AMATELUS Protocol Spec

Frourio, Inc.

October 21, 2025

Chapter 1

Nomenclature and Scope

1.1 Terminology

This document uses two complementary names for the same underlying technology:

1.1.1 AMT Protocol (Technical Name)

AMT (Autonomous Meta-Trust) is the *technical name* for the protocol, designed for international standardization. This is the name used in technical discussions, standards bodies, and peer-reviewed literature.

1.1.2 AMATELUS (Marketing Name)

AMATELUS is the *marketing name*, indicating "Japanese origin." This is used for brand identity, commercial deployment, and public communication in Japan.

1.1.3 Relationship

- **AMT Protocol** = international-standard-oriented technical specification
- **AMATELUS** = Japan-specific implementation and deployment brand
- Both refer to the same underlying cryptographic authentication mechanism

1.2 Document Scope

This blueprint covers:

- The AMT Protocol (technical foundation)
- The did:amt DID Method Specification (standardized identifier format)
- AMATELUS implementation guidance (Japan-specific deployment)
- Cryptographic foundations and formal proofs

Chapter 2

Introduction

2.1 AMATELUS Protocol Overview

Definition 1. The AMATELUS protocol is a cryptographic authentication mechanism integrating:

- Decentralized Identifiers (DIDs)
- Verifiable Credentials (VCs)
- Zero-Knowledge Proofs (ZKPs)

2.1.1 What AMATELUS Provides

Proposition 2. *AMATELUS delivers the following cryptographic capabilities:*

- *Public key infrastructure (PKI) based distributed identifiers*
- *Zero-knowledge proof-based attribute ownership verification*
- *Verifiable credentials for claim issuance and storage*
- *Cryptographic trust guarantees*

2.1.2 What AMATELUS Does NOT Provide

Proposition 3. *AMATELUS* does not *provide:*

- *Centralized directory services (DID resolution is local-only)*
- *Mediation or relay infrastructure*
- *User authorization decisions*
- *Communication endpoint management*
- *Message delivery guarantees*

2.1.3 Design Philosophy

Proposition 4. *AMATELUS is a* cryptographic authentication mechanism*, not a communication infrastructure.*

Theorem 5. *The boundary between cryptographic verification (AMATELUS responsibility) and service provision (service provider responsibility) eliminates distributed infrastructure complexity while preserving cryptographic security properties.*

Chapter 3

did:amt Method Specification

3.1 Abstract

Definition 6. The `did:amt` method is a Decentralized Identifier (DID) that is algorithmically generated and resolved without reliance on any external Verifiable Data Registry (VDR) such as a blockchain. This method is designed for high-stakes environments (public administration, government) where data integrity and operational robustness are paramount.

3.2 did:amt Syntax

The `did:amt` syntax conforms to the W3C DID Core specification:

```
did-amt              := "did:amt:" method-specific-id
method-specific-id   := crockford-base32-encoded-sha3-512-hash
```

The `method-specific-id` is a Crockford's Base32 encoded string of the hash value generated through the local creation process.

3.2.1 Crockford's Base32 Character Set

`0123456789ABCDEFGHJKMNPQRSTVWXYZ`

This character set is chosen to minimize human transcription errors in administrative settings (e.g., avoiding confusion between `O` and `0`, or `I` and `l`).

3.3 CRUD Operations

3.3.1 Create: Local Generation

Theorem 7. *A `did:amt` identifier is generated locally on the owner's device without network registration.*

The generation process consists of:

1. *Generate Ed25519 key pair*
2. *Prepare information pair: AMT Version Number + Public Key*

3. *Select DID Document template corresponding to AMT version*

4. *Derive DID through local cryptographic operations*

3.3.2 Read: Local Resolution

Theorem 8. *The resolution of a* `did:amt` *is completed locally by a verifier.*

No external Verifiable Data Registry (blockchain, centralized service) or network calls are required. Verifier receives the `[AMT Version Number, Public Key]` *pair from the owner and executes the same local derivation steps for verification.*

3.3.3 Update: Not Supported

As `did:amt` DID Documents are immutable, Update operations are not supported. Key rotation is handled by issuing a new DID and linking it via a "DID Continuity Verifiable Credential" issued by a trusted third party.

3.3.4 Deactivate: Key Destruction

There is no explicit Deactivate operation. Deactivation is effectively achieved by destroying the associated private key.

3.4 Cryptographic Properties

3.4.1 DID Identifier Security

- **Hash function**: SHA3-512 (post-quantum collision resistance)
- **Security level**: 256-bit (quantum-resistant)
- **Uniqueness**: Permanent uniqueness guaranteed by collision resistance

3.4.2 DID Ownership Proof

- **Signature algorithm**: Ed25519 (current version)
- **Security level**: 128-bit (classical only, see future evolution)
- **Signature verification**: Public key validation against DID Document

3.4.3 Privacy

The avoidance of a Verifiable Data Registry (VDR) ensures that:

- DIDs are not publicly enumerable
- No central authority records DID creation
- High degree of privacy maintained

3.4.4 Operational Robustness

Crockford's Base32 encoding minimizes human transcription errors during manual entry in administrative processes.

3.5 Future Evolution: PQC Transition

3.5.1 Versioning for Cryptographic Agility

The AMT protocol is designed with "cryptographic agility," allowing for the upgrade of its cryptographic suite through versioning.

3.5.2 Foreseeable Changes (AMT v1+)

Post-Quantum Signature Migration

The most critical change will be migration from Ed25519 to a NIST-selected PQC signature algorithm (e.g., CRYSTALS-Dilithium). This ensures DID ownership proof is also secure against quantum computers.

Binary Data Format Challenge

PQC signature algorithms require significantly larger public key and signature sizes (several to tens of kilobytes). Future versions will likely specify a binary representation format such as CBOR (Concise Binary Object Representation) for DID Documents to maintain efficiency.

Interoperability Through Versioning

The `AMT Version Number` presented by the owner allows verifiers to accurately determine:

- Which cryptographic algorithms to use (Ed25519 vs PQC)
- Which data formats to expect (JSON-LD vs CBOR)
- Secure interoperability during transition periods

3.5.3 Example: Version 0 DID Document

```
{
  "@context": ["https://www.w3.org/ns/did/v1"],
  "id": "did:amt:0V3R4T7K1Q2P3N4M5J6H7G8F5D4C3B2A...",
  "verificationMethod": [{
    "id": "did:amt:0V3R4T7K1Q2P3N4M5J6H7G8F5D4C3B2A...#key-1",
    "type": "Ed25519VerificationKey2020",
    "controller": "did:amt:0V3R4T7K1Q2P3N4M5J6H7G8F5D4C3B2A...",
    "publicKeyMultibase": "k3t635r7r1c0kdf41n2p5h3t2d3n2g5r..."
  }],
  "authentication": ["#key-1"],
  "assertionMethod": ["#key-1"]
}
```

Chapter 4

Deployment Models

Definition 9. This chapter covers deployment models aspects of AMATELUS.

4.1 Service-Driven Model (Primary)

In the predominant deployment model:

1. Service provider application (bank, government, SNS) initiates
2. Wallet is invoked from service app via OS-level Intent/deeplink
3. Wallet generates DIDComm message containing:
   - Communication DID (ephemeral or persistent based on service)
   - DID Document (public key only)
   - VC/ZKP proving required attributes
4. DIDComm message sent to service provider (HTTPS, typically)
5. Service provider performs:
   - Cryptographic verification of ZKP/VC
   - Authorization decision
   - Service execution

Trust origin: Service provider (centralized, established).
Endpoint lifecycle: Service-scoped (session-based).

4.2 Physical Proximity Model (Supplementary)

For in-person scenarios:

1. Bluetooth Low Energy (BLE) provides discovery via physical proximity
2. Public services (municipal offices, event gates)
3. Session completes within local network

4. No persistent endpoint contracts

Examples:

- Age verification at municipal counter
- Facility access control
- In-person credential verification

Chapter 5

Technical Specifications

Definition 10. This chapter covers technical specifications aspects of AMATELUS.

5.1 DID Document Design Principles

The AMT protocol employs did:amt for distributed identifier generation and resolution. The DID Document structure is minimal and stateless:

```
DID := did:amt:H(DIDDoc)

DIDDoc := {
  id: DID,
  publicKey: PublicKey,
  metadata: Metadata
}
```

where H is SHA3-512 (collision-resistant hash function).

5.1.1 Critical Design Decision: Abolition of serviceEndpoint

Traditional DID specifications include `serviceEndpoint` in DIDDocuments. The AMT protocol rejects this for principled reasons:

Problem with serviceEndpoint

- **Short lifetime**: Service endpoint contracts last months to few years
- **Individual burden**: Users must manage contract renewals and provider changes
- **Unrealistic assumption**: Assumes individual can maintain stable, contracted endpoints
- **Real-world failure**: When provider fails or service terminates, users cannot migrate

AMT Solution

Remove `serviceEndpoint` from DIDDocument entirely.
Instead:

- DID lifetime: Long-term (public-key-based, years to decades)
- Endpoint lifetime: Session-scoped (minutes to hours)
- Endpoint management: Service provider responsibility only
- Individual burden: Zero

Implementation Reality

- Wallet is invoked by service app (already knows its own endpoint)
- DIDDocument contains only public key for cryptographic verification
- Endpoint information implicit in application context
- No distributed endpoint discovery needed

For complete did:amt specification details, see Chapter 2 (did:amt Method Specification).

5.2 DIDComm Integration

AMATELUS employs DIDComm Messaging v2.1 as the sole communication protocol.

5.2.1 Message Structure

```
DIDCommMessageSend := {
  senderDID: ValidDID,
  senderDoc: Option<ValidDIDDocument>,
  vcs: List<ValidVC>,
  zkp: Option<ValidZKP>
}
```

5.2.2 Security Properties

- **ECDH-1PU authenticated encryption**: Sender identity authenticated exclusively to recipient
- **Message-level security**: Independent of transport layer
- **Transport agnosticism**: Works over HTTPS, BLE, WebSocket, etc.
- **Sender anonymity (optional)**: Anoncrypt layer available if needed

5.3 Verifiable Credentials

5.3.1 VC Structure

```
VC := {
  issuer: DID_issuer,
  subject: DID_subject,
  claims: Claims,
  signature: Signature,
  credentialStatus: RevocationInfo,
  deLinkageInfo: Option<DeLinkageInfo>
}
```

5.3.2 Delinkage Information

To prevent cross-service correlation:

```
DeLinkageInfo := {
  identityDID: DID,        // Long-term identity DID
  communicationDID: DID    // Session-specific DID
}
```

ZKP proves ownership of both DIDs without revealing either to external parties.

5.4 Zero-Knowledge Proofs

5.4.1 Security Guarantees

AMATELUS provides cryptographic proof generation and verification. Impersonation attacks are prevented through DIDComm's requirement for explicit public key transmission.

Theorem 11. *Impersonation attacks (attacker with different secret key) are cryptographically prevented through DIDComm.*

Mechanism:

- *Verifier receives ZKP along with sender's public key via DIDComm (`senderDoc`)*
- *ZKP must be cryptographically signed with secret key corresponding to this public key*
- *If attacker uses different secret key, the signature verification fails*
- *Attacker cannot reuse legitimate ZKP with different SK (cryptographically impossible)*

***Security guarantee**: Impersonation attacks are prevented by DIDComm alone.*

Theorem 12. *DIDComm establishes cryptographic certainty between ZKP and secret key correspondence.*

By requiring explicit transmission of DIDDocument (containing public key), the protocol ensures:

- *Verifier definitively knows which public key corresponds to the ZKP*
- *ZKP is bound to exactly one secret key (the corresponding private key)*
- *Any ZKP signed with different secret key will fail verification*

***Result**: Cryptographic correspondence is certain, making impersonation impossible.*

5.4.2 Replay Prevention (Application Layer Responsibility)

Important: Replay prevention (preventing legitimate users from reusing the same ZKP across multiple sessions) is NOT part of AMATELUS protocol. This is an application-layer responsibility.

Services that require single-use ZKP semantics (e.g., one-time registrations, authorization grants) should:

- Implement nonce-based mechanisms in their application layer
- Generate unique session nonces for each verification request
- Record and verify nonce freshness (checking that the nonce has not been used before)
- Maintain nonce history within their application database

Services where ZKP reuse is acceptable (e.g., age verification for each transaction) do not require nonce mechanisms.

Important: AMATELUS does not specify or enforce nonce handling. This is entirely the responsibility of the application implementing the protocol.

5.4.3 Computational Efficiency

- **Offline precomputation**: Heavy circuit evaluation (minutes to hours)
- **Real-time verification**: Light operations only (milliseconds to hundreds of milliseconds)
- **UX compatibility**: User interaction completes within tolerance (3 seconds)

Chapter 6

Trust Architecture

Definition 13. This chapter covers trust architecture aspects of AMATELUS.

6.1 Responsibility Boundaries

Component	AMATELUS	Service Provider
Public Key Infrastructure	Yes	–
DID generation	Yes	–
VC issuance/validation (1-layer)	Yes	Policy
ZKP generation/verification	Yes	–
Endpoint management	–	Yes
Message delivery	–	Yes
Authorization decisions	–	Yes
Communication security (TLS)	–	Yes

6.2 Trust Origin

- **Cryptographic trust**: Originating from AMATELUS protocol
- **Operational trust**: Originating from service provider (centralized)
- **Authorization trust**: Originating from service provider (centralized)

The separation prevents AMATELUS from assuming responsibilities it cannot scale to manage globally.

6.3 One-Layer Trust Limitation

AMATELUS validates only 1-layer VC chains:

- **0-layer**: Direct issuance from trusted anchor
- **1-layer**: Delegated issuance (trustee validated against anchor)
- **2+ layers**: Explicitly *not* validated by AMATELUS protocol

This prevents:

- Delegation chain attacks
- Circular credential verification
- Unbounded revocation propagation

Chapter 7

Cryptographic Foundations

7.1 Security Assumptions

Definition 14. AMATELUS relies on the following cryptographic security assumptions:

- **Collision-resistant hash**: SHA3-512 provides 128-bit security against quantum adversaries
- **Unforgeable signatures**: Dilithium2 provides 128-bit security against quantum adversaries
- **ZKP soundness**: Standard zero-knowledge properties (completeness, soundness, zero-knowledge)

7.2 Threat Model and Mitigations

7.2.1 Impersonation Attack (Different Secret Key)

Theorem 15. *Impersonation attacks with different secret keys are cryptographically prevented.*

If an attacker uses a different secret key to forge a ZKP, the signature verification will fail because DIDComm makes the sender's public key known to the recipient.

