Universal Digital Identification Framework – UDIF

Revision 1a, September 05, 2025

John G. Underhill – john.underhill@protonmail.com

This document is an engineering level description of the universal digital identification framework (UDIF). This document describes the UDIF system, a quantum-secure cryptographic identification and auditable object storage framework.

Contents **Page**

Foreword

This document is intended as the preliminary draft of a new standards proposal, and as a basis from which that standard can be implemented. We intend that this serves as an explanation of this new technology, and as a complete description of the protocol.
This document is the first revision of the specification of UDIF, further revisions may become necessary during the pursuit of a standard model, and revision numbers shall be incremented with changes to the specification. The reader is asked to consider only the most recent revision of this draft, as the authoritative implementation of the UDIF specification.
The author of this specification is John G. Underhill, and can be reached at john.underhill@protonmail.com

1. Introduction

The Universal Digital Identity Framework (UDIF) is a polymorphic, cryptographically enforced identity verification and object container system. It addresses a longstanding gap in the world of digital identity and asset provenance: there is no universally recognized, cryptographically secure way to represent and manage people, organizations, and the objects they own across multiple domains. Existing identity systems such as social security numbers, legal entity identifiers and password-based authentication are siloed, brittle and easily abused. Asset registries for money, securities or provenance often reside on legacy infrastructure with little interoperability and minimal cryptographic binding between the owner and the object. As digital assets become more valuable and cross-border, a stronger foundation for authentication, authorization and auditability is necessary. UDIF fills this gap by providing a post-quantum secure container model for any identity or object, complete with hierarchical trust anchors, deterministic encoding, minimal-disclosure queries and verifiable audit trails.

At its core UDIF defines a tree of trust. The root of the tree is a universally trusted authority (for example, a national regulator or standards body) that issues certificates to intermediate branches. Each branch can be configured to administer either sub-branches or end-users, never both at once. Branches in branch-admin mode create subordinate branches, forming the hierarchy; branches in user-admin mode become group controllers (GCs) and enroll user agents (UAs). Users are the leaf nodes: they are the only entities that actually own objects.
This topology is deliberately agnostic to any particular organization. Governments, financial institutions, supply chains or device manufacturers can superimpose their own structures onto the UDIF tree, but the security hierarchy stands independent of those organizations. Every node possesses a certificate signed by its parent, forming a verifiable chain up to the root. The root certificate includes the signature suite identifier and version string used across the entire domain, ensuring that all participants compile against the same cryptographic algorithms and configurations.

The fundamental unit of the framework is an object container. An object represents any commodifiable entity: a citizen identity, a bank account, a legal entity, a digital token or a physical asset, objects are record containers. Each object has a 32-byte serial number, a creator identifier and a commit to its attributes.
Attributes may include metadata such as a registry reference, policy flags and a Merkle root committing to any additional data. Objects are immutable in spirit: their core identity never changes, but the mutable state (ownership, registry membership, policy flags) is updated through signed events. These events are appended to a transaction log administered by the relevant group controller. To prevent tampering or rollback, every object contains the hash of its previous state and a monotonic counter, so that any attempted modification without authorization immediately breaks the hash chain. When a user transfers an object to another user, the group controller updates the object record, re-signs it and logs the transfer event.
A similar model is applied to registries: each user has a registry per group that contains a Merkle tree of the objects they own. Adding or removing an object updates this tree and increments a state counter, and the group controller co-signs this registry update, providing a non-repudiable proof of ownership at any point in time.

UDIF's capabilities are minimalistic tokens that grant finely scoped rights. A capability encodes which *verbs* a holder may perform such as *create*, *read*, *update*, *delete*, *transfer* or *query*, and the relation (*owner*, *peer*, *group*) under which those verbs apply.
Capabilities are issued by parents as MAC-tagged bitmaps tied to a subject's certificate and a policy epoch; there is no domain-specific language or interpreter in the core, by design. This approach avoids the complexity of generic authorization languages while still allowing rich permission graphs through the intersection of certificates, capabilities, and explicit treaties.
Every operation in UDIF is authorized by taking the intersection of the caller's capability bitmaps with the subject's certificate scope and the local policy. If the intersection is empty, the operation is denied without revealing any information about the target. Query types are kept minimal-existence, owner binding, attribute bucket membership and membership proof, each returning only *yes* or *no*, with optional digest proofs where authorized.
This enforces a privacy-by-design posture that prevents adversaries from probing state unless they already hold explicit rights to do so.

To provide cryptographically sound audit trails, UDIF uses a two-level logging and anchoring model. Every group controller maintains a membership log of identity lifecycle events (*enroll*, *suspend*, *resume*, *revoke*, *registry* commits) and a transaction log of object mutations (*creates*, *updates*, *transfers*).
These logs are append-only and keyed by digests rather than raw identifiers, ensuring that no personal data is stored on-chain.
At regular intervals, the group controller computes Merkle roots over the exported view of both logs and submits a signed *Anchor Record* to its parent. The Anchor Record contains the Merkle roots, counters and a rolling hash of the previous anchor.
Parents append these anchors into their own logs and include them in their next anchor update to the root. This anchoring mechanism recursively commits all lower-level logs up the trust tree, creating a tamper-evident chain of custody. Because the Anchor Record contains only hashes and counters, no confidential data is leaked; auditors can verify inclusion of any event by requesting a Merkle proof from the relevant controller.

A vital feature of UDIF is its transport and session system, which is borrowed from the Quantum Secure Tunnelling Protocol (QSTP) and adapted for identity management. All communications occur over TCP tunnels with authenticated headers and strict sequence numbers. Each packet carries a monotonic sequence counter and a timestamp; if any packet is missing or out of order, the session is terminated, the error evaluated, and a new session is either established or denied.
The header is authenticated using RCS-256 and KMAC-256, providing a post-quantum secure AEAD that covers both data and metadata. Session keys are negotiated once using a Kyber or McEliece key encapsulation mechanism (QSMP); the shared secret then seeds a cSHAKE-based ratchet. On long-lived branch-to-branch tunnels, the ratchet re-seeds every hour (with ±5-minute jitter) by hashing the old key state with a fresh KEM secret and the session identifier. This ratchet ensures forward secrecy and post-compromise security: even if an attacker recovers the session key at some epoch, they cannot decrypt past or future epochs without breaking the KEM. UDIF integrates these cryptographic primitives into a cohesive whole to solve several real-world problems:

Identity interoperability and privacy.
Traditional identity schemes bind a person to a single identifier (SSN, passport, driver's license) that is reused across contexts, creating a massive correlation surface.
UDIF allows the same person to hold multiple identities (different chains and registries) without linkability, while still enabling a trusted controller to resolve their identity under due process. By

returning only Boolean *yes*/*no* answers to queries and using digests in logs, UDIF prevents arbitrary correlation of identities across services.

Cross-domain ownership and transfer.
Assets often move across organizational boundaries; a bank may need to recognize a government-issued digital identity, or a supply chain might involve multiple jurisdictions. UDIF defines peering treaties between domains that specify exactly which queries or operations are permitted. Because capabilities are explicit bitmaps and treaties are explicit enumerations of allowed verbs and scopes, there is no risk of 'treaty creep', where rights bleed across domains. Transfers require explicit consent from both the current owner and the recipient's controller. Anchors ensure that both sides record the same transfer in their logs preventing equivocation and enforcing non-repudiation.

Post-quantum assurance.
Existing identity and asset systems largely rely on classical cryptography (RSA, ECDSA, AES-GCM) that may be vulnerable once quantum computers scale. UDIF mandates Dilithium (ML-DSA) or SPHINCS+ (SLH-DSA) signatures, Kyber (ML-KEM) or classic McEliece (NPQ-R4) key exchange and a sponge-based AEAD (RCS with KMAC). These primitives are selected for their high-security margins and simplicity, and are compiled into the suite string embedded in the certificates. There is no negotiation; all parties must agree on the suite defined at compile time. This design eliminates downgrade and middle-box attacks, providing confidence that the system remains secure into the quantum era.

Auditability without surveillance.
Regulatory frameworks require detailed audit trails for financial transactions and identity operations. UDIF meets these requirements through deterministic logs, Merkle roots and Anchor Records, giving auditors verifiable evidence of actions without exposing sensitive details. There is no need to record complete transaction payloads in the anchor chain; the digest and counters suffice.
Organizations can choose to anchor their logs into an external public ledger if additional cross-domain tamper-evidence is required, but this is optional and clearly delineated as an extension.

In summary, UDIF is designed to be a universal, cryptographically sound substrate for digital identity and asset management. It provides a hierarchical certificate and capability architecture that prevents unauthorized privilege escalation; lean object and registry containers that support ownership tracking and transfer; and deterministic logging and anchoring that enable tamper-evident provenance without compromising privacy.
UDIF shows how a universal framework can leverage proven cryptographic patterns while tailoring them for cross-domain identity and asset custody. The goal is not to replace existing identity systems overnight, but to provide a secure and flexible layer that can bridge them and support future applications; from regulated finance to supply chain provenance and digital public infrastructure. A polymorphic design means UDIF can represent any object or identity, and future profiles can add richer semantics (e.g., zero-knowledge predicates or privacy-max pseudonymization) without breaking the core. The remainder of this specification details how each component is constructed, how messages flow, how policies are enforced and how logs are anchored, but the introduction sets the stage: UDIF is a practical framework designed from first principles to solve the intertwined problems of identity, privacy, auditability, and post-quantum resilience in a world of increasingly interconnected systems.

2. Scope

This document specifies the Universal Digital Identity Framework (UDIF), a polymorphic and cryptographically enforced system for managing digital identities, objects, and their relationships. UDIF defines a universal structure for representing entities, whether individuals, institutions, or commodifiable objects; using secure, hierarchical certificates, attribute containers, and auditable records. The framework provides an architecture for authentication, authorization, and accountability across domains, while remaining agnostic to organizational superstructures and adaptable to diverse applications.

The scope of UDIF encompasses the following areas:

Identity and Object Containers: Formal definitions of how users, institutions, and assets are represented in the framework. This includes canonical attribute sets, object registries, and the mechanisms by which ownership and provenance are established and maintained.

Certificates and Capabilities: A hierarchical certificate model that binds authority through post-quantum digital signatures, augmented with access masks and capability bitmaps that define granular rights within domains.

Audit and Logging Mechanisms: Membership logs, transaction logs, and anchor records that ensure every action and state change is verifiable, tamper-evident, and bound by cryptographic proof.

Transport and Session Model: Methods for secure communication between framework participants, including tunnel authentication, sequence control, ratchet-based key refresh, and compliance with post-quantum cryptographic standards.

Query and Response Semantics: A minimal-disclosure predicate model where queries return the least information required, enforced by capability tokens and access masks.

Cross-Domain Operation: Bilateral treaty mechanisms that allow controlled query forwarding and interoperability between distinct branches or federated systems.

Application

UDIF is designed to serve as a foundational identity and asset container framework, adaptable to multiple contexts:

Personal Identity: Providing secure, verifiable digital identity for individuals, bound to post-quantum certificates and extensible attribute sets.

Institutional Identity: Representing organizations, agencies, and authorities with the same rigor and authenticity guarantees as individuals.

Asset Management: Defining transferable digital objects, including financial instruments, commodities, intellectual property, or electronic currency, with strong provenance and ownership records.

Infrastructure Access: Enabling regulated, cryptographically enforced access to services, secure storage, and critical systems.

Cross-Border Federation: Allowing distinct domains (e.g., national authorities, enterprises, or coalitions) to interoperate without ceding full control, using treaties and controlled predicate exposure.

The framework is deliberately polymorphic: implementations may apply UDIF to diverse sectors such as finance, healthcare, logistics, digital governance, or secure communications, while sharing the same core data model and cryptographic principles.

Protocol Flexibility and Use Cases

UDIF is structured as a minimal core with optional profiles. The core guarantees authenticity, non-repudiation, and immutable accountability; higher-level profiles may extend it with advanced privacy mechanisms, zero-knowledge proofs, or ledger integration. This layered approach ensures that UDIF can evolve without disrupting base compliance.

Key use cases include:

- **Regulated Financial Systems:** Cryptographically enforced identity and asset tracking integrated into existing messaging frameworks.
- **Government Registries:** Secure issuance and verification of citizen, institutional, or commodity identifiers.
- **Cross-Institutional Treaties:** Controlled interoperability between domains, with selective disclosure.
- **Digital Asset Provenance:** Secure lifecycle management of objects, ensuring their authenticity and ownership history cannot be repudiated.

Compliance and Interoperability

UDIF ensures strict compliance with post-quantum cryptographic primitives while maintaining interoperability across domains. Its canonical encoding model eliminates ambiguity, enabling consistent verification regardless of implementation. The certificate and capability model is modular, supporting substitution or upgrade of cryptographic suites at compile time.

Key compliance elements include:

- **Cryptographic Consistency:** All participants within a domain must compile against the same suite, versioned and embedded in certificates.
- **Interoperability Standards:** UDIF's minimal encoding, digest, and audit formats ensure different implementations can communicate without ambiguity.
- **Extensibility:** Optional modules allow new cryptographic primitives, advanced query models, or ledger integrations to be incorporated without breaking the core.

Recommendations for Secure Implementation

- **Regular Updates:** Domains should update cryptographic primitives in line with advances in post-quantum research.
- **Audit Cadence:** Implementations must maintain strict anchoring intervals and audit trails to preserve verifiability.
- **Least Privilege:** Capabilities and access masks should be issued under a default-deny policy, ensuring minimal disclosure.
- **Infrastructure Alignment:** Systems integrating UDIF should optimize network and storage configurations for high-throughput cryptographic operations.

Document Organization

This specification is organized into sections describing terms and definitions, cryptographic primitives, core structures, transport model, query semantics, governance, and security analysis. Appendices provide canonical encoding rules, capability registries, and policy hooks. Together, these define the complete normative behavior of UDIF.

Threat Model and Goals

Threat Model

The Universal Digital Identity Framework (UDIF) is designed for deployment in adversarial environments where both external attackers and insider misuse must be assumed. The framework is expected to operate across domains that may have different levels of trust and security maturity.

The following threats are considered:

- **Passive adversaries:** Capable of observing all communications, attempting correlation attacks, and exploiting metadata leakage to infer relationships between entities, transactions, or domains.
- **Active adversaries:** Able to intercept, modify, replay, reorder, or inject messages between participants. This includes attempts at impersonation, downgrade attacks, and denial-of-service.
- **Compromised nodes:** A user agent (UA), group controller (GC), or even a branch controller (BC) may be malicious or compromised. UDIF must contain mechanisms for suspension, revocation, and audit to detect and limit damage.
- **Cross-domain abuse:** Malicious domains may attempt to misuse peering treaties to escalate their visibility into foreign registries or bypass access restrictions.
- **Cryptanalytic attacks:** Both classical and quantum adversaries are in scope. All core cryptographic primitives must be resistant to large-scale quantum adversaries.
- **Side-channel leakage:** Timing, error codes, and packet sizes may be exploited. UDIF minimizes this surface by fixed-size responses, bounded timing jitter, and constant-time cryptographic operations.
- **Replay and state manipulation:** Replaying old transaction records, anchors, or certificates to create confusion about current status. Strict sequence, timing windows, and anchored logs mitigate these attempts.

- **Organizational misuse:** Even trusted administrators may overreach by accessing data beyond their scope. UDIF enforces a default-deny model and limits administrative actions to direct parent-child relationships.

Threats **out of scope** for the core specification include physical attacks on hardware, coercion of users, or failures of external policy. Such threats must be addressed at the profile or organizational level.

Security Goals

In addressing these threats, UDIF establishes the following explicit goals:

1. **Authenticity and Non-Repudiation**
   All actions, whether administrative or transactional, must be provably bound to the originator via post-quantum signatures. Logs and Anchor Records provide immutable, auditable records.
2. **Confidentiality and Integrity**
   Communications between nodes must be encrypted with AEAD ciphers (RCS-256 + KMAC-256) to prevent eavesdropping or tampering. Key establishment relies on post-quantum KEMs (Kyber or McEliece).
3. **Minimal Disclosure**
   Responses to queries must reveal only the minimal necessary information (yes/no, bucket, or proof), never raw identifiers or values, unless explicitly permitted by capabilities.
4. **Granular Access Control**
   Capabilities and access masks define exactly which verbs, predicates, and scopes are allowed. Default state is *deny-all*; capabilities must be explicitly delegated.
5. **Auditability and Tamper Evidence**
   All significant events (registrations, transfers, suspensions, revocations) are logged and periodically committed into Anchor Records. Parents verify child anchors, creating a verifiable chain of custody.
6. **Post-Quantum Resistance**
   All cryptographic primitives (KEM, signatures, AEAD, hashes) are quantum-safe. UDIF aims to remain secure in the face of large-scale quantum adversaries.
7. **Inter-Domain Containment**
   Peering treaties strictly bound cross-domain queries to agreed capabilities. Domains may not assume trust beyond explicit treaty scopes.
8. **Resilience and Recovery**
   Any mis-sequenced or replayed message results in a fatal session reset, ensuring clean recovery. Revocation and suspension mechanisms ensure compromised nodes are contained.
9. **Simplicity of Core**
   By minimizing moving parts, canonical encoding, bitmap capabilities, anchored logs; UDIF reduces attack surface and facilitates correct implementation.

3. Terms and Definitions

3.1 Protocol Components

Root
The universally trusted authority that issues the initial certificates for branches.
The Root is responsible for defining the cryptographic suite of a domain and anchoring the top of the certificate hierarchy. It may be a global entity, a state-level authority, or a designated institutional root.

Branch Controller (BC)
A node responsible for administering a subtree in the UDIF hierarchy.

A Branch may operate in one of two permanent modes:

- **Branch-admin mode:** managing subordinate branches.
- **Group-admin mode (Group Controller):** managing User Agents.

Branches do not own objects directly; they provide administration, governance, and verification services for their subtree.

Group Controller (GC)
A special form of Branch Controller that directly manages a group of User Agents.
The GC is responsible for user registration, certificate issuance, capability assignment, and enforcing audit and revocation policies at the group level.

User Agent (UA)
An end-entity within the framework.
A UA may represent an individual, a device, or a service. UAs hold certificates, own objects, and operate object registries. UAs interact only with their immediate GC and cannot directly query or transact with other UAs outside their group.

Object
A canonical digital representation of a commodity, asset, or value within the UDIF hierarchy.
Objects consist of fixed identifiers (description, serial numbers, type codes, creator signature) and a Merkle-committed attribute root. Objects are always bound to a UA as owner and may be transferred only through signed and logged transactions.

Object Registry
A per-UA container that maintains the membership of all objects belonging to that UA.
The registry is implemented as a Merkle tree of object digests. The registry root is committed periodically and logged via the GC, providing a verifiable audit trail of ownership.

Capability Token
A cryptographically bound token representing the rights of a UA, GC, or BC within UDIF.
Capabilities are implemented as lean bitmaps, augmented with KMAC tags signed by the issuer. They define verbs (e.g., query, prove, forward) and scopes (e.g., intra-domain, treaty-peer). Default state is ‘deny’; explicit capability must be granted.

Access Mask

A set of permissions embedded in a certificate that determine which attributes or operations are available to the holder. Access masks enforce default-deny semantics and prevent disclosure beyond explicit authorization.

Membership Log
A tamper-evident record maintained by each GC or BC that tracks user enrollment, revocation, suspension, registry updates, and administrative actions.
Membership logs are periodically committed upstream via Anchor Records.

Transaction Log
A tamper-evident record of object transfers and transactions between UAs or across domains.
Like membership logs, transaction logs are committed upstream, creating an immutable chain of custody.

Anchor Record
A signed digest package transmitted from a child to its parent, containing the current registry root, transaction root, and counters.
Anchor Records provide the parent with a verifiable summary of the child's state and create a continuous audit chain up to the Root.

Treaty
A bilateral agreement between domains that establishes the scope of permissible cross-domain queries. Treaties are defined by capability masks and do not grant administrative control. They are strictly limited to query and proof forwarding.

Suite
The compile-time cryptographic configuration for a domain.
The suite string (e.g., UDIF:RCS256-KMAC256-MLKEM5-MLDSA5) is embedded in certificates and must match across all participants within a domain. No runtime negotiation is permitted.

Revocation
The permanent invalidation of a certificate or capability, initiated by the parent authority.
Revocation cascades downstream, rendering all subordinate certificates invalid.

Suspension
A temporary disabling of a UA, GC, or BC, usually triggered by audit anomalies or suspected compromise. Suspension prevents upstream queries and forwarding until resolution.

3.2 Cryptographic Primitives

Kyber

The Kyber asymmetric cipher and NIST Post Quantum Competition winner ML-KEM.

McEliece

The McEliece asymmetric cipher and NIST Round 4 Post Quantum Competition candidate.

Dilithium

The Dilithium asymmetric signature scheme and NIST Post Quantum Competition winner ML-DSA.

SPHINCS+

The SPHINCS+ asymmetric signature scheme and NIST Post Quantum Competition winner SLH-DSA.

RCS

The wide-block Rijndael hybrid AEAD symmetric stream cipher.

SHA-3

The SHA3 hash function NIST standard, as defined in the NIST standards document FIPS-202; SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions.

SHAKE

The NIST standard Extended Output Function (XOF) defined in the SHA-3 standard publication FIPS-202; SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions.

KMAC

The SHA3 derived Message Authentication Code generator (MAC) function defined in NIST special publication SP800-185: SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash and ParallelHash.

3.3 Network References

Bandwidth

The maximum rate of data transfer across a given path, measured in bits per second (bps).

Byte

Eight bits of data, represented as an unsigned integer ranged 0-255.

Certificate

A digital certificate, a structure that contains a signature verification key, expiration time, and serial number and other identifying information. A certificate is used to verify the authenticity of a message signed with an asymmetric signature scheme.

Domain

A virtual grouping of devices under the same authoritative control that shares resources between members. Domains are not constrained to an IP subnet or physical location but are a virtual group of devices, with server resources typically under the control of a network administrator, and clients accessing those resources from different networks or locations.

Duplex

The ability of a communication system to transmit and receive data; half-duplex allows one direction at a time, while full-duplex allows simultaneous two-way communication.

Gateway: A network point that acts as an entrance to another network, often connecting a local network to the internet.

IP Address

A unique numerical label assigned to each device connected to a network that uses the Internet Protocol for communication.

IPv4 (Internet Protocol version 4): The fourth version of the Internet Protocol, using 32-bit addresses to identify devices on a network.

IPv6 (Internet Protocol version 6): The most recent version of the Internet Protocol, using 128-bit addresses to overcome IPv4 address exhaustion.

LAN (Local Area Network)
A network that connects computers within a limited area such as a residence, school, or office building.

Latency
The time it takes for a data packet to move from source to destination, affecting the speed and performance of a network.

Network Topology
The arrangement of different elements (links, nodes) of a computer network, including physical and logical aspects.

Packet
A unit of data transmitted over a network, containing both control information and user data.

TCP/IP (Transmission Control Protocol/Internet Protocol)
A suite of communication protocols used to interconnect network devices.

Throughput: The actual rate at which data is successfully transferred over a communication channel.

VLAN (Virtual Local Area Network)
A logical grouping of network devices that appear to be on the same LAN regardless of their physical location.

VPN (Virtual Private Network)
Creates a secure network connection over a public network such as the internet.

3.4 Normative References

The following documents serve as references for cryptographic components used by DKTP:

FIPS 202: SHA-3 Standard: Permutation-Based Hash and Extendable Output Functions: This standard specifies the SHA-3 family of hash functions, including SHAKE extendable-output functions. https://doi.org/10.6028/NIST.FIPS.202

FIPS 203: Module-Lattice-Based Key Encapsulation Mechanism (ML-KEM): This standard specifies ML-KEM, a key encapsulation mechanism designed to be secure against quantum computer attacks. https://doi.org/10.6028/NIST.FIPS.203

FIPS 204: Module-Lattice-Based Digital Signature Standard (ML-DSA): This standard specifies ML-DSA, a set of algorithms for generating and verifying digital signatures, believed to be secure even against adversaries with quantum computing capabilities.
https://doi.org/10.6028/NIST.FIPS.204

NIST SP 800-185: SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash, and ParallelHash: This publication specifies four SHA-3-derived functions: cSHAKE, KMAC, TupleHash, and ParallelHash. https://doi.org/10.6028/NIST.SP.800-185

NIST SP 800-90A Rev. 1: Recommendation for Random Number Generation Using Deterministic Random Bit Generators: This publication provides recommendations for the generation of random numbers using deterministic random bit generators.
https://doi.org/10.6028/NIST.SP.800-90Ar1

NIST SP 800-108: Recommendation for Key Derivation Using Pseudorandom Functions: This publication offers recommendations for key derivation using pseudorandom functions.
https://doi.org/10.6028/NIST.SP.800-108

FIPS 197: The Advanced Encryption Standard (AES): This standard specifies the Advanced Encryption Standard (AES), a symmetric block cipher used widely across the globe.
https://doi.org/10.6028/NIST.FIPS.197

4. Protocol Architecture

4.1 Server, User, and Object Types

The Universal Digital Identity Framework (UDIF) is structured as a hierarchical tree of cryptographically verified entities. At its core, UDIF separates *administrative authority* from *object ownership*, ensuring that branches manage groups and users but never hold assets directly. This separation enforces clear boundaries of accountability and simplifies audit and revocation.

4.1.1 Root Authority

At the top of the hierarchy is the **Root**, a universally trusted authority.

- The Root issues initial certificates to branches.
- It defines the cryptographic suite for the domain and anchors the certificate chain.
- The Root may be global, national, or institutional, depending on deployment context.

4.1.2 Branch Controllers (BCs)

Branches extend the Root's trust downward. Each Branch operates in one of two fixed roles:

1. **Branch-admin mode:** managing subordinate branches.
2. **Group-admin mode (Group Controller):** directly managing User Agents.

- Branches never own objects; they administrate, enforce policy, and validate logs.
- Certificates issued by branches embed capability masks, constraining subordinate authority.
- Branches periodically anchor their logs upstream to maintain accountability.

4.1.3 Group Controllers (GCs)

A Group Controller is a Branch in user-admin mode.

- GCs register User Agents, issue their certificates, and assign capabilities.
- They maintain *Membership Logs* (UA lifecycle events) and *Transaction Logs* (object transfers).
- GCs validate and proxy queries from UAs, forwarding them upstream or across treaties only when capability permits.
- GCs are the primary enforcement point for least-privilege access and audit integrity.

4.1.4 User Agents (UAs)

User Agents are end-nodes representing individuals, devices, or services.

