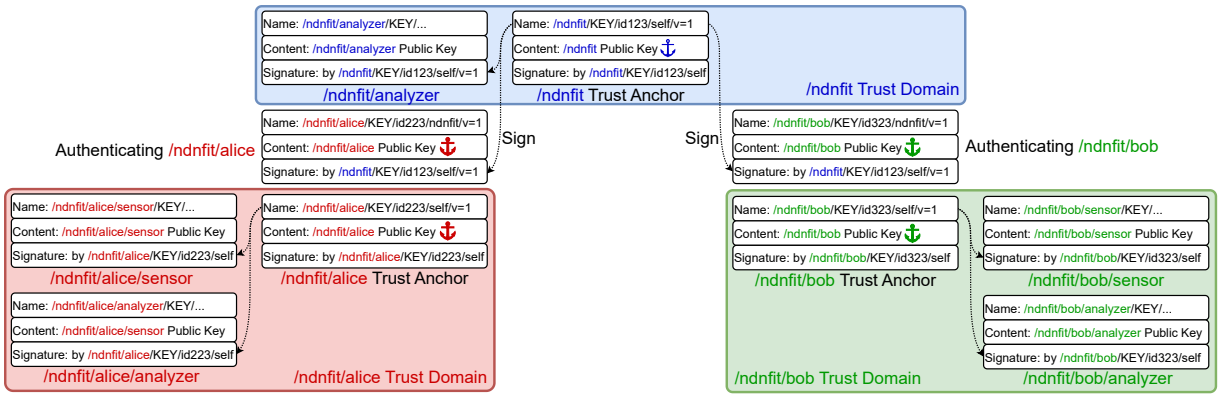


Figure 3.2: The relationships among the trust domains "`/ndnfit`", "`/ndnfit/alice`", and "`/ndnfit/bob`" and their internal entities.

3.4 Authentication Design

In this section, we use an illustrative example to describe how trust domains are established and how data are authenticated in intra- and inter-domain communications.

3.4.1 Trust Domains and Intra-Domain Communications

In Figure 3.2, we depict a realization of the trust domain using a prototype application NDNFit described in [Zha18], which tracks and shares personal fitness activities. NDNFit developers start the trust domain "`ndnfit.org`" ("`/ndnfit`" for short) by establishing its trust domain controller and the data analytics service "`/ndnfit/analyzer`", which provides an advanced data analysis service to NDNFit users. The controller of the "`/ndnfit`" domain further authenticates "`alice@example.com`" ("`/alice`" for short) and "`bob@foobar.org`" ("`/bob`" for short) as "`/ndnfit/alice`" and "`/ndnfit/bob`" issuing certificates "`/ndnfit/alice/KEY/id223/ndnfit/v=1`" and "`/ndnfit/bob/KEY/id323/ndnfit/v=1`" respectively.

Alice can set up her own trust domain "`/ndnfit/alice`", by self-signing "`/ndnfit/alice`" as her domain trust anchor. Since both users are authenticated by the "`/ndnfit`" controller, the trust anchor of the "`/ndnfit`" domain is also installed as an authenticated external trust

anchor for her local systems. Locally, Alice runs a "Sensor" app "`/ndnfit/alice/sensor`" on her smartphone to collect daily location information (such as "`/ndnfit/alice/sensor/seq=123`") and runs a "Analyzer" app "`/ndnfit/alice/analyzer`" on her laptop to produce results from personal data analysis. The "`/ndnfit/alice`" domain controller bootstraps the "Sensor" and the "Analyzer" with trust anchor, certificates, and trust schema (discussed later in Section 3.5). On the left side of the figure, Bob creates his own trust domain "`/ndnfit/bob`" and bootstraps his analyzer and sensor in the same way.

Upon the creation of a new trust domain, the administrative control over its entities becomes fully independent. Alice's sensor data are signed by the sensor certificate, which is issued by the domain trust anchor. The analyzer does not need to consult "`/ndnfit`" to authenticate the sensor data, but simply trace the signing chain that terminates within the local trust domain.

3.4.2 Inter-Domain Communications

We start the inter-domain discussion with the following case. Bob shares his sensor data "`/ndnfit/bob/sensor/seq=123`" with his friend Alice, whose personal data analyzer "`/ndnfit/alice/analyzer`" consumes the data and generates a report. As illustrated in Figure 3.2, Alice's analyzer can only trace the signing chain that terminates at the trust anchor of "`/ndnfit/bob`" and is unaware of its indirect trust relationships with Alice through "`/ndnfit`".

For this purpose, we introduce *CertList*. It is a named secured data that intuitively codifies the common practice in human society: looking for common trust relationships in authenticating an external party. In operation, the trust domain controller produces CertList to inform external data consumers of a list of certificates of the public key of the trust anchor. CertList follows the naming convention of "`/<domain>/KEY/<keyid>/auth/<version>`", where "`/<domain>/KEY/<keyid>`" identifies the key name of a trust anchor, and "`<version>`" indicates the version number of it. As illustrated in Figure 3.3, when Bob establishes his own domain after obtaining a certificate from "`/ndnfit`", bob produces a CertList data whose content is

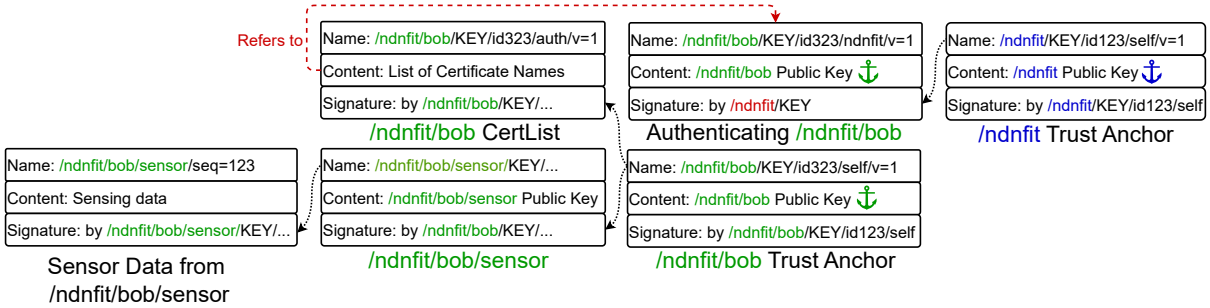


Figure 3.3: Bob shares sensor data "`/ndnfit/bob/sensor/seq=123`" to Alice, who follows the signing relationships and the CertList of "`/ndnfit/bob`" and eventually traces to an common, authenticated external trust anchor.

the name of his NDNFit certificate "`/ndnfit/bob/KEY/id323/ndnfit/v=1`" and signs it with the local trust anchor. At the time of Alice analyzer authenticating sensor data, it traces the signing chain until reaching Bob's trust anchor, then, based on naming convention, express an Interest "`/ndnfit/bob/KEY/id323/auth`" to fetch the CertList of the "`/ndnfit/bob`". Once Bob's certificate from "`/ndnfit`" is discovered via the CertList, Alice's analyzer continues the authentication until reaching its "`/ndnfit`" trust anchor, which is already out of band authenticated and installed once installing the NDNFit application.

3.5 Authorization Design

Data Authorization concerns two complementary questions: *who may read a piece of data* and *who may produce (write) a piece of data*. Read access corresponds to controlling which entities may decrypt or retrieve specific data, while write access determines which entities are allowed to create or sign particular data. Previous work on Name-Based Access Control (NAC) [ZYR18] establishes this mechanism with public key encryption. This section focuses on the design of write-access authorization.

To express write-access policies, we developed a domain-specific language LightVerSec,

a variant of VerSec [Nic21] with a simplified and more flexible syntax. LightVerSec enables authorization to be expressed as schematized constraints (trust schema) on names: a rule states which key names are permitted to sign which data names, and validation is performed by matching data names and signer names against these patterns. This section introduces LightVerSec and shows how to apply it to NDNFit.

3.5.1 LightVerSec Syntax

LightVerSec expresses authorization rules using pattern matching on semantic names. A *name pattern* consists of literal name components enclosed in quotes and pattern variables written as identifiers. Variables beginning with "_" denote temporary variables and are not bound. User-defined predicates (e.g., `isVer()`, `isSeq()`) may constrain components such as version or sequence numbers. A rule in LightVerSec has the form

`#DataPattern <= #KeyPattern`,

meaning that any data matching `#DataPattern` must be signed by a key whose name matches `#KeyPattern`, and any variables shared between the two patterns must agree.

3.5.2 Sample Trust Schema for NDNFit

The following LightVerSec schema expresses the write-access policy for user domains (both Alice and Bob) in NDNFit:

```
#KEY:    "KEY"/_/_/_version & {_version: isVer()}
#ndnfit: /"ndnfit"/#KEY
#domain: /"ndnfit"/"usr"
#anchor: #domain/#KEY <= #ndnfit
#entity: #domain/entity
#entityCert: #entity/#KEY <= #anchor
#data:  #entity/_seq & {_seq: isSeq()} <= #entity
```

This schema captures the following intent:

- **Constant naming convention:** The `#KEY` rule represents the common name suffix of a certificate, while the `#ndnfit` rule represents the pattern of name of the trust anchor in the "`/ndnfit`" domain.

- **User domains:** Each user domain resides under `/"ndnfit"/usr`, while the `usr` captures a user name pattern (such as "`alice`") that later policy checking would use. The rule `#anchor <= #ndnfit` expresses that the "`/ndnfit`" domain trust anchor is authorized to issue a certificate to a user domain such as "`/ndnfit/alice`".

- **Entities and their certificates:** Sensor or analyzer applications in a user domain are represented by the name pattern `/ndnfit/usr/entity`, and the rule `#entityCert <= #anchor` ensures that the entity certificate must be issued by the corresponding user domain anchor.

- **Application data:** The rule `#data <= #entity` allows an entity to sign only data whose names lie within its namespace and whose final component is a valid sequence number, so that Alice's analyzer cannot impersonate her sensor.

The resulting authorization policy illustrates the expressive power of semantic naming: Authorization follows naturally from the semantics encoded in the namespace, rather than requiring additional, separately maintained mappings between identities and access-control attributes or roles that most of today's systems rely upon.

It is also worth noting that in the inter-domain case where Bob shares his sensor data with Alice, Alice does not need to modify her trust schema to accept Bob's data. Because both trust domains run the same application and, therefore, adhere to the same semantic naming conventions, the relevant name patterns are already recognized by her schema.

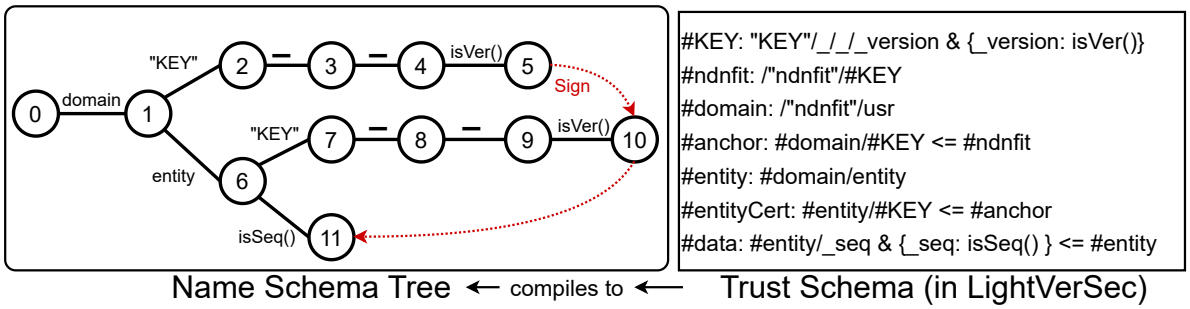


Figure 3.4: The NDNFit trust schema is compiled into a name schema tree to enable efficient policy checking. Each branch of the tree corresponds to a data name pattern, and the arrows between leaf nodes denote the authorized signing relationships derived from the schema. The node identifiers shown in the figure serve only as reference markers to aid explanation and have no semantic meaning within the schema itself.

3.5.3 Name Schema Tree

To support efficient policy checking, LightVerSec compiles the trust schema into a *name schema tree*, a trie-like structure whose edges represent literal components or pattern variables. Figure 3.4 gives an example of a name schema tree. The leaf nodes correspond to the data name patterns as defined in the rules, and each leaf attaches its corresponding rule pointer. When validating a data, the implementation walk through the tree to the matching leaf node and retrieves the expected key pattern, enabling efficient constraint checking.

For the sample NDNFit trust schema, the name schema tree organizes the name patterns so that, for example, a data named "`/ndnfit/alice/sensor/seq=123`", signed by Alice's sensor key "`/ndnfit/alice/sensor/KEY/id423/anchor/v=1`", is matched to the branch (0)(6)(11). When Alice's analyzer consumes this data, it first matches the data name to the schema tree and identifies the corresponding branch. Then it matches the signer's certificate name with the tree, obtaining the branch (0)(1)(6)(7)(8)(9)(10). Because the schema tree encodes authorized signing relationships as edges between leaf nodes, the analyzer can confirm that the sensor certificate is authorized to sign sensor data. The analyzer then follows the tree to

validate the signer's certificate itself, which is also authorized by the schema. Through this process, the analyzer concludes that the sensor data is an authorized publication.

3.6 Trust Information Base (TIB) Design and Communication API

The Trust Information Base (TIB) is the core module that records the trust anchor, certificates, and trust schema. The communication API we developed leverages TIB to provide authenticated and authorized production and consumption of named secured data. Because the trust plane treats named secured data as the basic unit of communication, the API relies on NDN (Section 2.5) for its native data-centric networking support. The rest of the section first describes the internal structure of TIB, then discusses how the API works with an illustrative example of inter-domain communications.

TIB Structure: Figure 3.5 illustrates how TIB sits at the boundary between applications and the NDN network. Internally, a TIB instance consists of four main components:

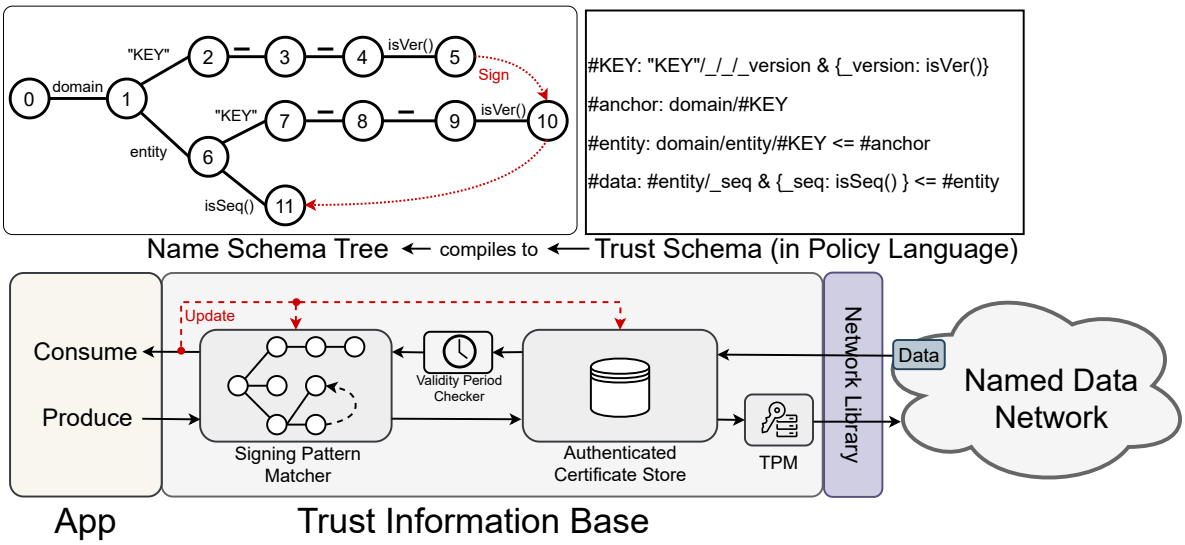


Figure 3.5: An example of TIB usage and the Produce/Consume API for NDNFit.

- Authenticated Certificate Store (store): A persistent storage that maintains all authenticated certificates known to the entity, including trust anchors, intermediate certificates, and entity certificates discovered during validation.

- Signing Pattern Matcher (matcher): A runtime structure that stores the name schema tree compiled from the trust schema and determines which entity certificate is authorized to sign a given data name.

- Validity Period Checker: A component that verifies the temporal validity of certificates encountered along a signing chain (e.g., checking `NotBefore`/`NotAfter`-style constraints), ensuring that expired or not-yet-valid credentials are rejected.

- Trusted Platform Module (TPM): A hardware or software module that securely stores private keys and performs signing operations on behalf of TIB, preventing private keys from leaving the trusted environment.

Produce/Consume API: The Produce/Consume API makes use of TIB to offer the following functions to applications.

- `Produce(name, content)`: Create a named secured data. The API queries the matcher and the store to determine the appropriate signing key based on schematized signing rules and certificate versions, computes the signature, and publishes the signed data onto the underlying NDN network.

- `Consume(name)`: Retrieve named secured data from the network and rely on TIB to authenticate it and verify its legitimacy before returning it to the application. During this process, the API automatically fetches any required certificates and CertLists for validation and inserts all validated certificates into the store. Additionally, if updated trust schemas are obtained, the API compiles them and refreshes the matcher accordingly.

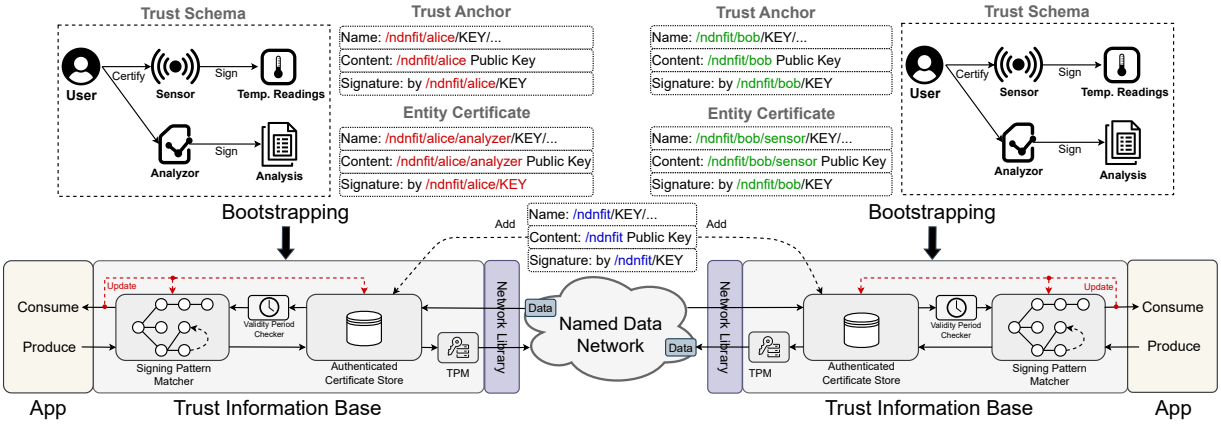


Figure 3.6: Bootstrapped TIBs of "`/ndnfit/alice/analyzer`" and "`/ndnfit/bob/sensor`" help the two entities securely produce and consume named data.

A Case Study of Secured Inter-Domain Communications: When Bob's sensor produces a new reading, *e.g.,*"`/ndnfit/bob/sensor/seq=123`", it passes the data name and content to its local TIB instance with the produce API, which searches the store for a signer whose name is authorized, according to the name schema tree, to sign this data name. In this example, the certificate named "`/ndnfit/bob/sensor/KEY/id423/anchor/v=1`" matches the pattern for `#entityCert` and is recognized by the matcher as an eligible signer for data under "`/ndnfit/bob/sensor/seq=123`". The produce API uses this mapping to select the sensor certificate, consult the TPM for the corresponding private key, and compute the signature on the data. Finally, the produce API puts the signed data available to the network and responds to the secured data as a Data packet upon incoming Interest packet.

Knowing the data name, Alice's analyzer fetches the data with the consume API. It fetches the desired data by sending an Interest. After receiving the data, the consume API uses the local TIB to begin the validation process. First, it extracts the key locator name "`/ndnfit/bob/sensor/KEY/...`" from the data and checks whether the corresponding certificate is present in the store. If the certificate is missing, the consume API also issues

an Interest to retrieve it. The retrieved sensor certificate, in turn, points to Bob's domain trust anchor "`/ndnfit/bob/KEY/...`". Following the CertList naming convention introduced in Section 3.4.2, the consume API then fetches the CertList of the "`/ndnfit/bob`" domain to discover the certificate "`/ndnfit/bob/KEY/id323/ndnfit/v=1`" issued by "`/ndnfit`", whose self-signed certificate is already installed as an authenticated external anchor (as part of the NDNFit installation). Data authentication then occurs by verifying the signing chain.

Then the consume API performs two checks. First, if the retrieved data is a certificate, it runs the validity period checker ensuring that the certificate is not expired or otherwise temporally invalid. Then it uses the matcher to evaluate each pair of (data name, signer name) to confirm that the signer is authorized to sign that data. If all checks are successful, the API concludes that the data is authentically signed and authorized by the trust schema. Then it adds the validated certificate chain to the store for future use and delivers the data name and content to the analyzer application.

CHAPTER 4

Security Bootstrapping

A new entity E_{new} needs to go through a bootstrapping process to configure TIB with a trust anchor, trust schema, and certificate, so that E_{new} can securely produce and consume named data. Although out-of-band configuration is always referred to as the last resort, it is necessary to develop a systematic process to remotely bootstrap these parameters into new entities over unsecured network. This chapter proposes a four-step process that provides a foundation for designing new bootstrapping solutions. We then present bootstrapping solutions for two applications based on this process. Finally, we compare with related works and point out their limitations.

4.1 A Four-Step Process

Step 1: Mutual Authentication: Both the bootstrapper B and E_{new} must authenticate each other before any credential exchange based on common trust relationships. In local bootstrapping scenarios, this authentication can leverage physical proximity assumptions, while in remote scenarios, it requires establishing common trust relationships through existing authentication infrastructures, such as Web PKI and DNSSEC. The result of each-way authentication can be a secured session key, token, or certificate used in subsequent bootstrapping communications.

Step 2: Trust Anchor Installation: Once mutual authentication is established, B provides E_{new} with the trust anchor(s) of the target system securely. The trust anchor serves

as the root of trust for the entire system and enables E_{new} to authenticate all subsequent communications within the trust domain. This step can be performed through the secure channels established in the previous step.

Step 3: Certificate Issuance: E_{new} obtains a certificate for its named public key that establishes its identity within the trust domain. This involves E_{new} generating a key pair, creating a certificate request, and having B (or a designated certificate authority) issue a certificate that binds the assigned name of E_{new} to its public key. It is worth noting that the certificate obtained during bootstrapping has a finite lifetime, so E_{new} must refresh this certificate in the future.

Step 4: Trust Schema Distribution: Finally, E_{new} obtains the trust schema(s) that define the security policies for the system. These can be distributed along with trust anchors or fetched separately using established naming conventions. Bootstrapping only installs the initial trust schema for E_{new}, who still needs to periodically learn the latest trust schema after bootstrapping.

4.2 Hydra Bootstrapping

To ground the aforementioned process in practice, we next consider Hydra, a federated data storage system that needs to remotely bootstrap the storage servers and users. After introducing Hydra, we then explore the design space of each step in remote bootstrapping. Finally, we present *Cornerstone* as our solution to Hydra bootstrapping.

4.2.1 Hydra Overview

To illustrate the requirements of remote bootstrapping in practice, we consider Hydra [PWB22], a distributed and federated data storage system for sharing genomic data across multiple campuses. Hydra operates under a federated trust model in which different organizations contribute storage servers, while a Network Operating Center (NOC) establishes and main-

tains the trust domain for the system.

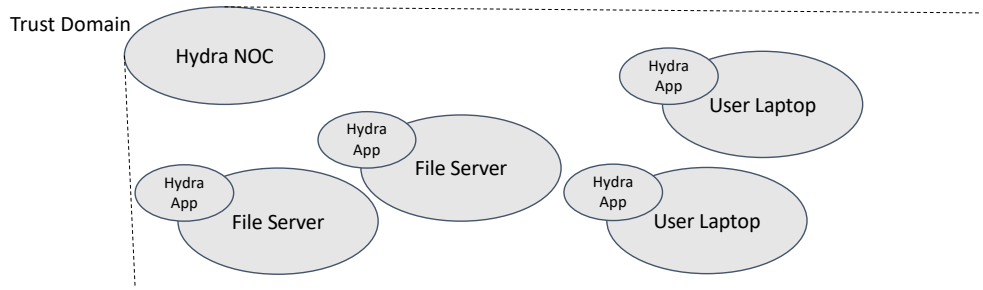


Figure 4.1: The entities in the Hydra trust domain. Not shown here are the trust relationships among the entities.

Figure 4.1 shows a simplified view of the Hydra trust domain. The NOC self-signs a certificate as the system trust anchor and initializes the trust schema based on administrator input. Membership is controlled via two lists: the *server list*, containing the DNS names of contributed servers, and the *user list*, containing the email addresses of authorized users. Although Hydra assumes that organizations have full control of their servers and that authorized users are trustworthy, communication between servers, users, and the NOC occurs over unsecured networks.

4.2.2 Exploring the Design Space

Leveraging Existing Authentication Systems. In remote scenarios, mutual authentication must be based on external authentication systems that both parties already trust, and the trust relationship between the trust domain and E_{new} must be outside the band *e.g.,* defined by the domain controller (such as whitelist of identifiers). Assuming that E_{new} has a unique and semantically meaningful identifier on the Internet, this identifier may also carry sufficient information to help the trust domain controller assign E_{new} a name that is meaningful in the specific application context, as we show later. A practical approach is to utilize existing authentication systems [ROS23, BSP08, BHM19a, SBJ14] on the Internet

that associate semantic identifiers such as emails and DNS names with defined trust relationships. These authenticated semantic identifiers from today's Internet can serve as a starting point to bootstrap remote entities.

Distributing Trust Anchor via A Third Party. Current practice in Web PKI distributes trust anchors through software update channels, thus combining B authentication with the software distributor. Alternatives such as DNS-based Authentication of Named Entities (DANE) [HS12] can bind trust anchors to DNS names using TLSA records.

Certificate Issuance. Step 3 requires E_{new} to obtain a certificate in the target trust domain. Once authentication and trust anchors are in place, ACME-like protocols [BHM19a, ZWS20] can be used to automate certificate requests and issuance.

Trust Schema Distribution. Finally, Step 4 concerns the delivery of trust schemas. Two models are possible: (i) E_{new} retrieves the trust schema as secured named data by naming convention, or (ii) B bundles the trust schema with the software package. The latter is simpler, but requires new releases when schemas change.

4.2.3 Cornerstone Overview

As Figure 4.2 shows, the security bootstrapping starts with *mutual authentication* between the remote entity and the NOC. Cornerstone makes use of the trust relations that already exist in today's Internet operational environment to allow both sides to authenticate each other. A new user or server of Hydra authenticates the NOC through the Hydra software distribution chain (Section 4.2.4), from which E_{new} can also install the Hydra trust anchor and the trust schema.

