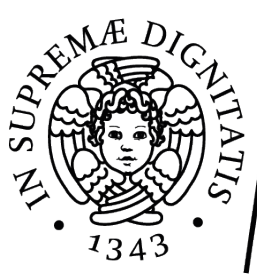


UNIVERSITÀ
DI PISA

University of Pisa

Department of Information Engineering

FOC project

Nicolò Mariano Fragale, Federico Rossetti

May 2025

Contents

1 DSS Analysis

System Objective An organization uses a digital signature service (Digital Signature Server, DSS) that acts as a trusted third party: it generates key pairs on behalf of employees, stores them, and produces digital signatures upon the user's request.

Initial registration (off-line) and credentials Users/employees are registered *off-line*. During registration they receive:

- the **public key of the DSS** (to be kept);
- an **initial password**, which *must be changed at the first access*.

Secure channel and authentication Before invoking any operation, the user must establish a **secure channel** with the DSS, which must satisfy the requirements of:

- **Perfect Forward Secrecy (PFS)**;
- **message integrity**;
- **protection from replay** (no-replay);
- **non-malleability**.

Authentication is carried out as follows:

- **server authentication** through its public key (known to the user);
- **user authentication** through their password.

Operations exposed by the service After establishing a secure connection and authentication, the DSS exposes the following application-level operations:

1. **CreateKeys**: creates and stores a *(private, public)* pair for the invoking user. If the pair *already exists*, the operation *has no effect* (idempotence).

   Precondition: active secure session; user authenticated.

   Postcondition: a key pair exists associated with the user (unchanged if already present).

2. **SignDoc(document)**: returns the **digital signature** of the document provided as argument; the DSS signs *on behalf of the invoking user*.

   Precondition: secure session; user authenticated; *key pair exists*.

   Postcondition: the digital signature of the document is obtained and returned.

3. **GetPublicKey(user)**: returns the **public key** of the specified user.

   Precondition: secure session; user authenticated; key pair exists.

Postcondition: delivery of the requested public key.

4. **DeleteKeys**: deletes the key pair of the invoking user. *After deletion, the user cannot create a new one* unless a **new off-line registration** takes place.

   Precondition: secure session; user authenticated; key pair exists.

   Postcondition: no key pair associated with the user; recreation blocked until new registration.

Key management and protection The server **stores users' private keys in encrypted form**.

Operational order (typical flow consistent with the assignment)

1. **Off-line registration**: the user receives the DSS public key and the initial password.
2. **Establish the secure channel** with the DSS (properties: PFS, integrity, no-replay, non-malleability).
3. **Authenticate**: validate the server through its public key; authenticate the user via password.
4. **Password change at first access**.
5. **Key creation (one-time)**: invoke `CreateKeys` if the user does not yet have a pair.
6. **Ordinary use**:
   - to sign a document: `SignDoc(document)`;
   - to obtain a user's public key: `GetPublicKey(user)`.
7. **Termination**: if the user wishes to dispose of their keys, they invoke `DeleteKeys`. To have them again, a **new off-line registration** will be required.

Constraints and requirements to be respected

- All application operations occur **after** establishing the secure channel.
- `CreateKeys` is **idempotent** if the pair already exists.
- `DeleteKeys` has a **binding effect**: it prevents the creation of new keys until a new registration.
- Private keys are **always stored encrypted** on the server side.

2 System Architecture

The architecture consists of:

1. a TCP socket–based server, which listens for new connections, establishes a secure channel with clients, and manages the available functionalities;
2. a client, also implemented as a TCP socket, which contacts the server, negotiates the secure connection, and enables interaction with the end user.

The core of the architecture is the establishment of a secure channel, implemented through the following algorithms:

1. **Ephemeral Diffie–Hellman (X25519)** for the generation of temporary keys and the computation of the shared secret;
2. **HKDF** (HMAC-based Key Derivation Function with SHA-256) to securely derive the session key;
3. **AES-256-GCM** to encrypt and authenticate application messages and confirmation messages within the handshake.

Once the encrypted channel is established, the server proceeds with client authentication and subsequently enables access to the various application functionalities.