7.2.2 Replay Attack (Same ZKP, Same User)

Proposition 16. *Replay attack prevention (preventing legitimate users from reusing the same ZKP) is NOT a responsibility of AMATELUS protocol. This is an application-layer responsibility.*

***Important distinction**:*

- ***Impersonation attacks** (attacker with different secret key): PREVENTED by AMATELUS through DIDComm*
- ***Replay attacks** (legitimate user reusing same ZKP): NOT handled by AMATELUS. Applications requiring single-use semantics must implement nonce mechanisms.*

Applications that require ZKP single-use guarantees should implement nonce handling at the application layer:

- *Generate unique session nonces for each verification*
- *Record and verify nonce freshness (checking that the nonce has not been used before)*
- *Maintain nonce history in application database*

Applications where ZKP reuse is acceptable (e.g., age verification, permission checks, one-time access grants) do not require additional replay prevention mechanisms.

7.2.3 Man-in-the-Middle Attack

Proposition 17. *Man-in-the-Middle attacks are mitigated at the transport layer.*

While AMATELUS provides cryptographic identity verification, ECDH-1PU authenticated encryption and TLS/HTTPS are the responsibility of service providers.

7.2.4 Sybil Attack (Multiple DIDs)

Proposition 18. *Multiple DID possession is intentional protocol design for privacy protection.*

While a single entity can control multiple DIDs, Anonymous Hash Identifiers (AHI) restrict per-audit-domain abuse through cryptographic binding to national identity systems.

Chapter 8

Privacy Architecture

Definition 19. This chapter covers privacy architecture aspects of AMATELUS.

8.1 Anti-Linkability Across Services

Multiple DIDs enable cross-service unlinkability:

```
For all DID_1, DID_2, Service_1, Service_2:
  (Service_1 != Service_2) AND
  Link(DID_1, DID_2) requires 2^128 quantum operations
```

8.2 Zero-Knowledge Property

ZKP reveal attribute ownership without revealing identity:

- **Public**: Attribute claimed (age ≥ 18)
- **Hidden**: Identity proving the attribute
- **Hidden**: Secret key generating the proof
- **Verified**: ZKP authenticity via public key

8.2.1 Privacy Preservation Under DIDComm

A critical design question: How can DIDComm's requirement for explicit public key transmission (for impersonation prevention) coexist with privacy protection?

Theorem 20. *Privacy is maintained despite public key transmission via DIDComm due to:*

- ***Ephemeral Communication DIDs**: Each session uses distinct DIDs with different key pairs*
- ***Different services, different keys**: User generates new keys for each service*
- ***Cryptographic unlinkability**: Public keys from different sessions cannot be linked (requires 2^{128} quantum operations to link, assuming quantum-resistant hash functions)*

- ***Service-specific communication DIDs**: Verifier knows only the communication DID, not the user's persistent identity DID*

***Result**: While DIDComm requires public key disclosure for impersonation prevention, the use of ephemeral DIDs with distinct key pairs per service ensures cross-service unlinkability and privacy.*

8.2.2 Application-Layer Privacy Considerations

Applications using AMATELUS should consider additional privacy measures:

- **Session management**: Create new communication DIDs for each session to prevent cross-session linkability
- **Retention policies**: Do not retain communication DIDs or ZKPs longer than necessary
- **Logging**: Minimize logging of ZKPs and DIDDocuments; log only what is necessary for application function
- **Nonce handling**: If implementing nonce mechanisms (application responsibility), handle nonce values with privacy in mind

8.3 Deniable Authentication

For privacy-sensitive scenarios, anonymous encryption (Anoncrypt) available:

- Sender identity hidden from intermediaries
- Recipient verifies proof authenticity (still authenticated)
- Sender maintains plausible deniability

Chapter 9

Implementation Guidance

Definition 21. This chapter covers implementation guidance aspects of AMATELUS.

9.1 Wallet Implementation Requirements

9.1.1 Secret Key Management

- Store private keys in secure enclave (iOS/Android hardware)
- Never export unencrypted private keys
- Implement key derivation from user-memorizable seeds (BIP39 compatible)

9.1.2 DID Lifecycle

Identity DIDs

- Generated once per credential lifetime (typically 1-10 years)
- Persisted across wallet updates and device changes
- Associated with formal credentials (passport, license, etc.)

Communication DIDs

- Generated per service engagement or per session (configurable)
- Destroyed if VC not issued (one-time verification)
- Persisted if VC issued (login reuse case)
- Linked to counterparty DID in wallet for session continuation

9.1.3 VC Storage

- Store issued VCs with `deLinkageInfo`
- Enable offline VC verification

- Implement selective disclosure (claim subset presentation)
- Track VC issuance context (service, timestamp, nonce)

9.2 Service Provider Integration

9.2.1 DIDComm Endpoint Setup

1. Publish service endpoint URL (HTTPS with valid TLS)
2. Accept POST requests with DIDComm messages
3. Parse and cryptographically verify ZKP
4. Execute service logic based on verified claims
5. Return VC (if registration service) or result (if verification-only)

9.2.2 Authorization Decision Points

Services must decide:

- Which claims are required for service access
- Which DIDs are trusted for VC validation
- How to handle multiple DIDs from same user (user linking policy)
- Whether to issue persistent VC or one-time verification

9.2.3 Nonce Management (Optional Service Feature)

If replay prevention beyond DIDComm is required:

1. Generate unique nonce per session
2. Send nonce to wallet
3. Verify returned ZKP includes nonce in cryptographic binding
4. Record used nonces to prevent reuse
5. Clean up nonce history based on session TTL

Note: Nonce management is *optional* at service layer, not protocol-required.

9.3 Deployment Scenarios

9.3.1 One-Time Verification (Age Confirmation)

- Holder generates ephemeral communication DID
- Generates ZKP proving age ≥ 18
- Verifier validates ZKP, permits or denies access
- Communication DID destroyed (no VC issued)
- No persistent state

9.3.2 Service Registration (Account Opening)

- Holder generates communication DID (persistent)
- Generates ZKP proving identity claims
- Issuer validates, creates account
- Issuer generates VC (e.g., account holder credential)
- Holder stores VC and communication DID-to-service association
- Later: Login reuses same communication DID

9.3.3 Continuous Authentication (SNS, Messaging)

- Holder generates ephemeral communication DID per login
- Generates ZKP proving phone number ownership
- Service validates, establishes session
- Session token replaces ZKP for subsequent requests
- Communication DID may be stored for optional reconnection hint

Chapter 10

Formally Verifiable JSON Schema Subset

Definition 22. This chapter covers formally verifiable json schema subset aspects of AMATELUS.

10.1 Abstract

The AMATELUS protocol employs a formally verifiable subset of JSON Schema 2020-12 for Verifiable Credentials validation. This subset is specifically designed to enable formal verification in Lean 4 without axioms or partial definitions.

10.2 Design Principles

The JSON Schema subset is built on these principles:

- **Formal Verification**: All features MUST be provably terminating in Lean 4
- **Protocol Guarantee**: Only features in this subset are guaranteed by AMATELUS protocol
- **Implementation Freedom**: Wallets MAY support full JSON Schema 2020-12 beyond protocol guarantees
- **Deterministic Validation**: Same schema produces identical results across all implementations
- **Future Evolution**: As Lean capability increases, subset expands toward full JSON Schema 2020-12

10.3 Formalization Strategy

10.3.1 Why a Subset?

Full JSON Schema 2020-12 includes features that prevent formal verification:

- **Circular references** (via `$ref`, `$defs`): Cause non-termination
- **Dynamic property validation** (`additionalProperties`, `patternProperties`): Require unbounded iteration
- **Infinite nesting**: Composition keywords can nest arbitrarily
- **Format annotations**: Depend on external specifications

The subset eliminates these features to enable formal verification.

10.3.2 Lean Formalization Structure

```
inductive SchemaKeyword
  | type : List String -> SchemaKeyword
  | maxLength : Nat -> SchemaKeyword
  | minLength : Nat -> SchemaKeyword
  | pattern : String -> SchemaKeyword
  | maximum : Float -> SchemaKeyword
  | minimum : Float -> SchemaKeyword
  | multipleOf : Float -> SchemaKeyword
  | maxItems : Nat -> SchemaKeyword
  | minItems : Nat -> SchemaKeyword
  | uniqueItems : Bool -> SchemaKeyword
  | items : Schema -> SchemaKeyword
  | maxProperties : Nat -> SchemaKeyword
  | minProperties : Nat -> SchemaKeyword
  | required : List String -> SchemaKeyword
  | properties : List (String × Schema) -> SchemaKeyword
  | enum : List JSONValue -> SchemaKeyword
  | const : JSONValue -> SchemaKeyword
  | allOf : List Schema -> SchemaKeyword
  | anyOf : List Schema -> SchemaKeyword
  | oneOf : List Schema -> SchemaKeyword
  | not : Schema -> SchemaKeyword
  | title : String -> SchemaKeyword
  | description : String -> SchemaKeyword
```

Note: Excluded keywords like `$ref`, `additionalProperties`, `format` are not in the inductive type.

10.3.3 Termination Guarantee

All supported features are *provably terminating*:

1. **No circular references**: `$ref` excluded entirely
2. **Bounded recursion**: Composition nesting limited to 3 levels
3. **Finite structure**: All schemas have finite size

10.4 Supported Features

The following JSON Schema keywords are **GUARANTEED** by AMATELUS protocol:

10.4.1 Type System

- `type`: String or array of strings (null, boolean, object, array, number, string, integer)

10.4.2 String Validation

- `maxLength`: Non-negative integer
- `minLength`: Non-negative integer
- `pattern`: ECMA-262 regular expression (abstracted in Lean)

10.4.3 Numeric Validation

- `maximum`: Inclusive upper bound
- `minimum`: Inclusive lower bound
- `multipleOf`: Value must be multiple of given number

10.4.4 Array Validation

- `maxItems`: Non-negative integer
- `minItems`: Non-negative integer
- `uniqueItems`: Boolean (default: false)
- `items`: Schema applied to all array items

10.4.5 Object Validation

- `maxProperties`: Non-negative integer
- `minProperties`: Non-negative integer
- `required`: Array of unique property names that must exist
- `properties`: Object mapping property names to schemas

10.4.6 Generic Validation

- `enum`: Non-empty array of allowed values
- `const`: Single fixed JSON value

10.4.7 Schema Composition (Depth-Limited)

- `allOf`: Instance must validate against ALL subschemas (max 3 levels)
- `anyOf`: Instance must validate against AT LEAST ONE subschema (max 3 levels)
- `oneOf`: Instance must validate against EXACTLY ONE subschema (max 3 levels)
- `not`: Instance must NOT validate against subschema (max 3 levels)

10.4.8 Annotation Keywords

- `title`: Human-readable title (informational only)
- `description`: Detailed description (informational only)

10.5 Excluded Features

The following JSON Schema features are **NOT GUARANTEED** at protocol level:

10.5.1 Reference Keywords

- `$ref`: Schema references (prevents termination proof)
- `$defs`: Schema definitions (only useful with `$ref`)
- `$dynamicRef`, `$dynamicAnchor`: Dynamic references

10.5.2 Dynamic Property Validation

- `additionalProperties`: Ignored at protocol level (wallets MAY validate)
- `patternProperties`: Not supported
- `propertyNames`: Not supported
- `unevaluatedProperties`, `unevaluatedItems`: Not supported

10.5.3 Other Excluded Keywords

- `if`, `then`, `else`: Conditional schemas (use `oneOf` instead)
- `prefixItems`, `contains`: Advanced array features
- `dependentRequired`, `dependentSchemas`: Dependency keywords
- `exclusiveMaximum`, `exclusiveMinimum`: Exclusive bounds
- `format`: Format annotations (use `pattern` instead)

10.6 Conformance Requirements

10.6.1 Conformant Schema

A schema is AMATELUS-conformant if:

1. Contains ONLY keywords from supported features
2. Composition nesting does NOT exceed 3 levels
3. `required` array contains unique strings
4. `enum` array is non-empty
5. Numeric constraints are valid numbers
6. Size constraints are non-negative integers
7. `multipleOf` is greater than 0

10.6.2 Conformant Validator

An AMATELUS-conformant validator MUST:

1. Validate all supported keywords correctly
2. **Ignore** excluded keywords without error (e.g., `additionalProperties`)
3. Enforce nesting depth limit (3 levels)
4. Produce deterministic results
5. Reject schemas exceeding nesting limits

10.7 Future Evolution: Convergence Toward Full JSON Schema

10.7.1 Current Status

This subset represents a *formally verifiable foundation*. As Lean's formalization capabilities advance, the subset will gradually expand toward full JSON Schema 2020-12 compliance.

10.7.2 Formalization-Driven Expansion

The expansion strategy follows a principle: **expand the subset only as new features become provably terminating in Lean**.