Afterwards, the new entity runs the *Name Authentication and Assignment Protocol (NAAP)* (§4.2.5) with the NOC to request authentication and name assignment using its *existing identifier* (§4.2.6), which could be an X.509 certificate owned by a Hydra server, or an email identifier assigned by some recognized external identity provider to a new Hydra

user. For example, a server with domain name "`node1.medicalx.com`" can be authenticated with its X.509 certificate. Alice, a user with account "`alice`" at "`medicalx.com`", can be authenticated with her *OpenID Connect* ID token signed by "`medicalx.com`". During the authentication process, each E_{new} uses a temporary key pair to uniquely identify itself from other new entities.

After the authentication step, the NOC assigns an name to E_{new} based on its external identity used for authentication (Section 4.2.6). The name assignment rules are configurable by the NOC deployment. In the example above, if Hydra runs at the name prefix "`/hydra`", the server may obtain the name "`/hydra/servers/com/medicalx/node1`" and Alice may obtain "`/hydra/users/com/medicalx/alice`". The assigned name is given to Alice in the format of *Proof-of-Possession (PoP)*, a NOC-signed data that binds Alice's temporary public key to her assigned name.

Lastly, Alice uses the NDNCERT [ZWS20] protocol to interact with the Hydra Certificate Authority (CA), an entity delegated by the Hydra NOC for certification. Alice presents the PoP signed by the NOC to the CA as her name assignment. Upon verifying Alice's PoP, the CA issues Alice the final certificate that she can use in Hydra (Section 4.2.7).

4.2.4 Installing Trust Anchor and Trust Schema

Hydra embeds the trust anchor and trust schema in its software distribution, therefore, all users and servers will install the same trust schema. By doing so, we assume that all Hydra users and servers can receive these security components securely through the existing software distribution channel, such as GitHub, which can provide an authenticated communication channel that Hydra administrators use to distribute Hydra software package. As a result, when Hydra users and servers receive the software package, they can be confident that the embedded trust anchor and trust schema are authentic and have not been tampered with.

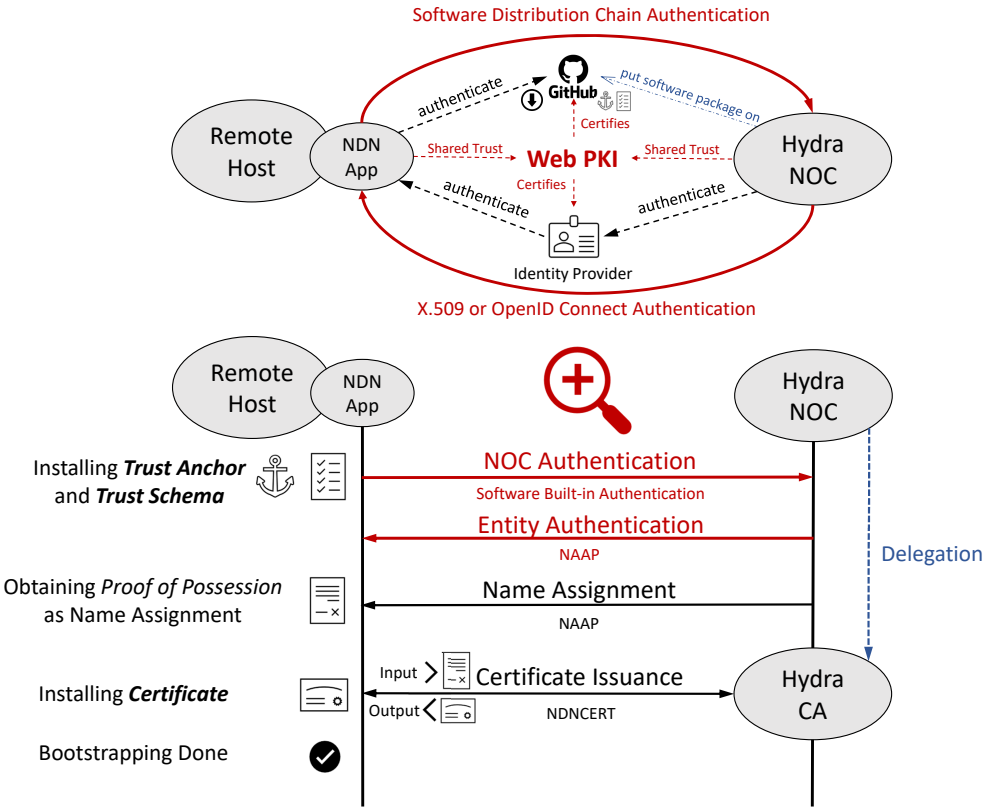


Figure 4.2: Overall workflow of Cornerstone. The magnifier focuses on the mutual authentication process and the upper part of this figure reveals more details of this two-way authentication.

4.2.5 Authenticating New Hydra Entity

Cornerstone leverages *existing* identifier authentication systems to authenticate server and users. If a storage server has a DNS name, existing authentication systems include DNSSEC and X.509 certificates issued by certificate authorities [BSP08]. For user authentication, Cornerstone uses identity verification protocols such as OpenID Connect (OIDC) [SBJ14]. OIDC allows an application to verify the identity of users through identity providers such as Google, Facebook, or GitHub. Hydra NOC can request OIDC authentication from user's identity provider, and this process involves validating a provider-signed token, which indicates the user's account name and authentication status.

As Figure 4.3 depicts, Cornerstone's new overall entity authentication workflow is a two-phase protocol NAAP (Name Authentication and Assignment Protocol), executed between the new entity, which is either a new user or a new server to be bootstrapped, and the Hydra NOC. The first phase is Request and Response (A), where the new entity requests NOC authentication, providing the necessary parameters, including a public key to uniquely identify the new entity during the bootstrapping process. The second phase is Identity Proof Request and Response (P), where the new Hydra entity proves its identity to the NOC and receives a PoP which contains its name assignment (discussed in Section 4.2.6).

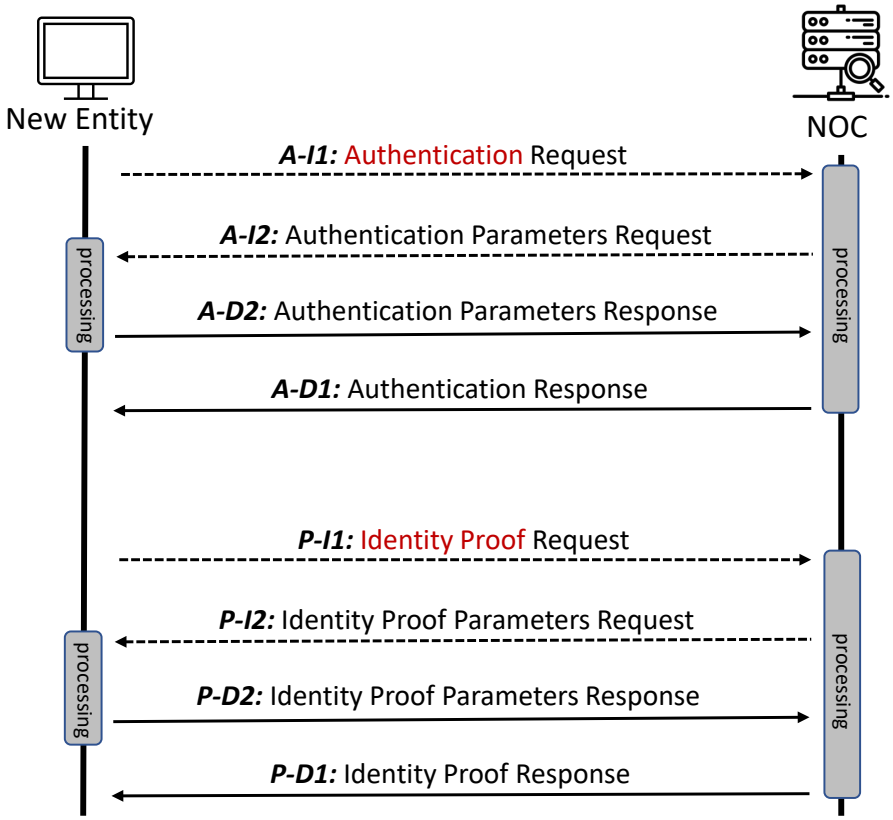


Figure 4.3: NAAP Overview

Each phase consists of the following Interest-Data packet exchanges:

- *I1*: an Interest sent by the new entity to notify the NOC[1].
- *I2*: an Interest sent by the NOC to the new entity to fetch the necessary parameters.
- *D2*: a Data sent by the new entity in response to *I2*, carrying parameters requested by the NOC.
- *D1*: a Data sent by the NOC in response to *I1*, carrying the result of this phase.

The rest of this section will describe the server authentication workflow and user authentication workflow, respectively. Detailed packet exchanges are presented in Appendix .1.

Server Workflow: In the server authentication flow, it is assumed that the server has a valid X.509 certificate and intends to use the same key pair to bootstrap in Cornerstone. The NOC verifies the X.509 certificate chain and authenticates the server entity by requiring it to sign a random number using the certified key pair.

Specifically, in *A* phase, the authentication parameter *A-D2* is the server's X.509 certificate chain. Then in *A-D1*, the NOC generates a random number and requires that the signature of the new server be sent back in *P-D2*. Upon receiving the signature, the NOC examines that the X.509 certificate chain is properly signed by a known CA and that the random number is properly signed by the certified key.

User Workflow: In the user authentication workflow, a new user presents an identity from an OIDC provider known to the NOC (discussed shortly), proves ownership and is authenticated. In *A* phase, the authentication parameters *A-D2* are the OIDC provider's name, the user's username at the provider, and a newly generated public key. Then in *A-D1*, the NOC gives a URI that the user can use to perform an OIDC authentication. Generally, this URI embeds the application's credentials registered at the OIDC provider. In *P* phase, the *P-D2* is the OIDC ID token provided by the OIDC provider. Receiving *P-D2*, the NOC

[1] *I1* contains the new entity's temporary name prefix in its content. The NOC uses this prefix to construct *I2* and fetch parameters. For example, a user Alice connected to the MedicalX network can use the name prefix "`/Medical/Alice/HydraBoot`" in this authentication.

examines whether the OIDC ID token is properly signed by the key of the user's OIDC provider.

Automation: The automation of server authentication is based on two factors. First, the Hydra NOC has established trust relationships with CAs through out-of-band installation of CA root certificates. This requirement is typically fulfilled by the operating system's software distribution chain, as modern operating systems come pre-installed with CA root certificates. Second, servers need to be able to request X.509 certificates and obtain them automatically. To address this, Cornerstone leverages Let's Encrypt [ABC19], which provides a free and automated X.509 certificate issuance service for servers. The automation of authentication relies on Hydra NOC having pre-established trust relationships with the relevant identity provider and Web PKI.

4.2.6 Naming New Hydra Entity

Name assignment occurs between the NOC receiving *P-D2* and sending *P-D1*. Note that the NOC only assigns a name to new entities that are authorized by the user list and the server list. To simplify the writing of trust schema rules and ensure semantically meaningful names in the system, it is desirable to assign names that leverage existing identifiers used for authentication.

Based on our communications with Hydra administrators, we have learned that they prefer to manage trust relationships with organizations groups. Therefore, the semantics of DNS names and email addresses are suitable for reuse in the name assignment process. In the assignment of names for Hydra entities, we split the domain names into multiple name components and concatenate them with a name prefix that indicates whether the entity is a server or a user. This approach allows us to preserve the semantics of existing identifiers while providing unique and semantically meaningful names within the system.

As shown in Figure 4.4, the entity naming convention is configurable by administrators

programming name assignment functions using Cornerstone's predefined rules. For example, the rule *New* creates a base name "`/hydra/servers`", while the *DomainSplit* rule converts a DNS name "`node1.medicalx.com`" into an name "`/com/medicalx/node1`". The *Concat* rule then adds the converted name after "`/hydra/users`", resulting in the name "`/hydra/servers /com/medicalx/node1`". Similarly, by combining the rules *New*, *EmailSplit*, and *Concat*, Alice's email address can be converted into the name "`/hydra/users/com/medicalx/alice`".

The programmable naming rules in Cornerstone enable Hydra administrators to define entity naming conventions without dealing with the low-level implementation details. For advanced administrators who wish to define custom name conversion rules, Cornerstone also provides APIs that allow the design of customized naming rules. This flexibility enables administrators to tailor the naming process according to their specific requirements.

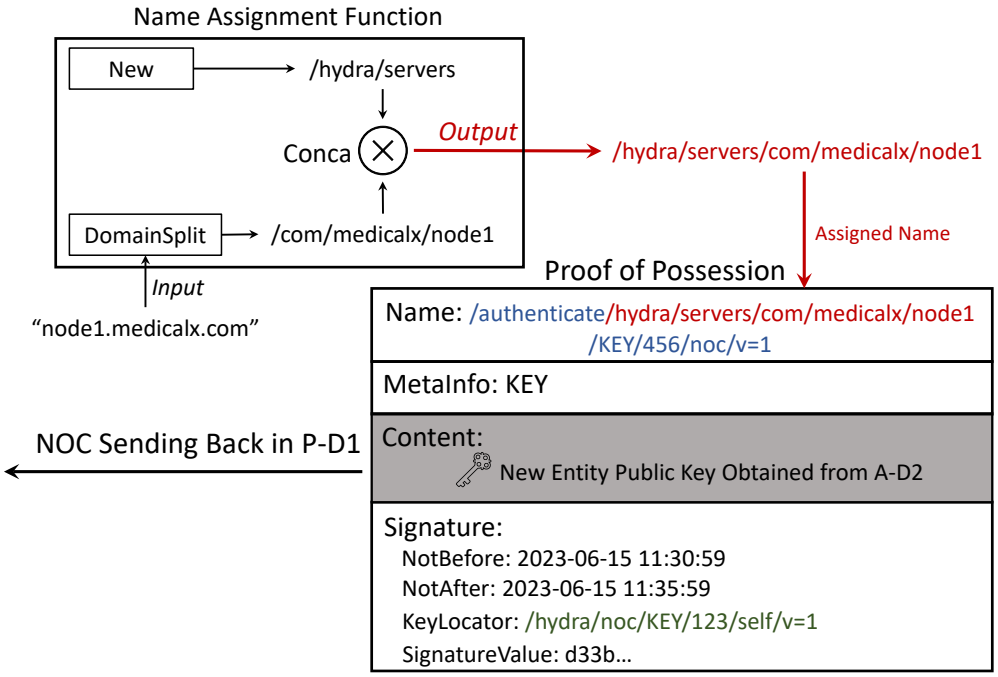


Figure 4.4: Example of name assignment based on domain names

Proof-of-Possession: After the NOC assigns an name to a new Hydra entity, it generates a Proof of Possession (PoP) by signing a Data with its private key, and sends PoP back in

Data *P-D1*. The PoP can be considered as a temporary certificate used to bind the entity's public key with the name which the NOC assigned to the new entity. The content of the Data packet includes the public key of the new Hydra entity obtained during the authentication process (Data *A-D2*).

The PoP follows a naming convention that starts with the keyword prefix "`/authenticate`", followed by the assigned entity name, the identifier of the corresponding public key, the issuer's information and a version number. In the example of Figure 4.4 in which the NOC assigns the name "`/hydra/servers/com/medicalx/node1`" to a new server with the public key identifier "`456`", the NOC signs a PoP with the name "`/authenticate/hydra/servers/com/medicalx/node1/KEY/456/noc/v=1`". The component "`noc`" indicates that the PoP is issued by the NOC and "`v=1`" represents the version number. PoP also includes a validity period that restricts the lifetime of key-name binding. Within the validity period of the PoP, the new entity must complete the security bootstrapping process by requesting a certificate from a Hydra CA (Section 4.2.7) using this PoP. If the entity fails to complete the security bootstrapping within the validity period, a new PoP must be obtained through re-executing the entire protocol. Note that the public key contained in the PoP is supposed to be an ephemeral key that serves to uniquely identify the new entity during the security bootstrapping process. It may not be the same public key that will be included in the final certificate issued to the entity. In summary, the PoP serves as a proof of ownership for the assigned name and facilitates the security bootstrapping process by establishing a binding between the entity's public key and its assigned name.

4.2.7 Certification

One key aspect that sets Cornerstone apart from previous bootstrapping solutions is the decoupling of authentication and naming from certification. This is achieved by allowing Hydra NOC to delegate the certification process to NDNCERT [ZWS20], a separate CA entity within Hydra trust domain. Hydra NOC configures the Hydra CA in the trust domain

setup time, delegating a name prefix to host NDNCERT CA, signing the CA certificates and running the CA software.

A new Hydra entity learns the Hydra CA prefix by naming convention. Once the new Hydra entity has obtained its PoP, it runs the NDNCERT protocol. During this challenge, the CA validates the entity's PoP and requests the entity to provide proof of ownership of the corresponding private key. Upon successful verification, the CA issues the final certificate for the entity under the name appointed by the PoP.

4.3 Ownly Bootrapping

This section first presents a brief overview of the target application Ownly, followed by the technical challenges we faced in bootstrapping Ownly users. We then present our developed bootstrapping solution.

4.3.1 Ownly Overview

Ownly [YMP24] is a fully user-controlled, serverless collaborative editor that runs in the browser[2] and supports shared text editing and whiteboard drawing. The shared text and drawing files are further organized into directories. For simplicity, we refer to an Ownly instance as a workspace. From the network's perspective, Ownly operates in a true peer-to-peer fashion. Users communicate directly with each other, with only packet forwarders between. Ownly names user edits with the naming convention combined with the Ownly workspace name, username, and a per-user sequence number (as shown in Figure 4.5). Therefore, users can securely exchange their edits to the shared directories by (i) producing local edits as named secured data, (ii) joining a Sync group (Section 2.5.3) and notifying state updates, serving data to the corresponding incoming Interests and (iii) fetching and validating newly

[2] `https://ownly.work`

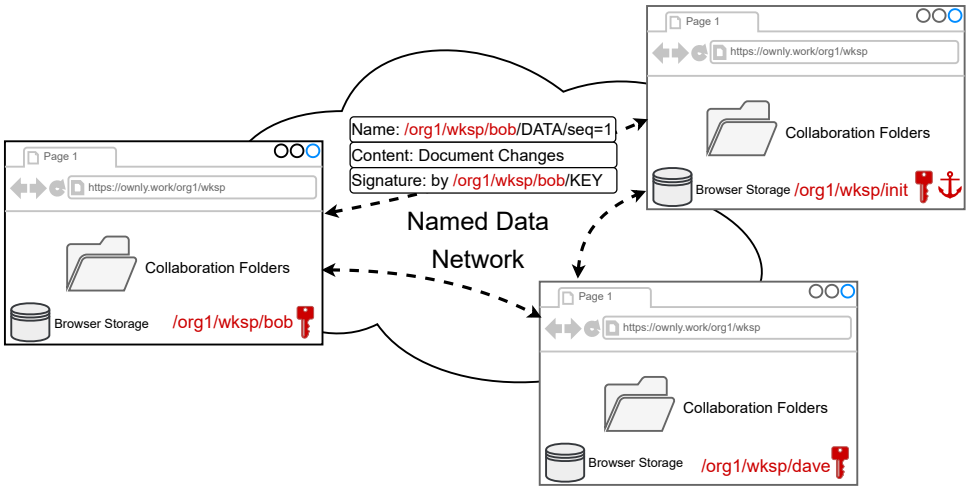


Figure 4.5: In Ownly instance "`/org1/wksp`", two users and the initiator collaboratively edit documents in a shared folder. Document changes are named secured data and are exchanged in a named data network.

learned data from the network.

Anyone can create an Ownly workspace and invite people by email address. The workspace creator (initiator) is the only one able to invite others, but everyone in the group has the same read and write permissions. All users can create directories and edit the shared files within them. Upon creation, the workspace is named in its initiator's namespace and forms a trust domain with the initiator as the domain controller. For example, Alice owns the DNS name "`foobar.org`" ("`/org1`" for short) and initiates an Ownly workspace "`/org1/wksp`" as its domain controller. As an initiator, she invites Bob by his email address "`bob@example.com`" ("`/bob`" for short), and Bob uses the identity "`/org1/wksp/bob`" to sign his produced data. The design goal of Ownly bootstrapping is to enable Alice to remotely provision Bob with the trust anchor of the workspace domain "`/org1/wksp`", a certificate for "`/org1/wksp/bob`", and the domain trust schema. Throughout the rest of this section, we assume that collaborative users have already exchanged their self-signed certificates and cross-signed through out-of-band channels.

4.3.2 Addressing Technical Challenges

A seemingly feasible approach is to directly adopt the approach of Hydra bootstrapping (Section 4.2), that is, to set up a dedicated domain controller to bootstrap new users that may occur at any time. However, this approach contradicts with the overall design goal of Ownly that users communicate directly with each other without application intermediaries. Packet-level exchanges, if possible, should be constrained to only between Alice and Bob. The initial trust schema, as they are commonly shared by all workspaces (*e.g.,* all users can only produce data under their own sub-namespaces), can still be embedded with the Ownly application package. However, in Ownly bootstrapping, there are two technical challenges.

Distributing Trust Anchors: Hydra is a single instance application that everyone can only recognize and join the Hydra trust domain. However, in Ownly, as anyone can create its own workspace, there are multiple trust domains. Users can join multiple workspaces in parallel, each of which has its own trust anchor. As a result, embedding domain (such as "`/org1/wksp`") trust anchors in the software package is impractical.

We address this challenge by leveraging the fact that collaborative users already trust relationships before collaboration. For example, Alice and Bob have previously mutually authenticated by their names "`/org1`" and "`/bob`". As the owner of "`/org1`", Alice controls the trust domain "`/org1`", where she may manage various services. Similarly, Bob owns the trust domain "`/bob`", which he uses to manage his personal devices. Figure 4.6 illustrates the associated trust domains and their relationships in this example. Consequently, Bob can authenticate the "`/org1/wksp`" domain controller (the initiator) by verifying its certificate issued by "`/org1`", then securely fetch the trust anchor from it.

Supporting Asynchronous Workflow: The network reachability' of the user is intermittent and unpredictable. The initiator and invited user are not always reachable, which means that network communications during bootstrapping can be asynchronous.

This challenge is addressed by an application-agnostic in-network data repository (*Repo*)

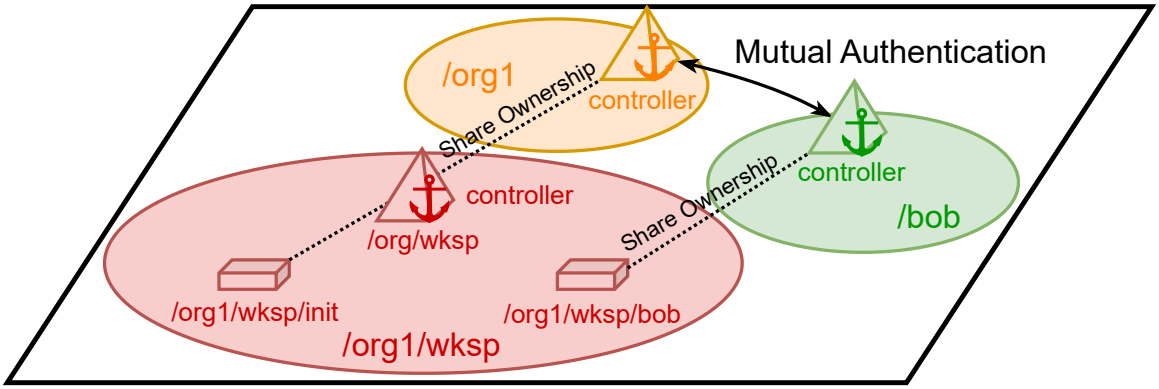


Figure 4.6: The relationships of trust domains for Ownly user bootstrapping: the workspace initiator Alice and the new user Bob each manage their own trust domain and have mutually authenticated and cross-signed their respective certificates. Alice uses "`/org1/wksp/init`" as her workspace user name, while also owning the trust anchor for "`/org1/wksp`". The goal of Ownly bootstrapping is to bring "`/bob`" into the workspace domain as "`/org1/wksp/bob`".

that stores named secured data produced by users and serves them within the network. During bootstrapping, both the initiator and the invited user publish their respective certificates and bootstrapping-related data to Repo, enabling the other party, whenever they next become reachable, to retrieve these objects directly by name. A detailed description of the design and operation of the Repo is provided in Chapter 10.

4.3.3 Workspace Initialization

Figures 4.7 illustrate the key derivation necessary to create a workspace from "`/org1`" and bootstrap "`/bob`" into it. When creating workspace "`/org1/wksp`", Alice (via the Ownly application) generates a new pair of keys for "`/org1/wksp`", produces a self-signed certificate for the new public key (referred to as the workspace trust anchor), issues it a certificate (referred to as the workspace precertificate) with her "`/org1`" key, and creates a CertList (Section 3.4.2) linking the workspace trust anchor to "`/org1`". An initiator identity "`/org1/wksp/init`" is also generated at workspace creation time and is certified by the trust anchor.

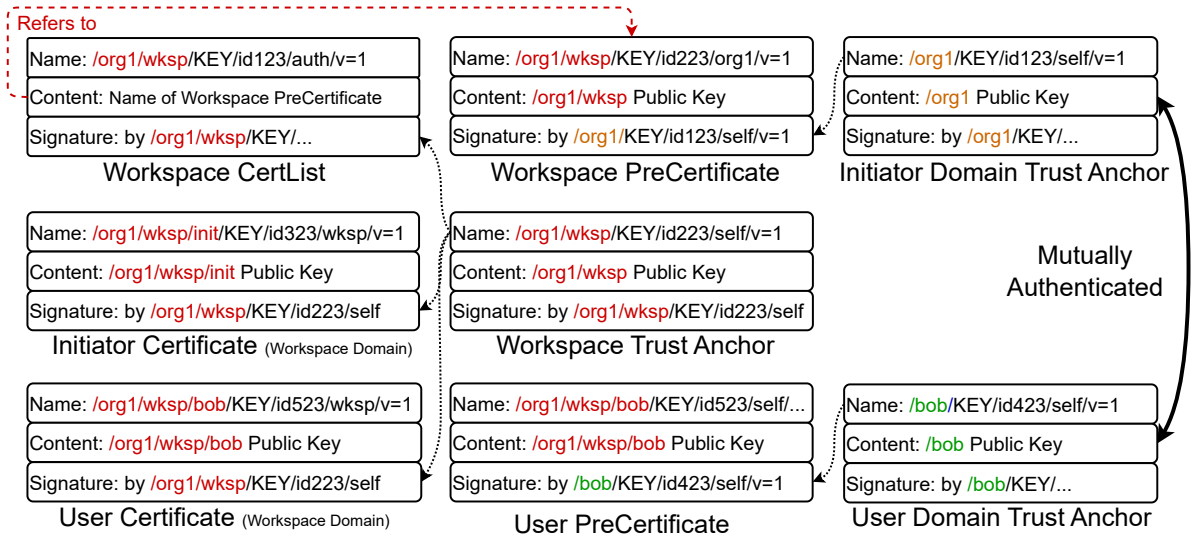


Figure 4.7: Certificate chains involved in Ownly bootstrapping. The initiator and invitee have already mutually authenticated by their own names.

