Capstone Project

Long-life Verifiable Credentials with Decentralized Identifiers

Professor : Kotaro Kataoka

Traditional Educational Credentials

- **Physical and Tangible (eg- Paper Diplomas):**
  - Physical documents can be easily lost, damaged or destroyed.
  - They are also susceptible to forgery and counterfeiting making it prone to fraud.

- **Problem- Issuer-Controlled Verification:**
  - To verify a credential, verifier (employer) must contact the issuing institution (university) directly.
  - Manually contacting an issuer to verify a credential is slow & inefficient.
  - If the issuing institution closes down, it becomes difficult to authenticate credential.

Example use case: Job Application The Traditional Way: Paper Diploma

Sarah (university graduate) is applying for a new job at a company. She need to verify her degree.

- **Verification is Tedious:**
  - Sarah has to manually provide a copy of her physical diploma to the company.
  - The employer then has to contact the university's registrar's office to verify its authenticity.
  - This process can take days or even weeks.
- **Inconvenience:**
  - Sarah has to carry a physical document or a photocopy of it.
  - If she loses it, she has to request an official replacement from her university, which can involve fees and waiting periods.
- **Risk of Fraud:** Paper diplomas are vulnerable to counterfeiting.

Solving problem with Verifiable Credentials (VCs)

Verifiable credentials solve previous issues with security & ease that paper credentials can't.

- **Instant Verification with vc:**
  - Sarah sends her digital diploma (a VC) from her digital wallet to the employer.
  - The employer's system can **instantly verify** the credential's authenticity and integrity.
  - This **eliminates the need to contact the university** and speeds up the process.

- **No Counterfeiting with vc:**
  - VCs are cryptographically signed by the issuer.
  - Any tamper with credential breaks the cryptographic signature, making it invalid.

- **Always Available with vc:**
  - The VC is stored securely on her wallet.
  - She can access it anytime and can back it up, eliminating the risk of physical loss.
  - If a digital file is lost, a new one can be re-issued by the university with little effort

Verifiable Credentials Advantages[1]

- **Recipient-Owned** credentials in a personal and secure digital wallet.
- **Recipient-Controlled** credentials over who they share their information with & when.
- **Instantly and Independently Verifiable** without ever needing to contact the issuer.
- **Secure and Tamper-Proof** with the help of cryptographic signatures
- **Globally Interoperable** on open international standards, works across borders seamlessly.
- **Durable and Long-Lasting due to** verification being independent of the issuer

[1] Clemens Brunner, Ulrich Gallersdörfer, Fabian Knirsch, Dominik Engel, and Florian Matthes. 2021. DID and VC:Untangling Decentralized Identifiers and Verifiable Credentials for the Web of Trust. In Proceedings of the 2020 3rd International Conference on Blockchain Technology and Applications (ICBTA '20). Association for Computing Machinery, New York, NY, USA, 61–66.

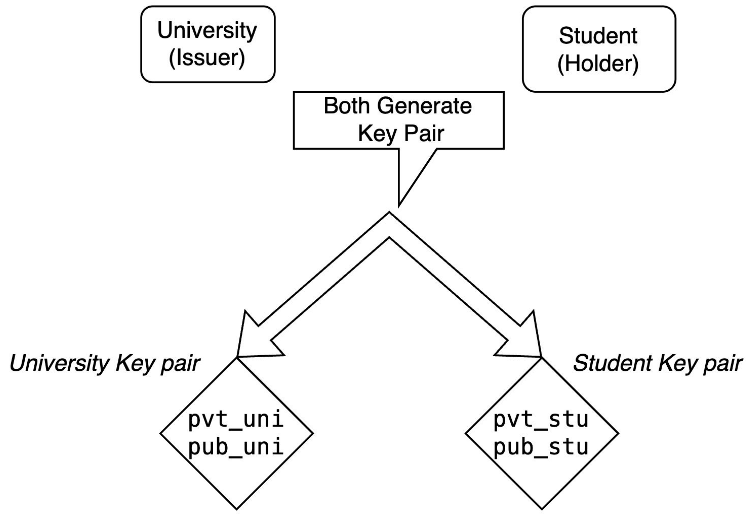
How Verifiable Credentials Work [1]

