Sommario

L'utilizzo del Cloud per la conservazione di informazioni è sempre più diffuso, ma è possibile inserire dati sensibili in maniera sicura? CryptoCloud si pone come intermediario tra l'utente e il cloud provider in modo da criptare tutto ciò che è necessario condividere in cloud nonostante la natura sensibile. Perciò se anche il servizio dovesse subire una violazione, coloro che hanno perpetuato l'attacco non riuscirebbero ad ottenere delle informazioni sul come decriptare i file inseriti grazie all'applicazione. CryptoCloud nasce come software opensource per essere utilizzato da aziende, implementando una gestione dei gruppi e un sistema di notifiche per la comunicazione tra utenti. Tutto ciò è realizzato adoperando esclusivamente il cloud evitando così di dipendere dalle strutture interne all'azienda stessa.

Introduzione

La tesi ha come scopo la realizzazione di CryptoCloud, un software per aziende in grado di conservare e gestire dati sensibili che non devono essere violati, raggiungibili in ogni circostanza e quindi indipendenti dalle infrastrutture dell'azienda stessa.

Per rispondere a questa ultima esigenza si è deciso di utilizzare il cloud. Al giorno d'oggi esistono molte soluzioni che offrono questo servizio: per l'implementazione di CryptoCloud si è deciso di affidarsi a Dropbox anche se il sistema che si è costruito è replicabile con qualsiasi cloud provider.
La scelta di utilizzare il cloud risponde perciò alla necessità di poter sempre accedere alle risorse, ma crea molteplici problemi riguardo alla loro sicurezza: le informazioni inserite sono veramente inaccessibili? Gli amministratori del cloud provider sarebbero ugualmente in grado di spiare il contenuto?

Per far fronte a questo bisogno si è deciso di utilizzare Cryptomator: software opensource che permette di cifrare un folder attraverso una password scelta dall'utente. Senza conoscere la chiave di accesso il contenuto della cartella appare indecifrabile e, in questo modo, è possibile cifrare i dati sensibili da occhi indiscreti.

L'ultimo requisito richiesto a CryptoCloud è l'implementazione di una policy di condivisione dei dati salvati tra gruppi di utenti, replicando in questo modo le tecniche di lavoro utilizzate all'interno dell'azienda. Per creare questa feature esiste la possibilità di scegliere tra due soluzioni: affidarsi nuovamente al cloud provider che, oltre ad offrire un servizio di condivisione dei file, spesso mette a disposizione una policy di gestione dei gruppi, oppure

costruire in proprio una gestione dei gruppi indipendente.
Sicuramente la prima idea proposta è la più semplice, ma concede ai gestori del provider troppi poteri: nessuno infatti garantisce che non siano in grado di modificare a proprio piacere la composizione dei gruppi ottenendo in questo modo accesso ai dati sensibili cifrati. La seconda soluzione invece, anche se più laboriosa, evita questo problema: si è così implementato il gruppo sotto forma di file JSON, e contestualmente si è creato un sistema di message-passing, sempre utilizzando file di questo formato, per la comunicazione tra utenti.

La tesi mostra tutte le fasi che hanno portato alla realizzazione di CryptoCloud, in particolare:

- il primo capitolo rappresenta la stesura delle specifiche ottenute dopo diversi incontri con il Dott. Angelo Neri, responsabile dell'area IT&DP del Cineca.

- il secondo capitolo consiste in una analisi effettuata, prima della progettazione di CryptoCloud, di soluzioni commerciali che rispondono ad un problema analogo a quello proposto. In questa fase si scopre l'esistenza di Cryptomator e si suppone un suo futuro utilizzo.

- il terzo capitolo mostra l'implementazione delle specifiche: i ruoli che possono rivestire gli utenti, la rappresentazione delle entità e la codifica delle operazioni. Inoltre è spiegata la struttura di folder che viene creata all'interno di Dropbox per il corretto funzionamento di CryptoCloud.

- il quarto e ultimo capitolo spiega nel dettaglio le classi che sono state realizzate, le loro motivazioni e implementazioni.