3 Communication Flow:

This section describes in a linear way how client and server establish the secure channel and how the exchange of application messages takes place. References to the "logical components" (e.g. *ClientHello*, *ServerHello*, *Transcript*, *HKDF*, *AES-GCM*) correspond to the homonymous parts implemented in the project.

3.1 Objective and Preconditions

- **Objective:** establish an authenticated and confidential channel with *Perfect Forward Secrecy* (PFS) between user and DSS, and then send/respond to application requests (e.g. `SignDoc`).
- **Preconditions:** the user possesses the **server's Ed25519 public key** (distributed off-line during registration).

3.2 Protocol Communication Model

M1: $C \rightarrow S : N_c, K_{e_c}$

M2: $S \rightarrow C : K_{e_s}, N_s, \{ H(\text{PROTO}, K_{e_c}, K_{e_s}, N_c, N_s) \}_{K_{\text{priv}S}}$

M3: $C \rightarrow S : \langle IV,\ AE_{K_{\text{session}}}(\text{"client-finish"} \parallel H(\text{transcript})) \rangle$

M4: $S \rightarrow C : \langle IV,\ AE_{K_{\text{session}}}(\text{"server-finish"} \parallel H(\text{transcript})) \rangle$

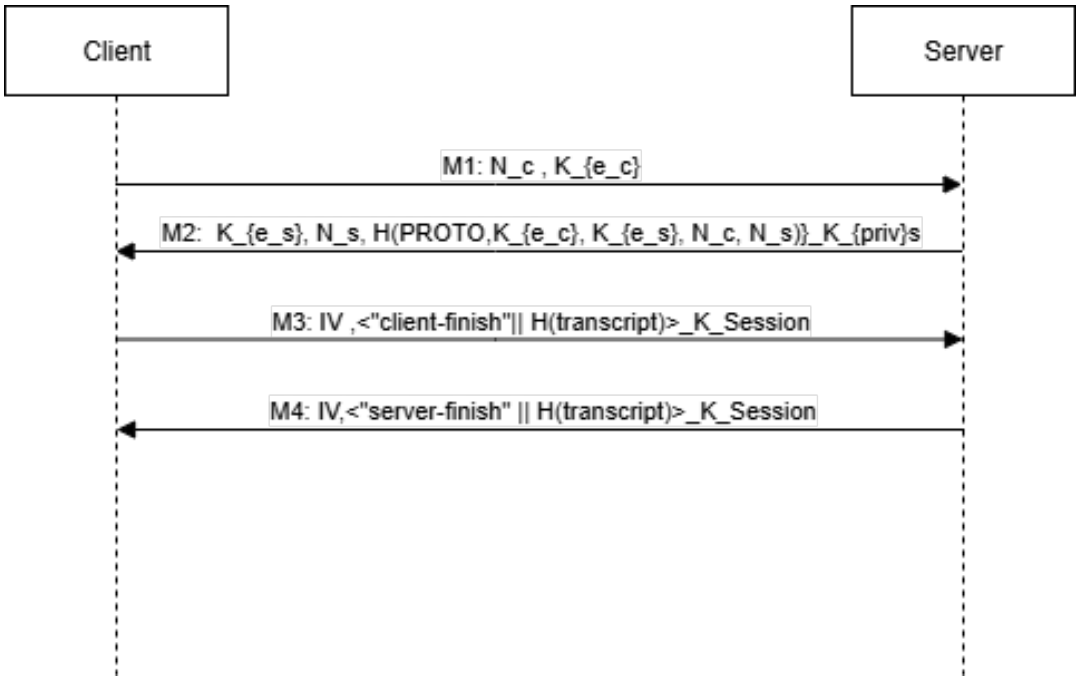


Figure 1: Sequnce diagram of Handshake

3.3 Handshake: negotiation, server authentication, derivation of K

1. **Server key loading (client side).** The client loads the Ed25519 public key of the DSS (obtained off-line). It will be used to verify the server's signature.
2. **TCP connection.** The client opens a connection to the DSS; the server accepts.
3. **ClientHello,** M1. The client generates an *ephemeral* X25519 key for the session K_{e_c} and a nonce N_c, then sends its ephemeral public key and N_c to the server.
4. **ServerHello with signature**, M2. The server in turn generates an ephemeral X25519 key K_{e_s} and a nonce N_s, computes the ECDH *shared secret*, and builds the **transcript** (Tab. 3). The server signs the hash of the transcript with its Ed25519 private key and sends the client: its ephemeral public key, N_s, and the **signature** on the transcript.
5. **Verification of server authenticity (client side).** The client reconstructs the same transcript and verifies its signature with the server's Ed25519 *public* key. If the verification fails, the session is terminated.
6. **Shared derivation of the session key K.** Both parties compute the same ECDH *shared secret*. From this they extract and *bind to the context* the session key K using **HKDF-SHA256**, with the parameters in Tab. 2. In this way, K is unique for the session and bound to the transcript (protocol, ephemeral keys, nonces).
7. **Bidirectional key confirmation.** Client and server exchange two short authenticated messages (*ClientFinish* and *ServerFinish*) encrypted with K. Each side verifies that it can decrypt and validate the authentication tag, thus proving knowledge of K.

Table 1: Handshake transcript (bytes in canonical order).

Field	Meaning
`PROTO`	Protocol/version identifier (e.g. `DSS/1`); prevents cross-protocol reuse.
`client_pub`	Client's *ephemeral* X25519 public key (PFS).
`server_pub`	Server's *ephemeral* X25519 public key (PFS).
N_c	Client-side nonce (anti-replay; contributes to session uniqueness).
N_s	Server-side nonce (anti-replay; contributes to session uniqueness).

Outcome of the handshake. If the verifications succeed, client and server share the same key K and have proven it (key-confirmation). The secure channel is *active*.