- The system working involves **3** key roles:
    - **The Issuer:** Creates & digitally signs a credential.
    - **The Holder:** Receives the signed credential & stores in secured wallet.
    - **The Verifier:** Verifies credential that is presented by the holder.
- **VC** contains a **set of claims about attributes**, the issuer wants to attribute to the receiver. (eg DOB)
- **Machine-readable** and **Cryptographically secured.**
- Created by the issuer (bound to a identifier), sent to receiver.
- In order to **forward a claim to a verifier**, a **presentation** (of vc) is created in front of the verifier.

[1] Clemens Brunner, Ulrich Gallersdörfer, Fabian Knirsch, Dominik Engel, and Florian Matthes. 2021. DID and VC:Untangling Decentralized Identifiers and Verifiable Credentials for the Web of Trust. In Proceedings of the 2020 3rd International Conference on Blockchain Technology and Applications (ICBTA '20). Association for Computing Machinery, New York, NY, USA, 61–66.

Traditional Verifiable credentials (Generate key pairs)

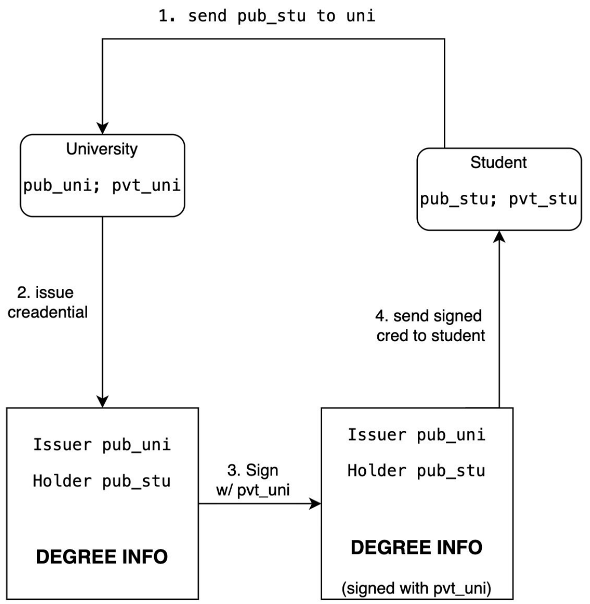
- University has to issue a **degree** to a student using **verifiable credentials**.
- For credentials to be verifiable, it has to contain the **identifiers** of both issuer & holder.
- Traditionally **public key** is used as identifiers.
- Hence both issuer & holder need to have the **key pairs**.



Traditional Verifiable credentials (Issuing credential)

- Student sends its public key **pub_stu** to university.
- University issues verifiable credential with hardcoding both public key identifiers of issuer & recipient in the vc.
- University signs the credential with its private key **pvt_uni** & sends to the student.
- Student stores the signed verifiable credential in a secured wallet.

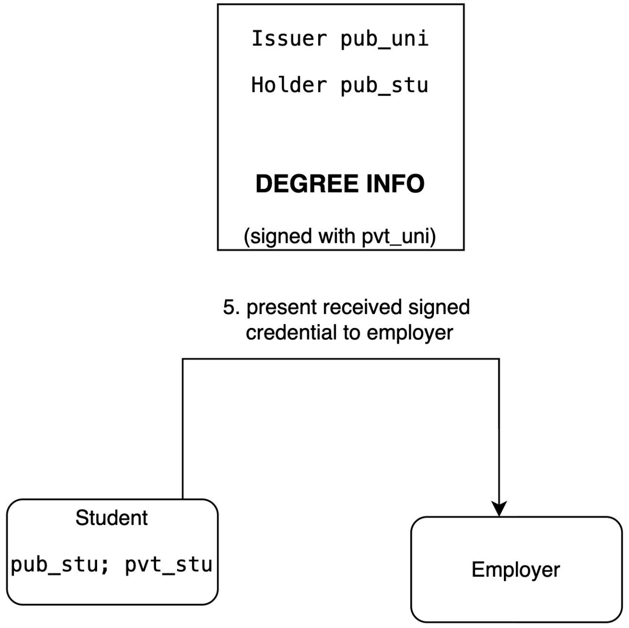
**Private keys are kept secured respectively.*