- **Lean 4 v1.0 onwards**: Recursive reference support (with termination metrics) may enable `$ref` formalization
- **Future versions**: Advanced pattern matching and dependent types may enable:
  - Dynamic property validation with bounded iteration proofs
  - Conditional schema validation
  - More complex nesting patterns
- **Goal**: Eventually, full JSON Schema 2020-12 becomes formally verifiable

10.7.3 Backward Compatibility

- Current AMATELUS-conformant schemas remain valid
- New features are only added when formally verified
- Protocol versioning ensures interoperability during transitions
- Wallets always free to support full JSON Schema 2020-12 beyond protocol guarantees

10.7.4 Versioning Strategy

```
AMATELUS JSON Schema Subset v1.0 (current)
- Formally verified features: 20+ keywords
- Excluded features: circular refs, dynamic validation, format

AMATELUS JSON Schema Subset v1.1 (hypothetical)
- New formalization: conditional validation via oneOf expansion
- Result: if/then/else pattern becomes recommended practice

AMATELUS JSON Schema Subset v2.0 (future)
- Lean capability: recursive references with termination proof
- Result: $ref support added with formal verification guarantees
```

10.8 Example: AMATELUS-Conformant VC Schema

```
{
  "title": "Identity Credential",
  "type": "object",
  "properties": {
    "type": {
      "type": "string",
      "enum": ["IdentityCredential"]
    },
    "credentialSubject": {
      "type": "object",
      "properties": {
        "givenName": {
          "type": "string",
          "minLength": 1,
          "maxLength": 100
        },
        "familyName": {
          "type": "string",
          "minLength": 1,
          "maxLength": 100
        },
        "birthDate": {
          "type": "string",
          "pattern": "^\\d{4}-\\d{2}-\\d{2}$"
```

```
        }
      },
      "required": ["givenName", "familyName"],
      "minProperties": 2,
      "maxProperties": 5
    }
  },
  "required": ["type", "credentialSubject"]
}
```

10.8.1 Why This Schema is Conformant

- Uses only supported keywords (`type`, `properties`, `required`, `enum`, `pattern`)
- Nesting depth is 2 levels (within limit of 3)
- All constraints are valid (minLength, maxLength, minProperties, maxProperties)
- No circular references or dynamic validation
- Formally verifiable in Lean 4

10.9 Security and Robustness

10.9.1 Validation Guarantees

- **Termination**: All schemas guaranteed to validate in finite time
- **Determinism**: Same input always produces identical output
- **No side effects**: Validation is pure computation

10.9.2 Regular Expression Safety

- **ReDoS prevention**: Implementation SHOULD enforce regex timeout limits
- **Abstraction in Lean**: Regex matching abstracted as `MatchedString`/`UnmatchedString`
- **Implementation responsibility**: Wallet implementations must validate regex syntax

10.9.3 Schema Complexity Limits

- **Max nesting**: 3 levels prevents deeply nested schemas
- **Max properties**: No explicit limit, but implementations MAY enforce reasonable bounds
- **No recursion**: Eliminates infinite loops and stack overflow risks

Chapter 11

Trust Chain Architecture

Definition 23. This chapter covers trust chain architecture aspects of AMATELUS.

11.1 Overview

The Trust Chain specification provides a hierarchical delegation mechanism that enables authority transfer through cryptographically verified credentials. Key capabilities include:

- **Authority Delegation**: Upper organizations delegate authority to lower organizations
- **Cryptographic Verification**: Delegation chains are cryptographically verifiable
- **Schema-Based Delegation**: Delegation content is structurally defined using JSON Schema
- **Dynamic Depth Limitation**: Delegators specify `maxDepth`; Nat monotonic decrease prevents infinite hierarchies
- **Holder-Centric Design**: VCs can be issued by anyone (including Holders themselves); verifiers check only the delegation chain's `grantorDID`
- **Per-Claim Signatures**: Recipients sign each claim individually, dramatically reducing VP/ZKP input sizes
- **ZKP-Only Submission**: VPs are internal structures; all submissions use only ZKPs
- **Field-Level Selective Disclosure**: ZKPs enable revelation of only required fields, maximizing privacy

11.2 Trust Chain Components

11.2.1 Architecture

A trust chain flows from a root authority through delegatees to holders:

```
Root Anchor (Trusted Authority)
    v Delegation Credential
    | (delegated authority -> recipient)
```

```
    v
Delegatee (Authority)
    v Attribute Credential
    | (claim with embedded delegation)
    v
Holder (Subject)
    v Re-packaging (optional)
    | (Holder re-issues with own signature)
    v
Verifier (Service Provider)
    verifies:
    - ZKP validity
    - grantorDID in trustedAnchors
    does NOT verify:
    - VC issuer
    - granteeDID matching
```

11.2.2 Critical Design Principles

- **VC Issuer Flexibility**: The `issuer` field can be anyone; trust derives from the delegation chain's `grantorDID`
- **Delegation Chain Focus**: Verifiers inspect `grantorDID` (trust root), not `issuer`
- **ZKP Compatibility**: VCs issued by Holders can still prove delegated authority via ZKP
- **Holder Re-packaging**: Holders can re-issue received VCs with their own signature; each claim's proof remains unchanged

11.3 Delegation Credentials

11.3.1 Structure

A Delegation Credential contains multiple delegations, each authorizing a specific claim type:

- **issuer**: Delegator's DID (must be in `trustedAnchors`)
- **credentialSubject.id**: Original recipient's DID (historical information)
- **credentialSubject.delegations**: Array of delegation objects
- **proof**: Delegator's signature over the entire credential

Each delegation element contains:

- **grantorDID**: Authority granting the right (verification anchor)
- **granteeDID**: Initial recipient DID (historical information)
- **label**: Human-readable claim label (e.g., "Resident Certificate")
- **claimSchema**: JSON Schema (AMATELUS Subset) defining allowed claim structure
- **maxDepth**: Maximum further delegation depth (Nat ≥ 1)
- **proof**: Delegator's signature over the delegation object

11.3.2 Dynamic Depth Limitation

The protocol prevents infinite delegation chains through monotonic depth decrease:

1. Each delegator specifies `maxDepth` (maximum delegations from this point forward)
2. Each recipient computes: `nextDepth = min(parentDepth - 1, delegation.maxDepth)`
3. When depth reaches zero, further delegation is impossible
4. Lean 4 formally proves termination via `termination_by` clause

Example: Government $\xrightarrow{\text{maxDepth=5}}$ Prefecture $\xrightarrow{\text{maxDepth=2}}$ Municipality $\xrightarrow{\text{maxDepth=1}}$ Department $\xrightarrow{\text{maxDepth=0}}$ (chain stops)

11.4 Attribute Credentials with Embedded Delegation

11.4.1 Direct Issuance (0-Layer)

Standard W3C VC issued directly by a trusted authority:

- No delegation chain
- Simple structure: `issuer`, `credentialSubject.claims`, `proof`
- Verification: `issuer` must be in `trustedAnchors`

11.4.2 Delegated Issuance (1-Layer and Beyond)

When issued under delegated authority, each claim embeds delegation information:

- **content**: Actual claim data
- **delegation**: Reference to the delegation authorization
- **delegationProof**: Grantor's signature over the delegation
- **proof**: Grantee's signature over the content (prevents Holder modification)

Verification checks:

1. `delegation.grantorDID` is in `trustedAnchors`
2. `content` conforms to `delegation.claimSchema`
3. `delegationProof` is valid under `grantorDID`
4. `proof` is valid under `granteeDID` (enables ZKP verification)

Does **NOT** verify:

- VC's `issuer` field (can be anyone, including the Holder)
- `granteeDID` matches VC's `issuer`

11.4.3 Holder Re-Packaging

Holders can re-issue received VCs under their own DID:

- VC's `issuer` becomes Holder's DID
- VC's `proof` becomes Holder's signature
- Each claim's `content`, `delegation`, `delegationProof`, `proof` remain unchanged
- Verification remains unchanged (still trusts `grantorDID`)

This enables Holders to maintain privacy from service providers while proving delegated authority through ZKP.

11.5 Verifiable Presentations and ZKP Generation

11.5.1 VP as Internal Structure

Unlike W3C specifications, AMATELUS uses VPs internally only:

- VPs are **never submitted directly** to issuers or verifiers
- VPs serve to organize claims before ZKP generation
- Holders extract only required claims from multiple VCs and assemble them into a VP
- ZKP is generated from the VP's selected claims, not the original VCs

11.5.2 ZKP Input Size Reduction

This design dramatically reduces computational cost:

- **Traditional method**: ZKP includes all claims from all VCs
- **VP+Individual Signatures method**: ZKP includes only selected claims
- **Efficiency gain**: If Holder needs 3 claims from 23 total, ZKP input shrinks to $\approx 1/8$
- **Mobile impact**: Battery consumption, memory use, and response time all improve proportionally

Each claim's individual `proof` (recipient's signature) enables this efficiency:

- Allows claims to be verified independently
- Prevents Holder from modifying claim content (signature would fail)
- Enables selective disclosure without revealing unneeded claims

11.5.3 Submission to Issuers

When requesting a new VC from an issuer:

1. Holder creates a VP (internal only) from relevant credentials
2. Holder generates a ZKP with public inputs including:
   - Holder's DID (issuer needs to record who is requesting)
   - Required fields (e.g., name, age category, income threshold proof)
   - `grantorDIDs` (proof authorities)
3. Holder submits: `{zkp, publicInputs}` only
4. Issuer verifies ZKP and creates new VC with `credentialSubject.id` from public input

11.5.4 Submission to Verifiers

When proving attributes to a service provider:

1. Holder creates a VP (internal only) from relevant credentials
2. Holder generates a ZKP with public inputs including:
   - Required properties only (e.g., `age` $\geq$ `20`, `residence in Japan`)
   - Holder's DID is **hidden** in ZKP's secret inputs
   - `grantorDIDs` (proof authorities)
3. Holder submits: `{zkp, publicInputs}` only
4. Verifier verifies ZKP without learning:
   - Holder's identity (DID)
   - Specific personal information (exact birthdate, address, etc.)
   - Which claims or VCs were used
   - Original credential content

This achieves complete privacy for cross-service authentication: the verifier learns only what is cryptographically necessary.

11.6 Security Properties

11.6.1 Depth Limitation Proofs

Lean 4 formally proves three key properties:

1. **Termination**: `verifyChain` always completes in finite time (via `termination_by remainingDepth`)
2. **Finite Chain Length**: No valid chain can exceed `initialMaxDepth`
3. **N-Layer Finiteness**: Multi-layer delegations remain bounded

11.6.2 Circular Delegation Prevention

DIDs in a chain must be unique (O(n) check):

- Detects cycles by comparing `getAllDIDs(chain)` with `getAllDIDs(chain).eraseDups`
- `verifyChain` returns false if duplicates found

11.6.3 Holder-Centric Verification

By focusing verification on `grantorDID` rather than `issuer`:

- **Issuer Independence**: Credentials can be re-issued by different parties without verification failure
- **Holder Privacy**: Holders can re-package credentials under their own DID without disclosure
- **ZKP Compatibility**: Delegation authority persists even when issuer is hidden via ZKP

11.6.4 Per-Claim Signature Guarantees

Each claim's `proof` (recipient's signature):

- Prevents Holders from modifying claim content
- Enables independent claim verification (critical for VP-based ZKP generation)
- Survives Holder re-packaging (signature remains unchanged)

11.7 Examples

11.7.1 Single-Layer Delegation

Government delegates "Resident Certificate" authority to municipalities:

```
Government (grantorDID)
  v Delegation Credential
    (grants Resident Certificate authority)
    maxDepth = 1
  v
Municipality (granteeDID)
  v Attribute Credential
    (issues resident data with embedded delegation)
  v
Resident (Holder)
  v Requests Bank Account
    (verifier: Bank)
  v
Bank
  - Verifies ZKP
  - Confirms government DID in trustedAnchors
  - Approves account
```

11.7.2 Multi-Layer Delegation with Holder Privacy

```
Government (grantorDID, maxDepth=3)
  v Prefecture (maxDepth=2)
  v Municipality (maxDepth=1)
  v Resident (Holder, issues self-signed VC)
  v Applies for Job
    (submits ZKP: age >= 18, no criminal record)
  v
Employer (Verifier)
  - Verifies ZKP
  - Does NOT learn:
    - Resident's identity
    - Exact age/birthdate
    - Criminal history status
    - Original credentials
  - Learns only:
    - Age threshold met
    - Background check passed
    - Government issued proof
```

Chapter 12

Multi-Device Support

Definition 24. This chapter covers multi-device support aspects of AMATELUS.

12.1 Overview

The Multi-Device Support specification enables a single Holder to operate wallets with different DIDs on multiple devices and safely transfer claims between devices on a per-claim basis.

12.1.1 Background

Identity-related VCs have the following characteristics:

- **Local Reception Requirement**: Most identity VCs are issued in-person at municipal offices, police stations, etc.
- **Smartphone Reception**: Smartphone is the typical device for in-person VC reception
- **PC Utilization Need**: When participating in the AMATELUS network from home, users typically use a PC

Multi-device support is essential to meet these diverse requirements.

12.1.2 Design Principles

1. **Non-Shared Private Keys**: Private keys are never transferred between devices
2. **Per-Claim Transfer**: Claims (not entire VCs) are transferred individually
3. **Dual Signatures**: Issuer signature + original subject transfer signature
4. **W3C Standard Compliance**: Supports `holder` $\neq$ `credentialSubject`
5. **DIDComm Protocol**: Uses standard DID-to-DID communication protocol
6. **End-to-End Encryption**: Transferred data is always encrypted
7. **Protocol-Level Rules**: Claims without signatures are ignored

12.2 Design Philosophy

12.2.1 Comparison with Trust Chain

AMATELUS provides two distinct mechanisms:

Function	Use Case	Mechanism	DID Relationship
Trust Chain	Schema inheritance, authority delegation	DelegationChain	Different organizational DIDs
Multi-Device	Claim sharing	Per-claim transfer	Same Holder's different DIDs

12.2.2 Claim Transfer Paths

In the AMATELUS protocol, claims are transferred via two paths:

1. **Issuer** → **Holder_A**: Initial claim issuance
2. **Holder_A** → **Holder_B**: Per-claim transfer (subject of this specification)

External submissions always use only ZKPs; claims themselves are never revealed externally.

12.2.3 Claim Structure

All claims must have an issuer signature:

```
structure SignedClaim where
  content : String
  delegationChain : Option DelegationChain
  issuerSignature : Signature
```

Protocol-Level Rule: Claims without signatures are ignored.

12.2.4 Transferred Claim Structure

Claims transferred between devices carry dual signatures:

```
structure TransferredClaim where
  originalClaim : SignedClaim
  originalSubjectDID : ValidDID
  currentHolderDID : ValidDID
  transferProof : Signature
  transferredAt : Nat
```

Dual Signature Roles:

1. **issuerSignature**: Guarantees claim authenticity (issued by municipality)
2. **transferProof**: Proves claim ownership and transfer consent (owned and transferred by original DID)

12.2.5 W3C Standard Alignment

W3C VC standards allow `holder` (VC possessor) and `credentialSubject` (VC subject) to be different ($\neq$).

