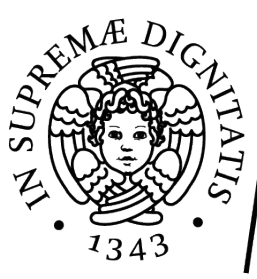


UNIVERSITÀ
DI PISA

University of Pisa

Department of Information Engineering

FOC project

Nicolò Mariano Fragale, Federico Rossetti

May 2025

Contents

1 Analisi DSS

Obiettivo del sistema Un'organizzazione utilizza un servizio di firma digitale (Digital Signature Server, DSS) che agisce come terza parte fidata: genera coppie di chiavi per conto dei dipendenti, le conserva e produce firme digitali su richiesta dell'utente.

Registrazione iniziale (off-line) e credenziali Gli utenti/dipendenti sono registrati *off-line*. In fase di registrazione ricevono:

- la **chiave pubblica del DSS** (da conservare);
- una **password iniziale**, che *deve essere cambiata al primo accesso.*

Canale sicuro e autenticazione Prima di invocare qualsiasi operazione, l'utente deve stabilire un **canale sicuro** verso il DSS, che soddisfi i requisiti di:

- **Perfect Forward Secrecy (PFS)**;
- **integrità dei messaggi**;
- **protezione dal replay** (no-replay);
- **non-malleabilità**.

L'autenticazione avviene come segue:

- **autenticazione del server** tramite la sua chiave pubblica (nota all'utente);
- **autenticazione dell'utente** tramite la sua password.

Operazioni esposte dal servizio Dopo la connessione sicura e l'autenticazione, il DSS espone le seguenti operazioni di livello applicativo:

1. **CreateKeys**: crea e memorizza una coppia *(privata, pubblica)* per l'utente invocante. Se la coppia *esiste già*, l'operazione *non ha effetto* (idempotenza).

   Precondizione: sessione sicura attiva; utente autenticato.

   Postcondizione: esiste una coppia di chiavi associata all'utente (invariata se già presente).

2. **SignDoc(documento)**: restituisce la **firma digitale** del documento passato come argomento; il DSS firma *per conto dell'utente invocante.*

   Precondizione: sessione sicura; utente autenticato; *coppia di chiavi esistente.*

   Postcondizione: ottenuta e restituita la firma digitale sul documento.

3. **GetPublicKey(utente)**: restituisce la **chiave pubblica** dell'utente indicato.

Precondizione: sessione sicura; utente autenticato, coppia di chiavi esistente.

Postcondizione: consegna della chiave pubblica richiesta.

4. **DeleteKeys**: elimina la coppia di chiavi dell'utente invocante. *Dopo l'eliminazione, l'utente non può crearne una nuova* a meno di una **nuova registrazione off-line**.

   Precondizione: sessione sicura; utente autenticato; coppia di chiavi esistente.

   Postcondizione: nessuna coppia di chiavi associata all'utente; blocco della ricreazione fino a nuova registrazione.

Gestione e protezione delle chiavi Il server **memorizza le chiavi private degli utenti in forma cifrata**.

Ordine operativo (flusso tipico coerente con la consegna)

1. **Registrazione off-line**: consegna all'utente della chiave pubblica del DSS e della password iniziale.
2. **Stabilire il canale sicuro** con il DSS (proprietà PFS, integrità, no-replay, non-malleabilità).
3. **Autenticarsi**: validare il server tramite la sua chiave pubblica; autenticare l'utente via password.
4. **Cambio password al primo accesso**.
5. **Creazione chiavi (una tantum)**: invocare `CreateKeys` se l'utente non ha ancora una coppia.
6. **Uso ordinario**:
   - per firmare un documento: `SignDoc(documento)`;
   - per ottenere la chiave pubblica di un utente: `GetPublicKey(utente)`.
7. **Cessazione**: se l'utente vuole dismettere le proprie chiavi, invoca `DeleteKeys`. Per poterle avere di nuovo, sarà necessaria **una nuova registrazione off-line**.

Vincoli e requisiti da rispettare

- Tutte le operazioni applicative avvengono **dopo** l'instaurazione del canale sicuro.
- `CreateKeys` è **idempotente** se la coppia esiste già.
- `DeleteKeys` ha effetto **vincolante**: impedisce la creazione di nuove chiavi fino a nuova registrazione.

