POLITECNICO DI TORINO

Corso di Laurea Magistrale
in Ingegneria Informatica

Tesi di Laurea Magistrale

Gerarchia di chiavi d'identità per wallet deterministici in un sistema di Self-Sovereign Identity

Relatori
Prof. Danilo Bazzanella

Candidato
Carmine Ruggieri

Tutors
LINKS FOUNDATION
Andrea Vesco, Ph.D
Davide Margaria

Anno Accademico 2022-2023

Sommario

La Self-Sovereign Identity (SSI) è un nuovo paradigma per la gestione dell'identità digitale dell'utente che ha l'obiettivo di restituire il controllo dell'identità e dei propri dati all'individuo anziché affidarli a terze parti. Nel contesto della SSI l'utente può creare la propria identità, controllarla e fornire una prova che lui stesso è il detentore dell'identità, quindi il titolare. Due sono gli standard principali proposti dal W3C Group che fanno riferimento a questo background, il primo è quello relativo ai Decentralized Identifiers (DID) che definisce un nuovo tipo di identificativo che sia verificabile e non centralizzato, proprio come da definizione, e il secondo invece fa riferimento alle Verifiable Credentials (VC) ovvero alla definizione di uno standard per la struttura e per lo scambio sicuro di credenziali in un ambiente digitale. Tuttavia gli standard non forniscono una descrizione di come le chiavi di identità debbano essere generate e memorizzate all'interno del sistema. Nelle primissime implementazioni le chiavi venivano generate in modo randomico, non correlate tra loro, secondo le necessità dell'utente costringendo l'utente a dover memorizzare ogni chiave privata individualmente (Just-a-Bunch-of-Keys). Quello invece che si è voluto fare in questo elaborato è proporre una soluzione per la gestione delle chiavi d'identità dell'utente, definendo una nuova gerarchia basata sugli standard Bitcoin Improvements Proposal (BIP) e memorizzarle in un ambiente isolato per custodirle in modo sicuro. La soluzione proposta è l'implementazione di un DID method scritto in linguaggio Rust che fa uso di smart-contract sviluppati in Solidity. Invece per implementare la nuova gerarchia delle chiavi si è scelto di estendere la libreria open-source IOTA Stronghold.

Ringraziamenti

Ringrazio il prof. Danilo Bazzanella e i tutor di Links Andrea Vesco e Davide Margaria per avermi permesso di lavorare su questo progetto. Un ringraziamento particolare va anche a Luca Giorgino per avermi seguito con costanza e soprattutto pazienza ed è stato di grande supporto per tutto il lavoro svolto che ha portato a questo risultato finale.

Ringrazio la mia famiglia per avermi sempre sostenuto nelle mie scelte, Massimo per essermi sempre stato accanto nonostante non sia facile e i miei amici che ci sono sempre stati in qualunque momento.

Indice

Capitolo 1

Introduzione

Oggigiorno è sempre più frequente parlare di identità digitale, soprattutto bisogna accedere a servizi pubblici o privati di un qualsiasi ente dove spesso viene richiesto di autenticarsi utilizzando sistemi terzi. Si pensi a cosa si fa quasi tutti i giorni qui in Italia con lo SPID o la Carta d'Identità Elettronica (1). Quindi l'identità digitale può essere vista come la rappresentazione digitale di tutto ciò che ha attinenza con l'identità fisica dell'utente titolare della relativa controparte e quindi come una credenziale unica per accedere a tutti i servizi. Ovviamente questi servizi sono forniti da terze parti accreditate, dove prima di poter rilasciare questa identità digitale bisogna effettuare una registrazione inserendo tutti i dati identificativi e i riferimenti alle credenziali fisiche in possesso per poi fare una verifica sul titolare. A questo punto il provider terzo diventa il detentore della nostra identità digitale e dei dati ad esso associata con tutta una serie di conseguenze che potrebbero verificarsi in alcuni casi: si pensi ad esempio al furto e alla replica dei dati da parte di un utente fraudolento che potrebbe spacciarsi per il soggetto come conseguenza di un "data breach" avvenuto, all'inserimento di dati personali inutili e quindi problemi di privacy ed altro ancora.

Nel tempo vari studi, soprattutto con l'avvento delle blockchain e dei distributed ledger, hanno portato a definire un nuovo modello di identità digitale che prende il nome di **Self-Sovereign Identity** o **SSI**, che ha lo scopo di restituire all'utente il pieno controllo della propria identità, di gestirla come meglio crede e di controllare appunto tutti i dati ad essa associati, senza la dipendenza dal third-party provider.

In questo elaborato si parlerà di SSI, soprattutto cosa è, mettendo in evidenza tutto ciò che rappresenta l'ecosistema di questo nuovo paradigma. Verrà descritto tutto ciò che serve a progettarla e realizzarla e gli standard che suggeriscono le linee guida su come creare l'identificativo univoco globale (DID) e come strutturare e scambiare le credenziali verificabili in modo sicuro (Verifiable Credentials). In particolare, verrà proposta una soluzione a ciò che lo standard non dice come deve essere fatto, ovvero come generare le chiavi d'identità del titolare e quindi quelle chiavi che servono poi a generare la prova di essere veramente lui il detentore dell'identità stessa e soprattutto come conservarle in modo sicuro, trattandosi di sequenze di bit e non più di un documento cartaceo ad esempio o semplicemente di una firma manuale.

Capitolo 2

Self-Sovereign Identity

La **Self-Sovereign Identity** (2), più comunemente abbreviata SSI, nasce come nuovo paradigma che offre agli individui il pieno controllo della propria identità digitale senza fare affidamento su terze parti utilizzabile attraverso qualsiasi servizio. Un sistema di identità digitale deve essere in grado di soddisfare tre requisiti fondamentali: sicurezza (bisogna essere protetti da furto d'identità, data breaches problemi di privacy dovuti a come gli intermediari gestiscono i dati dell'utente), controllo (colui che possiede l'identità deve avere il controllo su chi può vedere i suoi dati e per quale scopo), portabilità (si deve essere in grado di poter usare l'identità ovunque e non essere legati a specifici servizi, come ad esempio accade se si usa un third-party provider) (3).

Dopo vari studi, la SSI risulta essere appunto il modello finale che risponde a tutti e tre i requisiti citati; non c'è quindi più bisogno di terze parti per garantire l'identità, come avviene all'interno dei sistemi centralizzati e federati, e permette di avere il pieno controllo dell'identità e livelli di sicurezza elevati grazie all'uso delle tecnologie basate su Blockchain e Distributed Ledger.

Gli elementi tecnici alla base di questo ecosistema sono i **Decentralized Identifiers (DIDs)** e le **Verifiable Credentials (VCs)**.

2.1 Decentralized Identifiers

I DIDs sono un nuovo tipo di identificativi univoci globali che fanno riferimento ad un'identità digitale (4). Questi identificativi possono essere controllati

da una persona, un'organizzazione, un gruppo, delle cose ecc. Nello specifico il DID è un tipo di URI, quindi una sequenza di caratteri, che fa riferimento ad una risorsa globalmente univoca, ossia chi si vuole identificare, ed è composta da tre parti, come mostrato in figura.

Figura 2.1. Esempio di DID (4)

Nello specifico abbiamo:

1. una parte fissa denominata schema;
2. un DID method;
3. un identificativo univoco per il *DID method* specificato (è tipicamente generato usando funzioni di hash crittografiche).

La URI può essere anche estesa con altre componenti indicate dallo standard, come path, query e fragment, con lo scopo di dettagliare meglio a quale risorsa specifica si sta facendo riferimento, per esempio le chiavi crittografiche contenute nel DID document oppure delle risorse esterne al documento stesso (5). Il DID per sua definizione è solo un identificativo, perciò è utilizzato in combinazione con le Verifiable Credentials che rappresentano degli attributi che possono essere verificati e scambiati in modo sicuro. Il DID deve anche essere risolvibile, ciò significa che dalla URI si deve poter recuperare il DID document corrispondente.

Il **DID document** è appunto un documento o una struttura dati che contiene tutte le informazioni inerenti il DID. Ogni DID ha associato a sé esattamente un solo DID document. Questo tipicamente contiene una o più chiavi pubbliche, uno o più servizi associati. ad esempio degli endpoint che permettono di operare su alcuni servizi ed infine altri metadati (6). Il DID document può avere diverse rappresentazioni, quelle più comuni sono il formato JSON o JSON-LD (7). Di seguito troviamo un esempio di DID document:

```
{
  "@context": [
    "https://www.w3.org/ns/did/v1",
    "https://w3id.org/security/suites/ed25519-2020/v1"
  ]
  "id": "did:example:123456789abcdefghi",
  "authentication": [{
    "id": "did:example:123456789abcdefghi#keys-1",
    "type": "Ed25519VerificationKey2020",
    "controller": "did:example:123456789abcdefghi",
    "publicKeyMultibase": "
       zH3C2AVvLMv6gmMNam3uVAjZpfkcJCwDwnZn6z3wXmqPV"
  }]
}
```

Il possesso del DID e del DID document può essere verificato attraverso meccanismi crittografici. Il processo che permette la risoluzione di un DID nel DID document prende il nome di **DID resolution**; questo processo permette di recuperare il documento corrispondente per far sì che le applicazioni supportate possano utilizzare dati e metadati contenuti in esso per operazioni come verificare la firma digitale, autenticarsi con l'altra parte se ha bisogno di fare accesso in una web application oppure una applicazione mobile, social network ecc.

Il **DID controller** è colui che ha il diritto di apportare modifiche al DID document a lui associato. Quindi spesso *DID subject* e *DID controller* coincidono.

Il **DID method** è il meccanismo con cui un particolare tipo di DID e il suo DID document viene creato, risolto, aggiornato o revocato. Queste sono le quattro operazione base che possono essere effettuate su un DID e seguono il paradigma CRUD:

- Create - come il DID e il DID document possono essere creati.
- Resolve - come dal DID viene risolto il DID document.
- Update - come viene aggiornato il contenuto del DID document.
- Deactivate - come il DID viene revocato e non può essere più usato

Ogni *DID method* definisce una sua implementazione delle quattro operazioni base; ad esempio se basato su blockchain o DLT (Distributed Ledger) tipicamente le operazioni di create e update implicano una transazione sul ledger (6).

Per essere risolvibili nel corrispondente DID document, i DID devono essere memorizzati in qualche database o DLT che prende il nome di **Verifiable Data Registry**. Se vogliamo dare una definizione per un Verifiable Data Registry possiamo dire che questo corrisponde ad un registro o una base dati dove possiamo memorizzare i nostri DID e i corrispondenti DID document. Esempi di registri possono essere le blockchain, i distributed ledger, reti peer-to-peer e tutti sistemi che sono in qualche modo fidati. Le stesse operazioni CRUD fatte sul DID sono generalmente associate al tipo di Verifiable Data Registry che stiamo usando e a Verifiable Data Registry distinti.

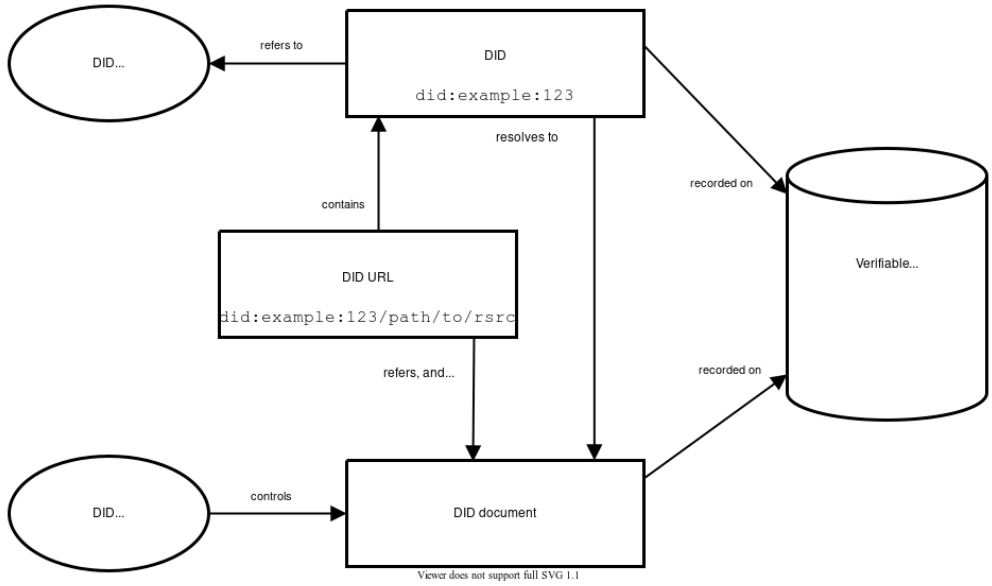

Figura 2.2. Architettura di un DID (4)

2.2 Verifiable Credentials

Una **verifiable credential** (8) è una struttura dati contenente tutta una serie di attributi legati ad un soggetto, serializzata in un formato verificabile attraverso operazioni crittografiche e che contiene una rappresentazione di

tutte le informazioni che possono essere descritte anche in una credenziale fisica. Nel mondo fisico le credenziali possono riferirsi a:

- Identità e quindi al soggetto possessore dell'identità stessa (il codice fiscale ad esempio);
- Autorità emettitrice della credenziale (ad esempio chi emette la carta d'identità);
- Tipo di credenziale (ad esempio patente di guida, numero di matricola del dipendente ecc.);
- Informazioni relative ad uno specifico attributo (ad esempio la nazionalità, data di nascita ecc.);
- Informazioni relative ad uno specifico vincolo (ad esempio la data di scadenza);
- Altre evidenze su come questa credenziale è stata generata.