After the workspace instance is established, Alice publishes all newly generated workspace data (the workspace precertificate, its trust anchor and CertList) to the Repo. Repo serves as an in-network data store for these certificates for Bob to retrieve later.

4.3.4 Invitation New Users

To invite Bob, Alice shares a URL to Bob out-of-band. The URL includes the pointer to the Ownly application, and the name of the trust anchor key in the workspace "`/org1/wksp/KEY/id223`". Bob initiates the bootstrapping process once he receives a URL. It starts by loading the application and then fetching the trust anchor via "`/org1/wksp/KEY/id223/self`" and its CertList. The CertList points to the workspace precertificate, which can be authenticated using the "`/org1`" trust anchor. Once the workspace CertList is authenticated, the workspace controller is authenticated, and Bob installs the retrieved trust anchor "`/org1/wksp/KEY/id223/self/v=1`" in his local TIB (Section 3.6).

Bob then generates a new key pair for "`/org1/wksp/bob/KEY/id523`" and signs the public

key with his personal identity "`/bob`". This newly signed certificate is referred to as a user precertificate, which is an intermediary credential during bootstrapping. Bob publishes his precertificate to the Repo, which stores the data for Alice's later retrieval. Then Bob no longer needs to wait for Alice and can go offline.

When Alice comes online, she fetches the precertificates of all invited users (including Bob), authenticates them by their personal identity (such as "`/bob`"), signs the user certificate with the trust anchor, and publishes the certificates back to Repo. Eventually, Bob comes online again to fetch the final user certificate to finish bootstrapping.

4.3.5 A Summary of Operational Workflow

Figure 4.8 summarizes the asynchronous workflow of Ownly bootstrapping. In this process, only the actor named in each phase needs to be online.

Phase 0: Preparation. Ownly application developers define the application trust schema, embed it in the software package, and host it under a well-known URL. The initiator Alice creates the workspace and publishes the workspace trust anchor and its CertList to Repo.

Phase 1: Initiator Inviting User. The initiator produces an invitation URL that contains the web location to load the Ownly application and the workspace trust anchor key name. Alice shares this URL with the invitee (Bob) through an out-of-band communication channel.

Phase 2: Invitee Authenticating Initiator, Obtaining Trust Anchor and Trust Schema. Upon receiving the workspace URL, Bob opens it in the browser, loads Ownly and its embedded trust schema, then authenticates the workspace controller and installs the anchor by fetching "`/org1/wksp/KEY/id223/self/v=1`" and "`/org1/wksp/KEY/id223/auth/v=1`", verifying the chain to "`/org1`", and installing the workspace anchor in his TIB. Bob then generates a new key pair "`/org1/wksp/bob/KEY/id523`", signs the public key with his personal key "`/bob`", obtains a precertificate, and publishes it to Repo. Bob may then go offline; Repo durably retains the precertificate.

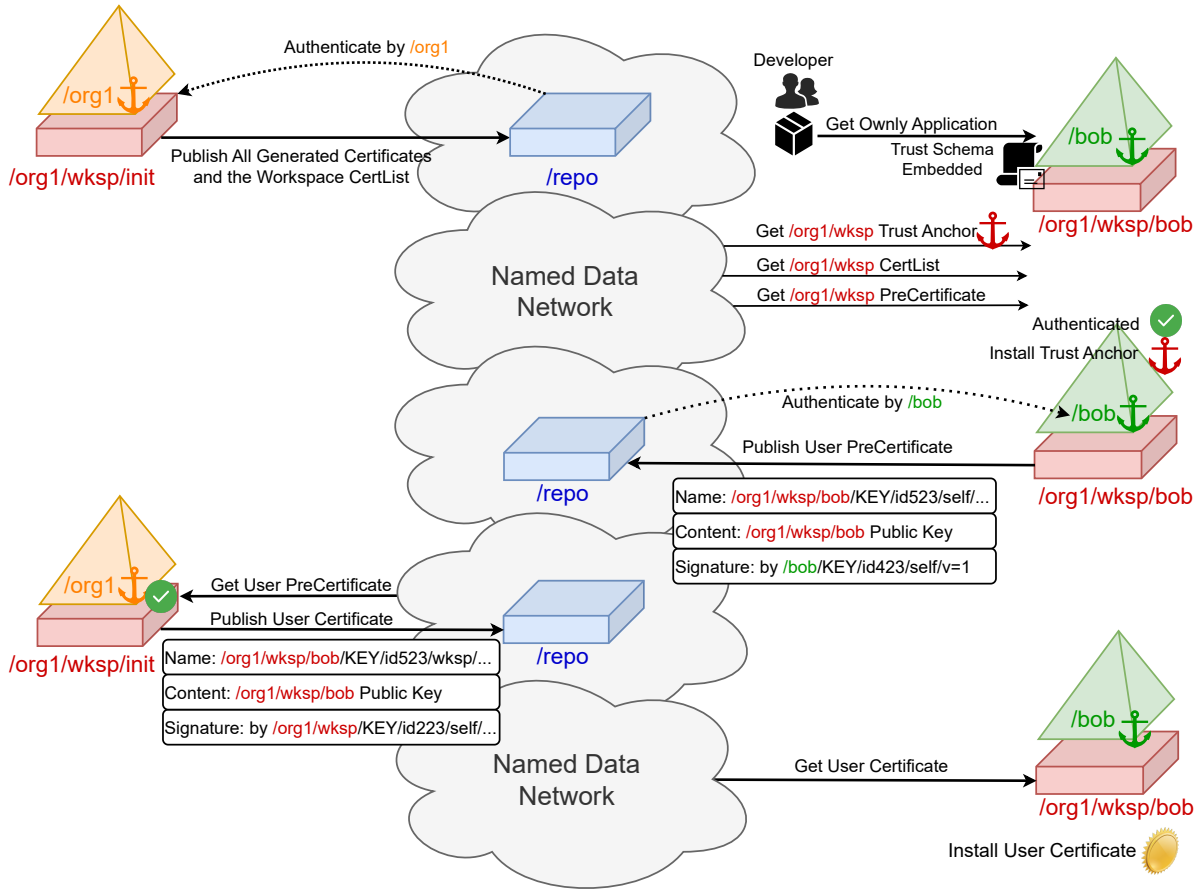


Figure 4.8: The asynchronous workflow of Ownly bootstrapping using Repo, which is available at "`/repo`" and authenticates its users by their personal identity keys.

Phase 3: Initiator Authenticating Invitee, Issuing Certificate. When next online, Alice retrieves Bob's precertificate from Repo, verifies its signature by "`/bob`" and the chain to Bob's personal key. She then issues a user certificate for "`/org1/wksp/bob/KEY/id523`", signed by the workspace trust anchor, and publishes it to Repo.

Phase 4: Invitee Obtaining Certificate. In his next online period, Bob pulls his user certificate from Repo, verifies it against the installed workspace anchor, and installs it in his TIB. At this point, Bob is fully bootstrapped: he can use the Produce/Consume API (Secion 3.6) to securely participate in collaboration in the "`/org1/wksp`" domain.

4.4 Related Work

Existing bootstrapping solutions have focused primarily on *local deployment scenarios*, where physical proximity or direct connections can be leveraged to provision trust anchors, trust schemas and certificates. This section reviews representative approaches and highlights their key characteristics.

4.4.1 Leveraging Direct Connection

In these approaches, the bootstrapper B establishes a physical or remote connection to the entity E and installs the necessary security credentials. Representative examples include NDNFit, the NDN Testbed, DCT, and PION.

NDNFit [SBL16] is a mobile health application (mHealth) built on NDN. User-owned sensors and analyzer applications securely produce and consume health data authorized by trust schemas. NDNFit users serve as the trust anchor of their own devices: they directly attach to entities, install a self-signed certificate as the trust anchor, and configure device/application certificates and trust schemas.

NDN Testbed [Nam22] is a global NDN overlay network with a dozen routers. Routing daemons are configured as entities that must securely exchange data from the routing protocol [HAA13]. To operate securely, each router is provisioned out-of-band with the Testbed trust anchor and a routing-specific trust schema. When requesting a certificate from the Testbed CA, routers authenticate themselves using predistributed credentials (e.g., a password or token) that are assigned by the Testbed operators. Once authenticated, the CA issues a certificate that binds the router's name to its public key, enabling the router to securely produce routing updates.

DCT (Data-Centric Toolkit) [Nic19, Nic21] provides a domain-specific language to specify trust schemas and includes tools for managing IoT identities and certificates. DCT introduces *identity bundles*, which bundles an entity's certificate, the trust anchor of the

target trust domain, and the corresponding trust schema. The IoT controller under user's control bootstraps IoT devices by directly attaching to the devices and distribute identity bundles.

PION [PSM22] proposes a password-based onboarding protocol for NDN-based IoT systems. The protocol assumes a direct connection between the device and an authenticator - usually a smartphone application - so that the two can run SPAKE2 [Lad23], a password-authenticated key exchange, using a short out-of-band password pre-installed on the device. Through this direct link, the authenticator derives a strong shared key with the device, encrypts a temporary certificate, and provisions it to the device. The authenticator is assumed to have an existing trust relationship with a CA. Once the device demonstrates possession of the temporary certificate, it uses NDNCERT [ZWS20] to obtain its long-term certificate from the CA.

4.4.2 Leveraging Physical Proximity

Other approaches use physical proximity as the basis of authentication, where keying material is pre-shared between E and B through short-range channels (*e.g.,* scanning a QR code). Examples include Sovereign and NDNViber.

Sovereign [ZYM22] is an NDN-based smart home framework in which users retain control of the system trust anchor for all IoT devices in their home. Device bootstrapping [LZW19] begins by scanning a printed QR code, which includes a manufacturing credential and a symmetric key. Assuming knowledge of the manufacturer's public key and physical line-of-sight access, the smartphone authenticates the device and provides it with the home trust anchor, device certificate, and trust schemas.

NDNViber [RPA20] leverages vibration as a physical side channel for authentication. An IoT controller uses its vibration motor to transmit trust anchors, trust schemas, and a verification code, while the IoT device receives these signals through its accelerometer and

decodes them using on–off keying (OOK) modulation. Then the device is authenticated by the verification code. The physical proximity requirement within $1-1.5$ cm provides a built-in form of authentication: the device can trust that only a co-located controller could have generated the vibration pattern, while the controller can verify that the device is physically present and capable of interpreting the encoded signals.

4.4.3 Limitations

Aforementioned bootstrapping mechanisms [SBL16, Nam22, HAA13, Nic19, Nic21, ZYM22, RPA20, PSM22] assume physical proximity, pre-installed secrets, or shared administrative control—assumptions that do not hold in a more general application scenario when E_{new} and B are remote. This reliance on contextual trust often yields designs that authenticate only the joining entity, implicitly treating the B as trustworthy because its legitimacy is embedded in the application context (*e.g.,* line-of-sight or co-location) [ZYM22, RPA20].

CHAPTER 5

CertRevoke

A deployed trust plane must be continuously maintained to reflect evolving trust relationships in cyberspace. This maintenance spans several distinct activities. First, certificates and trust anchors require periodic renewal, which can be viewed as a repeat of the bootstrapping process. Second, trust schemas, as named secured data that encode trust policies, must evolve as application design or trust relationships change; maintaining them requires a mechanism for publishing and subscribing to schema updates. Third, certificate revocation is needed to remove unwanted or compromised keys from legitimate operations.

Although automated certificate renewal mechanisms are relatively well understood and schema updates - as yet another named secured data - can naturally be supported by the Produce/Consume API, a systematic understanding of the certificate revocation design remains lacking. In particular, three research questions about how to express and distribute revocation information remain unanswered. This chapter addresses these open problems.

RQ1: Which entity can legitimately revoke a certificate? Each certificate has an issuer and a verification chain that eventually terminates at a trust anchor. If the certificate issuer (*e.g.,* a site CA on the NDN Testbed [Nam22]) is not itself the trust anchor, it must be clarified what role the trust anchor can or should play in the revocation process.

RQ2: What procedures are needed to revoke a certificate? Certificate revocation requires publishing a revocation record. We need designs that specify *who can revoke which certificate and when*, as well as procedures to securely create these records.

RQ3: How do data consumers learn the revocation status of certificates? To enable

data consumers to learn the validity status of the revoked certificate, the revocation design must consider the authenticity and timeliness of the revocation record without incurring high communication costs in the data validation process.

In this chapter, we first clarify the basic situations in which certificate revocation is needed and then present our CertRevoke design that answers these three questions. We also provide preliminary evaluations of its effectiveness.

5.1 When We Need Revocation

5.1.1 Invalidating Certificates

A certificate represents the trust domain controller's endorsement of a name-key binding within a validity period. However, there may be times when this endorsement may need to be terminated before the validity period expires. For example, when the entity renews its certificate before the expiration time, when the entity's private key gets compromised, or when the trust domain controller finds that it issued a certificate to a wrong entity by mistake. All of the above cases require the issued certificates to be invalidated. There exist several possible approaches to invalidate a certificate.

Short Certificate Lifetime: If a certificate is issued with a short validity period, it automatically becomes invalidated upon expiration, reducing the chance of being compromised or causing great damage even in case it becomes compromised. The validity period of a certificate may be a few days or even a few hours. The drawback of this approach is the increased workload of certificate issuers, which increases reversely proportional to the certificates' validity period lengths. However, the recent success of Let's Encrypt [ABC19] in the existing CA market suggests that certificate renewal tasks can be effectively managed through automation, such as using the Automatic Certificate Management Environment (ACME) protocol [BHM19b].

Using short-lived certificates can be an effective engineering alternative to certificate revocations for leaf or edge (*e.g.,* user) certificates. However, renewing higher-level certificates may still incur substantial cost, as a change to a high-level certificate leads to re-issuing the certificates for all the entities beneath it. Therefore, upstream certificates in a trust chain of nontrivial length may need a relatively long validity period (*e.g.,* on the scale of multiple months). The flip side of a longer validity period is an increased chance that a certificate is to be revoked before it expires.

Certificate Revocation: Certificate revocation becomes necessary in cases where the above approach is infeasible. An illustrative example is the NDN Testbed [Nam22]. The Testbed trust anchor certifies each participating organization, which may further delegate its subnamespaces to individual departments and issue corresponding certificates for routing purposes[1]. Because certificates for user applications may be further downstream along this signing chain, short-lived organization-level certificates burden user applications with re-issuing their downstream certificates frequently[2]. The above operational feasibility concerns lead to the need for issuing organizational certificates with a relatively long lifetime. This extended lifetime must be combined with a certificate revocation tool to handle the cases where a long-lived certificate must be revoked before its expiration. The revocation tool must record the revocation events and provide a means to inform the data consumers about the revocation status of the certificates.

5.1.2 An Example Scenario

We continue to use NDN routing security as an example scenario below, which will be used throughout the work to facilitate the explanation and discussion.

[1]We view routing as application that also need to securely consume and produce named secured data.

[2]In practice, we observe that the current NDN Testbed issues year-long certificates to participating organizations.

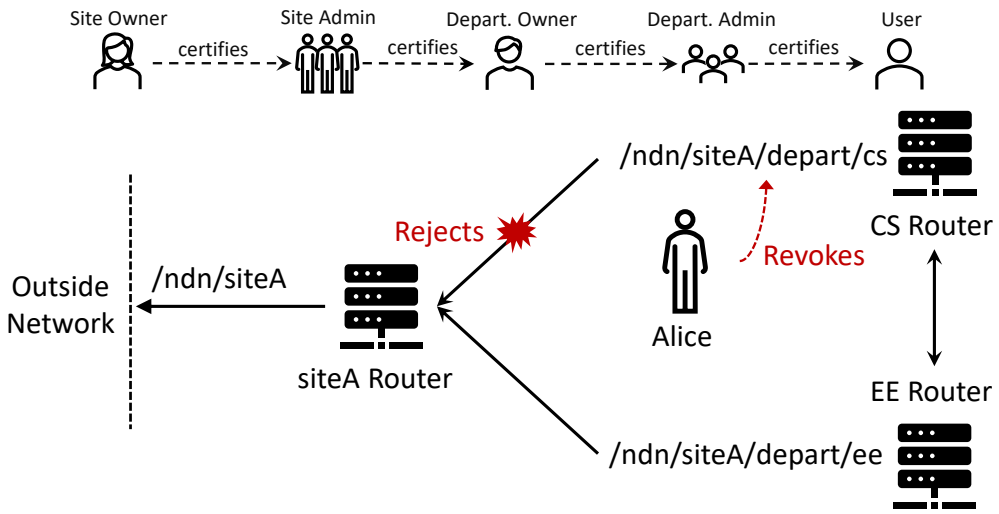


Figure 5.1: A routing prefix registration scenario where certificate revocation happens.

As shown in Figure 5.1, an organization site owner delegates her signing power to a few site administrators to manage each department, and each department owner delegates the signing power to a few department administrators to manage individual users. The routers of each department register department-level prefixes to the site router, while the site router aggregates and registers the site prefix with the external network. Figure 5.1 shows two routers from "`ee`" and "`cs`" department register name prefixes "`/ndn/siteA/depart/cs`" and "`/ndn/siteA/depart/ee`" on the site A router, respectively. The two routers use their renewed monthly certificate to sign prefix announcements and propagate them to the site router. The site router validates the certificate and registers routes for the two departments, then further uses the site certificate to sign the prefix announcement for "`/ndn/siteA`" and propagates them to the outside network. Consider a site administrator Alice discovers that the certificate of "`/ndn/siteA/depart/cs`" was misissued and revokes it. Then, the site A router should know about this revocation event and reject the next prefix announcement signed by this certificate.

5.1.3 Revoking Name-Key Endorsements

Revoking a certificate indicates the *removal of a name-key endorsement* from a specific certificate issuer, which *breaks the corresponding edge* in the authentication graph. Note that revoking one's certificate does not necessarily revoke one's authenticity completely, when multipath authentications exist [YAC15]. For the example shown in Figure 5.2, the trust schema allows the site owner "`/ndn/siteA`" to certify Alice and Bob for the name prefix "`/ndn/siteA/admin`", and both administrators can further certify the department owners for the name prefix "`/ndn/siteA/depart`". Thus, the entity "`/ndn/siteA/depart/cs`" has two authentication chains, and entities who trust "`/ndn/siteA`" can validate "`/ndn/siteA/depart/cs`" through Alice or Bob. In this case, when Alice revokes her issued certificate for "`/ndn/siteA/depart/cs`", it does not disable downstream authenticity, but only removes one of the authentication chains. If the CS router receives certificates from Alice and Bob, it can use the other certificate from Bob to sign prefix announcements. The site A router will consider the routing announcement as legitimately produced data.

Re-validating a certificate means removing an edge that connects two entities in the authentication graph. In this work, we allow either end node of an edge to remove the edge. In other words, the name-key binding endorser (*i.e.,* the certificate issuer) and the endorsee (*i.e.,* the certificate owner) can legitimately invalidate the certificate. Although the certificate owner is not the "producer" of the certificate, it can use its private key to sign a revocation record.

5.2 The CertRevoke Framework

In this section, we show how the CertRevoke design answers the three research questions and meets the design goals.

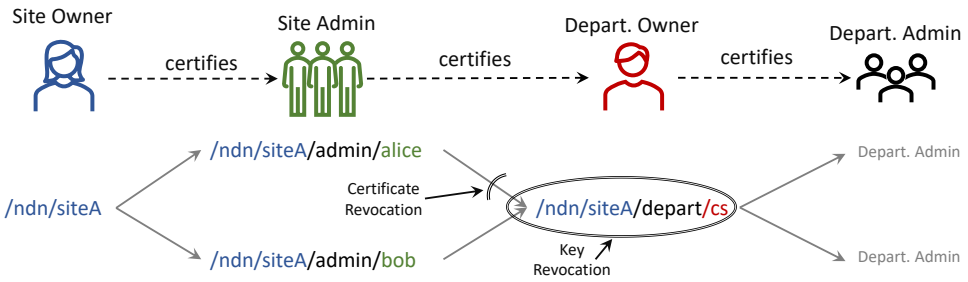


Figure 5.2: An example of multi-path authentication.

5.2.1 Problem Statements

In this section, we analyze the certificate revocation problem in trust plane and propose a model to investigate the design spaces.

System Model: We consider a certificate revocation framework, as shown in Figure 5.3. There are three categories of entities in this protocol, *i.e.,* the *ledger*, *revokers*, and *checkers*. Their roles are discussed below.

- Ledger: The ledger is a set of entities that as a whole provide immutable data storage. It receives the revocation records from revokers and responds to checkers' Interests to retrieve revocation records. In a practical deployment, the ledger's immutable data storage may be implemented by multiple independent Merkle Trees [Lau14, LK12a] or DAG replicas (as Chapter 6 describes).

- Revoker: Revokers are the entities that revoke certificates. They send a revocation record to the ledger when they revoke a specific certificate.

- Checker: Checkers are entities that want to know whether a given certificate is revoked. They send an Interest to retrieve the revocation record from Ledger when they check the revocation status of a certificate.

Assumptions: Since the checker is the party interested in the actual revocation status, it has no incentive to be malicious. We assume that some malicious revokers may try to

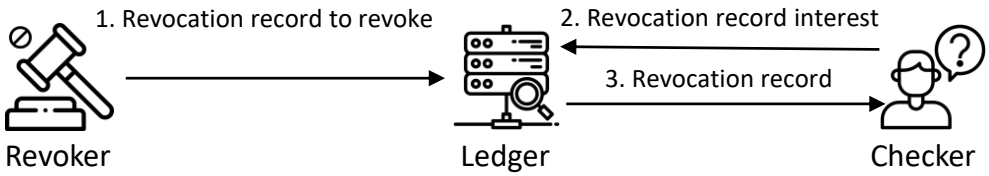


Figure 5.3: CertRevoke System Model

revoke the certificates illegitimately *i.e.,* to revoke the certificates they are not permitted to revoke. Similarly to CT [Lau14], the integrity of the ledger must be verified by external auditors, and we leave the specific design of CertRevoke auditing for future research. For simplicity, we also assume that the ledger, revokers, and checkers are in the same trust domain, and the ledger has a well-known prefix (*e.g.,*"`/ndn/siteA/LEDGER`"). We argue that extending CertRevoke to support multiple domains requires only trivial efforts as defined in Chapter 3. The domain controller bootstraps the aforementioned entities with the domain's trust anchor, issues them certificates, and defines their trust schema.

Design Goals: Based on the above system model and assumptions, the design goals of our proposed framework are defined as follows.

- *Validating Revocation Legitimacy*: The framework can determine the legitimacy of a revoker in a certificate revocation attempt.
- *Maintaining Revocation Recording*: Revocation attempts are recorded by the ledger with the revoker information to prove its legitimacy.
- *Providing Record Accessibility*: Checkers can access legitimate revocation records to determine the revocation status of a certificate.

5.2.2 Revocation Records

Each revocation record is a piece of named secured data. A revoker generates a revocation record to revoke a certificate. Let R^C denote the revocation record of a certificate C. R^c is defined as Eq. 5.1.

$$R^c = \{name, \{timestamp, rCode, cKeyH\}, sig\} \tag{5.1}$$

where *name* is the name of R^C, *timestamp* indicates the time of data signing, *rCode* is the code that indicates the reason why C is revoked, $cKeyH$ is the hash value of the public key in C, and *sig* is the corresponding data signature produced by the revoker.

R^C binds itself to the corresponding certificate C by naming conventions. R^C name is defined as "`/<prefix>/REVOKE/<keyid>/<issuer>/<version>/<revoker>`", where the name components "`<prefix>`", "`<keyid>`", "`<issuer>`" and "`<version>`" are the same as the corresponding parts of the name of C, while "`<revoker>`" carries the revoker information. The naming convention enables checkers to automate the revocation status check by converting the C' name into the corresponding R^C name and expressing the Interest to retrieve it (see Section 5.2.4) .

Revocation Legitimacy: Since revokers sign revocation records with their certificates and the R^C names reveal the corresponding Cs, the trust domain controller manages the revoker's legitimacy by controlling the signing restrictions in the trust schema. *CertRevoke considers the certificate issuer and the certificate owner are legitimate revokers (see Section5.1.3).*

We illustrate how we leverage the trust schema to ensure the legitimacy of the revocation in Figure 5.4, where Alice, as the certificate issuer, revokes the certificate of "`/ndn/siteA/depart/cs`" by producing the revocation record "`/ndn/siteA/depart/cs/REVOKE/223/alice/001/alice`". The trust schema enforces that the revoker component in the name R^C must be equal to the issuer component; (ii) records under the name prefix "`/ndn/siteA/depart`" must be signed by the revokers under the name prefix "`/ndn/siteA/admin`"; (iii) "`/ndn/siteA/admin/alice`" uses the same signing certificate to produce the corresponding revocation record.

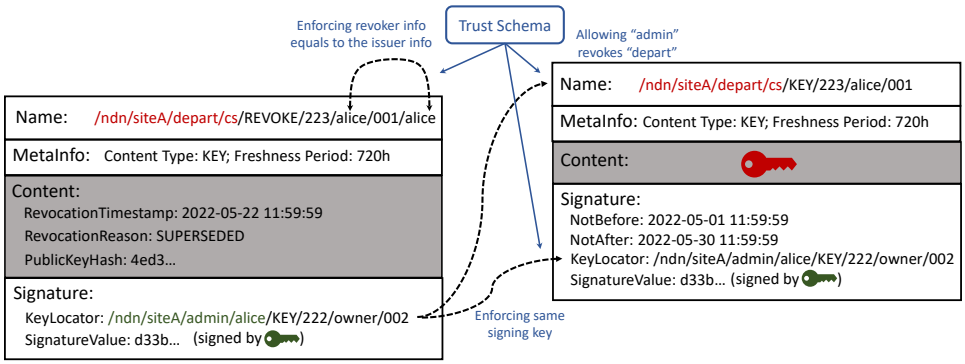


Figure 5.4: Site administrator Alice revokes the certificate of CS department

If the certificate owner revokes the certificate, CertRevoke uses trust schema to enforce (i) that the revoker component must be "`self`"; (ii) the signing key must be the same as the key name prefix in this certificate.

5.2.3 Revocation Submission Protocol

In this section, we show the protocol in which revocation records are submitted to the ledger, and the ledger utilizes the trust schema and the key matching policy to ensure the legitimacy of the revocation.

As shown in Figure 5.5, a revoker attempts to revoke a certificate C by generating a revocation record R^C and submitting R^C to the Ledger. It initiates the submission process by expressing Interest "`/<ledger-prefix>/submit/notify/<param-digest>`" to notify the ledger on the submission event. The notification Interest *I1* carries the revoker's prefix and a nonce to reach this revoker[3], while "`<param-digest>`" is the digest of three parameters. *I1* indicates the submission is under the name "`/<revoker-prefix>/msg/<ledger-prefix>/submit/<nonce>`" and retrievable.