Indice

Capitolo 1

Specifiche

1.1 Progetto

Il progetto consiste nella realizzazione di un repository diffuso nel cloud e accessibile in ogni momento, indipendentemente quindi dall'accessibilità o meno delle strutture dell'azienda: si vuole creare un'infrastruttura in grado di gestire un numero sufficientemente elevato di utenti, i quali possono appartenere a uno o più gruppi.
È inoltre importante che l'amministratore del provider utilizzato non abbia la possibilità di conoscere il contenuto inserito nel cloud, poiché si vuole inserire dati sensibili quali credenziali di accesso.
Ogni gruppo dovrà poter accedere a uno o più folder, dentro al quale saranno inseriti i file che i membri dovranno essere in grado di leggere; allo stesso tempo ogni folder potrà essere consultato da gruppi diversi, e per questo motivo sarà necessario implementare un sistema di sharing non banale.
Il software che si vuole realizzare è perciò suddivisibile in due sezioni: la prima è la gestione dei gruppi di utenti e i permessi di accesso ai folder, la seconda consiste nel criptare e decriptare file e cartelle che dovranno essere inseriti o prelevati dal cloud.
Dovrà essere implementato anche una policy di fiducia per gli utenti e di conseguenza per i gruppi: un utente, per poter affermare di essere chi dice di

essere, deve autenticarsi in un qualche modo: o ricevendo fiducia attraverso le firme di più persone, esattamente come funziona in GPG, o effettuando un login attraverso credenziali solo da lui conosciute.
Dovranno essere anche presenti amministratori con il compito di confermare o meno la creazione di gruppi; sarà inoltre necessario il ruolo di owner di un gruppo con il compito di amministrare i membri al suo interno.
Le tre possibili opzioni per la gestione dei folder sono: gestione da parte degli amministratori, creazione da un qualsiasi utente e approvazione da un amministratore, completa gestione da parte dell'utente creatore del folder stesso.
Si è preferita la terza opzione in maniera da evitare di appesantire troppo il ruolo di amministratore e per riuscire ad avere una concessione dei privilegi al massimo livello di granularità possibile: se si fosse scelta la prima delle tre opzioni elencate, un account amministratore compromesso provocherebbe il collasso del sistema intero in quanto un eventuale attaccante avrebbe accesso ad ogni possibile file sensibile posto sulla rete; con la terza opzione il danno più grande possibile è l'acquisizione delle sole credenziali di un utente possessore di un folder.
Gli utenti che creeranno una cartella contenente le informazioni da condividere con i gruppi avranno anche il compito di definire i permessi (o di sola lettura, o di scrittura con conseguente modifica dei file) che i gruppi avranno all'interno del folder.
Deve essere inoltre realizzato un sistema di notifica per avvertire gli utenti di cambiamenti importanti: quando un utente entra a far parte di un nuovo gruppo o ha ricevuto l'accesso ad un nuovo folder deve ricevere una notifica, così come quando un utente viene rimosso da un gruppo.
Come il progetto, anche le motivazioni alla base sono suddivisibili in due sezioni: la prima inerente alla necessità di una gestione dei gruppi, la seconda relativa alla volontà di porre dati sensibili sul cloud.
La spiegazione del secondo punto è abbastanza banale: non esiste sistema più diffuso, più indipendente da qualsiasi struttura, sempre accessibile, del cloud. È una tecnologia che chiunque riterrebbe utile ma raramente utiliz-

zata per inserire risorse sensibili per problemi intrinseci che può presentare. La necessità dei gruppi deriva dal fatto che all'interno di una azienda i dipendenti non lavorano in maniera isolata, ma come membri di gruppi: è perciò ovvio che alcune informazioni debbano essere condivise con i colleghi del gruppo e solamente con loro.