Le verifiable credential rappresentano le stesse informazioni di una credenziale fisica ma con elementi aggiuntivi, ad esempio la firma digitale, che le rendono più affidabili perché a prova di manomissione rispetto a quelle fisiche. I tre attori principali dell'ecosistema delle verifiable credential, sono **issuer**, **holder** e **verifier**. La relazione che esiste tra queste tre entità viene chiamata **trust triangle**, che è fondamentale perché rappresenta un po' come la fiducia umana viene convogliata in un ambiente digitale (6). L'issuer è colui che crea le credenziali basandosi su un'asserzione verificata ricevuta dal soggetto di cui si vuole generare la credenziale. L'holder è quell'entità che possiede una o più verifiable credentials. Il verifier è colui che le riceve, le elabora e poi le usa per gli scopi destinati. Il trust triangle può essere schematizzato come segue:

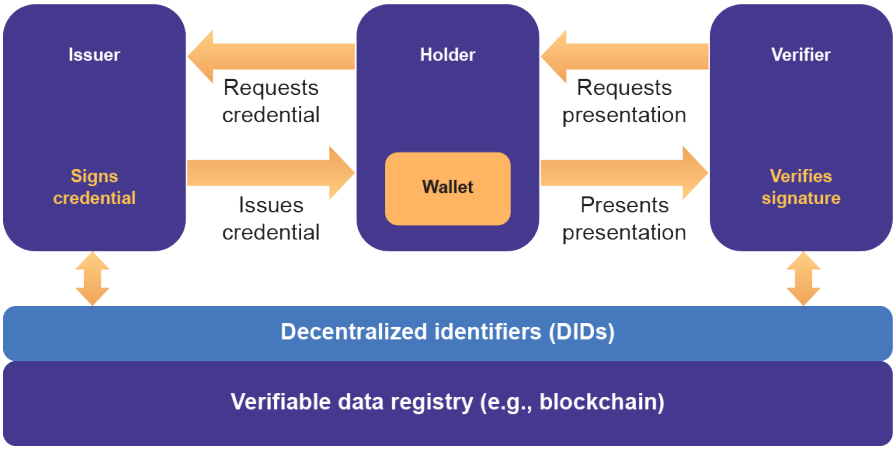

Figura 2.3. Trust triangle applicato alla blockchain (6)

Una **verifiable presentation** è una struttura che esprime i dati provenienti da uno o più verifiable credentials ed è creata in modo tale da verificarne autenticità, integrità e validità delle informazioni in esso contenute attraverso l'uso della crittografia.

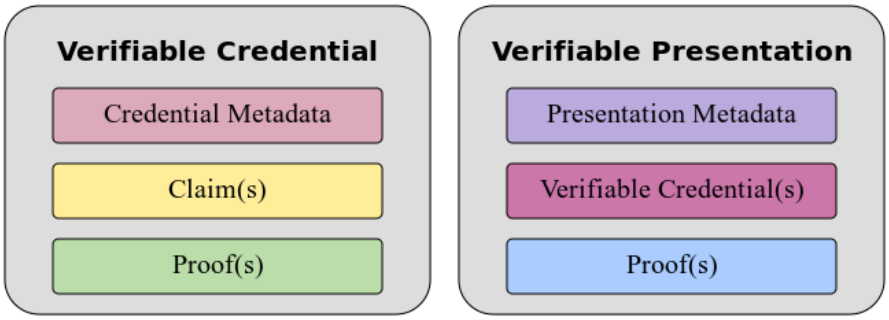

Figura 2.4. Diagramma del contenuto di una Verifiable Credential e di una Verifiable Presentation

Alcuni esempi di verifiable credential e verifiable presentation sono i seguenti (8):

```
{
  "@context": [
    "https://www.w3.org/2018/credentials/v1",
    "https://www.w3.org/2018/credentials/examples/v1"
  ],
```

```
  "id": "http://example.edu/credentials/1872",
  "type": ["VerifiableCredential", "AlumniCredential"],
  "issuer": "https://example.edu/issuers/565049",
  "issuanceDate": "2010-01-01T19:23:24Z",
  "credentialSubject": {
    "id": "did:example:ebfeb1f712ebc6f1c276e12ec21",
    "alumniOf": {
      "id": "did:example:c276e12ec21ebfeb1f712ebc6f1",
      "name": [{
        "value": "Example University",
        "lang": "en"
      }]
    }
  },
  "proof": {
    "type": "RsaSignature2018",
    "created": "2017-06-18T21:19:10Z",
    "proofPurpose": "assertionMethod",
    "verificationMethod": "https://example.edu/issuers/565049#
       key-1",
    "jws": "
       eyJhbGciOiJSUzI1NiIsImI2NCI6ZmFsc2UsImNyaXQiOlsiYjY0Il19
       ..TCYt5X
      sITJX1CxPCT8yAV-TVkIEq_PbChOMqsLfRoPsnsgw5WEuts01mq-pQy7
         UJiN5mgRxD-WUc
      X16dUEMGlv50aqzpqh4Qktb3rk-BuQy72IFLOqV0G_zS245-kronKb78
         cPN25DGlcTwLtj
      PAYuNzVBAh4vGHSrQyHUdBBPM"
  }
}
```

```
{
  "@context": [
    "https://www.w3.org/2018/credentials/v1",
    "https://www.w3.org/2018/credentials/examples/v1"
  ],
  "type": "VerifiablePresentation",

  "verifiableCredential": [{
```

```
    "@context": [
      "https://www.w3.org/2018/credentials/v1",
      "https://www.w3.org/2018/credentials/examples/v1"
    ],
    "id": "http://example.edu/credentials/1872",
    "type": ["VerifiableCredential", "AlumniCredential"],
    "issuer": "https://example.edu/issuers/565049",
    "issuanceDate": "2010-01-01T19:23:24Z",
    "credentialSubject": {
      "id": "did:example:ebfeb1f712ebc6f1c276e12ec21",
      "alumniOf": {
        "id": "did:example:c276e12ec21ebfeb1f712ebc6f1",
        "name": [{
          "value": "Example University",
          "lang": "en"
        }]
      }
    },
    "proof": {
      "type": "RsaSignature2018",
      "created": "2017-06-18T21:19:10Z",
      "proofPurpose": "assertionMethod",
      "verificationMethod": "https://example.edu/issuers/565049#
         key-1",
      "jws": "
         eyJhbGciOiJSUzI1NiIsImI2NCI6ZmFsc2UsImNyaXQiOlsiYjY0Il19
         ..TCYt5XsITJX1CxPCT8yAV-
         TVkIEq_PbChOMqsLfRoPsnsgw5WEuts01mq-pQy7UJiN5mgRxD-
         WUcX16dUEMGlv50aqzpqh4Qktb3rk-BuQy72IFLOqV0G_zS245-
         kronKb78cPN25DGlcTwLtjPAYuNzVBAh4vGHSrQyHUdBBPM"
    }
  }],
  "proof": {
    "type": "RsaSignature2018",
    "created": "2018-09-14T21:19:10Z",
    "proofPurpose": "authentication",
    "verificationMethod": "did:example:
       ebfeb1f712ebc6f1c276e12ec21#keys-1",
```

```
    "challenge": "1f44d55f-f161-4938-a659-f8026467f126",
    "domain": "4jt78h47fh47",
    "jws": "
       eyJhbGciOiJSUzI1NiIsImI2NCI6ZmFsc2UsImNyaXQiOlsiYjY0Il19
       ..kTCYt5XsITJX1CxPCT8yAV-TVIw5WEuts01mq-
       pQy7UJiN5mgREEMGlv50aqzpqh4Qq_PbChOMqsLfRoPsnsgxD-
       WUcX16dUOqV0G_zS245-kronKb78cPktb3rk-
       BuQy72IFLN25DYuNzVBAh4vGHSrQyHUGlcTwLtjPAnKb78"
  }
}
```


2.3 Distributed Ledger Tecnologies (DLTs)

Un **distributed ledger** è un sistema volto ad archiviare, condividere e sincronizzare dati digitali che sono distribuiti geograficamente su più siti (9). I dati memorizzati all'interno di questo sistema godono della proprietà di essere **immutabili** perché una volta memorizzato il dato questo non può essere più né aggiornato né cancellato. Infatti, le uniche due operazioni CRUD che possono essere fatte su un sistema di questo genere sono certamente la Create e la Read; Update e Delete non sono possibili. La decentralizzazione in un sistema di questo tipo è data da una rete peer-to-peer: il nodo riceve il dato, lo aggiunge e lo inoltra ai vicini. Ogni nodo, quindi, può aggiungere un nuovo record nel suo registro ma vengono appunto accettati solo quando il dato è verificato da tutti i nodi della rete e soddisfa un determinato requisito come può essere una firma digitale valida. Ogni nodo ha un suo protocollo di consenso per confermare le transazioni verificandone la firma e permettendo di mantenere tutti i dati allineati tra i peer. Altra caratteristica che possiamo assoggettare ad un DLT è che non ha un single-point-of-failure grazie alla sua natura distribuita, infatti ogni nodo previene la perdita e l'alterazione dei dati presenti, rendendo inoltre il sistema sempre disponibile. Tra i principali DLT possiamo trovare le **Blockchain** e i **Directed Acyclic Graph (DAG)**. Ciò che è stato proposto e sviluppato è compatibile con entrambi i tipi di DLT.

2.3.1 Blockchain

Una **blockchain** è un distributed ledger caratterizzato da record o transazioni che sono memorizzati come una catena di blocchi (da qui il nome blockchain) legati tra di loro mediante una relazione di tipo crittografico;

ogni blocco infatti contiene un hash del blocco precedente, un *timestamp* e i dati della transazione stessa. Ogni transazione inoltre dà origine a un nuovo blocco e su questo blocco viene calcolato l'hash attuale e apposto l'hash del blocco precedente con lo scopo di legare i blocchi. Questo lo rende immutabile perché se cambiano i dati della transazione cambia anche l'hash invalidando il tutto. Ogni nuovo record pubblicato viene condiviso con la rete dove altri *miners* ne verificano l'hash e i dati della transazione per confermarne la validità. Una volta verificato e accettato dal *miner* viene pubblicato nella catena. Questa operazione è un processo sequenziale, nel senso che non si può convalidare una transazione se la precedente non è stata verificata e quindi spesso diventa un processo molto oneroso, soprattutto in presenza di grossi carichi di transazioni. Il processo di verifica del blocco prende il nome di **protocollo del consenso (detto anche *Proof-of-Work*)** e rappresenta l'algoritmo che stabilisce le regole che i peers devono seguire nel sistema per concordare sullo stato della blockchain.

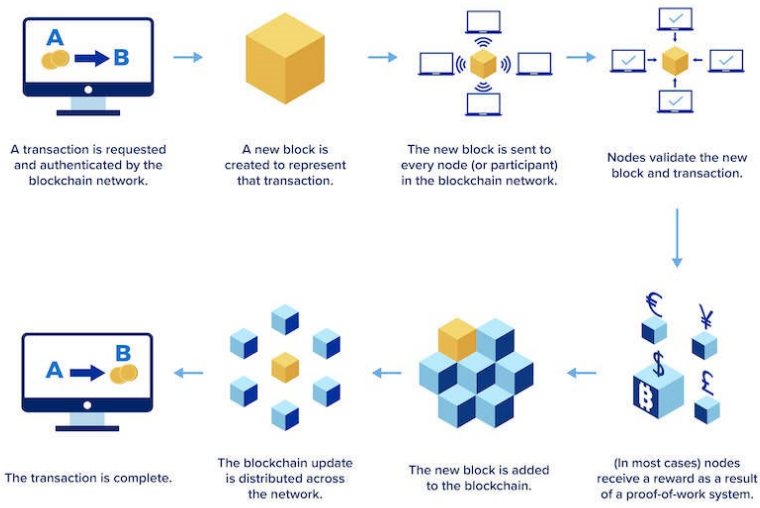

Figura 2.5. Processamento di una transazione in una blockchain (10)

2.3.2 Directed Acyclic Graph (DAG)

Un **DAG (Directed Acyclic Graph)** è un grafo diretto senza circuiti, se scegliamo un nodo qualsiasi del grafo percorrendo i vari archi non riusciremo mai a ritornare a quel nodo. Questo concetto matematico può essere

applicato anche ai DLTs. I vertici del grafo infatti possono rappresentare le transazioni così come gli archi rappresentano sostanzialmente la loro validazione nel distributed ledger. Un esempio di sistema che si basa su un DAG è Il **Tangle** sviluppato da **IOTA**. Il Tangle nasce come un tipo di DLT gratuita sviluppato per superare i limiti delle blockchain in termini di scalabilità e per ridurre il carico computazionale derivato dalla Proof-of-Work (11).

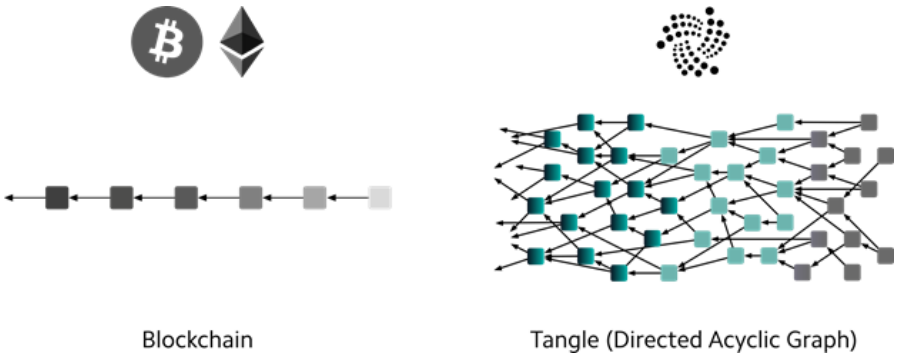

Figura 2.6. Differenza tra blockchain e IOTA (12)

Grazie al DAG le transazioni possono essere convalidate in modo parallelo e senza necessità di ordinamento. Questo processo prende il nome di ***transaction confirmation***. Analizziamone in linea di massima il funzionamento. Ogni nodo consiste in una transazione e contiene i suoi dettagli, l'hash e una firma digitale. I *tips* invece rappresentano i blocchi non ancora confermati, quindi le transazioni non ancora confermate. Ogni arco del grafo rappresenta un'approvazione. Di conseguenza, gli archi diretti relativi al nodo sono approvazioni dirette, il percorso indiretto invece rappresenta quelle che sono le approvazioni indirette. Il peso della transazione è direttamente proporzionale alla costruzione della transazione stessa mentre la somma pesata è data dal peso della transazione più i pesi delle transazioni che approvano direttamente o indirettamente la transazione in esame. Se la somma pesata raggiunge una determinata soglia dipendente dalla configurazione del DAG allora si ritiene confermata, altrimenti no. Il processo può essere schematizzato come in figura:

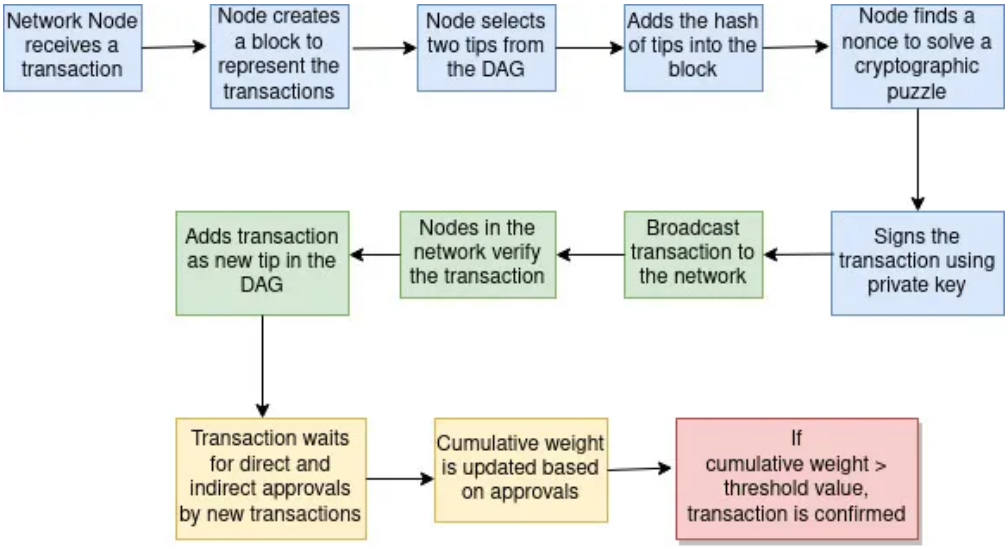

Figura 2.7. Protocollo di consenso in un DAG (13)

Ogni nodo che crea una transazione seleziona due transazioni non ancora approvate sulla base di un algoritmo. Ne consegue che ogni nodo approva due transazioni precedenti, che sono quelle collegate con archi diretti, verificandone prima la correttezza e che esse siano prive di conflitti, risolvendo una prova crittografica (detta puzzle) simile al proof-of-work. La transazione firmata viene trasmessa poi alla rete e agli altri nodi che la considereranno come tip suggerito. Infine il processo continua fino alla conferma (13). La sicurezza del sistema e il suo stato di integrità è basato appunto sulla conferma delle transazioni da parte degli utenti che ne emettono delle nuove; difatti se un nodo emette una transazione che approva transazioni che sono in conflitto, potrebbe non ottenere l'approvazione della sua da parte della rete che viene quindi rifiutata.