Table 2: HKDF derivation parameters for the session key K.

Parameter	Value/role
`input keying material`	*shared secret* from ECDH X25519.
`salt`	Concatenation $N_c \| N_s$; ensures uniqueness across sessions.
`info`	`hash(transcript)`; binds K to the context (protocol, keys, nonces).
`length`	32 bytes (suitable for AES-256-GCM).

3.4 Application phase: confidentiality, integrity, and anti-replay The channel satisfies the required properties in the following way:

- **Perfect Forward Secrecy:** achieved through the ephemeral Diffie–Hellman algorithm, where both client and server generate ephemeral keys for each session. The session key $K_{session}$ is derived through HKDF-SHA256, which strengthens security because the shared secret is not used directly but transformed into a unique session key bound to the session nonces and the transcript hash, ensuring that each run produces a distinct key tied to its context. In this way, even if an attacker records a session and later obtains the server's long-term private key, they cannot regenerate the same session key: the ephemeral keys are missing and the additional parameters used in HKDF cannot be reconstructed. Ephemeral keys are never stored on disk and vanish at the end of the handshake, so future compromise of long-term keys does not reveal past sessions.
- **Anti-replay:** each message includes an *increasing* sequence counter and a *unique* application nonce; in addition, an AAD (additional authenticated data) binds each message to the session context (via the transcript hash) and to the anti-replay parameters (seq, nonce).
- **Integrity:** every message (handshake and application) is protected by AES-256-GCM, which computes an authentication tag. If even a single bit of the message or of the AAD is modified, decryption fails.
- **Non-malleability:** ensured by the use of AES-GCM: an attacker cannot modify a ciphertext to produce a different but still valid plaintext. The AAD binds each message to the specific session context (H(transcript), seq, nonce), preventing "recycling" of message parts from other sessions.

4 Authentication

Server authentication is a cornerstone of the protocol: it is the mechanism that allows the user to ensure they are actually communicating with the *Digital Signature Server* (DSS) and not with an attacker (*man-in-the-middle*). To this end, we made several design choices:

1. **Start only over a secure channel.** Authentication begins after the handshake and the activation of AES-GCM, so username and password already travel encrypted and authenticated.
2. **AEAD protection with AAD bound to seq/nonce.** Each authentication message is encrypted with AES-GCM, and the AAD contains: app || info || seq || nonce. This ensures that seq and nonce, while remaining in cleartext, cannot be modified without breaking the tag.
3. **Use of seq and nonce as anti-replay.** The client increments seq and generates a new nonce for each request. The server stores the last seq and the set of seen nonces to reject duplicates or out-of-order messages. In responses, the server reuses the client's seq and nonce, so each reply is bound to the request.
4. **State and time management.** The server maintains pending_user and authed_user to track login state. There is a timeout (30 seconds) to prevent a session from remaining undefined and to avoid abuse.