- Each UA holds a certificate signed by its GC and owns a Registry of objects.
- UAs do not interact laterally; all communications flow through their GC.
- When objects are transferred, UAs sign the transaction, and the GC records it in the Transaction Log.

- UAs cannot exceed the scope of their GC's permissions; their certificates inherit upstream constraints.

4.1.5 Objects and Registries

Objects are cryptographically bound records of commodities, assets, or values.

- Each object has fixed identifiers, creator signature, and a Merkle-committed attribute root.
- Objects always belong to a UA; branches never hold them.
- Each UA maintains a Registry, a Merkle tree of its object digests.
- The registry root is updated on any ownership change and logged upstream, enabling tamper-evident audits.

4.1.6 Logs and Anchor Records

Every GC or BC maintains two logs:

- **Membership Log:** user registrations, suspensions, revocations, and registry updates.
- **Transaction Log:** object transfers and treaty-scoped interactions.

At fixed intervals, children produce *Anchor Records*: signed packages containing registry roots, transaction roots, and counters. Parents verify and incorporate these anchors, creating a continuous chain of custody up to the Root.

4.1.7 Cross-Domain Treaties

Domains interoperate through *Treaties* bilateral agreements that define which queries may be forwarded.

- Treaties never grant administrative authority; they only allow limited predicate proofs.
- Queries across domains are mediated by GCs or BCs and logged as transactions.
- Access remains constrained by capability masks and default-deny enforcement.

4.1.8 Transport Layer

UDIF assumes TCP transport.

- Tunnels are authenticated using post-quantum certificates, KEM exchanges, and RCS-256 AEAD sessions.
- Headers include sequence, epoch, suite ID, and timestamps for replay protection.
- Sessions re-key hourly on branch trunks.

Summary

The architecture ensures that trust flows downward, audit flows upward, and objects remain bound to end-users. Branches enforce policy and accountability, Roots anchor global trust, and treaties

allow controlled federation. This design creates a cryptographically verifiable identity and object framework that is adaptable, tamper-evident, and resilient against compromise.

4.2 Cryptographic Suites and Domain Separation

4.2.1 Compile-Time Suite Model

UDIF adopts a compile-time cryptographic suite model. Each domain selects a suite at build time, and this suite is embedded in every certificate issued within the domain. Runtime negotiation is not supported; connections with mismatched suites are rejected.

This eliminates downgrade attacks, ensures consistent security assumptions across a domain, and simplifies implementation and audit. Domains may evolve by publishing a new suite string and re-issuing certificates under that profile.

A typical suite identifier is a fixed ASCII string such as:

UDIF:RCS256-KMAC256-MLKEM5-MLDSA5

This string identifies the domain, the authenticated encryption scheme, the KEM, and the signature algorithm.

Core Algorithms

AEAD Cipher: RCS-256 combined with KMAC-256 for authentication.
Provides confidentiality and integrity of tunnel messages with 256-bit strength.

Key Encapsulation Mechanism (KEM):
Domains may compile with either:
Kyber (ML-KEM), lattice-based KEM standardized by NIST.
Classic McEliece, code-based KEM offering long-term resistance.

Digital Signatures:
Domains may compile with either:
ML-DSA (Dilithium) — efficient lattice-based signatures.
SPHINCS+ (SLH-DSA) stateless hash-based signatures.

Hash and KDF Functions:
SHA3-256 for digests (object IDs, registry roots, anchor hashes).
cSHAKE-256 for KDF and ratchet expansion.

4.2.2 Domain Separation Labels

All hash and KDF invocations in UDIF are domain-separated using fixed ASCII labels. Each label identifies both the function and its context, ensuring that identical byte sequences used in different contexts cannot be misinterpreted or substituted.

Examples include:

- UDIF:OBJ-DIGEST:V*n* - object digest commitments.
- UDIF:REGROOT:V*n* - registry Merkle root digests.
- UDIF:TXID:V*n* - transaction identifiers.
- UDIF:ANCHOR:V*n* - anchor record digests.
- UDIF:CAP-DIGEST:V*n* - capability digests.
- UDIF:SESS-KDF:V*n* - session key derivation.
- UDIF:RATCHET:V*n* - asymmetric re-key ratchet derivation.

These labels are constants and never negotiated. Version tags (V1, V2, ...) allow safe protocol evolution.

4.2.3 Ratchet and KDF Binding

Re-keying and session ratchets use the following binding:

1. Input material: IKM = state_prev || KEM_shared || session_id || transcript_hash
2. Derivation: PRK = cSHAKE256(IKM, N = "UDIF:RATCHET:V1", S = "")
3. Key/Nonce Slices (example for RCS-256): TX key = bytes 0-31, TX nonce = bytes 32-63, RX key = bytes 64-95, RX nonce = bytes 96-127
4. State chaining: state_new = SHA3-256(TX_key || TX_nonce || RX_key || RX_nonce)

This ensures forward secrecy, strong separation of transmit and receive directions, and fresh state for each ratchet interval.

Cryptographic Agility

Although UDIF avoids runtime negotiation, it maintains agility at compile-time. By fixing cryptographic choices per suite string, domains can evolve over time without breaking existing deployments. Implementers are encouraged to:

- Re-evaluate suite selections in light of PQC research.
- Provide migration tooling for re-issuing certificates under new suites.
- Retire older suites gracefully, ensuring all anchors and logs remain verifiable through retained digests.

Security Rationale

- Compile-time lock: Prevents downgrade and simplifies verification.
- Post-quantum only: Guarantees resistance to both classical and quantum adversaries.
- Domain separation labels: Block cross-context substitution attacks and aid auditability.

- Full-block ratchet inputs: Maximize entropy use and prevent correlations between key and nonce outputs.
- Suite string embedding: Ensures every certificate and anchor is explicitly bound to its security level.

4.3 Canonical Data Model (TLV/uvarint C14N)

Overview

All structures in UDIF; certificates, objects, registries, logs, anchors, and capabilities, are serialized into a single canonical binary format. This ensures that every participant encodes and interprets data in exactly the same way, eliminating ambiguity and preventing malleability attacks. The canonical model is based on Tag–Length–Value (TLV) containers, with uvarint-encoded lengths, and enforces strict ordering and uniqueness of fields.

TLV Structure

Each field is encoded as:

T (Tag): a variable-length unsigned integer (uvarint).

L (Length): a variable-length unsigned integer (uvarint), encoding the number of bytes in V.

V (Value): raw bytes or a nested TLV sequence.

The resulting encoding is deterministic, compact, and simple to parse in constant time.

Encoding Rules

To maintain canonical form, the following rules apply:

1. **Tag Order:** Fields must appear in ascending tag order. Any out-of-order tag results in an invalid encoding.
2. **Unique Tags:** Each tag may appear only once in a container, unless explicitly declared as a repeated field.
3. **Repeated Fields:** If repetition is allowed, repeated instances must be encoded contiguously and in ascending tag order.
4. **Integers:** Unsigned integers are encoded as minimal-length uvarints.
5. **Fixed-Length Fields:** (e.g., digests, keys) are encoded as raw binary under their tag.
6. **Strings:** UTF-8 encoded, normalized to NFKC before hashing or signing.
7. **Booleans:** Encoded as a single byte 0x00 (false) or 0x01 (true).
8. **Time Values:** 64-bit unsigned integers representing **seconds** since Unix epoch (UTC).
9. **Maps:** Encoded as repeated TLVs, each containing a key (tag=1) and value (tag=2) sub-TLV. Keys must be sorted lexicographically by raw bytes.
10. **Omitted Defaults:** Fields with default values are omitted entirely.

Canonicalization Benefits

- **Determinism:** A given logical record always maps to one exact byte sequence. This ensures digests and signatures are stable and reproducible.
- **Compactness:** Uvarint encoding minimizes storage and bandwidth usage, especially for small values.
- **Simplicity:** A linear parser can scan TLV sequences without schema lookups, making the format implementable in constrained environments.
- **Security:** Strict tag ordering and shortest-form integers prevent duplicate encodings and ambiguity, reducing attack surface for malleability or replay.
- **Auditability:** Canonical form means that anchored digests remain verifiable across all compliant implementations, regardless of platform.

Use in UDIF Structures

- **Certificates:** Contain TLV fields for suite ID, role, serial, issuer serial, validity window, public key, capability bitmap, and signature.
- **Objects:** Encoded with serial, type, creator cert digest, attribute Merkle root, current owner, timestamps, and signature.
- **Registries:** Encoded as Merkle roots of sorted object digests.
- **Logs:** Membership and transaction events are TLV-encoded with reason codes, identifiers, and timestamps.
- **Anchor Records:** Contain TLV fields for child serial, sequence, registry root, transaction root, counters, and signature.
- **Capabilities:** Encoded as TLV bitmaps, tagged, and signed with KMAC.

In all cases, canonical encoding precedes hashing and signing. Signatures are never computed over non-canonicalized structures.

Security Rationale

The canonical TLV/uvarint model was chosen over alternatives such as ASN.1 DER, CBOR, or JSON because:

- ASN.1 and CBOR are flexible but complex, creating parser divergence risks.
- JSON is human-readable but non-deterministic due to whitespace, ordering, and encoding variants.
- TLV/uvarint is small, auditable, and easy to implement in constant time.

By mandating one universal representation, UDIF avoids redundancy, keeps the attack surface minimal, and ensures interoperability across domains.

4.4 Identities, Certificates, and Capabilities

4.4.1 Identities

UDIF represents every participant and object through cryptographically bound identities. These identities are not arbitrary labels; they are structured, canonicalized, and signed containers that provide authenticity, non-repudiation, and traceability.

4.4.2 Identity Types

- **Root Identity** – The ultimate authority in a domain. Issues certificates to branches and defines the cryptographic suite.
- **Branch Controller Identity (BC)** – An administrative node that either manages subordinate branches (branch-admin mode) or user groups (group-admin mode).
- **Group Controller Identity (GC)** – A branch in group-admin mode that manages User Agents directly.
- **User Agent Identity (UA)** – The end-entity, representing a person, device, or service. UAs own objects and registries, but rely on their GC for all interactions.
- **Object Identity** – A canonicalized record representing a commodity, asset, or value. Objects are always bound to a UA; branches never own objects.

Each identity is expressed through a certificate signed by its parent authority, ensuring that every entity can be verified along a chain up to the Root.

4.5 Certificates

4.5.1 Certificate Structure

Certificates in UDIF are lean, canonical TLV records with the following mandatory fields:

Name	Type	Size (bits)	Description
suite_id	Enum	8	Compile-time suite string
role	Enum	8	ROOT, BRANCH, GC, UA, OBJECT
serial	Uint8 Array	128	Unique serial per certificate
issuer_serial	Uint8 Array	128	Absent for Root
valid_from	Uint64	64	Seconds since epoch
valid_to	Uint64	64	Expiry time from epoch
pubkey	Uint8 Array	Variable	Signature public key
policy_epoch	Uint32	32	Policy set version
cap_bitmap	Uint8 Array	Variable	Embedded capability bits
signature	Uint8 Array	Variable	Parent's signature over TLV(1–9)

Table 4.5.1 Certificate Structure

Signature Algorithms: Either ML-DSA (Dilithium) or SLH-DSA (SPHINCS+), selected at compile time.

- **Inheritance:** Capabilities and permissions may only be equal or more restrictive downstream. A child may never inherit broader permissions than its parent.
- **Revocation:** A parent may revoke or suspend any child certificate. Revocation cascades down the tree.

4.5.2 Roles

- **Root Certificates** anchor trust in a domain.
- **Branch Certificates** propagate authority and administrate subtrees.
- **Group Certificates** issue UA certificates and enforce policies.
- **UA Certificates** prove user or device authenticity.
- **Object Certificates** prove an object's creation, provenance, and binding to its creator UA.

4.6 Capabilities

Capabilities define what an entity can do. They are implemented as *lean bitmaps,* augmented by cryptographic tags, and are bound into certificates or issued as separate tokens.

4.6.1 Capabilities Structure

Name	Type	Size (bits)	Description
verbs_bitmap	Uint32	32	Allowed actions
scope_bitmap			LOCAL, INTRA_DOMAIN, TREATY
issued_to	Uint8 Array	128	Certificate serial of holder
issued_by	Uint8 Array	128	Certificate serial of issuer
valid_to	Uint64		Expiry time from epoch
digest	Uint8 Array	256	SHA3-256("UDIF:CAP-DIGEST:V1" || TLV(1–5))
tag	Uint8 Array	256	KMAC-256(issuer_key, digest)

Table 4.6.1 Capabilities Structure

- **Verbs:** QUERY_EXIST, QUERY_OWNER_BINDING, QUERY_ATTR_BUCKET, PROVE_MEMBERSHIP, FORWARD, ANCHOR_VIEW.
- **Scopes:** LOCAL_ONLY, INTRA_DOMAIN, TREATY_PEER.
- **Digest:** Domain-separated SHA3-256 ensures unique binding.
- **Tag:** KMAC-256 authenticates issuance by the parent.

4.6.2 Properties

- **Default Deny:** Without a valid capability, all operations are rejected.
- **Granularity:** Capabilities can restrict specific query types, domains, or time intervals.
- **Delegation:** Only direct parents can issue capabilities to children.
- **Auditability:** Capability issuance and revocation are logged and anchored.

4.7 Access Masks

4.7.1 Mask Description

Access masks are embedded directly in certificates and describe what subsets of attributes or operations are available to the certificate holder. They serve as static constraints, complementing capabilities which are more dynamic and revocable.

- **Default:** All access set to none.
- **Additive Only:** Permissions can only be explicitly granted, never implied.
- **Scope:** Access masks apply to object attributes, registry views, or transaction queries.

4.7.2 Security Rationale

- **Certificate Chains:** Bind every entity back to the Root; tampering is impossible without breaking PQ signatures.
- **Capabilities:** Provide fine-grained control without complex interpreter logic; bitmaps keep parsing constant-time and simple.
- **Access Masks:** Enforce least privilege by default, ensuring no entity can act outside its parent's granted rights.
- **KMAC Tags:** Lightweight, constant-time, and quantum-safe; prevent forged or replayed capabilities.

Together, certificates, capabilities, and access masks create a layered trust and permission model that is simple, auditable, and resistant to misuse.

4.8 Objects and Registries

4.8.1 Objects

Objects in UDIF are canonical digital representations of commodities, assets, or values. They are always bound to a User Agent (UA) as owner, and are never directly held by Branches or Group Controllers. This strict separation ensures that administration and ownership remain distinct.

4.8.2 Object Structure

Objects are encoded as canonical TLV containers with the following fields:

Name	Type	Size (bits)	Description
object_serial	Uint8 Array	256	Unique serial assigned by creator
type_code	Uint32	32	Registry-specific object type
creator_cert	Uint8 Array	128	Serial of creator's UA certificate
attr_root	Uint8 Array	256	Merkle root of object attributes
current_owner	Uint8 Array	128	UA certificate serial of current owner
created_at	Uin64	64	Creation time, UTC seconds
updated_at	Uin64	64	Last modification time
signature	Uint8 Array	Variable	Signature of current owner

Table 4.8.2 Object Structure

4.8.3 Attributes

- Attributes are organized into a Merkle tree, producing the *attr_root*.
- Attributes may include optional metadata (e.g., category, status, provenance proofs) but are not directly exposed in UDIF-Core.
- Queries about attributes are predicate-based and always bound to capability masks.

4.8.4 Object Lifecycle

- **Creation:** A UA creates an object by assigning a serial, defining a type, and committing an attribute root. The GC validates and logs the creation.
- **Transfer:** Ownership transfer requires a bi-modal signature (sender + receiver) and logging by the GC.
- **Update:** Changing attributes requires recomputing the attribute root, re-signing by the owner, and updating logs.
- **Destruction:** Objects may be marked as destroyed in the registry but never removed from logs, preserving audit history.

4.8.5 Object Registries

Every UA maintains a *Registry*, a canonical container of all objects it owns. The registry provides a verifiable, tamper-evident snapshot of UA ownership at any given time.

4.8.6 Registry Structure

The registry is implemented as a Merkle tree of object digests:

Name	Type	Size (bits)	Description
object_digest	Uint8 Array	256	SHA3-256("UDIF:OBJ-DIGEST:V1" || ObjectRecord)
owner_digest	Uint8 Array	256	Digest of UA cert
flags	Uint16	16	Status (active, suspended, destroyed)
timestamp	Uint64	64	Last update

Table 4.8.6 Registry Leaf Structure

- **Sorting:** Leaves are sorted lexicographically by object_digest.
- **Merkle Root:** The registry root is SHA3-256("UDIF:REGROOT:V1" || serialize(sorted_leaves)).
- **Commitment:** The root is periodically logged and included in Anchor Records, binding UA state to the GC and ultimately to the Root.

4.8.7 Registry Operations

- **Insert:** Add a new object leaf when a UA creates or acquires an object.
- **Update:** Replace the leaf with updated flags or owner digest on transfer or status change.
- **Revoke:** Set destruction or suspension flag; never delete the leaf.
- **Proofs:** Merkle proofs allow efficient verification of membership or non-membership for audit and treaty queries.

4.8.8 Registry Root Commitments

- The GC commits each UA's registry root into its Membership Log.
- During anchoring, the GC aggregates registry roots into the Anchor Record for transmission upstream.
- This ensures every object's existence and ownership is cryptographically committed and tamper-evident.

4.8.9 Security Properties

- **Tamper Evidence:** Any unauthorized change to object attributes or ownership alters the registry root, detectable during anchoring.
- **Immutability:** Transaction and membership logs, combined with registry commitments, create an irreversible record of ownership.
- **Minimal Disclosure:** Proofs reveal only the digest path necessary, never raw attributes or unrelated objects.
- **Auditability:** Registry roots chained through Anchor Records allow any auditor to verify object ownership history without needing the full dataset.

4.8.9 Rationale

Objects and registries form the foundation of UDIF's polymorphic design. By abstracting assets into canonical digests and Merkle commitments, UDIF enables a single framework to manage identities, commodities, financial instruments, or digital property. This abstraction provides flexibility while enforcing strong cryptographic accountability, ensuring that provenance and ownership are secure, auditable, and non-repudiable.

4.9 Audit Logs and Anchor Records

4.9.1 Overview

Auditability is central to UDIF. Every significant event; *enrollment*, *revocation*, *object transfer*, *registry update*, is logged, signed, and periodically committed upstream. This creates a tamper-evident, verifiable chain of custody that extends from User Agents (UAs) through Group Controllers (GCs) and Branch Controllers (BCs), up to the Root. Logs ensure that no action can occur without cryptographic evidence, and Anchor Records guarantee that this evidence is periodically consolidated and attested.

4.9.2 Membership Logs

Each GC or BC maintains a Membership Log. This log records administrative events that affect entities under its control.

4.9.3 Events Recorded

- **UA lifecycle:** enrollment, suspension, resumption, revocation.
- **Certificate changes:** issuance of new certificates, capability grants, revocations.

- **Registry commits:** updates to UA registry roots.
- **Branch actions:** creation, suspension, or retirement of subordinate branches (for BCs).

4.9.4 Membership Structure

Name	Type	Size (bits)	Description
event_code	Uint16	16	Enumerated event code type
subject_serial	Uint8 Array	128	UA or BC serial affected
time	Uint64	64	Event timestamp, UTC
data	Uint8 Array	Variable	optional contextual data
signature	Uint8 Array	Variable	Issuer signature

Table 4.9.4 Membership Structure

Event Codes: ENROLL, SUSPEND, RESUME, REVOKE, CAP_GRANT, CAP_REVOKE, REGISTRY_COMMIT, BRANCH_CREATE, BRANCH_SUSPEND, BRANCH_REVOKE.

Events are strictly append-only; once recorded, they cannot be deleted or modified.

4.9.5 Transaction Logs

Transaction Logs track object-level events. They are maintained by GCs and record all transfers or modifications of objects under their domain.

4.9.6 Events Recorded

- **Object creation** (by UA).
- **Object transfer** (bi-modal signatures by sender and receiver).
- **Object attribute update** (e.g., new Merkle attribute root).
- **Object suspension or destruction.**

4.9.7 Transaction Event Structure

Name	Type	Size (bits)	Description
tx_id	Uint8 Array	256	SHA3-256("UDIF:TXID:V1" || content)
object_serial	Uint8 Array	256	Object affected
parties (TLV[])	Uint8 Array	Variable	Involved UA serials
time	Uint64	64	Event timestamp
digest	Uint8 Array	256	Object digest at time of tx
signature	Uint8 Array	256	Owner (and counterparty) signatures

Table 4.9.7 Transaction Event Structure

- Transactions must be signed by the current owner (and receiver if applicable).
- Every Transaction Event is validated by the GC before logging.
- Transactions are append-only and never retroactively altered.

4.9.8 Anchor Records

Anchor Records serve as periodic, upstream commitments of a node's state. They compress local logs into verifiable digests and transmit them to the parent authority.

4.9.9 Anchor Record Structure

Name	Type	Size (bits)	Description
child_serial	Uint8 Array	128	Serial of GC or BC
seq	Uint64	64	Monotonically increasing anchor sequence
time	Uint64	64	UTC timestamp
regroot	Uint8 Array	256	Merkle root of all UA registries
txroot	Uint8 Array	256	Merkle root of Transaction Log segment
mroot	Uint8 Array	256	Merkle root of Membership Log segment
counters (TLV{})	Uint8 Array	Variable	Optional event counters
signature	Uint8 Array	Variable	Signature of child over fields 1–7

Table 4.9.9 Anchor Record Structure

- **Child Serial**: The serial number of the BC or GC.
- **Sequence:** incremented on each anchor commit.
- **Time:** The anchor commitment time.
- **Regroot:** commitment to UA registries ensures object ownership proofs.
- **Txroot:** commitment to transactions ensures provenance.
- **Mroot:** commitment to membership events ensures lifecycle accountability.
- **Counters:** optional metrics (e.g., number of UAs, number of transfers).
- **Signature:** The anchors signed hash.

4.9.10 Anchoring Process

1. Logs are rolled into Merkle trees.
2. Roots are compiled into an Anchor Record.
3. Anchor Record is signed and transmitted to the parent.
4. Parent verifies signature, validates sequence, and appends the anchor to its own logs.

4.9.11 Upstream Verification

- Parents maintain a record of received Anchor Records from children.
- If a child fails to deliver an anchor within its required cadence, the parent may mark it as suspended until compliance resumes.
- Auditors can traverse Anchor Records up to the Root, verifying that each child's commitments match recorded states.

4.9.12 Security Properties

- **Tamper Evidence:** Unauthorized changes to logs are detected because Merkle roots will not match previously anchored commitments.
- **Continuity:** Anchor sequences enforce forward progress and detect rollback.
- **Accountability:** Every actor's actions are bound to their certificate and signature, preventing repudiation.
- **Auditability:** Auditors can reconstruct the history of any UA, object, or transaction through anchored proofs, without requiring access to raw PII.

4.9.13 Rationale

Logs provide a detailed record of events, while Anchor Records provide compact, verifiable commitments that prevent rewriting history. By enforcing strict cadence, signed roots, and upstream verification, UDIF ensures a continuous chain of custody for both identities and objects. This mechanism is central to UDIF's design goal of immutable accountability.

4.10 Transport: Sessions, Headers, Timing, Ratchets

This section defines how UDIF participants establish, maintain, and retire secure tunnels. It explains the role of each message, header field, timer, key, and state transition, and how these parts enforce authenticity, confidentiality, minimal disclosure, and tamper-evident audit.

4.10.1 Overview and Design Goals

A UDIF "transport session" is a long-lived, mutually authenticated, TCP tunnel protected by a post-quantum AEAD. Each session has:

- a **compile-time suite** (no runtime negotiation),
- a **strictly monotonic sequence** (any gap is fatal),
- a **bounded time window** (±60 s),
- periodic **asymmetric ratchets** (hourly, for branch ↔ branch trunks),
- and **minimal control messages** with authenticated headers (AAD).

TCP gives ordered, reliable delivery, so UDIF can forbid out-of-order packets, simplify replay logic, and fail fast on any gap. Fixed suites eliminate downgrade and reduce parser complexity. Periodic ratchets limit long-term key exposure and compartmentalize state.

4.10.2 Session Roles and State Machine

Roles. Either side may be a GC/BC/UA, but the *role pair* determines cadence and ratchet rules:

- **UA ↔ GC**: short-lived ephemeral tunnel; no periodic asymmetric ratchet.
- **BC ↔ BC (trunk)**: long-lived; hourly asymmetric ratchet (±5 min jitter).

States:

IDLE → HELLO_SENT → HELLO_RCVD → ESTABLISHED → {RATCHETING ↔ ESTABLISHED} → CLOSED

What each state means:

- HELLO_* exchange authenticates certificates, matches suites, and sets up KEM materials.
- ESTABLISHED means AEAD keys are derived and seq=0, epoch=0.
- RATCHETING is the hourly (trunks) asymmetric re-key; on success: epoch++, seq→0.
- CLOSED zeroizes keys and discards state.

4.10.3 Handshake Protocol (Mutual-Auth, PQ)

Bind both parties' identities to a fresh session secret using post-quantum KEM plus signatures over the transcript, with no suite negotiation.

4.10.4 Messages (conceptual)

1. **HelloInit (A → B)**
   Fields: { suite_id, timeA, nonceA, certA, kem_pubA, sigA }

   Role: A proposes a session under its compile-time suite.

   Checks (B): suite match, time window, verify certA chain, verify sigA over the hello (binds A's identity to this transcript), enforce revocation/suspension.

2. **HelloResp (B → A)**
   Fields: { suite_id, timeB, nonceB, certB, kem_pubB, sigB, ctB }

   Role: B accepts suite, authenticates to A, and encapsulates to kem_pubA producing ctB/ssB.

   Checks (A): suite match, time window, verify certB, verify sigB, KEM-decaps ctB → ssB.

3. **KeyConfirm (A → B)**
   Fields: { ctA, kc_sigA }

   Role: A encapsulates to kem_pubB producing ctA/ssA and signs a key-confirm (binds to transcript).

   Checks (B): decaps ctA → ssA, verify kc_sigA.

4. **Finish (B → A)**
   Fields: { kc_sigB }

   Role: B signs final transcript digest; both sides derive keys and enter ESTABLISHED.

   Checks (A): verify kc_sigB.

Two KEM encapsulations create two independent shared secrets (ssA, ssB) mixed into the KDF with both nonces and the transcript hash. Either party's contribution is required; this resists key compromise and reflection.