After receiving *I1*, the ledger further expresses the Interest *I2:* "`/<revoker-prefix>/msg/<ledger-prefix>/submit/<nonce>`" to retrieve the submission Data *D2*. Then the revoker

[3] *I1* can also carry an optional forwarding hint if this revoker is not directly reachable

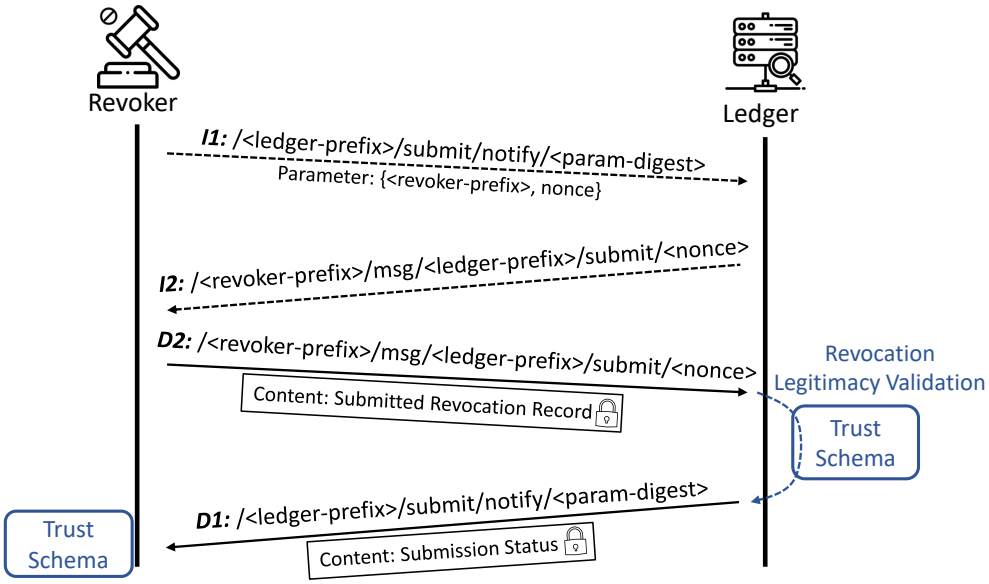


Figure 5.5: Interest-Data exchanges between Revoker and Ledger

encapsulates the revocation record into *D2* with the same name, signs it, and replies.

Because signed *D2* contains the R^C, the ledger executes a trust schema and key matching policy on *D2* to validate (i) whether *D2* itself is legitimately produced; (ii) whether R^C is legitimately produced; (iii) whether the signer $R^{C'}$ and the signer C' satisfy certain name constraints. Upon successful validation, the ledger inserts the record into its storage backend. Note that CertRevoke decouples the submission legitimacy from the revoker legitimacy via validating *D2* and its content separately, allowing other parties to voluntarily submit collected revocation records to the ledger.

After submission validation, the ledger replies *I1* with signed *D1*, which has the same name and encapsulates the submission status. Like the ledger, the revoker also executes the trust schema to validate *D1*'s legitimacy and learns the submission status from the validated data content.

Batch Submission: The record submission protocol allows the submitter to batch revocation records in *D2*. Accordingly, the ledger encodes the submission status for each record in *D1*. When the batch size is large, the submitter needs to segment *D1* into "`/<D1-name>`

/<segment-number>", and use the *FinalBlockId* field of the Data packet to specify the highest segment number.

5.2.4 Certificate Revocation Checking

CertRevoke designs each R^C as a piece of semantically named and secured data. Thus, checking a certificate's revocation status is a data accessibility problem: *How can the checker access the revocation record in the ledger?* CertRevoke addresses this problem by using *Application Layer NACK* and dynamically controlling *Freshness Period*.

Since we consider the certificate issuer and owner to be legitimate revokers, checkers can start from C's name to automatically construct the R^C's name, if it exists, by following the defined naming convention below. In order to access the records, it expresses two Interests "/<prefix>/REVOKE/<keyid>/<issuer>/<version>/<issuer>" and "/<prefix>/REVOKE/<keyid>/<issuer>/<version>/self" together with the forwarding hint to the ledger. If the ledger finds R^C corresponding to the data name carried in the Interest in its backend storage, it replies to the Interest with the R^C. Receiving either record indicates that the checked certificate was revoked. If the corresponding R^C does not exist, the ledger replies with a Data packet that carries an application layer NACK. This Data packet has the name "/<record-prefix>/nack/<timestamp>" with empty content and is signed by the ledger. After verifying the NACK received with the trust schema, it learns that the certificate has not been revoked at this time.

In-network Caching: Because an R^C NACK is a Data packet, the ledger can dynamically adjust its *Freshness Period* to balance the NACK timeliness and traffic load. A long freshness period indicates the chance that future checkers' Interest packets may hit the R^C NACK from the in-network cache, and reduces the workload at the ledger. On the flip side, it increases the chance that the certificate in question may be revoked, making the R^C NACK in the cache no longer valid. In short, the Freshness Period of each R^C NACK must be carefully selected to minimize the chance of obsolete data in the cache while keeping the ledger and

network traffic at a reasonable level.

5.2.5 Putting Together A Case Study

The previous sections explained the individual procedures of CertRevoke. This section shows how everything works together to build a certificate revocation framework that answers the three design questions.

When starting the site network, Site A specifies the trust schema, in which only administrators can legitimately revoke a department's certificate. The revoker component in R^C must be the same as the issuer component in C's name. The owner of site A also runs a ledger instance "`/ndn/siteA/LEDGER`" on the router of site A. During security bootstrapping, Site A's owner installs the trust schema into all site routers. Alice, as site administrator, revokes its previously issued certificate "`/ndn/siteA/depart/cs/KEY/223/alice/001`" by producing a revocation record (see Figure 5.4), then submits R^C to "`/ndn/siteA/LEDGER`" following the protocol mentioned in Section 5.2.3. An hour later, the CS router signs the prefix announcement and sends it to the site router. The routing application on site A's router checks the revocation status of the signing certificate using the mechanism in Section 5.2.4 and learns that the signing certificate was revoked for the reason "`SUPERSEDED`", thus rejecting the announcement.

Meanwhile, the routing application on the site router also receives the prefix announcement from the EE department router. It checks the revocation status and receives an R^C NACK from the ledger. Upon validation of R^C NACK, the routing application continues to validate the prefix announcement and renews the route of "`/ndn/siteA/depart/ee`".

5.3 Evaluation

5.3.1 Implementation

We have implemented CertRevoke in C++ and provide the library for developers to integrate CertRevoke protocols into their specific applications[4]. Our implementation supports the ledger for serving revocation records using persistent memory and storage (*e.g.,* SQL database). It can also work seamlessly with the ledger described in Chapter 6 to provide distributed and immutable data storage.

5.3.2 Evaluation Scenario

Figure 5.6 represents the evaluation scenario. We assume an environment with three forwarders. The hosts of each forwarder are connected through IP overlay links. The checkers and the revoker are virtually connected to the forwarder running on the same host through a Unix socket. The ledger can be connected to Forwarder 1 via an IP overlay or virtual links. We assume that there is no packet loss on each link, but the latency of each IP overlay link between the forwarders varies from 0.5 to 6 ms. The forwarders know the route to each entity. Thus, no route configuration is needed. In addition, all caches on the forwarder were empty when the evaluation started.

We emulated the hosts by running three containers on a Ubuntu 20.04 server. This server features an AMD EPYC 7702P processor with 64 physical cores (128 threads) and 256 GB of RAM. We first evaluate the submission of the record and check the performance with a local ledger connected to Forwarder 1. We also deploy the ledger on a remote host that is connected to Host 1 with a 100 ms propagation delay and investigate the effect of cache on the data freshness period and certificate popularity in Section 5.3.5.

[4]Code has been published at `https://github.com/UCLA-IRL/ndnrevoke`

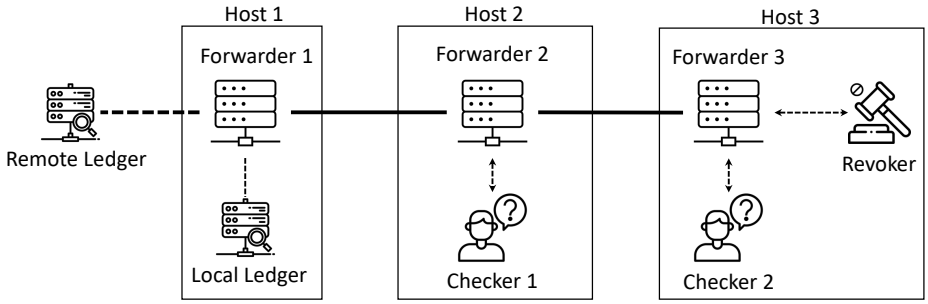


Figure 5.6: Evaluation scenario where checker and ledger, revoker and ledger are multiple hops away.

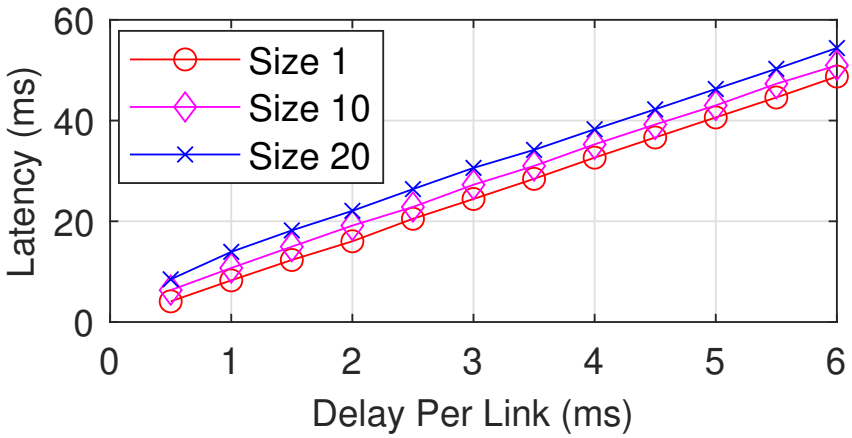


Figure 5.7: The latency of Revoker submitting revocation records to the Local Ledger

5.3.3 Revocation Submission

In this section, we evaluate the record submission performance with different link delays and different batch sizes, as illustrated in Fig. 5.7. We let the revoker revoke 100 certificates and submit the revocation records to the ledger. The results show that the time cost increases linearly with increasing link delay, but is stable at about 2RTT. With increasing batch size, the latency also increases slightly. Specifically, it only takes 54.4 ms when the RTT is set to 24 ms, and the batch size is 20. The results show that our submission protocol is usable in a practical network.

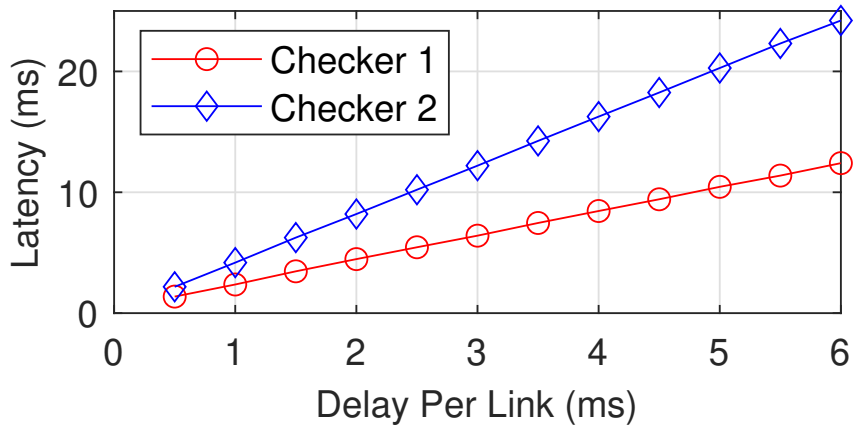


Figure 5.8: The latency of checking revocation records from the Local Ledger

5.3.4 Revocation Record Checking

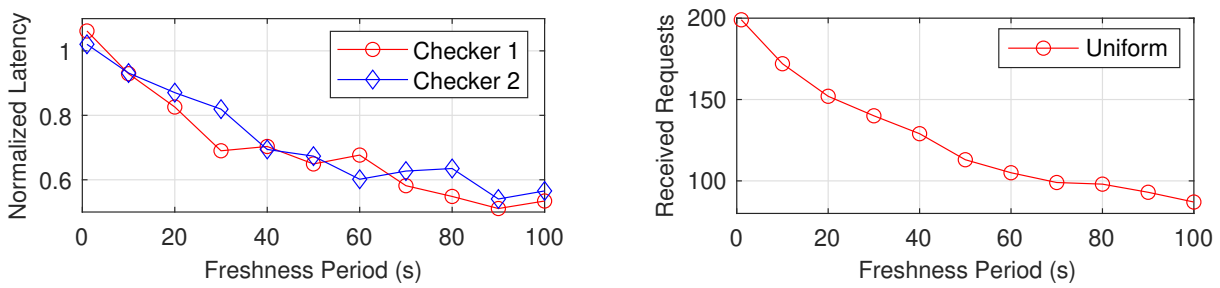
In this section, we assume that Checkers 1 and 2 learn a certificate out-of-band and check its revocation status by expressing Interests for the corresponding revocation records. We assumed that both Checker 1 and Checker 2 were interested in 100 potential certificate revocation records, of which two percent existed. We disabled the cache so that every Interest packet reaches the ledger. As we can see in Figure 5.8, the time costs of both Checker 1 and Checker 2 increase linearly along with the delay per link. With the same per-link delay, the latency of Checker 1 is smaller because it is closer to the ledger. For example, when the delay per link is 6 ms, the latency of Checker 2 is 24.21 ms, while the latency of Checker 1 is 12.41 ms.

5.3.5 Benefit from In-network Caching

CertRevoke relies on in-network caching to efficiently distribute revocation records and record NACK packets. If the local network does not have enough resources to run a ledger, the trust domain controller may remotely bootstrap an entity as a ledger deployment. The performance improvement from the cache depends on the network topology, the data freshness

period, and the average RTT of the record check. In this experiment, we particularly focused on the record NACK distribution for two reasons. First, revocation records are supposed to be long-lived data. Therefore, the benefit of the cache depends a lot on the network topology and cache capacity. Second, revoked certificates only account for a small portion of all issued certificates. In most cases, when an application checks a revocation record, it will receive a NACK.

In this experiment, we deployed the ledger on the NDN Testbed [Nam22] and investigated the record-checking latency. We assumed that Checkers 1 and 2 were interested in the same 100 potential revocation records in Section 5.3.4, and randomly requested one of them every second. Each examination event was scheduled with a 1 to 250 ms random backoff. In order to have a better comparison, we also normalized the latency with the average RTT latency from the checker to the ledger.



(a) Normalized latency of revocation record checking

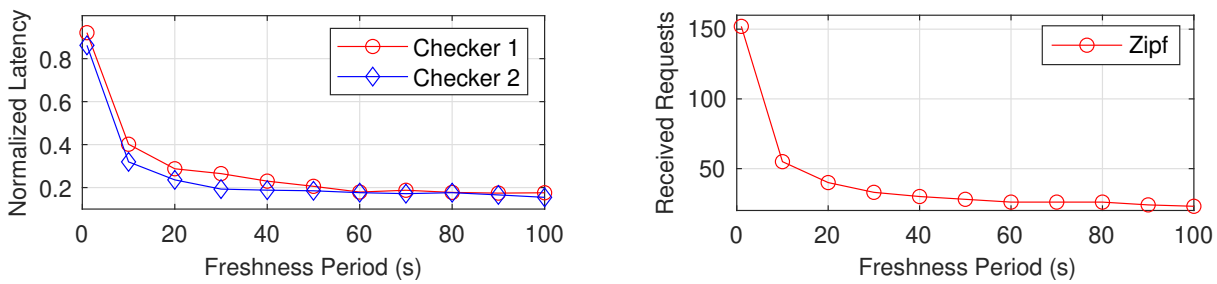
(b) Requests received by Local Ledger

Figure 5.9: Evaluation when checked records follow Uniform distribution

As we can see in Figure 5.9a, the normalized latency follows a similar downward trend as the data freshness period increases. That is because the caches in intermediate forwarders can respond to some requests. The longer the freshness period, the greater the proportion of requests that hit the cache. When the caching benefit converged, the record checking latency was improved by 45%. We also investigated the number of requests that Remote Ledger received in Figure 5.9b. For the same reason, the number of requests received also

decreases along with the data freshness period. For example, among the first 200 requests that the two Checkers sent, the ledger only received 87 requests when the freshness period was 100 s. In other words, in-network caching decreases the ledger's load by 56.5%.

However, if the average RTT takes 200 ms (approximately from southern Europe to North America on the NDN Testbed), 45% improvement still requires the checker application to wait >100 ms while checking a record. Fortunately, in real applications, not every certificate has the same possibility of being checked. We know that the popularity of content on the Internet today closely follows the Zipf distribution [FLT13]. We can assume that NDN certificates, as data content signers, follow the same popularity distribution. Higher-ranked certificates are checked more often and thus benefit more from in-network caching.



(a) Normalized latency of revocation record checking

(b) Requests received by Remote Ledger

Figure 5.10: Evaluation when checked records follow Zipf distribution

Therefore, in the third setting, to illustrate the real performance, we use the same ledger deployment on the NDN Testbed, but assume that the popularity of the certificate follows the Zipf distribution with $s = 2$. As illustrated in Figure 5.10a and Figure 5.10b, both average normalized latency and received requests decrease exponentially. Moreover, they are also significantly lower than those when certificates share the same popularity. For example, the latency of Checker 2 in Zipf distribution is only 15. 5% of the original RTT when the freshness period is 100 s, which is significantly lower than the converged performance 45% in Figure **??**. The number of requests received among the first 200 requests was only 23 when

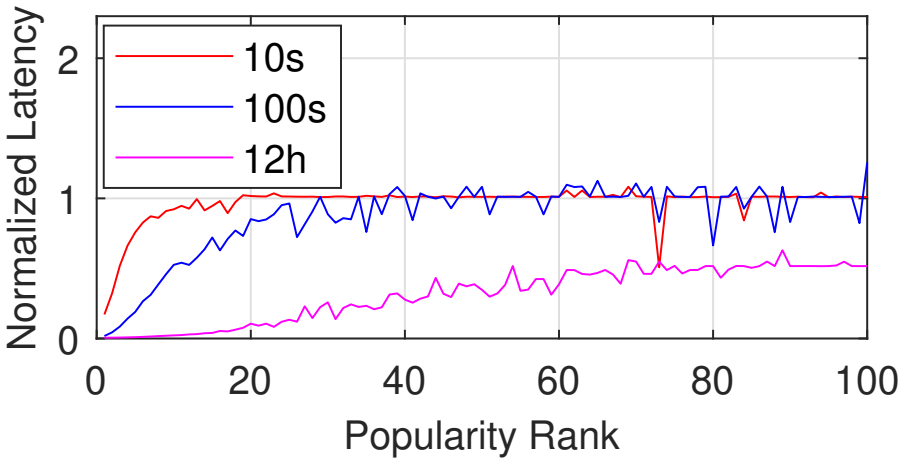


Figure 5.11: Normalized latency with popularity rank

the freshness period was 100 s.

In order to have a closer look at how certificate popularity affects the record checking latency, we also measured the average record checking performance for each certificate. Similarly to previous experiments, we normalized the latency with the average RTT from the checker to the ledger. When data expires more often (*e.g.,* 10 s), checking a popular certificate benefits less from in-network caching. As we can see in Figure 5.11, the normalized latency quickly converges to 1 as the popularity rank decreases. Only the top 20% certificates take a shorter time to examine. This number becomes the top 40% when the data freshness period is set to 100 s, and the top 10% takes 0.5 RTT for Checkers to examine their revocation statuses. Based on the above evaluations, we show that even when the ledger is deployed remotely, the record checking latency is acceptable with the appropriate configuration of caches.

Cache Utilization in Practical Deployments: There are two major factors that affect cache utilization in practical CertRevoke deployments: the network topology and the NACK freshness period of the revocation record. We are aware that the network topology used in our experiments enables every record Interest to hit the cache. In order to improve the cache utilization in a practical deployment, one may deploy distributed ledger instances in

different locations in the network. A practical revocation record NACK's freshness period can be multiple hours or even a few days. Our evaluation shows that the performance of record checking promptly converges as the NACK freshness period increases, even the freshness period in this experiment is significantly smaller than the practical network uses (*e.g.,* one day or more).

5.4 Discussions

Distributing Revocation Records: Domain controllers can also directly embed revocation records in the update of the trust schema. Data consumers do not need a separate round trip at data validation time to check the revocation status, but rather learn the revocation records ahead of time. It also scales because each domain takes care of the revocation records with which they are concerned. We further argue that both approaches are more lightweight compared to today's practice that either (i) browser vendors usually compile their own filtered CRLs and embed them directly in the browser (ii) CAs sign OCSP [SMA13] responses upon certificate status queries.

Revocation Impact: When an upstream, long-lived certificate is revoked, it invalidates all downstream Data in the authentication chain, including potentially innocent packets. For example, as shown in Figure 5.12, Alice discovers that the certificate of "`/ndn/siteA/depart/cs`" is misused at *T1* and decides to revoke it. The downstream certificate "`/ndn/siteA/depart/cs/author`" issued at *T0*, which is earlier than *T1*, might be innocent. To reduce the impact of the revocation, an optional field "`notBefore`" can be added to the revocation record to indicate the start timestamp of the invalidation. When the checker receives a revocation record containing "`notBefore`", it compares the validity period of the certificate with "`notBefore`" to determine whether it can be exempted. Note that "`notBefore`" is based on the best estimate of the revoker. Thus, the usage of "`notBefore`" objectively increases the security risk and is not recommended.

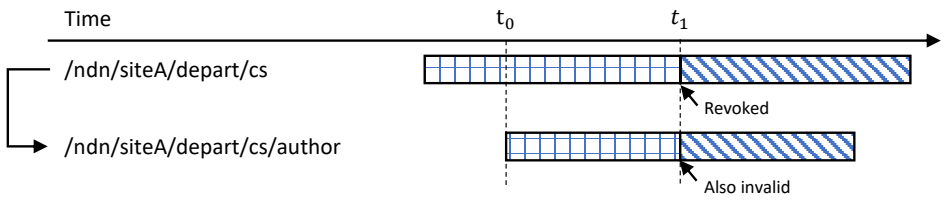


Figure 5.12: Example of Revocation Impact

Record Checking Latency: The verification of the revocation status for each certificate along the certificate chain requires data consumers to take at least one RTT per certificate to retrieve revocation records. Although in-network caching mitigates this issue, one can take the following approaches to further reduce the record checking latency: (i) Sending the record fetching Interests together with the Interest to fetch the upstream certificate. (ii) Similar to OCSP Stapling, providing fresh Record NACK packets in the certificate bundle [MAZ17] or trust schema updates. (iii) Only perform the revocation record check for long-lived certificates (e.g., validity period longer than a threshold). Also note that, unlike OCSP, a CertRevoke revocation record is *not* a certificate status, and the record name can be directly converted from the corresponding certificate name. Therefore, a CertRevoke ledger does not need to manage the revocation status for each certificate, but rather a key-value storage. This advantage enables fast record look up and at the ledger side.

Certificate Revocation versus Key Revocation: In Section 5.1.3, we mentioned that the revocation of the certificate is to remove the relationship between two entities. It is worth mentioning that certificate revocation does not revoke the corresponding key pair. *Key revocation* is a separate concept that directly invalidates the key, *i.e.,* removing the corresponding node in the authentication graph. We show the difference between certificate revocation and key revocation in Figure 5.2. If each entity node in the authentication graph has only one key, revoking the key of entity "`/ndn/siteA/depart/cs`" breaks all authentication chains from "`/ndn/siteA`" to this entity by distrusting the revoked key itself. Of course, an entity may possess multiple keys to mitigate the impact of losing one key. Nevertheless, we

believe that key revocation is a separable research problem in trust management, and leave it for our future work.

Privacy Implications: The privacy of a certificate checker is an important concern in certificate revocation designs. For example, The OCSP design lets each client check with an OCSP server about whether a given certificate has been revoked, and this OCSP request allows the OCSP server to learn that a specific client is visiting the domain being checked [LCL17]. In contrast, because NDN Interest packets do not carry consumer identity information, the Interests to check revocation records do not disclose any checker information. If a checking request does not hit a cache along the way before reaching a ledger, the ledger can only learn the domain being checked, but not the consumer who sent the record checking Interest.

CHAPTER 6

CLedger: A Secure Distributed Certificate Ledger

A trust domain needs an immutable ledger to provide a verifiable history for data auditing. We propose CLedger, a distributed, secure, and resilient certificate ledger, as initial efforts to meet the need. CLedger consists of a set of distributed loggers that store certificates and revocation records from authorized producers. For convenience of explanation, we use certificate logging as the driving case in the rest of this chapter. CLedger utilizes an append-only data structure, HashDAG, which is based on a Directed Acyclic Graph (DAG). HashDAG ensures that once a certificate is stored in CLedger, it cannot be removed. We further argue that the same design is also applicable to log all system actions as secured named data.

6.1 Background

Directed Acyclic Graph (DAG) is a directed graph without cycles. In the field of storage, the DAG is a graph-based data structure in which nodes are data records, and directed edges are reference relations among data records. DAG has been used in several distributed storage designs [LCP20, ZVK19, LMZ21]. Unlike blockchain-based designs, a storage node can insert a received record into its local DAG copy immediately, and the DAG in each storage node can be different at a specific time point. These DAGs can merge efficiently and reach eventual consistency, *i.e.,* every record is stored by all DAGs eventually. Moreover, if references are built using hash functions, a record will contain a cryptographic hash of the previous record. Modifications to that record are detectable unless its referrers are also compromised.

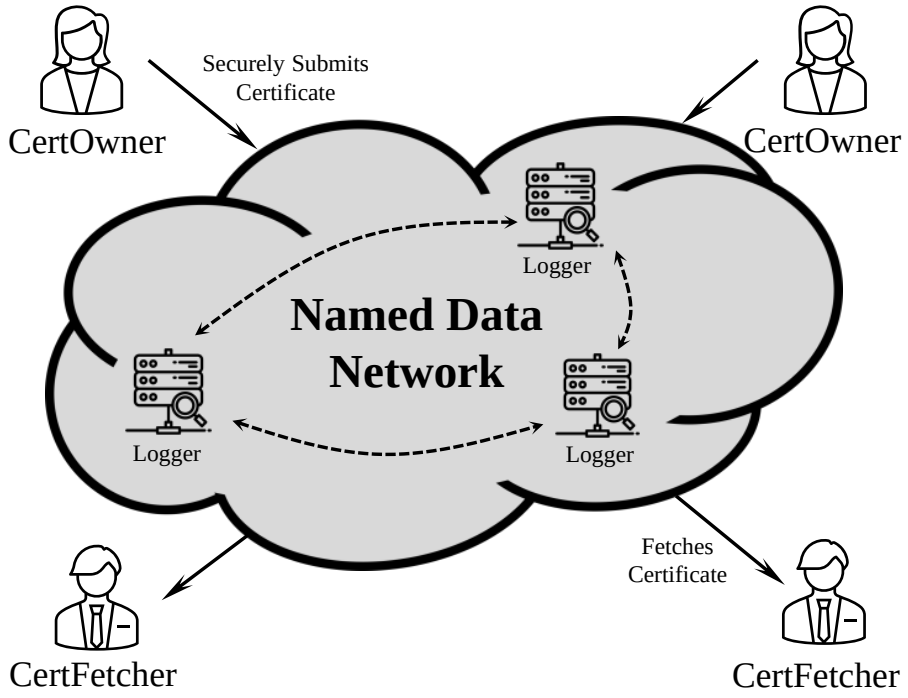


Figure 6.1: A scenario where CLedger has Loggers deployed in the network, and CertOwners would like to use CLedger to provide their certificate availability, while CertFetchers want to obtain certificates from CLedger in order to validate application data.