- Le chiavi private sono **sempre archiviate cifrate** lato server.

Contenuti obbligatori della relazione La relazione del progetto deve includere:

1. **Specifiche e scelte progettuali**, con particolare attenzione al **protocollo di autenticazione** tra utente e servizio.
2. **Formato di tutti i messaggi** scambiati (livello applicativo).
3. **Diagrammi di sequenza** di ogni protocollo di comunicazione utilizzato (livello applicativo).

2 Digital Signature Server

2.1 __init__ Questa è la funzione costruttore della classe.

- **host='0.0.0.0':** L'indirizzo IP su cui il server ascolterà le connessioni. Il valore '0.0.0.0' indica che il server sarà accessibile su tutte le interfacce di rete disponibili del computer (locale, LAN, Internet).
- **port=5000:** La porta su cui il server ascolterà le connessioni in ingresso.
- **certfile='server-cert.pem'**
- **keyfile='server-key.pem'**
- **self.server_socket = socket.socket(...):** Crea un nuovo socket TCP/IP.
  - **socket.AF_INET:** Specifica che il server utilizzerà l'IPv4 per la comunicazione di rete.
  - **socket.SOCK_STREAM:** Indica che il socket è di tipo stream (flusso)
- **self.server_socket.bind(...):** Associa il socket a una porta e un indirizzo specifici sul server, (0.0.0.0 e 5000).
- **self.server_socket.listen(5):** unghezza massima della coda di connessioni in attesa.

3 Autenticazione

L'autenticazione del server costituisce un punto cardine del protocollo: è il meccanismo che permette all'utente di accertarsi di comunicare effettivamente con il *Digital Signature Server* (DSS) e non con un attaccante (*man-in-the-middle*). A tal fine, il progetto adotta due scelte fondamentali:

1. L'utilizzo dello schema di firma digitale **Ed25519**.
2. L'inclusione di un identificatore di protocollo/versione (`PROTO = "DSS/1"`) nel *transcript* dell'handshake.

3.1 Ed25519 è uno schema di firma digitale basato su curve ellittiche, che utilizza l'algoritmo EdDSA (*Edwards-curve Digital Signature Algorithm*) sulla curva Curve25519.

Motivazioni della scelta

- **Sicurezza elevata**: offre un livello di sicurezza paragonabile a RSA-3072, ma con chiavi e firme molto più corte.
- **Efficienza**: genera e verifica firme in tempi ridottissimi, riducendo il carico computazionale.
- **Compattezza**: le chiavi (32 byte) e le firme (64 byte) sono estremamente leggere e adatte a protocolli di rete.
- **Standard consolidato**: ampiamente utilizzato in progetti reali (OpenSSH, GnuPG, Tor), quindi ben testato e supportato.

Funzionamento nel protocollo Il server possiede una coppia di chiavi Ed25519 *a lungo termine* (generata una volta sola e fornita agli utenti in fase di registrazione offline). Durante l'handshake:

1. Viene costruito un *transcript* che lega i parametri effimeri di Diffie–Hellman, i nonces e l'identificatore di protocollo.
2. Il server calcola l'hash del transcript e lo firma con la propria chiave privata Ed25519.
3. L'utente, in possesso della chiave pubblica del server (ricevuta offline), verifica la firma.

In questo modo si garantisce che la chiave effimera e i parametri di sessione provengano realmente dal server, e non da un attaccante.

Applicazioni reali Ed25519 è usato in contesti dove sono fondamentali autenticità e prestazioni:

- **OpenSSH** come formato di chiave per autenticazione sicura dei server.
- **Protocolli di messaggistica sicura** (Signal, Wire) per firme veloci e leggere.

- **Blockchain e criptovalute** (es. Monero) per firme compatte ed efficienti.

Punti di forza

- Algoritmo moderno, con parametri ben scelti per evitare debolezze note in curve ellittiche.
- Non richiede infrastruttura PKI esterna: basta distribuire la chiave pubblica del server offline.
- Alta resistenza a implementazioni insicure (side-channel).

Possibili migliorie

- Proteggere ulteriormente la chiave privata a lungo termine con *Hardware Security Modules* (HSM) o enclave sicure.
- Introdurre un meccanismo di *certificate pinning* o catena di fiducia per evitare distribuzione manuale della chiave.