Per-claim transfer maintains:

- Issuer signature remains unchanged (original issuer's signature)
- `originalSubjectDID` remains unchanged (original DID maintained)
- `currentHolderDID` is the new device's DID
- PC wallet can possess and use for ZKP generation

12.3 Use Cases

12.3.1 Municipal Certificate Reception

1. Alice brings smartphone to municipal office
2. Officer verifies smartphone DID: `did:amt:alice_smartphone`
3. Municipality issues residence certificate VC to smartphone wallet
4. Smartphone wallet receives and stores VC

12.3.2 Claim Transfer to Home PC

1. Alice returns home
2. PC wallet boots: `did:amt:alice_pc`
3. Smartphone wallet initiates claim transfer request to PC wallet
4. Smartphone wallet generates transfer signature (signed with DID__A private key)
5. DIDComm protocol encrypts and transfers claim
6. PC wallet receives, verifies, and stores claim
7. PC can now participate in AMATELUS network

12.3.3 Claim Utilization (ZKP Generation)

When PC wallet generates ZKP:

```
{
  "public_inputs": {
    "holder_did": "did:amt:alice_pc",
    "age_gte": 20,
    "residence": "Tokyo"
  },
  "proof": "..."
}
```

Important Notes:

- ZKP generation is performed by PC wallet
- `originalSubjectDID` (smartphone's original DID) is secret input
- `currentHolderDID` (PC's DID) is public input
- Both issuer and transfer signatures are verified within ZKP
- **Only required claims are input** (other claims are unnecessary)

12.4 DIDComm Communication Protocol

12.4.1 DIDComm Overview

DIDComm (DID Communication) is the standard protocol enabling secure communication between DIDs:

- **Specification**: DIDComm Messaging v2.0
- **Features**:
  - End-to-end encryption
  - Authenticated messaging
  - Transport-agnostic (HTTP, WebSocket, Bluetooth, etc.)

12.4.2 DIDComm Message Structure

Claim Transfer Request

```
{
  "type": "https://amatelus.org/protocols/claim-transfer/2.0/request",
  "id": "uuid-1234-5678",
  "from": "did:amt:alice_pc",
  "to": "did:amt:alice_smartphone",
  "created_time": 1673568000,
  "body": {
    "request_type": "filtered_claims",
    "filters": {
      "content_patterns": ["name", "address"],
      "issuer": "did:amt:municipality123"
    }
  }
}
```

Claim Transfer Response

```
{
  "type": "https://amatelus.org/protocols/claim-transfer/2.0/response",
  "from": "did:amt:alice_smartphone",
  "to": "did:amt:alice_pc",
```

```
  "body": {
    "transferred_claims": [
      {
        "original_claim": {
          "content": "{\"name\": \"Alice\", \"address\": \"Tokyo\"}",
          "issuer_signature": "..."
        },
        "original_subject_did": "did:amt:alice_smartphone",
        "current_holder_did": "did:amt:alice_pc",
        "transfer_proof": "...",
        "transferred_at": 1673568010
      }
    ],
    "success": true
  }
}
```

12.4.3 Authentication Flow

Device Pairing

Initial pairing of two wallets:

1. PC displays QR code (DID + temporary token)
2. Smartphone scans QR code
3. Smartphone sends DIDComm connection request
4. PC prompts user for approval
5. User approves on PC
6. Both devices add peer DID to trusted list

Mutual Authentication

Communication between paired devices:

1. Request side signs DIDComm message
2. Response side verifies signature and checks trusted list
3. Response side sends encrypted response
4. Request side decrypts and verifies response

12.5 Claim Transfer Mechanism

12.5.1 Transferred Data

Data sent during claim transfer:

```
{
  "original_claim": {
    "content": "{\"name\": \"Alice\", \"address\": \"Tokyo\"}",
    "issuer_signature": "..."
  },
  "original_subject_did": "did:amt:alice_smartphone",
  "current_holder_did": "did:amt:alice_pc",
  "transfer_proof": "...",
  "transferred_at": 1673568010
}
```

Unchanged Elements:

- `original_claim.content`: Claim content
- `original_claim.issuer_signature`: Issuer signature
- `original_subject_did`: Original holder DID (smartphone)

Added Elements:

- `transfer_proof`: Transfer signature by original subject (required)
- `current_holder_did`: Current holder DID (PC)
- `transferred_at`: Transfer timestamp

12.5.2 Transfer Signature Generation

Smartphone wallet generates transfer signature:

```
def prepareClaimTransfer
    (claim : SignedClaim)
    (originalSubjectDID : ValidDID)
    (currentHolderDID : ValidDID)
    (timestamp : Nat) : TransferredClaim :=
  let message := encodeTransferMessage claim.content
                   originalSubjectDID currentHolderDID
  let transferProof := sign message originalSubjectSecretKey
  {
    originalClaim := claim,
    originalSubjectDID := originalSubjectDID,
    currentHolderDID := currentHolderDID,
    transferProof := transferProof,
    transferredAt := timestamp
  }
```

12.5.3 Receiving Wallet Validation

PC wallet validates received claim:

```
def validateClaim
    (tc : TransferredClaim)
    (issuerDID : ValidDID)
    (trustedAnchors : List ValidDID) : Bool :=
  let issuerSigValid := tc.originalClaim.verify issuerDID
  let transferSigValid := tc.verifyTransferProof
  let issuerTrusted := trustedAnchors.contains issuerDID
  issuerSigValid && transferSigValid && issuerTrusted
```

12.5.4 Receiving Wallet Storage

PC wallet stores received claim in this structure:

```
{
  "claim_id": "uuid-claim-001",
  "original_claim": {
    "content": "{\"name\": \"Alice\"}",
    "issuer_signature": "..."
  },
  "original_subject_did": "did:amt:alice_smartphone",
  "current_holder_did": "did:amt:alice_pc",
  "transfer_proof": "...",
  "storage_metadata": {
    "received_at": "2025-01-13T12:00:00Z",
    "received_from": "did:amt:alice_smartphone",
    "issuer_did": "did:amt:municipality123"
  }
}
```

12.6 ZKP Efficiency

12.6.1 Per-Claim Transfer Advantages

Problem with V1.0 (VC-based transfer):

- Transferring entire VC requires all claims as ZKP inputs during generation
- Unnecessary claims bloat the circuit size
- High computational cost with privacy concerns

Advantages of V2.0 (Per-claim transfer):

- Only required claims input to ZKP circuit
- Circuit size minimized, computational cost reduced
- Enhanced privacy (unnecessary claims not handled)
- Dual signatures provide complete security guarantee

12.6.2 ZKP Generation Process

When PC wallet generates ZKP:

```
structure ZKPSecretInputsForTransferredClaim where
  claimContent : String
  issuerSignature : Signature
  transferSignature : Signature
  originalSubjectDID : ValidDID

structure ZKPPublicInputsForTransferredClaim where
  currentHolderDID : ValidDID
  publicAttributes : List (String $\times$ String)
```

ZKP Circuit Verification:

1. `issuerSignature` is valid (municipality issued)
2. `transferSignature` is valid (DID_A owns and transferred)
3. `originalSubjectDID` matches subject in claim (integrity)

12.6.3 Computational Cost Comparison

Method	Input Claims	Circuit Size	Computation Cost
V1.0 (VC transfer)	All claims (5)	Large	High
V2.0 (Per-claim transfer)	Required only (1)	Small	Low

12.6.4 Privacy Enhancement

Per-claim transfer enables:

- Excluding unnecessary claims from ZKP circuit
- Hiding existence of unnecessary claims
- Improved selective disclosure granularity

12.7 Security Considerations

12.7.1 Private Key Non-Sharing

Critical Principle: Private keys are never transferred between devices

- Each device maintains independent DID and key pair
- Only claim data (issuer signature + transfer signature) is transferred
- Private key compromise impact is limited to that device

12.7.2 Dual Signature Security

Per-claim transfer provides complete security through dual signatures:

1. **issuerSignature**:
   - Guarantees claim authenticity
   - Proves issuance by issuer (municipality)
   - Prevents tampering
2. **transferProof**:
   - Proves claim ownership
   - Proves original subject (DID_A) owned it
   - Proves transfer consent

Security Theorem:

```
theorem claim_transfer_preserves_issuer_signature :
  forall (claim : SignedClaim)
    (originalSubjectDID currentHolderDID : ValidDID)
    (timestamp : Nat),
  let tc := prepareClaimTransfer claim originalSubjectDID
              currentHolderDID timestamp
  tc.originalClaim.issuerSignature = claim.issuerSignature
```

12.7.3 End-to-End Encryption

All claim transfers use DIDComm encryption:

1. Sender encrypts with recipient's public key
2. Transport layer applies additional encryption (optional, e.g., TLS)
3. Recipient decrypts with own private key

12.7.4 Device Authentication

Pairing-Time Authentication

- QR code + temporary token
- Explicit user approval
- Registration in trusted list

Transfer-Time Authentication

- Trusted list verification
- DIDComm signature verification
- Timestamp verification (replay attack prevention)

12.7.5 Claim Integrity Verification

Receiving wallet verifies:

```
def validateClaim
    (tc : TransferredClaim)
    (issuerDID : ValidDID)
    (trustedAnchors : List ValidDID) : Bool :=
  let issuerSigValid := tc.originalClaim.verify issuerDID
  let transferSigValid := tc.verifyTransferProof
  let issuerTrusted := trustedAnchors.contains issuerDID
  issuerSigValid && transferSigValid && issuerTrusted
```

12.7.6 Man-in-the-Middle (MITM) Attack Mitigation

DIDComm protocol provides:

1. **AEAD**: Tampering detection
2. **DID Signatures**: Sender authenticity guarantee
3. **Pairing Confirmation**: User approval
4. **Dual Signatures**: Complete integrity guarantee

12.7.7 Privacy Protection

- **Claim Transfer Secrecy**: Transfer fact unknown to third parties
- **originalSubjectDID Protection**: Secret input during ZKP generation
- **Selective Transfer**: Only required claims transferred
- **Metadata Separation**: Transfer history stored locally only

12.7.8 Protocol-Level Guarantee

Claims without signatures are ignored:

- All claims must have issuer signature
- Transferred claims must have transfer signature
- Unsigned claims automatically rejected at protocol level
- Tampering attempts automatically fail

12.8 Implementation Guidelines

12.8.1 Wallet Implementation Requirements

Essential Features

1. **DIDComm Support**
   - Implement DIDComm v2.0 protocol
   - Support end-to-end encryption
2. **Device Management**
   - Manage trusted device list
   - Implement pairing (QR code, etc.)
3. **Claim Transfer**
   - Support claim sending (with filtering)
   - Generate transfer signatures (original subject's private key)
   - Support claim receiving (with dual signature verification)
4. **ZKP Generation Extension**
   - Support `currentHolderDID` as public input
   - Support `originalSubjectDID` as secret input
   - Verify both issuer and transfer signatures in ZKP
   - Input only required claims (others unnecessary)
5. **Signature Verification**
   - Verify issuer signatures
   - Verify transfer signatures
   - Reject unsigned claims (protocol-level)

Recommended Features

1. **Selective Transfer**
   - Transfer specific claims only
   - Content pattern filtering
   - Issuer-based filtering
2. **Transfer History**
   - Log device and transfer timestamp
   - Support transfer revocation (revocation notification)
3. **Automatic Synchronization**
   - Auto-transfer new claims

- Configure inter-device sync

4. **Claim Management**
   - Per-claim storage and search
   - Display delegation chains
   - Claim selection UI for ZKP generation

12.8.2 Transport Layer Options

DIDComm supports multiple transports:

Transport	Use Case	Characteristics
HTTP/HTTPS	Internet via cloud	Cloud relay possible
WebSocket	Real-time communication	Bidirectional
Bluetooth	Local communication	No internet required
NFC	Proximity communication	Very short range

Recommended configuration:

- Same network: WebSocket (fast)
- Different networks: HTTPS (stable)
- Offline: Bluetooth (no internet)

12.8.3 Error Handling

Transfer Failure Response

```
{
  "type": "https://amatelus.org/protocols/claim-transfer/2.0/error",
  "thid": "uuid-1234-5678",
  "body": {
    "error_code": "SIGNATURE_VERIFICATION_FAILED",
    "error_message": "Signature verification failed",
    "details": {
      "claim_id": "uuid-claim-001",
      "issuer": "did:amt:municipality123"
    }
  }
}
```

Retry Logic

1. Detect transfer failure
2. Retry with exponential backoff (max 3 times)
3. Notify user on final failure
4. Log failure

12.8.4 Testing Strategy

Functional Testing

- Pairing success/failure
- Claim transfer success/failure
- Issuer signature verification
- Transfer signature verification
- Dual signature verification
- Filtering functionality
- Unsigned claim rejection

Security Testing

- MITM attack simulation
- Invalid claim rejection
- Signature tampering detection
- Untrusted device rejection
- Automatic unsigned claim rejection

Performance Testing

- Large volume claim transfer performance
- Encryption/decryption overhead
- Network latency handling
- ZKP generation time comparison (V1.0 vs V2.0)

ZKP Efficiency Testing

- Circuit size comparison by claim count
- Performance with required claims only
- Comparison with all claims

12.9 Future Extensions

12.9.1 Cloud Synchronization

Use trusted cloud service as relay:

```
Smartphone -> [Encrypted] -> Cloud -> [Decrypted] -> PC
```

Requirements:

- End-to-end encryption mandatory
- Cloud cannot access claim content
- Zero-knowledge proof-based backup
- Per-claim synchronization

12.9.2 Group Wallet

Safe claim sharing within family or organization:

```
Parent Wallet <-> Child Wallet (guardian function)
Corporate Wallet <-> Employee Wallet (role proof)
```

Requirements:

- Per-claim sharing control
- Dual signatures guarantee integrity
- Improved selective disclosure granularity

12.9.3 Conditional Transfer

Enable claim transfer only under specific conditions:

```
{
  "transfer_policy": {
    "allowed_devices": ["did:amt:alice_pc", "did:amt:alice_tablet"],
    "allowed_time": "09:00-18:00",
    "require_biometric": true,
    "max_transfers": 3,
    "allowed_claims": ["name", "address"]
  }
}
```

12.9.4 Trust Chain Integration

Combine N-level delegation with per-claim transfer:

- Transfer claims containing delegation chains
- Verify delegation chains at transfer destination
- ZKP efficiency (required claims only)

12.10 Formal Verification

Theorems formally proven in `AMATELUS/MultiDevice.lean`:

1. **claim_transfer_preserves_issuer_signature**
   - Issuer signature preserved during transfer
2. **claim_transfer_preserves_content**
   - Claim content preserved during transfer
3. **transferred_claim_has_transfer_proof**
   - Transferred claims always have transfer signature
4. **device_trust_symmetric**
   - Device trust verification symmetry
5. **valid_claim_stays_valid_after_transfer**
   - Valid claims remain valid after transfer

These theorems formally guarantee the security of per-claim transfer.