6.2 System Model

We consider a scenario as follows. There is a distributed ledger CLedger, multiple CertOwners who want their certificates to be stored in the ledger, and multiple CertFetchers who want to fetch certificates. As shown in Figure 6.1, their roles are summarized below. For simplicity, we assume they are from the same trust domain and are already bootstrapped.

- *Logger:* CLedger consists of Loggers who accept legitimate certificate submissions from CertOwners and publish *records* within CLedger. Each record is a semantically named and secured data object that includes a certificate from CertOwner.

- *CertOwner:* CertOwners are legitimate certificate owners who are authorized by the

CLedger trust schema to submit certificates to Loggers.

- *CertFetcher:* CertFetchers are application data consumers who use CLedger to retrieve data producers' certificates.

A viable design consists of the above components and should have no single point of failure and ensure record immutability, so that a recorded certificate cannot be removed from the system.

6.3 Design Goals

We design CLedger for systems that have fewer than K loggers that make operational errors or perform malicious attacks, either refusing the CertFetcher certificate fetching request or deleting the record from their local storage. This assumption is a generalization of the assumption in Practical Byzantine Fault Tolerance [CL99]. With the above assumption, our design achieves the following three goals.

- Certificates are available if any Logger is available.
- Only authorized CertOwners are allowed to submit certificates to Loggers.
- If a Logger accepts an authorized submission from a CertOwner, this record is stored in at least K loggers.

6.4 CLedger Design

This section describes the CLedger design by first explaining the individual functions of the system and then showing the overall workflows of the system when CLedger is in operation.

6.4.1 Certificate Submissions

We use the submission protocol developed by our previous work [YXL22] to allow the submission of secured certificates.

When submitting a certificate, which is a named secured data, CertOwner encapsulates it into another data packet and sends an interest notification to inform CLedger that there is a certificate ready to be submitted. CLedger loggers register a shared routing prefix (*e.g.,*"`/ndnfit/LEDGER`"). Therefore, when CertOwners express an Interest with CLedger's prefix, the network will anycast the packet to its nearest logger.

After receiving the notification, a Logger sends another Interest back to retrieve the submission. After validating both the CertOwner's submission Data and the submitted certificate in these Data, this Logger replies to the CertOwner's notification Interest with a Data packet as an acknowledgment. Thus, CertOwner is informed that the submission process has been completed.

When submitting certificates, a CertOwner signs the submission with its certificate private key, so Loggers are able to authenticate the CertOwner and validate whether the CertOwner is authorized by the trust schema to use CLedger. Upon receiving a validated submission, a Logger produces a record to log the submission. Loggers securely synchronize records via SVSPS [MPZ21], where Loggers join the same Publish/Subscribe group and subscribe to new records produced by other Loggers.

6.4.2 Interlocking Records

CLedger defines record immutability as having replications in at least K loggers and realizes this idea by introducing a new HashDAG data structure inspired by IOTA Tangle [LCP20]. In HashDAG, each vertex is a record, and each directed edge refers to a previous record, which is implemented by storing the hash of the previous record in the current record. The edges of the HashDAG represent the replication relationships. For example, $R5$ in

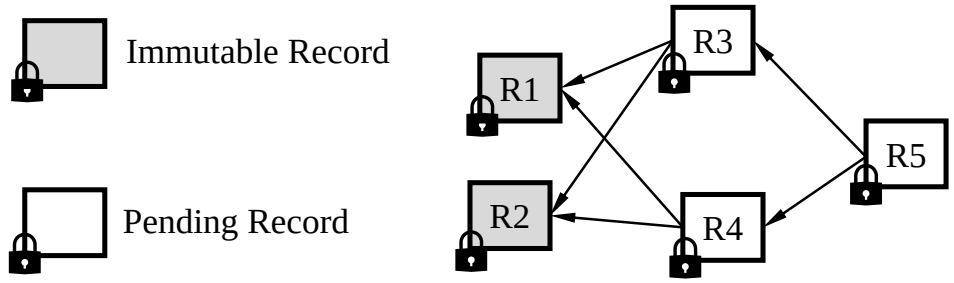


Figure 6.2: A DAG example where $K = 3$ and each record is published by an unique Logger.

Figure 6.2 refers to $R3$ and $R4$ directly and refers to $R1$ and $R2$ indirectly. This reference relation indicates that the Logger that produces $R5$ claims to have seen and replicated $R3$ and $R4$ and all their dependencies (*i.e.,* $R1$ and $R2$) and replicated them in its local copy of HashDAG. Therefore, if a record has been referred by more than K unique loggers, that record becomes immutable; otherwise, we refer to its status as pending. For example, $R1$ and $R2$ in Figure 6.2 are immutable since their parents are produced by three unique Loggers, while $R3$, $R4$, and $R5$ are still pending.

Obviously, the latency for a pending record to become immutable is up to how often Loggers are publishing. However, the logger publication rate also depends on how many nearby CertOwners are submitting certificates. If only a few Loggers are publishing records, the immutability latency increases, or even becomes infinite when there is no new Logger contribute references by producing record and refer to the existing tail.

In order to address this problem, we design the *dummy record* mechanism for CLedger. Each Logger maintains a timer and refreshes it whenever the Logger receives a new record from SVSPS. If no records are received in a given amount of time, the Logger publishes a dummy record that does not contain a certificate but only refers to all tail records.

We made two decisions to prevent CLedger from keeping publishing dummy records when there is no new certificate submission. First, a Logger will not refresh its timer when receiving a dummy record. Second, a dummy record can only refer to records that contain

certificates. Therefore, when HashDAG growth stalls, more Loggers can participate to help pending records to be immutable.

6.4.3 System Operations

Now we use an example shown in Figure 6.3 to further demonstrate CLedger operations.

We consider a case where Alice is a CertOwner and Bob is a CertFetcher. At first, Alice submits her certificate to CLedger using the submission protocol. Alice's nearest logger receives the submission, validates it with trust schema, and publishes a record $R5$ that includes Alice's certificate and references to all HashDAG tail records $R3$ and $R4$.

As all loggers are in the Publish/Subscribe group, other loggers receive this $R5$ from the record subscription and append it to their HashDAG. Because $R5$ has become the tail record, when other Loggers publish new records, their records will refer to $R5$. When there are at least K Loggers have directly or indirectly referred to $R5$ in their record publications, $R5$ becomes immutable.

We assume that Bob starts fetching Alice's certificate after Alice's certificate submission finishes. Bob expresses an Interest with Alice's certificate name[1]. Bob's nearest Logger receives the Interest and looks up its local copy of HashDAG for the record that contains Alice's certificate.

If such a record exists in HashDAG, the Logger extracts Alice's certificate from it and replies to the Interest with the certificate found. Otherwise, the Logger replies to the Interest with a Data packet that shares the same name prefix as Bob's Interest but has an empty content. Logger signs the reply to inform Bob that CLedger has no record of *which* certificate[2].

[1]Interest packets can carry forwarding hint that hints the routers where to forward the packet. In our case, Bob's certificate fetching Interest carries the forwarding hint to the CLedger shared routing prefix.

[2]The Logger can use the Data FreshnessPeriod to control the timeliness of this information, so that Bob knows when to resend the request.

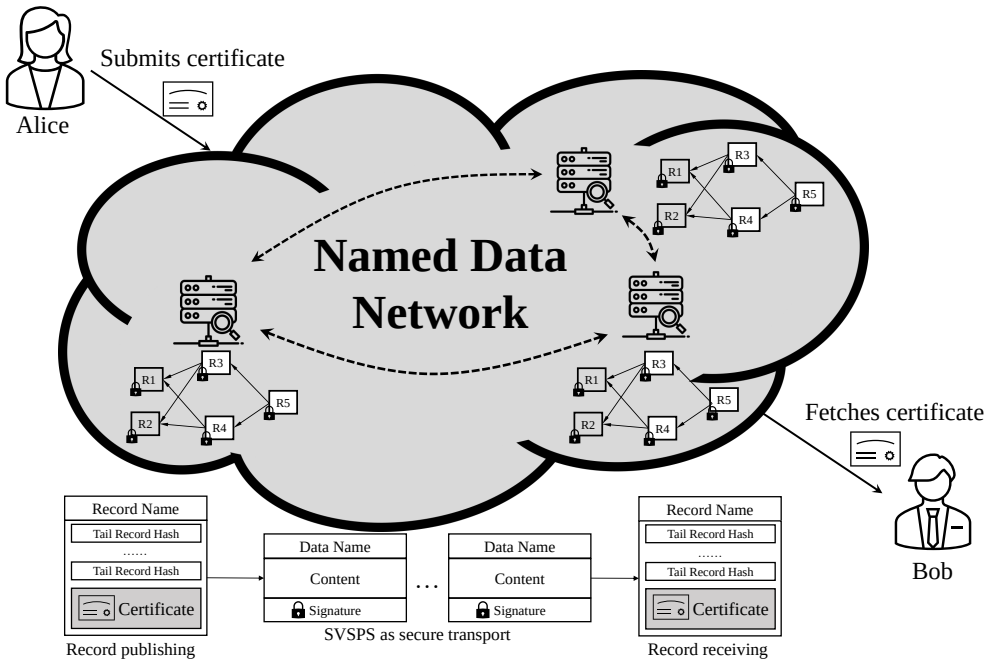


Figure 6.3: An CLedger example where Alice is the CertOwner while Bob is the CertFetcher.

In the case shown in Figure 6.3, the logger closest to Bob has learned $R5$ from SVSPS, and hence successfully looks up $R5$ from its local copy of HashDAG and replies to Bob's interest with Alice's certificate.

6.5 Evaluation

In this section, we evaluate the performance of CLedger with various parameter settings, including submission latency, fetch latency, immutability latency, and certificate / total record ratio.

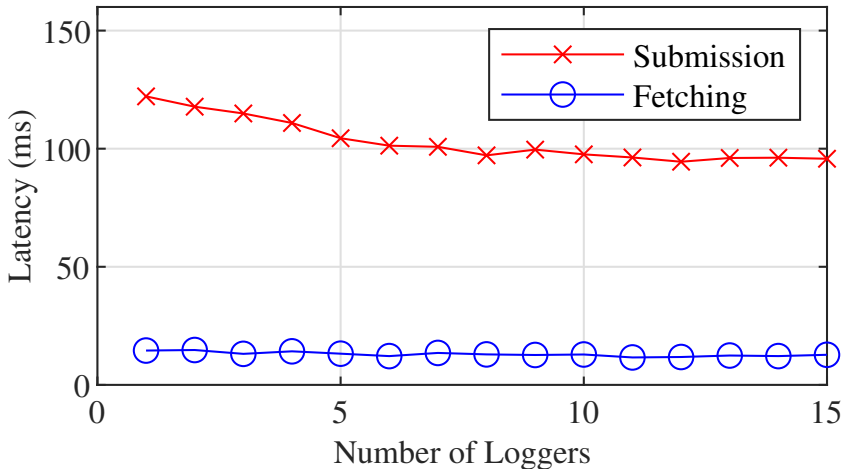


Figure 6.4: Availability Latency

6.5.1 Evaluation Setup

To evaluate CLedger's performance, we implemented a prototype[3] based on ndn-cxx. Our prototype uses LevelDB as the local database for the loggers. We use Mini-NDN [The22] to emulate CLedger in a real-world ISP network topology [SMW02] that includes 115 nodes. They are connected to each other via the delay 10 ms per link. Our evaluations were hosted on a server equipped with Ubuntu 20.04, AMD EPYC 7702P with 64 cores and 256 GB RAM. Each evaluation result is the mean of 10 trials.

6.5.2 Submission Latency and Fetching Latency

We measure the certificate submission latency and the fetching latency in this section, to evaluate the performance of certificate availability provided by CLedger. According to Section 6.4, we define submission latency as the time elapsed from CertOwner sending the notification interest to receiving the acknowledgment. Fetching latency is the time elapsed from CertFetch sending a certificate Interest to receiving the requested certificate.

[3] `https://github.com/UCLA-IRL/cert-ledger`

In order to serve more CertOwners, Loggers in actual deployments are more likely on the nodes which have more neighbors. Therefore, to emulate the logger selection process, we sort all the nodes in descending order of edge counts, and select the top N nodes as loggers in each trial[4], with N ranging from 1 to 15.

For submission latency evaluation, we assume that submission events have a Poisson distribution and set λ as 10, *i.e.,* the expected number of submissions per second is 10. For each submission, we randomly choose a node as a CertOwner and submit its certificate. The submission evaluation lasts for 120 seconds, and we evaluate the mean submission latency of all submitted certificates. As shown in Figure 6.4, the submission latency decreases as the number of loggers increases because the larger the number of loggers, the higher the probability that a CertOwner can find a nearby logger with small hops.

After all certificates are submitted into the system, we evaluate the certificate fetching latency. We also assume that fetching events have the same distribution as submission events. For each fetch request, we randomly choose a node as CertFetcher and fetch a random certificate already in the system. We assume that the fetched certificates have the Zipf distribution. It also lasts for 120 seconds, and we evaluate the mean latency of all fetching requests. As shown in Figure 6.4, the fetch latency remains stable as the number of Loggers increases, because popular certificates are fetched several times under the Zipf distribution. Due to the cache mechanism in NDN, they are cached by the intermediate nodes so that a CertFetcher can fetch them from the intermediate routers.

6.5.3 Immutability Latency

In this section, we measure the immutability latency to evaluate the immutability performance provided by CLedger. In this experiment, we chose 15 loggers with the same node selection process. The dummy record timer for each Logger is set randomly from 0.1 to 20 s.

[4]We exclude the top 2% nodes before the selection, because these nodes are probably routers in the real-world scenario.

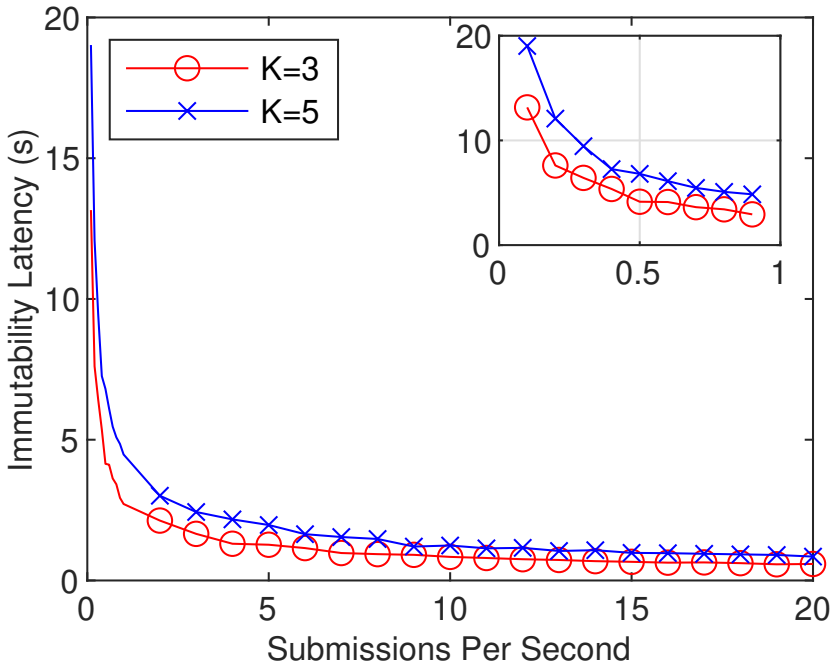


Figure 6.5: Immutability Latency

As illustrated in Figure 6.5, we evaluate the immutability latency with different submission speed rates and K settings, where the figure in the corner shows the immutability latency with a low submission speed. The immutability latency in both cases $K = 3$ and $K = 5$ decreases as the submission rate increases. The increasing number of records submitted per second makes a newly submitted record quickly referenced. The latency in the case $K = 5$ is higher than in the case $K = 3$, because a submitted record needs more descendant records for larger K. Our evaluation results show that our record can be immutable after a short period of time. When the submission speed is 20 records per second and $K = 5$, a record only needs 862.89 ms to be immutable.

6.5.4 Certificate/Total Record Ratio

As CLedger uses dummy records to accelerate immutability, we evaluated the proportion of certificate records in all records. In Figure 6.6, we evaluate the proportion of CertOwner

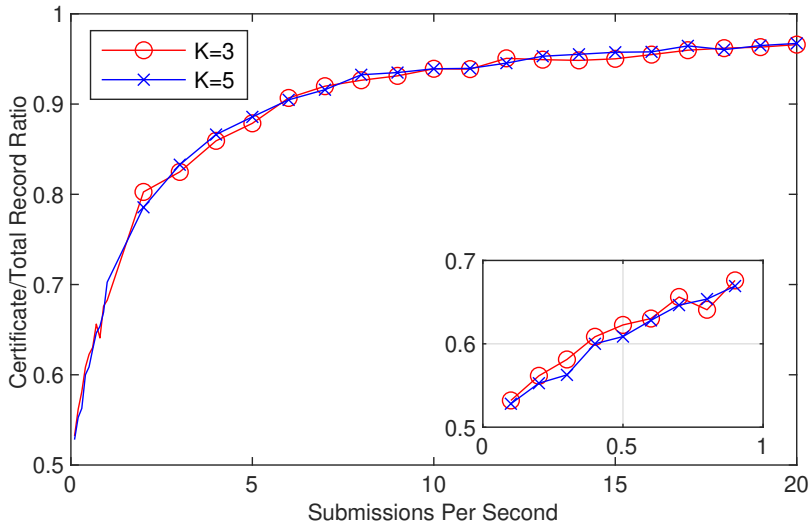


Figure 6.6: Certificate/Total Record Ratio

certificates with different submission speed rates and K settings, where the figure in the corner shows the immutability latency with low submission speed. The number of loggers is set to 15, and we select them based on the same policy that we used in Section 6.5.3. We can find that the proportions in both cases $K = 3$ and $K = 5$ increase when the submission speed rate increases. When the submission speed rates are greater than 15 records per second, the proportions of certificate records are over 95%.

6.6 Discussion

Extending to Inter-domain Ledger: Although the current CLedger design started by providing certificate availability within one trust domain, CLedger can also achieve the same goals for multiple domains. In a multi-domain deployment, each domain contributes Loggers with trust schema that specifies rules on how to authenticate and authorize data from other trust domains. Therefore, federated loggers can provide immutable certificate storage for CertOwners and CertFetchers from multiple domains.

Dummy Record Timer: CLedger relies on a dummy record timer to enable records to

be immutable on time. However, system operators should carefully consider the timer interval. First, the interval of the timer should present system operators' consideration of how urgently records need to be immutable. As shown in Figure 6.6, when CLedger only has a few submissions per second, the ratio between CertOwner-submitted records and dummy records drops significantly. This indicates that CLedger immutably stores certificates with a higher communication overhead. Second, in order to prevent multiple loggers from publishing dummy records simultaneously, each logger should use a variable length timer. In our evaluation, each logger uses a randomized timer whose interval is evenly distributed but with the same mean value. This might not be the optimal choice, since the urgency of record immutability is increasing as time increases when no one is publishing new records. A better randomization strategy is to have the probability distribution function prefer shorter timers.

Toward Generic Data Ledger and Auditing: There are no functional limitations that prevent a networked system using CLedger to log system actions as named secured data. We envision that a generic data ledger should take data submissions from the actuators in the system, who can be a smart home door lock, a file server that takes insertion and deletion commands, or a Virtual Network Function (we will discuss it in the next chapter). Whenever a received data triggers them to perform an action, they submit the data together with its authentication chain to the CLedger. In this case, the DAG becomes a non-linear history of system operations with casual order, so that external auditors can trace the event chain encoded in DAG to continuously monitor system behaviors.

CHAPTER 7

TrustNG

In this chapter, we apply the trust plane framework in the cellular core network. We introduce TrustNG (Trust over Named Data Networking for the Next Generation of cellular networking), a novel cellular core architecture that leverages the trust plane to provide structured support for the 3As in cellular control plane operations. We first give a brief overview of how 5G is deployed in cloud environments and identify the problem that the architecture lacks a systematic framework with intrinsic security protection. We then describe the design and implementation of TrustNG, followed by evaluations that demonstrate its effectiveness in providing systematic security support.

7.1 Background

The 5G core is built as a Service-Based Architecture (SBA), where network functions are implemented as distributed software components, denoted as VNFs, that communicate over HTTP-based APIs. The core is logically divided into two domains: the **Home Network**, which handles subscriber data and authentication, and the **Serving Network**, which manages access and mobility services. Figure 7.1 illustrates the key architectural components.

The Access and Mobility Management Function (AMF) anchors the user to the core and interacts with the Session Management Function (SMF) to establish and manage data sessions. The Unified Data Management (UDM) and Unified Data Repository (UDR) maintain subscription profiles. The Authentication Server Function (AUSF) performs authentication

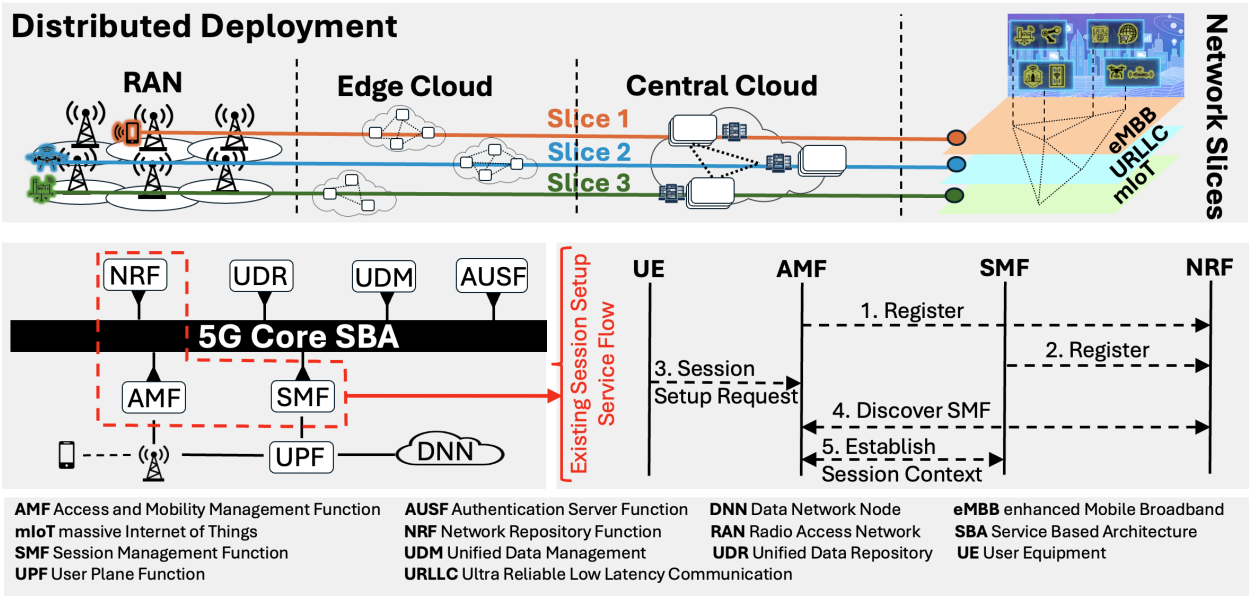


Figure 7.1: 5G network slicing architecture with distributed VNF deployment, highlighting session setup, registration, and discovery processes.

procedures, while the Network Repository Function (NRF) maintains a registry of VNFs and their profiles.

Service Chaining and PDU Session Establishment: Within the 5G core, network services are created by chaining VNFs based on the required use case. One of the most critical and frequent service chains is the Packet Data Unit (PDU) session establishment, which allows the User Equipment (UE) to exchange user data. This service involves interactions between the AMF and the SMF to create a session context and configure the user plane. The session setup process begins with the UE sending a request to the AMF. The AMF queries the NRF to discover a suitable SMF and initiates a *point-to-point secure session* using TLS. Once the session is authenticated, the AMF establishes the session context with the SMF.

TLS and Session-Based Security: Security in the 5G core is implemented using session-based mechanisms such as TLS. Each VNF possesses an X.509 certificate issued by a CA. This certificate binds the identity of the VNF to its public key. During the TLS handshake, the server (*e.g.,* SMF) first presents its certificate to the client (*e.g.,* AMF). The client then

verifies the certificate against a trusted CA chain. If the certificate is valid and not revoked, the client proceeds to establish an encrypted session using symmetric keys negotiated over the authenticated channel, and the signaling authorization uses OAuth2 tokens. Figure 7.1 shows the simplified session setting that focuses on AMF–SMF communication. The key security step is the establishment of a TLS session, which relies on certificate validation.

7.2 Problem Statement

The security solutions of existing cellular networks remain anchored in legacy, session-bound mechanisms. This approach was adequate for static systems, but causes delays, extra overhead, and potential points of failure in highly dynamic environments. The problem lies in the layer *between network functions, slices, and domains*. This is the infrastructure layer where the core and RAN entities must continuously authenticate, authorize, and exchange control and user plane information. Each time VNFs or slices are instantiated, migrated, or scaled, secure channels need to be reconstructed, certificates validated, and tokens exchanged. What is missing is a *systematic framework* to make security protection intrinsic, persistent, and verifiable regardless of how or where services are deployed. Furthermore, today's CA infrastructure enforces trust only at coarse domain-level boundaries rather than at the fine granularity of specific functions, slices, or domain relationships.

Session-bound security overhead: Each interaction between VNFs requires establishing a new session, repeated certificate validation, and token exchange. In ephemeral deployments where VNFs are frequently instantiated, migrated, or scaled, this causes excessive signaling and latency that directly undermines the low-latency promises of 5G.

Coarse-grained certification: Certificates are issued at the domain level, not at the granularity of cellular functions or slice-specific relationships. This means that two VNFs might be forced to trust each other simply because they belong to the same CA domain, even if fine-grained policies specify otherwise. Additionally, cross-slice or cross-domain collaboration

is fragile and requires manual configuration of new trust anchors.

Without reconstructing how security is established and enforced between VNFs, slices, and domains, next-generation cellular systems will continue to face problems with scalability, operational complexity that is inadequate for cloud-native environments. Looking ahead, we see that a trust plane with NDN realization has the potential to help develop a systematic security framework for the cellular core.