4.10.5 Key Schedule

transcript = Hash("UDIF:SESS-KDF:V1" || all_hellos_and_cts)

IKM = ssA || ssB || nonceA || nonceB || transcript

One cSHAKE-256 invocation expands to a 136-byte *key-block* (Keccak-256 rate). Use the entire block as seed material and slice deterministically:

TX_key = bytes[0..31]

TX_nonce= bytes[32..63]

RX_key = bytes[64..95]

RX_nonce= bytes[96..127]

state   = SHA3-256(key_block)

Role of each piece.
nonceA/B prevent cross-session collisions; transcript binds identities and KEM artifacts; full-block cSHAKE uses all entropy; state is the chaining value for future ratchets.

4.10.6 Record Format and Authenticated Header (AAD)

UDIF frames are AEAD-sealed records with a compact, authenticated header:

Name	Type	Size (bits)	Description
flags	Uint8	8	Bits: 0=DATA, 1=KEEPALIVE, 2=CLOSE, 3=CONTROL (ANCHOR etc.)
seq	Uint64	64	Strictly monotonic per direction; starts at 0 per epoch
utctime	Uint64	64	Sender's wall time (UTC seconds)
epoch	Uint64	64	Ratchet counter; starts at 0; ++ on ratchet
suite_id	Uint8	8	Compile-time suite guard

Table 4.10.6 AEAD Header Structure

Why each field exists:

- **flags:** Minimal control channel (keepalive, orderly close, anchor/control).
- **seq:** Anti-replay and anti-reordering. Since TCP is ordered, any gap or regression is fatal (session reset).
- **utctime:** Enforces ±60 s acceptance window; mitigates delayed replay.
- **epoch:** Separates key epochs; old-epoch packets are dropped.
- **suite_id:** Hard guard against cross-suite injection.

Payload: Encrypted under TX_key/TX_nonce using AEAD (RCS-256 + KMAC-256); associated data is exactly the Header AAD. Empty payload (flags=KEEPALIVE) is allowed.

4.11 Message Processing Rules

4.11.1 Sender

1. Build Header AAD with current seq, utctime=now, epoch, suite_id.
2. Seal payload with AEAD using TX_key/TX_nonce, AAD=Header AAD.
3. Transmit; increment seq.

4.11.2 Receiver

1. **Time gate:** reject if |now - utctime| > 60 s.
2. **Sequence gate:** require seq == last+1; else fatal (SEQ_INVALID).
3. **Epoch gate:** require epoch == current_epoch; otherwise drop (old) or fatal (future epoch without negotiated ratchet).
4. **AAD** verify + tag check: if AEAD fails → AAD_INVALID (fatal).
5. Deliver plaintext to application.

Short-circuit low-cost checks (time/seq/epoch) before AEAD to mitigate DoS.

4.11.3 Liveness: Keepalive, Idle, Close

- **Keepalive:** If no app traffic for 120 s, send flags=KEEPALIVE (empty payload).
- **Idle teardown:** If no traffic received for 2 × keepalive, close session (zeroize keys).
- **Orderly close:** Send flags=CLOSE record; receiver acknowledges by closing after delivering in-flight data.

Role of timers: they bound state lifetimes and enable fast detection of dead peers without excessive chatter.

4.12 Asymmetric Ratchet

4.12.1 Asymmetric Ratchet (BC ↔ BC trunks)

Cadence: Every 1 hour with ±5 min jitter per side (staggered to reduce simultaneity).
Goal: Refresh IKM using new KEM contributions; reset seq; epoch++.

Procedure (one round-trip):

RatchetInit (X → Y): { ratchet_nonceX, kem_pubX, sigX } over the ratchet transcript.

RatchetResp (Y → X): { ratchet_nonceY, kem_pubY, ctY, sigY } where: ctY = Encaps(kem_pubX) → ssY.

RatchetFin (X → Y): { ctX, sigX2 } where ctX = Encaps(kem_pubY) → ssX.

Derive new keys exactly as in the initial handshake but with ratchet_nonce X/Y and new ssX/ssY. On success:

- epoch = epoch + 1
- seq = 0 (per direction)

- old keys are zeroized immediately.

New KEM inputs create fresh entropy independent of past state, improving forward-secrecy and compromise containment for long-lived trunks.

4.12.2 Errors and Fault Handling

Fatal (close session):
SUITE_MISMATCH, AUTH_FAILURE, CERT_EXPIRED, CERT_REVOKED, CAP_REVOKED, AAD_INVALID, SEQ_INVALID, TIME_WINDOW_EXCEEDED, EPOCH_MISMATCH, INTERNAL_ERROR.

Non-fatal (application-layer result):
DENY, NOT_OWNER, NO_SUCH_OBJECT, PREDICATE_NOT_ALLOWED, REGISTRY_STALE, PROOF_NOT_AVAILABLE.

Behavior: On fatal transport error, zeroize keys/state and tear down TCP. Log the fault (digest-only metadata) and increment telemetry counters. Reconnection uses a fresh handshake.

Role of telemetry. Counters (rx_ok, rx_drop_time, seq_faults, auth_fail, ratchets) help detect abuse or misconfiguration without leaking PII.

4.12.3 Memory Hygiene and Constant-Time

- All cryptographic operations use constant-time primitives.
- Secrets (KEM shares, keys, nonces, state) are copied with secure memutils and **explicitly zeroized** on rotation/close.
- Tag comparisons are constant-time.
- Errors return fixed-size responses; DENY vs OK is padded so as not to signal policy through length.

Many attacks arise from tiny timing or memory-lifetime mistakes. Hygiene is part of the transport contract, not an afterthought.

4.12.4 How Parts Fit Together

Hello/Key Confirm bind **identity** (signatures) and freshness (nonces) to confidentiality (KEM), then expand via cSHAKE into direction-separated AEAD keys.

Header fields make the record self-verifiable: time, suite, epoch, and sequence allow immediate rejection of stale, cross-suite, or mis-ordered data—before heavy cryptography.

TCP + strict seq removes a class of replay/OOO complexity; any gap is a loud failure → fail fast, recover clean.

Hourly ratchets on trunks limit key exposure and compartmentalize compromise.

Timers (keepalive/idleness) give predictable liveness checks and bound resource usage.

Zeroization and constant-time requirements keep side channels out of the transport.

Together, these choices deliver a small, auditable, and robust transport core that supports UDIF's higher-level guarantees: minimal disclosure, immutable accountability, and strong, PQ-secure authenticity across domains.

4.13 Query Model (Predicates and Responses)

This section defines how *queries* are expressed, processed, and answered within UDIF. Queries are the *fundamental mechanism* for retrieving information from the framework. Their design reflects UDIF's overarching principles: minimal disclosure, default-deny, strong authentication, and immutable audit.

Every element; the query types, predicate language, capability enforcement, evaluation rules, and response forms, is designed to balance privacy with accountability. This section explains the purpose of each design element and how it supports UDIF's goals.

4.13.1 Design Principles

- **Minimal Disclosure:** Queries reveal the *least information necessary*. Instead of "what is the value?", the model asks "does this property hold?".
- **Predicate-Based:** Queries are expressed as *predicates* (*yes*/*no* checks) against committed state (registries, attributes, logs).
- **Capability-Governed:** No query executes unless the caller holds a valid *capability* authorizing that predicate type.
- **Default-Deny:** The absence of an explicit capability means automatic denial.
- **Canonical Encoding:** Queries and responses are canonical TLV containers, ensuring determinism and tamper-evidence.
- **Auditability:** Every query; permitted or denied is logged by the Group Controller (GC) or Branch Controller (BC) that processes it.

4.13.2 Query Types (Predicate Families)

UDIF-Core supports four core predicate families. Each maps to fundamental verification needs in identity and asset management.

4.13.3 Existence Queries

Purpose: Determine if an identity or object exists in the framework.

- Example: *"Does UA with serial X exist under GC Y?"*
- Response: *yes* or *no*.
- Why needed: Supports service access checks without disclosing details.
- Capability required: QUERY_EXIST.

4.13.4 Ownership / Binding Queries

Purpose: Verify the relationship between an object and a UA.

- Example: *"Is Object O currently owned by UA U?"*
- Response: *yes* or *no*.
- Why needed: Supports proof of ownership and authorization checks.
- Capability required: QUERY_OWNER_BINDING.

4.13.5 Attribute-Bucket Queries

Purpose: Check if an attribute falls within a permitted bucket, without exposing raw values.

- Example: *"Is UA U's status = active?"*
- Example: *"Is Object O marked as destroyed?"*
- Response: *yes* or *no*.
- Why needed: Allows operational checks (active/suspended) and regulatory checks (sanctions, KYC tier) without full disclosure.
- Capability required: QUERY_ATTR_BUCKET.

Note: Exact-value queries are forbidden in core; only predefined buckets are supported.

4.13.6 Membership Proof Queries

Purpose: Prove that an object is a member of a UA's registry at a specific epoch.

- Example: *"Prove Object O was in UA U's registry at epoch E."*
- Response:
  - *yes* or *no*
  - Optional Merkle proof path (if capability allows).
- Why needed: Enables verifiable provenance without revealing unrelated registry entries.
- Capability required: PROVE_MEMBERSHIP.

4.13.7 Query Structure

Name	Type	Size (bits)	Description
query_id	Uint8 Array	128	Unique ID for audit correlation
type_code	Uint16	16	EXIS, OWN, ATTR, PROOF
target_serial	Uint8 Array	128	UA or Object serial
predicate (TLV)	Uint8 Array	Variable	E.g., owner=U, status=active
time	Uint64	64	Optional epoch/time anchor
capability_ref	Uint8 Array	256	Digest of capability used

Table 4.13.7 Query Structure

Role of each field:

- query_id: ensures responses can be matched to queries and logged consistently.
- type_code: enforces canonical predicate types.
- target_serial: binds query to a specific UA or Object.
- predicate: contains bucket checks or ownership binding.
- time: allows queries against anchored history, not just current state.

- capability_ref: binds execution to a specific authorized capability.

4.13.8 Response Structure

Responses are TLV-encoded, canonically bound to queries.

Name	Type	Size (bits)	Description
query_id	Uint8 Array	128	Echoed from Query
verdict	Uint8	8	0=NO, 1=YES, 2=DENY
proof	Uint8 Array	Variable	Optional Merkle path or attestation
time	Uint64	64	Time of response
signature	Uint8 Array	Variable	Signature of GC/BC issuing response

Table 4.13.8 Query Response Structure

Role of each field:

- query_id: guarantees one-to-one binding of request and reply.
- verdict: simple tri-state result (yes, no, deny).
- proof: optional evidence, only if capability allows (e.g., Merkle proof).
- time: timestamps the response for replay protection.
- signature: authenticates response origin and ensures non-repudiation.

4.13.9 Query Processing Rules

1. **Capability Check:** GC/BC verifies the query is authorized by the caller's certificate and capabilities.
    a. If not authorized → respond with verdict=DENY.
2. **Predicate Evaluation:**
    a. Existence → check membership log.
    b. Ownership → check registry root binding.
    c. Attribute bucket → check status flag.
    d. Membership proof → generate Merkle path if permitted.
3. **Logging:** All queries (*yes*/*no*/*deny*) are appended to Membership Log with query digest and verdict.
4. **Anchoring:** Query digests are eventually incorporated into Anchor Records for tamper-evident audit.

4.13.10 Cross-Domain Queries (Treaties)

- Queries across domains flow only through treaty-enabled GCs/BCs.
- The treaty defines which predicate families are allowed.
- Forwarded queries are double-logged: once by the origin GC, once by the treaty peer.
- Responses are signed by the treaty peer and relayed back through the origin GC.
- No raw data crosses domains; only predicates and authorized proofs.

4.13.11 Security Properties

- **Minimal disclosure:** Queries never return raw identifiers or attributes.
- **Non-repudiation:** All responses are signed; all queries are logged.

- **Replay protection:** Responses are bound to query IDs and timestamps.
- **Auditability:** Every query leaves a digest trail in logs and Anchor Records.
- **Treaty containment:** Cross-domain queries cannot exceed explicitly authorized predicates.
- **Default deny:** If capability is absent or invalid, the result is always DENY.

4.13.12 Rationale

The Query Model embodies UDIF's philosophy: ask only what you are allowed to know, and get only what you need to act. By constraining queries to canonical predicate families, enforcing strict capability checks, and logging every event, UDIF achieves both privacy (minimal information exposure) and accountability (everything is signed, logged, and anchored).

4.14 Peering Treaties and Cross-Domain Operation

This section defines how distinct UDIF domains interoperate without undermining the framework's principles of least privilege, default-deny, and immutable accountability. Peering treaties are the sole mechanism by which controlled cross-domain queries are permitted. Their design ensures that cooperation is possible, but trust is bounded, explicit, and auditable.

4.14.1 Purpose of Treaties

In UDIF, each domain (rooted by its own Root authority) is sovereign. Cross-domain queries must not grant unilateral authority to foreign entities. To support federation between states, institutions, or enterprises UDIF defines Peering Treaties:

- **Scope-limited:** Treaties enumerate exactly which predicate families (Existence, Ownership, Attribute-Bucket, Membership Proof) may be forwarded.
- **Capability-enforced:** Each treaty is bound to a capability mask agreed by both parties.
- **Auditable:** All treaty queries and responses are logged in both domains.
- **Revocable:** Either party may terminate the treaty by revoking the capability chain.

4.14.2 Treaty Definition

A treaty is a bilateral, cryptographically signed agreement between two domain controllers (BCs or GCs).

4.14.1 Treaty Structure

Name	Type	Size (bits)	Description
treaty_id	Uint8 Array	128	Unique identifier
domainA_serial	Uint8 Array	128	Serial of Domain A branch
domainB_serial	Uint8 Array	128	Serial of Domain B branch
scope_bitmap	Uint32	32	Allowed predicate families
duration	Uint64	64	Expiry time
policy_epoch	Uint32	32	Policy profile version
signatureA	Uint8 Array	Variable	Domain A's signature

signatureB	Uint8 Array	Variable	Domain B's signature

Table 4.14.1 Treaty Structure

4.14.2 Role of each field:

- treaty_id: anchors treaty references in logs and proofs.
- domain*_serial: binds treaty to specific controllers; cannot be relayed elsewhere.
- scope_bitmap: defines which queries are allowed.
- duration: enforces automatic expiry.
- policy_epoch: pins treaty to a defined policy set.
- signatures: provide non-repudiation by both parties.

Cross-Domain Query Flow

4.14.3 Originating Query

- A UA submits a query to its GC.
- GC checks local capabilities. If the predicate is cross-domain, the GC verifies that a valid treaty exists.
- If no treaty → DENY.

4.14.4 Treaty Forwarding

- GC encodes the query into canonical TLV, attaching treaty ID.
- GC logs the query in its own Membership Log with verdict=PENDING_TREATY.
- Query is transmitted over the treaty tunnel (a standard UDIF transport session between domain controllers).

4.20.5 Remote Processing

- Remote GC/BC validates treaty scope, enforces capability masks, and evaluates predicate.
- Response is signed and logged locally.
- Proofs (e.g., Merkle paths) are only attached if the treaty scope allows it.

4.14.6 Returning Result

- Response is transmitted back over the treaty tunnel.
- Origin GC signs receipt, logs verdict in its own Membership Log, and delivers result to UA.

4.14.7 Logging and Anchoring

- **Dual Logging:** Both the origin domain and the treaty domain log every query and response.
- **Anchor Inclusion:** Treaty-related events are rolled into Anchor Records, ensuring they propagate upstream for external audit.
- **Cross-verifiability:** Auditors can compare Anchor Records from both domains to confirm consistency.

4.14.8 Security Properties

- **Non-transitive:** Treaties bind only the two explicit parties; queries cannot be relayed to third domains.
- **Scoped:** Only predicate families in scope_bitmap may be used. For example, a treaty may allow existence checks but forbid ownership proofs.
- **Revocable:** Either domain may revoke by suspending the treaty capability or letting it expire.
- **Tamper-evident:** Every query and response is signed and logged; inconsistencies between domains can be detected.
- **Default-deny:** Without a treaty, cross-domain queries are rejected outright.

4.14.9 Example Use Cases

- **Financial Federation:** Two banks in different jurisdictions allow cross-domain proof that an account exists and is active, without exposing balances.
- **Government Cooperation:** Two national ID systems allow existence and age-of-majority predicates for border control.
- **Enterprise Alliances:** Partner companies establish treaties to verify employee credentials across organizations.

Rationale

Treaties enable federation without collapsing trust boundaries. By scoping cross-domain operations to narrowly defined predicates, treaties prevent data overexposure while still providing regulators, institutions, and users with verifiable, auditable, and limited proofs. This mechanism transforms what could be unbounded interconnection into controlled, policy-driven, cryptographically bound cooperation.

4.15 Governance and Administrative Verbs

This section defines the control operations available to parents (Roots, Branch Controllers, Group Controllers) over their immediate children. Governance in UDIF is hierarchical: each entity can only manage those directly below it in the trust tree. These verbs form the administrative backbone of UDIF, allowing enrollment, suspension, revocation, and registry updates to be conducted in a controlled and auditable manner.

Every governance action is signed by the issuing authority, logged locally, and anchored upstream. This guarantees non-repudiation, auditability, and tamper evidence across the domain.

4.15.1 Principles of Governance

- **Parent-only control:** An entity may only govern its direct children. Cross-branch or cross-domain control is not possible.
- **Default-deny:** No rights are assumed. Rights and status are explicitly granted, suspended, or revoked.
- **Dual recording:** Both the issuer and the recipient log the event, ensuring cross-verification.
- **Immutable audit:** All verbs produce Membership Log entries and propagate through Anchor Records.

- **Scoped authority:** Branches administrate groups or sub-branches but do not own objects; Users own objects but cannot alter governance.

4.15.2 User Agent Lifecycle

- **UA_ENROLL**
  Purpose: Admit a new UA into a group.
  Action: GC issues a UA certificate and records enrollment in the Membership Log.
  Role in design: Defines entry point for identities; ensures each UA is cryptographically bound to a GC.
- **UA_SUSPEND**
  Purpose: Temporarily restrict a UA's rights (e.g., failed audit, suspected compromise).
  Action: GC marks UA as suspended; UA cannot forward queries upstream.
  Role in design: Provides containment without permanent revocation; forces audit resolution.
- **UA_RESUME**
  Purpose: Reinstate a previously suspended UA.
  Action: GC records reinstatement; UA regains its prior capabilities.
  Role in design: Minimizes disruption once audit or compromise is resolved.
- **UA_REVOKE**
  Purpose: Permanently remove a UA from a group.
  Action: Certificate marked revoked; all associated capabilities invalidated.
  Role in design: Provides hard stop for compromised or terminated identities.

4.15.3 Branch Lifecycle

- **BRANCH_CREATE**
  Purpose: Instantiate a new subordinate branch or group.
  Action: Parent BC issues branch certificate; records creation in log.
  Role in design: Expands hierarchy while keeping trust anchored.
- **BRANCH_SUSPEND**
  Purpose: Temporarily disable a branch's ability to forward queries or anchor records.
  Action: Parent BC records suspension; children of suspended branch cannot interact upstream.
  Role in design: Limits potential spread of compromise; supports administrative sanctions.
- **BRANCH_RESUME**
  Purpose: Reinstate suspended branch.
  Action: Parent BC records resumption; branch regains prior capabilities.
  Role in design: Ensures recovery pathway without permanent restructuring.
- **BRANCH_REVOKE**
  Purpose: Retire a branch permanently.
  Action: Certificate revoked; downstream certificates invalidated.
  Role in design: Provides hard reset; usually invoked for breach or restructuring.

4.15.4 Capability Management

- **CAP_GRANT**
  Purpose: Issue new capabilities to a child (UA or branch).
  Action: Parent generates capability bitmap + KMAC tag; embeds in certificate or issues as

token.
Role in design: Grants least-privilege rights explicitly; enforces default-deny.

- **CAP_REVOKE**
  Purpose: Invalidate a previously granted capability.
  Action: Parent logs revocation; capability digest marked revoked.
  Role in design: Limits misuse of outdated or overextended rights.

4.15.5 Commit Actions

- **REGISTRY_COMMIT**
  Purpose: Commit a UA's registry root after update.
  Action: GC logs and signs root digest.
  Role in design: Binds object ownership changes into the audit chain.
- **ANCHOR_COMMIT**
  Purpose: Transmit an Anchor Record upstream.
  Action: Child signs Anchor Record; parent logs and verifies.
  Role in design: Maintains chain of custody across hierarchy; ensures accountability.

4.15.6 Event Structure

Each governance action is encoded as:

Name	Type	Size (bits)	Description
event_code	Uint16	16	verb identifier
subject_serial	Uint8 Array	128	target UA/BC certificate serial
parent_serial	Uint8 Array	128	issuer certificate serial
time	Uint64	64	UTC timestamp
data	Uint8 Array	Variable	optional context (capability bitmap, registry root)
signature	Uint8 Array	Variable	parent's signature over fields 1–5

Table 4.15.6 Governance Event Structure

Roles of fields:

- event_code: unambiguous verb reference.
- subject_serial: binds event to target.
- parent_serial: identifies issuer for audit.
- time: ensures replay protection and audit sequence.
- data: contextual payload (e.g., revoked capability digest).
- signature: provides non-repudiation.

4.15.7 Logging and Anchoring

- **Local Logging:** Every Admin Event is appended to the Membership Log.
- **Dual Logging:** Both parent and child log the event for reconciliation.
- **Anchoring:** Events are rolled into Anchor Records, ensuring higher authorities validate them.
- **Auditability:** External auditors can traverse Anchor Records to verify that every child's state transitions are authorized and properly logged.

4.15.8 Security Properties

- **Non-repudiation:** Every verb is signed by the parent.
- **Tamper evidence:** Anchoring prevents rewriting history.
- **Containment:** Suspension isolates compromised nodes without immediate revocation.
- **Revocability:** Hard revocation provides finality when required.
- **Least privilege:** Capabilities are only granted, never assumed; revocation is explicit.

Governance verbs provide the control plane of UDIF. They enforce hierarchical authority, restrict overreach, and ensure every state transition is authenticated, logged, and auditable. By combining enrollment, suspension, revocation, and capability management into a canonical set of verbs, UDIF achieves clarity, consistency, and enforceable accountability across domains.

4.16 Error Taxonomy and Telemetry

This section defines how UDIF classifies, reports, and manages *errors* in operation. Error handling is critical because it directly influences security posture, auditability, and resilience. Mismanaged errors can create side channels, inconsistent state, or ambiguity that attackers may exploit. UDIF therefore mandates a clear, minimal, and canonical taxonomy of error codes, and requires that all errors be logged, signed, and anchored for audit.

4.16.1 Principles

- **Clarity:** Every failure is assigned an explicit error code; no silent drops.
- **Minimal disclosure:** Errors reveal only what is strictly necessary (never internal state or secrets).
- **Canonical encoding:** Errors are represented in TLV form, signed, and bound to context.
- **Auditability:** All errors are logged by the authority (GC/BC) handling them and eventually committed upstream via Anchor Records.
- **Fixed shape:** Error responses are constant-length where possible, to prevent information leakage through size.
- **Separation of concerns:** Transport-layer errors close sessions; application-layer errors return denials while maintaining the session.

4.16.2 Error Classes

UDIF defines two primary classes of errors: Transport/Handshake (fatal) and Application/Policy (non-fatal).

4.16.3 Transport and Handshake Errors (Fatal)

These indicate protocol violations or cryptographic failures. They always result in immediate session closure and key zeroization.

- SUITE_MISMATCH – peer presented a suite_id different from the domain's compile-time suite.
- AUTH_FAILURE – signature verification or certificate chain validation failed.

- CERT_EXPIRED – certificate validity window is past.
- CERT_REVOKED – certificate appears on revocation list.
- CAP_REVOKED – presented capability is revoked or invalid.
- AAD_INVALID – authenticated header failed AEAD verification.
- SEQ_INVALID – sequence number not strictly monotonic.
- TIME_WINDOW_EXCEEDED – header timestamp outside ±60s window.
- EPOCH_MISMATCH – epoch in header does not match session epoch.
- INTERNAL_ERROR – unrecoverable system error (e.g., failed memory operation).

4.16.4 Application and Policy Errors (Non-fatal)

These indicate denied requests at the query or administrative level. Sessions continue, but the specific request is refused.

- DENY – request not authorized by capabilities.
- NO_SUCH_OBJECT – referenced object does not exist in registry.
- NOT_OWNER – requesting UA does not own referenced object.
- PREDICATE_NOT_ALLOWED – predicate type not permitted by capability.
- REGISTRY_STALE – requested proof references an outdated registry epoch.
- PROOF_NOT_AVAILABLE – Merkle path or historical proof cannot be provided.

4.16.5 Error Encoding

Errors are TLV-encoded for consistency and audit.

Name	Type	Size (bits)	Description
error_code	Uint16	16	Enumerated code
context_id	Uint8 Array	128	Query_id or tx_id causing error
subject_serial	Uint8 Array	128	UA/BC certificate serial
time	Uint64	64	UTC timestamp
signature	Uint8 Array	Variable	GC/BC signature over 1–4

Table 4.16.5 Error Encoding Structure

Roles of fields:

- error_code: unambiguous taxonomy reference.
- context_id: binds error to originating request.
- subject_serial: identifies entity at fault (or target).
- time: records moment of detection.
- signature: ensures non-repudiation and prevents forged error events.

4.16.6 Logging and Anchoring

- **Local logging:** Every error event is appended to the Membership Log.
- **Dual recording:** If relevant, both issuer and recipient record the error.
- **Anchor propagation:** Errors are included in the next Anchor Record (via Merkle root commitments).
- **Auditor visibility:** Auditors can confirm whether errors were handled consistently across domains.

4.16.7 Telemetry Counters

UDIF requires each GC/BC to maintain telemetry counters to track error statistics. These counters help detect misuse, failures, or potential attacks without exposing PII.

4.16.8 Core Counters

- rx_ok – successfully received packets.
- tx_ok – successfully transmitted packets.
- rx_drop_time – packets dropped due to time window violations.
- seq_faults – sequence errors detected.
- auth_fail – authentication failures.
- denies – number of DENY responses issued.
- ratchets – successful ratchet operations performed.
- anchors_sent / anchors_recv – anchors transmitted/received.

4.16.9 Usage

- **Internal monitoring:** Telemetry helps detect anomalies (e.g., many auth_fail may indicate an attack).
- **Anchoring:** Aggregated counters may be included in Anchor Records under optional counters field.
- **Privacy:** Counters must never contain user identifiers or raw queries; only aggregate counts.

4.16.10 Security Properties

- **Fail-fast on fatal errors:** Prevents partial session state from leaking or being exploited.
- **Tamper-evident errors:** Signed error events ensure that misbehavior or anomalies are auditable.
- **Minimal exposure:** Deny vs. allow is padded; error messages do not reveal sensitive policy details.
- **Operational insight:** Telemetry allows domains to measure performance and detect abuse without privacy loss.