Chapter 13

Merkle Tree Revocation

Definition 25. This chapter covers merkle tree revocation aspects of AMATELUS.

13.1 Abstract

This chapter defines the detailed specification of **Merkle Tree-based revocation verification flow** in the AMATELUS protocol. Unlike W3C Bitstring Status List which reveals credential identification information, the Merkle Tree approach preserves zero-knowledge properties.

13.2 Background and Motivation

13.2.1 Problem with Bitstring Status List

W3C Bitstring Status List requires public disclosure of `statusListIndex`, creating a fundamental contradiction with Zero-Knowledge Proofs:

```
Traditional approach (Bitstring Status List):
  Holder -> statusListIndex -> Issuer

  Problems:
  - statusListIndex disclosure = credential identification
  - ZKP zero-knowledge property collapses
  - Privacy protection impossible
```

13.2.2 Design Goals

This specification achieves:

1. **Zero-Knowledge Preservation**: Specific credential presentation never identified
2. **Revocation Safety**: Revoked VCs cannot generate ZKPs
3. **Scalability**: O(log N) computation (N = active VC count)
4. **Disaster Availability**: Revocation check skippable offline
5. **W3C VC Compatibility**: Implemented as credentialStatus extension

6. **Personal Issuer Support**: Non-server-managing issuers can operate

13.2.3 Personal Issuer Challenge and Solution

Challenge

Merkle Tree revocation requires issuers to operate centralized web servers (for Merkle Root publication):

```
Problems:
  - Personal issuers may not maintain 24/7 HTTP servers
  - Server management costs (domain, SSL, infrastructure)
  - Difficulty guaranteeing constant availability
```

Solution: revocationEnabled Flag

Design Principle:

```
Issuers include revocation capability at issuance time
  down arrow
Holders input this flag during ZKP generation
  down arrow
Verifiers can mathematically determine revocation status
```

Implementation:

1. Issuer includes `revocationEnabled: true/false` in claims at issuance
2. This flag is under issuer signature (tamper-proof)
3. ZKP circuit verifies:
   - `revocationEnabled = true` requires Merkle proof verification
   - `revocationEnabled = false` skips Merkle proof verification
4. Verifier receives `revocationEnabled` in public ZKP inputs

Benefits:

- Personal issuers require no server management (`revocationEnabled = false`)
- Verifiers mathematically confirm revocation status (Holder cannot hide)
- Organizational issuers provide high trust (`revocationEnabled = true`)

13.3 Architecture Overview

13.3.1 Overall Flow

```
Issuer

1. Issue VC
```

```
  2. Manage Active List
     [H(VC_1), H(VC_2), ...]
  3. Generate Merkle Root
     root = H(...)
  4. Publish Root (signed)


            Merkle Root + Signature
            (updated hourly)
          down arrow

   Holder

  1. Fetch Merkle Root
  2. Generate Merkle Proof
     proof = [h_1, h_2, ...]
  3. Generate ZKP
     - VC content (secret)
     - Merkle proof (secret)
     - Merkle Root (public)


            ZKP + Merkle Root

          down arrow

   Verifier

  1. Verify Merkle Root
     - Check issuer signature
  2. Verify timestamp
     - Check validUntil (issuer-signed)
     - Check version
  3. Verify ZKP
     - Verify Merkle proof in circuit
     - VC in Active List? -> OK
     - VC not in list? -> NG
```

13.3.2 Data Structures

Merkle Revocation List

Information managed by issuer:

```
structure MerkleRevocationList where
  activeVCHashes : List Hash
  merkleRoot : Hash
  updatedAt : Timestamp
  validUntil : Timestamp
```

```
  version : Nat
  issuerSignature : Signature
```

Merkle Proof

Inclusion proof generated by holder:

```
structure MerkleProof where
  leafIndex : Nat
  siblingHashes : List Hash
  treeDepth : Nat
```

ZKP Inputs with Revocation

Secret Inputs:

```
structure ZKPSecretInputWithRevocation where
  vcContent : String
  issuerSignature : Signature
  merkleProof : Option MerkleProof
  additionalSecrets : List (String * String)
```

Public Inputs:

```
structure ZKPPublicInputWithRevocation where
  revocationEnabled : Bool
  merkleRoot : Option Hash
  merkleRootVersion : Option Nat
  publicAttributes : List (String * String)
  verifierNonce : Nonce
  holderNonce : Nonce
```

13.4 Merkle Tree Construction

13.4.1 Hash Function

- **SHA-256**: Merkle Tree construction (ZKP circuit compatible)
- **SHA3-512**: Other purposes (quantum safe)

13.4.2 Tree Construction Algorithm

```
Input: activeVCHashes = [h_1, h_2, ..., h_n]
Output: merkleRoot

Algorithm:
  1. Padding (adjust to power of 2)
     if n not power of 2:
       pad with H("") until next power of 2

  2. Level 0 (leaves)
```

```
    leaves = activeVCHashes + padding

  3. Upward computation
    while len(leaves) > 1:
      new_level = []
      for i in range(0, len(leaves), 2):
        parent = SHA-256(leaves[i] || leaves[i+1])
        new_level.append(parent)
      leaves = new_level

  4. Return root
    return leaves[0]
```

13.4.3 Merkle Proof Generation

```
Input:
  - vcHash (VC to prove)
  - activeVCHashes (all active VCs)
  - merkleRoot (for verification)

Output: MerkleProof

Algorithm:
  1. Find VC position
    leafIndex = activeVCHashes.indexOf(vcHash)
    if leafIndex == -1: return None

  2. Collect proof path
    siblingHashes = []
    currentIndex = leafIndex
    currentLevel = activeVCHashes + padding

    while len(currentLevel) > 1:
      siblingIndex = currentIndex XOR 1
      siblingHashes.append(currentLevel[siblingIndex])
      currentIndex = currentIndex / 2
      currentLevel = computeParentLevel(currentLevel)

  3. Return proof
    return MerkleProof {
      leafIndex,
      siblingHashes,
      treeDepth = log_2(len(activeVCHashes))
    }
```

13.4.4 Merkle Proof Verification

```
Input:
  - vcHash (VC to verify)
```

```
  - proof (MerkleProof)
  - merkleRoot (expected root)

Output: Bool

Algorithm:
  1. Start from leaf
     currentHash = vcHash
     currentIndex = proof.leafIndex

  2. Compute to root
     for siblingHash in proof.siblingHashes:
       if currentIndex % 2 == 0:
         currentHash = SHA-256(currentHash || siblingHash)
       else:
         currentHash = SHA-256(siblingHash || currentHash)
       currentIndex = currentIndex / 2

  3. Compare with root
     return currentHash == merkleRoot
```

13.5 ZKP Circuit Integration

13.5.1 Circuit Constraints

Public Inputs:

```
- revocation_enabled (enablement flag from VC claim)
- merkle_root (latest, if revocation_enabled = true)
- merkle_root_version (version, if revocation_enabled = true)
- claimed_attributes (age >= 20, etc.)
- verifier_nonce
- holder_nonce
```

Private Inputs:

```
- vc_full (complete VC content)
- issuer_signature (VC signature from issuer)
- merkle_proof.leafIndex (if revocation_enabled = true)
- merkle_proof.siblingHashes (if revocation_enabled = true)
- merkle_proof.treeDepth (if revocation_enabled = true)
```

Constraints:

1. Verify issuer signature
2. Extract and match revocationEnabled flag from VC
3. Compute VC hash: `vc_hash = SHA-256(Canonicalize(vc_full))`
4. If `revocation_enabled = true`: Verify Merkle proof matches `merkle_root`

5. If `revocation_enabled = false`: Skip Merkle proof verification
6. Verify selective attribute disclosure
7. Verify nonce binding: `nonce_combined = SHA-256(holder_nonce || verifier_nonce)`

13.6 Issuer Operations

13.6.1 VC Issuance

With Revocation Enabled

For organizational issuers with server infrastructure:

1. Generate VC with `revocationEnabled: true`
2. Compute VC hash
3. Add to Active List
4. Generate Merkle Root
5. Publish signed Merkle Root with `validUntil`
6. Send VC + Merkle Proof to Holder

With Revocation Disabled

For personal issuers without servers:

1. Generate VC with `revocationEnabled: false`
2. Skip all Merkle operations
3. Send VC to Holder only

13.6.2 VC Revocation

For `revocationEnabled = true` credentials:

1. Receive revocation request from Holder
2. Compute VC hash
3. Remove from Active List
4. Generate new Merkle Root
5. Publish signed with new version
6. Log revocation for audit

For `revocationEnabled = false` credentials:

- No revocation mechanism exists
- Request holder to discard credential
- Or issue replacement with `revocationEnabled = true`

13.6.3 Periodic Merkle Root Updates

Execute hourly (Cron Job):

1. Load current Active List
2. Recompute Merkle Root
3. Increment version number
4. Set `validUntil = now() + 1 hour`
5. Sign: `issuerSignature = Sign(merkleRoot || version || validUntil)`
6. Publish

13.7 Holder Operations

13.7.1 ZKP Generation with Revocation

When revocationEnabled = true

1. Extract `revocationEnabled` flag from VC
2. Fetch latest Merkle Root (verify issuer signature)
3. Verify Merkle Root expiration (`validUntil`)
4. Verify Merkle Proof locally
5. Generate holder nonce
6. Prepare ZKP secret inputs (include Merkle Proof)
7. Prepare ZKP public inputs (include merkleRoot, merkleRootVersion)
8. Generate ZKP
9. Return ZKP + revocationEnabled flag + merkleRoot + version + nonce

When revocationEnabled = false

1. Extract `revocationEnabled` flag from VC
2. Skip Merkle Root fetch
3. Skip Merkle Proof generation
4. Generate holder nonce
5. Prepare ZKP secret inputs (no Merkle Proof)
6. Prepare ZKP public inputs (no merkleRoot, merkleRootVersion)
7. Generate ZKP
8. Return ZKP + revocationEnabled flag + nonce

13.8 Verifier Operations

13.8.1 ZKP Verification with Revocation

When revocationEnabled = true

1. Fetch latest Merkle Root from issuer
2. Verify issuer signature on Merkle Root
3. Fetch historical Merkle Root (by version) used by holder
4. Verify issuer signature on historical root
5. Verify timestamp: `now() <= validUntil`
6. Verify version lag (e.g., MAX_VERSION_LAG = 5)
7. Verify Merkle Root matches
8. Verify ZKP with public inputs (`revocationEnabled = true`, `merkleRoot`, `merkleRootVersion`)
9. Verify nonce uniqueness (prevent replay)
10. Accept if all checks pass

When revocationEnabled = false

1. Check verifier policy (accept non-revocable VCs?)
2. If rejected by policy, return False
3. Verify ZKP with public inputs (`revocationEnabled = false`, no merkleRoot)
4. Verify nonce uniqueness
5. Log warning (revocation check skipped)
6. Accept if policy allows

13.8.2 Offline Verification Mode

During disasters when issuer server unavailable:

1. Use cached issuer public key
2. Verify ZKP without revocation check
3. Log warning (offline mode)
4. Accept if previous successful verification logged

13.9 W3C VC Integration

13.9.1 credentialStatus Extension

With Revocation Enabled

```
{
  "@context": [
    "https://www.w3.org/ns/credentials/v2",
    "https://amatelus.example/context/revocation/v1"
  ],
  "credentialStatus": {
    "id": "https://issuer.example/status/merkle/v1",
    "type": "MerkleTreeRevocationList2024",
    "merkleRootEndpoint": "https://issuer.example/api/merkle-root",
    "merkleProofEndpoint": "https://issuer.example/api/merkle-proof",
    "vcHash": "0x1234567890abcdef..."
  },
  "credentialSubject": {
    "revocationEnabled": true,
    ...
  }
}
```

With Revocation Disabled

```
{
  "@context": [
    "https://www.w3.org/ns/credentials/v2"
  ],
  "credentialSubject": {
    "revocationEnabled": false,
    ...
  }
  // no credentialStatus field
}
```

13.10 Security Analysis

13.10.1 Zero-Knowledge Guarantee

Method	VC Identifiable?	Information Leaked	ZK Property
Bitstring Status List	Yes	statusListIndex	No
Merkle Tree	No	merkleRoot only	Yes

Proof: Verifier receives only `merkleRoot` (hash of all active VCs). No information identifies specific VC. Merkle proof path remains secret in ZKP circuit.