7.3 TrustNG Design

The 5G core is inherently organized into multiple administrative layers. At the top lies the PLMN domain managed by a Mobile Network Operator (MNO), such as `Verizon` or `T-Mobile`. Within a domain, multiple *tenants*—including MVNOs or enterprise customers (*e.g.,* `Visible` or `Mint`)—can operate. Each tenant provides one or more *network slices*, each customized to specific service classes such as eMBB, URLLC, or mMTC [SA24]. A slice is composed of a set of VNFs deployed across the edge, regional, or central NFVI, each exposing specific service interfaces for session management, policy enforcement, mobility anchoring, and more. This architecture results in a complex, multi-domain, multi-tenant, multi-slice ecosystem. Given the ubiquitous service orientation of 5G, it is increasingly common for VNFs of different organizations to communicate even when they belong to separate slices, tenants, or administrative domains. For example, an **Operator A** might deploy a custom Application Function (AF) [NT24] designed to collect performance metrics across multiple slices - necessitating secure, fine-grained trust relationships that span these administrative boundaries. This environment creates three interrelated challenges: *Authentication*, verify that each VNF or domain entity is who it claims to be; *Authorization*, ensure that cross-slice or cross-domain interactions occur only within the permissions granted; and *Auditing*, provide verifiable logs of trust decisions and data flows across slices and domains. TrustNG design focuses on the first two, and we argue auditing can be readily supported by deploying

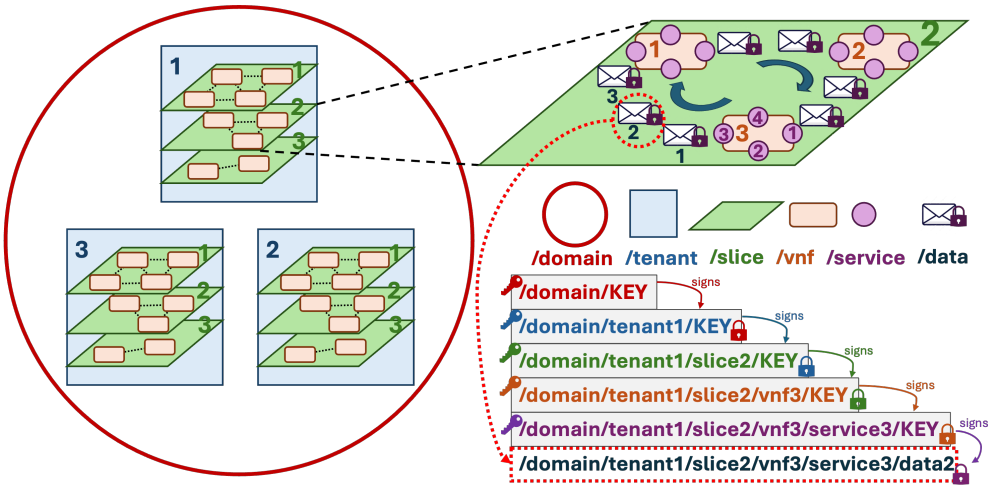


Figure 7.2: naming convention for trust management in next-generation cellular networks with multi-domain, -tenant, -slice deployments.

a distributed ledger described in Chapter 6.

7.3.1 Semantic Naming

We use the following naming convention to capture the aforementioned 5G semantics:

"/<domain>/<tenant>/<slice>/<vnf>/<service>/<dataID>*"

where "<domain>" designates the PLMN managed by a MNO, serving as the root of the trust hierarchy; "<tenant>" identifies entities such as MVNOs or enterprise tenants leasing resources within the PLMN. Multiple tenants can co-exist under the same PLMN, with trust relationships logically isolated at this level; "<slice>" represents the specific network slice allocated to a tenant, aligned with Slice Service Types (SSTs) [SA24] such as eMBB or URLLC; "<vnf>" and "<service>" specify the functional microservice components within the slice; "<dataID>*" refers to a sequence of name components that identifies the unique data object requested or delivered.

7.3.2 Authentication and Authorization

The trust anchor delegates signing authority to downstream entities such as tenants, slices, and VNFs. The VNF key then signs the data produced by services in its namespace. Figure 7.2 illustrates the data, keys, and their relationships. Each step of delegation is performed with a certificate and explicitly defined signing rule (discussed shortly), enabling data authenticity and authorization checks without relying on session-based mechanisms, and achieve the finest-level of delegation, eliminating coarse-grained certificate problem in the traditional solutions. We design the following trust schema to express the signing relationships. Figure 7.3 specifies the trust schema for an exemplified tenant "`tenant1`". The schema has two sections, with the first one defining generic naming convention rules, and the second section uses those generic rules to define signing constraints.

```
#KEY: "KEY"/_/_/_v & {_v: isVer()}
#tenant: /domain/"tenant1"
#slice: #tenant/slice
#vnf: #slice/vnf
#service: #vnf/service
#dataId: _
#dataId: _/_
#dataId: _/_/_
#dataId: _/_/_/_
#dataId: _/_/_/_/_
```

The section above defines that the certificate name suffix should be four name components, with the first being the keyword component "`KEY`" and the last being a version number. Domain, tenants, network slices, VNFs and individual services within VNFs follow a hierarchical naming convention, as illustrated in Figure 7.2. Specifically, a VNF can use at most five name components as a "`<dataId>*`" in the name of VNF service data.

```
#domain_key: domain/#KEY
#tenant_key: #tenant/#KEY <= #domain_key
#slice_key: #slice/#KEY <= #tenant_key
#vnf_key: #vnf/#KEY <= #slice_key
#service_key:   #service/#KEY <= #vnf_key
#service_data: #service/#dataId <= #service_key
```

The second section defines that a domain operator has certificates, and the certified keys can sign tenant certificates, which can be used to sign individual network slices that are under the tenant control. Delegated by the corresponding network slices, each VNF has multiple service keys and each is signed by the VNF certificate and can sign the corresponding service data. Note that, the above schema does not require that all cryptographic verifications terminate at the operators. In practice, the trust schema is only executed when the data signer is not the trust anchor. A VNF under the tenant's administration is bootstrapped with the tenant self-signed certificate as the trust anchor. Therefore, for another VNF in the same tenant domain to validate service data, signature verifications terminate early at the tenant level.

The trust schema is compiled to a name schema tree as shown in Figure 7.3. Each branch of the tree represents a potential name pattern of legitimate data, and each node on a branch identifies a name component. For example, the branch (0)(1)(6)(11)(12)(13)(14)(15) defines the name pattern of a certificate of a network slice in the "`tenant1`" domain, and the branch (0)(1)(6)(11)(16)(21)(26) defines one name pattern of VNF service data. Between branches, the signing constraints indicate the specific branch for data consumers to match against which to inspect the name of data signers when validating data. A piece of VNF service data is matched to the aforementioned second branch, and the branch is associated with the VNF service certificate branch, indicating that the legitimate signer of service data must conform to this naming convention.

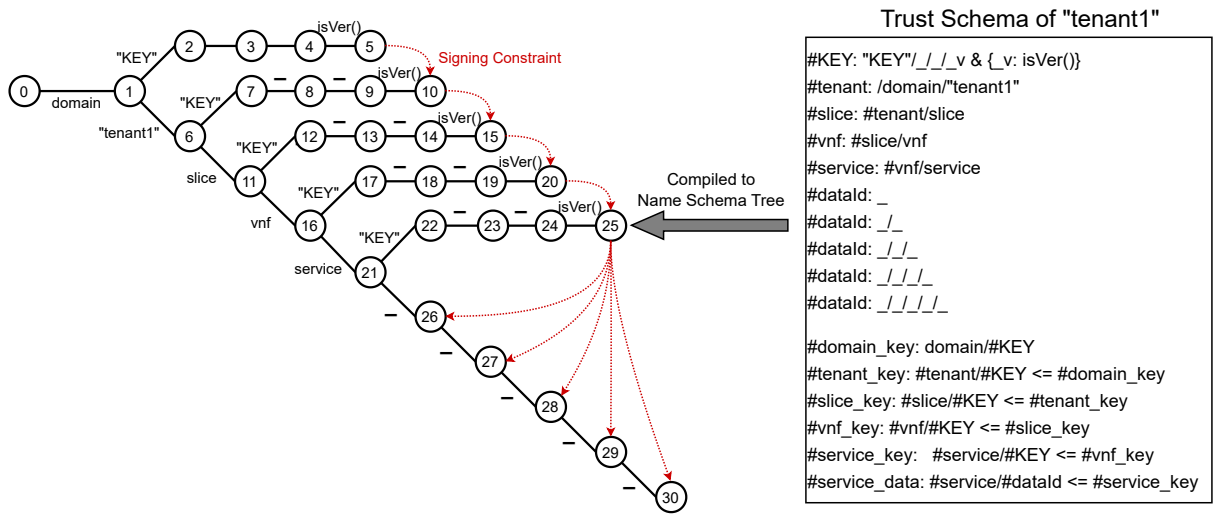


Figure 7.3: Name schema tree compiled from the trust schema of "`tenant1`". The node identifiers shown in the figure serve only as reference markers to aid explanation and have no semantic meaning within the schema itself.

7.3.3 Policy Execution

Each VNF is equipped with a TIB that is bootstrapped with the tenant self-signed certificate as the trust anchor, individual VNF and service certificates, and the trust schema.

Intra-domain VNF interations: When validating VNF service data produced in the same tenant domain with the, the Consume API (Section 3.6) traces the authentication chain through the recursive key locators and matches each signing relationship with the name schema tree. Figure 7.4 gives an example of a signing chain. "`/operaterA/tenant1/slice-0/smf-0/nsmf-pdusession/sm-contexts/t=1737072300/req-sha256=9f2c1a`" names a VNF data produced by a SMF instance when an AMF instance requests to create a PDU session. The local TIB of the consumer VNF (*i.e.,* AMF) traces the authentication chain by recursively looking and fetching the signer certificate until reaching the configured trust anchor "`/operatorA/tenant1/KEY/id123/self/v=1`", followed by signature verifications. Then it matches against every data involved in the chain to the name schema tree. For example, the

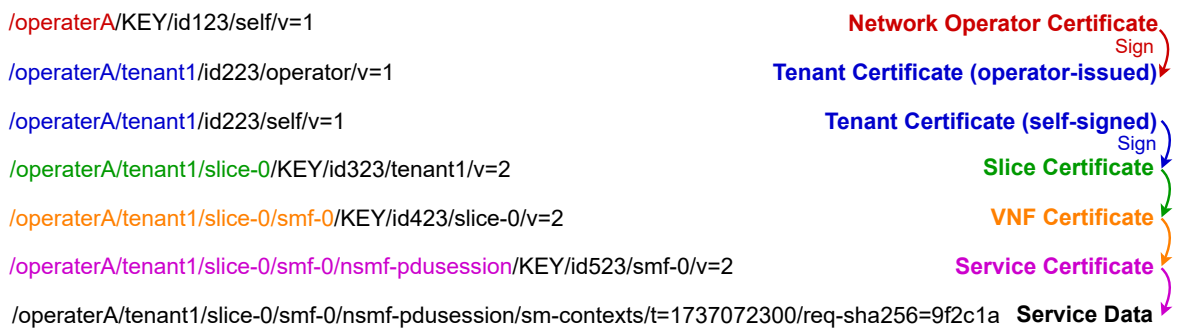


Figure 7.4: The signing chain of a service data produced by a SMF instance under "`tenant1`" control.

slice certificate "`/operatorA/tenant1/slice-0/id323/tenant1/v=1`" is a legitimate signer of "`/operatorA/tenant1/slice-0/smf-0/id423/slice-0/v=1`" because each of them can match to a branch in the name schema tree and the two branches are connected by the signing constraint between (15) and (20). Matching every pair of signer-signee relationship against the schema tree, TIB confirms that the signing chain conform to trust schema and passes the service data content for AMF for further processing.

Inter-domain VNF interations: As long as the two domains have mutually authenticated, inter-domain VNF interactions are enabled. When validating inter-domain VNF service data, the consuming VNF traverses the signer's certificate chain until it reaches the producer-domain trust anchor. The accompanying CertList (Section 3.4.2) then guides the consumer to the certificate issued by its own domain controller that authenticates the producer-domain trust anchor. As a result, authentication and authorization are handled entirely within the cellular core data plane, without relying on external PKI services or centralized entities such as the NRF. This allows VNFs to migrate and scale freely across heterogeneous domains while preserving continuous and verifiable trust relationships.

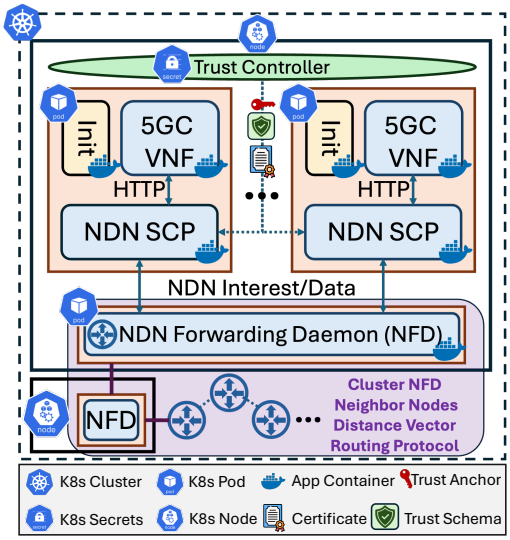


Figure 7.5: TrustNG system architecture: A Kubernetes-based 5G core deployment with NDN integration. Each VNF Pod includes an NDN-SCP sidecar container for NDN communication. NFD pods enable routing and forwarding.

7.4 Implementation

Figure 7.5 illustrates the design of the TrustNG system, which consists of three core components: (i) **VNF Pods**, each running a 5G core function; (ii) **NDN Forwarder (NFD) Pods**, providing name-based routing and forwarding; and (iii) **Trust Controller**, responsible for provisioning trust anchors, trust schemas, and identity certificates to sidecar proxies.

The core enabler of NDN functionality is the **NDN Sidecar Proxy (NDN-SCP)**, a custom-designed container deployed alongside each VNF within the same pod. Implemented using the Go-based `ndnd` library, the NDN-SCP contains the VNF TIB and translates standard HTTP-based CAPIF messages into semantically meaningful NDN Interest and Data packets, allowing VNFs to communicate using NDN without any changes to their source code. Each VNF pod establishes a persistent TCP face with the local NFD upon initialization. The sidecar proxy registers its service-specific NDN prefixes with the NFD to enable correct forwarding of Interest packets. All inter-VNF communication is mediated through the NFD,

which acts as a local router. The Trust Controller acts as the network administrator and is responsible for bootstrapping the TIB on each NDN-SCP with identity certificates, trust anchors, and trust schema. These are securely provisioned via Kubernetes secrets during pod initialization. In our deployment, we integrate the six primary 5G core VNFs: AMF, AUSF, UDM, UDR, SMF, and User Plane Function (UPF). Together, they enable UE registration and session establishment with the data network. Through this architecture, TrustNG enables semantically meaningful, sessionless, and verifiable communication among 5G core components - providing not only scalability and resilience, but also per-packet authentication and authorization as built-in properties of the NDN communication model. In the next subsection, we describe the internal design and implementation details of the NDN-SCP.

To enable HTTP-to-NDN semantic translation within the 5G core without requiring modifications of the source code, each VNF pod in TrustNG includes three components: the 5G core VNF, the NDN-SCP, and an **init container**. The init container is responsible for injecting `iptables` rules into the pod's network stack, redirecting all outgoing HTTP traffic from the VNF to the local NDN-SCP on a designated port. This approach ensures that VNF service requests are transparently intercepted and processed by the sidecar.

Figure 7.6 illustrates the internal architecture of the NDN-SCP, which consists of four primary modules: (1) **NDN Engine**; (2) **HTTP Frontend**, (3) **Consumer**, (4) **Producer**.

NDN Engine: This core component is responsible for establishing the communication interface between the sidecar and the local NFD. It registers service-specific name prefixes with the NFD to advertise the availability of VNF data. These prefixes serve as routing hints that allow the NFD to forward incoming Interest packets to the correct VNF Pod. Each VNF Pod registers the prefixes it serves during initialization, allowing the NFD to route Interests based solely on data names, without requiring endpoint addresses. For example, if a pod is responsible for the "`/operatorA/tenant1/slice-0/ausf-0/nausf-auth`" namespace, the NFD will route any incoming Interest with a name prefix that matches the producer of

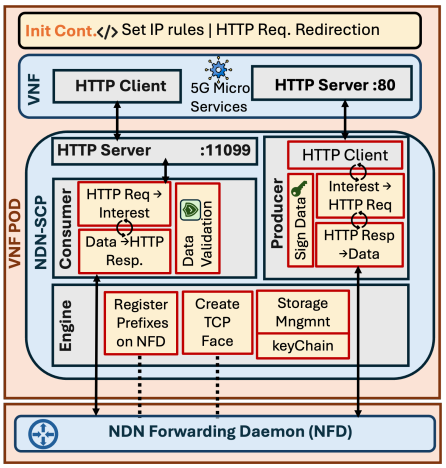


Figure 7.6: Internal architecture of the NDN-SCP. The proxy transparently intercepts HTTP traffic and translates it into semantically meaningful NDN Interest/Data exchanges.

that pod.

HTTP Frontend: The HTTP server listens to the designated sidecar port (*e.g.,* 11099) and accepts redirected HTTP requests from the VNF These requests are parsed and forwarded to the Consumer component for translation into NDN semantics.

Consumer: The consumer handles outbound request logic. Upon receiving an HTTP request, it parses the URI and converts it into a non-URI name by discarding the authority and address components and preserving only the meaningful service path (as illustrated in Figure 7.7). Then it uses the Consume API to securely fetch the VNF service data. Specifically, the service request itself is signed as separate Data encapsulated in the Interest, so that the service request can be authenticated by certificate and authorized by trust schema.

Producer: The producer first validates the encapsulated data using the bootstrapped trust schema to enforce request-level authorization. Upon successful validation, the producer reconstructs an HTTP request from the Interest and forwards it to the local VNF's HTTP server using an internal HTTP client. After the VNF processes the request and generates a response, the producer uses the Produce API to securely publish a Data containing the response content, which is eventually forwarded back the remote Consumer.

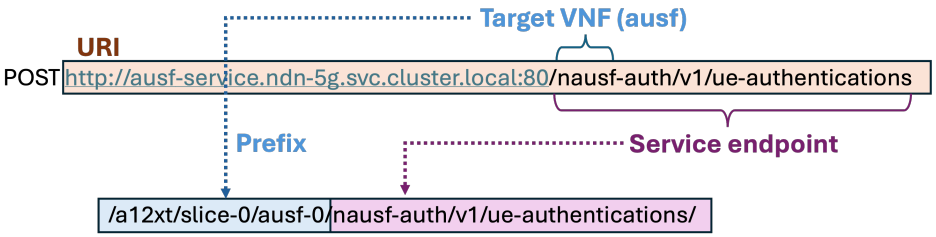


Figure 7.7: HTTP-to-NDN translation example. An HTTP request URI is semantically mapped to an NDN name.

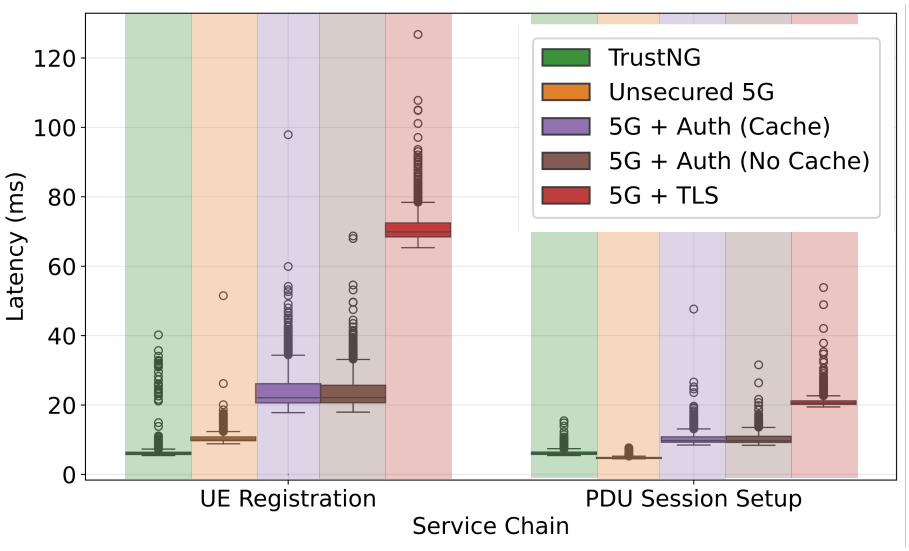
7.5 Evaluation

To evaluate the performance impact of NDN-SCP, we compare it with four baseline 5G core deployments. Our analysis spans end-to-end service latency, load resilience, resource utilization, and bandwidth consumption. The five configurations include:

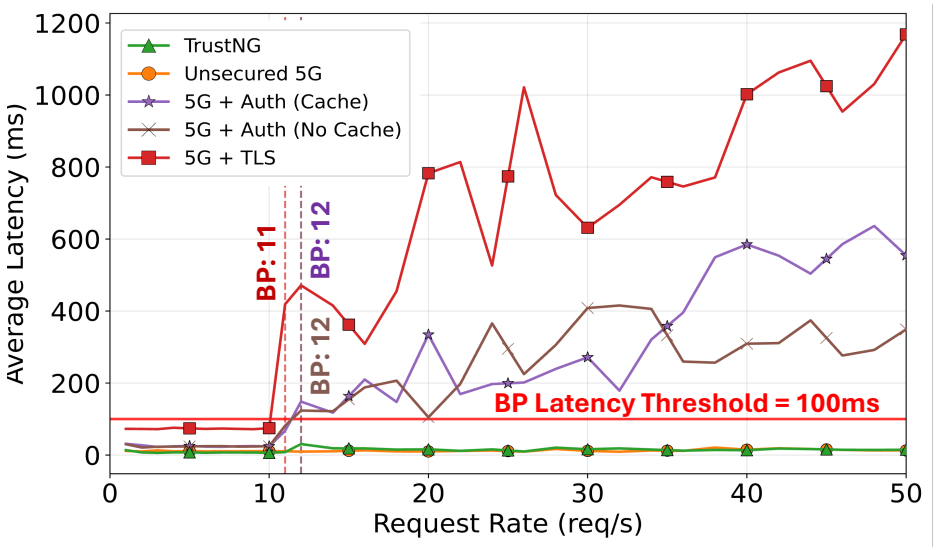
- **Unsecured 5G:** OpenAirInterface 5G core with no security (no authentication or authorization).
- **5G + TLS:** End-to-end TLS encryption using HTTP/2 sidecar proxies (highest overhead).
- **5G + Auth (C):** Token-based authorization with caching enabled.
- **5G + Auth (NC):** Token-based authorization without caching.
- **TrustNG:** Our 5G core implementation with trust plane.

7.5.1 Latency and Load Resilience

Figure 7.8a presents the end-to-end service chain latency for the UE registration and the setup of the PDU session, each triggered at a constant rate of five requests per second for 300



(a) Service chain latency (UE Reg. & PDU Setup)



(b) Average service chain latency under increasing load where line BP denotes the **Breaking Point**.

Figure 7.8: Comparison of TrustNG with traditional 5G core operational variants

seconds. In particular, NDN-SCP achieves lower latency than even Unsecured 5G for the UE registration service chain, despite performing data signing and validation. This is due to the fundamental architectural shift that NDN introduces: instead of creating a new TCP connection for each request, NDN-SCP establishes persistent TCP faces between the NDN sidecar proxies and NFDs. As a result, it avoids repeated session setup delays, including TCP and TLS handshakes.

Additionally, trust schema model validates certificates once and caches them for future use, eliminating the per-request overhead of authorization or TLS session establishment. In contrast, OAuth2 and TLS deployments require repeated access token handling or cryptographic negotiation, increasing end-to-end latency.

Figure 7.8b shows the scalability of the system under increasing load. As the UE registration request rate ramps from 1 to 50 requests per second, NDN-SCP and Unsecured 5G maintain consistent latency well below the 100 ms breaking point (BP). In contrast, 5G + TLS and 5G + Auth (C/NC) deployments exceed the BP threshold at 11-12 requests / sec, highlighting how state-based security mechanisms hinder scalability under load.

7.5.2 Resource Overhead

Figure 7.9 reports the CPU and memory usage of the AMF pod under burst loads (25 and 50 requests/sec). For each deployment, we aggregate resource usage across all containers within the pod, including VNFs and their corresponding sidecars.

Despite supporting cryptographic operations and trust validation, NDN-SCP maintains resource usage similar to vanilla 5G. This efficiency stems from persistent Interest/Data flows and the amortization of trust chain verification via in-memory caching [YMX23]. Meanwhile, TLS and token-based deployments consume significantly more memory and CPU due to connection setup, cryptographic context switching, and token management overheads.

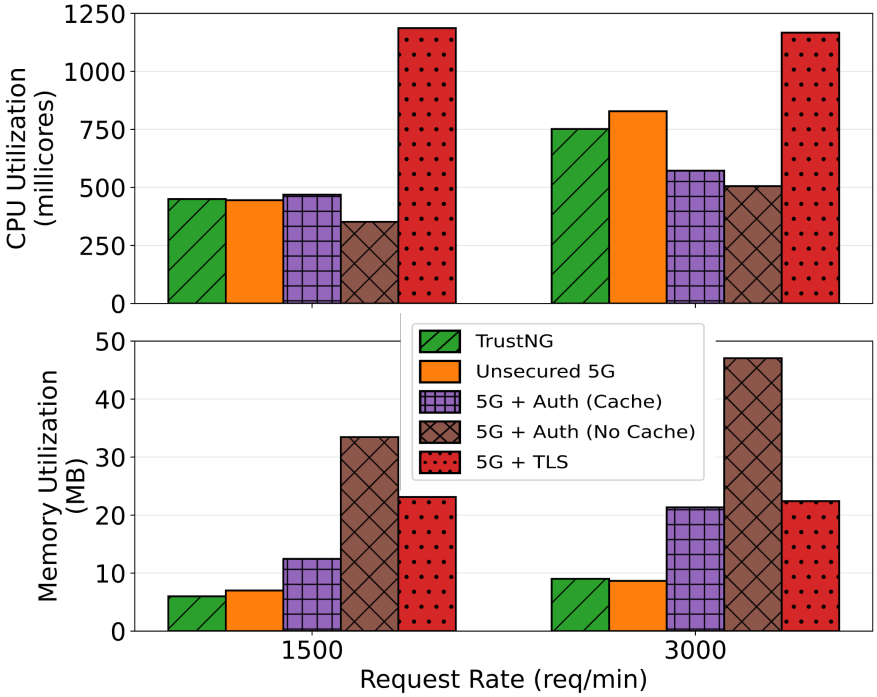


Figure 7.9: Average CPU and Memory Utilization

7.5.3 Bandwidth Utilization

We further measure total bandwidth consumption during a 600-second experiment, where each deployment handles one UE registration request per second. Table 7.1 summarizes the results.