Traditional Verifiable credentials (Verifying credential auth)

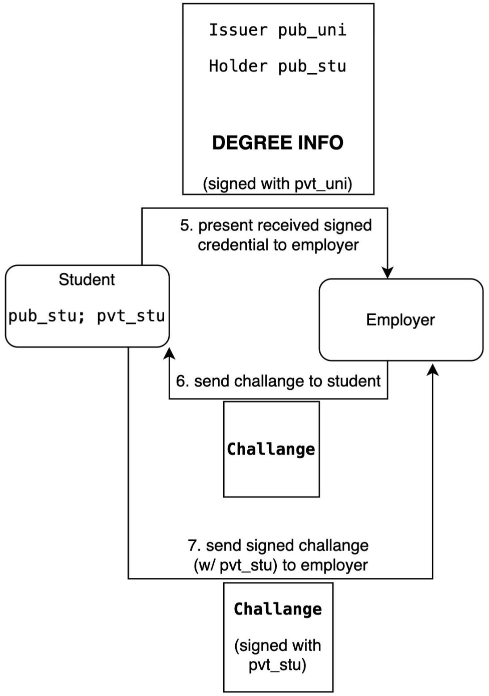
- When needed student can present the credential to some verifier (eg employer) from its secured wallet.

- Employer verifies the credential authenticity by verifyin the signature of the issuer with **pub_uni** (included in vc



Traditional Verifiable credentials (Verifying student ownership)

- To verify the student, employer sends a **challenge** to the student.
- Student signs the challenge with its private key **pvt_stu** & sends back to employer.
- Employer verifies signed challenge with public key of the student **pub_stu** (included in vc).



Problems with Traditional public-private key pair based identifiers

They present challenges mainly for **rotating keys**.

- With the public key embedded in the digitally signed credential, it is impossible to update the signing key.
- If the credential relies on a centralized key registry authority for managing rotation, then that registry becomes the centralized point of failure.
- If the credential's cryptographic technology becomes outdated, there is no way to update the credential to use a more robust technology;
- **In all above cases, a recipient must contact the issuer for reissuance of credential.**

Decentralised Identifiers (DIDs)

- DID: Globally unique reference linking to a DID document

  (did:<DID method>:<method-specific identifier>)

- Given a DID, we can retrieve the referenced DID Document (like URL).
- A DID method is a specific description of
  - how a DID is resolved in a particular blockchain or distributed ledger
  - how DID Documents are written and updated.
- The method-specific identifier allows to resolve the DID within that reference.
- A DID Document is a structure expressed in JSON
  - Contains information about the identity, such as public keys.
  - ALso includes references to service endpoints, where the issuer can operate certain services, such as a repository for VCs.

Root problem & Solution

The Problem: Changing Keys Over a Lifetime

- Over time, a person naturally changes its digital security methods, hence changing its cryptographic keys.
- If a credential is tied to a specific key that gets outdated, the owner can no longer use it.