2.3.3 Smart contracts

Uno **smart contract** è una applicazione che viene eseguita all'interno di un distributed ledger, di solito una blockchain basata su Ethereum, dove più utenti eseguono e convalidano sempre lo stesso codice. Ciò garantisce che l'applicazione si comporti sempre alla stessa maniera e che non vi siano alterazioni significative nell'esecuzione del programma. Gli smart contracts sono programmi che vengono memorizzati nella blockchain in modo immutabile; una volta che viene effettivamente distribuito non può più essere

modificato. L'interazione con esso avviene mediante delle transazioni; in poche parole gli account che vogliono interagire con il contratto possono farlo inviando delle transazioni che eseguono una funzione definita all'interno dello smart contract (14). La struttura logica di uno smart contract può essere rappresentata come segue:

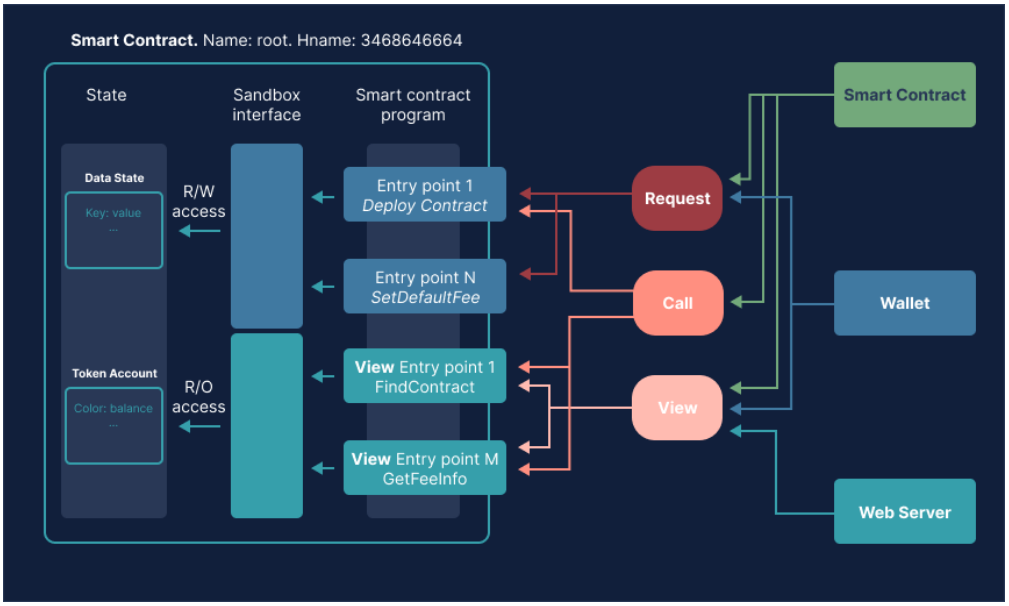

Figura 2.8. Struttura di uno smart contract (15)

Lo **stato** rappresenta i dati posseduti dal contratto e archiviati nella DLT. Tipicamente questi dati sono una collezione di coppie chiave/valore rappresentati come vettore di bytes di dimensione arbitraria. Si può pensare lo stato come una partizione dati per ogni catena del ledger dove lo smart contract può solo scriverci. Ogni smart contract poi ha al suo interno delle **funzioni** che eseguono determinate logiche applicative. Esistono due tipi di funzioni:

1. Funzioni che possono modificare, o meglio mutare, lo stato di uno smart contract.

2. Funzioni che fanno operazioni di sola lettura e quindi possono solo leggere lo stato del contratto, che vengono chiamate **viste**.

Quando viene eseguita una chiamata ad una funzione vengono prodotti in output tutta una serie di dati identificativi della transazione inerente la chiamata. Se la transazione è avvenuta correttamente allora avremmo come esito dati quali ad esempio il blocco in cui essa è avvenuta, la firma della stessa ecc. Oltre ai dati della transazione una funzione può emettere anche degli eventi che a loro volta possono contenere dati inerenti il contesto di esecuzione. In caso di errore, invece, semplicemente viene memorizzato l'errore che si è verificato (15). Il deployment del contratto all'interno di una blockchain è anch'esso una transazione e così come anche le chiamate alle funzioni hanno bisogno di Gas per poter essere eseguite. Il **Gas** rappresenta l'unità di misura dell'effort necessario per eseguire le operazioni su un distributed ledger. Questo serve anche per garantire che la rete sia protetta da spam e non rimanga bloccata in cicli infiniti. Il costo totale è rappresentato dalla quantità di gas utilizzata moltiplicata per il costo per unità di Gas e viene addebitato sia in caso di successo della transazione che in caso di fallimento. Benché ogni transazione abbia comunque un limite di Gas rappresentante il massimo valore necessario per quella transazione, quello che non viene speso nell'esecuzione viene restituito all'utente (14). Quando un contratto viene distribuito all'interno della blockchain viene creato un **contract account** ossia un indirizzo utilizzabile dal chiamante per interagire con lo smart contract; quindi inviare una transazione a questo indirizzo vuol dire appunto eseguire una funzione dello smart contract. Uno smart contract può essere scritto mediante linguaggi di programmazione specifici come Solidity. Solidity è un linguaggio ad oggetti ad alto livello specializzato per la scrittura di smart contracts. Il codice sorgente del contratto viene poi compilato e si ottengono due elementi: il **bytecode** che contiene tutte le istruzioni macchina per eseguire quanto dichiarato nel contratto e l'**Application Binary Interface (ABI)** che corrisponde ad un file JSON che descrive il contratto e le sue funzioni. Le istruzioni contenute nel bytecode vengono infine eseguite sul nodo all'interno della **Ethereum Virtual Machine (EVM)**.

Capitolo 3

Stronghold

3.1 Che cosa è Stronghold

Stronghold (16) nasce come una libreria open-source, scritta in linguaggio Rust, che può essere utilizzata con lo scopo di proteggere qualsiasi informazione sensibile come chiavi private, seed usati per la generazione delle chiavi e altri dati utente che non devono essere esposti all'interno di un nodo. Può essere visto come un contenitore isolato dove c'è un database protetto che fa da custode di dati sensibili e che espone delle procedure per lavorare con dati del *vault*, archiviarli in modo sicuro e anche eseguire gli algoritmi crittografici che permettono di generare e derivare coppie di chiavi, generare seed, generare una firma digitale ed altre operazioni ancora. L'accesso ai dati sensibili può essere effettuato mediante delle procedure apposite; non vi è possibilità di accederci direttamente. Le componenti principali di Stronghold sono due:

1. Client.
2. Snapshot.

Il **Client** si occupa di effettuare le operazioni crittografiche. Quindi è quello che esegue le chiamate alle procedure. È un container la cui istanza viene creata appositamente per quel contesto di esecuzione, che include tutte le funzionalità necessarie per lavorare con le informazioni sensibili memorizzate.

Lo **Snapshot** è la componente che contiene tutte le funzionalità per rendere persistente lo stato del Client. Esso in realtà ha duplice funzionalità. A

livello più basso consiste in un file cifrato che contiene la fotografia inerente l'ultima chiamata di salvataggio dell'istanza in memoria del client e quindi di tutte le aree di memoria popolate con le informazioni sensibili che sono state inserite o manipolate. La cifratura del file avviene mediante le stesse procedure di Stronghold. A livello più alto invece abbiamo lo **Store** che invece è una cache (mappa chiave-valore) per archiviare dati non sensibili, come ad esempio i dati di configurazione della sessione. Nota bene che la memorizzazione e la lettura dello stato dallo snapshot, essendo cifrata, necessita di una password che deve essere fornita ogni qualvolta è necessaria un'operazione di questo tipo.

Le operazioni crittografiche sono eseguite in Stronghold seguendo delle pipeline ben precise; questo pattern nasce come soluzione al poter chiamare più funzioni insieme a catena. Ogni stage della pipeline ha una sua logica ben precisa e può quindi produrre un valore (ad esempio la generazione di una chiave), processare un dato già esistente (ad esempio derivare una chiave a partire da una chiave master o un seed) e infine esportarne il valore (ad esempio ritornare il valore di una chiave pubblica associata alla privata data in input). A ogni stage della pipeline inoltre viene assegnata un'area di memoria all'interno del *vault* definita appositamente per accedere o per memorizzare il dato dello stage.
Facciamo un esempio pratico. Supponiamo di voler generare una coppia di chiavi privata/pubblica, derivare una chiave secondaria ed esportare la chiave pubblica associata. Il primo step della pipeline che chiamerà la routine atta a produrre la coppia di chiavi lavorerà con un'area di memoria ad essa associata (chiamiamola *location1*); la funzione genera la chiave privata e la salva all'interno di quest'area. Il secondo step che deriverà la nuova chiave a partire dalla chiave master precedentemente creata dovrà accedere all'area in cui è memorizzata quella chiave in sola lettura (accederà quindi a *location1* per leggere la chiave) e poi memorizzerà la chiave derivata all'interno di un'altra area di memoria che chiameremo *location2*. Infine il terzo stage che si occuperà di esportare la chiave pubblica accederà a *location2* per leggere la chiave privata derivata, la routine genererà la chiave pubblica associata e la restituirà in output come risultato della funzione, senza memorizzazione. Il processo può essere schematizzato come segue:

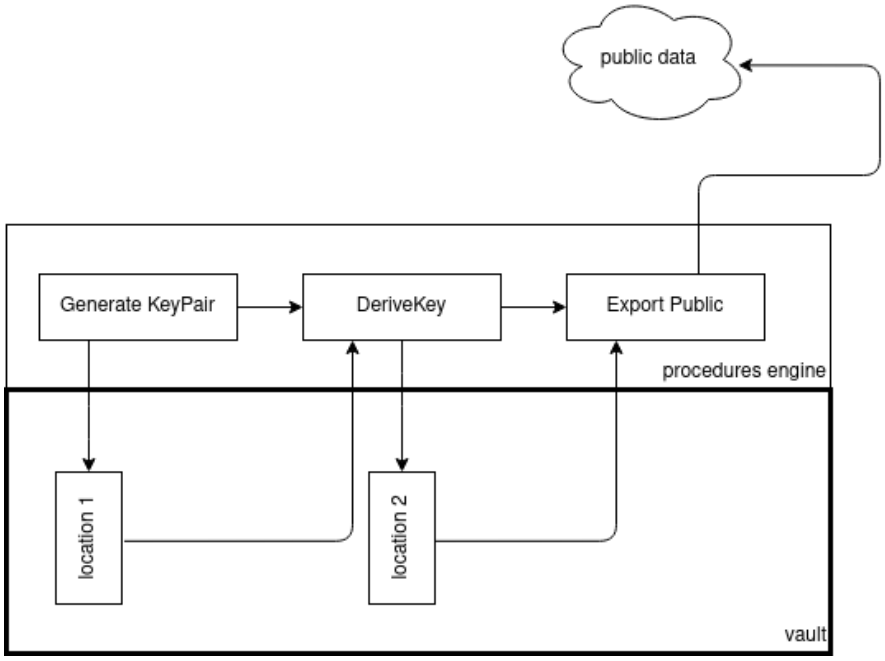

Figura 3.1. Struttura di una pipeline Stronghold (17)

Stronghold gestisce tre tipi di primitive che si basano appunto sulle pipeline. Queste sono (17):

1. **Generator / Source** che genera delle chiavi oppure seeds. Nella maggior parte dei casi sono funzioni che generano una coppia di chiavi privata/pubblica oppure una sequenza mnemonica per generare un seed per wallet deterministici. Appartengono a questa categoria procedure come la generazione di una sequenza mnemomica basata su standard BIP39.

2. **Processor** che prendono in input un riferimento ad un'area di memoria dove presumibilmente è memorizzato il dato segreto di partenza, che sia esso una chiave padre o un seed, derivano un'altra chiave da questi e la memorizzano in una seconda area di memoria generando come output il riferimento a quell'area. Appartengono a questa categoria procedure come la derivazione di chiavi basata su BIP44 e Slip10.

3. **Receiver / Sink** che prendono in input un riferimento ad un'area di memoria dove presumibilmente è memorizzato il dato di partenza, producono un valore e lo ritornano senza memorizzare nulla nel vault.

Un ultimo cenno relativo a questo argomento va fatto in merito alla gestione della memoria sicura utilizzata dalla libreria. Per ragioni di sicurezza e per evitare che possa essere generato un dump (una copia) della memoria sicura dove Stronghold registra i dati sensibili, si è scelto di utilizzare uno schema di allocazione della memoria di tipo non-contiguo. Ovviamente il meccanismo dipende anche dal sistema operativo del nodo ospitante. L'astrazione scelta da Stronghold è denominata **Boxed Memory**, dove è un meccanismo di guardia (guard) che rende l'area inaccessibile, di sola lettura oppure con permessi di scrittura/lettura. La Boxed Memory appone un blocco (lock) alla memoria allocata e ne impedisce la copia di quella specifica area; difatti viene impedito che l'attuale spazio virtuale in memoria del processo venga paginato in un'area di swap e che quindi venga poi esposto alla lettura esterna.

3.2 Bitcoin Improvement Proposal - BIP

Il **BIP**, acronimo di **Bitcoin Improvement Proposal** (18), è un documento di testo nel quale sono contenute le linee guida, quindi design e progettazione, delle nuove funzionalità per bitcoin. Ogni proposta è mantenuta appunto in formato testuale attraverso un sistema di versioning che ne rappresenta lo storico. É di fatto il meccanismo principale per proporre le nuove funzionalità di bitcoin, raccogliere opinioni su un determinato problema o documentare le scelte di progettazione di una determinata funzione.