Despite incorporating cryptographic authentication, the NDN-SCP architecture achieves total bandwidth usage nearly identical to vanilla 5G. This is a direct result of its sessionless design: by avoiding TCP and TLS handshakes, and omitting IP and connection-level headers in favor of semantically meaningful names, NDN-SCP reduces per-request overhead. Additionally, Interest names are derived from the HTTP path header, which further eliminates metadata duplication and keeps messages compact. In contrast, TLS deployments incur significantly higher traffic (over 6.4 MB total) due to session negotiation and encrypted payloads. Similarly, token-based deployments transmit over 75% more traffic than vanilla

Table 7.1: Bandwidth usage during 600 seconds of UE registration at 1 req/sec. C: Caching Access Token, NC: No Caching

Deployment	**Total (MB)**	**RX (MB)**	**TX (MB)**	**B/Req**
TrustNG	1.034	0.647	0.387	1826
Unsecured 5G	1.045	0.564	0.481	1844
5G + TLS	6.449	3.821	2.629	11385
5G + Auth (C)	1.841	0.993	0.848	3250
5G + Auth (NC)	1.848	0.997	0.850	3256

due to repeated token exchanges and validation metadata.

CHAPTER 8

Related Work

This chapter reviews previous work on concepts related to the trust plane developed in this dissertation, with particular focus on systems that incorporate the notion of *trust domains* and how their interpretations differ from the trust domains defined here. We then discuss SPKI/SDSI, a decentralized security management framework whose core principles are philosophically aligned with those of the trust plane.

8.1 Trust Domains

The term *trust domain* appears in several security and networking systems, often with different scopes and assumptions. At a high level, these designs agree that a trust domain groups entities whose identities and security policies are rooted in a common trust anchor, but they differ in how domains are structured, how they interact, and whether the concept extends uniformly to data objects, services, and users. This section discusses several uses of trust domains (or equivalent notions) in existing system designs.

SCION: SCION [PSR17a, PSR17b] is a next-generation inter-domain routing architecture that aims to provide highly available, secure end-to-end communication even in the presence of misconfigurations or malicious networks. A key abstraction in SCION is the *Isolation Domain* (ISD), which the SCION literature and documentation explicitly describe as a domain a trust. An ISD groups ASes under a common set of trust roots and governance rules. Each ISD maintains a Trust Root Configuration (TRC) that specifies which entities are trusted

to issue routing and AS certificates and how these keys may be updated or revoked [SPR16]. Thus, SCION's trust domains only govern trust relationships in the routing plane: which AS certificates and path segments are considered valid when constructing end-to-end paths. Trust domains are explicit, but their scope is fundamentally tied to inter-domain routing and path control, not to application data.

SPIFFE/SPIRE: SPIFFE (Secure Production Identity Framework For Everyone) [SPI25, SPI23] defines a framework for workload identity in cloud-native environments. Its core is the *SPIFFE ID*, a URI-like identifier (e.g., `spiffe://example.org/ns/default/sa/frontend`) bound to a workload through short-lived X.509 or JWT-based *SVIDs* (SPIFFE Verifiable Identity Documents). SPIFFE's concept of trust domain corresponds to the trust root of a SPIFFE identity provider and "could represent an individual, organization, environment, or department running their own independent SPIFFE infrastructure" [SPI25]. All SVIDs in the same trust domain are verifiable using the domain's root keys.

The SPIRE implementation of SPIFFE [CNC23] further supports federated trust domains, where one trust domain can authenticate identities issued by another by exchanging trust bundles, which are collections of X.509 root certificates (or JWT authorities). This model enables secure cross-cluster or cross-organization mTLS between workloads, while preserving administrative isolation between domains. In practice, organizations often map trust domains to Kubernetes clusters, regions, or administrative boundaries, and trust domain names are used as identity namespaces in SPIFFE IDs.

However, it focuses mainly on workload authentication. Authorization is typically handled by higher-level systems (such as Istio [RT17, Ist24] and Envoy [Env16]). These systems consume SPIFFE-issued identities as principals in their own policy languages. Moreover, authorization policies typically refer to SPIFFE IDs and service names but do not use semantic naming of either for deeper reasoning. In practice, SPIFFE-based systems treat an SPIFFE ID as an opaque identifier that represents a workload principal, and policy engines match

these identifiers as atomic strings when defining allow/deny rules. The internal structure of the SPIFFE ID is not used to codify trust relationships, nor does it express application-level semantics such as ownership or object hierarchy.

8.2 SPKI/SDSI

SPKI/SDSI [EFL01, RL98] combines the SDSI model of local, principal-defined namespaces with the SPKI certificate formats to attach names to keys and express authorization policies. In SPKI, an authorization certificate binds a public key to a set of permitted actions over a particular object or namespace, thereby specifying *who* is allowed to act and *what* they are allowed to do, and delegation can be realized by chaining such certificates. The trust plane philosophically aligns with this decentralized security design. However, SPKI/SDSI does not use names semantically. SDSI names are intentionally local to a principal's namespace, as opposed to the global DNS namespace that trust plane uses. As a result, a verifier cannot infer authorization conditions from the structure of names; instead, it must interpret the rights explicitly encoded in authorization certificates.

CHAPTER 9

Discussion

This chapter discusses several implications of the trust plane by clarifying its applicability to network infrastructure security, performance considerations for signing data, and independence from the underlying networking architecture. Because a number of works referenced in earlier chapters were initiated before the trust plane concept was fully developed, we also provide a retrospective that revisits those designs.

9.1 Infrastructure Security

The trust plane is developed primarily for applications, but its abstractions can also be applied to network infrastructures. From the perspective of the trust plane, infrastructure components, such as routing, can be modeled as applications that produce and consume named secured data and therefore can participate in trust domains in the same way. When secured data can be delivered to the intended recipient, many traditional firewall functions become unnecessary, since correctness no longer depends on constraining packet paths or filtering traffic by origin.

9.2 Cost of Signing Data

The trust plane requires cryptographic signatures on data objects, which may raise concerns about computational overhead. In practice, this cost is manageable. Contemporary processors provide hardware support for modern signature schemes, so that optimized software

implementations can achieve sufficient data signing and verification throughput for practical deployments. Thus, per-object signing and verification do not impose a prohibitive cost.

For large or streaming data, applications may avoid signing each fragment using a manifest structure. A manifest is a compact metadata object that lists the hashes of all segments of a larger data set. Rather than signing each segment individually, the producer signs the manifest once. Consumers verify the signature on the manifest and then validate each segment by checking that its hash matches the value recorded in the manifest. This approach authenticates the entire dataset while requiring only a single signature operation, and can be easily supported by the Produce/Consume API.

9.3 Independence from Networking Architecture

Although this dissertation realized trust plane on top of NDN, the design is not dependent on it. A trust plane implementation only requires communication supports for named secured data and any system that supports these minimal requirements can serve as the underlying transport substrate. Technically, any message broker (RabbitMQ or ActiveMQ) and Pub-/Sub system can carry named secured data. NDN offers advantages such as name-based newtork forwarding and in-network caching, but these features are not prerequisites for the trust plane. The trust plane can operate in the same way regardless of how named secured data is routed or delivered in the network.

9.4 Evolution of Several Works

Several works referenced in earlier chapters were initiated before the trust plane concept was fully developed, and our understanding of these topics has matured over time. For example, certificate revocation records could have been delivered directly to domain entities through trust schema updates (as discussed in Section 5.4), rather than being submitted

to a ledger for data availability. CLedger was originally designed to address the certificate *availability* problem through immutable replication; only later did we recognize that data availability and data auditing are distinct requirements and that a ledger should focus on the latter. Its DAG structure was influenced by the earlier DLedger [ZVK19], which operated under a different trust model. Subsequent analysis showed that maintaining a DAG is unnecessary: synchronized loggers that each maintain a local Merkle tree are sufficient for providing tamper-evident auditing. At the time of writing, we are actively advancing the ledger design in this direction.

CHAPTER 10

Other Contribution: In-Network Data Repository

This chapter is a brief overview of the author's contributions to the NDN ecosystem not described previously but was used to support the work described in the previous chapters. We describe an in-network data repository (Repo) that we developed to support asynchronous data-centric communications and its usage for applications.

NDN Repos are application processes running on nodes with storage resources to provide persistent storage for other applications. Repos are *managed* in-network storage system, which ensure data packet availability until data are evicted upon request by their applications. They accept data insertion requests, fetches named data objects from requesters, and makes data available; meanwhile, they can be transparent to data consumers, who simply fetch desired data by names, without needing to know where the data come from.

10.1 Repo Design

In this section, we first define the basic operations of Repo and then describe our design assumptions and goals. Afterwards, we give an overview of the the Repo workflow, followed by the Repo operations details.

Repo Operations: Repo runs as an application process on nodes with storage resources. It interacts with *users*, a generic term we use to refer to NDN entities that use repos by inserting or deleting data objects. An observation we make from existing NDN applications, such as those described in [DAT22, PWB22], produces application data objects of various

sizes; each object may be segmented to multiple Data packets. We refer to the application data object as the Application Data Unit (ADU) [CT90]. Repo uses ADU as the basic data unit in its operations.

Design Assumptions: We assume that both Users and Repo go through a bootstrapping process before starting operations. Therefore, they possess the necessary security parameters to secure and validate the data exchange between each other. Consumers express Interests to fetch desired data from the network. They validate received data following the security policies defined by their applications, independently of where the Data are retrieved.

Design Goals: Repo has the following two design goals:

- **User Authenticity and Authorization:** Repo should accept ADU insertion and deletion requests from authenticated and authorized Users only.
- **ADU Availability:** after a User successfully inserts an ADU, Repo should keep this ADU available persistently.

Design Overview: Repo takes ADU insertion and deletion requests from the application and performs the corresponding tasks. Since the request must carry the necessary ADU information that Repo needs to know, it should be a piece of semantically named and secured data that Repo fetches from the application. Therefore, it is the user application that initiates the ADU insertion or deletion process by notifying Repo there is a new request to be processed. Upon receiving the request, Repo checks whether the request is produced by an authorized User through validating the request with the bootstrapped trust schema. If the request is signed by an authorized User, Repo proceeds to fetch the ADU from the network with the information provided within an insertion request, or delete the ADU from its local storage for a deletion request. After sending an ADU Insertion Request to Repo, the User can optionally check whether the ADU is ready for the Consumer to retrieve. When Repo is ready to serve the ADU, Consumers can fetch individual segments of the ADU as fetching normal Data packets from the network.

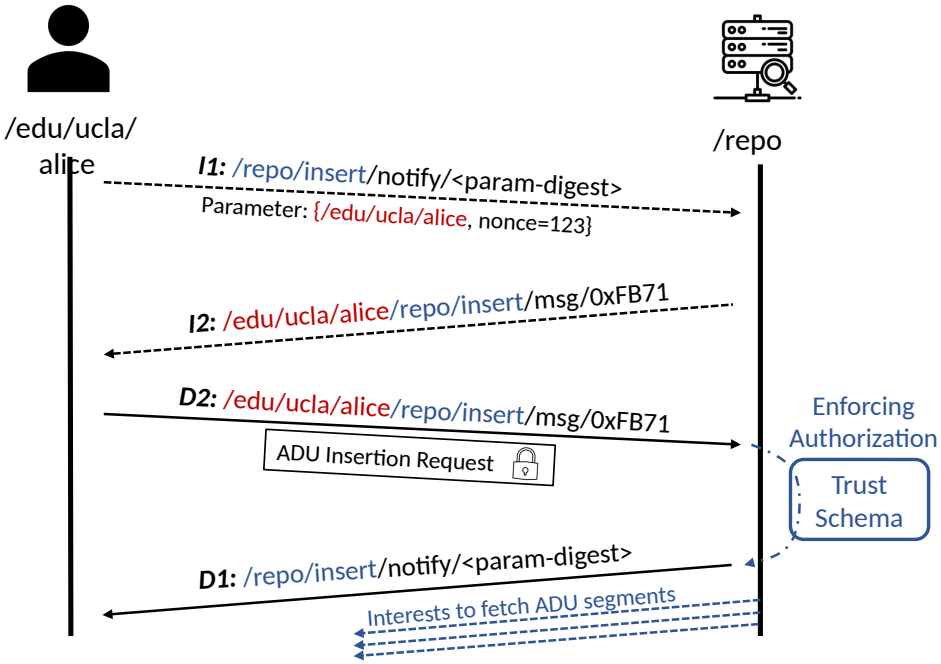


Figure 10.1: Alice sends Object Insertion Commands to Repo

10.2 Repo in Operation

In the rest of this section, we use an example to demonstrate Repo's protocol design. Assuming the building manager Alice "`/edu/ucla/alice`" monitors offices with smart sensors. The monitor system produces sensor data every hour, and pushes sensor data to a Repo named "`/repo`", to be fetched by a remote data analytics application.

Inserting ADU into Repo: In order to insert data to Repo, Alice first prepares an ADU Insertion Request that informs "`/repo`" *what are the ADU names and where to fetch ADUs.* The naming convention of the request is "`/<user-prefix>/<repo-prefix>/<operation>/msg/<nonce>`". The prefix "`<user-prefix>`" and "`<repo-prefix>`" are the User prefix and Repo prefix, respectively, and "`<operation>`" represents the operation name which is "`insert`" for insertion. The name component "`<nonce>`" is a 32-bit randomly generated number uniquely identifying the request. As shown in Figure 10.2, Alice puts two *Insertion Request Pa-*

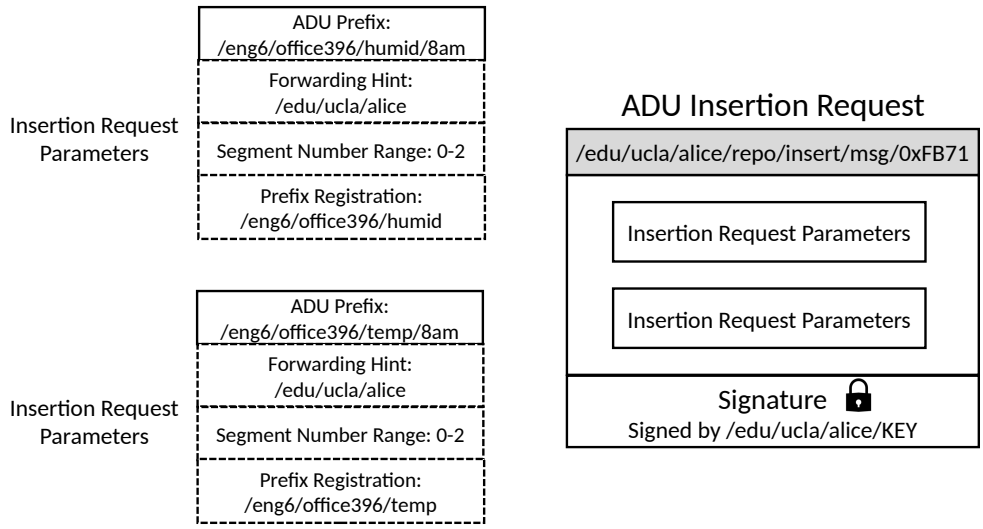


Figure 10.2: ADU Insertion Request and Insertion Request Parameters

rameters blocks into an ADU Insertion Request. Each parameter block includes the request description for an ADU[1]. The first block specifies the ADU prefix of "`/eng6/office365/humid/8am`". The block has segment number range of "0-3", indicating that this ADU has four segments in total and the segment number starts from zero; The forwarding hint "`/edu/ucla/alice`" instructs Repo to fetch this ADU from Alice; The prefix registration field instructs Repo to register the name prefix "`/eng6/office365/humid`" in order to serve this ADU to consumers. Finally, Alice signs the request with the private key "`/edu/ucla/alice/KEY`".

After preparing the request, Alice initiates the ADU insertion process by first expressing a notification Interest $I1$ to Repo's insertion prefix "`/repo/insert`" with the application parameters carrying Alice's prefix "`/edu/ucla/alice`" and request nonce "`0xFB71`".

On receiving $I1$, Repo learns Alice's prefix and the nonce "`0xFB71`" that uniquely identifies her request, expresses Interest $I2$ to fetch Alice's ADU Insertion Request $D2$, and validates the request's authenticity and legitimacy using its trust schema. For example, if the trust schema allows keys under the prefix "`/edu/ucla/<user>`" to be the legitimate signers for

[1]Deletion Request Parameters follow a similar structure, but without the forwarding hint [**?**] and prefix registration fields.

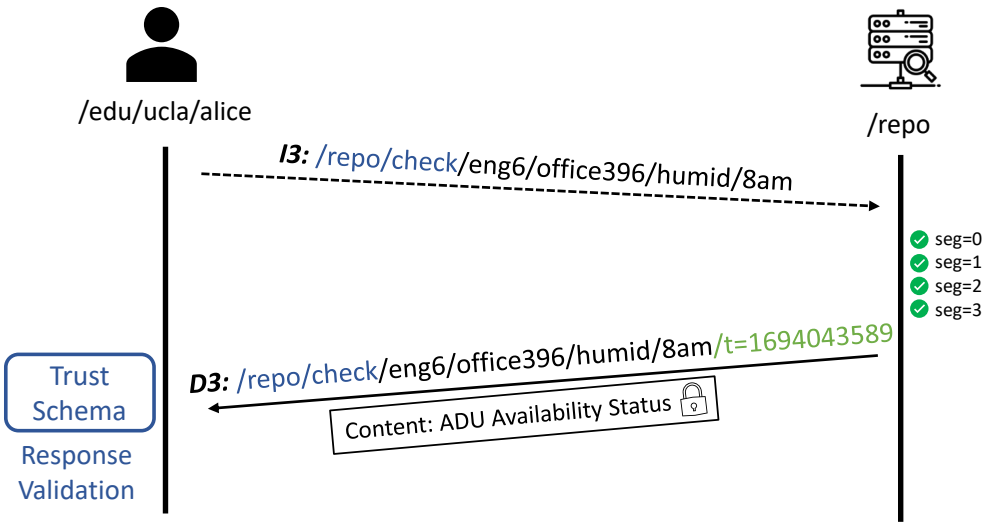


Figure 10.3: Checking ADU Availability Status

Data under the prefix "`/edu/ucla/<user>/repo/insert`", then Alice is authorized by the trust schema, thereby a legitimate User to insert ADUs in Repo. If the request validation succeeds, Repo replies to $I1$ with $D1$ with empty content, and begins fetching the ADU that Alice has requested to insert.

Checking ADU Availability: Since Repo processes requests asynchronously, it needs to provide a mechanism for its Users to check whether an insertion request has succeeded (i.e., all inserted ADUs have become available), or if the request has failed due to ADU fetching failure[2], unauthorized requests, or full storage. To this end, Repo allows the Users to check ADU availability using commands under the prefix "`/<repo-prefix>/check/<adu-prefix>`", where the suffix "`<adu-prefix>`" is the ADU prefix Alice wants to check.

In the example as shown in Figure 10.3, Alice checks the ADU availability "`/eng6/office396/humid/8am`" by expressing an Interest $I3$ following the aforementioned naming convention. As Repo receives $I3$, it checks its local database to see if all ADU segments are available and returns the signed ADU availability status code in $D3$.

[2]Repo will perform basic retransmissions up to a certain number of times to overcome packet losses.

Upon receiving $D3$, Alice validates it using its trust schema to ensure that the data is indeed produced by Repo. Therefore, if Alice's insertion request triggers errors, she can learn the reason from the status code.

Fetching Data from Repo Consumers send Interests to retrieve individual ADU segments from Repo in the same way as fetching Data packets from the original data producers. Specifically, Consumers express Interests that are attached a forwarding hint containing the Repo prefix "`/repo`" to the Interests, so that the network is able to steer the Interests with the ADU Interest prefix to the repo.

Note that attaching forwarding hints to Interests is not encapsulation. Encapsulated packets do not benefit from NDN's in-network caching. However, even with forwarding hints, the original Interest name is still visible to the forwarders. Hence, when the Data packet matching the Interest name follows the reverse path back to the Consumers, it can be cached by intermediate routers and match other Interests with the same name.

This approach requires the original ADU producer (not necessarily the User who inserts the ADU to Repo) to inform the Consumers of the Repo prefix out-of-band, so that the Consumers can attach forwarding hints to Interests. We argue that applications are responsible for conveying the forwarding hints for Consumers.

10.3 Becoming Application Oblivious

As demonstrated in Section 10.2, Repo can provide application-agnostic storage in the network. However, applications must explicitly send insertion requests with *specific data names*. In this section, we describe how Repo can be application-oblivious by using NDN Sync.

Oblivious Sync Participation: As described in Section 2.5, multi-party applications (such as Ownly) can utilize NDN Sync to exchange the latest data production information among end users. Therefore, Repo can learn the latest synchronization states of an Owly workspace instance by joining the application's Sync group G to learn about all data productions.

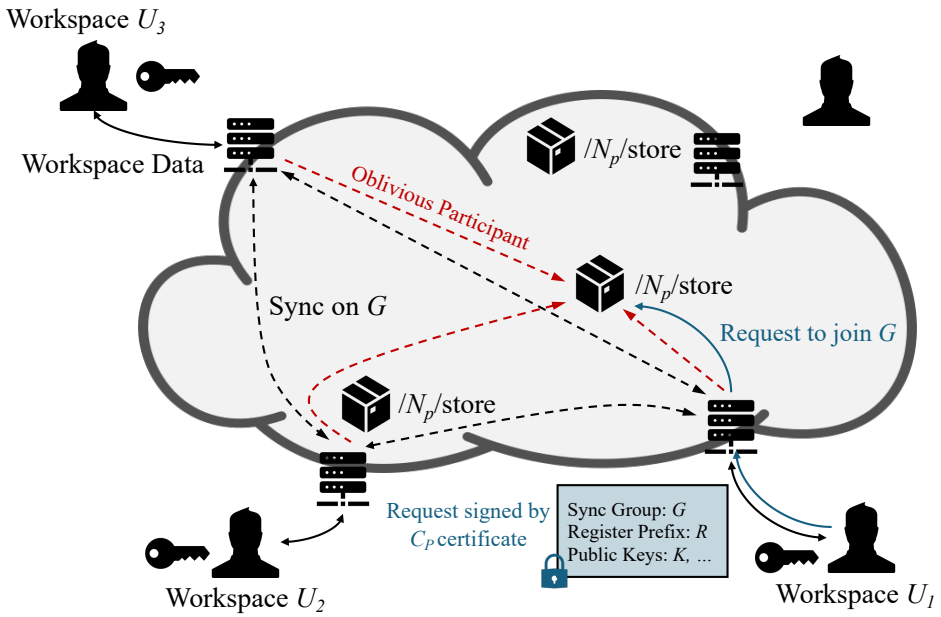


Figure 10.4: In network P, three Ownly workspaces users are running NDN Sync over group G. Repo is an oblivious participant in G, as requested by U_1.

Whenever Repo learns that there is a new publication in the group G, it expresses Interest to fetch and store the latest published data D. Whenever an user wants to fetch D, Repo can directly reply with D from its storage. In this process, Repo is a passive participant in a Sync group – it neither generates Sync Interest nor produces new data packets, making the application unaware of the existence of Repo.

Requesting Repo Joining Sync: Any customer of a network P, referred to as C_p, can make use of Repo provided by the network P by sending a Repo service request to ask Repo (i) to join a synchronization group G; (ii) to retrieve all data published in G; and (iii) to serve them under a prefix for the data name R. The request contains the name of G, the data name prefix R, and all necessary certificates that can be used to verify application Data and Sync Interests (to be discussed shortly). Repo validates each received service request by C_p certificate, and starts joining synchronization group G. Requesting Repo to leave G follows a similar procedure.

10.4 Routing and Data Security:

Repo verifies the ownership of the name for the registration of the data prefix and only registers the data prefixes with the routing system when customers of the network P can prove the ownership of the name prefix. Therefore, all data prefix registrations Repo makes come from the original owners of the prefixes.

Repo operates according to the security policies installed by its operators (*e.g.,* network providers). Whenever receiving G Sync Interest and the application Data published in it, Repo always matches the signing key with authenticated keys obtained from the service request and verify the signature. Repo performs data authenticity verification when fetching published data to prevent storage pollution. The default authentication policy is to compare the Data name with the signing key name to check whether the signer is the data prefix owner. Additionally, Repo can also take C_p defined specific application policies carried in the Repo service request to further refine the Data authentication verification. Note that, although Repo may execute application-specific policies, the policy execution is application-agnostic, since Repo only performs trust schena which is generic to all applications.

10.5 Supporting Fully Asynchronous Applications

In addition to store all application-produced data, to help newly arrived users quickly learn about the latest data production, Repo also keeps the latest Sync Interest it has received for each application group, which represents the latest data production state of an application. When Repo receives Sync Interests carrying outdated state vectors, which are likely produced by users who are just back online, it re-sends the latest Sync Interest to inform users of the latest data production that they may have missed when offline. Such Sync Interest retransmissions do not lead to replay attacks, as receiving duplicate Sync Interest has no effect on user applications.

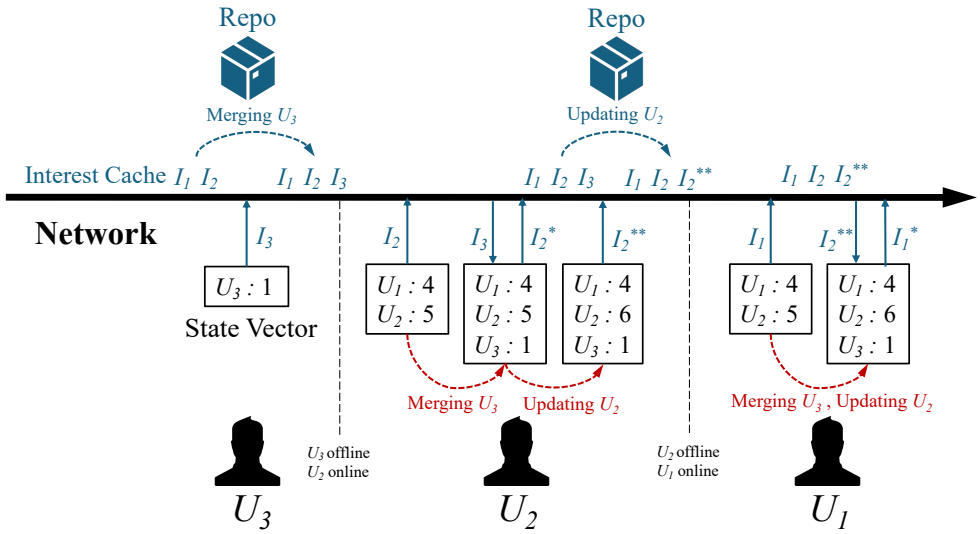


Figure 10.5: An example where three Ownly users fully asynchronously collaborate by Repo updating Interest cache and replaying them upon updated Sync Interests.

The example in Figure 10.5 illustrates how Repo enables the Ownly to be fully asynchronous. Suppose that there are three users collaboratively editing a document, and U_3 is a new user in G who is not aware of the other two, who are already synchronized with each other.