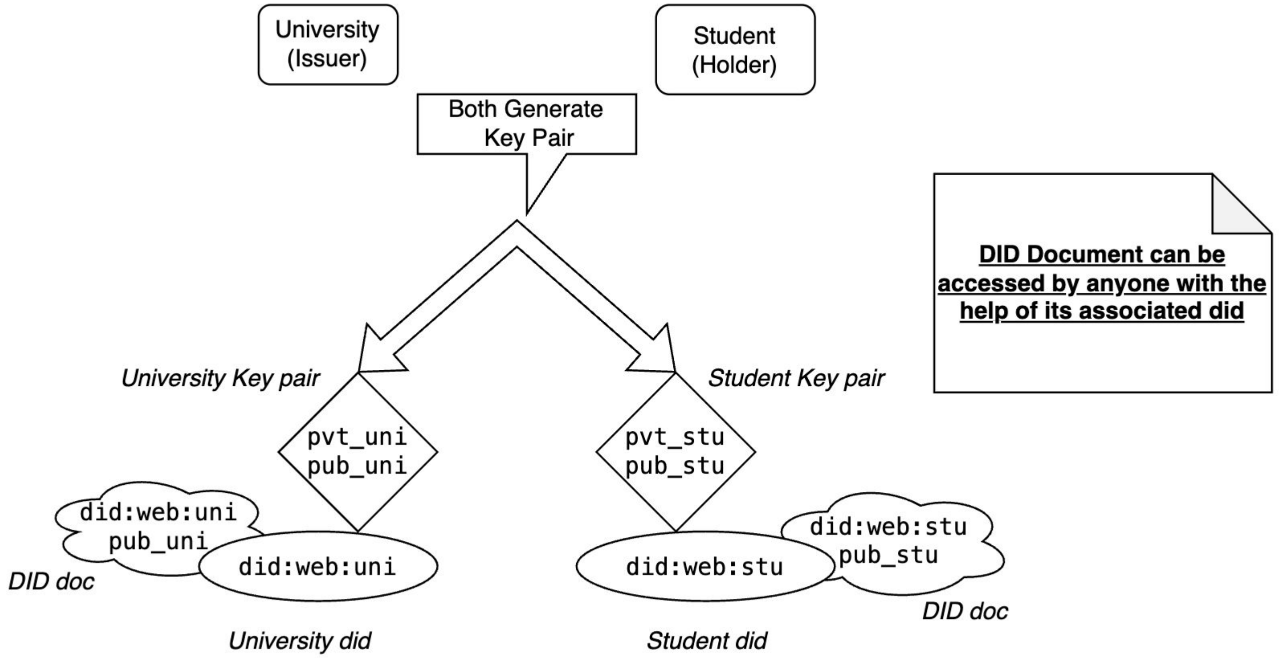
The Solution: Issuing to a Decentralized Identifier (DID)

- A credential is linked to the **owner's permanent DID**, not to a temporary key.
- The owner can update associated cryptographic keys with their DID at any time.
- **Result:** By issuing an educational credential to a recipient's DID, it gets ability to prove ownership of a credential even if the cryptographic material used for authenticating changes over time.