13.10.2 Revocation Safety

Theorem (Formal Verification): When VC is revoked:

1. `activeVCHashes` no longer contains `vc_hash`
2. New `merkleRoot` generated without `vc_hash`
3. Holder attempts ZKP generation with revoked VC
4. Merkle proof verification fails in ZKP circuit
5. ZKP generation fails
6. Mathematical guarantee: revoked VCs cannot generate valid ZKPs

13.10.3 Timestamp Forgery Resistance

Problem: ZKP circuit cannot safely verify timestamps (holder controls time input)
Solution: Verifier-side verification with issuer signature

```
issuerSignature = Sign(merkleRoot || version || validUntil)

Verifier verification:
  1. Verify(issuerSignature, issuer_pubkey) = True
  2. now() <= validUntil

Guarantee:
  - Holder cannot forge timestamp (requires issuer private key)
  - Old merkleRoot (before revocation) cannot be used
  - Dilithium2 signature forgery requires 2^128 operations
```

13.10.4 Replay Attack Resistance

Dual-Nonce Mechanism:

```
nonce_combined = SHA-256(holder_nonce || verifier_nonce)

Protection:
- If holder's nonce generation bugs: verifier's nonce randomness protects
- If verifier's nonce generation bugs: holder's nonce randomness protects
- Replay: old nonces recorded, duplicates rejected
```

13.11 Computational Complexity

Operation	Complexity	Example (N=1M)
Merkle Tree Construction	O(N)	1M hash operations
Merkle Proof Generation	O(log N)	20 hash operations
Merkle Proof Verification	O(log N)	20 hash operations
ZKP Circuit Verification	O(log N)	20 SHA-256 ops

Scalability: 100 million active VCs produces Merkle proof depth = 27, requiring 27 SHA-256 computations in ZKP circuit (practical).

13.12 Implementation Checklist

13.12.1 Issuer Requirements

- Active VC list database design
- Merkle Tree library (SHA-256)
- Hourly update Cron Job
- API Endpoints: `GET /api/merkle-root`, `GET /api/merkle-proof`, `POST /api/revoke`
- Issuer private key management (HSM recommended)
- Revocation audit logging
- Store multiple Merkle Root versions (latest 5+)

13.12.2 Holder Requirements

- Merkle Root fetch logic
- Merkle proof generation/verification
- ZKP circuit integration (SHA-256, issuer signature verification, attribute extraction)
- Background precomputation management
- Offline mode support
- Error handling (revocation notification)

13.12.3 Verifier Requirements

- Issuer public key retrieval and verification
- Merkle Root fetch and signature verification
- ZKP verification library
- Nonce management (replay prevention)
- Timestamp verification
- Offline mode support (optional)
- Audit logging

13.13 Conclusion

This Merkle Tree revocation mechanism solves the fundamental contradiction between Verifiable Credentials' privacy requirements and revocation safety:

1. **Zero-Knowledge Preservation**: Unlike Bitstring Status List
2. **Revocation Guarantee**: Mathematical certainty that revoked VCs cannot generate ZKPs
3. **Timestamp Security**: Issuer signature prevents Holder timestamp forgery
4. **Scalability**: O(log N) logarithmic complexity
5. **Personal Issuer Support**: No server infrastructure required (revocationEnabled = false)
6. **Disaster Tolerance**: Offline verification possible
7. **W3C Compatible**: Standard credentialStatus extension

The core innovation is the **revocationEnabled flag**, which enables: - Mathematical determination of revocation capability - Support for personal issuers (no server required) - Verifier policy enforcement (accept or reject non-revocable credentials) - Complete accountability (Holder cannot hide revocation status from Verifier)

Chapter 14

Audit Mechanism: Anonymous Hash Identifier (AHI)

Definition 26. This chapter covers audit mechanism: anonymous hash identifier (ahi) aspects of AMATELUS.

14.1 Overview

This chapter specifies the **optional audit mechanism** within AMATELUS protocol. The **Anonymous Hash Identifier (AHI)** is used **only when required by Issuer or Verifier**, protecting citizen privacy while enabling audits based on legal procedures and accountability.

14.1.1 Key Premise: AHI is Optional

AHI is used only in the following cases:

1. **Services requiring audit**:
   - Tax and benefit services
   - Licensing and permits
   - Financial institution compliance
2. **Anti-Sybil services**:
   - Social networks (misinformation prevention)
   - Ticket sales (scalping prevention)
   - Online gaming (multi-accounting prevention)
3. **Services explicitly requiring AHI**

When AHI is not required, standard VC and ZKP mechanisms provide complete privacy.

14.1.2 Problem Statement

Traditional audit DIDs had critical vulnerabilities:

- **Intentional key loss**: Citizens could discard secret keys to evade tracking
- **Cross-service linkage**: Single audit DID creates profiling risks across services
- **Privacy-security tradeoff**: Difficult to balance audit capability with privacy

AHI resolves these through cryptographic binding to national identity numbers.

14.1.3 Design Goals

- **Optionality**: AHI is protocol-optional, not mandatory
- **Privacy**: Prevent government centralization of citizen information
- **Anti-linkage**: Prevent profiling across different services
- **Audit guarantee**: Make evasion technically impossible for audit-required services
- **Legal alignment**: Enable audits based on court warrants and legal procedure
- **Global applicability**: Work with or without national ID systems

14.2 Architecture

14.2.1 Actors and Roles

Actor	Role	Responsibility
Citizen	Service user	Generate and manage AHI, create ZKP
Municipality	VC issuer	Execute court-ordered national ID disclosure
Government/Audit Authority	Audit administrator	Issue and publish audit section IDs
Service Provider	Service operator	Verify ZKP, record AHI, detect fraud
Court	Legal authority	Warrant issuance and legitimacy review
Investigating Agency	Investigation authority	Request disclosure and investigate

14.2.2 System Architecture

The audit mechanism involves three key components:

1. **Audit Section Identifier**: Public identifier issued by government for each audit purpose
2. **Anonymous Hash Identifier**: Hash computed as $H(\text{AuditSectionID} \parallel \text{NationalID})$
3. **Zero-Knowledge Proof**: Proves hash generation correctness without revealing NationalID

Critical: These components are used **only when AHI is required**.

14.2.3 National ID System Abstraction

AMATELUS does not depend on specific national ID schemes. Compatible systems include:

- My Number (Japan)
- Social Security Number (USA)
- National Insurance Number (UK)
- Other national identification schemes

For countries without national ID systems, AHI functionality is unavailable, but other AMATELUS features (VC, ZKP, DID) work normally.

14.3 Data Structures

14.3.1 Audit Section Identifier

```
structure AuditCategoryId where
  id : String
  purpose : String              -- e.g., "taxation", "benefits", "licensing"
  issuer : ValidDID             -- Government or audit authority DID
  issuedAt : Nat                -- Issuance timestamp
  deriving Repr, DecidableEq
```

Properties:

- Issued by government for each audit purpose
- Verifiable in public registry
- Guaranteed unique (UUID or random string)
- Example: `"tax-2025"`, `"subsidy-education-2025"`, `"permit-building-tokyo"`

14.3.2 Anonymous Hash Identifier

```
def generateAnonymousHashId
    (categoryId : AuditCategoryId)
    (myNumber : String) : ByteArray :=
  -- Post-quantum cryptography resistant hash function
  Hash.pqcHash (categoryId.id ++ "||" ++ myNumber)
```

Mathematical definition:

$$\text{AHI} = H(\text{AuditCategoryID} \parallel \text{NationalID})$$

where H is a PQC-resistant hash function.
Properties:

- Uses post-quantum cryptography (SHA-3, BLAKE3, etc.)
- Service-specific identifiers (prevents cross-service linking)
- One-way function (irreversible)
- Collision-resistant

14.3.3 National ID Verifiable Credential

Important: This VC is required only when using AHI. Services not requiring AHI do not need it.

```
{
  "@context": ["https://www.w3.org/2018/credentials/v1"],
  "type": ["VerifiableCredential", "HouseholdCredential"],
  "issuer": "did:amatelus:city-hall-tokyo",
  "issuanceDate": "2025-10-14T00:00:00Z",
  "credentialSubject": {
    "id": "did:amatelus:citizen123...",
    "myNumber": "encrypted_my_number",
    "name": "Taro Tanaka",
    "address": "Tokyo, Shinjuku...",
    "birthDate": "1990-01-01"
  },
  "proof": {
    "type": "Dilithium2Signature2025",
    "verificationMethod": "did:amatelus:city-hall-tokyo#key-1",
    "proofValue": "..."
  }
}
```

Critical design:

- National ID is encrypted in VC's data field
- Wallet decrypts only for AHI generation (when required)
- Services not requesting AHI never see the encrypted data

14.3.4 Zero-Knowledge Proof for AHI

```
structure AnonymousHashZKP where
  -- Public inputs
  publicInputs : {
    anonymousHashId : ByteArray        -- AHI
    categoryId : AuditCategoryId       -- Audit section ID
    vcIssuer : ValidDID                -- Household VC issuer DID
  }
  -- Secret inputs (used only within ZKP)
  witness : {
    myNumber : String                  -- National ID
    householdVC : HouseholdVC          -- Household VC
    vcSignature : Signature            -- VC signature
  }
  -- ZKP proof data
  proof : ZKProof
  deriving Repr
```

Proof contents verified within ZKP circuit:

1. `householdVC.proof` is valid (issued by municipality)
2. `myNumber` matches `householdVC.credentialSubject.myNumber`
3. `anonymousHashId = Hash(categoryId.id || myNumber)` computation is correct
4. `vcIssuer` matches `householdVC.issuer`

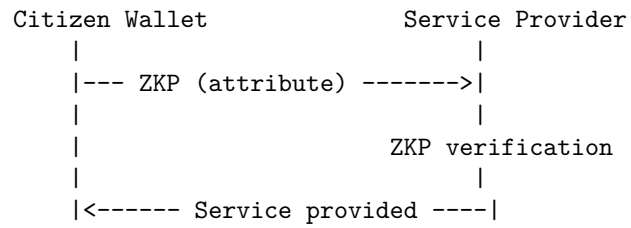
14.4 Audit Flows

14.4.1 Standard Flow: Without AHI (Privacy-Preserving Mode)

Most services use this flow. AHI is not required.

1. Citizen presents attribute proofs via ZKP (no personal information)
2. Service verifies ZKP
3. Service provides service

Sequence:



In this flow:

- National ID is never used
- No hash identifiers created
- Complete privacy is maintained

14.4.2 Audit-Required Flow: With AHI

Only services explicitly requiring AHI use this flow.

Preconditions:

- Service provider explicitly requires AHI
- Citizen resides in country with national ID system
- Citizen possesses national ID VC

Flow:

1. Citizen presents ZKP (attribute proof), AHI, and ZKP(AHI validity)
2. Service verifies ZKP and AHI validity
3. Service records AHI (not national ID)

4. Service provides service

Sequence:

```
Citizen Wallet              Service Provider
    |                             |
    |--- ZKP + AHI + ZKP(AHI) ->|
    |                             |
    |                       ZKP verification
    |                       AHI validation
    |                       AHI recording
    |                             |
    |<------ Service provided ----|
```

Critical property: **National ID remains non-disclosed (secret input within ZKP)**.

14.4.3 Audit Section ID Generation and Publication

This step is performed only for audit-required services.

1. Government/audit authority defines audit purpose
2. Generates unique audit section ID
3. Publishes to public registry
4. Service providers requesting AHI obtain the ID
5. Service notifies citizens that AHI is required

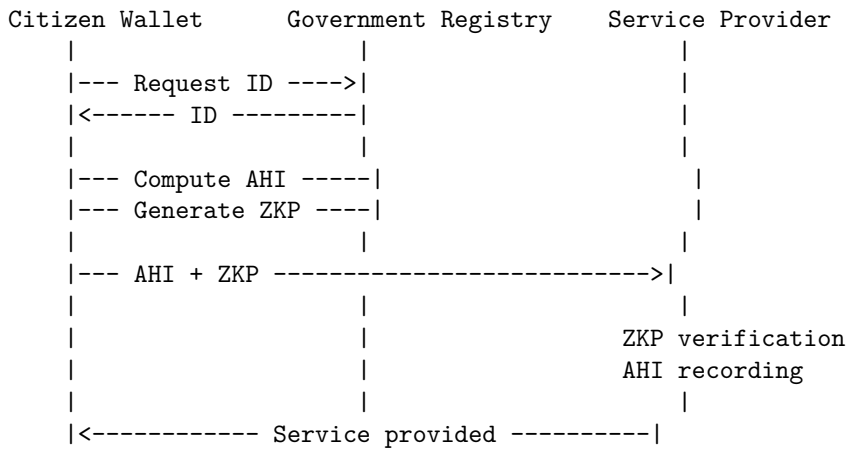
Examples:

- Government services: `"tax-2025"`, `"subsidy-education-2025"`
- Private services: `"sns-service-x"`, `"ticket-sales-y"`

14.4.4 AHI Generation Flow

1. **Wallet obtains national ID** from household VC
2. **Wallet obtains audit section ID** from government registry
3. **Wallet computes**: AHI = $H(\text{AuditCategoryID} \parallel \text{NationalID})$
4. **Wallet generates ZKP** proving valid hash without revealing national ID
5. **Wallet presents** AHI and ZKP to service provider
6. **Service verifies** ZKP and records AHI

Sequence diagram:



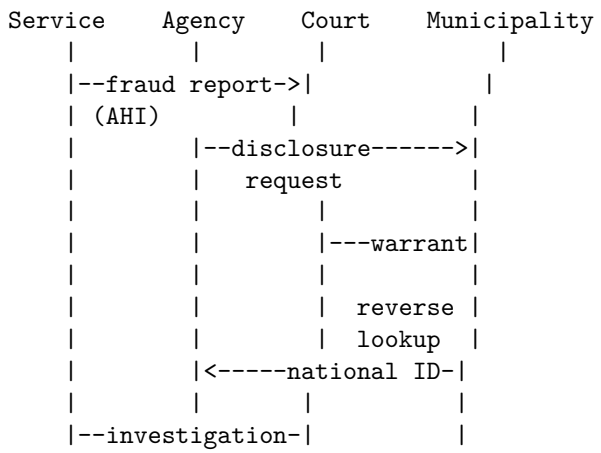
14.4.5 Fraud Investigation and Audit Flow

1. **Fraud detection**: Service provider detects suspicious activity
2. **Report**: Service identifies corresponding AHI
3. **Legal request**: Investigating agency requests disclosure
4. **Court review**: Court verifies legitimacy and issues warrant
5. **Reverse lookup**: Municipality matches AHI to national ID
6. **Disclosure**: Municipality reveals national ID to agency
7. **Investigation**: Agency identifies and investigates individual

Reverse lookup algorithm:

```
For each nationalID in municipality_database:
  if Hash(auditSectionID || nationalID) == suspicious_AHI:
    return nationalID  // Individual identified
```