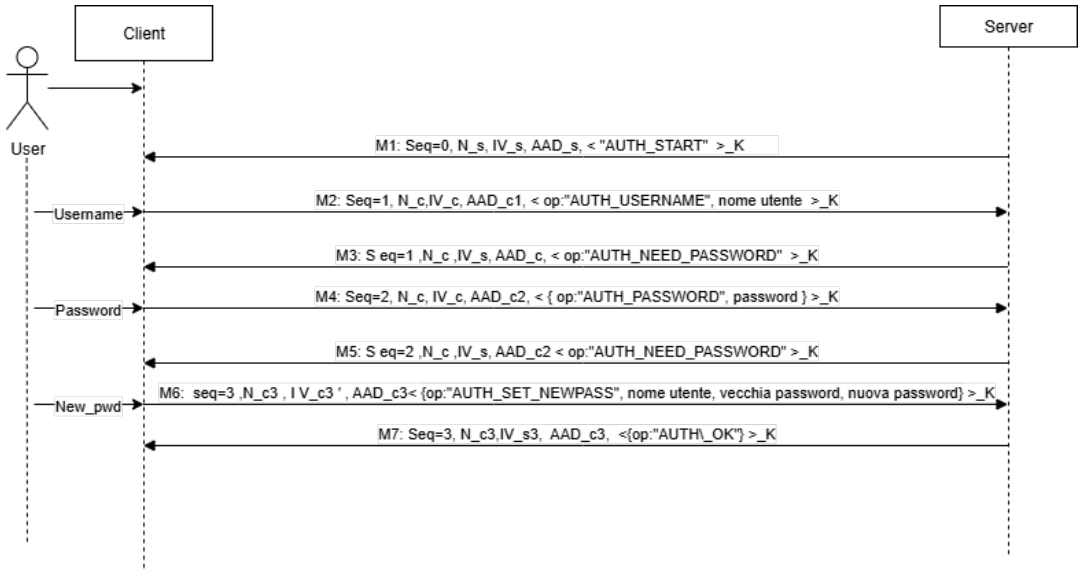


Figure 2: Sequnce diagram of Authentication

Scheme

M1: $S \to C : \{\, \texttt{seq} = 0,\ \texttt{nonce} = n_S,\ \texttt{iv}_S,\ \texttt{ct} = AE_K(\{\text{op:"AUTH\_START"}\}),\ \text{AAD}\,\}$

M2: $C \to S : \{\, \texttt{seq} = 1,\ \texttt{nonce} = n_{C1},\ \texttt{iv}_C,\ \texttt{ct} = AE_K(\{\text{op:"AUTH\_USERNAME"}, U\}),\ \text{AAD}\,\}$

M3: $S \to C : \{\, \texttt{seq} = 1,\ \texttt{nonce} = n_{C1},\ \texttt{iv}_S,\ \texttt{ct} = AE_K(\{\text{op:"AUTH\_NEED\_PASSWORD"}\}),\ \text{AAD}\,\}$

M4: $C \to S : \{\, \texttt{seq} = 2,\ \texttt{nonce} = n_{C2},\ \texttt{iv}_C,\ \texttt{ct} = AE_K(\{\text{op:"AUTH\_PASSWORD"}), P\},\ \text{AAD}\,\}$

M5: $S \to C : \{\, \texttt{seq} = 2,\ \texttt{nonce} = n_{C2},\ \texttt{iv}_S,\ \texttt{ct} = AE_K(\{\text{op:"AUTH\_NEED\_PWCHANGE"}), \text{AAD}\,\}$

M6: $C \to S : \{\, \texttt{seq} = 3,\ \texttt{nonce} = n_{C3},\ \texttt{iv}_C,\ \texttt{ct} = AE_K(\{\text{op:"AUTH\_SET\_NEWPASS"}, U, P, N\}),\ \text{AAD}\,\}$

M7: $S \to C : \{\, \texttt{seq} = 3,\ \texttt{nonce} = n_{C3},\ \texttt{iv}_S,\ \texttt{ct} = AE_K(\{\text{op:"AUTH\_OK" or "AUTH\_FAIL"}\}),\ \text{AAD}\,\}$

Protocol operation After the handshake is completed and the session key has been derived, the server initiates the user authentication phase. All messages in this phase are protected with AES-256-GCM, using as additional authenticated data (AAD) the handshake transcript hash, the sequence number, and a 16-byte nonce. In this way, even fields transmitted in cleartext (such as seq and nonce) are protected by the integrity check of the GCM tag.