3.2.1 BIP-32

BIP-32 descrive tutto ciò che è inerente alla generazione di wallet deterministici gerarchici (19). Un **wallet deterministico** è un sistema per derivare chiavi usando come unico punto di partenza un **seed**. Questo seed ha anche la funzionalità di essere usato dall'utente per poter fare facilmente il backup o un ripristino del wallet senza bisogno di usare altri dati. Tipicamente il seed è una sequenza mnemonica di parole, come definito nel BIP-39. I primi client utilizzati nelle blockchain generavano un buffer di chiavi private pseudocasuale che venivano poi usate per generare indirizzi sul quale l'utente poteva ricevere le transazioni; questi indirizzi erano usati anche in caso di backup. La possibilità di fare backup, ad esempio in caso di trasferimento su un altro dispositivo, veniva meno quando questo pool di chiavi si esauriva, con la conseguenza che non si potevano ricevere più transazioni. Inoltre è onere del detentore del wallet conservare tutte le chiavi private generate

e che insieme a quanto detto precedentemente portava ad un meccanismo inefficiente (Just-a-Bunch-of-Keys). Grazie ai wallet deterministici questo problema non sussiste perché mediante uno schema ben definito è possibile generare, a partire da un unico seed, un numero illimitato di indirizzi ben noti perché derivati a mezzo di un algoritmo. Qualora ci sia necessità può essere usato il solo seed per fare il backup. Si utilizza la crittografia a curve ellittiche per derivare le chiavi dal seed, nello specifico la curva è **secp256k1**. Il BIP definisce una funzione che deriva delle chiavi figlie a partire da una chiave padre. Le chiavi secondarie non derivano esclusivamente dalla chiave padre, ma si utilizza anche un valore entropico di 256 bit (32 bytes) denominato **chain code**. Va detto che le 2^{32} chiavi che possono essere generate sono suddivise in 2^{31} chiavi *normali*, ovvero quelle che hanno indice da 0 a 2^{31-1} e 2^{31} chiavi *rafforzate (hardened)* ovvero quelle che hanno indice da 2^{31} a 2^{32-1}. Il perché dell'aggiunta di chiavi hardened è spiegato in successive considerazioni sulla derivazione delle chiavi.
Data una chiave padre estesa con un chain code e un indice i, è possibile calcolare una chiave figlia estesa solo nei seguenti casi:

1. chiave privata padre $\rightarrow$ chiave privata figlia;

2. chiave pubblica padre $\rightarrow$ chiave pubblica figlia;

3. chiave privata padre $\rightarrow$ chiave pubblica figlia.

Il caso in cui da una chiave pubblica padre si calcola una chiave privata figlia non è possibile.
Senza entrare nel dettaglio dell'algoritmo, che verrà poi generalizzato con lo standard SLIP-0010, che sarà illustrato successivamente, notiamo che quello che viene fuori da questa applicazione è un albero di chiavi. Infatti si parte dalla radice, dove abbiamo la chiave privata estesa `m`, se valutiamo di derivare le chiavi derivate a partire dal padre considerando un generico indice *i* avremmo un numero *i* di livello `n-1` chiavi derivate, che a loro volta ognuna di queste è ancora una chiave estesa ovviamente derivabile, e così via, generando appunto un albero di chiavi.

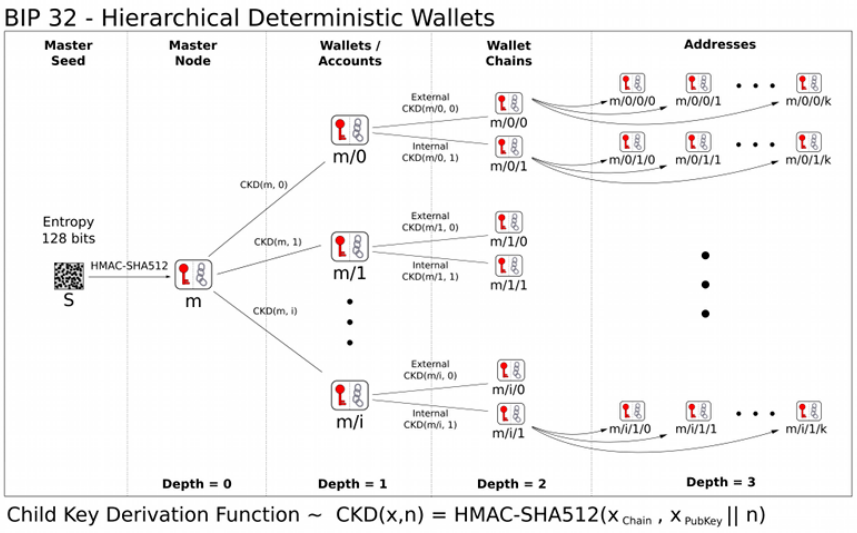

Figura 3.2. Albero di chiavi derivate ottenute dall'applicazione dello standard BIP-32 (19)

Ogni nodo foglia dell'albero corrisponde ad una chiave vera e propria, mentre i nodi interni corrispondono alle collezioni di chiavi che discendono da esse. Possiamo generalizzare la notazione come segue (19)

- per chiavi derivate normali:
  $N(m/a/b/c) = N(m/a/b)/c = N(m/a)/b/c = N(m)/a/b/c = M/a/b/c$.
- per chiavi derivate hardened:
  $N(m/a_H/b/c) = N(m/a_H/b)/c = N(m/a_H)/b/c$.

Per come vengono costruite le chiavi conoscere una chiave primaria estesa permette la ricostruzione di tutte le chiavi private estese discendenti e di conseguenza anche le pubbliche, invece conoscere una chiave pubblica estesa permette di ricostruire tutte le chiavi pubbliche discendenti non rafforzate però. Quindi attenzione a conservare le chiavi, soprattutto quelle estese intermedie, che corrispondono all'intero sotto-albero di chiavi. La conoscenza di una chiave pubblica estesa padre più una qualsiasi chiave privata non hardened che discende da essa equivale a conoscere anche la chiave privata padre. Ciò significa che le chiavi pubbliche estese devono essere trattate con maggiore attenzione rispetto alle chiavi pubbliche normali. È anche la ragione dell'esistenza delle chiavi rafforzate e del motivo per cui vengono utilizzate a livello di account nell'albero. In questo modo, perdere una chiave privata non rischia mai di compromettere l'account principale o gli altri.

3.2.2 BIP-39

BIP-39 nasce come proposta per descrivere come implementare una sequenza mnemonica, ovvero una sequenza di parole facile da ricordare, dalla quale poi verrà ricavato il seed che serve per la generazione di wallet deterministici (20). Nella scelta delle parole da inserire nella lista, che poi diventa la base di partenza per individuare la sequenza, bisognerebbe evitare parole simili tra loro, fare in modo che digitando i primi quattro caratteri la parola sia facilmente completabile e infine fare in modo che le parole siano ordinate nella sequenza. La sequenza poi deve essere normalizzata, ovvero non devono essere presenti caratteri nativi della lingua che potrebbero non essere codificati correttamente in termini informatici. Il numero di parole scelto da inserire nella sequenza è ricavato partendo da un parametro denominato entropia (*entropy*), multiplo di 32 bit. Il passaggio da questo parametro di entropia al numero totale di parole può essere riassunto come segue:

- Si sceglie il valore dell'entropia (tipicamente compreso tra 128 e 256 bit), supponiamo di indicarlo con ENT;
- Si calcola la checksum, indicata con CS, come ENT/32;
- Il numero di parole, indicato con MS, è dato infine da (ENT + CS)/11.

Infatti con un'entropia di 256 bit il numero di parole che compongono la sequenza mnemonica è pari a 24.
Per proteggere il mnemonico l'utente può anche scegliere eventualmente una passphrase. Ultimo step è passare da una sequenza di parole ad un seed in formato binario. Si richiama la funzione crittografica **PBKDF2** usando come password la sequenza normalizzata e come sale (*salt*) la concatenazione della parola "mnemonic" con la passphrase scelta dall'utente (verrà usata una sequenza vuota nel caso in cui non si sia scelta una password). Viene impostato un numero di iterazioni pari a 2048 e infine viene applicata la funzione **HMAC-SHA512** generando un seed di 512 bit (64 bytes). Il seed è poi usato come input per le funzioni di generazione di wallet deterministici, come descritto nel BIP-32.

3.2.3 SLIP-0010

Lo **SLIP-0010** (21) è sostanzialmente una generalizzazione di ciò che è stato proposto nel BIP-32 e permette di derivare chiavi pubbliche o private, che siano primarie o figlie, da tipi di curve che sono differenti da secp256k1,

ovvero quella usata in bitcoin. Partendo da una sequenza mnemonica e una passphrase, seguendo il protocollo BIP-39, possiamo calcolare il seed dal quale possiamo creare diverse chiavi padre, una per ogni tipo di curva. Le curve supportate da questo algoritmo sono **NIST P-256** e **ed25519**, oltre che anche la curva secp256k1. Per evitare chiavi padre non valide, se la chiave appunto non è valida, l'algoritmo viene nuovamente invocato utilizzando l'hash intermedio come nuovo seed. Tramite questo algoritmo è possibile derivare la chiave padre e a partire dalla chiave padre è possibile fare le seguenti derivazioni:

1. chiave privata padre $\rightarrow$ chiave privata figlia;
2. chiave pubblica padre $\rightarrow$ chiave pubblica figlia (questa derivazione non è valida però per curve di tipo ed25519);

Avendo un seed S di lunghezza compresa tra 128 e 512 bit, si può derivare la chiave padre nel seguente modo:

1. Si calcola il valore di I = HMAC-SHA512(Key = Curve, Data = S);
2. Si divide il valore precedentemente calcolato in due sequenze di 32 bytes definendo I_L e I_R;
3. Si usa la funzione $parse_{256}(I_L)$ come chiave padre e I_R come chain code padre (notare che la funzione $parse_{256}$ interpreta la sequenza di 32 bytes come un numero su 256 bit);
4. se la curva non è di tipo ed25519 e I_L è 0 oppure $\geq$ n, dove n è il modulo intero della curva (chiave non valida), allora S = I e ritorna allo step 2.

Nella generazione delle chiavi figlie bisogna sottolineare prima alcuni aspetti. Il primo è che la derivazione delle chiavi figlie per curve di tipo NIST P-256 è identica a quella delle curve secp256k1 ovviamente usando i parametri di dominio per la curva di quel tipo. Il secondo aspetto è che con curve di tipo ed25519 è possibile generare solo chiavi private figlie di tipo hardened. Dopo le seguenti affermazioni possiamo vedere nel dettaglio l'algoritmo di generazione delle chiavi figlie nelle varie casistiche.

Si analizzi ora il caso in cui da una chiave privata padre si voglia derivare una chiave privata figlia di indice *i*. Indichiamo con *n* l'ordine della curva. I passi sono seguenti:

1. Si verifica se la chiave figlia da generare è normale o hardened, ovvero $i \geq 2^{31}$

- se la condizione non è verificata e quindi in caso di chiave figlia normale, allora si avrà che se la curva è di tipo ed25519 verrà restituito errore, altrimenti si calcola il valore di I = HMAC-SHA512(Key = c_{par}, Data = $ser_P(point(k_{par}))$ || $ser_{32}(i)$).
- se la condizione è verificata ovvero in caso di chiave figlia hardened, allora si calcola I = HMAC-SHA512(Key = c_{par}, Data = 0x00 || $ser_{256}(k_{par})$ || $ser_{32}(i)$). (Alla chiave viene aggiunto del padding che la rende di lunghezza 33 bytes).

2. Si divide I in due sequenze da 32 bytes ognuno denotate con I_L and I_R.
3. Il chain code ritornato c_i è I_R.
4. Se la curva è di tipo ed25519 allora k_i è data da $parse_{256}(I_L)$.
5. Se $parse_{256}(I_L) \geq n$ oppure $parse_{256}(I_L) + k_{par} \pmod n = 0$ (caso in cui la chiave è invalida):
   - I = HMAC-SHA512(Key = c_{par}, Data = 0x01 || I_R || $ser_{32}(i)$ e si ritorna allo step 2.
   - $k_i = parse_{256}(I_L) + k_{par} \pmod n$. Questo è il valore della chiave derivata.

Si analizzi invece il caso di derivazione da una chiave pubblica padre di una chiave pubblica figlia, ricordando che questa casistica **non è valida per curve ed25519** ed **è definita solo per chiavi derivate non-hardened**.

1. Si verifica se la chiave figlia da generare è normale o hardened, ovvero $i \geq 2^{31}$
   - se hardened allora verrà restuituito errore
   - se normale si calcola I = HMAC-SHA512(Key = c_{par}, Data = $ser_P(K_{par})$ || $ser_{32}(i)$).
2. Si divide I in due sequenze da 32 bytes ognuno denotate con I_L and
3. La chiave derivata figlia è data da $K_i = point(parse_{256}(I_L)) + K_{par}$.
4. Il chain code ritornato c_i è I_R.
5. Se $parse_{256}(I_L) \geq n$ oppure K_i è un punto infinito (caso di chiave non valida):
   - I = HMAC-SHA512(Key = c_{par}, Data = 0x01 || I_R || $ser_{32}(i)$) e si ritorna allo step 2.

Capitolo 4

Progettazione e Realizzazione

Gli standard W3C che fanno riferimento alla Self-Sovereign Identity non forniscono indicazioni su come generare le chiavi d'identità relative all'utente che detiene il DID. L'utilizzo di chiavi randomiche non è una buona soluzione in quanto l'utente si ritroverebbe costretto a custodire ogni singola chiave privata e il backup dell'identità sarebbe compromesso in caso di esaurimento del pool di chiavi random. Gli standard BIP invece hanno descritto, in particolare il BIP-32, come implementare una gerarchia di chiavi, ovvero un insieme di chiavi che alla fine andranno a formare una struttura ad albero. Il primo problema da affrontare è come definire il percorso di derivazione delle chiavi, a cui segue, come custodire in modo sicuro tutte le informazioni sensibili che ne verrebbero fuori come il seed di derivazione delle chiavi e le chiavi generate.

4.1 Gerarchia per la generazione delle chiavi d'identità

La soluzione al primo problema rispecchia quanto descritto già dallo standard BIP-32. Si è definita una gerarchia per la generazione delle chiavi d'identità basata su 6 livelli di profondità:

```
m / purpose / registry / method_type / method_identifier / verification_method
/ index
```

Con **m** è indicato il seed generato secondo le specifiche dettate dallo standard BIP-39 dal quale sarà generata una chiave privata padre.