Clear error taxonomy and structured telemetry provide predictability for implementers and verifiability for auditors. Fatal errors protect against replay, downgrade, and state desynchronization. Non-fatal errors enforce capability boundaries. Telemetry makes abuse visible while keeping user data private. Together, these mechanisms prevent ambiguity, reduce attack surface, and reinforce UDIF's principles of clarity, minimal disclosure, and immutable accountability.

4.17 Privacy Posture and Data Minimization

This section describes how UDIF enforces privacy by construction. UDIF is not only an accountability framework; it is also deliberately designed to prevent overexposure of sensitive data.

Every role, object, and log is engineered to disclose only the minimum necessary information while still delivering cryptographic verifiability and non-repudiation.

4.17.1 Core Principles

- **Default-Deny:** No entity has access to another's attributes unless explicitly granted by capabilities or access masks.
- **Minimal Disclosure:** Queries and responses return only *yes/no* verdicts, or proofs limited to the predicate requested. No raw attributes or identifiers are exposed by default.
- **Digest-Based References:** All internal references use cryptographic digests or pseudonyms, not plaintext identifiers.
- **Audit without Exposure:** Logs and Anchor Records commit only to hashes, roots, and counters never to raw PII.
- **Scoped Permissions:** Every capability is narrowly defined (verb + scope); scope determines whether results are local, intra-domain, or treaty-scoped.
- **Policy Hooking:** Profiles may overlay stricter rules (e.g., jurisdictional constraints, zero-knowledge proofs) without altering the core model.

Identity and Object Abstraction

4.17.2 User Identities

- Certificates contain only structural metadata (serial, validity, suite ID, public key).
- No personal details (names, emails, addresses) are included in UDIF-Core.
- Where real-world identifiers are required (e.g., national IDs), they exist only in **external vaults**, referenced via digest commitments.

4.17.3 Objects

- Objects are identified by serial numbers and attribute Merkle roots.
- Attributes (e.g., "asset class = bond", "status = destroyed") are stored in registries but only exposed through bucketed predicates.
- Proofs expose path membership only, not full registry contents.

4.17.4 Queries and Responses

- **Yes/No Model:** Existence, ownership, and attribute bucket queries always return binary verdicts.
- **Predicate Buckets:** Instead of raw values, attributes are checked against predefined categories (e.g., status = active, tier ≥ 2).
- **Forbidden Direct Values:** Queries like "is user age = 23" are disallowed in UDIF-Core.
- **Proof Optionality:** Membership proofs are returned only if the capability explicitly allows; otherwise, only a verdict is given.

4.17.5 Logs and Anchors

- **Membership Logs:** Record events (enrollment, suspension, revocation) using digests of certificate serials, never names or attributes.

- **Transaction Logs:** Contain only object serials, owner digests, and transaction IDs—not raw attribute sets.
- **Anchor Records:** Commit to Merkle roots and counters. Roots prove consistency without revealing underlying objects or identities.
- **Error Events:** Bound to context IDs and digests, not descriptive user data.

4.17.6 Pseudonymization

- **Optional Profile Feature:** Domains may derive per-context pseudonyms for UAs and objects using KMAC(master_key, UA_id || context).
- **Non-linkability:** Different contexts produce different pseudonyms, preventing global correlation.
- **De-pseudonymization:** Only the GC or BC with the master key can map pseudonyms back to actual serials.
- **Use Case:** Allows treaty peers to reference pseudonyms consistently without knowing true underlying identities.

4.17.7 Timing, Size, and Side-Channel Minimization

- **Fixed-size responses:** DENY, YES, NO responses are padded to equal length.
- **Timing uniformity:** Responses are delayed within a bounded jitter window to mask policy differences.
- **Constant-time crypto:** All cryptographic primitives execute in constant time to prevent side-channel leakage.
- **Error uniformity:** Application-layer denies are indistinguishable in form from valid responses (except in verdict bit).

4.17.8 Policy Hooks and Extensions

- **Zero-Knowledge Proofs:** Profiles may extend UDIF with ZK mechanisms (e.g., zk-STARKs for set membership).
- **Jurisdictional Restrictions:** A profile may require that certain predicates (e.g., residency) be evaluated only by domestic authorities.
- **Data Retention Policies:** Profiles may enforce stricter log pruning or aggregation beyond the default 180-day horizon.
- **Anonymized Telemetry:** Counters exported in Anchor Records include only aggregate statistics, never individual identifiers.

4.17.9 Security and Privacy Properties

- **Default Non-Exposure:** No raw attribute, PII, or identifier leaves its controlling domain without explicit policy.
- **Scoped Delegation:** Even with cross-domain treaties, queries are restricted to agreed predicate buckets.
- **Accountability without Overreach:** Every action is logged and anchored, but logs reveal only hashes and verdicts, never the raw data behind them.
- **Resilience Against Correlation:** Digest-based references and optional pseudonyms prevent global tracking.

- **Compliance Ready:** The model aligns with privacy regulations (e.g., GDPR's data minimization principle) by never storing or disclosing more than necessary.

Privacy in UDIF is not an afterthought; it is embedded into the architecture. By enforcing canonical digests, bucketed predicates, and deny-by-default access masks, UDIF ensures that sensitive information remains under the strict control of its rightful owner. At the same time, audit logs and Anchor Records guarantee accountability and non-repudiation. This balance, minimal exposure with maximal accountability is what distinguishes UDIF from legacy identity and asset systems.

4.18 Compliance and Interoperability Profiles

This section defines how UDIF ensures interoperability across implementations and deployments while maintaining strict cryptographic consistency. UDIF separates core compliance requirements from optional profiles, allowing systems to adopt a common foundation while tailoring features to regulatory or operational needs.

4.18.1 Compliance Levels

UDIF defines four levels of compliance. Each level is *cumulative*; higher levels add features but must preserve compatibility with lower ones.

4.18.2 Level-Core (Mandatory)

- **Canonical TLV/uvarint encoding** for all structures.
- **Compile-time suite lock**; no runtime negotiation.
- **Post-quantum primitives only:** RCS-256 AEAD, Kyber or McEliece KEM, Dilithium or SPHINCS+ signatures, SHA3/cSHAKE.
- **Registry roots and Merkle proofs** for objects.
- **Membership/Transaction Logs** with Anchor Records.
- **Capability bitmaps + access masks** with default-deny enforcement.
- **Transport sessions** with strict sequence, time window, and replay protection.

4.18.3 Level-A (Audit-Enhanced)

- Adds **stapled status proofs** for revocation/suspension checks.
- Anchor Records include optional **inclusion proofs** (Merkle paths for selected events).
- More detailed telemetry counters anchored upstream.

4.18.4 Level-P (Privacy-Maximized)

- Adds **pseudonymization** of UA/object identifiers via KMAC-derived context keys.
- Requires **bounded timing jitter** in responses to mask policy differences.
- Enforces **strict attribute buckets only**; forbids even limited raw disclosures.
- Telemetry reports are aggregated and anonymized.

4.18.5 Level-Z (Zero-Knowledge Ready, Optional/Future)

- Experimental profile supporting **ZK-STARK/zk-PSI** proofs for predicates.
- Out of scope for UDIF-Core, but reserved for future extensions.

4.18.6 Interoperability Rules

- **Lowest Common Denominator:** When two peers interoperate, they operate at the highest level supported by both, but always at least Level-Core.
- **Anchor Compatibility:** Anchor Records must always include Core fields (regroot, txroot, mroot) regardless of profile.
- **Deterministic Encoding:** All implementations must follow canonical TLV/uvarint rules to guarantee byte-exact consistency across platforms.
- **Suite Enforcement:** All participants within a domain must share the same compile-time suite; mismatches trigger SUITE_MISMATCH and abort sessions.
- **Cross-Domain Treaties:** Treaty peers must agree on compliance level for treaty queries; lowest common profile applies.

4.18.7 Migration and Evolution

- **Version Tags:** All domain separation labels carry version tags (e.g., UDIF:OBJ-DIGEST:V1). If labels change, a new version is incremented, ensuring backward compatibility.
- **Suite Upgrades:** Domains may upgrade to stronger PQ primitives by issuing new suite IDs and re-issuing certificates. Old suites may remain verifiable through anchored digests.
- **Profile Rollout:** Domains may phase in higher levels (A, P, Z) gradually without breaking Core compatibility.

4.18.8 Interoperability with Existing Standards

UDIF is designed to overlay or complement existing frameworks rather than replace them.

- **ISO Identifiers (LEI, ISIN, IBAN):** UDIF can encapsulate them as object attributes, exposing only predicate proofs ("LEI not revoked") rather than raw values.
- **ISO 20022 / SWIFT Messaging:** UDIF proofs may be attached as extensions, binding cryptographic accountability to financial transactions.
- **PKI/X.509:** UDIF certificates may coexist with X.509; UDIF differs by enforcing default-deny, predicate queries, and PQ primitives.
- **W3C VCs/DIDs:** UDIF shares selective-disclosure principles but grounds itself in state-anchored, federated trust rather than self-sovereign models.

4.18.9 Security Properties

- **Consistency:** Canonical data model + suite lock prevents ambiguity and downgrade.
- **Auditability:** Compliance levels guarantee a minimum set of logs and roots; higher levels add richer audit.
- **Privacy Assurance:** Level-P ensures non-linkability and minimal exposure, aligning with global privacy regulations.
- **Future-Proofing:** Reserved Level-Z ensures extensibility into zero-knowledge paradigms without disrupting Core.

By defining clear compliance levels, UDIF balances universality (a minimal Core all must support) with flexibility (optional profiles tailored for audit, privacy, or advanced cryptography). This layered approach mirrors standards like TLS, where mandatory baselines coexist with optional extensions. It allows regulators, enterprises, and technical implementers to adopt UDIF at the right level of assurance, while maintaining interoperability across all deployments.

5. Operational Overview

5.0 General Overview

The Universal Digital Identity Framework (UDIF) operates as a layered, hierarchical system of authorities and participants, each fulfilling defined roles within a secured trust structure. The framework is polymorphic by design, allowing deployment across diverse domains such as financial networks, government registries, supply chains, or institutional infrastructures. Despite this flexibility, the operational flow remains constant: initialization begins at the root authority, extends through subordinate controllers, and culminates with end-user agents and the objects they own. Every step of this process is cryptographically anchored, auditable, and governed by explicit capabilities.

A UDIF network is instantiated by a **Root Authority**. The root generates its asymmetric signature keypair and defines the first certificate in the hierarchy. This certificate acts as the immutable trust anchor for all subordinate branches and establishes baseline operational parameters, including the suite identifier, policy epoch, audit cadence, and default capability masks. From this root, subordinate entities are admitted into the system in a controlled, certificate-based sequence, each inheriting only those rights explicitly delegated downward.

Branch Controllers (BCs) are the next operational tier. A branch controller may be initialized to manage either other subordinate branches or groups of users, but never both simultaneously. This one-way role assignment avoids ambiguity in delegation paths. Upon registration, the branch receives its certificate signed by the root or its parent branch. It initializes its **membership log** for administrative actions and its **transaction log** for object events. The branch is responsible for issuing certificates to its own subordinates, applying strict least-privilege rules via capability bitmaps.

Group Controllers (GCs) operate beneath branches. Their function is to manage **User Agents (UAs)**, which represent end-users, devices, or systems that hold and transact objects. A GC enforces the group's policy and access regime, authenticates its user agents, and acts as their proxy to the wider domain. UAs do not communicate directly with other users or branches; instead, all queries and transactions flow through the GC, ensuring consistent enforcement of capabilities and logging.

Each **User Agent** is enrolled into a group by receiving a certificate signed by the GC and countersigned indirectly by the parent branch. The UA initializes its **object registry**, a container for all owned objects and their associated attribute sets. The UA may originate queries, submit transactions, and exercise only those rights granted by its certificate and capability bitmap. All actions are logged locally by the GC and periodically committed upstream via Anchor Records.

Objects are instantiated within user registries. An object represents a commodity, identifier, or record with defined attributes. Its creation involves the owner UA signing the initial attribute set and registering the object with the GC. The GC logs the event, incorporates it into its transaction log, and ensures the object's Merkle root is included in the next Anchor Record. Subsequent modifications, transfers, or revocations of the object are handled through bi-modal transactions signed by both originator and counterparty, with every event recorded in the transaction log.

The operational model of UDIF is cyclic: participants generate events, these events update logs, and the logs are periodically anchored upward. At defined intervals, each GC or BC computes the Merkle

roots of its membership and transaction logs, combines them into an Anchor Record, signs it, and forwards it to its parent. The parent verifies sequence continuity, time validity, and signature authenticity before appending the digest to its own membership log. This cycle establishes a *chain of accountability*, where each entity is held to account by its parent, and the root maintains the authoritative record of the entire domain.

In this way, UDIF achieves a balance between *flexibility* (supporting arbitrary object and identity types), *security* (using post-quantum cryptography and strict least-privilege enforcement), and *auditability* (ensuring every state transition is cryptographically provable). The framework is designed so that no event can be silently omitted, altered, or replayed; every action leaves a trace in both local logs and upstream anchors.

5.1 Root Initialization

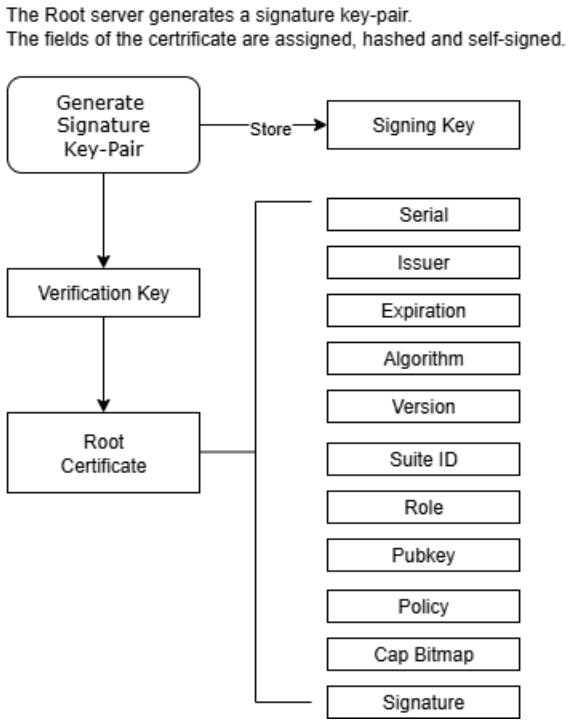


Figure 5.1 The root certificate structure

The Root Authority is the first entity created in a UDIF deployment and represents the universal trust anchor of the entire domain. Its initialization sequence is deliberately rigorous, as every certificate, transaction, and anchor chain that follows derives assurance from the correctness of this initial state. At the moment of creation, the Root generates its asymmetric signature keypair, using a post-quantum secure primitive such as Dilithium or SPHINCS+, depending on the compiled suite string assigned to the domain. This pair is generated within a protected environment, bound to the system's entropy source, and immediately used to construct the Root Certificate. The Root Certificate defines the suite identifier, the policy epoch, the validity window, and the initial

capability mask; it is self-signed, establishing an immutable genesis from which all subordinate certificates derive.

Once the certificate is complete, the Root instantiates its local membership and transaction logs. The membership log is opened with an ANCHOR_INIT record, committing the genesis certificate to the log and producing the first Merkle root. This root is hashed using SHA3-256 under the domain separation label "UDIF:ANCHOR:V1" and retained as the baseline digest for all future audits. At this stage the Root's local anchor cadence is defined; for most deployments this is an hourly or daily interval at which the Root will expect anchors from subordinate branches and aggregate them into its own record. The cadence, suite identifier, and validity parameters form the operational contract of the domain and are immutable for the life of the Root's certificate.

After establishing its logs, the Root begins its role as an issuer. When the first Branch Controllers are registered, the Root verifies their enrollment request, which is presented as a certificate signing request (CSR) containing the branch's serial identifier, generated public key, and requested role mode. The Root checks that the request conforms to the domain suite, ensures that no unauthorized capabilities are present, and applies the appropriate access mask. The Root then signs the branch certificate with its private key, producing the first subordinate certificate in the hierarchy. This exchange establishes the secure join of a Branch Controller into the network: the branch sends its CSR across a mutually authenticated tunnel, the Root replies with the signed certificate, and the branch acknowledges by committing the certificate to its local log and producing its own Anchor Record.

In parallel with subordinate enrollment, the Root initializes its treaty and governance subsystems. Where a domain is expected to interact with external domains, the Root pre-defines its treaty policy structures, including the default capability downgrades and predicate families that may be permitted. Although no treaty is active at the moment of Root creation, the structures for treaty negotiation are bound into the Root's certificate so that auditors can verify that all subsequent cross-domain agreements were contemplated from the beginning.

Internally, the Root also seeds its cryptographic state for transport and ratchet operations. It derives its initial session key material using cSHAKE-256, generating both AEAD keys and nonces for its administrative channels. This ensures that every subsequent message, even those carrying certificates or anchors during the bootstrap, is authenticated and replay-protected. When the Root first communicates with a branch, the handshake proceeds by exchanging certificates, deriving a shared session secret through a post-quantum KEM such as Kyber or McEliece, and then initializing the symmetric cipher state for the tunnel. These steps occur automatically at each join, ensuring that no cleartext or unauthenticated message ever traverses the network.

With these processes completed, the Root Authority transitions from an isolated initialization phase into the operational state. It holds the only self-signed certificate in the hierarchy, maintains the genesis anchor in its membership log, and has at least one subordinate branch securely attached. From this point forward, the Root's duties are continuous: it verifies anchors from its children, aggregates their log roots into its own anchor records, enforces policy epochs, and provides the definitive record of domain state. The correctness of every future audit rests upon this initial sequence, and the Root's role as trust anchor is both cryptographic and operational, ensuring that the network has a single, provable foundation from which the hierarchy can safely expand.

5.2 Branch Initialization

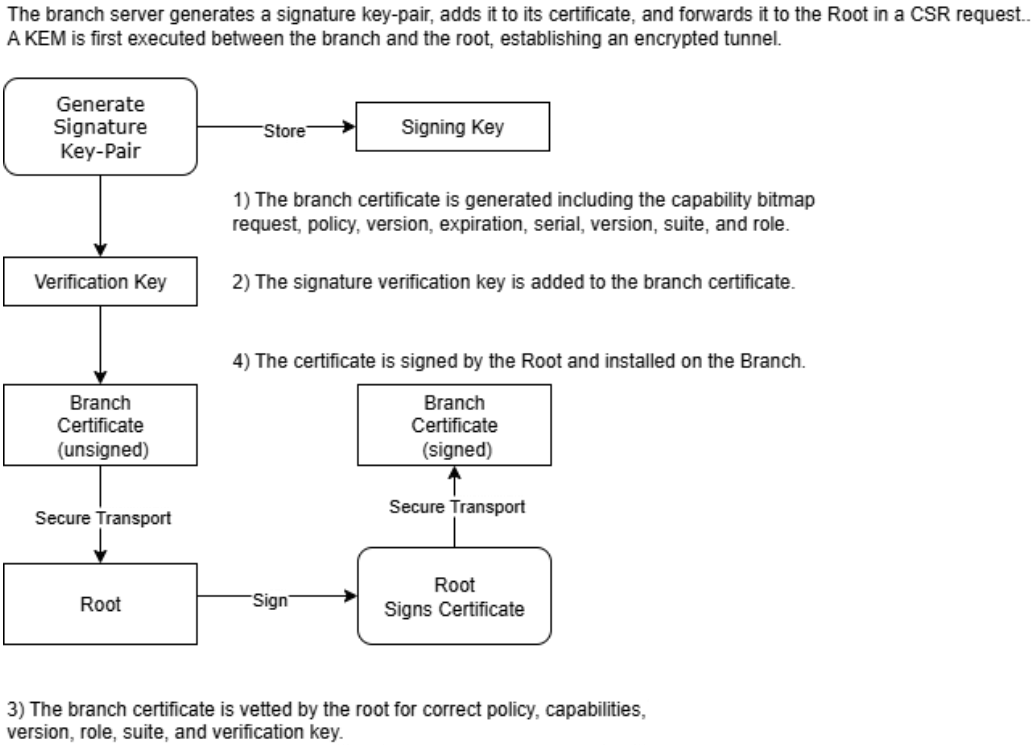


Figure 5.2 Branch certificate generation and Root signing.

A Branch Controller is the first subordinate entity admitted beneath the Root and represents the initial layer of distributed authority in the UDIF hierarchy. Its creation is not a unilateral event; rather, it is the product of an authenticated exchange between the Root and the candidate Branch, during which identities are verified, certificates are issued, and secure tunnels are established. This process ensures that the Branch joins the network under the strict control of the Root's certificate policy and inherits only the rights explicitly granted by the Root's access mask.

The Branch begins its initialization by generating a post-quantum keypair aligned with the domain's suite identifier. Typically, this will involve a signature keypair using the same primitive selected for the domain (Dilithium or SPHINCS+) and a KEM keypair for transport initialization (Kyber or McEliece). With these keys, the Branch constructs a certificate signing request (CSR). The CSR contains its serial identifier, requested role mode (branch-admin or group-admin), the public keys for signing and transport, and a proof of possession binding the request to the generated private keys.

This CSR is transmitted to the Root over a bootstrap channel. Even at this stage, all communications are protected by temporary session keys derived from an ephemeral KEM exchange. The Branch sends its CSR encapsulated under this session, and the Root validates the submission against its certificate policy. The Root ensures that the branch's suite identifier matches the domain, checks the CSR for any invalid or unauthorized capability bits, and verifies the authenticity of the enclosed public keys. If the request passes inspection, the Root signs the Branch Certificate with its own private key.

The Root then returns the signed certificate to the Branch across the authenticated tunnel. The Branch verifies the Root's signature and commits the certificate to its local membership log, opening its operational history with an enrollment event. The Branch now has authority to act within the domain, but only within the scope of its assigned mode. If it is initialized as a branch-administrator, it may issue certificates to further subordinate branches; if as a group-admin, it may issue certificates to Group Controllers and User Agents. This role is permanently fixed at initialization and cannot be changed without pruning and re-enrollment.

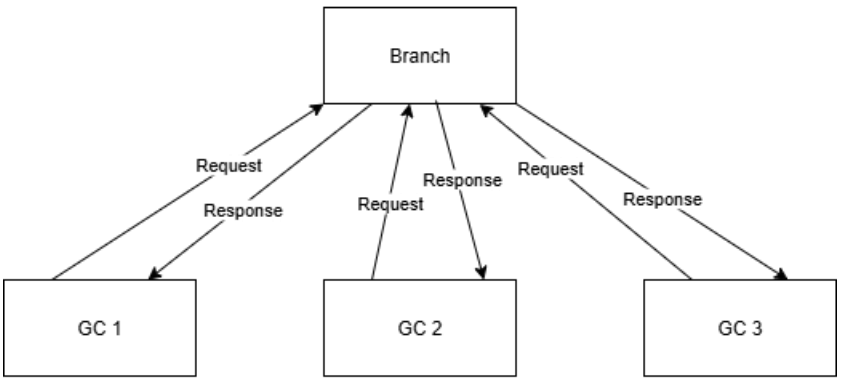
Once the Branch certificate is accepted, the Root and Branch establish their long-term administrative tunnel. This channel is built from a post-quantum KEM handshake using the Branch's certified public key. The derived shared secret seeds cSHAKE-256, which expands the entropy into AEAD keys and nonces for the RCS-256 cipher. This tunnel will carry all future control traffic, including Anchor Records, audit proofs, and treaty negotiations. Sequence numbers, time anchors, and authentication tags ensure that no replay or injection is possible.

With the secure channel established, the Branch instantiates its local membership and transaction logs. The first entries are the receipt of its own certificate and the Root's confirmation of enrollment. From this point forward, every object event and administrative action under the Branch's scope is recorded, hashed into Merkle trees, and periodically summarized into Anchor Records. At the cadence defined by the Root, the Branch computes the digests of its logs, signs them with its private key, and transmits them upstream to the Root. The Root verifies these anchors against the expected sequence and time window, logging them into its own record.

The completion of these steps marks the Branch as a fully operational subordinate in the UDIF network. The Root holds the definitive record of the Branch's creation and anchors, while the Branch maintains its own logs and begins to enroll its subordinates. The exchange of the CSR, the issuance of the certificate, the creation of the administrative tunnel, and the first anchored log entries together define the Branch's entry into the hierarchy.

5.3 Group Controller Initialization

The Branch manages the CSR exchange between GC and Branch, the issuance of the GC certificate, the establishment of the GC-Branch tunnel, and the first Anchor Records flowing upstream.



The Group Controllers register with the Branch, sending their certificates over a secure channel to be vetted and signed by the Branch server.

Figure 5.3 Group Controller registration.

A Group Controller (GC) is the point where a Branch begins to manage users directly. Its initialization process mirrors that of a Branch joining the Root, but introduces additional mechanisms specific to user and object management. The GC sits at the boundary between the administrative controllers and the user agents, and it is through the GC that all end-node certificates, queries, and transactions must pass. Its correct initialization ensures that user agents are enrolled securely, that object registries are under the strict governance of the domain, and that every action can be traced upstream to the Branch and Root.

The sequence begins when a GC candidate generates its keying material. As with branches, the GC produces a signing keypair and a transport KEM keypair consistent with the domain's compiled suite. A certificate signing request is then prepared, containing the GC's serial identifier, role declaration, its public keys, and a proof of possession. This CSR is transmitted to its parent Branch Controller over an ephemeral KEM-protected channel, ensuring that even at bootstrap the message cannot be replayed or intercepted.

The Branch receives the CSR and verifies that the suite identifier matches its own compiled suite, that the requested role is permitted under its capability bitmap, and that no unauthorized access bits are requested. Upon validation, the Branch signs the GC's certificate with its private key, thereby linking the GC into the trust hierarchy. The signed certificate is transmitted back to the GC through the authenticated channel. The GC confirms the signature using the Branch's certified public key, and then commits the certificate to its membership log as its first event.

Once the certificate has been accepted, a long-term administrative tunnel is established between the Branch and the GC. This tunnel is derived from a post-quantum KEM handshake using the GC's certified key, seeding cSHAKE-256 to produce RCS-256 session keys and nonces. The tunnel is bound by sequence numbers and validity windows, ensuring that all subsequent messages between the GC and Branch are authenticated, replay-protected, and resistant to manipulation.