1.2 Entità

Le entità che dovranno essere prese in considerazione sono quattro: l'utente, il gruppo, la PwdEntry, e il PwdFolder.
L'utente dovrà possedere:

`Id` chiave univoca dell'utente

`NameSurname` nome e cognome per avere una relazione tra utente virtuale e persona reale

`NotifyMethod` metodo scelto dall'utente(E-mail, SMS) per essere avvisato di modifiche importanti

Il gruppo è caratterizzato da tre attributi:

`Name` nome univoco che identifica il gruppo

`Members` lista di utenti che fanno parte del gruppo

`Owner` il capogruppo che si occuperà dell'inserimento e della rimozione dei membri

Il PwdFolder invece è il contenitore cifrato di un numero arbitrario di PwdEntry al quale devono poter accedere uno o più gruppi; esso sarà formato da:

`Name` nome univoco che identifica il PwdFolder

`PwdEntries` insieme di PwdEntry contenute al suo interno

`Owner` proprietario del PwdFolder che deve garantire l'accesso ai corretti gruppi

`Groups` lista di gruppi che hanno accesso di lettura o di scrittura

La PwdEntry consiste nel esempio di file sensibile che deve essere protetto e condiviso nel cloud ed è associabile ad un file JSON con i seguenti campi:

`Name` il nome che identifica la PwdEntry

`PwdFolder` il nome del PwdFolder che la contiene

`System` il sistema in cui devono essere utilizzate le credenziali

`Username` username di accesso per il sistema

`Password` password di accesso per il sistema

`LastModify` l'informazione dell'ultima modifica e dell'autore della stessa

1.3 Operazioni

Una prima idea delle operazioni minime che devono essere implementate per la creazione di un software soddisfacente ed utilizzabile sono le seguenti:

`ListPwdEntries()` mostra tutte le PwdEntry disponibili all'utente

`ListPwdFolders()` mostra tutti i PwdFolder disponibili all'utente

`ListGroups()` mostra tutti i gruppi esistenti

`ListMyGroups()` mostra tutti i gruppi di cui l'utente fa parte

`CreateGroup()` crea un nuovo gruppo e ne inserisce i membri

`DeleteGroup()` elimina un gruppo

`AddMembers()` aggiunge nuovi membri ad un gruppo

`RemoveMembers()` rimuove membri da un gruppo

`CreatePwdFolder()` crea un nuovo PwdFolder e lo condivide con i gruppi

`RemovePwdFolder()` elimina un PwdFolder togliendone la condivisione con tutti i suoi gruppi

`AddGroups()` aggiunge un nuovo gruppo ad un PwdFolder potendone decidere i permessi di accesso.

`RemoveGroups()` rimuove un gruppo da un PwdFolder togliendone l'accesso

`CreatePwdEntry()` crea una nuova PwdEntry in un determinato PwdFolder

`GetCredentials()` restituisce la password e l'username contenuta in una PwdEntry

`ModifyPwdEntry()` modifica la password contenuta in una PwdEntry

`RemovePwdEntry()` elimina una PwdEntry in un determinato PwdFolder

`AuthenticateUser()` autentica un utente della sua identità

Capitolo 2

Analisi di Soluzioni Esistenti

Prima di decidere se fosse necessario o meno creare un software da zero è stata compiuta una analisi delle offerte esistenti per risolvere un problema, se non analogo, almeno simile a quello proposto.
In questo modo è stato possibile farsi un'idea su cosa esiste, cosa manca, cosa deve essere implementato e cosa è possibile utilizzare.

2.1 Cifratura Client-Side

I software del primo gruppo che viene presentato è principalmente client-side, in caso affiancati da un server per la gestione dei login e la comunicazione tra utenti, e si pongono come intermediari tra i vari cloud provider e l'user stesso.
Sono le migliori soluzioni se si desidera porre i dati in maniera altamente distribuita in quanto si possono interfacciare con un numero elevato di cloud provider e permettono la condivisione dei file con altri utenti grazie alle politiche implementate dai provider stessi.