The protocol begins with the server sending an AUTH_START message to the client, signaling the need for authentication. The client responds by incrementing the sequence number and including a fresh nonce, sending its username in an AUTH_USERNAME message. The server, reusing the same sequence number and nonce, replies with an AUTH_NEED_PASSWORD message, indicating that it expects the corresponding password.

Next, the client sends an AUTH_PASSWORD message, again with a new sequence number and a new nonce. The server verifies the credentials through the account management module and replies using the parameters of the request. The possible responses are:

- `AUTH_OK` – authentication successful;
- `AUTH_NEED_PWCHANGE` – password change is required;
- `AUTH_FAIL` – invalid credentials;
- `AUTH_TIMEOUT` – the procedure was not completed within the allowed time window.

If a password change is required, the client may send an AUTH_SET_NEWPASS message containing the old and new credentials. After verification, the server responds with either success or failure. Only after receiving `AUTH_OK` is the user considered authenticated and granted access to the cryptographic operations offered by the system.

Table 3: elements contained in the additional authenticated data (AAD)

Elements	Meaning
`app |`	The prefix acts as domain separation: it distinguishes application messages from handshake messages, prevents AAD collisions even with the same parameters, and strengthens non-malleability by avoiding reuse across different contexts.
`info`	Represents the hash of the handshake transcript. It links the authentication phase to the established secure channel, ensuring that messages cannot be reused in other sessions with a different transcript.
`seq`	The 8-byte progressive sequence number. Ensures that messages are processed in the correct order and that any duplicates of previously sent messages are detected and discarded.
`nonce`	A unique random 16-byte value generated by the client for each message. It guarantees freshness and uniqueness, preventing the creation of identical AADs and reinforcing defenses against replay attacks.

5 Operations

5.1 CreateKeys(*username*)

What it does The purpose of the operation is to securely generate and store *a single Ed25519 key pair* for each user. If the user already has an active key, another one is not created; instead, returns the metadata of the existing key, his property is called **idempotence**.

Formats and terminology

- **Raw format**: this term refers to the minimal binary representation of the key, without headers or additional metadata. For Ed25519 the public key is always 32 bytes long, and this sequence of 32 bytes is what is meant by "raw" format.
- **Unique IV (Initialization Vector)**: a random 12-byte value that is used every time the private key is encrypted with *AES-GCM*. It ensures that even when encrypting the same data twice, the results are always different, avoiding repeated patterns.
- **Encrypted blob**: this term refers to the result of encryption, namely a block of bytes containing both the ciphertext and the authentication tag generated by AES-GCM. It is not a strictly formal technical term, but in computing

jargon it is used to mean "an opaque block of binary data."

The saved record The result of creation is saved in a JSON file containing all the necessary information. A simplified example is:

```
{
  "key_id": "<unique_key_identifier>",
  "username": "<username>",
  "algo": "ed25519",
  "public_key_b64": "<public_key_in_base64>",
  "enc_priv_key_b64": "<encrypted_private_key_in_base64>",
  "nonce_b64": "<iv_used_for_encryption_in_base64>",
  "created_at": "<creation_timestamp_ISO_8601>",
  "status": "active"
}
```

Here we see the key identifier, the algorithm, the public key encoded in Base64, the encrypted private key (also in Base64), the IV used for encryption, the creation date, and the key status.

Atomic persistence Saving is carried out in a way that prevents corrupted files. The correct technical approach is called **atomic writing**: first, data is written to a temporary file, and only when the write has completed successfully is the temporary file renamed to replace the real one. This ensures that the disk always contains either the old version or the new one, never a file cut off halfway.