With communications secured, the GC initializes its local membership and transaction logs. The membership log begins with the certificate receipt event; the transaction log is empty at initialization, ready to record object activity once user agents are enrolled. At this stage the GC also initializes its Anchor Record cadence as defined by the Branch. Periodically, it will compute the Merkle roots of its membership and transaction logs, sign them, and forward the Anchor Records upstream. These are verified by the Branch and in turn logged into its own records, creating the audit trail that binds every GC action back to the Root.

Beyond basic enrollment, the GC must also prepare its environment for user agent management. This involves enabling the certificate issuance subsystem, configuring the access mask inheritance rules, and registering its governance parameters. The GC is responsible for enforcing the default-deny model: user agents cannot query or transact outside their granted rights, and all operations are proxied through the GC's logs. This is established at initialization by configuring the GC's policy engine against the capability bitmap issued by the Branch.

At the conclusion of the initialization, the GC stands as an operational intermediary between the Branch and its user agents. It is equipped with its own certificate, a secure tunnel to its parent, initialized logs, and an Anchor Record sequence that binds its state to the Branch. From this point

forward, the GC may begin to admit user agents into the domain, creating the leaf layer of the hierarchy.

5.4 User Agent Initialization

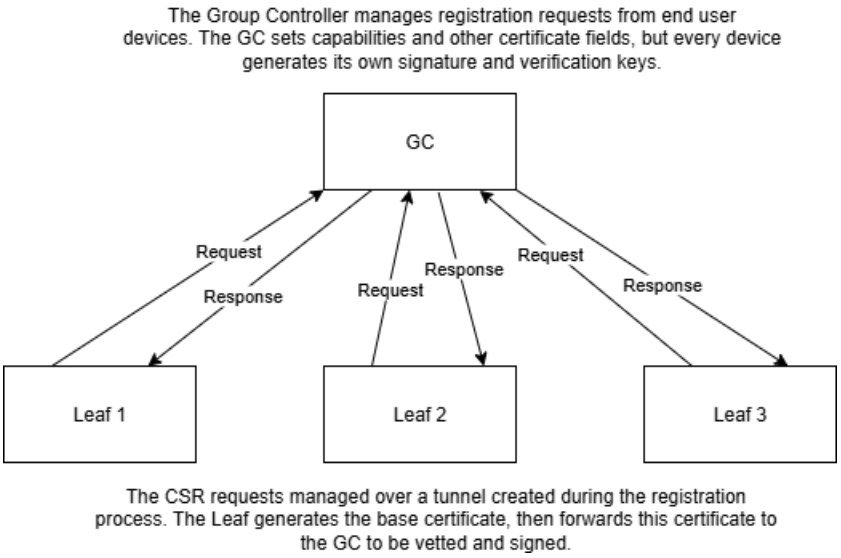


Figure 5.4 User Agent registration.

The User Agent (UA) represents the final link in the UDIF hierarchy, embodying the individual, device, or system that owns and transacts objects. Its initialization is the point at which abstract authority is extended to a concrete participant. The process ensures that a UA cannot exist without a traceable path back to the Group Controller, the parent Branch, and ultimately the Root. Every object owned by the UA, every transaction it engages in, and every query it originates is cryptographically tied to this initialization sequence.

A UA begins by generating its post-quantum keying material, comprising both a signature keypair and a transport KEM keypair conformant to the domain's suite identifier. With these keys it constructs a certificate signing request (CSR) that includes its serial identifier, its role declaration as a user agent, the generated public keys, and a proof of possession. The CSR is transmitted to the Group Controller (GC) over an ephemeral KEM-derived session. Even at this lowest level, the bootstrap channel is authenticated and replay-resistant, ensuring that enrollment cannot be intercepted or forged.

The GC receives the CSR and validates its contents. It verifies that the suite identifier matches the domain's compiled suite, checks that the UA's request falls within the capability constraints inherited from the Branch, and ensures that no unauthorized functions are requested. Once validated, the GC signs the UA certificate with its private key, binding the UA to its group. The signed certificate is returned to the UA across the authenticated session, and the UA verifies the GC's signature before committing the certificate to its local state. This certificate is also recorded in the GC's membership log, creating the first official record of the UA's existence within the domain.

Following certification, the GC and UA establish their long-term secure tunnel. Using the UA's certified public key, they perform a post-quantum KEM exchange, derive a shared secret, and expand it through cSHAKE-256 into session keys and nonces for RCS-256. This tunnel is bound by sequence numbers and time anchors; every message is authenticated and replay-protected, providing confidentiality and integrity for all UA communications.

The UA now initializes its **object registry**, the container that will hold all objects it creates or acquires. The registry is bound to the UA's certificate, and its root digest is logged at initialization in the GC's transaction log. From this point forward, any object owned by the UA must be committed into this registry, and every modification or transfer will be recorded in the GC's logs and reflected upstream in Anchor Records. The UA has no authority to transact outside its registry or beyond the capabilities defined in its certificate.

At initialization, the UA's permissions are strictly limited. It cannot query other users directly, nor can it transact outside its group. All requests flow through the GC, which enforces policy, validates access masks, and logs each action. When the UA issues its first query, such as a request to verify the existence of an object attribute, this request is signed with its private key, transmitted over the tunnel, validated by the GC against its permissions, and either answered locally or forwarded upstream if allowed. This pattern, established at initialization, ensures that no UA can operate outside the boundaries of its GC.

The initialization concludes with the UA's first Anchor Records contribution, indirect though it may be. The UA itself does not send anchors to the Branch or Root, but its certificate and registry initialization are captured in the GC's membership and transaction logs. At the next cadence, the GC computes its Merkle roots, includes the UA's enrollment events, and signs an Anchor Records that flows upstream. In this way, the Root acquires a verifiable proof that the UA exists, that its registry has been established, and that its operations are properly bound to the hierarchy.

5.5 Object Initialization

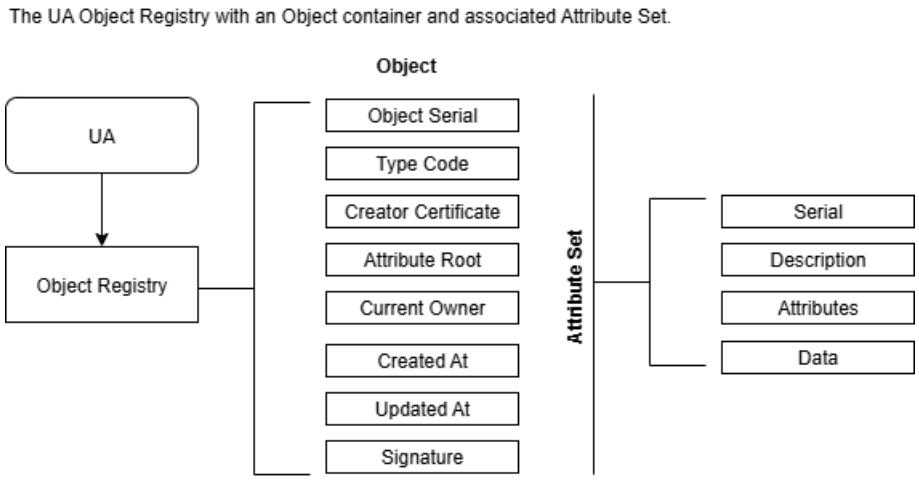


Figure 5.5 Object Initialization

Objects are the elemental data containers in UDIF, representing any commodity, identifier, or digital record under a user's control. Their initialization process establishes ownership, provenance, and accountability at the moment of creation. This process binds each object into the hierarchy by requiring both the user and the Group Controller to participate in its certification and logging. Once created, an object becomes an immutable entry in the user's registry, with all subsequent modifications or transfers appended as verifiable events.

The sequence begins when a User Agent (UA) elects to create a new object. The UA composes the object's attribute set, including its unique object serial, its descriptive metadata, and any required provenance data such as a signed hash of the creation parameters. These attributes are assembled into a canonical TLV encoding and committed into the UA's local object registry. Before the object is considered valid, the UA signs the object's digest with its private key, establishing irrefutable ownership of the object at the moment of creation.

The signed object record is then transmitted to the Group Controller (GC) over the secure tunnel established at UA initialization. The GC validates the record by verifying the UA's signature against its certified public key and ensuring that the object serial is unique within the group. Once validated, the GC appends the event to its transaction log, recording the object's serial, owner identifier, and digest. The GC then signs the transaction entry with its own private key, providing a secondary layer of attestation.

At this point, the object is considered live within the domain. The UA holds the signed object certificate in its registry, and the GC maintains a corresponding transaction record. Both entries are hashed into their respective Merkle trees, ensuring that any subsequent audit will capture the object's creation event. At the next Anchor Records cadence, the GC will compute the digest of its transaction log, which now includes the object creation event, and forward this digest to its parent Branch. The Branch, and eventually the Root, will verify and log the anchor, extending the chain of accountability. In this way, the object's existence is cryptographically bound not only to its UA owner but also to the entire domain hierarchy.

The initialization of objects also sets the stage for future transactions. Once recorded, the object can be modified (subject to capability limits), transferred to another user, or revoked. Each such action will generate a new transaction log entry, signed by the appropriate parties, and incorporated into the anchor flow. But the very first creation event is special: it establishes the canonical form of the object and binds it permanently to its owner's identity.

5.6 Network Growth

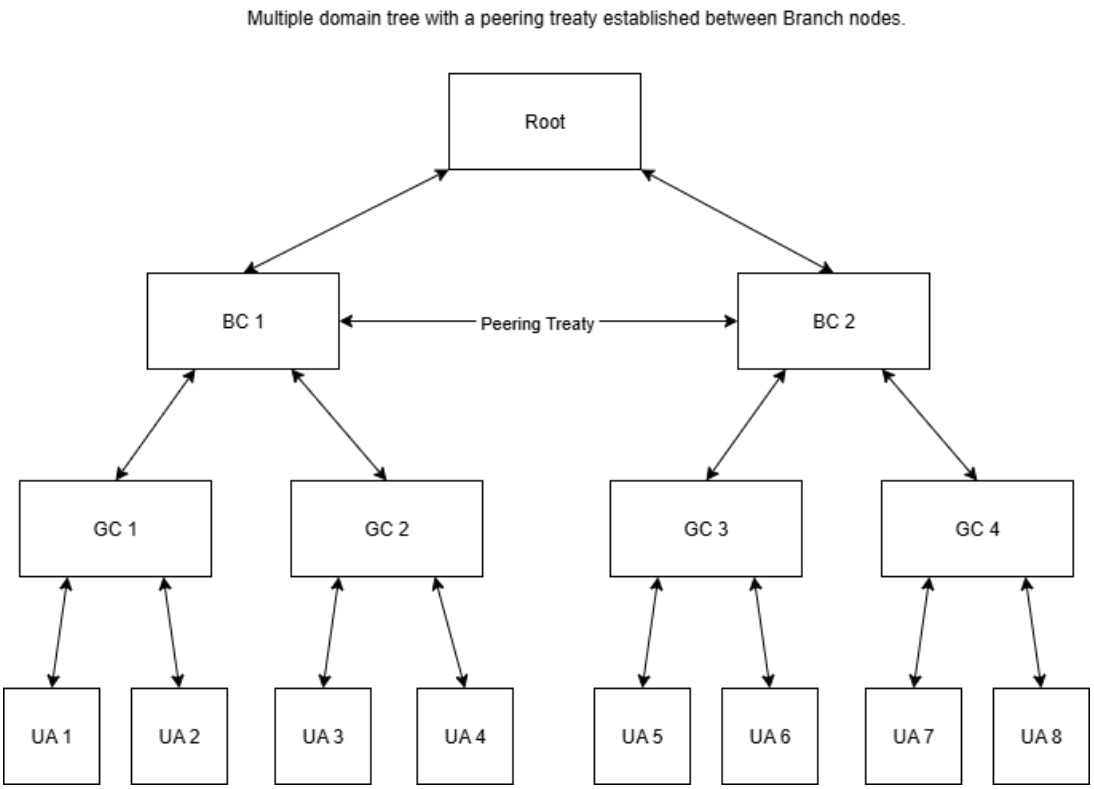


Figure 5.6 Network Growth

As a UDIF deployment matures, the network is expected to grow beyond the initial Root and a single Branch. This growth must preserve the same security guarantees established at genesis, ensuring that every new authority, group, user, and object is traceable through the certificate chain and anchor flow. Expansion is therefore a controlled process: new branches are admitted only through authenticated exchanges with their parents, group controllers are instantiated under branch authority, and user agents are registered exclusively through group controllers. At each step, network communications and cryptographic commitments enforce continuity of trust.

The first stage of network growth typically occurs at the Branch layer. A Branch initialized under the Root may create subordinate branches in order to scale administration across regions, organizational divisions, or functional units. To create a new Branch, the existing Branch acts as an issuer: it receives a certificate signing request (CSR) from the candidate, verifies it against its capability bitmap and the domain suite identifier, and then issues a Branch Certificate. This certificate is bound to the Branch's membership log, ensuring that the enrollment is recorded immutably. The new Branch then establishes its secure tunnel to the parent through a post-quantum KEM exchange, derives its session keys via cSHAKE-256, and begins producing Anchor Records. The parent Branch validates and logs these anchors, while the Root indirectly witnesses them as part of its aggregated log. In this way, the authority of the Root propagates through multiple layers without loss of auditability.

Growth also occurs horizontally as new Group Controllers are registered under a Branch. Each GC brings with it a new set of User Agents and object registries. The enrollment sequence is identical to that described previously, but scaled to multiple instances. Each GC's anchors flow to the Branch,

and the Branch consolidates them into its own Anchor Records. By scaling horizontally in this way, a Branch can manage large numbers of users and objects without overburdening a single GC.

User Agents contribute to growth at the leaf level. As more users or devices are admitted, their certificates and registries expand the scope of the group. Although UAs do not anchor directly to the Root, their enrollment and object creation events are incorporated into GC logs and Anchor Records, which in turn are validated and logged upstream. This layered anchoring ensures that even as the number of UAs grows into the millions, the Root can still audit their existence and activity by verifying a compact sequence of Merkle roots.

Network growth is not limited to vertical hierarchy. Domains may establish peering agreements across boundaries, enabling controlled queries and transactions between otherwise independent trees. Such treaties are negotiated bilaterally between Branch Controllers, authenticated through their certificates, and logged in both domains' membership logs. Once a treaty is in place, the branches establish a secure tunnel, exchange capability masks, and agree on predicate families that can be processed cross-domain. Treaty traffic is subject to the same logging and anchoring rules as intra-domain communications, ensuring that growth through federation does not dilute accountability.

Finally, growth must be balanced by mechanisms for pruning and retirement. If a branch is compromised, sanctioned, or simply no longer required, it may be pruned by its parent. This action is recorded as a revocation in the membership log, and the branch's anchors are no longer accepted. All of its subordinate controllers and users are effectively suspended until they can be re-enrolled under a new branch. In this way, network expansion remains reversible and bounded by governance.

5.7 Audit and Anchor Flow

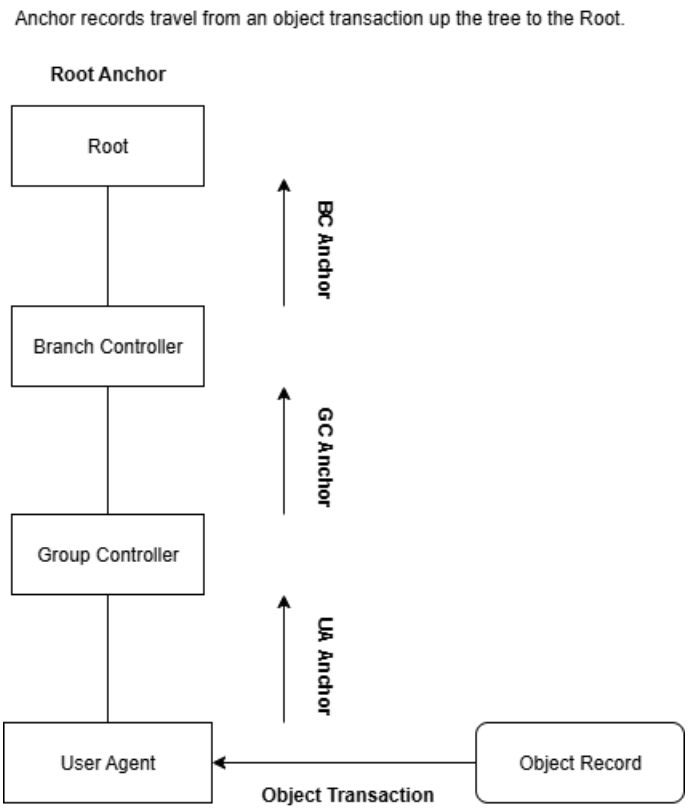


Figure 5.7 Anchor record flow

The audit and anchoring process is the heartbeat of a UDIF deployment. It is through this continual exchange of digests and verifications that the network maintains accountability, detects misbehavior, and preserves the integrity of its logs. Every authority in the hierarchy; from the Group Controller up to the Root, participates in this process, producing and verifying Anchor Records on a predictable cadence. The result is a chain of evidence that binds user actions, object events, and administrative operations to cryptographic commitments that can be verified at any point in time.

The cycle begins at the Group Controller (GC). As user agents register, create objects, and transact, their events are appended to the GC's membership and transaction logs. At the configured interval, the GC computes the Merkle roots of each log, producing compact digests that represent the entirety of the recorded events since genesis. These roots are combined into an Anchor Record, which also includes a sequence number, a timestamp, optional telemetry counters, and the GC's certificate serial. The GC signs the Anchor Record with its private key, ensuring authenticity and binding it to its identity.

The Anchor Record is then transmitted upstream to the parent Branch over the secure tunnel established at initialization. The Branch verifies the GC's signature, checks that the sequence number is strictly greater than the last accepted value, and ensures that the timestamp falls within the permissible drift window. If all checks succeed, the Branch appends the digest of the Anchor Record to its membership log. The Branch thus holds verifiable proof of the GC's state without needing to store or inspect the GC's raw logs.

The Branch itself follows the same process. At its cadence, it computes Merkle roots of its own logs, including the recently received GC anchors, and produces its own Anchor Record. This record is signed, logged, and forwarded to its parent in turn. As the process continues upward, the Root ultimately receives and verifies Anchor Records from its immediate branches, indirectly witnessing the logs of every entity in the domain. The Root appends these digests into its own membership log, creating a definitive record of domain state.

Auditing is continuous, not one-time. External auditors or oversight processes can challenge any node by requesting proofs of membership for specific events. Because logs are committed as Merkle trees, the challenged node can supply a compact path from the event digest to the anchored root, proving inclusion without disclosing unrelated entries. If a node fails to produce a valid proof, or if its Anchor Records sequence is broken, the parent suspends it. Suspension means that the node's anchors are no longer accepted and its subordinate entities are frozen; user agents under a suspended GC, for example, cannot transact until the audit failure is resolved.

Failures bubble upward. If a GC fails, its users are restricted; if a Branch fails, its GCs and their users are suspended; if a Root fails, the entire domain is considered invalid until a new root is established. This cascading suspension ensures that misbehavior or compromise cannot be hidden within a subtree; the effects are always visible to the wider network.

Anchoring also extends across domain boundaries. Where peering treaties exist, Anchor Records may be exchanged between domains to prove compliance with treaty terms. Each domain logs the peer's anchor digests, ensuring that both sides maintain verifiable state. In this way, federation does not weaken auditability but strengthens it by binding foreign entities to the same cycle of commitments and verification.

6. Mathematical Description

This section provides a formal description of the principal cryptographic operations defined in UDIF. Each operation is represented with an overview of its function in the protocol, the corresponding API calls, the entities to which it applies, a mathematical description of the underlying construction, and a proof of security outlining correctness, integrity, and replay resistance.

The operations documented here cover the essential lifecycle of UDIF devices and objects: enrollment of certificates, establishment of secure tunnels, generation and verification of Anchor Records, application of capability bitmaps, creation and transfer of objects, logging and suspension events, query predicates, treaty assignments, and audit verification. Together, these descriptions define the cryptographic fabric of UDIF and provide a rigorous foundation for both analysis and implementation.

Each subsection is self-contained but interoperates with the others. Certificate enrollment ensures that all entities are cryptographically bound to the Root. Tunnel establishment provides secure, authenticated communication channels. Anchor Records and logs guarantee that every event is tamper-evident and auditable. Capability bitmaps enforce least-privilege access by cryptographic means. Object operations bind ownership to signed records. Suspension, revocation, and treaty events regulate authority and cross-domain interaction. Queries and proofs provide minimal disclosure while ensuring verifiable responses. Audit verification ties the entire system together, enabling independent confirmation of correctness at every level.

In the following subsections, each operation is described using standard mathematical notation and cryptographic shorthand. The structure follows the convention: *Overview, API, Applies to, Mathematical Description, and Proof of Security*. This ensures that every operation can be implemented, reasoned about, and formally verified within the UDIF framework.

6.1 Certificate Enrollment

Overview:

Certificate enrollment is the foundational process by which a subordinate entity is admitted into the UDIF hierarchy. Enrollment occurs at every level: Root to Branch, Branch to Group Controller, and Group Controller to User Agent. The subordinate first generates its key pairs (signature and KEM transport keys), constructs a certificate signing request (CSR), and submits it to its parent over a temporary secure channel. The parent verifies the CSR, ensures the suite identifier and requested capabilities conform to policy, and then signs the subordinate certificate with its private key. The signed certificate is returned, committed to both the parent's membership log and the subordinate's local log, and becomes the basis for subsequent operations including tunnel establishment and anchoring.

API:

- udif_cert_enroll_request()
- udif_cert_enroll_response()

- udif_cert_verify()

Applies to:

- Root
- Branch Controllers (BC)
- Group Controllers (GC)
- User Agents (UA)

Mathematical Description:

Let:

- $C_D\sigma$ be the certificate of device D, signed by parent authority.
- CSR_D be the certificate signing request generated by device *D*.
- K_{pri} be the private signing key of the parent.
- K_{pub} be the verification key of the parent.
- σ be the asymmetric signature on the child's certificate.
- H be the hash function (SHA3-256 or cSHAKE-256).
- *st* be the sequence number and valid-time timestamp.
- Sign be the asymmetric signing function.
- Verify be the signature verification function.

The subordinate generates a CSR and transmits:

$CSR_D = (ID_D \parallel K_{pubsigD} \parallel K_{pubkemD} \parallel cap_D \parallel st)$

where:

- ID_D = serial identifier of device *D*
- $K_{pubsigD}$ = public key for signatures
- $K_{pubkemD}$ = public key for KEM transport
- cap_D = requested capability bitmap
- *st* = timestamp and sequence

The parent constructs the certificate:

$C_D\sigma = Sign_{Kpri}(H(CSR_D))$

The enrollment exchange is:

$Enroll\_Request(D \rightarrow Parent) = CSR_D$

$Enroll\_Response(Parent \rightarrow D) = C_D\sigma$

Devices receiving a certificate verify:

$Verify_{Kpub}(C_D\sigma) = H(CSR_D)$

If verification succeeds, the certificate is added to the device's local store, the membership log is updated with ($ID_D \parallel C_D\sigma \parallel st$), and the digest is prepared for inclusion in the next Anchor Records.

Proof of Security

Correctness: The certificate is correctly signed by the parent authority. Verification succeeds if and only if the signature σ was produced using the matching private key of the parent.

$\sigma = Sign_{Kpri}(H(CSR_D))$

$Verify_{Kpub}(\sigma) = H(CSR_D)$

Integrity: Since $H(CSR_D)$ is hashed and signed, any alteration of the CSR or certificate will cause verification to fail. The use of SHA3 ensures collision resistance, preventing forgery.

Replay Protection: The inclusion of sequence number and timestamp (*st*) in the CSR prevents reuse of old requests or responses. Parents reject CSRs or certificates whose timestamps fall outside the validity window.

Containment: The capability bitmap cap_D is enforced at the parent and embedded in the signed certificate, ensuring that a subordinate cannot obtain unauthorized rights.

6.2 Secure Tunnel Establishment

Overview:

Secure tunnel establishment provides confidentiality, authenticity, and replay protection for communications between UDIF entities. Once a subordinate has been enrolled and issued a certificate, it must establish a long-term session with its parent authority. This session is realized as a post-quantum secure tunnel constructed from a Key Encapsulation Mechanism (KEM) and authenticated encryption with associated data (AEAD). The subordinate and parent exchange KEM messages to derive a shared secret, expand this secret using a domain-separated cSHAKE function, and initialize the RCS cipher for session protection. Sequence numbers and valid-time timestamps provide replay resistance, while signatures on the certificates bind the tunnel endpoints to the UDIF hierarchy.

API:

- udif_tunnel_init_request()
- udif_tunnel_init_response()
- udif_tunnel_verify()

Applies to:

- Root
- Branch Controllers (BC)

- Group Controllers (GC)
- User Agents (UA)

Mathematical Description:

Let:

- C_D^{σ} be the root signed certificate of device D.
- ct be the asymmetric cipher-text.
- Encap be the asymmetric encapsulation function.
- Decap be the asymmetric decapsulation function.
- H be the hash function.
- H_{CS}^{σ} be the signed certificate and timestamp hash.
- H_{ES}^{σ} be the signed asymmetric ciphertext and timestamp hash.
- H_{PS}^{σ} be the signed public cipher key and timestamp hash.
- KGen be the asymmetric cipher key generation function.
- K_{pub} be the asymmetric signature public key.
- K_{pri} be the asymmetric signature private key.
- ss be the asymmetric shared secret.
- pk be the asymmetric cipher public key.
- sk be the asymmetric cipher secret key.
- Sign is the asymmetric signing function.
- ss be the shared secret.
- Verify is the asymmetric verification function.

The requestor sends a tunnel initialization request to the device. The message contains the requestors serialized certificate, and a valid-time timestamp.

Note: Group Controllers do not cache User Agent certificates.

$H_{CS}^{\sigma} = \text{Sign}_{reqtKpri}(H(C_D^{\sigma} \,||\, st))$

$\text{Request}(C_D^{\sigma}) = (C_D^{\sigma} \,||\, H_{CS}^{\sigma})$

The responder verifies the certificates root signature.

$\text{Verify}_{rootKpub}(C_D^{\sigma}) = H(C_D)$

The responder validates the requestors certificate and the valid-time timestamp are then verified using the validated responder's certificate.

$\text{Verify}_{devKpub}(H_{CS}^{\sigma}) = H(C_D^{\sigma} \,||\, st)$

The responder generates a keypair using the asymmetric cipher. It stores the private key, hashes and signs the public cipher key and valid-time timestamp, and sends it to the requestor.