Sequence diagram:



14.5 Private Sector Applications

14.5.1 Multi-Account Prevention

Services that may require AHI:

- Social networks
- Ticket sales platforms
- Online gaming
- Marketplace fraud prevention

Account registration flow:

1. Service generates and publishes service-specific identifier
2. Service notifies citizens: "AHI required for registration"
3. Citizen computes AHI from service ID and national ID
4. Service verifies AHI validity via ZKP
5. Service checks AHI uniqueness in database:
    - Duplicate found ⇒ Reject (multi-account prevention)
    - No duplicate ⇒ Accept and record AHI

Account ban enforcement:

- Service terminates account for policy violation
- Service blacklists corresponding AHI
- If same person attempts re-registration: Same national ID ⇒ Same AHI
- Service queries blacklist ⇒ Match found ⇒ Registration denied
- Result: Re-registration is prevented

14.6 Financial Institution Applications

14.6.1 Account Opening with AHI

Preconditions:

- Institution legally requires AHI (compliance mandate)
- Customer has national ID VC

Flow:

1. Institution generates purpose-specific identifiers
2. Institution notifies: "AHI required for compliance"

3. Customer computes AHI
4. Institution verifies AHI via ZKP
5. Institution links AHI to customer information (not national ID)
6. System enables cross-institutional AML tracking using AHI

Benefits:

- Money laundering prevention
- Traceability of high-value transactions
- Reduced national ID exposure risk
- Cross-bank transparency for regulatory compliance

14.7 Security Requirements

14.7.1 Hash Function

- **Requirement**: Post-quantum cryptography resistant
- **Recommended**: SHA-3, BLAKE3
- **Properties**:
  - One-way (preimage resistance)
  - Collision resistance
  - Second preimage resistance

14.7.2 Zero-Knowledge Proof

- **Requirement**: PQC-resistant ZKP scheme
- **Proof contents**:
  - Household VC validity
  - National ID possession
  - Hash computation correctness
- **UX consideration**: Pre-computation for minimal response time

14.7.3 Household VC Management

- Encrypted storage in wallet
- National ID tampering prevention
- VC issuer signature verification

14.7.4 Municipality National ID Management

- Warrant legitimacy verification
- Access logging for all reverse lookups
- Unauthorized access prevention
- Database encryption
- Multi-factor authentication for staff

14.7.5 Audit Section ID Management

- ID uniqueness guarantee
- Public registry verifiability
- ID misuse prevention (rate limiting)

14.8 Privacy Protections

14.8.1 Anti-Linkage Across Services

Scenario	**Prevention Mechanism**
Different government services	Different audit section IDs ⇒ Different hashes
Same service, multiple users	Same AHI per person, not linkable across services
Different private services	Different service IDs ⇒ Different hashes

14.8.2 National ID Non-Disclosure

- Citizens never directly present national ID
- Service providers cannot learn national ID
- Only municipalities and investigating agencies handle national IDs

14.8.3 Evasion Prevention

- Even if citizen discards secret keys, audit is still possible
- AHI is deterministically derived from national ID
- Municipality can always reverse-lookup the AHI
- Intentional evasion is technically impossible

14.9 Legal and Regulatory Requirements

14.9.1 Warrant Issuance Standards

- Reasonable suspicion of fraud
- Investigative necessity and proportionality
- Privacy violation minimization

14.9.2 Disclosure Procedure Transparency

- Warrant issuance records
- Municipality disclosure execution logs
- Post-disclosure citizen notification (where legally permitted)

14.9.3 Social Consensus Formation

- Clear definition of audit-required services
- Appropriate audit section design and granularity
- Legal interpretation of national ID hash usage

14.10 Implementation Guidelines

14.10.1 Wallet Implementation

Required Functions

```
class AuditWallet where
  -- Household VC storage
  storeHouseholdVC : HouseholdVC → IO Unit

  -- AHI generation
  generateHashId : AuditCategoryId → IO ByteArray

  -- ZKP generation
  generateZKP : AuditCategoryId → IO AnonymousHashZKP

  -- Secure national ID storage (encrypted)
  secureStore : EncryptedData → IO Unit
```

Security Requirements

- **National ID encryption**: Use device secure storage (Keychain, TEE)
- **Hash generation UX**: One-click generation
- **ZKP pre-computation**: Minimize response time via offline generation
- **Audit history**: Local logging of when and which audit section identifiers were used

14.10.2 Service Provider Implementation

Database Schema

```
CREATE TABLE anonymous_hash_identifiers (
  hash_id BYTEA PRIMARY KEY,
  audit_category_id VARCHAR(255) NOT NULL,
  first_seen_at TIMESTAMP NOT NULL,
  service_data JSONB,
  is_blacklisted BOOLEAN DEFAULT FALSE,
  INDEX idx_category (audit_category_id),
  INDEX idx_blacklist (is_blacklisted)
);
```

ZKP Verification

```
def verifyServiceRequest
    (hashId : ByteArray)
    (zkp : AnonymousHashZKP)
    (categoryId : AuditCategoryId) : IO (Result ServiceToken) := do
  -- 1. ZKP verification
  if !verifyAnonymousHashZKP zkp then
    return .error "ZKP verification failed"

  -- 2. Audit section consistency
  if zkp.publicInputs.categoryId  categoryId then
    return .error "Category mismatch"

  -- 3. Duplicate check
  if ← isDuplicate hashId then
    return .error "Duplicate registration"

  -- 4. Blacklist check
  if ← isBlacklisted hashId then
    return .error "Blacklisted hash"

  -- 5. Issue service token
  issueServiceToken hashId
```

Performance Optimization

- Parallel ZKP verification
- Short-term ZKP verification caching
- Index optimization on `hash_id`, `audit_category_id`, `is_blacklisted`

14.10.3 Municipality Implementation

Reverse Lookup

```
def reverseLookup
```

```
    (hashId : ByteArray)
    (categoryId : AuditCategoryId)
    (warrant : Warrant) : IO (Option String) := do
  -- 1. Warrant validity verification
  if !verifyWarrant warrant then
    return none

  -- 2. Access logging
  logAccess warrant hashId categoryId

  -- 3. Reverse lookup in national ID database
  for myNumber in ← getAllMyNumbers do
    let computed := generateAnonymousHashId categoryId myNumber
    if computed = hashId then
      -- 4. Disclosure logging
      logDisclosure myNumber hashId warrant
      return some myNumber

  return none
```

Security Requirements

- Warrant management system integration with real-time verification
- Complete access logging for all reverse lookups
- National ID database encryption at rest
- Multi-factor authentication and role-based access control

14.11 Formal Verification

14.11.1 Theorem: Anti-Linkability Across Audit Domains

```
theorem different_categories_prevent_linkability
    (myNumber : String)
    (cat1 cat2 : AuditCategoryId)
    (h : cat1.id  cat2.id) :
  generateAnonymousHashId cat1 myNumber
  generateAnonymousHashId cat2 myNumber := by
  -- By hash function collision resistance,
  -- different inputs produce different outputs
  apply Hash.collision_resistance
  simp [generateAnonymousHashId]
  exact String.append_ne_of_prefix_ne h
```

Result: Linking AHIs across services is computationally infeasible.

14.11.2 Theorem: Evasion Prevention

```
theorem audit_always_possible
    (citizen : Citizen)
    (myNumber : String)
    (cat : AuditCategoryId)
    (h1 : citizen.householdVC.credentialSubject.myNumber = myNumber) :
   (hashId : ByteArray),
    hashId = generateAnonymousHashId cat myNumber
    canReverseLookup hashId myNumber cat := by
  -- As long as municipality maintains national ID database,
  -- reverse lookup is always possible
  exists generateAnonymousHashId cat myNumber
  constructor
  · rfl
  · apply municipality_has_database
    exact h1
```

Result: Audit is always possible via legal procedure.

14.11.3 Theorem: Privacy Preservation

```
theorem zkp_preserves_privacy
    (zkp : AnonymousHashZKP)
    (adversary : Adversary) :
  computationallyInfeasible
    (adversary.extractMyNumber zkp) := by
  -- By ZKP soundness property,
  -- secret inputs cannot be extracted from proof
  apply ZKP.knowledge_soundness
  apply ZKP.zero_knowledge_property
```

Result: Service providers and third parties cannot learn national IDs.

14.11.4 Cryptographic Security Summary

Property	Cryptographic Basis	Security Level
Anti-linkability	Hash collision resistance	Computational (128-bit)
Evasion prevention	One-way function property	Information-theoretic
Privacy protection	ZKP knowledge soundness	Computational (128-bit)
Hash correctness	ZKP completeness	Mathematical proof

14.12 Future Research Directions

14.12.1 Technical Challenges

- PQC ZKP optimization (proof generation time and size)
- UX improvement for hash and ZKP generation

- Scalability for large-scale reverse lookups

14.12.2 Institutional Challenges

- Clear definition of audit-required services
- Legal framework for national ID hash usage
- Appropriate audit section granularity design

14.12.3 Standardization

- AMATELUS protocol detailed specifications
- Interoperability with other SSI systems
- International standardization (W3C, ISO)

Key Design Principles (Audit Mechanism)

1. **Optionality**: AHI is used only when required by service
2. **Privacy by default**: Services not requiring AHI provide complete privacy
3. **Service-specific identifiers**: Each service gets different hash, preventing profiling
4. **Legal alignment**: Audit is possible only via court warrants and legal procedure
5. **Transparency**: All reverse lookups are logged and auditable
6. **Citizen control**: Citizens retain cryptographic control over their identities
7. **Non-repudiation**: Service providers cannot deny or forge audit records

Chapter 15

Secret Key Lifecycle Management

Definition 27. This chapter covers secret key lifecycle management aspects of AMATELUS.

15.1 Overview

This chapter specifies the management of secret keys in AMATELUS, particularly emergency procedures for: device theft, loss, damage, device replacement, account rotation, death, and guardianship.

15.1.1 Key Principles

1. **Determinism Principle**: Only operations with secret key signatures are cryptographically deterministic
2. **Contactability Limit**: Only reachable Issuers can respond; unreachable Issuers cannot help
3. **Mathematical Safety Limit**: Safe recovery of DIDs without identity credential linkage is impossible

15.1.2 Scenario Classification

Secret key issues fall into five categories:

1. **Leakage**: Secret key in hand, possibly compromised to third party
2. **Loss with Abuse Concern**: Suspected theft or compromise (backup available)
3. **Loss without Abuse Concern**: Device damage/loss without compromise risk
4. **Update**: Planned key rotation (both old and new keys available)
5. **Death**: Permanent loss of physical access
6. **Guardianship**: Capacity-restricted individual under legal guardianship

15.2 Secret Key Leakage Response Flow

15.2.1 Preconditions

- **Old secret key available**: Holder still possesses possibly-compromised key
- **Signature operations possible**: Can cryptographically sign revocation requests
- **Third party compromise suspected**: Key copy may exist with attackers

15.2.2 Response Procedure

Step 1: Emergency Response Initiation

1. Holder generates new secret key pair $(newDID, newSK)$
2. Holder creates revocation signature using old key:
    - $signatureByOldKey = \text{Sign}(oldSK, H(oldDID \parallel newDID \parallel timestamp))$

Step 2: Revocation Request to All Issuers

Holder sends to **all Issuers** that issued VCs:

```
{
  "requestType": "revocationDueToLeakage",
  "oldDID": "did:amatelus:old123...",
  "newDID": "did:amatelus:new456...",
  "timestamp": "2025-10-14T10:30:00Z",
  "revocationReason": "Secret key leakage suspected",
  "signatureByOldKey": "signature(oldSK, hash(oldDID || newDID || timestamp))"
}
```

Cryptographic guarantee:

- `signatureByOldKey` cryptographically proves rightful old key owner
- Third party cannot forge signature (Dilithium2 security)

Step 3: Issuer Processing

Each Issuer executes:

1. Verify `signatureByOldKey` (reject if verification fails)
2. Add all VCs issued for `oldDID` to revocation list
3. Update and publish new Merkle Root
4. Permanently reject new VC requests from `oldDID`
5. (Optional) Re-issue VC for `newDID` following policy

Step 4: Revocation Completion Verification

Holder confirms:

1. Check each Issuer's MerkleRevocationList (VCs revoked)
2. List unreachable Issuers (unrevoked VCs remain)

Timeline

```
T=0:      Leakage detection
T=+5min:  New key generation, revocation signature
T=+30min: Revocation requests sent to all Issuers
T=+1h:    Issuer revocation processing complete (target SLA)
T=+24h:   Revocation confirmation, re-issuance begins
```

Security Characteristics

Property	Guarantee Level	Basis
Old VC revocation	**Deterministic**	Cryptographic signature
Old DID rejection	**Deterministic**	Issuer blacklist
Unreachable Issuers	**None**	No contact possible
Third-party misuse	**Time-dependent**	Issuer revocation speed

15.3 Secret Key Loss: Abuse Concern

15.3.1 Scenario

- Device theft, malware infection, or compromise suspected
- Old secret key unavailable, but backup recovery possible

15.3.2 Response

If backup recovery possible: Execute Section 3 (leakage response)
If no backup: Proceed to Section 5 (trust inheritance flow)

15.4 Secret Key Loss: Trust Inheritance Flow

15.4.1 Preconditions

- Old secret key unavailable (device lost/damaged)
- No abuse concern (device not stolen)
- New secret key pair generated

15.4.2 Case A: Identity Credential Available

Trust Inheritance Principle

```
Old DID (oldDID)
    Identity VC (e.g., driver's license VC)
        Issuer = Police/Municipality
        Linked to real-world identity document

↓ Trust Inheritance

New DID (newDID)
    DID Migration VC (DIDMigrationVC)
        Issuer = Police/Municipality
        Proves same-person relationship
```

Response Procedure

Step 1: Real-World Identity Verification at Municipality
Holder presents:

1. Physical identity document (driver's license, national ID card, etc.)
2. New DID (newDID)
3. Old DID information if available

Step 2: Issuer Record Verification
Issuer checks:

1. Physical identity document authenticity
2. Past VC issuance records
3. Link between document and oldDID

Step 3: DID Migration VC Issuance

```
{
  "@context": ["https://www.w3.org/2018/credentials/v1"],
  "type": ["VerifiableCredential", "DIDMigrationCredential"],
  "issuer": "did:amatelus:police-station-tokyo",
  "issuanceDate": "2025-10-14T11:00:00Z",
  "credentialSubject": {
    "id": "did:amatelus:new456...",
    "oldDID": "did:amatelus:old123...",
    "migrationType": "trustInheritance",
    "migrationReason": "Device damage causing key loss",
    "verificationMethod": "Physical identity document verification",
    "identityDocument": {
      "type": "DriversLicense",
      "documentNumber": "123456789012",
      "issuerName": "Tokyo Public Safety Commission",
```

```
      "verifiedAt": "2025-10-14T10:30:00Z"
    }
  },
  "proof": {
    "type": "Dilithium2Signature2025",
    "verificationMethod": "did:amatelus:police-station-tokyo#key-1"
  }
}
```

Step 4: Re-issuance from Other Issuers
Holder presents to other Issuers:

1. DID Migration VC
2. New ZKP generated with newDID

Issuer judgment: Accept re-issuance following policy

Security Characteristics

Property	Level
Identity proof	High (physical document verification)
DID migration legitimacy	High (trusted Issuer proof)
Other Issuers acceptance	Policy-dependent
Old VC revocation	Impossible (no old key)

15.4.3 Case B: No Identity Credential

Problem

- No cryptographic signature capability (old key lost)
- No real-world identity link in system
- Issuer cannot verify "who" was the oldDID owner

Mathematical Safety Limit Theorem:

Safe recovery of DIDs without identity credential linkage is information-theoretically impossible.