Il **purpose** è un valore costante pari a 1361 e indica il BIP di riferimento che riporta le specifiche dello standard.

Il **registry** è un valore che indica quale è il DLT che ha il ruolo di Verifiable Data Registry. Attualmente i valori scelti sono i seguenti:

Valore	**Descrizione**
0x00000001	IOTA
0x00000002	Ethereum-based blockchain

Il **method_type** è il parametro indicante il DID method utilizzato. Anche per questo campo si sono definiti i seguenti valori:

Method Type	**Valore**	**Descrizione**
:iota	0x00000001	IOTA
:eth	0x00000002	Ethereum-based blockchain

Il parametro **method_identifier** invece si riferisce al *DID method identifier* e rappresenta l'identificativo univoco all'interno del *DID method*.

Il **verification_method** è il riferimento a quale scopo è associata la chiave derivata, ad esempio per autenticarsi, per la firma digitale ecc. I metodi attualmente censiti sono i seguenti:

Valore	**Descrizione**
0x00000000	Verification Material
0x00000001	Authentication (Verification Relationship)
0x00000002	Assertion (Verification Relationship)
0x00000003	Key Agreement (Verification Relationship)
0x00000004	Capability Invocation (Verification Relationship)
0x00000005	Capability Delegation (Verification Relationship)

Infine il parametro **index** indica l'indice della chiave derivata, ricordando che gli indici che vanno da 0 a 2^{31-1} indicano chiavi normali e indici da 2^{31} a 2^{32-1} indicano chiavi rafforzate (hardened).

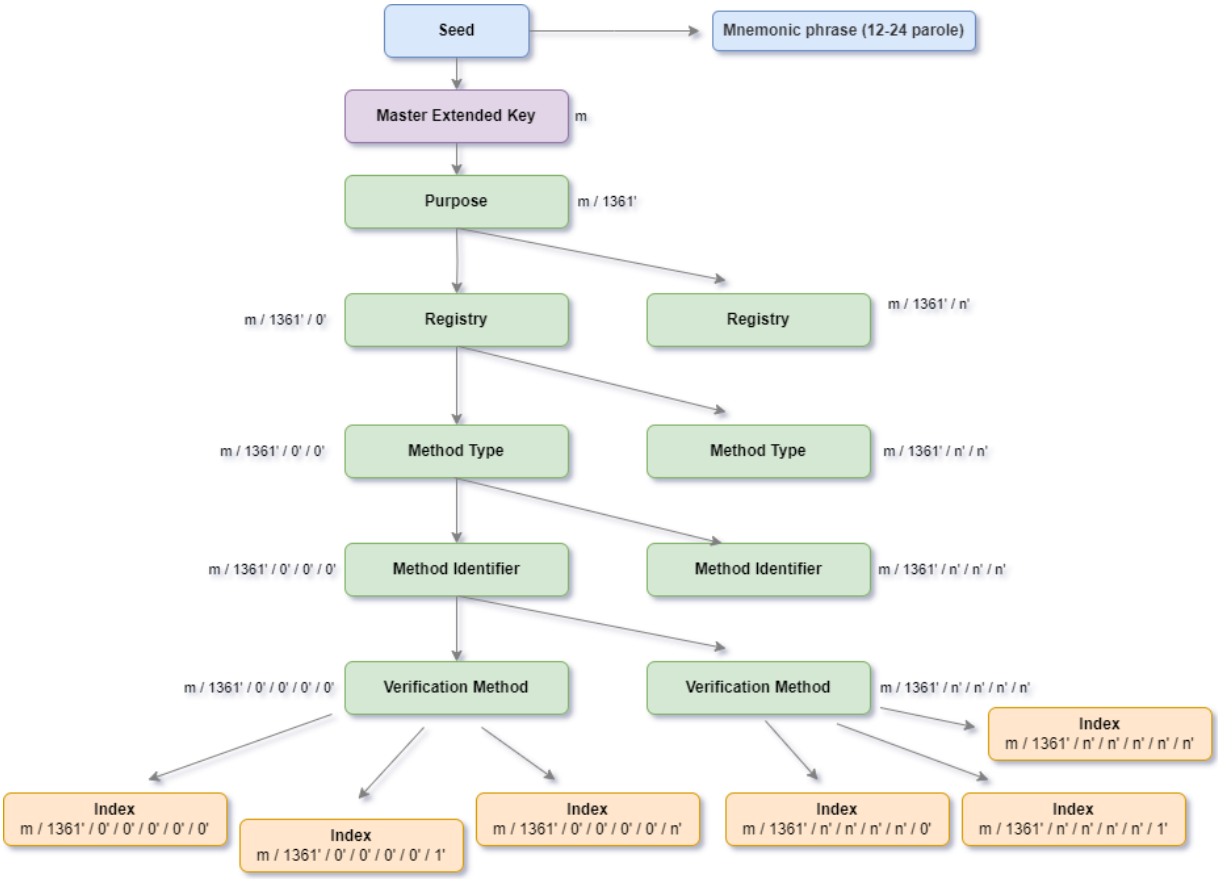

Figura 4.1. Albero della gerarchia definita per le chiavi di identità

4.2 Estensione di IOTA Stronghold

Definita la gerarchia delle chiavi va ora implementata nella procedura di derivazione delle chiavi. Stronghold implementa al suo interno delle procedure che permettono di generare il seed basato su BIP-39 e procedure di derivazione delle chiavi basate su standard SLIP-10 (algoritmo generalizzato dello standard BIP-32 utilizzante anche altre curve ellittiche) e i dati generati sono conservati in maniera sicura attraverso la stessa libreria perché Stronghold è di fatto pensato per essere un contenitore isolato per conservare queste informazioni sensibili.

Partendo da queste basi, è stata sviluppata in Stronghold la procedura che

permette di derivare delle chiavi gerarchiche implementanti il path definito in precedenza. La procedura usa come algoritmo di derivazione delle chiavi sempre quello definito in SLIP-10 utilizzando una curva di tipo ed25519. I parametri di input passati alla funzione sono gli stessi definiti del path. La struttura dichiarata contiene come parametri un riferimento all'area di memoria dove è memorizzato il seed o la chiave privata padre, il registry, il method type, l'indirizzo del contratto come method identifier, il verification method, l'indice della chiave e infine il riferimento all'area di memoria dove deve essere memorizzata la chiave privata derivata generata dalla funzione. Di seguito la definizione:

```
pub enum DidKeyDeriveInput {
    /// Note that BIP39 seeds are allowed to be used as SLIP10
   seeds
    Seed(Location),
    Key(Location), // Master key into Vault
}

pub struct DidKeyDerive {
    pub input: DidKeyDeriveInput,
    pub registry: u32,
    pub method_type: u32,
    pub contract_addr: String, /// Hex encoded did-contract
   address
    pub verification_method: u32,
    pub index: u32,
    pub output: Location,
}
```

La procedura genera il chain code a partire dagli attributi in ingresso, tenendo conto che nel caso di curva ed25519 sono permesse solo la generazione di chiavi derivate di tipo hardened, come da standard. Una volta generato il chain code viene passato come parametro alla funzione che si occupa di generare la chiave derivata. La procedura interna SLIP-10 deriverà la chiave eseguendo gli step dell'algoritmo analizzato nella parte introduttiva e infine la memorizza, ritornando un codice di successo nel caso in cui la procedura sia andata a buon fine, fornendo anche il valore del chain code calcolato, oppure errore in caso di qualsiasi errore di derivazione o accesso alla memoria.

```
// -- Start Polito DID research section
```

```
const PURPOSE: u32 = 1361;

fn derive(self, guards: [Buffer<u8>; 1]) ->
Result<Products<ChainCode>, FatalProcedureError> {

        let sanitized_addr = sanitize_addr(self.contract_addr);
        let mut decoded_addr = [0; 20];
        hex::decode_to_slice(sanitized_addr, &mut decoded_addr).unwrap
   ();

        let firsts = &decoded_addr[0..4].try_into().unwrap();
        let account = byte_array_as_u32(firsts);

        let chain = Chain::from_u32_hardened(vec![
            PURPOSE, // BIP proposal
            self.registry, // registry
            self.method_type, // method type
            account, // account (i.e. first 32 bits of the address of
   the deployed contract)
            self.verification_method, // verification method
            self.index, // index
        ]);

        let dk = match self.input {
            DidKeyDeriveInput::Key(_) => {
                slip10::Key::try_from(&*guards[0].borrow())
                .and_then(|parent| parent.derive(&chain))
            }
            DidKeyDeriveInput::Seed(_) => {
                slip10::Seed::from_bytes(&guards[0].borrow())
                .derive(slip10::Curve::Ed25519, &chain)
            }
        }?;
        Ok(Products {
            secret: dk.into(),
            output: dk.chain_code(),
        })
    }

// Utils functions

// Convert four-bytes array into u32
fn byte_array_as_u32(array: &[u8; 4]) -> u32 {
```

```
    ((array[0] as u32) << 24) |
    ((array[1] as u32) << 16) |
    ((array[2] as u32) <<  8) |
    ((array[3] as u32) <<  0)
}

// Remove the 0x from the address string if present
fn sanitize_addr(mut addr: String) -> String {
    if addr.contains("0x") {
        addr = addr.replace("0x", "");
    }

    //println!("Sanitized address: {}", addr);
    return addr;
}

// -- End Polito DID research section
```

Il valore assegnato al *method_identifier*, che farà riferimento all'indirizzo dello smart-contract del DID method, è calcolato prendendo solamente i primi 32 bits dell'indirizzo del contratto che saranno poi rappresentati su un intero sempre di 32 bits senza segno. Ad esempio se l'indirizzo del contratto corrisponde al valore
`0x24a0ca094f13fe1376f8f79e58a7b192da5a7e01`,
i primi 32 bit sono dati da `24a0ca09` la quale conversione del dato è pari a `614517257`.
L'area di memoria dove la chiave viene memorizzata è costruita con due componenti: il primo che indica il percorso identificante il *vault*, generato tramite concatenazione della variabile d'ambiente da definire con chiave `DID_POLITO_STRONGHOLD_BASE_VAULT_PATH` e l'indirizzo dello smart-contract, mentre invece ciò che indica il record è dato dalla concatenazione dei parametri di derivazione al netto dell'indirizzo dello smart contract, già presente nel *vault path*.
È di nostro interesse però anche recuperare la chiave pubblica associata alla chiave privata derivata. Rispecchiando i principi di isolamento su cui si fonda Stronghold, una funzionalità di questo genere deve essere eseguita in uno step a parte. Altra operazione che potrebbe essere di nostro interesse è la possibilità di rimuovere una chiave derivata che per qualsiasi ragione può essere invalidata. Quindi mantenendo le best practices date definiamo due pipeline, una per l'aggiunta di una chiave derivata a partire dal seed generato in fase di creazione dell'identità e una per la rimozione di una

chiave identificata da determinati attributi che costituiranno il record path, ovvero il percorso in memoria dove la chiave è memorizzata.

Definita la struttura **DidKey**, che contiene quanto serve per derivare o cancellare le chiavi, come segue

```
pub struct DidKey {
    pub passphrase: String,
    pub seed_location_id: String,
    pub contract_address: String,
    pub registry: u32,
    pub method_type: u32,
    pub verification_method: u32,
    pub index: u32,
}
```

sono state aggiunte due funzioni dichiarate come `add_did_key()` e `remove_did_key()` che implementano appunto queste due pipeline. La funzione di aggiunta della chiave esegue due step. A partire dai parametri definiti nella struct `DidKey`, richiama la procedura `DidKeyDerive` che genera la chiave derivata e la memorizza nell'area indirizzata come descritto in precedenza. A questo punto interviene la procedura `PublicKey` che legge dalla posizione in memoria la chiave privata derivata e ne genera la pubblica che verrà poi ritornata dalla stessa funzione insieme al record path per recuperare la privata quando richiesto o errore in caso di fallimento. La funzione di rimozione della chiave, in modo duale, dopo controlli sull'esistenza del dato che si vuole rimuovere, procede alla cancellazione della chiave da quell'area di memoria. Ultimo step, che viene eseguito indistintamente da una funzione o dall'altra, è quello di aggiornare lo stato dello Snapshot con quanto eseguito del client che si è preoccupato dell'esecuzione delle procedure crittografiche.

Analogamente a quanto definito per la derivazione delle chiavi anche per la generazione del seed, che avviene in fase di creazione del DID, è stata dichiarata una funzione che si occupa di invocare la procedura che segue lo standard BIP-39, passando in input una passphrase da associare al seed. La procedura internamente esegue l'algoritmo, memorizza in una specifica area di memoria il dato calcolato e ritorna in output, se la procedura è terminata con successo, la sequenza mnemonica. La definizione dell'area di memoria

tiene sempre conto del vault path definito come per la funzione di derivazione chiavi e del record path dato da concatenazione di un'altra variabile d'ambiente da dichiarare, ovvero `DID_POLITO_STRONGHOLD_BASE_RECORD_PATH`, l'indicazione che si fa riferimento al seed (ovvero la stringa _seed) e un identificativo specifico per quell'area di memoria. Quindi a partire da una struttura `SeedGeneratorForDid` dichiarata come segue

```
pub struct SeedGeneratorForDid {
    pub passphrase: String,
    pub seed_location_id: String
}
```

si può invocare la funzione `generate_seed()` che esegue quanto descritto in precedenza.

4.3 Realizzazione del DID Method

Registrare un DID in sostanza vuol dire creare un identificativo univoco nel distributed ledger con il relativo DID document a cui saranno associate una o più chiavi pubbliche. Nel momento in cui lo smart contract viene distribuito, viene generato un contract address che ne rappresenta appunto l'identificativo univoco appena menzionato. Lo smart contract deve essere in grado di memorizzare una o più chiavi pubbliche che il DID ha generato, deve poter fornire la lettura della singola chiave o anche restituirne una lista, deve poter eventualmente cancellare logicamente una chiave e infine anche cancellare logicamente il DID. Parliamo di cancellazione logica perché, data la loro immutabilità, un contratto una volta distribuito non può più essere cancellato fisicamente dal ledger.