$pk, sk = \text{KGen}(\lambda, r)$

$H_{PS}^{\sigma} = Sign_{respKpri}(H(pk \,||\, st))$

$Response(pk) = (pk \,\|\, H_{PS}^{\sigma})$

The signed public key is sent to the requestor. The signature, hash, and timestamp are verified, and the requestor uses the public key to encapsulate a shared secret.

$Verify_{respKpub}(H_{PS}^{\sigma}) = H(pk \,||\, st)$

$ct = Encap_{pk}(ss)$

The shared secret is retained by the requestor and is the *master fragment key*. The ciphertext is hashed along with the valid-time timestamp, and the hash is signed by the requestors signing key.

$H_{ES}^{\sigma} = Sign_{reqtKpri}(H(ct \,||\, st))$

$Request(ct) = (ct \,\|\, H_{ES}^{\sigma})$

The responder verifies the message hash using the requestors public verification key, then compares the hash against the hashed ciphertext and timestamp.

$Verify_{devKpub}(H_{ES}^{\sigma}) = H(ct \,||\, st)$

If the ciphertext is validated, the ciphertext is decrypted using the responders private cipher key.

$ss = Decap_{sk}(ct)$

Proof of Security

Correctness: The key exchange consists of three steps:

- The requestor sends (in the case of a UA its serialized certificate), a signed hash of the certificate and timestamp to the responder.
- The responder signs a hash of the public cipher key and timestamp and sends it to the requestor.
- The requestor signs a copy of the ciphertext and timestamp and sends it to the responder.

Proof: Given the definition of digital signatures and the message *m*:

$\sigma(H(m \,||\, st)) = Sign_{Kpri}(H(m \,||\, st))$

The verification function computes:

$Verify_{Kpub}(\sigma(H(m \,||\, st))) = H(m \,||\, st)$

Since $Verify_{Kpub}$ is the inverse of $Sign_{Kpri}$, the signature is valid if it was signed by the matching private key. The hash is generated from the message and compared to the signed hash for equality.

Integrity: Since H(m || *st*) is hashed and signed, any change to the certificate or signature would cause the verification to fail. The hash function used (e.g., SHAKE) is collision-resistant, ensuring that an attacker cannot forge C_D or σ(H(m || *st*)).

Replay Protection: A timestamp and sequence number are included in the hash H(m || *ts*) and checked to ensure that broadcasts cannot be reused maliciously.

6.3 Anchor Record Generation and Verification

Overview:

Anchor Records provide the tamper-evident commitments that bind a subordinate's state to its parent in the UDIF hierarchy. Each controller (Group Controller or Branch) periodically computes Merkle roots over its membership log and transaction log. These roots, together with sequence and timestamp values, are concatenated into an Anchor Record and signed with the controller's private key. The Anchor Record is then transmitted to the parent over the secure tunnel. The parent verifies the signature, checks the sequence and valid-time window, and appends the digest of the Anchor Record into its own membership log. This recursive process creates an unbroken chain of accountability from every User Agent up to the Root.

API:

- udif_anchor_generate()
- udif_anchor_verify()
- udif_anchor_commit()

Applies to:

- Branch Controllers (BC)
- Group Controllers (GC)
- Root (as verifier of Branch anchors)

Mathematical Description:

Let:

- ML = membership log entries.
- TL = transaction log entries.
- Merkle() = Merkle root computation over a set of entries.
- RM = Merkle root of membership log.
- RT = Merkle root of transaction log.
- st = sequence number and timestamp.
- ID_D = serial identifier of device *D*.
- K_{priD} = private signing key of device *D*.
- K_{pubD} = verification key of device *D*.
- Sign() = asymmetric signing function.
- Verify() = asymmetric signature verification function.

- H() = hash function (SHA3-256).

Anchor Records Creation:

Each controller computes the Merkle roots of its logs:

RM = Merkle(ML)

RT = Merkle(TL)

It then constructs the Anchor Records payload:

$AR_{payload}$ = (ID_D ‖ RM ‖ RT ‖ *st*)

The controller signs this payload:

ARσ = $Sign_{KpriD}$(H($AR_{payload}$))

The Anchor Record transmitted to the parent is:

Anchor Record($D \rightarrow$ Parent) = ($AR_{payload}$ ‖ ARσ)

Anchor Verification:

Upon receipt, the parent verifies the signature:

$Verify_{KpubD}$(ARσ) = H($AR_{payload}$)

If verification succeeds and *st* is valid (sequence strictly increasing, timestamp within window), the parent logs:

Commit = H($AR_{payload}$ ‖ ARσ)

This commit value is appended to the parent's membership log and incorporated into its next Merkle root.

Proof of Security:

Correctness: The Anchor Record is valid if and only if the signature ARσ was produced with $Kpri_D$. Since the parent holds *KpubD* from the certificate chain, it can confirm authenticity:

$Verify_{KpubD}(Sign_{KpriD}(H(AR_{payload}))) = H(AR_{payload})$

Integrity: The Merkle construction ensures that any change in the membership or transaction logs results in a different root. Since RM and RT are included in the signed payload, no modification of logs can escape detection. Collision resistance of SHA3-256 prevents an adversary from forging logs that map to the same root.

Replay Protection: The Anchor Record includes Parents reject anchors with stale sequence numbers or timestamps outside the acceptable window. This prevents reuse of old anchors.

Non-repudiation: Once signed and committed, an Anchor Record becomes part of both the child's and parent's logs. It cannot be repudiated, as the digital signature binds the child's identity and the commit hash is recorded upstream.

6.4 Capability Bitmap Application

Overview:

Capabilities in UDIF are enforced through fixed-size bitmaps signed into certificates. Each bit corresponds to an allowable action (e.g., query existence, create object, transfer object, issue certificate). Parents delegate rights to subordinates by embedding a capability bitmap into the subordinate's certificate. These bitmaps are enforced cryptographically: they are signed at issuance, verified at use, and evaluated during query or transaction processing. The default posture is deny-all; only bits explicitly set in the certificate grant authority.

API:

- udif_cap_apply()
- udif_cap_verify()
- udif_cap_check()

Applies to:

- Root (issuing capabilities to Branches)
- Branch Controllers (issuing to GCs)
- Group Controllers (issuing to UAs)
- All devices (evaluating capabilities when processing requests)

Mathematical Description:

Let:

- *cap* = capability bitmap of fixed length (64 bits).
- CSR_D = certificate signing request of subordinate *D*.
- CDσ = signed certificate of device *D*.
- K_{priP} = private key of parent authority *P*.
- K_{pubP} = public key of parent authority *P*.
- Sign() = asymmetric signing function.
- Verify() = signature verification function.
- H() = hash function.
- *op* = requested operation code.

Certificate Issuance with Capabilities:

The parent embeds the capability bitmap into the certificate payload:

$Cert_{payload} = (ID_D \parallel K_{pubsigD} \parallel K_{pubkemD} \parallel cap \parallel st)$

$CD\sigma = Sign_{KpriP}(H(Cert_{payload}))$

The subordinate receives:

$Certificate(D) = (Cert_{payload} \parallel CD\sigma)$

Capability Verification at Runtime:

When device D requests operation op, the verifier checks:

1. Certificate validity:

   $VerifyKpub\_P(CD\sigma) = H(Cert_{payload})$

2. Capability check:

   cap[op] = 1 ? allow : deny

If the bit corresponding to operation op is not set, the request is rejected regardless of other attributes.

Proof of Security

Correctness: Capabilities are correctly applied if and only if the bitmap in the subordinate's certificate was signed by the parent. This ensures that the subordinate cannot self-assign capabilities.

Integrity: Since cap is hashed and signed within the certificate payload, any modification of the bitmap invalidates the signature. Collision resistance of SHA3-256 prevents an adversary from finding a second bitmap with the same digest.

Containment: By construction, a subordinate cannot have greater privileges than its parent, because the parent controls the bitmap during issuance. Inheritance is strictly downward.

Replay Protection: Capabilities are bound to sequence and timestamp (st). Reuse of expired or revoked certificates is prevented by checking validity windows and revocation lists during verification.

Non-repudiation: All requests are logged together with the evaluated capability bits. Auditors can confirm that an operation was permitted or denied by reconstructing the bitmap from the certificate.

6.5 Object Creation and Logging

Overview:

Object creation in UDIF establishes the existence, ownership, and provenance of a digital object. A User Agent (UA) composes an object attribute set, signs it with its private key, and submits it to its Group Controller (GC). The GC validates the signature, checks capability permissions, and commits the object to its transaction log. Both UA and GC hold cryptographic evidence of creation, and the object becomes part of the Anchor Record chain.

API:

- udif_object_create()
- udif_object_log()
- udif_object_verify()

Applies to:

- User Agents (UA)
- Group Controllers (GC)
- Branch Controllers (as anchor verifiers)

Mathematical Description

Let:

- Attr = object attribute set, encoded in canonical TLV.
- ID_O = object serial identifier (unique per domain).
- H() = hash function (SHA3-256).
- $Kpri_{UA}$, $Kpub_{UA}$ = UA signing key pair.
- $Kpri_{GC}$, $Kpub_{GC}$ = GC signing key pair.
- Sig() = signature function.
- Verify() = signature verification function.
- *st* = sequence number and timestamp.
- TL = transaction log of GC.

Object Creation by UA:

The UA computes the digest of the object attributes and signs it:

$hO = H(ID_O \parallel Attr \parallel st)$

$\sigma UA = Sig_{KpriUA}(hO)$

The object creation request transmitted to the GC is:

$ObjCreateReq = (ID_O \parallel Attr \parallel st \parallel \sigma UA)$

Validation and Logging by GC:

The GC verifies the UA signature:

$\text{Verify}_{KpubUA}(\sigma UA) = hO$

If valid and permitted by capability mask, the GC commits the event to its transaction log:

$\text{LogEntry} = (ID_O \parallel hO \parallel \sigma UA \parallel st)$

$TL = TL \parallel \text{LogEntry}$

The GC then signs the log entry for attestation:

$\sigma GC = \text{Sig}_{KpriGC}(H(\text{LogEntry}))$

and acknowledges the creation to the UA:

$\text{ObjCreateResp} = (\text{LogEntry} \parallel \sigma GC)$

Anchor Inclusion:

At the next cadence, the GC computes the Merkle root of TL, which includes the new object entry, and incorporates this root into its Anchor Records. The Anchor Record is signed and transmitted to the parent Branch, binding the object creation event to the global audit chain.

Proof of Security

Correctness: The UA's signature σUA proves authorship of the object attributes. The GC's signature σGC proves validation and inclusion in the domain logs.

Integrity: The digest hO covers the object serial, attributes, and timestamp. Any alteration invalidates both UA and GC signatures. The collision resistance of SHA3-256 ensures no two different attribute sets can yield the same digest.

Non-repudiation: Both UA and GC signatures are logged. The UA cannot deny ownership, and the GC cannot deny logging. Auditors can verify the Anchor Record to confirm inclusion.

Replay Protection: The sequence number and timestamp (st) bind object creation to a unique moment. Duplicate or replayed object creation requests are rejected by the GC.

6.6 Object Transfer

Overview:

Object transfer in UDIF reassigns ownership of an object from one User Agent (UA_A) to another (UA_B). The operation requires both parties to participate: the sender signs a transfer request, the receiver signs an acceptance, and the Group Controller (GC) validates and logs the event. This bi-modal signing ensures that ownership cannot be reassigned unilaterally. The transfer is then appended to the transaction log and propagated through Anchor Records, preserving accountability and auditability across the hierarchy.

API:

- udif_object_transfer_request()
- udif_object_transfer_accept()
- udif_object_transfer_commit()

Applies to:

- User Agents (UA_A, UA_B)
- Group Controllers (GC)
- Branch Controllers (for Anchor Record validation)

Mathematical Description:

Let:

- ID_O = object serial identifier.
- H() = hash function (SHA3-256).
- K_{priA}, K_{pubA} = signature keypair of sending UA_A.
- K_{priB}, K_{pubB} = signature keypair of receiving UA_B.
- K_{priGC}, K_{pubGC} = signature keypair of Group Controller.
- Sig() = signing function.
- Verify() = verification function.
- *st* = sequence number and timestamp.
- TL = transaction log of GC.

Step 1 – Transfer Request by Sender (UA_A):

The sender computes a digest binding the object and intended recipient:

$hT = H(ID_O \parallel ID_B \parallel st)$

$\sigma A = Sig_{KpriA}(hT)$

UA_A sends:

$TransferReq = (ID_O \parallel ID_B \parallel st \parallel \sigma A)$

Step 2 – Acceptance by Receiver (UA_B):

The receiver validates σA:

$\text{Verify}_{KpubA}(\sigma A) = hT$

If valid, UA_B produces its own acceptance signature:

$\sigma B = \text{Sig}_{KpriB}(hT)$

UA_B sends:

TransferAcc = (TransferReq ‖ σB)

Step 3 – Logging by GC:

The GC validates both signatures:

$\text{Verify}_{KpubA}(\sigma A) = hT$

$\text{Verify}_{KpubB}(\sigma B) = hT$

If correct, the GC appends the transfer entry:

LogEntry = (ID_O ‖ ID_A ‖ ID_B ‖ st ‖ σA ‖ σB)

TL = TL ‖ LogEntry

The GC attests by signing the log entry:

$\sigma GC = \text{Sig}_{KpriGC}(H(\text{LogEntry}))$

GC response:

TransferCommit = (LogEntry ‖ σGC)

Step 4 – Anchor Inclusion:

At the next cadence, the transfer entry is absorbed into the GC's transaction log root, included in its Anchor Records, and forwarded to the parent Branch. The Branch verifies and commits the digest, binding the transfer to the audit chain.

Proof of Security

Correctness: Ownership changes only if both UA_A and UA_B sign the same digest hT. The GC validates both signatures before committing.

Integrity: hT covers the object identifier, recipient ID, and timestamp. Any alteration invalidates both signatures. Collision resistance of SHA3-256 prevents forgery.

Non-repudiation: Both sender and receiver signatures are logged. UA_A cannot deny initiating transfer; UA_B cannot deny acceptance. The GC's signed commit provides additional attestation.

Replay Protection: The timestamp and sequence *st* ensure that an old transfer cannot be replayed. The GC enforces uniqueness of sequence numbers per object.

Containment: The GC enforces capability checks from certificates. Neither UA_A nor UA_B can transfer objects without proper rights signed into their certificates.

6.7 Suspension and Revocation Events

Overview:

Suspension and revocation are administrative actions by which a parent authority restricts or permanently removes a subordinate from the UDIF hierarchy. Suspension is temporary and may be followed by resumption, while revocation is final and invalidates a certificate permanently. Both actions are represented as signed log entries, committed to the parent's membership log, and propagated upward via Anchor Records. These events cascade downward: if a controller is suspended or revoked, all of its subordinate entities are implicitly restricted until governance restores or re-enrolls them.

API:

- udif_suspend_entity()
- udif_resume_entity()
- udif_revoke_entity()

Applies to:

- Root (suspending or revoking Branches)
- Branch Controllers (suspending or revoking GCs)
- Group Controllers (suspending or revoking UAs)

Mathematical Description:

Let:

- ID_X = identifier of the entity being acted upon.
- act = action type (suspend, resume, revoke).
- H() = hash function (SHA3-256).
- K_{priP}, K_{pubP} = signature keypair of parent authority.
- Sig() = signing function.
- Verify() = verification function.
- *st* = sequence number and timestamp.
- ML = membership log of the parent.

Suspension Event:

The parent computes the digest:

$hS = H(ID_X \| act="suspend" \| st)$

$\sigma P = Sig_{KpriP}(hS)$

Suspension record:

$SuspendEntry = (ID_X \| act="suspend" \| st \| \sigma P)$

Committed to parent's membership log:

$ML = ML \| SuspendEntry$

Revocation Event:

Similarly, for revocation:

$hR = H(ID_X \| act="revoke" \| st)$

$\sigma P = Sig_{KpriP}(hR)$

Revocation record:

$RevokeEntry = (ID_X \| act="revoke" \| st \| \sigma P)$

$ML = ML \| RevokeEntry$

Verification:

Subordinates and auditors validate:

$Verify_{KpubP}(\sigma P) = H(ID_X \| act \| st)$

If valid, the action is binding, and all subordinate activity under ID_X is restricted.

Anchor Inclusion:

At the next cadence, the membership log root including suspension or revocation entries is Merkle-computed, signed into an Anchor Record, and transmitted upstream. The parent's parent thereby witnesses and logs the administrative action, extending its visibility to the Root.

Proof of Security

Correctness: The action is valid if and only if signed with the parent's private key. The subordinate and auditors can verify authenticity with the parent's public key.

Integrity: The digest hS or hR covers the target identifier, action type, and timestamp. Any modification causes verification to fail. Collision resistance of SHA3-256 prevents adversaries from creating alternate valid digests.

Replay Protection: The inclusion of *st* prevents re-use of old suspension or revocation messages. Parents and auditors reject entries with stale or duplicate sequence numbers.

Non-repudiation: Once logged and anchored, suspension or revocation events are immutable. The parent cannot deny issuing the action, and the subordinate cannot deny receiving it, as both logs and anchors attest to the event.

Containment: Suspension and revocation cascade downward. Subordinate entities under a suspended or revoked controller lose operational rights until either resumption or re-enrollment. This ensures containment of compromised or non-compliant entities.

6.8 Query/Response Predicates

Overview:

Queries in UDIF are structured as predicate evaluations, designed to minimize disclosure. Instead of exposing raw attributes, a User Agent (UA) or external verifier may ask a Group Controller (GC) or Branch Controller (BC) to prove or deny a specific predicate about an object or identity. Responses are limited to *YES*, *NO*, or *DENY*, optionally accompanied by a compact proof such as a Merkle inclusion path. Queries and responses are signed, logged, and incorporated into Anchor Records to ensure accountability.

API:

- udif_query_predicate()
- udif_query_response()
- udif_query_verify()

Applies to:

- User Agents (UA)
- Group Controllers (GC)
- Branch Controllers (BC)
- Root (as ultimate verifier through anchors)

Mathematical Description:

Let:

- P(x) = predicate over attribute x.
- ID_Q = serial identifier of the querying entity.
- ID_T = serial identifier of the target entity or object.
- H() = hash function (SHA3-256).

- K_{priQ}, K_{pubQ} = signature keypair of querying entity.
- K_{priR}, K_{pubR} = signature keypair of responding authority.
- Sig() = signing function.
- Verify() = verification function.
- *st* = sequence number and timestamp.
- Verdict ∈ {YES, NO, DENY}.
- Proof = optional evidence (Merkle path).

Step 1 – Query Formation:

The querying entity Q constructs a query:

$hQ = H(ID_Q \parallel ID_T \parallel P(x) \parallel st)$

$\sigma Q = Sig_{KpriQ}(hQ)$

$Query = (ID_Q \parallel ID_T \parallel P(x) \parallel st \parallel \sigma Q)$

Step 2 – Response by Controller:

The responding controller R validates Q's signature:

$Verify_{KpubQ}(\sigma Q) = hQ$

If valid, R evaluates the predicate P(x) against its records. It then forms the response:

$Resp_{payload} = (ID_Q \parallel ID_T \parallel P(x) \parallel Verdict \parallel Proof \parallel st)$

$\sigma R = Sig_{KpriR}(H(Resp_{payload}))$

$Response = (Resp_{payload} \parallel \sigma R)$

Step 3 – Verification by Querying Entity:

The querying entity verifies:

$Verify_{KpubR}(\sigma R) = H(Resp_{payload})$

If correct, the verdict and optional proof are accepted.

Step 4 – Logging and Anchoring:

The response is appended to R's transaction log:

LogEntry = (Query ‖ Response)

At the next cadence, this entry is included in the Merkle root and bound into an Anchor Record sent upstream.

Proof of Security

Correctness: The query is authentic if and only if signed with Q's private key. The response is authentic if and only if signed with R's private key. Both can be independently verified with public keys from their certificates.

Integrity: The digests hQ and $H(Resp_{payload})$ cover the query, predicate, target, verdict, and timestamp. Any alteration invalidates signatures. SHA3-256 ensures collision resistance.

Replay Protection: The sequence number and timestamp st prevent old queries or responses from being replayed. Controllers reject queries outside validity windows.

Minimal Disclosure: Responses are limited to a verdict and optional proof. No raw attributes are exposed unless explicitly authorized by the capability bitmap.

Non-repudiation: Both query and response are logged and anchored. The querying entity cannot deny asking, and the controller cannot deny answering.

6.9 Peering Treaty Assignment

Overview:

Peering treaties enable controlled interoperability between independent UDIF domains. A treaty is a bilateral agreement between two Branch Controllers (BC_A and BC_B) that establishes the predicates, capabilities, and data minimization rules governing cross-domain queries and transactions. Treaty assignment proceeds as a certificate-like exchange: each party proposes terms, signs them with its private key, and transmits them over a secure tunnel. Once both sides verify and co-sign, the treaty is logged and anchored in both domains. From that point forward, cross-domain messages must comply with the treaty's capability downgrades and predicate whitelists.

API:

- udif_treaty_propose()
- udif_treaty_accept()
- udif_treaty_commit()

Applies to:

- Branch Controllers (BC_A, BC_B)
- Root (as final auditor of anchored treaties)

Mathematical Description:

Let:

- $Treaty_{payload}$ = treaty terms proposed (capability bitmap, predicate list, validity window).

- H() = hash function (SHA3-256).
- K_{priA}, K_{pubA} = signature keypair of BC_A.
- Kpri_B, Kpub_B = signature keypair of BC_B.
- Sig() = signing function.
- Verify() = verification function.
- *st* = sequence number and timestamp.
- ML_A, ML_B = membership logs of domains *A* and *B*.

Step 1 – Treaty Proposal ($BC_A \rightarrow BC_B$):

BC_A computes a digest of the proposed terms:

$hT_A = H(Treaty_{payload} \parallel st)$

$\sigma A = Sig_{KpriA}(hT_A)$

The proposal sent is:

$TreatyProposal = (Treaty_{payload} \parallel st \parallel \sigma A)$

Step 2 – Treaty Acceptance ($BC_B \rightarrow BC_A$):

BC_B verifies:

$Verify_{KpubA}(\sigma A) = hT_A$

If valid, BC_B co-signs:

$hT_B = H(Treaty_{payload} \parallel st)$

$\sigma B = Sig_{KpriB}(hT_B)$

$TreatyAccept = (TreatyProposal \parallel \sigma B)$

Step 3 – Commit and Logging:

Both BC_A and BC_B append the treaty acceptance to their membership logs:

$LogEntry_A = (Treaty_{payload} \parallel st \parallel \sigma A \parallel \sigma B)$

$ML_A = ML_A \parallel LogEntry_A$

$LogEntryB = (Treaty_{payload} \parallel st \parallel \sigma A \parallel \sigma B)$

$ML_B = ML_B \parallel LogEntry_B$

Step 4 – Anchor Inclusion:

At the next cadence, each BC includes the treaty log entry in its Merkle root, signs it into an Anchor Record, and transmits it upstream. The Root (or parent branch) validates and records the digest, ensuring global visibility of the treaty.

Proof of Security

Correctness: A treaty is valid only if co-signed by both BCs. Verification requires that:

$\text{Verify}_{KpubA}(\sigma A) = H(\text{Treaty}_{payload} \parallel st)$

$\text{Verify}_{KpubB}(\sigma B) = H(\text{Treaty}_{payload} \parallel st)$

Integrity: Any modification of treaty terms changes the digest. Since both signatures cover the identical payload, no party can alter the terms post-facto without invalidating signatures.

Replay Protection: The inclusion of st ensures that treaties cannot be replayed from expired sessions. Sequence enforcement prevents duplicates.

Containment: Capabilities and predicates defined in the treaty are enforced cryptographically as part of cross-domain message validation. No entity can exceed the scope defined in the treaty.

Non-repudiation: Treaties are logged in both domains and anchored upstream. Both parties' signatures ensure that neither can later deny agreement, and the Root maintains proof of existence.

6.10 Audit Verification

Overview:

Audit verification in UDIF provides cryptographic assurance that a claimed event (certificate issuance, object creation, transfer, suspension, or treaty) is present in a controller's log and anchored upstream. Auditors or parent controllers challenge a node by requesting a proof of inclusion for a log entry. The node responds with a Merkle path from the event digest to the log's Merkle root. The verifier then compares the computed root against the value signed in the most recent Anchor Record. This allows independent confirmation of correctness without requiring full log disclosure, preserving both integrity and privacy.

API:

- udif_audit_challenge()
- udif_audit_proof()
- udif_audit_verify()

Applies to:

- Group Controllers (GC)

- Branch Controllers (BC)
- Root (final verifier of anchored state)
- External Auditors (optional)

Mathematical Description:

Let:

- LE = log entry (event).
- *hLE* = H(LE) = digest of log entry.
- *path* = Merkle authentication path from hLE to root.
- Merkle() = Merkle root function.
- RM = Merkle root of membership log.
- RT = Merkle root of transaction log.
- ARσ = signed Anchor Record containing RM, RT, sequence, and timestamp.
- K_{pubD} = public key of device *D*.
- Verify() = signature verification function.

Step 1 – Audit Challenge:

The auditor or parent requests proof of inclusion for a log entry LE identified by its digest *hLE*.

Challenge = (*hLE*, *st*)

Step 2 – Audit Proof Response:

The challenged device returns:

Proof = (LE ‖ *path* ‖ ARσ)

where path authenticates *hLE* to the Merkle root, and ARσ is the signed Anchor Record including that root.

Step 3 – Verification:

1. Verify the Anchor Record signature:

   $\text{Verify}_{KpubD}(\text{AR}\sigma) = \text{H}(\text{RM} \parallel \text{RT} \parallel st)$

2. Recompute the Merkle root using the provided path:

   MerkleVerify(*hLE*, *path*) = RM or RT

3. Compare with the root contained in ARσ.

If the Merkle root matches and the Anchor Record signature is valid, the log entry is confirmed as anchored.

Proof of Security

Correctness: By construction, if *hLE* is included in the log and the Merkle path is correct, recomputation yields the same root committed in the Anchor Record. The signature ensures that the root is authentic.

Integrity: Any modification of the log entry, path, or root breaks the verification. Collision resistance of SHA3-256 prevents forging alternate entries with the same digest.

Replay Protection: The Anchor Record includes a sequence and timestamp. Auditors reject proofs based on stale or replayed anchors.

Non-repudiation: Once a log entry is included in an Anchor Record and signed by the device, it cannot be denied. The signature proves authorship, and the Merkle path proves inclusion.

Privacy: Audit verification requires only the challenged entry and the minimal Merkle path. Other log entries remain undisclosed, preserving confidentiality while still allowing verification.