https://www.w3.org/TR/did-use-cases/#edu

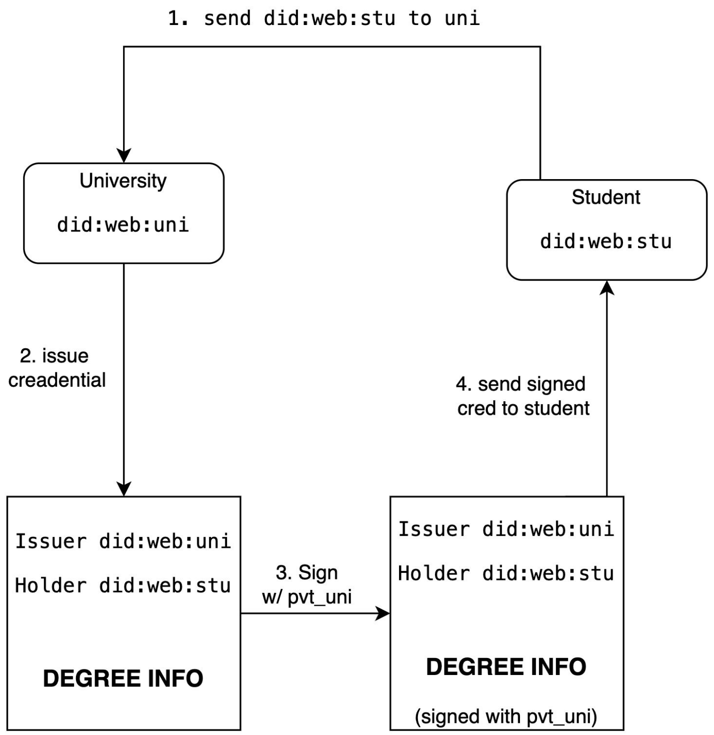
Generating & publishing DIDs

- Both university & student generate key pairs.
- Both create their unique DIDs.
- Embed their current public keys in their respective did doc.
- After publishing this did document, anyone can resolve the DID doc through the DID.



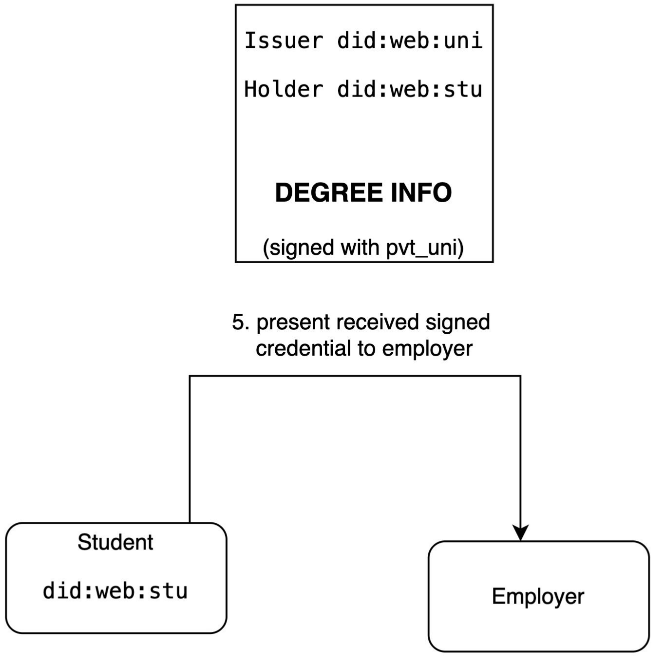
Issuing credential w/ did

- Instead of directly sending **pub_stu**, student sends its **did:web:stu** to the issuer
- Issuer issues credential (contains dids of issuer & holder) & signs it with **pvt_uni**
- Issuer sends signed verifiable credential to student



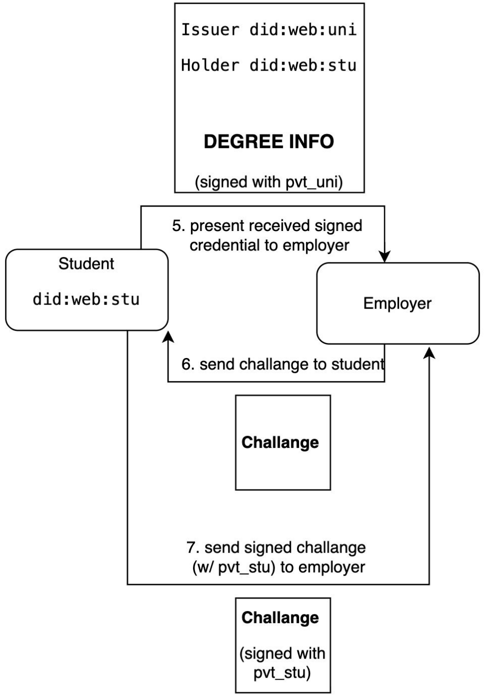
Verifying credential auth

- Student presents verifiable credential (signed by issuer with its **pvt_uni**) to employer
- Employee verifies the authenticity of credential with **pub_uni** (from did doc of **did:web:uni** mentioned in issued signed verifiable credential)