Il contratto implementato contiene al suo interno come stato un'oggetto `mapping` che in Solidity rappresenta una mappa chiave-valore che fa da contenitore delle chiavi pubbliche di riferimento al DID. Ogni singola chiave pubblica, rappresentata nel contratto nella sua codifica esadecimale, è identificata dalla concatenazione dei parametri purpose, registry, method type, verification method e indice della chiave, esattamente come lo è in Stronghold, anzi è direttamente l'invocazione della funzione di Stronghold che restituisce questo valore insieme alla chiave pubblica. L'indice della chiave viene inserito in formato binario dal DID controller in fase di richiesta di aggiunta di una chiave.

```
mapping (bytes => string) public _keys; //Map with key generated from public
    key to store
```

Il contratto contiene anche la dichiarazione di tre eventi che vengono emessi nel momento in cui viene aggiunta una nuova chiave pubblica, viene rimossa una chiave esistente e viene revocato il DID. Gli eventi sono emessi in caso di successo delle operazioni contenute nelle funzioni; ci indicano che lo stato del contratto è mutato e che sono stati aggiornati i dati in esso contenuto. In caso di errore è invocata la funzione `revert()` che si occupa di annullare l'intera transazione e tutte le modifiche ad essa associate ripristinandone lo stato originale.

```
    // Events declaration
    event KeyAdded(address _from, bytes _id, string key);
    event KeyRemoved(address _from, bytes _id, string key);
    event DIDRevoked(address _from, uint256 blockNumber);
```

Le funzioni esposte dal contratto e che ne mutano lo stato permettono di fare le seguenti operazioni:

- **Aggiunta di una chiave**: viene richiesto in input l'identificativo della chiave e la chiave pubblica codificata in formato esadecimale. La funzione dopo i controlli sulla validità degli input e sulla validità dell'identità, aggiunge la chiave nella mappa e aggiorna lo stato del contratto emettendo l'evento di `KeyAdded` che riporta sia l'id della chiave aggiunta che la chiave stessa.
- **Rimozione di una chiave**: si parla sempre di rimozione logica, quindi abbiamo definito una sequenza convenzionale che ci indica una chiave cancellata, ovvero una chiave con tutti i bit posti a 0. La funzione riceve in input l'identificativo della chiave da rimuovere, effettua i dovuti controlli e se memorizzata all'interno dello smart-contract allora viene sostituita, come indicato, con tutti i bit posti a 0. Viene emesso l'evento di `KeyRemoved` che riporta l'id della chiave rimossa e la vecchia chiave rimossa.
- **Revoca dell'identità**: un DID revocato equivale alla cancellazione logica dell'intera identità. La cancellazione è gestita tramite un flag che se impostato impedisce che vengano compiute tutte le azioni sullo smart-contract sia in lettura che scrittura. Quindi sostanzialmente la funzione di revoca dell'identità andrà ad impostare il valore del flag a 1. Nel completare l'operazione viene emesso l'evento di `DidRevoked`.

Le funzioni di sola lettura invece permettono di avere i seguenti output:

- **Fornire la lista di tutte le chiavi memorizzate all'interno del contratto con il loro identificativo**: a tal proposito è stata dichiarata una struttura dati atta a contenere queste informazioni

```
library DidRegistryShared {
    // Struct for getting Public Keys
    struct PublicKey {
        bytes id;
        string key;
    }
}
```

- **Ritornare la validità dell'identità**: quindi la funzione si preoccupa di fornire sotto forma di valore booleano (ovvero `true` o `false`) se l'identità è stata revocata o meno.

```
// The CRUD's CREATE function is the deploy of the smart-contract
contract DidRegistry {

    // CRUD's UPDATE function
    function addKey(bytes memory _id, string memory _publicKey) public { //
        ... }

    // CRUD's UPDATE function
    function removeKey(bytes memory _id) public { // ... }

    // CRUD's READ function
    function getKeys() public view returns (DidRegistryShared.PublicKey[]
        memory) {
        // ...
    }

    // CRUD'S DELETE function
    function deleteRegistry() public {
       // ...
    }

    // Function to get if contract is deleted
    function isContractDeleted() public view returns (bool){
        // ...
    }

}
```

Tra le operazioni descritte manca quella della creazione dell'identità. La creazione non è altro che il deployment del contratto nella DLT. Questa è demandata al **FactoryContract**. Il **FactoryPattern** è un modello ampiamente

utilizzato nello sviluppo software. Nel contesto degli smart-contract si tratta di creare un contratto separato responsabile dell'implementazione degli altri contratti. Può essere visto come un contratto che si occupa di generare le istanze di un altro smart-contract. Questo definisce regole e parametri di creazione delle istanze e su quale istanza andare ad invocare poi le funzioni. Grazie a questo pattern abbiamo vantaggi in termini di efficienza intesa come risparmio di Gas per il deployment e per le chiamate a funzioni, ottimizzando e centralizzando appunto il tutto, scalabilità perché se abbiamo bisogno di creare tante istanze diventa difficile gestirle tutte e infine riusabilità, ovvero recepita come riutilizzo del codice del contratto riducendone la ridondanza (22).

Il factory contract definisce al suo interno una funzione che esegue l'operazione di CREATE tra le CRUD dichiarate, che si occupa di istanziare il registro e ritornare l'identificativo del DID ovvero il contract-address appena generato.
La funzione di creazione emette un evento dichiarato come segue

```
event RegistryCreated(address _address);
```

che contiene al suo interno il contract address appena istanziato all'interno del ledger.
Come stato invece mantiene una mappa la cui chiave di riferimento è l'indirizzo del contratto e valore l'istanza del contratto stesso. Quando c'è necessità di invocare una determinata procedura, le chiamate del factory contract invocano la funzione su quell'istanza ricavata dalla mappa.

```
contract DidRegistryFactory {
    mapping (address => DidRegistry) private _toInteractWith;
    event RegistryCreated(address _address);

    function createRegistry() external {
        DidRegistry _registry = new DidRegistry();
        _toInteractWith[address(_registry)] = _registry;
        emit RegistryCreated(address(_registry));
    }

    function addKey(address addr, bytes memory _id, string memory _publicKey)
         public {
        _toInteractWith[addr].addKey(_id, _publicKey);
    }

    function removeKey(address addr, bytes memory _id) public {
        _toInteractWith[addr].removeKey(_id);
```

```
    }

    function getKeys(address addr) public view returns (DidRegistryShared.
        PublicKey[] memory) {
        return _toInteractWith[addr].getKeys();
    }

    function deleteRegistry(address addr) public {
        _toInteractWith[addr].deleteRegistry();
    }

    function isContractDeleted(address addr) public view returns (bool){
        return _toInteractWith[addr].isContractDeleted();
    }
}
```

4.4 Realizzazione delle API

Facendo un breve riepilogo, il DID method è quel componente che ci permette di implementare le funzioni CRUD su un determinato DID. Nel caso di DID basato su una blockchain o su un distributed ledger le operazioni sono transazioni che vengono fatte su un determinato smart-contract distribuito all'interno del ledger, che rappresenta il Verifiable Data Registry. Quindi avendo visto come abbiamo definito lo smart-contract che fa da DID method e il factory contract che ne permette la gestione centralizzata, dobbiamo creare l'applicazione che interagirà con il nostro DID method, implementando le API che effettuano le operazioni di creazione dell'identità e del DID document, lettura del DID document, aggiungere o rimuovere chiavi e revocare l'identità.

L'applicazione che fa da DID method è scritta in linguaggio Rust. Mediante l'utilizzo degli strumenti messi a disposizione dalla libreria Ethers.rs, possiamo interagire con un distributed ledger e con smart-contract che sono eseguiti su una Ethereum Virtual Machine. Inoltre l'applicazione integra anche come libreria l'implementazione personalizzata di Stronghold che è stata descritta in precedenza che servirà per gestire le chiavi derivate e il seed dal quale generarle.

Il progetto è composto dai seguenti moduli:

1. Configuration

2. Contracts
3. DID
4. Models

Il modulo **Configurations** contiene tutte le funzioni necessarie a gestire le configurazioni del DID method. In particolare essendo compatibile con tutti i ledger che hanno una Ethereum Virtual Machine si è pensato di creare un file di configurazione serializzato in formato JSON (all'interno del progetto è chiamato `config.json`) che contiene ad esempio le configurazioni di rete per connettersi al wallet, la chiave privata usata per firmare le transazioni, piuttosto che il chain ID usato per la protezione da attacchi informatici di tipo Replay. Inoltre nel file di configurazione sono riportati anche l'indirizzo del factory contract con cui interagire e l'indirizzo del DID quando viene creato il contratto che fa da registro. Il modulo contiene tutte le dichiarazioni delle funzioni che sono usate per la lettura e scrittura degli attributi in questo file e inoltre anche l'inizializzazione delle variabili d'ambiente necessarie per il funzionamento di Stronghold.

Il modulo **Contracts** contiene tutto ciò che è necessario per interagire con gli smart-contract. A partire dal codice sorgente dello smart-contract scritto in Solidity, si ottengono al momento della compilazione il relativo ABI e il bytecode per il deployment. Per interagire però con lo smart-contract dobbiamo trovare un modo per codificare e decodificare i dati da e per lo smart-contract e per farlo abbiamo usato Abigen. **Abigen**, integrato nella libreria ethers.rs, permette di generare codice Rust a partire appunto dal file ABI, a cui si può aggiungere il bytecode qualora si volesse anche la funzione per distribuire il contratto sulla blockchain. Il codice Rust permette di interagire con lo smart-contract dato che esso è legato all'interfaccia del contratto, consentendo di chiamare i metodi per leggere e scrivere lo stato del contratto e per sottoscriversi agli eventi. Per usare Abigen abbiamo creato una funzione che prende in input il file ABI oppure un file JSON che contiene sia l'ABI che il bytecode che sono referenziati nel progetto e invoca Abigen generando il **ContractBinding** e poi scrivendolo sul file.

```
pub fn generate_factory_source() -> Result<()> {
    let abi_source = "./src/contracts/files/did-registry-factory
   .json"; // JSON contains ABI and bytecode
    let out_file = Path::new("./src/contracts/").join("
   did_registry_factory.rs");
```

```
    if out_file.exists() {
        std::fs::remove_file(&out_file)?;
    }
    Abigen::new("DidRegistryFactory", abi_source)?.generate()?.
   write_to_file(out_file)?;
    Ok(())
}

pub fn generate_contract_source() -> Result<()> {
    let abi_source = "./src/contracts/files/did-registry.abi";
    let out_file = Path::new("./src/contracts/").join("
   did_registry.rs");
    if out_file.exists() {
        std::fs::remove_file(&out_file)?;
    }
    Abigen::new("DidRegistry", abi_source)?.generate()?.
   write_to_file(out_file)?;
    Ok(())
}
```

Si può notare infatti sia la generazione del binding per il contract-factory indicato con **DidRegistryFactory**, dove il file di input è in formato JSON dato che contiene sia l'ABI che il bytecode rimappato nel file con la chiave `"bin"`, sia la generazione del binding per lo smart-contract **DidRegistry**. Rispettando il factory pattern tutte le chiamate al contratto verranno fatte attraverso la factory, quindi sarà il DidRegistryFactory sul quale invocheremo le funzioni per fare le operazioni sul contratto. Il binding dello smart-contract è comunque generato poiché essendo il did-registry ad emettere gli eventi e non il did-registry-factory abbiamo bisogno dello stesso per sottoscriverci alla ricezione degli eventi e catturarli quando arrivano. I due bindings saranno poi utilizzati all'interno del controller piuttosto che nella funzione per fare il deployment nella blockchain del did-registry-factory.

Il modulo **DID** contiene effettivamente tutta la logica di quanto implementato. Al suo interno troviamo l'implementazione delle API per comunicare con il DID method per effettuare le operazioni CRUD sull'identità. Come già accennato, si farà utilizzo dei binding generati per gli smart-contract did-registry e did-registry-factory e non solo, ma ci saranno anche le chiamate alla libreria Stronghold per la generazione del seed e per la gestione delle chiavi d'identità. L'interfaccia definita per il controller è la seguente:

```
#[async_trait]
pub trait Controller {

    // CREATE function for DID
    async fn create_identity(&self, force_creation: bool) ->
   Result<(), Box<dyn std::error::Error>>;


    // UPDATE function for DID
    async fn add_key(&self, registry: u32, method_type: u32,
   verification_method: u32, index: u32) -> Result<(), Box<dyn
   std::error::Error>>;


    async fn remove_key(&self, registry: u32, method_type: u32,
   verification_method: u32, index: u32) -> Result<(), Box<dyn
   std::error::Error>>;


    // READ function for DID; returns a JSON serialized DID
   document
    async fn get_did_document(&self, method_type: String) ->
   Result<String, Box<dyn std::error::Error>>;


    // DELETE function for DID;
    async fn revoke_identity(&self) -> Result<(), Box<dyn std::
   error::Error>>;
}
```

Prima di descrivere ciò che ogni funzione ci permette di fare sul DID è bene fare un riferimento anche all'operazione di deployment del contratto. Il deployment del factory contract sul ledger è una transazione che ci permette di pubblicare il bytecode all'interno della EVM e renderlo invocabile tramite l'indirizzo che viene restituito se la chiamata alla funzione, e di conseguenza la transazione, è andata a buon fine. A tal proposito è stata creata una funzione Rust apposita che si occupa di effettuare la transazione sopra descritta.

```
pub async fn deploy_contract_factory() -> Result<(), Box<dyn std::
   error::Error>>{
```

```
// check if factory is already deployed...otherwise deploy it
if !Configuration::is_factory_contract_set() {
    // get input for passphrase used for seed generation
    let mut passphrase = String::new();
    println!("Enter your passphrase to generate a seed: ");
    std::io::stdin().read_line(&mut passphrase).unwrap();

    // connect to the network
    let url = Configuration::read_from_config_file().raw_url;
    let ws_client = Provider::<Ws>::connect(url).await?;

    // wallet with chain ID
    let wallet: LocalWallet = Configuration::read_from_config_file
().wallet_private_key.parse().unwrap();
    let chain_id: u64 = u64::try_from(Configuration::
read_from_config_file().chain_id).unwrap();
    let wallet = wallet.with_chain_id(chain_id);

    // signer and client
    let signer = SignerMiddleware::new(ws_client, wallet);
    let client = std::sync::Arc::new(signer);

    // deploy contract
    let contract = DidRegistryFactory::deploy(client, ()).unwrap()
.send().await?;

    // get the contract's address
    let factory_address = format!("{}{}", "0x", hex::encode(
contract.address().0));
    Configuration::write_factory_address_on_config_file(
factory_address.clone());

    // generate seed for key derive and print mnemonic
    let seed_generator = iota_stronghold::procedures::
SeedGeneratorForDid {
        passphrase: passphrase.clone(),
        seed_location_id: factory_address.clone(),
    };

    let mnemonic = seed_generator.generate_seed().unwrap();
    println!("Your mnemonic: {}", mnemonic);
}
```

```
    Ok(())
}
```