3.2 Identificatore di protocollo e prevenzione cross-protocol Durante la costruzione del transcript viene inserito un campo costante:

```
PROTO = "DSS/1"
```

Tale campo identifica in modo univoco il protocollo e la sua versione.

Motivazioni della scelta

- **Versioning**: consente di distinguere handshake di versioni differenti del protocollo (es. `DSS/2`) e gestire la compatibilità.
- **Difesa da attacchi cross-protocol**: impedisce che firme valide nel contesto DSS possano essere riutilizzate in altri protocolli (ad es. SSH, TLS) o in versioni diverse.

Funzionamento Poiché il transcript firmato include l'identificatore di protocollo:

- Una firma prodotta per `DSS/1` non sarà valida per `TLS/1.3` o per `DSS/2`.
- Questo vincola crittograficamente le chiavi effimere e i nonces al protocollo specifico.

Applicazioni reali

- Protocolli moderni come TLS 1.3 includono un campo *context string* nelle funzioni di derivazione delle chiavi, con finalità analoghe.
- Signal e Noise Protocol Framework usano identificatori simili per legare le chiavi a un determinato protocollo e versione.

Punti di forza

- Semplicità: l'aggiunta di un identificatore non complica l'implementazione.
- Robustezza: previene classi di attacchi sottili e difficili da rilevare (firma riutilizzata in altro contesto).

Possibili migliorie

- Utilizzare identificatori strutturati (`DSS/1.0`, `DSS/1.1`, ...) per gestire release incrementali.
- Includere anche l'elenco delle *ciphersuite* supportate per rafforzare la negoziazione.

3.3 Sintesi L'autenticazione nel DSS si fonda su:

1. Uno schema di firma digitale moderno (Ed25519) che garantisce autenticità e prestazioni elevate.
2. L'uso di un identificatore di protocollo/versione nel transcript per assicurare il corretto contesto delle firme digitali e prevenire riutilizzi illeciti.

Queste scelte rendono il protocollo robusto, sicuro e allineato alle migliori pratiche dei protocolli crittografici moderni.

4 Flusso della comunicazione: descrizione lineare e schemi

Questa sezione descrive in modo lineare come client e server instaurano il canale sicuro e come avviene lo scambio dei messaggi applicativi, senza riportare codice. I riferimenti ai "componenti logici" (es. *ClientHello*, *ServerHello*, *Transcript*, *HKDF*, *AES-GCM*) corrispondono alle parti omonime implementate nel progetto.

4.1 Obiettivo e precondizioni

- **Obiettivo:** stabilire un canale autenticato e confidenziale con *Perfect Forward Secrecy* (PFS) tra utente e DSS, per poi inviare/rispondere a richieste applicative (es. `SignDoc`).
- **Precondizioni:** l'utente possiede la **chiave pubblica Ed25519 del server** (distribuita off-line in fase di registrazione).

4.2 Handshake: negoziazione, autenticazione del server, derivazione di K

1. **Caricamento chiave del server (lato client).** Il client carica la chiave pubblica Ed25519 del DSS (ottenuta off-line). Servirà a verificare la firma del server.
2. **Connessione TCP.** Il client apre una connessione verso il DSS; il server accetta.
3. **ClientHello.** Il client genera una chiave *effimera* X25519 per la sessione e un nonce N_c, quindi invia al server la propria chiave pubblica effimera e N_c.
4. **ServerHello con firma.** Il server genera a sua volta una chiave effimera X25519 e un nonce N_s, calcola lo *shared secret* ECDH e costruisce il **transcript** (Tab. 1). Il server firma l'hash del transcript con la propria chiave privata Ed25519 e invia al client: la sua chiave pubblica effimera, N_s e la **firma** sul transcript.
5. **Verifica autenticità del server (lato client).** Il client ricostruisce lo stesso transcript e ne verifica la firma con la *pubblica* Ed25519 del server. Se la verifica fallisce, la sessione viene interrotta.
6. **Derivazione condivisa della chiave di sessione K.** Entrambe le parti calcolano lo stesso *shared secret* ECDH. Da questo estraggono e *legano al contesto* la chiave di sessione K tramite **HKDF-SHA256**, usando i parametri di Tab. 2. In questo modo K è univoca per la sessione e vincolata al transcript (protocollo, chiavi effimere, nonces).
7. **Key-confirmation bidirezionale.** Client e server si scambiano due brevi messaggi autenticati (*ClientFinish* e *ServerFinish*) cifrati con K. Ogni lato verifica di poter decifrare e convalidare il tag di autenticazione, dimostrando di conoscere K.