Verifying student ownership

- Verifier sends a challenge to holder
- Student signs the challenge with its **pvt_stu**
- Student sends signed challenge to verifier
- Verifier verifies signed challenge with **pub_stu** (obtained from did doc of **did:web:stu** mentioned in signed cred)



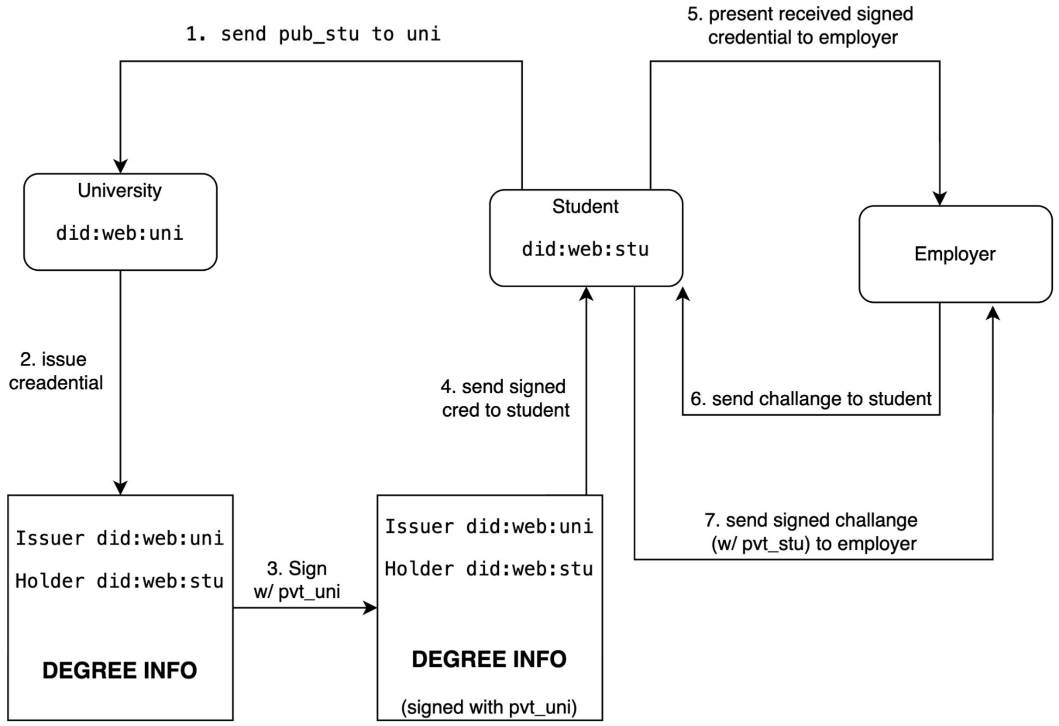
Handling Key rotation problem with DIDs

- If the private key **pvt_stu** of student gets lost or leaked, then the issued credential still remain valid because issued cred contains the did of student **did:web:stu** and student still owns did.
- To prove ownership of the credential, student signs challenge with its new **pvt_stu***
- And since the did doc of **did:web:stu** contains new **pub_stu***, verifier can access the new associated public key **pub_stu*** to the new private key **pvt_key*** with which the challenge was signed.

Verifiable Credential Example

```
{
  "@context": [
    "https://www.w3.org/2018/credentials/v1",
    "https://www.w3.org/2018/credentials/examples/v1"
  ],
  "id": "http://example.edu/credentials/1872",
  "type": ["VerifiableCredential", "UniversityDegreeCredential"],
  "issuer": "did:ion:EiD3j9kU2S...abcd",
  "issuanceDate": "2025-09-16T00:00:00Z",
  "credentialSubject": {
    "id": "did:example:ebfeb1f712ebc6f1c276e12ec21",
    "givenName": "Alice",
    "familyName": "Doe",
    "degree": {
      "type": "BachelorDegree",
      "name": "Bachelor of Science in Computer Science"
    },
    "college": "School of Engineering",
    "university": "Example University"
  },
  "proof": {
    "type": "Ed25519Signature2020",
    "created": "2025-09-16T00:00:00Z",
    "proofPurpose": "assertionMethod",
    "verificationMethod": "did:ion:EiD3j9kU2S...abcd#keys-1",
    "jws": "eyJhbGciOiJFZERTQSJ9..G0gdA78s..."
  }
}
```