Why this approach works The combination of these choices results in a robust system:

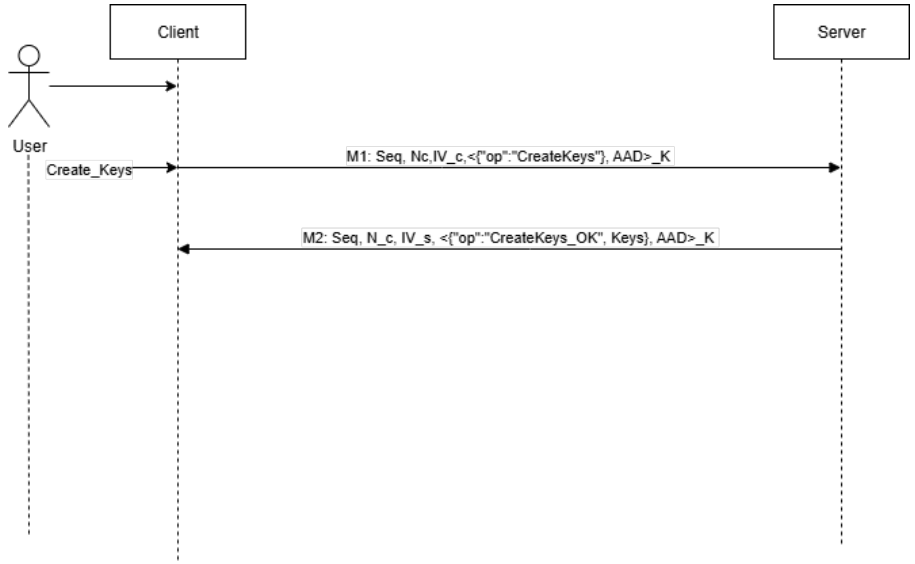
- **Idempotence** guarantees that each user always has one and only one active key: there is no risk of multiplying keys by mistake.
- **AEAD encryption** (AES-GCM) protects the confidentiality of the private key while also ensuring that the encrypted data cannot be modified undetected, thanks to the authentication tag.
- **Atomic persistence** eliminates the risk of incomplete or corrupted files in case of sudden interruptions (crash, power loss).
- The **user index** always keeps track of the active key to use, simplifying subsequent operations.

Comparison with weaker approaches

- Saving the private key **in cleartext** on disk would be much simpler, but it would mean that anyone with access to the file system could steal the keys and sign documents as if they were the user.

- Using an encryption mode that is not AEAD, such as *AES-CBC*, would ensure confidentiality but not integrity: an attacker could tamper with the encrypted file without the alteration being detected.
- Writing files directly, without atomic mechanisms, may leave truncated files or inconsistent indexes if the program is interrupted during the write.

Scheme

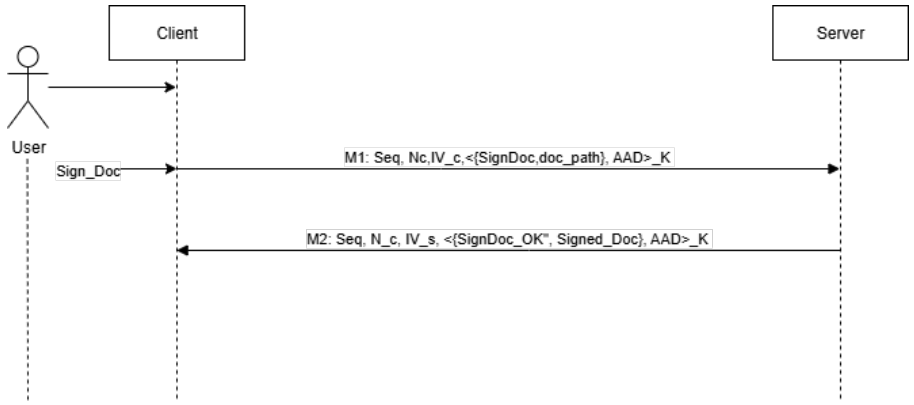


5.2 SignDoc(*auth_user*, *doc_path*) The `SignDoc` operation signs the digest of the document specified by *doc_path* with the private key of the user *auth_user*. The private key is stored encrypted "at rest" and is decrypted only in RAM for the strictly necessary time; the file's *SHA-256* hash is then computed in streaming and a "detached" *Ed25519* signature is produced and saved next to the document.