Si apre una connessione verso la rete del ledger usando un **Provider**, che rappresenta un'astrazione alla connessione vera e propria fornendo un'interfaccia coerente con le funzioni del nodo. Diversi sono i provider di rete che possono essere utilizzati e sono HTTPS, JSON-RPC, WebSocket ed altri ancora dipendenti dalle funzionalità di rete del nodo. La scelta è ricaduta sui **WebSocket**. Questo perché essendo che il contratto emette degli eventi, i WebSocket abilitano al loro interno il pattern **Publish/Subscribe** e quindi permettono di catturare questi eventi emessi e che poi possono essere gestiti come necessario. Si può notare infatti che qualunque connessione verso il ledger sia per eseguire il deployment sia per eseguire transazioni che mutano lo stato del contratto, tipo creazione del DID, aggiunta e rimozione delle chiavi, revoca dell'identità, verranno eseguite tramite WebSockets. Il provider riceve come parametro l'URL da utilizzare per aprire la connessione. Inizializzato il provider abbiamo bisogno dell'istanza del wallet. A partire appunto dalla chiave privata del wallet, possiamo ottenere l'istanza del **LocalWallet** che fa riferimento ad essa però in questo caso memorizzata localmente (all'interno del file di configurazione). Al wallet ora dobbiamo fornire il Chain ID del ledger di riferimento; infatti abbiamo a disposizione una funzione `with_chain_id<T:Into<u64»(mutself,chain_id:T)` che permette di impostarlo e infine ritornare nuovamente l'istanza del wallet. Tutte le transazioni devono essere firmate con la chiave privata prima di essere trasmesse nella rete; per questo motivo utilizziamo un middleware specifico denominato **SignerMiddleware**. Un middleware è un modo messo a disposizione dalla libreria ethers.rs per definire il comportamento di una determinata funzionalità, iniettando della logica ad hoc, che esegue quella personalizzazione durante il processo di comunicazione con il ledger. Può essere definito un middleware custom, sviluppato ad-hoc o utilizzare quelli già presenti così come si può creare una catena composta da più middleware ed agganciarli al provider. Il SignerMiddleware si preoccupa appunto di firmare la transazione prima di pubblicarla in rete. La catena costituita dal Signer e dal Client di rete verrà impacchettata in un **Atomically Reference Counted (Arc)** per la gestione della concorrenza sul sistema e a questo punto si può eseguire la transazione. Il binding DidRegistryFactory ha una funzione deploy che riceve il client correttamente configurato con tutti i suoi middleware ed eventuali parametri che saranno passati al costruttore dello smart-contract, se definito e se contenente attributi, costruendo un oggetto rappresentante la transazione. Quindi chiamando la funzione `send()` verrà inviata sulla rete la transazione che rimarrà in uno stato di attesa (pending) fino a quando non sarà confermata e validata. Quando sarà confermata il valore di ritorno della funzione conterrà il contratto vero e proprio e anche

il suo indirizzo, che sarà memorizzato nel file di configurazione e utilizzato successivamente per ricostruire l'istanza della factory e chiamare le funzioni CRUD sul contratto. Infine verrà invocata la procedura Stronghold per la generazione del seed da utilizzare nella derivazione delle chiavi di identità e che restituirà la sequenza mnemonica corrispondente. Viene richiesto all'utente di inserire una passphrase che verrà utilizzata sia per protezione del seed e sia per l'accesso a Stronghold quando verranno richieste le sue funzionalità, dato che per creare e successivamente accedere allo Snapshot è necessario proteggerlo con password, onde evitare di leggerne il contenuto e rivelare come ricostruire l'area sicura di memorizzazione delle chiavi. Un esempio di sequenza mnemonica generata è il seguente:

```
Your mnemonic:  ginger skull right duty oval month swift wide clap
junk awful bag palm away zoo pumpkin jeans sting lizard soon acid
require frost spoon
```

La funzione `create_identity()` è la funzione atta a generare il DID e il DID document, creando anche il contratto did-registry nel quale andremo a memorizzare le chiavi pubbliche d'identità del DID.

```
async fn create_identity(&self, force_creation: bool) -> Result<(),
   Box<dyn std::error::Error>>{
        // check if contract registry set or force creation
        if !Configuration::is_contract_registry_set() ||
   force_creation {
            // get input for passphrase used for seed generation
            let mut passphrase = String::new();
            println!("Enter your passphrase to create identity: ");
            std::io::stdin().read_line(&mut passphrase).unwrap();

            // connect to the network
            let url = Configuration::read_from_config_file().raw_url;
            let ws_client = Provider::<Ws>::connect(url).await?;

            // wallet with chain ID
            let wallet: LocalWallet = Configuration::
   read_from_config_file().wallet_private_key.parse().unwrap();
            let chain_id: u64 = u64::try_from(Configuration::
   read_from_config_file().chain_id).unwrap();
            let wallet = wallet.with_chain_id(chain_id);

            // signer and client
            let signer = SignerMiddleware::new(ws_client, wallet);
```

```
            let client = std::sync::Arc::new(signer);

            // retrieve factory contract
            let factory_address: Address = Configuration::
    read_from_config_file().factory_address.parse().unwrap();
            let factory_contract = DidRegistryFactory::new(
    factory_address, client.clone());
            // contract call to create_registry
            let _ = factory_contract.create_registry().send().await?;

            // waiting for events
            let events = Contract::event_of_type::<
    RegistryCreatedFilter>(client.clone());
            let mut stream = events.subscribe().await?.take(1);

            while let Some(Ok(log)) = stream.next().await {
                // RegistryCreated event captured, retrieve contract
    address
                let registry_created_event = log as
    RegistryCreatedFilter;
                // store the contract's address
                let contract_address = format!("{}{}", "0x", hex::
    encode(registry_created_event.address));
                Configuration::write_contract_address_on_config_file(
    contract_address.clone());
            }
        }

        Ok(())
    }
```

La parte transazionale è identica a quanto già analizzato in precedenza. Dopo aver inizializzato il client, viene ricreata l'istanza della factory a partire dal suo indirizzo, salvato nel file di configurazione, infine richiamata la `create_registry()` che invia una transazione per effettuare la chiamata alla relativa funzione dello smart contract. La transazione se eseguita con successo creerà il contratto dove memorizzare le chiavi pubbliche ed emette un evento **RegistryCreated** contenente il suo indirizzo, ovvero il DID. La funzione è in ascolto degli eventi generati filtrando esattamente quello di interesse tramite un **RegistryCreatedFilter** e una volta catturato utilizza l'indirizzo del DID per memorizzarlo nel file di configurazione e usarlo sia per costruire il DID document che per istanziare il binding per il DidRegistry, così da poter ascoltare gli eventi di generazione chiavi e revoca identità.

La funzione `add_key()` permette di aggiungere una nuova chiave d'identità al DID sulla base del suo utilizzo all'interno del sistema differenziandosi quindi per *verification method*. L'aggiunta segue quanto è stato proposto per la generazione delle chiavi. La creazione della nuova chiave d'identità avviene tramite Stronghold, che con la sua procedura la produce, la memorizza nell'area sicura e ci fornisce la chiave pubblica, memorizzata nel registro e visualizzabile nel *DID document* in una qualche codifica ben definita. Le codifiche supportate dal *DID document* sono esadecimale, base58 ed è prevista anche la codifica Json Web Key (JWK).

```
async fn add_key(&self, registry: u32, method_type: u32,
  verification_method: u32, index: u32) -> Result<(), Box<dyn std::
  error::Error>> {
       // connect to the network
       let url = Configuration::read_from_config_file().raw_url;
       let ws_client = Provider::<Ws>::connect(url).await?;

       // wallet with chain ID
       let wallet: LocalWallet = Configuration::read_from_config_file
  ().wallet_private_key.parse().unwrap();
       let chain_id: u64 = u64::try_from(Configuration::
  read_from_config_file().chain_id).unwrap();
       let wallet = wallet.with_chain_id(chain_id);

       // signer and client
       let signer = SignerMiddleware::new(ws_client, wallet);
       let client = std::sync::Arc::new(signer);

       // retrieve factory contract
       let factory_address: Address = Configuration::
  read_from_config_file().factory_address.parse().unwrap();
       let factory_contract = DidRegistryFactory::new(factory_address
  , client.clone());

       // get current contract address in configuration file
       let contract_address: Address = Configuration::
  read_from_config_file().contract_address.parse().unwrap();

       // check if identity is revoked
       let is_revoked = factory_contract.is_contract_deleted(
  contract_address).call().await?;
       if is_revoked {
```

```
            return Err(Box::new(DidRevokedError { contract_address:
Configuration::read_from_config_file().contract_address }));
        }

        // get input for passphrase
        let mut passphrase = String::new();
        println!("Enter your passphrase to add a new key: ");
        std::io::stdin().read_line(&mut passphrase).unwrap();

        // execute transaction to add a new key
        let did_key_generator = iota_stronghold::procedures::DidKey {
            passphrase: passphrase.clone(),
            seed_location_id: Configuration::read_from_config_file().
factory_address.clone(),
            contract_address: Configuration::read_from_config_file().
contract_address.clone(),
            registry: registry,
            method_type: method_type,
            verification_method: verification_method,
            index: index,
        };

        unsafe {
            let result = did_key_generator.add_did_key().unwrap();
            let record_path = result.0;
            let public_key = result.1;

            let encoded_public_key = hex::encode(public_key.clone());
            // contract call to add_key
            let id = ethers::core::types::Bytes::from(record_path);
            let _ = factory_contract.add_key(contract_address, id,
encoded_public_key).send().await?;

            // waiting for events
            let events = Contract::event_of_type::<KeyAddedFilter>(
client.clone());
            let mut stream = events.subscribe().await?.take(1);

            while let Some(Ok(_log)) = stream.next().await {
                println!("Key added with success!");
            }
        }
```

```
        Ok(())
    }
```

La parte transazionale anche qui è identica a quanto già visto. Si creano le istanze del DidRegistryFactory, che invocherà la funzione di aggiunta chiave pubblica sul factory contract, e l'istanza del DidRegistry per mettersi in ascolto sull'evento di **KeyAdded**. Viene fatta una verifica se l'identità è revocata prima di chiamare la funzione vera e propria; se revocata allora è previsto che venga restituito un errore ben specifico definito come **DidRevokedError** e contenente l'indicazione del DID revocato, altrimenti si prosegue con gli step successivi. Si inserisce la passphrase per Stronghold e si invoca la libreria, costruendo la `struct DidKey` e invocando su di essa la funzione `add_did_key()`. L'indicazione di come il record chiave d'identità è generato verrà usato come chiave della mappa contenente tutte le chiavi d'identità e la chiave pubblica in formato esadecimale verrà memorizzata nella mappa del contratto come corrispondente valore del record. Quindi si effettua la chiamata alla funzione `add_key()` sulla factory e in caso di successo verrà emesso l'evento KeyAdded sul quale siamo in ascolto e filtrato tramite **KeyAddedFilter**. Se ricevuto è indicatore che la transazione è stata eseguita con successo.

Duale della funzione di aggiunta chiave è la rimozione di una chiave d'identità. **Bisogna prestare attenzione alla diversa definizione di concetto di cancellazione della chiave d'identità nello smart-contract e nella libreria Stronghold**. Nello smart-contract si tratta di una rimozione logica, ovvero viene impostata una chiave fittizia contenente tutti i bit a 0 e che verrà ignorata quando letto il DID document, perché lo smart-contract contiene dati immutabili che non possono essere cancellati una volta confermati nella rete. In Stronghold, invece, la rimozione della chiave comporta una pulizia dell'area di memoria identificata dal record; quindi ne verrà cancellato tutto il contenuto. Anche in questo caso, a partire dai due indirizzi del file di configurazione, si creano le istanze del DidRegistryFactory, che invocherà la funzione di rimozione logica della chiave pubblica sul factory contract, e l'istanza del DidRegistry per mettersi in ascolto sull'evento di **KeyRemoved**. Se si ha bisogno di conoscere la vecchia chiave pubblica memorizzata nello smart-contract essa verrà restituita come attributo nell'evento emesso dalla funzione. Nuovamente, si verifica se l'identità è revocata prima di procedere con le operazioni, si inserisce la passphrase per Stronghold e dalla `struct DidKey` opportunamente popolata si invocherà la funzione `remove_did_key()` che ritorna il record della chiave cancellata in Stronghold e che verrà usato per recuperare la chiave dalla mappa del contratto e rimapparla con la chiave fittizia. Quindi verrà invocata la funzione `remove_key()` sulla factory e in caso di successo verrà emesso l'evento KeyRemoved filtrando per

KeyRemovedFilter tra tutti gli eventi in ascolto. Se ricevuto è indicatore che la transazione è stata eseguita con successo.

```
async fn remove_key(&self, registry: u32, method_type: u32,
    verification_method: u32, index: u32) -> Result<(), Box<dyn std::
    error::Error>> {
        // connect to the network
        let url = Configuration::read_from_config_file().raw_url;
        let ws_client = Provider::<Ws>::connect(url).await?;

        // wallet with chain ID
        let wallet: LocalWallet = Configuration::read_from_config_file
    ().wallet_private_key.parse().unwrap();
        let chain_id: u64 = u64::try_from(Configuration::
    read_from_config_file().chain_id).unwrap();
        let wallet = wallet.with_chain_id(chain_id);

        // signer and client
        let signer = SignerMiddleware::new(ws_client, wallet);
        let client = std::sync::Arc::new(signer);

        // retrieve factory contract
        let factory_address: Address = Configuration::
    read_from_config_file().factory_address.parse().unwrap();
        let factory_contract = DidRegistryFactory::new(factory_address
    , client.clone());

        // get current contract address in configuration file
        let contract_address: Address = Configuration::
    read_from_config_file().contract_address.parse().unwrap();

        // check if identity is revoked
        let is_revoked = factory_contract.is_contract_deleted(
    contract_address).call().await?;
        if is_revoked {
            return Err(Box::new(DidRevokedError { contract_address:
    Configuration::read_from_config_file().contract_address }));
        }

        // get input for passphrase
        let mut passphrase = String::new();
        println!("Enter your passphrase to remove a key: ");
        std::io::stdin().read_line(&mut passphrase).unwrap();
```

```
        // execute transaction to remove key
        let did_key_generator = iota_stronghold::procedures::DidKey {
            passphrase: passphrase.clone(),
            seed_location_id: Configuration::read_from_config_file().
    factory_address.clone(),
            contract_address: Configuration::read_from_config_file().
    contract_address.clone(),
            registry: registry,
            method_type: method_type,
            verification_method: verification_method,
            index: index,
        };

        unsafe {
            let result = did_key_generator.remove_did_key().unwrap();
            println!("Remove key internal procedure: {}", result.1);
            let removed_record_path = result.0;

            // contract call to remove key
            let id = ethers::core::types::Bytes::from(
    removed_record_path);
            let _ = factory_contract.remove_key(contract_address, id).
    send().await?;

            // waiting for events
            let events = Contract::event_of_type::<KeyRemovedFilter>(
    client.clone());
            let mut stream = events.subscribe().await?.take(1);

            while let Some(Ok(_log)) = stream.next().await {
                println!("Key removed with success!");
            }
        }

        Ok(())
    }
```