Full working



Need for Key Rotation

- **Risks**:
  - Key **compromise** (hacked, stolen, leaked)
  - Devices storing keys being **lost**/damaged
  - Cryptographic algorithms becoming **outdated**

- **Without rotation**:
  - Attacker can **impersonate** the actual holder permanently
  - Hence the **DID** becomes **useless** if keys cannot be trusted

Security Guarantees of Key Rotation

- **Continuity**: DID remains valid even if keys change
- **Tamper-resistance**: Updates anchored on Bitcoin
- **Granularity**:
    - Update = minor rotation (Auth keys)
    - Recovery = major reset (all keys)
- **Attack Mitigation**: Prevents long-term impersonation & ID-stealing attacks

DID ⌬ **ION (Identity Overlay Network)** method

- ION maintains DIDs through **batched operations** anchored in **Bitcoin** using Sidetree protocol

- Provides:
  - **Scalability** - thousands of operations without bloating Bitcoin
  - **Trust** - inherits Bitcoin's immutability & decentralized trust
  - **Self-sovereignty** - no central authority manages identity

- ION supports 4 operations:
  - **Create** – Generates DID with initial keys
  - **Update** – Rotates authentication keys using **Update Key**
  - **Recover** – Restores DID control using **Recovery Key**
  - **Deactivate** – Permanently deletes DID https://identity.foundation/sidetree/spec/

Ion keys

Key	Stored	Goal	Feature
Authentication	Pvt - kept pvt Pub - in **DID Document** under verificationMethod	Higher chance of compromise, short-lived	Used for proving holder's identity, signing vc, authentication
Update	Pvt - kept pvt commitment Pub - in the **DID State** under updateCommitment	Changed if update key gets lost or compromised	Used for performing DID updates (like rotating auth key)
Recovery (Master key)	Pvt - kept pvt commitment Pub - in the **DID State** under recoverCommitment	Very critical, kept highly secured	A "master reset" key in case we lose update key or it's compromised.

Publish the DID (anchoring)

We build a **Create** operation for DID creation.

It includes:

(A) DID State (control metadata)

ION keeps commitments (hash) to update and recovery keys.

```
{
  "updateCommitment": "EiA9xyz123...",
  "recoveryCommitment": "EiB7lmn456..."
}
```

(B) DID Document (public-facing)

```
{
  "id": "did:ion:EiD7...abc",
  "verificationMethod": [
    {
      "id": "did:ion:EiD7...abc#auth-key-1",
      "type": "JsonWebKey2020",
      "publicKeyJwk": {
        "kty": "EC",
        "crv": "secp256k1",
        "x": "....",
        "y": "...."
      }
    }
  ],
  "authentication": [
    "did:ion:EiD7...abc#auth-key-1"
  ],
  "service": [
    {
      "id": "did:ion:EiD7...abc#msg",
      "type": "MessagingService",
      "serviceEndpoint": "https://example.com/messages/123"
    }
  ]
}
```

Build a patch for DID Document (Auth key rotation)

- **Pre-requisites:**
  - **New auth key pair** (from newly generated pair).
  - **Current update private key**
  - **Current DID Document** (resolved)
  - **Current DID State** (with updateCommitment)
- **Creating a patch:**

  eg: Given patch says -

  - add the new key for authentication
  - remove the old one.

```
"patches": [
  {
    "action": "add-public-keys",
    "publicKeys": [
      {
        "id": "new-auth-key-1",
        "type": "JsonWebKey2020",
        "publicKeyJwk": {
          "kty": "EC",
          "crv": "secp256k1",
          "x": "NEW_X_VALUE",
          "y": "NEW_Y_VALUE"
        },
        "purposes": ["authentication"]
      }
    ]
  },
  {
    "action": "remove-public-keys",
    "ids": ["old-auth-key-1"]
  }
]
```