Table 1: Transcript di handshake (bytes in ordine canonico).

Campo	Significato
`PROTO`	Identificatore protocollo/versione (es. `DSS/1`); previene riusi cross-protocol.
`client_pub`	Chiave pubblica *effimera* X25519 del client (PFS).
`server_pub`	Chiave pubblica *effimera* X25519 del server (PFS).
N_c	Nonce lato client (anti-replay; contribuisce a unicità sessione).
N_s	Nonce lato server (anti-replay; contribuisce a unicità sessione).

Table 2: Parametri di derivazione HKDF per la chiave di sessione K.

Parametro	Valore/ruolo
`input keying material`	*shared secret* da ECDH X25519.
`salt`	Concatenazione $N_c \| N_s$; garantisce unicità tra sessioni.
`info`	`hash(transcript)`; lega K al contesto (protocollo, chiavi, nonces).
`length`	32 byte (idoneo ad AES-256-GCM).

Esito dell'handshake. Se le verifiche vanno a buon fine, client e server condividono la stessa chiave K e l'hanno dimostrato (key-confirmation). Il canale sicuro è *attivo*.

4.3 Fase applicativa: confidenzialità, integrità e anti-replay Una volta attivo il canale:

- **Cifratura/autenticazione:** tutti i messaggi applicativi sono protetti con **AES-GCM** sotto la chiave K, ottenendo confidenzialità, integrità e autenticità (tramite il tag AEAD).
- **Anti-replay:** ogni messaggio include un contatore di sequenza *crescente* e un nonce applicativo *unico*; inoltre, una *AAD* (dati autenticati addizionali) lega ogni messaggio al contesto di sessione (tramite l'hash del transcript) e ai parametri anti-replay (seq, nonce).
- **Vincolo richiesta–risposta:** la risposta del server è crittograficamente legata alla richiesta (riutilizzando lo stesso contesto AAD) così da impedire riordinamenti o riusi fuori contesto.

4.4 Mappa concettuale delle operazioni

Autenticazione del server. Realizzata firmando l'hash del transcript con Ed25519; verificata dal client con la chiave pubblica distribuita off-line.

PFS (Perfect Forward Secrecy). Ottenuta grazie alle chiavi X25519 *effimere per sessione* su entrambe le parti.

Derivazione K. HKDF-SHA256 con `salt` = $N_c \| N_s$ e `info` = `hash(transcript)` garantisce una chiave unica e contestualizzata.

Integrità e non-malleabilità. AES-GCM (AEAD) rifiuta qualsiasi modifica ai messaggi (tag non valido).

Protezione dal replay. Combinazione di *seq* crescente, *nonce* unici e AAD che incorpora contesto, seq e nonce.

4.5 Esempi di uso nella fase applicativa Una volta attivo il canale, le operazioni applicative (`Login`, `CreateKeys`, `SignDoc`, `GetPublicKey`, `DeleteKeys`) viaggiano all'interno del tunnel cifrato:

- **Richiesta:** il client invia un messaggio cifrato/autenticato con K, includendo *seq*, *nonce* e AAD vincolata al transcript.
- **Risposta:** il server elabora, quindi invia la risposta cifrata/autenticata con K, riutilizzando lo stesso contesto AAD per legare la risposta alla richiesta.

5 Operazioni

5.1 CreateKeys(*username*)

Che cosa fa L'operazione ha lo scopo di generare e memorizzare in maniera sicura *una sola coppia di chiavi Ed25519* per ogni utente. Se l'utente ha già una chiave attiva, non ne viene creata un'altra, ma vengono restituiti i metadati della chiave esistente: questa caratteristica prende il nome di **idempotenza**. In questo modo si evita di avere più chiavi diverse per lo stesso utente, rendendo il sistema più semplice e coerente.