2.1.1 Cryptomator 

Nasce come software opensource per cifrare client-side folder e il loro contenuto.
Non nasce con lo scopo di essere utilizzato per cifrare dati nel cloud, ma poiché molti provider forniscono un client che si integra direttamente con il filesystem, è possibile utilizzare Cryptomator anche per questo scopo.
Il funzionamento è semplice: dato un punto nel filesystem, è possibile creare al suo interno una vault protetta. Questa è protetta da una password, scelta dell'utente, e tutti i file al suo interno vengono criptati.
Cryptomator riconosce una vault da un file speciale, `.cryptomator`, il quale può essere creato o importato e aggiunto all'elenco delle proprie vault.
Non dando direttamente la possibilità di condividere i file, bisogna comunque appoggiarsi alle feature offerte dai cloud provider: se un folder viene condiviso, sarà condivisa anche il file `.cryptomator` al suo interno.
Nasce però il problema della condivisione della chiave di cifratura con il destinatario poiché è stata decisa dall'utente al momento della generazione della vault.
Un altro aspetto interessante di Cryptomator è l'essere opensource: fornisce liberamente sia il software in sé che una libreria facilmente fruibile contenente tutte le funzioni interne di Cryptomator. Tutte le operazioni di cifratura e decifrazione utilizzano il protocollo AES e le chiavi sono protette dall'attacco brute-force utilizzando Scrypt.
Il tutto sotto licenza GPLv3 per i progetti Free and Open Source Software.

2.1.2 BoxCryptor 

Si pone come intermediario tra diversi cloud provider e l'utente, permettendo di criptare file per poi appoggiarsi alle feature dei vari provider per la gestione dei permessi.

Oltre a supportare tutti i provider più diffusi, si interfaccia anche con molti altri meno conosciuti per arrivare ad un totale di 31.
Ogni utente e gruppo possiede una propria coppia di chiavi RSA, formata da privata e pubblica, e un insieme di chiavi simmetriche AES utilizzate per precise azioni: ogni file possiede due chiavi AES, una per il contenuto e una per il nome; ogni password ha la propria chiave AES con cui essere criptata, ogni gruppo ha la propria chiave simmetrica per velocizzare le operazioni.
Le informazioni che vengono immagazzinate dell'utente però sono diverse: oltre a tutti i dati per il login, anche le chiavi sono salvate sui loro database. Ogni informazione salvata è criptata con la chiave di cifratura del server, e solamente la chiave privata RSA e tutte le chiavi simmetriche AES sono criptate nuovamente con la password dell'utente.
La password è criptata con una funzione hash dal client, per poi essere hashata nuovamente dal server, ed è utilizzata solamente per l'autenticazione e per iniziare l'operazione di decifrazione dei file.
Utilizzando BoxCryptor la gestione dei gruppi deve avvenire in due passaggi: la prima fase consiste nell'utilizzare il cloud provider per una prima gestione dei gruppi e per impostare i permessi che essi avranno sul file in questione. È necessario infatti che il file sia condiviso con tutti i gli altri utenti affinchè ci possano accedere.
La seconda fase consiste nell'autorizzare le stesse persone attraverso BoxCryptor ad accedere al file: dovrà perciò essere autorizzato un gruppo separato ma composto dallo stesso personale affinché possano realmente conoscere il contenuto. Questo sicuramente comporta dell'overhead, in quanto ogni azione deve essere effettuata due volte. BoxCryptor però permette anche l'invio di file ad utenti che non utilizzano l'applicazione attraverso Whisply. Whisply è un servizio di trasferimento file nato per soddisfare questa necessità di BoxCryptor: il mittente può anche decidere di inserire un pin o una password e, dopo aver mandato il link al destinatario, egli sarà in grado di scaricare il file in questione in maniera totalmente sicura. Oltretutto, se il file dovesse essere modificato, il destinatario scaricando nuovamente dallo stesso

link, riceverà direttamente la versione aggiornata.
È stato inoltre reso pubblico l'algoritmo utilizzato per criptare i nomi dei file sulla piattaforma Github chiamato Base4k. Non è prevista una apertura totale all'open source da parte di Boxcryptor, ma hanno affermato di avere la volontà di rendere pubbliche anche altre parti del loro codice.
Sono forniti diversi client a seconda del sistema che si vuole utilizzare: esiste per Windows e Mac, Android e Ios, Windows Phone e Blackberry.
Per i sistemi Unix è fornita solo la versione Portable, la quale però manca di tutta la gestione dei permessi e dei gruppi, così come Whisply non è direttamente integrato ma è necessario utilizzare la versione browser.
Sono fornite due versioni di account: per utenti singoli e per aziende. L'utente singolo può disporre di tre varianti, a seconda di quanto è disposto a spendere: la versione free fornisce un solo provider a scelta, due dispositivi e l'integrazione con Whisply; la versione Personal da 3 euro mensili aggiunge anche la crittografia dei nomi dei file, un numero illimitato di dispositivi e di provider cloud; la versione Business da 6 euro mensili che aggiunge anche la possibilità di gestire gruppi.
La versione per aziende è suddivisibile a seconda del numero dei dipendenti: sotto i 50 utenti è offerta la versione Company che permette di avere una chiave master per poter decifrare sempre ogni file, un supporto per Active Directory, una gestione degli utenti e dei gruppi e una autenticazione a due fattori per un costo mensile di 7 euro ad utente. La versione Enterprise invece fornisce anche la possibilità di avere Single Sign-on, account manager dedicato e una assistenza per il setup personale.