Build the update operation

- **revealValue** ⟶ proves we know the public update key corresponding to the stored commitment.
- **delta.patches** ⟶ DID Document changes (auth key rotation).
- **delta.updateCommitment** ⟶ hash of a *new update key* (so the DID state will move forward).
- **signedData** ⟶ signed with **current update private key** to prove authorization.

```
{
  "type": "update",
  "didSuffix": "EiD7...abc",
  "revealValue": "EiA9xyz123...",   // hash of current update public key
  "delta": {
    "patches": [
      { "action": "add-public-keys", ... },
      { "action": "remove-public-keys", ... }
    ],
    "updateCommitment": "EiC9next789..."  // hash of *new* update public key
  },
  "signedData": "eyJhbGciOiJFUzI1NksifQ...." // JWS signed with current update private key
}
```

Submit to ION

- Submit Update Operation JSON, batched into Sidetree ION node
- Anchored in **Bitcoin block**; permanent, immutable record
- ION batches and anchors it in Bitcoin.
- Later, when the DID is resolved:
    - DID Document shows **new auth key**.
    - DID State points to the **new update commitment**.

```
{
  "updateCommitment": "EiA9xyz123...",
  "recoveryCommitment": "EiB7lmn456..."
}
```

```
{
  "updateCommitment": "EiC9next789...",    // moved forward
  "recoveryCommitment": "EiB7lmn456..."    // unchanged
}
```

The Role of Recovery Key

- If **Update Key is also compromised or lost** then use Recovery Key
- Recovery Operation replaces **all keys** with fresh ones
- Ensures DID control **cannot be permanently hijacked**
- Recovery Key should be **stored offline / highly protected**

Build the recover operation (Update key rotation)

Recovery operations **replace the DID's control entirely**.

- **Prerequisite:**
  - **New auth key pair** (from newly generated pair).
  - **Current recovery public key**
  - **Current recovery private key**
  - **Current DID State** (with recoveryCommitment)
  - New **update commitment** (hash of a new update key).
  - New **recovery commitment** (hash of new recovery key).
- Sign the operation with the **current recovery private key**.

```
{
  "type": "recover",
  "didSuffix": "EiD7...abc",
  "revealValue": "EiB7lmn456...",   // hash of current recovery public key
  "delta": {
    "patches": [
      {
        "action": "replace",
        "document": {
          "id": "did:ion:EiD7...abc",
          "verificationMethod": [
            {
              "id": "did:ion:EiD7...abc#new-auth-key-1",
              "type": "JsonWebKey2020",
              "publicKeyJwk": {
                "kty": "EC",
                "crv": "secp256k1",
                "x": "NEW_X",
                "y": "NEW_Y"
              }
            }
          ],
          "authentication": [
            "did:ion:EiD7...abc#new-auth-key-1"
          ]
        }
      }
    ],
    "updateCommitment": "EiU8newUpdateHash...",
    "recoveryCommitment": "EiR9newRecoveryHash..."
  },
  "signedData": "eyJhbGciOiJFUzI1NksifQ...." // signed with current recovery private key
}
```

Validation by ION

- ION checks the revealed **recovery public key** matches with the stored recoveryCommitment.
- It checks the operation is signed by the **current recovery private key**.
- If valid → the DID Document is replaced and the DID State is reset with new commitments.
- Finally we've:
  - **New auth key private key**
  - **New update private key**
  - **New recovery private key** (new master reset)

```
{
  "updateCommitment": "EiA9xyz123...",
  "recoveryCommitment": "EiB7lmn456..."
}
```

```
{
  "updateCommitment": "EiU8newUpdateHash...",   // brand new update key commitment
  "recoveryCommitment": "EiR9newRecoveryHash..." // brand new recovery key commitment
}
```