7. Security Analysis

This section evaluates the Universal Digital Identity Framework (UDIF) against its threat model and shows how each component; certificates, capabilities, registries, logs, anchors, queries, treaties, transport, and hygiene requirements, collectively defends against adversarial behavior. The analysis covers external attackers, compromised nodes, and cross-domain misuse, and demonstrates how UDIF maintains accountability, privacy, and post-quantum resilience.

7.1 Resistance to External Adversaries

7.1.1 Passive Eavesdroppers

Threat: Attacker observes traffic to infer identities, objects, or policies.

Defense:

- All transport is AEAD-encrypted (RCS-256 + KMAC-256).
- Only canonical digests, serials, and bucketed predicates leave domains.
- Responses (YES/NO/DENY) are constant-size and padded, preventing inference by length.
- Timing jitter masks differences in policy evaluation.

Result: Eavesdroppers gain no raw identifiers or values; traffic analysis yields minimal useful information.

7.1.2 Active Interceptors

Threat: Adversary injects, replays, or modifies packets.

Defense:

- Every record has monotonic sequence numbers; any gap or replay is fatal.
- ±60s time windows reject delayed injections.
- AEAD authentication tags prevent tampering.
- Fatal transport errors zeroize state and force fresh sessions.

Result: Modification or replay attempts are immediately detected and terminated.

7.2 Resistance to Compromised Entities

7.2.1 Compromised User Agent (UA)

Threat: Malicious UA attempts to impersonate other users, escalate rights, or alter logs.

Defense:

- UA's certificate chain is signed by GC; it cannot exceed its capabilities.
- All queries flow through GC; no lateral UA↔UA traffic is permitted.
- Registry modifications require bi-modal signatures (sender + receiver) and GC logging.

- GC may suspend or revoke UAs after audit failures.

Result: UA compromise is contained; fraud attempts are evident in logs and anchors.

7.2.2 Compromised Group Controller (GC)

Threat: Malicious GC issues false certificates, manipulates logs, or fabricates queries.

Defense:
- All GC events are included in Anchor Record, verified by parent BC.
- Parent compares Anchor Record sequences; missing or inconsistent anchors trigger suspension.
- Cross-domain queries require treaty signatures; a rogue GC cannot impersonate peers.

Result: GC compromise is detectable by parent through anchor verification; containment is enforced by upstream suspension.

7.2.3 Compromised Branch Controller (BC)

Threat: Malicious BC attempts to shield subordinate logs, manipulate treaties, or forward unauthorized queries.

Defense:
- BC anchors are validated by parent or Root; tampering is evident.
- Treaties require bilateral signatures; unilateral BC action cannot forge agreements.
- Suspension or revocation cascades downward, disabling compromised subtrees.

Result: Compromise is limited to subtree until detection; upstream anchoring ensures auditability.

7.3 Resistance to Cross-Domain Abuse

7.3.1 Treaty Overreach

Threat: Peer attempts to extract unauthorized information across domains.

Defense:
- Treaty scope defined by scope_bitmap.
- Non-authorized queries → DENY response.
- All treaty queries logged and anchored in both domains, enabling cross-verification.

7.3.2 Transitive Abuse

Threat: Peer attempts to forward queries through multiple treaties.

Defense:
- Treaties bind only to explicit domainA/domainB pairs.
- Queries cannot be relayed to third domains; this would fail treaty scope check.

Result: Cross-domain queries are strictly bounded by bilateral agreement; misuse is detectable and auditable.

7.4 Cryptanalytic Resilience

Post-quantum primitives only: No reliance on RSA/ECC.
Compile-time suite lock: No downgrade negotiation possible.
Domain separation labels: Prevent cross-context collisions in hashes and KDFs.
Full-block cSHAKE input: Prevents entropy leakage; ensures nonce/key independence.
Hourly asymmetric ratchets (trunks): Provide forward secrecy and compromise containment.

Result: UDIF is resilient against both classical and large-scale quantum adversaries.

7.5 Side-Channel Resilience

Constant-time primitives: Eliminates timing leaks on keys.
Fixed-length responses: Denies length-based inference.
Jittered response timing: Masks policy differences.
Secure memory functions: Zeroize secrets post-use; prevent residue attacks.
Cache-constant implementations: Avoid secret-dependent table lookups.

Result: Side-channel surfaces are minimized and bounded by strict implementation requirements.

7.6 Auditability and Accountability

Membership Logs: Capture all lifecycle events.
Transaction Logs: Record all object transfers.
Anchor Records: Aggregate logs into verifiable digests, signed and forwarded upstream.
Dual logging: Both parent and child record events for reconciliation.
Immutable roots: Merkle commitments guarantee tamper evidence.

Result: Every action is traceable, non-repudiable, and cryptographically bound to its originator.

7.7 Attack Scenarios and Outcomes

Replay Attack: Rejected due to sequence/time windows.
Rogue UA: Limited by GC's capabilities; easily suspended.
Rogue GC: Detected by parent during anchor validation.
Rogue BC: Cannot forge treaties or bypass upstream anchoring.
Cross-domain exploitation: Blocked by treaty scoping.
Cryptanalytic attack: Mitigated by PQ primitives.
Side-channel timing: Prevented by constant-time rules and response padding.

Summary

The security analysis shows that UDIF achieves its stated goals:

Authenticity and Non-repudiation – Every action is signed, logged, and anchored.
Minimal disclosure – Predicates and bucket queries replace raw attribute exposure.
Tamper evidence – Merkle roots and anchors prevent undetectable log alteration.
Compromise containment – Suspension and revocation mechanisms restrict damage to subtrees.
Cross-domain sovereignty – Treaties create explicit, revocable cooperation without trust sprawl.
Post-quantum durability – All primitives resist quantum adversaries.

UDIF thus provides a **robust, auditable, and privacy-preserving framework** for managing digital identities and objects across federated domains.

8. Deployment Profiles and Defaults

This section provides baseline deployment recommendations and default parameter values for UDIF. While domains may tune these values based on local policy, regulation, or performance constraints, the defaults ensure interoperability, security, and audit consistency across implementations.

8.1 Deployment Profiles

UDIF defines profiles to support different operational environments. Each profile inherits all *Level-Core compliance requirements* and adjusts timers, lifetimes, and cadence parameters.

8.1.1 Profile-A (Authority / Root Deployment)

Intended for Roots and high-level Branch Controllers.
Emphasizes strong audit cadence and long certificate validity.
Typical environment: government roots, global institutions.

8.1.2 Profile-E (Enterprise / Mid-Domain Deployment)

Intended for enterprise-scale Branch Controllers and Group Controllers.
Emphasizes balance of performance and security.
Typical environment: banks, healthcare providers, logistics operators.

8.1.3 Profile-U (User / Edge Deployment)

Intended for UAs and small-scale GCs (e.g., service access devices).
Emphasizes low overhead and simpler session lifecycle.
Typical environment: personal devices, client apps, IoT.

8.2 Default Cryptographic Settings

All profiles must use **post-quantum primitives only**:

- **AEAD:** RCS-256 + KMAC-256.
- **KEM:** Kyber-768 or Classic McEliece-348864 (compile-time choice).
- **Signatures:** Dilithium-3 (ML-DSA) or SPHINCS+-SHAKE-256s.
- **Hash/KDF:** SHA3-256 for digests, cSHAKE-256 for ratchets.
- **Domain separation:** Versioned labels (UDIF:* :V1) in every hash/KDF call.

8.3 Certificate Lifetimes

Root Certificates: 10 years (Profile-A).
Branch Certificates: 5 years (Profile-A/E).
Group Certificates: 2 years (Profile-E/U).
User Certificates: 1 year (Profile-U).
Object Certificates: match object lifecycle or 2 years, whichever is shorter.

Revocation/suspension always supersedes validity. Certificates may be re-issued earlier based on policy.

8.4 Transport Defaults

Session establishment: Mutual-auth PQ handshake; compile-time suite only.
Sequence window: Strictly monotonic; any gap is fatal.
Time window: ±60 seconds; packets outside are rejected.
Keepalive: 120 seconds (Profile-E/U); 300 seconds (Profile-A).
Idle teardown: 2 × keepalive interval.
Asymmetric ratchet cadence (BC ↔ BC trunks): 1 hour ±5 minutes jitter.
Symmetric re-key (UA ↔ GC): On session close.
Max session duration (without ratchet): 12 hours (Profile-E); 24 hours (Profile-A).

8.5 Log and Anchor Cadence

Membership and Transaction Logs: Append on every event; immutable.
Anchor Records:

- Profile-A: every 10 minutes.
- Profile-E: every 30 minutes.
- Profile-U: every 60 minutes (if GC-managed).

Audit horizon: Logs retained at least 180 days. Profiles may enforce longer retention for compliance.

8.6 Registry and Object Management

Registry commits: every object event triggers registry root re-computation and log entry.
Merkle proof depth: dependent on registry size; proofs must be verified against anchored roots.
Object transfer latency: logged and anchored within 1 anchor interval maximum.
Destroyed objects: flagged but never removed from registries; history immutable.

8.7 Capability and Access Control Defaults

Default mask: all permissions = NONE.
UA Capabilities:

- QUERY_EXIST, QUERY_OWNER_BINDING permitted for own objects.
- PROVE_MEMBERSHIP permitted only with GC-signed capability.

GC Capabilities: may forward queries to BCs, but only within treaty-defined scope.
Cross-domain treaties: must explicitly enumerate allowed predicate families.

8.8 Telemetry and Monitoring Defaults

Counters exported: rx_ok, tx_ok, denies, seq_faults, auth_fail, anchors_sent/recv.
Aggregation interval: aligned with Anchor Records cadence.
Privacy: Telemetry must be anonymized and never include raw PII.

8.9 Security Properties of Defaults

Strong PQ baseline: All defaults use ≥256-bit symmetric strength.

Least privilege enforced: Capabilities and masks deny by default.
Audit-ready cadence: Anchors frequent enough for tamper evidence without operational overload.
Session resilience: Hourly ratchets on trunks, re-key on close for UAs, ensuring forward secrecy.
Operational safety: Timeouts and keepalives prevent stale sessions; idle teardown limits resource exhaustion.

Deployment defaults ensure that new UDIF domains start with a secure, auditable configuration without requiring policy design from scratch. Profiles (Authority, Enterprise, User) give context-specific defaults while maintaining a common cryptographic baseline. Domains may tune intervals, lifetimes, and retention based on local needs, but must never weaken below Level-Core requirements.
These defaults embody UDIF's design philosophy: cryptographic accountability, minimal disclosure, strong auditability, and security first.

9. Conclusion

The Universal Digital Identity Framework (UDIF) defines a polymorphic, post-quantum secure, and audit-anchored identity container system. It was designed to address a fundamental gap in existing identity infrastructures: the lack of a universal, cryptographically enforced framework that can represent people, institutions, and objects while ensuring non-repudiation, minimal disclosure, and immutable accountability.
By unifying canonical data modeling, strict access controls, tamper-evident logging, and upstream anchor commitments, UDIF provides a coherent trust hierarchy that is independent of any particular political, regulatory, or technological context. Its agnostic core can be adopted in diverse domains, from finance and supply chains to digital identity and IoT deployments, without altering its underlying primitives.

Key features of UDIF include:

- **Hierarchical trust** through certificates signed by parents, rooted in universally trusted authorities.
- **Capability bitmaps** enforcing least-privilege access, inheritable only in a downward-restricted manner.
- **Canonical TLV/uvarint encoding**, ensuring deterministic serialization and preventing malleability.
- **Post-quantum cryptography only**, with compile-time suite locks to prevent downgrade or negotiation attacks.
- **Merkle-based registries and logs**, with Anchor Records binding state changes into a tamper-evident sequence.
- **Query model built on predicates**, where responses disclose nothing beyond a yes/no/deny verdict and optional proofs.
- **Cross-domain peering treaties** that allow interoperability without sacrificing sovereignty or auditability.
- **Audit horizon and anchor chain procedures** that make misbehavior provable and non-repudiable.
- **Privacy posture and data minimization**, ensuring that entities reveal no more than what is strictly required to prove legitimacy.

Through its appendices, UDIF specifies canonical encoding rules, side-channel requirements, capability registry, and verification procedures, giving implementers and auditors a complete reference framework. Each section ensures clarity, simplicity, and explicit justification, avoiding unnecessary complexity while preserving flexibility for future extensions.
In security analysis, UDIF demonstrates resilience against external adversaries, compromised nodes, and cross-domain abuse. Its reliance on post-quantum primitives future-proofs it against emerging cryptanalytic threats. Its audit and anchoring model ensures that even powerful actors cannot manipulate state without detection.
Ultimately, UDIF's strength lies in its elegance and extensibility. It does not prescribe a single application; instead, it provides a foundational substrate on which higher-level protocols; financial transfer systems, identity networks, compliance tools can be built. By embedding cryptographic accountability and minimal disclosure into the root of its design, UDIF positions itself as a future-ready framework for secure, interoperable, and privacy-respecting digital identity and asset management.

Annex A: Side-Channel and Constant-Time Requirements

This section specifies mandatory *implementation hygiene rules* for UDIF. Even if the cryptographic primitives are post-quantum secure in theory, side-channels timing differences, memory residue, error message leaks—can undermine security in practice. UDIF therefore codifies requirements that make attacks based on physical observation or timing infeasible in compliant implementations.

A.1 Principles

- **Constant-time execution:** All sensitive operations must run in time independent of secret data.
- **Memory zeroization:** Secret material must be erased promptly and reliably once no longer needed.
- **Fixed-form errors:** Responses must not leak information through message size, structure, or timing.
- **Minimal secret exposure:** Secrets must be kept in the smallest possible scope and never copied unnecessarily.
- **Deterministic outputs:** Identical inputs must always produce identical outputs, preventing oracle-style information leaks.

A.2 Constant-Time Cryptography

A.2.1 Required Areas

- **Hash functions:** SHA3-256, SHAKE-256, and cSHAKE-256 must execute in constant-time for all inputs.
- **KDF and ratchets:** cSHAKE block slicing must not branch on secret indices.
- **AEAD:** RCS-256 encryption/decryption and KMAC-256 tag generation must be constant-time with respect to keys and plaintexts.
- **Signatures:** Dilithium/ML-DSA and SPHINCS+ signing and verification routines must not branch on secret coefficients or seeds.
- **KEM:** Kyber and McEliece encapsulation/decapsulation must avoid secret-dependent memory lookups.

A.2.2 Tag Comparisons

All AEAD tag checks and digest comparisons must use constant-time memutils functions (e.g., memutils_compare_ct). Never use ordinary equality operators for secret data.

A.3 Memory Hygiene

A.3.1 Allocation and Erasure

- Allocate secret buffers with functions that guarantee alignment and zero-initialization.
- Copy secrets using secure functions (memutils_copy).
- Erase secrets with secure zeroization (memutils_clear) immediately after use.

A.3.2 Key Lifecycle

- Session keys, KEM shared secrets, signature seeds, and ratchet state must never persist beyond their intended lifetime.
- On ratchet or re-key, old keys must be destroyed before new keys are installed.
- On session close, *all keys and state must be zeroized* before releasing buffers.

A.4 Error and Response Discipline

A.4.1 Fixed Lengths

- YES, NO, and DENY responses must be padded to the same length.
- Error codes are TLV-encoded with fixed field sizes; variable-length context fields must be hashed before inclusion if necessary.

A.4.2 Uniform Timing

- Application-layer responses must return within a fixed time window, optionally with bounded jitter (e.g., ±5 ms) to mask processing differences.
- Transport-layer fatal errors (e.g., SEQ_INVALID) must terminate sessions immediately after zeroization; teardown timing must not vary by error type.

A.4.3 Non-Verbose Diagnostics

- External responses may include only enumerated error codes.
- Internals may log extended diagnostic details, but those must never be exposed outside the node.

A.5 Side-Channel Surfaces

A.5.1 Timing

Implementations must test for and eliminate timing variation dependent on key bits, branch outcomes, or data-dependent memory access.

A.5.2 Power and EM

Where hardware deployments are expected, implementers should adopt masking, blinding, or constant-power techniques to resist power analysis.

A.5.3 Caches

Avoid table-based implementations with key-dependent indices. Use bit-sliced or arithmetic formulations that execute without cache variation.

A.6 Testing and Verification

- **Self-tests:** Include constant-time test vectors where inputs differ but execution time must remain within a tolerance.
- **Code audits:** Ensure no standard library functions (memcmp, strcpy, etc.) are used with secrets.

- **Fuzzing:** Feed malformed TLVs and transport headers; check that all errors resolve into canonical codes without leaks.
- **Side-channel analysis:** For hardware, apply differential power analysis (DPA) and electromagnetic (EMA) tests during certification.

A.7 Security Properties

- **Forward secrecy preserved:** Old secrets zeroized cannot be recovered.
- **Indistinguishability:** Observers cannot tell a DENY from a YES or NO by timing or size.
- **Resilience against hardware attackers:** Cache, power, and EM side-channels are minimized by design requirements.
- **Robust transport closure:** Fatal errors always result in immediate zeroization, preventing leakage through partial states.

Side-channel resistance is often left to implementers, but in UDIF it is core to the specification. The framework assumes adversaries with strong observational power. Without constant-time discipline, minimal disclosure in queries and ratchets can be undermined. By mandating secure memory functions, fixed-length responses, and timing uniformity, UDIF reduces risk from timing, cache, and power side-channels, and enforces uniform security across all compliant implementations.

Annex B: Canonical Encoding Rules (TLV/uvarint C14N)

This appendix is normative. It specifies the exact, byte-level rules for UDIF canonical encoding ("C14N") used for certificates, objects, registries, queries, logs, anchors, and every other serialized structure. The goals are determinism, unambiguous parsing, constant-time implementation, and malleability resistance.

B.1 Canonical Container Model

UDIF uses **Tag–Length–Value (TLV)** containers with *uvarint* encoding for Tag and Length:

- **Tag (T):** unsigned variable-length integer (uvarint) identifying the field.
- **Length (L):** uvarint giving the number of bytes in Value.
- **Value (V):** byte string or an embedded TLV sequence.

A container is a *sequence of TLVs with strictly ascending tag order*.

Invariants (MUST):

1. Tags within a container are strictly *ascending*; equal or decreasing tags are invalid.
2. A tag appears *at most once* unless that field is explicitly declared "repeated". Repeated fields MUST be *contiguous* and in ascending order of any sub-key they define.
3. Lengths use the *shortest possible* uvarint form (no leading zero bytes, no longer-than-necessary encodings).
4. Containers MUST NOT contain padding, NUL terminators, or out-of-band metadata.
5. All serialization is *big-endian* where fixed-length integers are used (rare; UDIF prefers uvarints).

B.2 Uvarint Encoding

A uvarint encodes an unsigned integer in 7-bit groups, high bit as continuation:

- **Encoding:** For value x, emit bytes b0..bn where each bi stores 7 bits of x, low to high, and bit 7 set to 1 for all but the last byte (whose bit 7 is 0).
- **Minimal Form:** The most-significant emitted 7-bit group MUST be non-zero (unless the entire value is zero). Values 0..127 encode in one byte.

Forbidden forms (MUST reject):

- Encodings with extra leading zero groups.
- Encodings longer than necessary for the represented value.

B.3 Field Kinds and Canonical Representations

UDIF defines these *canonical value kinds*. Each is encoded as *one TLV* unless otherwise noted.

B.3.1 Unsigned Integer

- **Value:** uvarint of the integer.

- **Notes:** Prefer integers only where ranges are small or counters exist. Large numbers should be domain-specific digests.

B.3.2 Signed Integer

Value: *Zig-Zag* transform to map signed → unsigned, then uvarint.
ZZ(n) = (n << 1) ^ (n >> 63) for 64-bit signed input.

B.3.3 Boolean

- **Value:** single byte: 0x00 = false, 0x01 = true.
- **Length:** MUST be 1.

B.3.4 Binary Blob

- **Value:** raw bytes.
- **Length:** exact byte count.
- **Use:** digests, public keys, signatures, serials.

B.3.5 String (UTF-8)

- **Value:** UTF-8 bytes after *NFKC normalization* and validation (no ill-formed sequences).
- **Length:** exact byte count post normalization.
- **Security:** Normalization MUST occur **before** hashing/signing to avoid equivocation.

B.3.6 Timestamp

- **Value:** uvarint seconds since Unix epoch (UTC).
- **Windowing:** Window checks are semantic (not encoding rules).

B.3.7 Nested Container

- **Value:** a complete, canonical TLV sequence (sub-container).
- **Length:** total bytes of nested TLVs.
- **Rule:** Inner containers obey the **same** ordering, uniqueness, and minimal uvarint rules.

B.3.8 Map (String → Bytes) - Optional Pattern

When a schema needs associative pairs, use a repeated nested TLV with fixed inner tags:

- **Outer:** multiple TLVs with the same outer Tag (declared "repeated").
- **Inner (per entry):**

  1: key (string, UTF-8 canonical)
  2: value (binary or nested TLV)

- **Order:** Entries sorted by **lexicographic order of raw key bytes** after normalization.
- **Contiguity:** All map entries for that Tag appear **contiguously**.

B.3.9 Repeated Fields

If a schema marks a field as repeated, encode each element as a separate TLV with the *same Tag*, placed *contiguously*, and internally ordered by the rule the schema mandates (e.g., lexicographic by element digest). If no order is defined, order by the canonical byte string of each element.

B.4 Container Ordering and Uniqueness

Per container:

- Tags MUST strictly increase: t1 < t2 < ... < tn, except for "repeated" fields, where multiple TLVs with the same tag appear contiguously (and together occupy a single position in the global order).
- Within a repeated run: the *element ordering* MUST follow the schema's defined canonical order (commonly lexicographic on element digests or keys).

Decoder MUST reject:

- Out-of-order tags.
- Multiple distinct runs of the same repeated tag (non-contiguous repeats).
- Duplicate single-occurrence tags.

B.5 Defaults and Omitted Fields

- Fields with a specified *default value* MUST be *omitted* when set to that default.
- A missing optional field is semantically identical to the default value.
- Encoders MUST NOT emit a field with a default value; decoders MUST treat omitted as default.

B.6 Canonical Digest and Signature Inputs

UDIF digests and signatures are always computed over the *exact canonical bytes* of the relevant TLV sequence.

- **Digest function:** SHA3-256 unless a structure explicitly lists another.
- **Domain separation:** Prepend a label before hashing to avoid cross-context collision: Example: SHA3-256(DS || TLV_bytes) where DS is an ASCII label like "UDIF:OBJ-DIGEST:V1".
- **Merkle leaves:** leaf_hash = SHA3-256(DS_leaf || TLV_bytes).
- **Signatures:** Sign the **to-be-signed (TBS)** TLV (e.g., certificate fields 1..9) without the signature field itself.
- **KMAC tags (capabilities):** tag = KMAC-256(issuer_key, digest) with a capability-specific label fixed in the profile.

MUST NOT:

- Re-encode or normalize between digest and signature steps.
- Mix non-canonical encodings in any cryptographic operation.

B.7 Error Handling (Decoding)

A canonical decoder MUST enforce the following and signal *fatal decode error* on violation:

1. **Uvarint minimality** (no non-minimal encodings).
2. **Tag monotonicity** and *uniqueness* per container.
3. **Length bounds:** L must not exceed available bytes; nested decoding stops exactly at the declared length.
4. **Type conformity:** Field value must match the schema's type (e.g., boolean length=1).
5. **String validity:** UTF-8 well-formedness and NFKC normalization pass.
6. **Default elision:** If a field is present with a default value, *reject* (encoders must omit).
7. **Repeated field rules:** Single contiguous run per repeated field, intra-run canonical order.

On error, implementations MUST:

- Return the canonical *decode error code* (UDIF_ERR_DECODE), without exposing internal diagnostics externally.
- Not attempt recovery by skipping TLVs; fail closed.

B.8 Deterministic Examples

B.8.1 Simple Container

Logical fields:
- 1: suite_id = 0x11
- 2: role = 3 (GC)
- 3: valid_from = 1700000000
- 4: valid_to = 1800000000

Bytes (hex):
```
01 01 11          // tag=1, len=1, val=0x11
02 01 03          // tag=2, len=1, val=0x03
03 04 80 F0 FA 00 // tag=3, len=4, u32(1700000000)
04 04 6B 49 D2 00 // tag=4, len=4, u32(1800000000)
```
Digest (example): SHA3-256("UDIF:EXAMPLE:V1" || bytes) = a3 72 … b9 9f.

B.8.2 Nested Container

Outer:
- 1: serial (16 bytes)
- 2: validity (nested)
- 3: pubkey (binary)

Inner (validity container):
- 1: valid_from (uvarint)
- 2: valid_to (uvarint)

Order:

- Outer tags ascending: 1 < 2 < 3.
- Inner tags ascending: 1 < 2.
- Length of tag 2 equals total bytes of inner TLVs.

B.8.3 Repeated Field (Map Entries)

Suppose Tag 7 is a repeated map entry (key, value).

Encode entries:
07 <len> 01 <len> key_utf8 02 <len> value_bytes
07 <len> 01 <len> key_utf8 02 <len> value_bytes

All Tag 7 instances are contiguous and sorted by key bytes.

B.9 Canonical Ordering Rules (Summary Table)

- **Per container:** Tags strictly ascending; one tag per field unless repeated.
- **Repeated fields:** One contiguous run; elements internally ordered by schema rule (default: lexicographic by canonical element bytes).
- **Strings:** UTF-8 + NFKC before encode; store exact normalized bytes.
- **Integers:** uvarint minimal; signed via zig-zag then uvarint.
- **Timestamps:** uvarint seconds (UTC).
- **Booleans:** 1 byte (00 or 01) only.
- **Defaults:** Omit on encode; treat absent as default on decode.
- **No padding:** Any attempt to add padding is non-canonical and rejected.

B.10 Conformance Checklist (Encoder)

1. Start container with last_tag = 0.
2. For each field in ascending Tag:
    - If value equals default: **omit**.
    - Encode value to canonical bytes.
    - Compute L as exact byte count; encode T,L,V.
    - Update last_tag = T.
3. For repeated fields:
    - Sort elements canonically before emission.
    - Emit each as its own TLV with the same Tag, *contiguously*.
4. For nested containers:
    - Encode inner container canonically to a temporary buffer; use its exact bytes as V (no re-serialization).

B.11 Conformance Checklist (Decoder)

1. Initialize last_tag = 0.
2. While bytes remain:

    - Decode T (uvarint, minimal) and ensure T > last_tag unless within a single repeated run.
    - Decode L (uvarint, minimal); ensure sufficient remaining bytes.
    - Slice V as L bytes; *do not* read beyond.
    - If field is repeated:
        - Ensure all repeats are *contiguous*.
        - Verify intra-run canonical element order.
    - Validate type constraints (e.g., boolean length).
    - If value equals the default and field is optional: *reject* (encoders must omit).
    - Update last_tag appropriately (at the end of a repeated run).
3. If nested container:
    - Recursively apply the same rules to the V slice.