U_3 first connects to the network, makes a change to the document, and then disconnects. Its Workspace instance multicasts a Sync Interest I_3 to the network, indicating the existence of U_3 with the sequence number 1. Repo adds I_3 into its cache since this Interest packet updates G's state, and expresses an Interest to fetch U_3's data "`/G/`U_3`/DATA/seq=1`". Later, U_2 comes online, it also multicasts its last Sync Interest I_2 to synchronize with the network, and receives I_3 obliviously replayed by Repo. Since this Interest carries a new entry of state vectors, U_2 merges the state vectors and sends a new Sync Interest I_2^* to the network. Afterwards, it fetches "`/G/`U_3`/DATA/seq=1`" as indicated by the latest vector, makes its change to the document, and sends I_2^{**} that contains updated state vectors to the network. Finally, when U_1 goes online, it will discover I_2^{**} with the latest state vectors, and fetches "`/G/`U_3`/DATA/seq=1`" and "`/G/`U_2`/DATA/seq=6`".

Figure 10.6: The topology map of global NDN Testbed. Repo is deployed on every Testbed router.

In this process, each user makes document changes independently and asynchronously. As there were no other instances or third-party servers users could synchronize with when publishing, each user instead obliviously synchronizes with Repo. When the application is connected to the network P, the NDN Sync states always inform everyone of the latest publications, as memorized by the network.

10.6 Implementation and Deployment

Since 2020, Repo has been used in multiple projects, including smart home data storage [ZYM22], power plant sensor data management, mobile health applications [DAT22] and Ownly. The NDN Testbed [Nam22] also has globally deployed Repo instances on each site, serving as storage within the network for NDN applications. The design and implementation of Repo also inspired an ongoing Hydra project [PWB22], which aims to provide federated storage for genetic researchers.

CHAPTER 11

Conclusion and Future Work

Motivated by the fragmented security solution landscape of today's Internet, we contend that what is fundamentally lacking is a framework that systematically supports the core security mechanisms: authentication, authorization, and auditing (the 3As). This dissertation develops the *trust plane* framework to facilitate the systematic design of Internet security solutions grounded in the 3As.

The Internet establishes physical connections among numerous administrative domains. We define each administrative domain as a trust domain and model trust relationships between trust domains as links, thereby forming a trust plane. We use semantic names to identify all cyberspace entities and the data they generate, and we capture and formalize trust relationships by developing a Trust Information Base (TIB) for each entity. The TIB contains all the necessary information for implementing the 3As. Additionally, following the data-centric approach of Named Data Networking (NDN), we adopt named, secured data as the fundamental building block for implementing the trust plane.

The TIB stores trust anchors, certificates, and schematized trust policies at each entity. To bootstrap TIBs with these security parameters, we design and implement a four-step bootstrapping process. This bootstrapping process has been successfully applied to the development of bootstrapping solutions for a federated file storage system and a decentralized collaborative editor. To support trust plane maintenance, we develop *CertRevoke* as a certificate revocation solution and *CLedger* as an initial step towards supporting system auditing. Finally, we apply the trust plane framework to secure the cellular core network, and our

solution significantly reduces security complexity and operational overhead.

Looking ahead to the future development of the trust plane, we identify system auditing as a priority. Although CLedger provides an initial component to support auditing, designing a comprehensive, immutable logging system that meets broad audit requirements remains an open research challenge. Furthermore, immutable logs only provide data for ongoing, independent audits; conducting the actual audit analysis presents a greater challenge but also offers opportunities for innovation, such as developing new solutions to audit the auditors and to incorporate AI agents into the auditing process.

In conclusion, this dissertation presents a pathway for systematically developing Internet security solutions and offering applications with integrated support for authentication, authorization, and auditing. We encourage all interested parties to review the trust plane framework concept and utilize its initial implementations when developing their own security solutions.

REFERENCES

[AAL05] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. "DNS Security Introduction and Requirements." RFC 4033, March 2005. https://doi.org/10.17487/RFC4033.

[ABC19] Josh Aas, Richard Barnes, Benton Case, Zakir Durumeric, Peter Eckersley, Alan Flores-López, J Alex Halderman, Jacob Hoffman-Andrews, James Kasten, Eric Rescorla, et al. "Let's Encrypt: an automated certificate authority to encrypt the entire web." In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2473–2487, 2019.

[ARW18] Alexander Afanasyev, Tamer Refaei, Lan Wang, and Lixia Zhang. "A Brief Introduction to Named Data Networking." In *Proc. of IEEE MILCOM*, October 2018.

[BHM19a] R. Barnes, J. Hoffman-Andrews, D. McCarney, and J. Kasten. "Automatic Certificate Management Environment (ACME)." RFC 8555, RFC Editor, March 2019. https://www.rfc-editor.org/rfc/rfc8555.txt.

[BHM19b] Richard Barnes, Jacob Hoffman-Andrews, Daniel McCarney, and James Kasten. "Automatic certificate management environment (acme)." Technical report, 2019.

[BSP08] Sharon Boeyen, Stefan Santesson, Tim Polk, Russ Housley, Stephen Farrell, and David Cooper. "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile." RFC 5280, May 2008. https://www.rfc-editor.org/info/rfc5280.

[CL99] Miguel Castro and Barbara Liskov. "Practical Byzantine Fault Tolerance." In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, p. 173–186, USA, 1999. USENIX Association.

[Cla88] David Clark. "The design philosophy of the DARPA Internet protocols." In *Symposium proceedings on Communications architectures and protocols*, pp. 106–114, 1988.

[CNC23] CNCF SPIRE Project. "SPIRE: The SPIFFE Runtime Environment." https://spiffe.io/spire, 2023. Accessed: 2025-01-01.

[CT90] David D. Clark and David L. Tennenhouse. "Architectural Considerations for a New Generation of Protocols." In Phil Karn, editor, *Proceedings of the ACM Symposium on Communications Architectures & Protocols (SIGCOMM)*, pp. 200–208. ACM, 1990. https://doi.org/10.1145/99517.99553.

[DA99] Tim Dierks and Christopher Allen. "The TLS Protocol Version 1.0." RFC 2246, January 1999. https://www.rfc-editor.org/rfc/rfc2246.

[DAT22] Saurab Dulal, Nasir Ali, Adam Robert Thieme, Tianyuan Yu, Siqi Liu, Suravi Regmi, Lixia Zhang, and Lan Wang. "Building a secure mHealth data sharing infrastructure over NDN." In *Proceedings of the 9th ACM Conference on Information-Centric Networking*, ICN '22, pp. 114–124, New York, NY, USA, 2022. Association for Computing Machinery. https://dl.acm.org/doi/10.1145/3517212.3558091.

[EFL01] Carl Ellison, Bill Frantz, Butler Lampson, Ronald L. Rivest, Brian M. Thomas, and Tatu Ylonen. "SPKI Certificate Theory." *Journal of Computer Security*, **9**(3):215–264, 2001.

[Env16] Envoy Proxy Contributors. "Envoy Proxy: Layer 7 Edge and Service Proxy." https://www.envoyproxy.io/, 2016. Accessed: 2025-12-03.

[FLT13] Seyed Kaveh Fayazbakhsh, Yin Lin, Amin Tootoonchian, Ali Ghodsi, Teemu Koponen, Bruce Maggs, KC Ng, Vyas Sekar, and Scott Shenker. "Less Pain, Most of the Gain: Incrementally Deployable ICN." In *Proceedings of the ACM SIGCOMM 2013 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '13, pp. 147–158. ACM, 2013. https://doi.org/10.1145/2486001.2486023.

[Gar95] Simson Garfinkel. *PGP: pretty good privacy.* " O'Reilly Media, Inc.", 1995.

[Goo15] Google Security Team. "Maintaining digital certificate security: Google's decision on CNNIC." https://security.googleblog.com/2015/03/maintaining-digital-certificate-security.html, 2015. Google's announcement on removing trust in CNNIC certificates after misissuance events.

[HAA13] AKM Mahmudul Hoque, Syed Obaid Amin, Adam Alyyan, Beichuan Zhang, Lixia Zhang, and Lan Wang. "NLSR: Named-data link state routing protocol." In *Proceedings of the 3rd ACM SIGCOMM workshop on Information-centric networking*, pp. 15–20, 2013.

[Har12] David Hardt. "The OAuth 2.0 Authorization Framework." RFC 6749, October 2012. https://www.rfc-editor.org/rfc/rfc6749.

[HS12] Paul Hoffman and Jakob Schlyter. "The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA." RFC 6698, August 2012. https://www.rfc-editor.org/rfc/rfc6698.

[IBM25] IBM Corporation. "IBM Security QRadar SIEM." https://www.ibm.com/products/qradar-siem, 2025.

[Ist24] "Istio Security: Trust Domains and Trust Domain Aliases." https://istio.io/latest/docs/ops/configuration/security/, 2024. Official documentation.

[Lad23] Watson Ladd. "SPAKE2, a Password-Authenticated Key Exchange." RFC 9382, September 2023. https://www.rfc-editor.org/info/rfc9382.

[Lam04] Butler W Lampson. "Computer security in the real world." *Computer*, **37**(6):37–46, 2004.

[Lau14] Ben Laurie. "Certificate transparency: Public, verifiable, append-only logs." *Queue*, **12**(8):10–19, 2014.

[LCL17] James Larisch, David Choffnes, Dave Levin, Bruce M. Maggs, Alan Mislove, and Christo Wilson. "CRLite: A Scalable System for Pushing All TLS Revocations to All Browsers." In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pp. 719–736. IEEE, 2017.

[LCP20] Yixin Li, Bin Cao, Mugen Peng, Long Zhang, Lei Zhang, Daquan Feng, and Jihong Yu. "Direct acyclic graph-based ledger for Internet of Things: Performance and security analysis." *IEEE/ACM Transactions on Networking*, **28**(4):1643–1656, 2020.

[LK12a] Ben Laurie and Emilia Kasper. "Revocation transparency." *Google Research, September*, **33**, 2012.

[LK12b] Matt Lepinski and S Kent. "RFC 6480: an infrastructure to support secure Internet routing.", 2012.

[LLK13] Ben Laurie, Adam Langley, and Emilia Kasper. "Certificate Transparency." RFC 6962, June 2013. https://www.rfc-editor.org/rfc/rfc6962.

[LMZ21] Siqi Liu, Philipp Moll, and Lixia Zhang. "Mnemosyne: An Immutable Distributed Logging Framework over Named Data Networking." In *Proceedings of the 8th ACM Conference on Information-Centric Networking*, ICN '21, p. 130–132, New York, NY, USA, 2021. Association for Computing Machinery. https://doi.org/10.1145/3460417.3483375.

[LZW19] Yanbiao Li, Zhiyi Zhang, Xin Wang, Edward Lu, Dafang Zhang, and Lixia Zhang. "A secure sign-on protocol for smart homes over named data networking." *IEEE Communications Magazine*, **57**(7):62–68, 2019.

[MAZ17] Manika Mittal, Alexander Afanasyev, and Lixia Zhang. "NDN Certificate Bundle (Version 0.1)." Tech. report, University of California, Los Angeles, 2017. Version 0.1.

[Mic24] Microsoft Corporation. "What is Zero Trust?", 2024. Accessed: 2025-11-10. https://learn.microsoft.com/en-us/security/zero-trust/zero-trust-overview.

[Moz18] Mozilla Security Blog. "Distrust of Symantec TLS Certificates." https://blog.mozilla.org/security/2018/03/12/distrust-symantec-tls-certificates/, 2018. Official announcement of phasing out Symantec-issued certificates due to misissuance incidents.

[MPW22] Philipp Moll, Varun Patil, Lan Wang, and Lixia Zhang. "SoK: The evolution of distributed dataset synchronization solutions in NDN." In *Proceedings of the 9th ACM conference on information-centric networking*, pp. 33–44, 2022.

[MPZ21] Philipp Moll, Varun Patil, Lixia Zhang, and Davide Pesavento. "Resilient brokerless publish-subscribe over ndn." In *MILCOM 2021-2021 IEEE Military Communications Conference (MILCOM)*, pp. 438–444. IEEE, 2021.

[Nam22] Named Data Networking Project. "NDN Testbed." https://named-data.net/ndn-testbed/, 2022. Accessed August 10, 2025.

[Nic19] Kathleen Nichols. "Lessons learned building a secure network measurement framework using basic NDN." In *Proceedings of the 6th ACM Conference on Information-Centric Networking*, pp. 112–122, 2019.

[Nic21] Kathleen Nichols. "Trust schemas and icn: key to secure home iot." In *Proceedings of the 8th ACM Conference on Information-Centric Networking*, pp. 95–106, 2021.

[NT24] Technical Specification Group Core Network and Terminals. "5G System; Application Function Event Exposure Service; Stage 3." TS 29.517 V19.0.0 , 3rd Generation Partnership Project (3GPP), December 2024.

[OAS05] OASIS Security Services Technical Committee. "Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2.0." https://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf, 2005. Accessed June 18, 2025.

[Ope24] Open Information Security Foundation. "Suricata: The Open Source IDS/IPS/NSM Engine." https://suricata.io/, 2024. Accessed: 2025-02-16.

[Pal25] Palo Alto Networks. "Cortex XSOAR." https://www.paloaltonetworks.com/cortex/xsoar, 2025.

[Pri11] J. Prins. "DigiNotar Certificate Authority breach." In *Black Hat Conference*, 2011. Detailed report on the DigiNotar compromise and fraudulent certificates. https://www.blackhat.com/html/bh-eu-11/bh-eu-11-archives.html.

[PSM22] Davide Pesavento, Junxiao Shi, Kerry McKay, and Lotfi Benmohamed. "PION: Password-based IoT onboarding over named data networking." In *ICC 2022-IEEE International Conference on Communications*, pp. 1070–1075. IEEE, 2022.

[PSR17a] Adrian Perrig, Pawel Szalachowski, Raphael M. Reischuk, and Laurent Chuat. "SCION: A Secure Internet Architecture." In *Communications of the ACM*, volume 60, pp. 56–65. ACM, 2017.

[PSR17b] Adrian Perrig, Pawel Szalachowski, Raphael M. Reischuk, and Laurent Chuat. *SCION: A Secure Internet Architecture*. Springer, 2017.

[PWB22] Justin Presley, Xi Wang, Tym Brandel, Xusheng Ai, Proyash Podder, Tianyuan Yu, Varun Patil, Lixia Zhang, Alex Afanasyev, F Alex Feltus, et al. "Hydra–A Federated Data Repository over NDN." *arXiv preprint arXiv:2211.00919*, 2022.

[RBM20] Scott Rose, Oliver Borchert, Stu Mitchell, and Sean Connelly. "Zero Trust Architecture." Technical Report NIST Special Publication 800-207, National Institute of Standards and Technology, Gaithersburg, MD, August 2020. `https://doi.org/10.6028/NIST.SP.800-207`.

[RL98] Ronald L. Rivest and Butler Lampson. "SDSI – A Simple Distributed Security Infrastructure." *Journal of Computer Security*, **6**(1-2):3–48, 1998. Published 1999.

[ROS23] E. Rescorla, K. Oku, N. Sullivan, and M. Thomson. "TLS Certificate Compression." RFC 9364, RFC Editor, February 2023. `https://www.rfc-editor.org/rfc/rfc9364.txt`.

[RPA20] Sanjeev Kaushik Ramani, Proyash Podder, and Alex Afanasyev. "NDNViber: Vibration-Assisted Automated Bootstrapping of IoT Devices." In *2020 IEEE 21st International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, pp. 1–6. IEEE, 2020.

[RT17] Louis Ryan and Varun Talwar. "Istio: A Modern Approach to Microservice Connectivity." In *Usenix ATC (Invited Talk)*, 2017.

[SA24] Technical Specification Group Services and System Aspects. "System Architecture for the 5G System (5GS); Stage 2." TS 23.501 V19.0.0 , 3rd Generation Partnership Project (3GPP), June 2024.

[SBJ14] N. Sakimura, J. Bradley, M.B. Jones, B. de Medeiros, and C. Mortimore. "OpenID Connect Core 1.0 incorporating errata set 1.", 2014. `https://openid.net/specs/openid-connect-core-1{_}0.html`.

[SBL16] Wentao Shang, Adeola Bannis, Teng Liang, Zhehao Wang, Yingdi Yu, Alexander Afanasyev, Jeff Thompson, Jeff Burke, Beichuan Zhang, and Lixia Zhang. "NDNFit: An Open mHealth Application Built on Named Data Networking." https://escholarship.org/uc/item/8h8950n3, 2016. Accessed August 10, 2025.

[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, June 2013. https://www.rfc-editor.org/info/rfc6960.

[SMW02] Neil Spring, Ratul Mahajan, and David Wetherall. "Measuring ISP topologies with Rocketfuel." *ACM SIGCOMM Computer Communication Review*, **32**(4):133–145, 2002.

[SPI23] SPIFFE Project. "The SPIFFE Trust Domain and Workload Identity Framework." https://spiffe.io, 2023. Accessed: 2025-01-01.

[SPI25] SPIFFE Working Group. "Secure Production Identity Framework for Everyone (SPIFFE) Specification." https://spiffe.io/, 2025. Accessed: 2025-11-10.

[Spl25] Splunk Inc. "Splunk SOAR." https://www.splunk.com/en_us/software/splunk-security-orchestration-and-automation.html, 2025.

[SPR16] Pawel Szalachowski, Adrian Perrig, and R. Reischuk. "PKI Safety Net (PKISN): Addressing the Too-Big-to-Be-Revoked Problem of the PKI." In *IEEE S&P*, 2016.

[The22] The NDN Team. "Mini-NDN: A Mininet based NDN emulator.", 2022. accessed: 2022-11-01. https://github.com/named-data/mini-ndn.

[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 ACM Conference on Information-Centric Networking*, ACM-ICN '15, p. 177–186, New York, NY, USA, 2015. Association for Computing Machinery. https://doi.org/10.1145/2810156.2810170.

[YMP24] Tianyuan Yu, Xinyu Ma, Varun Patil, Yekta Kocaogullar, and Lixia Zhang. "Exploring the Design of Collaborative Applications via the Lens of NDN Workspace." In *2024 IEEE International Conference on Metaverse Computing, Networking, and Applications (MetaCom)*, pp. 89–96. IEEE, 2024.

[YMX23] Tianyuan Yu, Xinyu Ma, Hongcheng Xie, Yekta Kocaoğullar, and Lixia Zhang. "A New API in Support of NDN Trust Schema." pp. 46–54, 10 2023.

[YXL22] Tianyuan Yu, Hongcheng Xie, Siqi Liu, Xinyu Ma, Xiaohua Jia, and Lixia Zhang. "CertRevoke: A Certificate Revocation Framework for Named Data Networking." In *Proceedings of the 9th ACM Conference on Information-Centric Networking*, 2022.

[ZAB14] Lixia Zhang, Alexander Afanasyev, Jeffrey Burke, Van Jacobson, KC Claffy, Patrick Crowley, Christos Papadopoulos, Lan Wang, and Beichuan Zhang. "Named Data Networking." *ACM SIGCOMM Computer Communication Review*, **44**(3):66–73, 2014.

[Zee18] Zeek Project. "Zeek: The Network Security Monitor." `https://zeek.org/`, 2018. Accessed: 2025-02-16.

[Zha18] Haitao Zhang. *NDNFit: An Open mHealth Application Built on Named Data Networking*. University of California, Los Angeles, 2018.

[ZVK19] Zhiyi Zhang, Vishrant Vasavada, Randy King, and Lixia Zhang. "Proof of authentication for private distributed ledger." In *Proceedings of the NDSS Workshop on Decentralised IoT Systems and Security (DISS)*, 2019.

[ZWS20] Zhiyi Zhang, Su Yong Wong, Junxiao Shi, Davide Pesavento, Alexander Afanasyev, and Lixia Zhang. "On certificate management in named data networking." *arXiv preprint arXiv:2009.09339*, 2020.

[ZYM22] Zhiyi Zhang, Tianyuan Yu, Xinyu Ma, Yu Guan, Philipp Moll, and Lixia Zhang. "Sovereign: Self-Contained Smart Home With Data-Centric Network and Security." *IEEE Internet of Things Journal*, **9**(15):13808–13822, 2022.

[ZYR18] Zhiyi Zhang, Yingdi Yu, Sanjeev Kaushik Ramani, Alex Afanasyev, and Lixia Zhang. "Nac: Automating access control via named data." In *MILCOM 2018-2018 IEEE Military Communications Conference (MILCOM)*, pp. 626–633. IEEE, 2018.

.1 NAAP Protocol Details

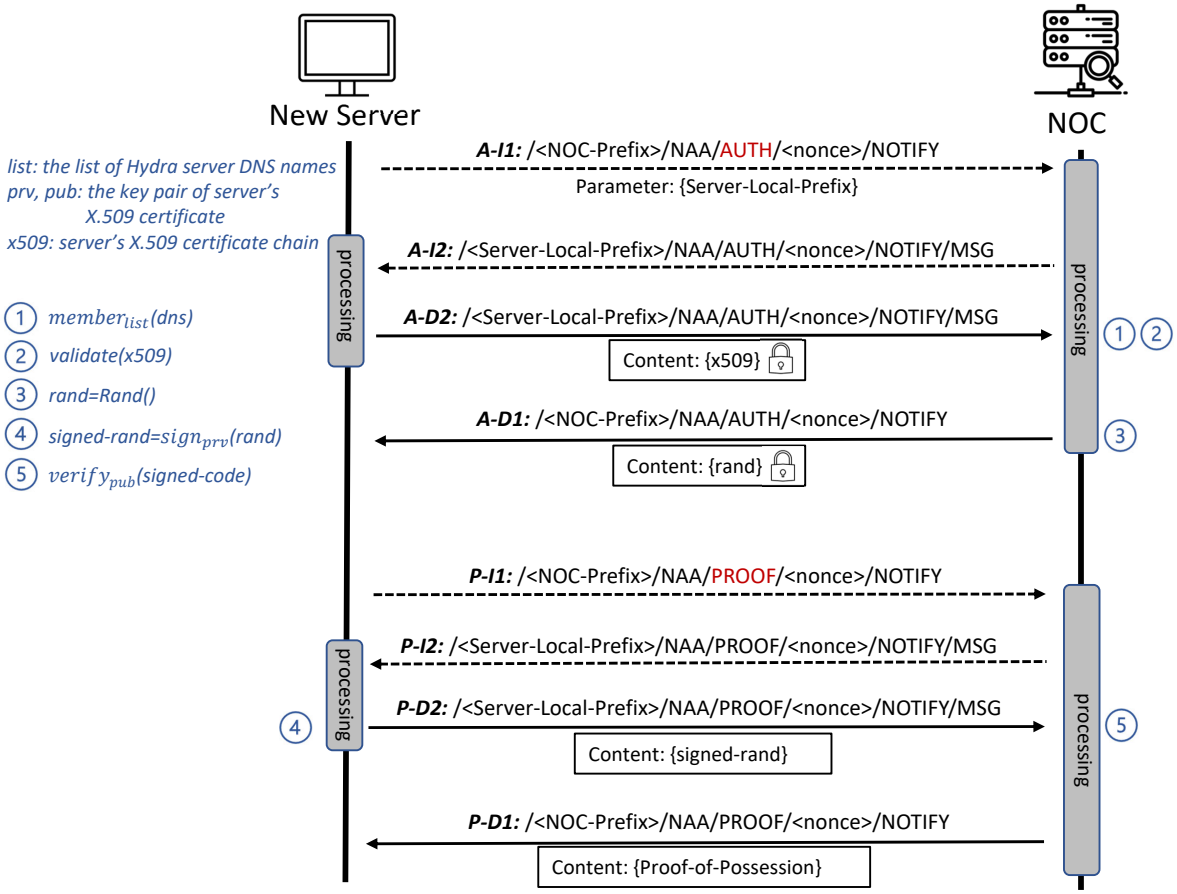


Figure .1: Hydra server authentication workflow

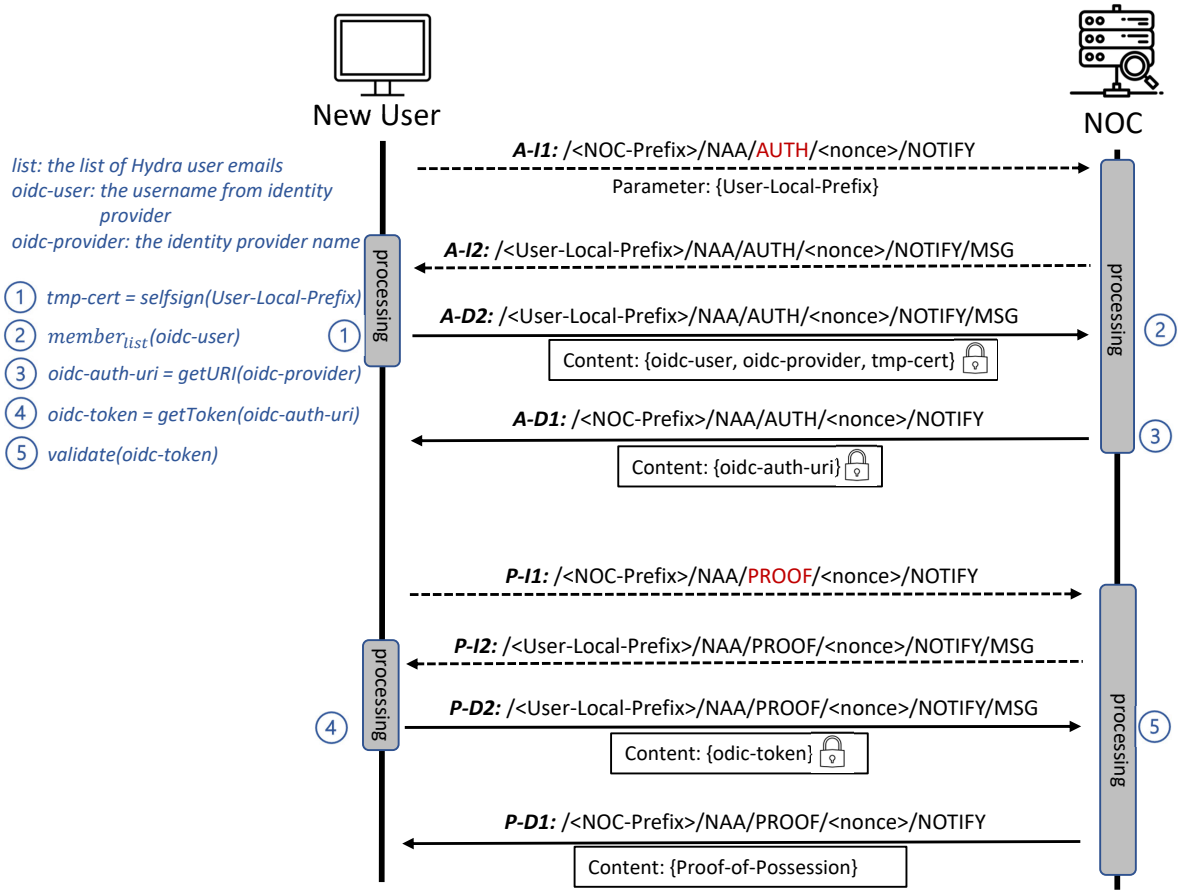


Figure .2: Hydra user authentication workflow