2.1.3 Sookasa

Ha le stesse finalità di BoxCryptor: è un intermediario tra il cloud e l'utente, con lo scopo di cifrare i file prima che vengano inseriti nella rete per renderli illeggibili in caso di violazioni.

A differenza di BoxCryptor i provider sono limitati a Google Drive e DropBox. Per condividere un folder o un file si utilizza la policy di sharing del provider stesso.
Per fare in modo che il destinatario riesca a poter utilizzare il folder condiviso è necessario che anche lui possegga un account Sookasa. Infatti questo software cripta e decripta solamente ciò che è contenuto all'interno del folder omonimo, creato al momento della installazione.
Permette però di inviare e di ricevere file con utenti che non possiedono un account Sookasa: basta scegliere l'opzione *Share Securely via Sookasa* e sarà possibile inserire l'indirizzo del destinatario che riceverà una email con un link per effettuare il download.
Durante l'invio del file è possibile impostare una durata di vita del link: una volta scaduto il ricevente non sarà più in grado di scaricare nulla.
Gli utenti che non possiedono un account Sookasa e vogliono inviare un file in maniera sicura devono prima di tutto ottenere il link univoco che identifica il possessore di Sookasa, dopo di che avranno la possibilità di inviare file con dimensioni massime di 100MB.
Per poter visualizzare e modificare i file che sono stati condivisi dal provider, Sookasa richiede un ulteriore passaggio: l'utente che dovrà ricevere il file condiviso deve aver inserito il mittente tra i propri contatti autorizzati, creando dell'overhead.
Esattamente come per BoxCryptor, la gestione dei folder di password può essere gestita in maniera molto semplice: ogni utente che si prende la responsabilità di diventare Owner di un folder dovrà condividere il folder con tutti i membri a cui vuole dare l'accesso, determinando anche i permessi che questi avranno attraverso il cloud provider.
Il client è offerto solamente per Windows, MAC e per le piattaforme Android e iOS: per i sistemi unix non vi è nessun tipo di servizio. Il prezzo richiesto è di 10 euro mensili per utente: costa più di BoxCryptor e offre la compatibilità a meno provider.
Inoltre è necessario acquistare la versione premium anche di DropBox o Goo-

gle Drive per avere la gestione dei gruppi, aumentando il costo complessivo della soluzione.

2.1.4 Odrive

A differenza degli altri software illustrati in precedenza, Odrive ha un'altra impostazione: la feature principale del programma è il linkaggio di tutti gli account dei cloud provider più diffusi in un unico sistema, creando perciò un punto di accesso unico.
Non nasce perciò come servizio per cifrare i file, ma la crittografia è offerta come feature aggiuntiva.
Odrive non necessita di una registrazione, ma è possibile iniziare ad utilizzarlo non appena si è in possesso di un account di uno dei provider che supportano.
Questo avviene generando un disco virtuale in cui vengono inseriti tanti folder quanti sono gli account linkati con al loro interno il contenuto che era presente in quel cloud provider prima della installazione di Okasa.
Una modifica, un'aggiunta o una rimozione di un file dentro questi folder è come se fosse eseguita direttamente nel cloud grazie all'operazione di sync.
Il sync avviene in maniera autonoma, se ci dovessero essere problemi nella sua esecuzione, come ad esempio se dovesse succedere che o l'utente è offline, o il server è offline, oppure troppi file contemporanei che provano ad effettuare il sync, i file interessati saranno posti nella Waiting list: una volta che la condizione che ha causato la sospensione sarà stata risolta, il sync verrà ripreso sempre in maniera autonoma.
Poiché ha un insieme troppo eterogeneo di soluzioni cloud da gestire, Odrive utilizza un proprio metodo per lo sharing di file e non utilizza le varie interfacce offerte dai vari provider.
È possibile condividere i folder con altri utenti nella seguente modalità: innanzitutto viene inviata una email al destinatario che contiene un link per