B.12 Security Rationale

- **Determinism and Malleability:** Strict tag order + minimal uvarints + no padding → byte-stable encodings, preventing signature malleability and canonicalization attacks.
- **Simplicity:** One encoding for all structures; identical parser/serializer paths reduce bugs and attack surface.
- **Constant-Time:** Minimal branching and fixed rules enable constant-time implementations (e.g., CT comparisons for digests/tags).
- **Auditability:** Identical bytes across implementations ensure that digests, Merkle leaves, and signatures verify universally.

B.13 Interop Notes and Common Pitfalls

- **Do not** emit textual numbers or platform-native endianness; always use uvarints or canonical blobs.
- **Do not** include optional fields when set to defaults.
- **Do not** rely on JSON/CBOR for canonical inputs; they are non-deterministic without extra profiles.
- **Do** pre-normalize strings to NFKC and validate UTF-8.
- **Do** keep repeated elements contiguous and sorted.

B.14 Minimal Test Vectors (Sanity)

- **Uvarint Minimality:**
    - 0x00 → 00
    - 0x7F → 7F
    - 0x80 → 80 01 (invalid encodings like 00 80 01 MUST be rejected)
- **Boolean:**
    - true → TLV T=1, L=1, V=01
    - false → TLV T=1, L=1, V=00
    - L≠1 MUST be rejected.
- **Ordering:**
    - {T=1}{T=3} valid; {T=3}{T=1} invalid.
    - Repeated {T=5}[a][b] contiguous; {T=5}[a]{T=6}{T=5}[b] invalid.

- **Defaults:**
    - If field enabled default is false, TLV with V=00 MUST NOT be emitted; decoder rejects.

B.15 ABNF-Like Summary (Informative)

```
container   = *( field / repeated-field )

field = tlv
repeated-field = 1 * (tlv);          same Tag, contiguous, sorted by element order
tlv = tag length value
tag = uvarint-min
length = uvarint-min
value = *OCTET;           exact 'length' octets

uvarint-min = %x00;    zero
        / %x01-7F;         7-bit values
        / 1*%x80-FF %x00-7F;  multi-byte, last byte < 0x80, no leading zero groups
```

Note: ABNF above is illustrative; enforcement of minimal encodings and ordering is done by algorithmic checks.

B.16 Backward/Forward Compatibility

- New fields MUST use new Tag numbers; old decoders *ignore unknown tags* after enforcing ordering rules (unknown tags still respect ascending order).
- Removing a field mandates treating it as *optional with default*; encoders stop emitting it, decoders accept its absence.
- Changing a field's type or semantics requires a *new tag*; never overload an existing tag with incompatible meaning.

Annex C: Capability Bitmap Registry

This appendix is *normative*. It defines the canonical *capability bitmap* for UDIF v1. Capabilities are *fine-grained rights* expressed as *bit positions* within a fixed 64-bit field (8 bytes). They enforce **default-deny access**: no bit is set unless explicitly granted by a parent certificate or capability token.
Each bit position has a *precise, non-overlapping meaning*. This registry ensures that all implementations interpret capability bits identically.

C.1 Canonical Rules

1. **Size:** Capability bitmaps are always 64 bits (8 bytes).
2. **Endian:** Encoded in TLV as a fixed 8-byte binary field, *big-endian*.
3. **Default:** All zeros (0x00..00) = no capabilities.
4. **Inheritance:** Child bitmap must be ≤ parent's bitmap (bitwise subset).
5. **Revocation:** Bits may be removed at any time; addition requires re-issuance or new capability token.
6. **Audit:** Capability digests are included in certificates and logged in Membership Events.

C.2 Bitmap Layout (v1)

Bit	Mnemonic	Scope	Description
0	QUERY_EXIST	Local/Domain/Treaty	Can issue "existence" queries (yes/no on UA or Object).
1	QUERY_OWNER_BINDING	Local/Domain/Treaty	Can query whether an Object is owned by a specific UA.
2	QUERY_ATTR_BUCKET	Local/Domain/Treaty	Can query attribute buckets (status=active, destroyed, etc).
3	PROVE_MEMBERSHIP	Local/Domain/Treaty	Can request Merkle membership proofs for Objects/registries.
4	FORWARD_QUERY	Domain/Treaty	GC/BC may forward queries upstream or across treaty peers.
5	ADMIN_ENROLL	Domain only	May enroll new UAs or subordinate branches.
6	ADMIN_SUSPEND	Domain only	May suspend UAs or branches pending audit.
7	ADMIN_RESUME	Domain only	May resume suspended UAs or branches.
8	ADMIN_REVOKE	Domain only	May revoke certificates (UA/branch).
9	ADMIN_BRANCH_CREATE	Domain only	May instantiate a new subordinate branch or group.
10	ADMIN_BRANCH_RETIRE	Domain only	May retire or prune a branch permanently.
11	REGISTRY_COMMIT	Local/Domain	May commit updated registry roots (UAs for own registries, GCs for group).

12	TX_CREATE	Local/Domain	May originate a transaction event (object creation, transfer, update).
13	TX_ACCEPT	Local/Domain	May co-sign and accept incoming transaction transfers.
14	LOG_ANCHOR_SEND	Domain only	May generate Anchor Records and send upstream.
15	LOG_ANCHOR_VERIFY	Domain only	May verify and append child Anchor Records.
16	TREATY_NEGOTIATE	Domain only	May negotiate and sign a Peering Treaty.
17	TREATY_QUERY_EXEC	Treaty only	May process treaty queries within allowed predicate families.
18	TREATY_QUERY_ORIGIN	Treaty only	May originate treaty queries to a peer domain.
19	TELEMETRY_EXPORT	Domain only	May export telemetry counters in Anchor Records.
20	ERROR_REPORT	Local/Domain	May issue signed error events into logs.
21–31	RESERVED_FUTURE_CORE	—	Reserved for core v1 extensions. Must be zero.
32–63	RESERVED_PROFILE	—	Reserved for profile-specific or jurisdictional use (privacy, audit extensions, ZKP).

C.3 Profiles and Scopes

- **Local:** UA capabilities apply only to their own registry and objects.
- **Domain:** GC/BC capabilities govern children in their branch or group.
- **Treaty:** Capabilities apply only across domains with a valid treaty.

Profiles (Core, Audit-Enhanced, Privacy-Maximized, ZK-Ready) may define stricter subset masks or assign meaning to Reserved_Profile bits, but must never overload core bits 0–20.

C.4 Encoding Example

Example bitmap granting a UA the ability to:

- Query existence (bit 0),
- Query ownership binding (bit 1),
- Accept transactions (bit 13).

Bitmap (little-endian for display):
0000 0000 0000 0000 0000 0010 0000 0011
Hex (big-endian 8-byte TLV value):
00 00 00 00 00 00 20 03

C.5 Digest and Logging

Capability digests are computed as:
cap_digest = SHA3-256("UDIF:CAP-DIGEST:V1" || suite_id || holder_serial || bitmap || valid_to)

- Digest ensures a capability cannot be silently altered.
- Included in certificates and Membership Events (CAP_GRANT, CAP_REVOKE).
- Parents may verify children's capabilities by recomputing digest and comparing against signed record.

C.6 Rationale

- **Bitmap form**: Compact, fast to verify, and extensible by reserving future bits.
- **Domain separation**: Ensures capabilities cannot be replayed across contexts.
- **Strict hierarchy**: Child rights never exceed parent's; revocation is explicit.
- **Reserved ranges**: Prevent accidental overlap and allow predictable growth.

Annex D: Anchor Chain Verification Procedure

This appendix is normative. It defines the procedure for verifying the Anchor Chain in UDIF, ensuring that Membership Logs, Transaction Logs, and Anchor Records remain consistent and tamper-evident across the hierarchy. Anchor verification is the mechanism by which parents keep children honest, and auditors can prove that no branch has silently altered its logs.

D.1 Concepts

- **Anchor Record:** A signed, periodic digest issued by a child (UA/GC/BC) to its parent. Contains regroot, txroot, mroot, counters, and sequence number.
- **Anchor Chain:** The ordered sequence of Anchor Records from a given child, monotonically increasing in seq.
- **Parent Verification:** The parent BC/Root must validate each Anchor Record upon receipt and append its digest to its own Membership Log.
- **Audit:** External auditors can independently recompute Merkle roots from Membership and Transaction Logs and compare with anchored values.

D.2 Verification Invariants

1. **Monotonic Sequence:**
    a. anchor.seq must be exactly expected_next_seq = last_seq + 1.
    b. Any jump, rollback, or duplicate sequence number is a fatal error.
2. **Time Window:**
    a. anchor.time_utc must be within ±Δ (default Δ = 300s) of parent's clock.
    b. Prevents replay of old anchors or artificially delayed anchors.
3. **Digest Consistency:**
    a. regroot, txroot, and mroot must be valid SHA3-256 Merkle roots over the child's logs at that epoch.
    b. Parent may request challenge proofs to recompute roots if suspicion arises.
4. **Signature Authenticity:**
    a. Anchor must be signed by child's valid certificate key.
    b. Signature must cover fields: (child_serial, seq, time, regroot, txroot, mroot, counters?).
5. **Parent Commitment:**
    a. Parent must log the Anchor Record digest in its own Membership Log.
    b. Digest = SHA3-256("UDIF:ANCHOR:V1" || anchor_TLV).
6. **Dual Logging:**
    a. Both child and parent log the Anchor Record.
    b. Discrepancy (child claims anchor sent, parent claims not received) is detectable by auditors.

D.3 Verification Procedure (Parent Algorithm)

Given input: anchor (from child), expected_next_seq, now_utc.

1. Verify child certificate chain is valid (not expired, not revoked).
2. Check suite_id matches domain suite.

3. Ensure anchor.seq == expected_next_seq.
4. Ensure |anchor.time_utc - now_utc| ≤ Δ.
5. Verify signature(anchor, child_pubkey).
6. Recompute digest = SHA3-256("UDIF:ANCHOR:V1" || canonical_TLV(anchor[fields 1..7])).
7. Append digest to parent's Membership Log (event: LOG_ANCHOR_VERIFY).
8. Update expected_next_seq = anchor.seq.
9. Optionally request Merkle proofs of logs if suspicion triggered (e.g., counters anomaly).
10. If any check fails → suspend child, log UDIF_ERR_AUTH_FAILURE or UDIF_ERR_SEQ_INVALID.

Output: Parent either accepts and logs anchor, or suspends child and raises fatal audit event.

D.4 Auditor Verification

External auditors verifying a domain must:

1. Fetch Anchor Records from each child in chronological order.
2. Recompute Merkle roots from raw Membership/Transaction Logs.
3. Confirm each recomputed root matches the value in the anchor.
4. Verify sequence continuity and timestamps.
5. Cross-check parent's logged digest against child's anchor digest.
6. If mismatches are found → child or parent misbehavior is provable.

D.5 Error Conditions

- **Sequence Break:** If anchor.seq skips or repeats, parent issues SEQ_INVALID fatal error and suspends child.
- **Time Drift:** If time_utc is outside window, parent rejects and logs TIME_WINDOW_EXCEEDED.
- **Signature Fail:** If signature invalid → AUTH_FAILURE, suspension.
- **Root Mismatch:** If challenge re-computation differs → REGISTRY_STALE or PROOF_NOT_AVAILABLE logged, followed by suspension.

D.6 Security Properties

- **Tamper Evidence:** Any log rewriting or omission results in root mismatch at next anchor.
- **Non-repudiation:** Child's signature + parent's log commit prove anchor issuance.
- **Containment:** Misbehavior at a child stops at its subtree; parent can suspend.
- **Auditability:** External auditors can detect divergence between child and parent anchors.
- **Default-Deny:** If anchor verification fails, child loses ability to forward queries or anchor logs.

The anchor chain procedure guarantees that every state transition in UDIF is witnessed upstream. Even if a child colludes to rewrite its logs, the inconsistency will surface at the parent or under audit. This layered commitment structure is the cornerstone of UDIF's tamper-evident hierarchy.

Annex E: Reserved Tags and Field Numbers

This appendix is **normative**. It defines the canonical allocation of TLV tags across UDIF structures. By fixing tags, UDIF ensures deterministic encoding, cross-implementation compatibility, and audit stability. Tags are integers encoded as minimal uvarints; every field in a TLV must use its assigned tag number.

E.1 Allocation Principles

1. **Global Uniqueness per Container:** Within each structure (Certificate, Query, Log, Anchor, etc.), each field has a unique tag number.
2. **Ascending Order:** Encoders MUST emit fields in strictly ascending tag order.
3. **Contiguous Assignment:** Tags start at 1 and increment without gaps unless reserved.
4. **Reserved Ranges:** Each structure reserves tags for future use and profile extensions.
5. **Stability:** Once assigned, tag meanings never change. Deprecation requires retiring the tag and marking it "OBSOLETE".

E.2 Certificates (UA, GC, BC, Root, Object)

Tag	Field	Type	Notes
1	suite_id	uvarint	Must equal compile-time suite ID.
2	role	uvarint	ROLE_ROOT=1, BRANCH=2, GC=3, UA=4, OBJECT=5.
3	serial	bytes[16]	Unique per entity.
4	issuer_serial	bytes[16]	Zero for root.
5	valid_from	uvarint	UTC seconds since epoch.
6	valid_to	uvarint	UTC seconds since epoch.
7	pubkey	bytes[]	Suite-dependent length.
8	policy_epoch	uvarint	Policy profile version.
9	cap_bitmap	bytes[8]	64-bit canonical capability bitmap.
10	signature	bytes[]	Parent's signature (covers tags 1–9).
11–15	RESERVED	—	For extensions.

E.3 Capability Tokens

Tag	Field	Type	Notes
1	verbs_bitmap	uvarint	Verb rights (QUERY, TX, ADMIN, etc.).
2	scope_bitmap	uvarint	Scope rights (LOCAL, DOMAIN, TREATY).
3	issued_to	bytes[16]	Holder serial.
4	issued_by	bytes[16]	Issuer serial.
5	valid_to	uvarint	Expiry time.
6	digest	bytes[32]	SHA3-256 over 1–5.
7	tag	bytes[32]	KMAC over digest.
8–15	RESERVED	—	Extensions.

E.4 Queries

Tag	Field	Type	Notes
1	query_id	bytes[16]	Random nonce / UUID16.
2	type_code	uvarint	Q_EXIS=1, Q_OWN=2, Q_ATTR=3, Q_PROOF=4.
3	target_serial	bytes[16]	UA or Object serial.
4	predicate	TLV	Sub-container (type-specific).
5	time_anchor	uvarint	Optional epoch/time binding.
6	capability_ref	bytes[32]	Digest of capability used.
7–15	RESERVED	—	For future predicates.

E.5 Responses

Tag	Field	Type	Notes
1	query_id	bytes[16]	Echo from query.
2	verdict	uvarint	0=NO, 1=YES, 2=DENY.
3	proof	TLV	Optional Merkle path.
4	time_reply	uvarint	UTC seconds since epoch.
5	signature	bytes[]	Responder signature.
6–15	RESERVED	—	Extensions.

E.6 Membership Events

Tag	Field	Type	Notes
1	event_code	uvarint	ME_ENROLL=1, SUSPEND=2, RESUME=3, REVOKE=4, CAP_GRANT=5, CAP_REVOKE=6, REGISTRY_COMMIT=7, BRANCH_CREATE=8, BRANCH_SUSPEND=9, BRANCH_REVOKE=10.
2	subject_serial	bytes[16]	UA/GC/BC serial.
3	parent_serial	bytes[16]	Issuer serial.
4	time	uvarint	UTC timestamp.
5	data	bytes[]	Context (registry root, etc.).
6	signature	bytes[]	Parent signature.
7–15	RESERVED	—	Extensions.

E.7 Transaction Events

Tag	Field	Type	Notes
1	tx_id	bytes[32]	SHA3-256 digest.
2	object_serial	bytes[32]	Object identifier.
3	parties	TLV	Sub-container (seller, buyer).
4	time	uvarint	UTC timestamp.
5	object_digest	bytes[32]	Object state digest.
6	owner_sig	bytes[]	Owner signature.

7	recip_sig	bytes[]	Receiver signature (if required).
8–15	RESERVED	—	Extensions.

E.8 Anchor Records

Tag	Field	Type	Notes
1	child_serial	bytes[16]	Serial of child branch/GC.
2	seq	uvarint	Sequence number (monotonic).
3	time	uvarint	UTC timestamp.
4	regroot	bytes[32]	Registry Merkle root.
5	txroot	bytes[32]	Transaction log root.
6	mroot	bytes[32]	Membership log root.
7	counters	TLV	Optional telemetry.
8	signature	bytes[]	Child signature.
9–15	RESERVED	—	Extensions.

E.9 Error Events

Tag	Field	Type	Notes
1	error_code	uvarint	Canonical error taxonomy.
2	context_id	bytes[16]	Query ID or TX ID.
3	subject_serial	bytes[16]	Entity serial.
4	time	uvarint	UTC timestamp.
5	signature	bytes[]	GC/BC signature.
6–15	RESERVED	—	Extensions.

E.10 Reserved Global Tags

- **0:** Forbidden (never used).
- **1–15:** Core fields (per-structure assignments above).
- **16–31:** Reserved for future UDIF-Core.
- **32–63:** Reserved for profile/jurisdictional extensions.
- **64+:** MUST NOT be used in UDIF v1.

E.11 Rationale

- Fixed tags guarantee canonical byte sequences → stable digests and signatures.
- Reserved ranges allow evolution without ambiguity.
- Common tag ranges across structures reduce parser complexity.
- Deterministic, ascending tags eliminate malleability and signature-equivalence attacks.

Annex F: Policy Hooks (Non-Normative)

This appendix is informative. It outlines optional "policy hooks" that can be overlaid onto UDIF without changing the core canonical framework. Hooks allow jurisdictions, industries, or organizations to introduce additional rules, filters, or requirements at runtime. Importantly, hooks operate outside the canonical encoding: they influence how nodes make decisions, but they do not alter TLV formats, certificate layouts, or log structures.

F.1 Purpose of Policy Hooks

- **Flexibility:** Adapt UDIF to diverse regulatory and operational environments.
- **Containment:** Keep extensions isolated from core, preserving interoperability.
- **Auditability:** Allow policy decisions to be logged and explained without breaking canonical encodings.
- **Minimal disclosure:** Hooks enforce stricter predicates or checks, but never loosen default-deny rules.

F.2 Common Policy Hook Categories

F.2.1 Predicate Catalog Extensions

- **Example:** A financial regulator may define custom predicates such as *"is not on AML watchlist vX"* or *"account balance ≥ threshold"*.
- **Hook Mechanism:** Extend the query interpreter with additional predicate IDs, bound to reserved tag space (per Appendix E).
- **Audit Impact:** Any non-core predicate must be flagged with a policy_epoch value, ensuring auditors know which catalog was active.

F.2.2 Jurisdictional Restrictions

- **Example:** EU data may not leave EU-based UDCs.
- **Hook Mechanism:** Treaty evaluators enforce *capability downgrading*—certain bits are masked out depending on jurisdiction.
- **Audit Impact:** Anchored counters can include "downgraded treaty queries" for later review.

F.2.3 Attribute Buckets

- **Example:** Health sector might define attributes such as *"vaccination_status = complete/partial/none"*.
- **Hook Mechanism:** Registries extend their bucket catalog, but proofs still commit only to Merkle digests.
- **Audit Impact:** Anchored registry commits include the epoch of the attribute schema.

F.2.4 Audit/Retention Controls

- **Example:** Jurisdictions requiring 7-year retention vs. others requiring strict deletion at 1 year.
- **Hook Mechanism:** Profile parameters alter log retention policies; root anchors embed the retention profile ID.

F.2.5 Consent and Delegation

- **Example:** A user may delegate query rights to a service provider with additional conditions.
- **Hook Mechanism:** A "delegation artifact" is appended to capability tokens as an optional TLV extension.
- **Audit Impact:** Delegations are logged as CAP_GRANT events with delegation reference digests.

F.2.3 Hook Enforcement Points

- **At GC:** Policy hooks most often enforce per-query rules (capability checks, jurisdiction checks, predicate catalogs).
- **At BC:** Hooks influence cross-domain treaties, log anchoring cadence, and retention policies.
- **At UA:** Hooks can restrict which queries are even constructed (e.g., "UA may never ask direct attribute values").

F.2.4 Hook Logging and Audit

- Every applied policy hook must be logged in **Membership Logs** as a special **POLICY_REF** event.
- Logs store a digest of the active policy schema (e.g., SHA3-256("Policy v2025.1 JSON")).
- Anchor Records include the policy_epoch tag so auditors can verify which policy set applied during that interval.

F.2.5 Security Considerations

- Hooks **cannot** override or weaken core invariants (default-deny, canonical encoding, cryptographic signatures).
- Hooks **may** restrict or add conditions to capabilities, queries, or logs.
- Hooks are non-normative; different domains may run different policy regimes.
- Interoperability is preserved because hooks operate at the decision layer, not the serialization layer.

F.2.6 Example Scenarios

- **Financial corridor:** Add FATF "Travel Rule" checks as required predicates.
- **Healthcare domain:** Add strict privacy hooks (deny treaty queries across domains).

- **IoT deployment:** Add bandwidth-saving hook (suppress Merkle proofs unless explicitly requested).
- **Cross-border treaty:** Add jurisdiction filter hook (capability downgrade for data minimization).

F.3 Rationale

UDIF aims to be a polymorphic framework: the canonical rules are minimal and fixed, while hooks allow adaptation to diverse contexts. By isolating hooks as non-normative overlays, the protocol stays clean, verifiable, and interoperable, while still enabling policy-driven flexibility where regulators or organizations demand it.

Annex G: Glossary

This glossary defines key terms used throughout the UDIF specification. It is **normative for terminology**, ensuring consistent usage across domains, implementations, and profiles.

Access Mask
A bitfield or bitmap that specifies which operations an entity is allowed to perform on objects or registries. Always enforced under **default-deny**: unset bits mean no rights.

AEAD (Authenticated Encryption with Associated Data)
A cryptographic construction that simultaneously encrypts and authenticates data. In UDIF, AEAD (RCS-256 + KMAC-256) protects all transport records and ensures that headers (AAD) are authenticated even if not encrypted.

Anchor Chain
The ordered sequence of **Anchor Records** issued by a child to its parent. Serves as tamper-evident proof of log consistency and chronological state progression.

Anchor Record
A signed, periodic digest issued by a child (UA/GC/BC) summarizing registry, transaction, and membership log roots plus telemetry counters. Verified and logged by the parent to enforce accountability.

Audit Horizon
The minimum period for which Membership and Transaction Logs must be retained (default: 180 days). Ensures auditors can replay log history to confirm consistency with Anchor Records.

Capability
A cryptographically bound token defining rights (verbs) and scope (local, domain, treaty). Capabilities are always signed or KMAC-tagged by a parent authority.

Capability Bitmap
A fixed 64-bit field where each bit corresponds to a specific capability (e.g., QUERY_EXIST, TX_ACCEPT, BRANCH_CREATE).

Certificate
A signed structure binding a serial identifier to a role (Root, Branch, GC, UA, Object), a public key, validity window, and capability bitmap. Certificates form the trust hierarchy in UDIF.

Child / Parent
Relationship in the UDIF hierarchy. A **child** (UA/GC/BC) is directly controlled by its **parent** (GC/BC/Root). Parents sign children's certificates, enforce capabilities, and verify anchors.

Default-Deny
The fundamental UDIF principle: no entity has rights by default. All capabilities and permissions must be explicitly granted and cryptographically bound.

Digest
The result of hashing canonical TLV bytes with SHA3-256 plus a domain-separation label. Used for object IDs, registry roots, transaction IDs, and anchors.

Domain
A logical tree rooted in a Branch Controller, containing subordinate Group Controllers and User Agents. Domains can form **treaties** to exchange queries cross-domain.

Domain Separation
The practice of labeling every cryptographic hash/KDF invocation with a context string (e.g., "UDIF:OBJ-DIGEST:V1"). Prevents cross-protocol or cross-structure collisions.

Group Controller (GC)
A branch node that manages a group of User Agents (UAs). GCs issue UA certificates, mediate queries, log transactions, and produce Anchor Records for their parent Branch Controller.

Log (Membership, Transaction)
- **Membership Log:** Records governance events (enroll, suspend, revoke, cap grant).
- **Transaction Log:** Records object events (create, transfer, update).

Both are append-only and committed to upstream via Anchor Records.

Merkle Root
The root hash of a binary Merkle tree built from sorted digests of leaves (objects, log events). Serves as compact tamper-evidence for registries and logs.

Object
A digital container representing a commodity, asset, or record. Identified by an **object serial** and committed by its attribute Merkle root. Owned by a User Agent.

Predicate
A boolean condition tested in a query (e.g., “Does this object exist?”, “Is this user the owner?”). UDIF queries return only **yes/no/deny** verdicts, with optional proofs.

Query
A canonical TLV request issued by a UA or GC, asking a parent to prove a predicate about an identity, object, or registry. Always signed or capability-bound.

Registry
The container of all objects belonging to a UA or group. Each update recomputes a registry Merkle root, which is logged and committed in Anchor Records.

Root (Root Authority)
The top-level certificate authority in a UDIF hierarchy. Issues certificates to first-level Branch Controllers. Considered universally trusted within its jurisdiction.

Scope
The context in which a capability may be exercised: **LOCAL** (UA on own objects), **DOMAIN** (intra-branch), or **TREATY** (cross-domain under a peering agreement).

Serial
A 16-byte unique identifier assigned to every certificate, UA, GC, BC, and object. Always referenced in TLV form, never plaintext names.

TLV (Tag–Length–Value)
Canonical serialization format used throughout UDIF. Tags and lengths are uvarints; values are raw bytes or nested TLVs. All TLVs must follow ascending tag order.

Treaty
A bilateral agreement between domains allowing limited cross-domain queries. Governed by **treaty capabilities** and logged for audit.

User Agent (UA)
A leaf entity representing an end-user or device. UAs own objects, issue queries via their GC, and may hold multiple group memberships.

Verifier
Any entity (parent, auditor, treaty peer) that checks signatures, digests, Merkle roots, or proofs.

Verdict
The outcome of a query: **YES**, **NO**, or **DENY**. Responses are constant-length and signed by the responding GC/BC.