Il DID document viene letto attraverso la funzione esposta dal controller `get_did_document()` che lo restituisce serializzato in formato JSON. Esso contiene tutte le informazioni necessarie che sono memorizzate nello smart-contract.

```
async fn get_did_document(&self, method_type: String) -> Result<String
    , Box<dyn std::error::Error>> {
        // connect to the network
        let url = Configuration::read_from_config_file().raw_url;
        let ws_client = Provider::<Ws>::connect(url).await?;

        // wallet with chain ID
        let wallet: LocalWallet = Configuration::read_from_config_file
    ().wallet_private_key.parse().unwrap();
        let chain_id: u64 = u64::try_from(Configuration::
    read_from_config_file().chain_id).unwrap();
        let wallet = wallet.with_chain_id(chain_id);

        // signer and client
        let signer = SignerMiddleware::new(ws_client, wallet);
        let client = std::sync::Arc::new(signer);

        // retrieve factory contract
        let factory_address: Address = Configuration::
    read_from_config_file().factory_address.parse().unwrap();
        let factory_contract = DidRegistryFactory::new(factory_address
    , client.clone());

        // get current contract address in configuration file
        let contract_address: Address = Configuration::
    read_from_config_file().contract_address.parse().unwrap();

        // check if identity is revoked
        let is_revoked = factory_contract.is_contract_deleted(
    contract_address).call().await?;
        if is_revoked {
            return Err(Box::new(DidRevokedError { contract_address:
    Configuration::read_from_config_file().contract_address }));
        }

        // get all keys into contract and filter deleted keys, i.e.
    public keys that contains all zeros char
        let mut keys: Vec<PublicKey> = factory_contract.get_keys(
    contract_address).call().await?;
        // filter keys removing all zero keys
        keys = keys.into_iter().filter(|el| !el.key.chars().all(|c| c
    == '0')).collect();
```

```
        let mut document = DidDocument::new();
        document.build_did_document(method_type.clone());

        // map keys from contract in struct ContractPublicKey
        let contract_keys: Vec<ContractPublicKey> = keys.into_iter().
map(|el| {
            let mut iter = el.id.0.chunks(4);
            let _ = iter.next().unwrap(); // purpose is ignored
            // registry
            let mut registry: Vec<u8> = iter.next().unwrap().to_vec();
            registry[3] = registry[3] & RECORD_MASK; // unmask
            // method type
            let mut method_type: Vec<u8> = iter.next().unwrap().to_vec
();
            method_type[3] = method_type[3] & RECORD_MASK; // unmask
            // verification method
            let mut verification_method: Vec<u8> = iter.next().unwrap
().to_vec();
            verification_method[3] = verification_method[3] &
RECORD_MASK;  // unmask
            // index
            let mut index: Vec<u8> = iter.next().unwrap().to_vec();
            index[3] = index[3] & RECORD_MASK;  // unmask

            // map into a struct ContractPublicKey that are used to
build a did document
            return ContractPublicKey {
                registry: (u32::from_le_bytes(registry[0..4].try_into
().unwrap())),
                method_type: (u32::from_le_bytes(method_type[0..4].
try_into().unwrap())),
                verification_method: (u32::from_le_bytes(
verification_method[0..4].try_into().unwrap())),
                index: (u32::from_le_bytes(index[0..4].try_into().
unwrap())),
                hex_encoded_key: el.key.clone(),
                bytes_encoded_key: hex::decode(el.key.clone()).unwrap
()
            }
        }).collect();

        // handle public keys
        let _ = document.handle_keys_in_document(contract_keys);
```

```
        // return a DID document as JSON format
        document.document_as_json()
    }
```

Se si analizzano gli step implementativi della funzione si possono notare alcune particolarità. La parte che riguarda la transazione in sé è comunque inizializzata allo stesso modo di tutte le altre già viste, ma nel caso della funzione per recuperare le chiavi pubbliche inerenti il DID, si invoca una vista ovvero una funzione di sola lettura, pertanto nel contratto dichiarata come **view**. Il SignerMiddleware, anche se presente nella sequenza di middleware, in questo caso non interverrà per apporre la firma alla transazione perché appunto il metodo non è transazionale.

```
// get all keys into contract and filter deleted keys, i.e.
   public keys that contains all zeros char
        let mut keys: Vec<PublicKey> = factory_contract.get_keys
   (contract_address).call().await?;
```

Si nota inoltre che invece della funzione `send()` sulla factory si invoca la funzione `call()`. Se invocata su una vista essa ritorna i dati contenuti nello stato dello smart-contract e poi grazie alla libreria vengono opportunamente lavorati e convertiti nel tipo dichiarato corrispondente a **PublicKey**. In caso la `call()` dovesse essere invocata su una funzione transazionale, esegue solamente un test di quella chiamata senza mutarne lo stato nello smart-contract.
Dopo aver recuperato dallo smart-contract tutte le chiavi sotto forma di lista di dati PublicKey, che contengono al loro interno i record delle chiavi e le loro codifiche esadecimali, su questa lista vengono filtrate tutte le chiavi che hanno solamente zeri, quindi quelle cancellate logicamente dal DID, e rimosse dalla stessa lista. Infine si costruisce la struttura Rust corrispondente al DID document.
Per costruire il DID document è stata creata una struttura apposita contente al suo interno il riferimento al documento vero e proprio e poi tre funzioni denominate rispettivamente `build_did_document()`, `handle_keys_in_document()` e `document_as_json()`. La prima permette di inizializzare un documento vuoto, contente solo il DID quindi. La seconda funzione invece prende in input una lista di **ContractPublicKey** e iterando su ogni elemento di questo tipo, sulla base di cosa rappresenta la chiave, popola il DID document con il verification method opportuno.

```
pub struct ContractPublicKey {
```

```
    pub registry: u32,
    pub method_type: u32,
    pub verification_method: u32,
    pub index: u32,
    pub hex_encoded_key: String,
    pub bytes_encoded_key: Vec<u8>,
}
```

La terza funzione infine esegue la serializzazione in formato JSON.

Quindi viene restituito il `DidDocument`, richiamando la funzione `build_did_document` e a questo punto abbiamo l'istanza necessaria per l'elaborazione. A partire dalla lista di chiavi recuperate dallo smart-contract viene fatto un mapping in una lista di ContractPublicKey e successivamente invocata la funzione `handle_keys_in_document` con questa lista. Concludendo, viene serializzato il documento ottenendo un output il JSON.

```
{
  "@context": [
    "https://www.w3.org/ns/did/v1",
    "https://w3id.org/security/suites/secp256k1recovery-2020/v2"
  ],
  "authentication": [
    "did:iota:0xc1fb8a871bddbbf78e8fe68ed76d1cfaf837b873?method=
       authentication#keys-0",
    "did:iota:0xc1fb8a871bddbbf78e8fe68ed76d1cfaf837b873?method=
       authentication#keys-2"
  ],
  "id": "did:iota:c1fb8a871bddbbf78e8fe68ed76d1cfaf837b873",
  "keyAgreement": [
    "did:iota:0xc1fb8a871bddbbf78e8fe68ed76d1cfaf837b873?method=
       key-agreement#keys-0"
  ],
  "verificationMethod": [
    {
      "controller": "did:iota:0
         xc1fb8a871bddbbf78e8fe68ed76d1cfaf837b873",
      "id": "did:iota:0xc1fb8a871bddbbf78e8fe68ed76d1cfaf837b873
         ?method=authentication#keys-0",
      "publicKeyHex": "966ccab3279e1c798d2ebbdc5ac4ee7a448d6b
```

```
      dd0afd956f76bfdbd7b9e108c9",
      "type": "EcdsaSecp256k1VerificationKey2019"
    },
    {
      "controller": "did:iota:0xc1fb8a871bddbbf78e8fe68ed76d1
         cfaf837b873",
      "id": "did:iota:0xc1fb8a871bddbbf78e8fe68ed76d1cfaf837b873
         ?method=authentication#keys-2",
      "publicKeyHex": "fcee802f9186e9fd9c94c11648432b97701583089
      7deec2a1409425073b9a864",
      "type": "EcdsaSecp256k1VerificationKey2019"
    },
    {
      "controller": "did:iota:0xc1fb8a871bddbbf78e8fe68ed76d1
         cfaf837b873",
      "id": "did:iota:0xc1fb8a871bddbbf78e8fe68ed76d1cfaf837b873
         ?method=key-agreement#keys-0",
      "publicKeyMultibase": "27SKidy4TvqG7ztpt1aNkYmCHZM1
         DwXuNMFvNwi7qsFQ",
      "type": "EcdsaSecp256k1VerificationKey2019"
    }
  ]
}
```

La revoca del DID corrisponde anche essa alla cancellazione logica dell'identità sul ledger. In questo caso nel contratto verrà apposto un flag che indica il DID come revocato. Oltre ai controlli già implementati nelle API, anche nel contratto nel caso in cui il flag sia impostato a 1 viene chiamata la funzione di reverse della transazione in modo da scatenare un errore. Nell'implementazione della funzione di revoca si richiama a sua volta la funzione del contratto `delete_registry()` che con una transazione imposta il flag con il valore 1. Nel caso in cui la transazione sia confermata viene emesso un evento di **DidRevoked** e tutte le altre chiamate successive alla revoca falliranno con errore.

```
async fn revoke_identity(&self) -> Result<(), Box<dyn std::error::
   Error>> {
        // connect to the network
        let url = Configuration::read_from_config_file().raw_url;
        let ws_client = Provider::<Ws>::connect(url).await?;
```

```
        // wallet with chain ID
        let wallet: LocalWallet = Configuration::read_from_config_file
    ().wallet_private_key.parse().unwrap();
        let chain_id: u64 = u64::try_from(Configuration::
    read_from_config_file().chain_id).unwrap();
        let wallet = wallet.with_chain_id(chain_id);

        // signer and client
        let signer = SignerMiddleware::new(ws_client, wallet);
        let client = std::sync::Arc::new(signer);

        // retrieve factory contract
        let factory_address: Address = Configuration::
    read_from_config_file().factory_address.parse().unwrap();
        let factory_contract = DidRegistryFactory::new(factory_address
    , client.clone());

        // get current contract address in configuration file
        let contract_address: Address = Configuration::
    read_from_config_file().contract_address.parse().unwrap();

        // call delete registry and revoke identity
        let _ = factory_contract.delete_registry(contract_address).
    send().await?;

        // waiting for events
        let events = Contract::event_of_type::<DidrevokedFilter>(
    client.clone());
        let mut stream = events.subscribe().await?.take(1);

        while let Some(Ok(_)) = stream.next().await {
            println!("Identity revoked!");
        }

        Ok(())
    }
```

Il modulo **Model** infine contiene solamente gli oggetti che rappresentano i dati, in questo caso la struttura dati ContractPublicKey.

Capitolo 5

Conclusioni e sviluppi futuri

In conclusione, questo lavoro propone una soluzione per derivare delle chiavi d'identità che siano legate all'utente, che possano essere utilizzate per differenti scopi quali ad esempio autenticazione, key agreement, apertura di una connessione sicura end-to-end per trasmissione di informazioni confidenziali. La definizione della gerarchia permette anche all'utente di recuperare le chiavi e la propria identità. Sono state sviluppate le procedure per derivare le chiavi secondo la gerarchia proposta per poterle poi utilizzare nel DID document. Infine è stato definito il DID method con le relative API che permettono la creazione di una propria identità digitale. Possibili sviluppi futuri sono l'analisi e l'implementazione di procedure per gestire chiavi di differenti algoritmi crittografici (es. chiavi per algoritmi di firma post-quantum) nella nuova gerarchia di chiavi di identità oppure la definizione del processo di recupero dell'identità a partire dalla frase mnemonica.

Bibliografia

[1] Identità digitale L'accesso semplice e veloce ai servizi digitali;. https://innovazione.gov.it/progetti/identita-digitale-spid-cie/.

[2] Falcone C. Self-Sovereign Identity (SSI): i concetti di base e alcune applicazioni internazionali; 2022. https://blog.osservatori.net/it_it/self-sovereign-identity-spiegazione-applicazioni.

[3] The Inevitable Rise of Self-Sovereign Identity;. https://sovrin.org/wp-content/uploads/2018/03/The-Inevitable-Rise-of-Self-Sovereign-Identity.pdf.

[4] Decentralized Identifiers (DIDs) v1.0;. https://www.w3.org/TR/did-core/.

[5] DID URL Syntax;. https://www.w3.org/TR/did-core/#did-url-syntax.

[6] Alex Preukschat DR. Self-Sovereign Identity. Manning Publications; 2021.

[7] JSON-LD 1.1;. https://www.w3.org/TR/json-ld11/.

[8] Verifiable Credentials Data Model v1.1;. https://www.w3.org/TR/vc-data-model/#what-is-a-verifiable-credential/.

[9] Distributed Ledger Technology: beyond block chain;. https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/492972/gs-16-1-distributed-ledger-technology.pdf.

[10] What Is Blockchain? A Beginner's Guide for 2021;. https://bootcamp.cvn.columbia.edu/blog/what-is-blockchain-beginners-guide/.

[11] IOTA;. https://wiki.iota.org/get-started/introduction/iota/introduction/.

[12] IOTA - Distributed ledger technology without blockchain;. https://crypto.marketswiki.com/index.php?title=IOTA.

[13] Directed Acyclic Graph (DAG) based Distributed Ledgers;. https://kbaiiitmk.medium.com/directed-acyclic-graph-dag-based-distributed-ledgers-e2f42c39366.

[14] Ethereum - INTRODUCTION TO SMART CONTRACTS;. https://ethereum.org/en/developers/docs/smart-contracts/.

[15] IOTA - Anatomy of a Smart Contract;. https://wiki.iota.org/learn/smart-contracts/smart-contract-anatomy/.

[16] IOTA - Stronghold;. https://wiki.iota.org/stronghold.rs/welcome/.

[17] IOTA - Stronghold - Cryptographic Procedures;. https://wiki.iota.org/stronghold.rs/explanations/procedures/.

[18] BIP 01 - What is a BIP?;. https://github.com/bitcoin/bips/blob/master/bip-0001.mediawiki.

[19] BIP 32 - Hierarchical Deterministic Wallets;. https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki.

[20] BIP 39 - Mnemonic code for generating deterministic keys;. https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki.

[21] SLIP-0010 - Universal private key derivation from master private key;. https://github.com/satoshilabs/slips/blob/master/slip-0010.md.

[22] Demystifying the Factory Pattern in Solidity: Efficient Contract Deployment with Factory Pattern;. https://medium.com/@solidity101/demystifying-the-factory-pattern-in-solidity-efficient-contract-deployment-with-factory-pattern-e233ea6d1ec0#:~:text=The%20Factory%20Pattern%20is%20a%20powerful%20tool%20for%20Ethereum%20developers,costs%20and%20simplifies%20contract%20management.