effettuare lo share; dopo averlo accettato, il ricevente si troverà tra i suoi folder anche quello condiviso. Non vi è però una gestione dei permessi, quando un folder viene condiviso il destinatario ha sempre sia i permessi di lettura che di scrittura.
Non è implementato nessun supporto per la gestione di gruppi di utenti né utilizza i servizi offerti dai vari provider poichè, come detto in precedenza, sono troppo eterogenei tra di loro.
Solamente la versione premium permette però di cifrare i file, ad un costo di circa 7 euro mensili.
La cifratura avviene in modalità client-side, quindi i provider non sono a conoscenza della chiave e utilizza AES 256-bit.

2.2 Provider Zero-Knowledge

Un'altra possibilità consiste nel evitare di utilizzare i tipici servizi di cloud, ma affidarsi a cloud provider che garantiscano la zero-knowledge e che forniscano delle API con cui lavorare.

2.2.1 Mega

Mega si presenta come fornitore di servizio cloud globale e sicuro, con cifratura end-to-end, codice sorgente parzialmente pubblico, backup live criptato e uno spazio di archiviazione gratuito di 35 GB.
Inoltre permette la condivisione con altri account mantenendo la cifratura end-to-end e permette di settare i permessi che avranno i destinatari in maniera molto fine: infatti permette di impostare per un folder il permesso di sola lettura e di impostare i folder interni con permessi sia di lettura che di scrittura.
Non viene diffuso un elenco di API per poter lavorare con MEGA ma viene

fornito l'intero codice sorgente sia del loro Web Client, sia del SDK sia di MegaSync.
Sono inoltre presenti i codici sorgenti delle estensioni per Chrome, Firefox e Thunderbird: il tutto è facilmente trovabile su Github.
Il codice sorgente client-side è stato pubblicato per due motivi: il primo è la diffidenza nel servizio offerto dopo che MegaUpload è stato sequestrato dal dipartimento di giustizia degli Stati Uniti d'America per violazione di copyright e pirateria con la conseguente condanna del suo fondatore.
In questo modo i gestori di Mega cercano di redimersi e mostrare che il servizio da loro offerto è veramente sicuro e senza backdoor: chiunque abbia accesso al codice può controllare tutte le operazioni che vengono svolte al suo interno.
La seconda motivazione riguarda la sicurezza: più persone sperimentano il sistema alla ricerca di falle, migliore sarà alla fine il servizio offerto.
Però è l'unico provider che non utilizza mai il termine zero-knowledge, ma sempre cifratura end-to-end: probabilmente qualche dato dell'utente è salvato.
Quattro sono le critiche che si possono muovere verso Mega.nz: la prima è che non gestisce la possibilità di avere gruppi di utenti che condividano risorse; la seconda è che non fornisce autenticazione a due fattori, la terza è che non permette né l'invio né la ricezione di file da parte di utenti che non utilizzano Mega e la quarta è il fondatore del servizio stesso: Kim Dotcom.
Dati i suoi trascorsi con la giustizia americana sia come Hacker che come fondatore di MegaUpload, la fiducia che si può concedere ad un servizio da lui fondato è limitata.
Un altro problema che era stato riscontrato con Mega, e che poi è stato corretto, è il recupero password.
Poiché Mega non salva la password di accesso, una volta perduta non è più possibile decriptare i file: infatti i dati sono leggibili solo attraverso una catena di decriptazione che inizia con la chiave master di criptazione dell'utente che è immagazzinata in modo criptato attraverso la password. Per questo

motivo viene fornito il servizio di salvataggio della chiave master localmente. Offre diversi piani premium, a seconda della dimensione dello spazio di archiviazione necessario e alla banda di trasferimento: Pro lite per 5 euro mensili, 200 GB di storage e 1 TB di trasferimento; con 10 euro mensili si ottiene il piano Pro1, 1 TB di storage e 2 TB di trasferimento; 20 euro mensili rappresenta Pro2, 4 TB di spazio di archiviazione, 8 TB di banda; infine la versione Pro3 costa 30 euro al mese e permette uno storage di 8 TB e 16 TB di trasferimento.