Operational steps

1. **Preliminary verification of the document** — checks that *doc_path* exists and aborts otherwise.
2. **Selection of the user's single active key** — identifies the user's key file, loads it, and validates that it is present.
3. **Decryption of the private key "at rest"** — loads the master key, initializes *AES-GCM*, and decrypts the private key blob using *nonce_b64* and *enc_priv_key_b64*; the AEAD tag ensures integrity.
4. **Reconstruction of the Ed25519 key** — creates the private key object from the 32 raw bytes.
5. **Streaming hash of the document** — computes *SHA-256* by reading the file in blocks (support for large files), produces the digest and its Base64 encoding for output.
6. **Signing the digest** — applies *Ed25519.sign* to the digest and encodes the signature in Base64.
7. **Issuing the "detached" signature** — writes `<doc>.sig` with minimal metadata (user, *key_id*, algorithm, hash, and signature) to facilitate verification without altering the document.
8. **Returning metadata** — returns to the caller the hash, signature, and signature path for audit and verification.

Scheme

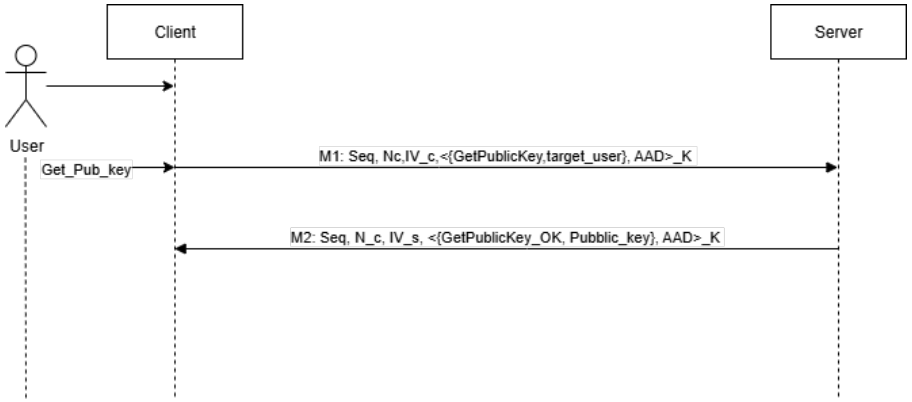

Client
Server
User
Sign_Doc
M1: Seq, Nc,IV_c,<{SignDoc,doc_path}, AAD>_K
M2: Seq, N_c, IV_s, <{SignDoc_OK", Signed_Doc}, AAD>_K

5.3 GetPublicKey(*target_user*, *optional key_id*) `GetPublicKey` returns the user's *public key*, in the "one pair per user" model. If a *key_id* is provided, the function checks that it matches the single record present; otherwise, it reports an error. The output is minimal and sufficient for verification: identifier, algorithm, and public key in Base64.

Operational steps (with references to lines)

1. **Identification of the user's record** — lists the user's key files and fails if none are found.
2. **Loading the record** — opens the JSON of the identified key and deserializes it into memory.
3. **State verification** — only accepts keys with status *active*; otherwise it reports `KeyNotFound`.
4. **Optional validation of *key_id*** — if provided, it must match the actual record identifier; otherwise, a consistent error is raised.
5. **Construction of minimal response** — prepares and returns {`key_id`, `algo`, `public_key_b64`} and records the result in the logs.

Scheme



5.4 DeleteKeys(*auth_user*) The operation allows the authenticated owner to delete their own key. The function identifies the user's single key record; if absent, it explicitly reports the error. If the record is present, it retrieves its identifier and proceeds with the physical removal of the file from the keystore, adopting a *hard delete* approach to minimize the attack surface in case of disk compromise. It then updates the index by setting the absence of a default key and enabling a registration lock (*registration_locked*) until a new controlled procedure (a new offline registration phase).

The persistence of index modifications is carried out *atomically*, preserving the invariant between logical state and filesystem content even in the presence of interruptions.

Scheme