Issuer Policy Examples

Each Issuer must develop independent policy:
University Transcript VC:

- Present student ID at window
- Photo identification
- Cross-reference past issued transcripts

- Re-issue for new DID

Online Service Member VC:

- Confirmation code to registered email
- Security questions
- Cross-reference usage history
- Re-issue for new DID

Anonymous Service VC:

- No identity verification possible
- Re-issuance impossible
- Re-register as new user

Practical Conclusion

DIDs without identity-credential linkage are effectively unrecoverable if lost.

Preventive strategies:

1. Always obtain identity credential first
2. Maintain secret key backups in multiple locations
3. Use multi-device support
4. Perform periodic key updates

15.5 Planned Secret Key Update Flow

15.5.1 Preconditions

- Both old and new secret keys available
- Both key operations possible
- Planned update (not emergency)

15.5.2 Use Cases

1. Device replacement (new smartphone)
2. Multi-device support (PC and smartphone)
3. Periodic key rotation (security policy)
4. Backup device preparation

15.5.3 Procedure

Step 1: Dual-Key Relation Proof

1. Generate new secret key pair $(newDID, newSK)$
2. Create dual-signature relation proof:
   - $sig_{old} = \text{Sign}(oldSK, H(oldDID \parallel newDID \parallel timestamp))$
   - $sig_{new} = \text{Sign}(newSK, H(newDID \parallel oldDID \parallel timestamp))$

Step 2: DID Update Notification

```
{
  "requestType": "didUpdate",
  "oldDID": "did:amatelus:old123...",
  "newDID": "did:amatelus:new456...",
  "timestamp": "2025-10-14T10:30:00Z",
  "updateReason": "Device replacement",
  "transitionPeriod": "30days",
  "signatureByOldKey": "...",
  "signatureByNewKey": "...",
  "requestedAction": "issueNewVCWithNewDID"
}
```

Cryptographic guarantee:

- Both signatures prove `oldDID` and `newDID` owner is same person

Step 3: Issuer Processing (Flexible Options)

Option A: Issue new VC with new DID

- New VC with `newDID` as subject
- Keep old VC valid

Option B: Extend old VC validity

- Set transition period
- Next update issues new VC

Option C: Issue DID Update Credential

- Prove `oldDID` to `newDID` relation
- Allow other Verifiers to verify link

Step 4: Transition Period Management

```
Transition period (e.g., 30 days):
- Both old and new DIDs valid
- Holder can use either

After transition:
- Old DID VC revoked (optional)
- New DID primary
```

Security Characteristics

Property	Level
Old-new DID relation	Deterministic (dual signature)
New VC issuance	Deterministic (Issuer signature)
Old VC continuation	Issuer policy-dependent
Multi-device support	Possible (device-specific DIDs)

15.5.4 Multi-Device Scenario

Recommendation: Each device gets distinct DID

```
Device A (Smartphone):
  DID: did:amatelus:user123-mobile
  SK: SK_mobile

Device B (PC):
  DID: did:amatelus:user123-pc
  SK: SK_pc

Approach:
1. Assign unique DID per device
2. Use Section 5 procedure for each device
3. Request VCs from each Issuer for both DIDs
4. Identify with Issuer: "Same person, different devices"
5. Dual-signature proves same-person relationship
```

15.6 Death of Holder

15.6.1 Key Characteristics

- Secret key permanently inaccessible
- Misuse risk essentially zero (key cannot be used)
- Procedure depends on family status
- VC revocation incomplete but damages minimal

15.6.2 Case A: No Family

Municipal automatic processing:

1. Death registration at municipality
2. Link with VC issuance records via identity document
3. Add holder's DIDs to municipality MerkleRevocationList
4. (Optional) Issue death certificate VC

Issuer Processing Limitations

- Municipality Issuers: Can revoke (access to death records)
- Private Issuers: Cannot revoke (no access to death records)

Security Impact:

- Private Issuer VCs remain valid
- But key is permanently unusable
- Misuse risk effectively zero

15.6.3 Case B: Family Present

Family-assisted processing:

1. Municipality issues legal representative VC (LegalRepresentativeCredential)
2. Family presents to each Issuer with representative VC
3. Issuer processes based on policy
4. Limited effectiveness (often not executed)

Security Impact:

- Some Issuers may revoke
- Unrevoked VCs remain
- Misuse essentially impossible (no key access)
- Damages minimal compared to leakage scenarios

15.7 Guardianship (Capacity Restriction)

15.7.1 Scenario

Holder becomes capacity-restricted (legal guardianship):

- Individual is alive and retains device access
- Judgment capacity is legally restricted
- Guardian has legal authority

Difference from death:

```
Death:
- Secret key access: Impossible → Misuse risk zero

Guardianship:
- Secret key access: Possible (person alive) → Misuse risk exists
- Problem: Individual may sign inappropriate contracts
- Solution: Guardian can revoke/cancel contracts
```

15.7.2 Guardianship Initiation Flow

Family Court Guardianship Order

1. Application to family court (family member, prosecutor, etc.)
2. Family court review of capacity
3. Capacity evaluation
4. Guardianship order and guardian appointment

Municipality Guardian VC Issuance

```
{
  "@context": ["https://www.w3.org/2018/credentials/v1"],
  "type": ["VerifiableCredential", "GuardianCredential"],
  "issuer": "did:amatelus:city-hall-tokyo",
  "issuanceDate": "2025-10-14T14:00:00Z",
  "credentialSubject": {
    "id": "did:amatelus:guardian-789...",
    "guardianOf": "did:amatelus:ward-123...",
    "guardianType": "   ",
    "authority": [
      "VC revocation request",
      "DID blacklist registration",
      "Contract cancellation",
      "Property management"
    ],
    "wardInfo": {
      "courtDecisionDate": "2025-10-01",
      "courtCaseNumber": "Reiwa 7 (Fam) No. 12345",
      "registrationNumber": "Reiwa 7 Guardianship No. 67890",
      "guardianshipStartDate": "2025-10-01"
    },
    "verificationMethod": "Court decision and registration certificate",
    "expirationDate": "2026-10-14T23:59:59Z"
  }
}
```

Key fields:

- `guardianOf`: Ward's DID
- `authority`: Authority types
- `courtCaseNumber`: Legal case reference
- `registrationNumber`: Guardianship registry number

15.7.3 Guardian VC Notification to Issuers

Guardian notifies all Issuers:

1. Present Guardian VC
2. Request VC revocation
3. Request DID blacklist

Issuer processing options:
Option A: Complete revocation

- Add to MerkleRevocationList
- Add DID to blacklist

Option B: Conditional validity

- Mark as "under guardianship"
- Verifier requires guardian approval
- Small transactions allowed

Option C: Defer to Verifier

- No Issuer action
- Verifier checks guardianship

15.7.4 Verifier Guardianship Confirmation Flow

Recommended at contract time:

1. Receive ZKP from Holder
2. Check MerkleRevocationList (standard)
3. **New**: Query guardianship status
    - Call municipal guardianship confirmation API
    - Check if DID is under guardianship

Guardianship confirmation API:

```
GET /api/v1/guardianship/check/{did}

Response:
{
  "did": "did:amatelus:ward-123...",
  "underGuardianship": true,
  "guardian": {
    "did": "did:amatelus:guardian-789...",
    "type": "Legal Guardian",
    "guardianVC": {
```

```
      "issuer": "did:amatelus:city-hall-tokyo",
      "expiresAt": "2026-10-14T23:59:59Z"
    }
  },
  "restrictions": [
    "No new VC issuance",
    "Large contracts require guardian approval",
    "Small transactions (under 10,000 yen) permitted"
  ]
}
```

15.7.5 Contract Cancellation by Guardian

If ward makes inappropriate contract:

Legal Basis

Civil Code Article 9: "A person under guardianship may cancel their legal acts."

Cancellation Procedure

1. Guardian identifies inappropriate contract
2. Presents Guardian VC to Verifier
3. Requests contract cancellation and refund
4. Verifier verifies Guardian VC and executes cancellation

Verifier verification:

1. Verify Guardian VC signature
2. Check `authority` includes "contract cancellation"
3. Verify contract was with ward's DID
4. Execute cancellation and refund

15.8 Security Considerations

15.8.1 Key Leakage Attacks

Attack: Attacker uses leaked key to generate ZKP
Defense:

- Holder immediately revokes old DIDs
- Issuer revokes old VCs
- Verifier checks current MerkleRevocationList

Vulnerability window: Time between leak and revocation completion

15.8.2 Fraudulent DID Migration

Attack: Attacker claims old DID as their own for re-issuance
Defense:

- With old key signature: Signature verification prevents fraud
- Without old key: Identity credential physical verification prevents fraud
- Without identity credential: No defense (information-theoretic impossibility)

15.8.3 Guardian VC Forgery

Attack: Attacker forges Guardian VC
Defense:

- Guardian VC issued by trusted municipality (cryptographic signature)
- Signature forgery requires breaking Dilithium2 (128-bit security)
- Recommended: Verify guardianship in legal registry API

15.9 Implementation Recommendations

15.9.1 Holder Responsibilities

- Store secret key in device secure storage (Keychain, TEE)
- Maintain encrypted backups in multiple locations
- Obtain identity credential before other VCs
- Update DIDs periodically (annually recommended)
- Keep Issuer contact list
- Test revocation procedures annually

15.9.2 Issuer Responsibilities

- Define and publish revocation SLA (e.g., 1 hour for high priority)
- Implement automatic signature verification
- Maintain up-to-date MerkleRevocationLists
- Provide 24/7/365 revocation acceptance (high SLA)
- Document DID migration trust policies
- Keep audit logs of revocation requests and completions

15.9.3 Verifier Responsibilities

- Always use current MerkleRevocationList
- Implement MAX_VERSION_LAG (recommended: 5)
- For guardianship scenarios: Query guardianship API
- Define transaction thresholds for guardian approval
- Process contract cancellation requests from guardians
- Log all guardian-related transactions

15.10 Formal Theorems

15.10.1 Theorem: Leakage Recovery Determinism

With old key signature capability, revocation is cryptographically deterministic.

```
theorem leakage_revocation_possible
    (holder : Holder)
    (oldSK : SecretKey)
    (oldDID : DID)
    (newDID : DID)
    (timestamp : Nat)
    (h_possession : holder.hasKey oldSK)
    (h_sign : CanSign oldSK (hash (oldDID ++ newDID ++ timestamp))) :
   (revocationProof : RevocationProof),
     (issuer : Issuer),
     issuer.hasVC oldDID →
       Verifiable revocationProof
       issuer.canRevoke oldDID revocationProof
```

15.10.2 Theorem: Recovery Impossibility Without Identity Credential

```
theorem recovery_impossible_without_identity_credential
    (holder : Holder)
    (oldDID : DID)
    (newDID : DID)
    (h_no_key : ¬holder.hasKey oldSK_old)
    (h_no_identity_vc : ¬ (vc : VC),
      vc.issuer.isIdentityAuthority
      vc.subject = oldDID) :
  ¬ (issuer : Issuer),
    issuer.canVerifyRecovery oldDID newDID
```

Result: Safe recovery is information-theoretically impossible.

Priority	Target	Maximum
High (leak/theft)	1 hour	24 hours
Medium (loss)	24 hours	7 days
Low (planned update)	7 days	30 days

15.11 Policy Recommendation Summary

15.11.1 Revocation SLA

15.11.2 Identity Verification Standards

- Leakage/Update: Signature verification (automatic)
- Loss with ID VC: Physical identity document + record linkage
- Loss without ID VC: Issuer-specific policy or impossible
- Death: Municipal record linkage
- Guardianship: Court order + family court registration verification

15.11.3 Key Design Principles

1. **Determinism when possible**: Use cryptography to avoid judgment
2. **Identity-first**: Obtain identity credential before other VCs
3. **Backup preparation**: Require users to backup keys and test recovery
4. **Transparency**: Publish all policies and procedures publicly
5. **Practical limits**: Accept that some scenarios are unrecoverable

Chapter 16

Conclusion

AMATELUS provides a clean separation of concerns:

- **Cryptographic layer**: AMATELUS ensures authentication and privacy
- **Service layer**: Service providers ensure authorization and delivery
- **User layer**: Users choose which services and DIDs to use

This architecture achieves:

- **Security**: Formal cryptographic guarantees
- **Scalability**: Linear throughput with users
- **Privacy**: Cross-service unlinkability
- **Simplicity**: Clear responsibility boundaries
- **Practicality**: No distributed infrastructure burden on users

The protocol enables centralized services to provide decentralized authentication—a pragmatic foundation for digital governance in the real world.