Formati e terminologia

- **Formato grezzo (raw)**: con questo termine si intende la rappresentazione binaria minima della chiave, senza intestazioni o metadati aggiuntivi. Per Ed25519 la chiave pubblica è sempre lunga 32 byte, e questa sequenza di 32 byte è ciò che si intende con formato "grezzo".
- **IV univoco (Initialization Vector)**: è un valore casuale di 12 byte che viene usato ogni volta che si cifra la chiave privata con *AES-GCM*. Serve a garantire che anche cifrando lo stesso dato due volte si ottengano risultati sempre diversi, evitando pattern ripetuti.
- **Blob cifrato**: con questo termine si indica il risultato della cifratura, ovvero un blocco di byte che contiene sia il testo cifrato sia il tag di autenticazione generato da AES-GCM. Non è un termine strettamente tecnico formale, ma in gergo informatico viene usato per riferirsi a "un blocco opaco di dati binari".

Il record salvato Il risultato della creazione viene salvato in un file JSON contenente tutte le informazioni necessarie. Un esempio semplificato è:

```
{
  "key_id": "<identificativo_univoco_della_chiave>",
  "username": "<nome_utente>",
  "algo": "ed25519",
  "public_key_b64": "<chiave_pubblica_in_base64>",
  "enc_priv_key_b64": "<chiave_privata_cifrata_in_base64>",
  "nonce_b64": "<iv_usato_per_la_cifratura_in_base64>",
  "created_at": "<timestamp_ISO_8601_di_creazione>",
  "status": "active"
}
```

Qui si vede l'identificativo della chiave, l'algoritmo, la chiave pubblica codificata in Base64, la chiave privata cifrata (anch'essa in Base64), l'IV usato per la cifratura, la data di creazione e lo stato della chiave.

Persistenza atomica Il salvataggio viene fatto in modo da evitare file corrotti. La tecnica corretta e tecnica si chiama **scrittura atomica**: si scrive prima su un

file temporaneo e solo quando la scrittura è andata a buon fine il file temporaneo viene rinominato al posto di quello vero. Questo assicura che sul disco ci sia sempre o la vecchia versione o la nuova, mai un file interrotto a metà.

Aggiornamento dell'indice Oltre al file della chiave, esiste un file di indice per utente (`index.json`), che serve a sapere qual è la chiave predefinita. Dopo la creazione viene aggiornato così:

```
{
  "default_key_id": "ed25519-20250825-1a2b",
  "registration_locked": false
}
```

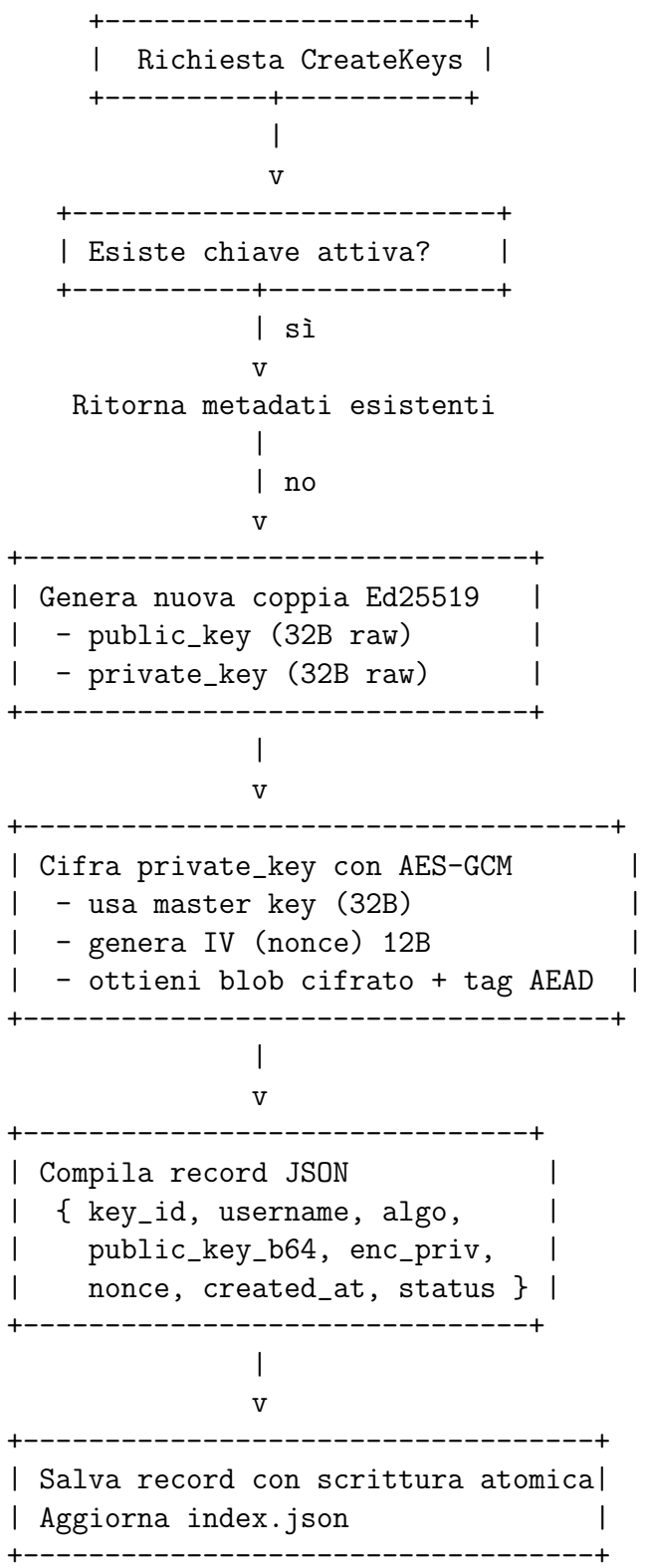
Perché questo approccio funziona L'insieme di queste scelte porta a un sistema robusto:

- L'**idempotenza** garantisce che ogni utente abbia sempre e solo una chiave attiva: non si rischia di moltiplicare chiavi per errore.
- La **cifratura con AEAD** (AES-GCM) protegge la confidenzialità della chiave privata e allo stesso tempo assicura che i dati cifrati non possano essere modificati senza essere scoperti, grazie al tag di autenticazione.
- La **persistenza atomica** elimina il rischio di avere file incompleti o corrotti in caso di interruzioni improvvise (crash, power off).
- L'**indice utente** mantiene sempre chiara la chiave attiva da usare, semplificando le operazioni successive.

Confronto con approcci più deboli

- Salvare la chiave privata **in chiaro** su disco sarebbe molto più semplice, ma significherebbe che chiunque abbia accesso al file system può rubare le chiavi e firmare documenti come se fosse l'utente.
- Usare una modalità di cifratura non AEAD, come *AES-CBC*, garantirebbe la riservatezza ma non l'integrità: un attaccante potrebbe manipolare il file cifrato senza che l'alterazione venga rilevata.
- Scrivere i file direttamente, senza meccanismi atomici, può lasciare file tronchi o indici incoerenti se il programma viene interrotto durante la scrittura.

Schema del flusso

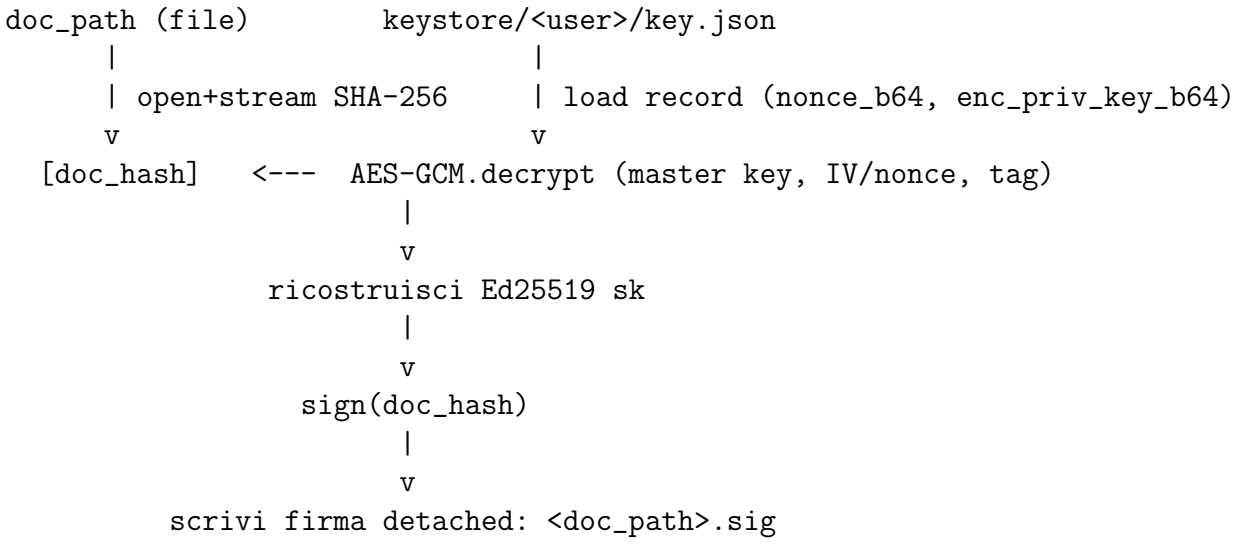


5.2 SignDoc(*auth_user*, *doc_path*) L'operazione `SignDoc` firma il digest del documento indicato da *doc_path* con la chiave privata dell'utente *auth_user*. La chiave privata è conservata cifrata "a riposo" e viene decifrata solo in RAM per il tempo strettamente necessario; si calcola quindi l'hash *SHA-256* del file in streaming e si produce una firma *Ed25519* "detached" salvata accanto al documento.

Fasi operative

1. **Verifica preliminare del documento** — controlla che *doc_path* esista ed interrompe in caso contrario.
2. **Selezione dell'unica chiave attiva dell'utente** — individua il file-chiave dell'utente, lo carica e valida che sia presente.
3. **Decifratura della privata "a riposo"** — carica la master key, inizializza *AES-GCM* e decifra il blob della privata usando *nonce_b64* e *enc_priv_key_b64*; il tag AEAD garantisce integrità.
4. **Ricostruzione della chiave Ed25519** — crea l'oggetto chiave privata a partire dai 32 byte grezzi.
5. **Hash del documento in streaming** — calcola *SHA-256* leggendo il file a blocchi (supporto a file grandi), produce il digest e la sua codifica Base64 per l'output.
6. **Firma del digest** — applica *Ed25519.sign* al digest e codifica la firma in Base64.
7. **Emissione della firma "detached"** — scrive `<doc>.sig` con metadati minimi (utente, *key_id*, algoritmo, hash e firma) per facilitare verifiche senza alterare il documento.
8. **Ritorno metadati** — restituisce al chiamante hash, firma e percorso della firma per audit e verifica.

Schema del flusso



5.3 GetPublicKey(*target_user*, *key_id opzionale*) `GetPublicKey` restituisce la *chiave pubblica* dell'utente richiesto, nel modello "una sola coppia per utente". Se viene passato un *key_id*, la funzione verifica che coincida con l'unico record presente; in caso contrario segnala errore. L'output è minimale e sufficiente alla verifica: identificativo, algoritmo e pubblica in Base64.

Fasi operative (con riferimenti alle righe)

1. **Individuazione del record dell'utente** — elenca i file-chiave dell'utente e fallisce se non ne trova.
2. **Caricamento del record** — apre il JSON della chiave individuata e lo deserializza in memoria.
3. **Verifica di stato** — accetta solo chiavi con stato *active*; altrimenti segnala `KeyNotFound`.
4. **Validazione opzionale di *key_id*** — se fornito, deve coincidere con l'identificativo effettivo del record; in caso contrario errore coerente.
5. **Costruzione risposta minimale** — prepara e restituisce {`key_id`, `algo`, `public_key_b64`} e registra l'esito nei log.

Schema del flusso

```
target_user
    |
    v
list key files  --(none)--> errore: KeyNotFound
    |
    v
open & parse record JSON
    |
    +--(status != active)--> errore: KeyNotFound
    |
    +--(key_id passato e != actual_id)--> errore: KeyNotFound
    |
    v
{ key_id, algo, public_key_b64 }  --> return
```

5.4 DeleteKeys(*auth_user*) L'operazione consente al proprietario autenticato di eliminare la propria chiave. La funzione individua l'unico record-chiave dell'utente; se assente, segnala esplicitamente l'errore. In presenza del record, ne rileva l'identificativo e procede alla rimozione fisica del file dal keystore, adottando un approccio di *hard delete* per minimizzare la superficie d'attacco in caso di compromissione del disco. Successivamente aggiorna l'indice impostando l'assenza di chiave predefinita e abilitando un blocco di registrazione (*registration_locked*) fino a una nuova procedura controllata (una nuova fase di registrazione offline).

La persistenza delle modifiche all'indice avviene in modo *atomico*, preservando l'invariante tra stato logico e contenuto del filesystem anche in presenza di interruzioni.