2.2.2 pCloud

pCloud ha iniziato a far parlare di sé lanciando una sfida: chiunque fosse riuscito a violare la loro cifratura client-side avrebbe vinto 100.000 dollari. Nell'arco di sei mesi ci sono stati 2860 partecipanti, tra cui studenti delle università di Berkeley, Boston e MIT, ma nessuno di questi è riuscito a forzare la cifratura.
Si presenta come disco virtuale, esattamente come DropBox o Google Drive, e per criptare i file è sufficiente posizionarli al suo interno. A differenza di Mega, la versione gratuita di pCloud offre uno spazio di archiviazione molto limitato e non tutte le funzionalità che il provider realmente offre: la creazione e la gestione dei Crypto Folder è ottenibile solo acquistando un account premium così come il recupero di file cestinati.
Come Mega non è contemplata l'esistenza dei gruppi. Nonostante ciò è gestita la possibilità che i file vengano condivisi con altri utenti, ed è possibile decidere come impostare i permessi: se di sola lettura o anche di scrittura.
Non c'è invece la possibilità di invio o ricezione di file da utenti che non possiedono un account pCloud, ma soprattutto i file condivisi non possono essere criptati.
Infatti in pCloud vengono considerate come due tipologie separate il Crypto Folder, o semplicemente il folder protetto dalla loro cifratura zero-knowledge,

e lo Shared Folder, cioè quelle cartelle che sono condivise con altri utenti. Data questa premessa, in pCloud un Crypto Folder non può anche essere uno Shared Folder.
Inoltre all'interno di uno Shared Folder non è permessa una gestione dei permessi a grana fine: una volta che un folder è stato condiviso con un utente, anche le sue cartelle interne avranno automaticamente e irrevocabilmente gli stessi permessi della radice.
Gestisce il versioning dei documenti, per una durata massima di 15 giorni per gli account gratuiti e 30 per quelli a pagamento; non impone dei limiti sulla dimensione dei file che vengono caricati.
Fornisce per gli sviluppatori un API descritta molto dettagliatamente: tutte le strutture che vengono utilizzate nelle chiamate, i metodi utilizzabili e i protocolli di accesso sono elencati e descritti in maniera molto chiara e precisa. Sono inoltre disponibili 5 SDK, tutte inserite su github, per 5 linguaggi diversi: C, Java, Javascript, Php e Swift.
Le critiche che si possono fare sono tre: non vi è gestione di gruppi e non supporta autenticazione a due fattori, oltre alla principale che è il fatto che i file condivisi non siano criptati.
I piani premium che offre sono di due tipi: o Premium, con 500 GB di storage e altrettanti di banda in download, oppure Premium Plus, aumentando sia lo spazio che il traffico a 2 TB.
Entrambi i piani hanno traffico di caricamento remoto illimitato, entrambi hanno 30 giorni di cronologia cestino, e tutti e due sono offerti con due modalità di pagamento: annuale o a vita.
È possibile acquistare il premium per 175 euro una tanum o per 4 euro mensili; il premium plus invece costa 8 euro mensili oppure 350 euro.

2.2.3 Tresorit 

Definito come uno dei più sicuri cloud storage disponibili, Tresorit basa la sua reputazione sullo zero-knowledge: il suo ad pubblicitario infatti inizia affermando

Dropbox has knowledge, Tresorit has zero-knowledge.

La potenza della cifratura di Tresorit è suddivisa in tre sezioni: end-to-end encryption, condivisione delle chiavi, protezione dell'integrità lato client.
Tresorit infatti cripta ogni file e i relativi metadati lato client, non inviando mai la chiave utilizzata per la cifratura e rendendo l'accesso al file disponibile solo all'utente che lo ha generato: viene utilizzata una chiave scelta dall'utente e poi criptata con AES256.
Inoltre le chiavi di cifratura per i gruppi sono cambiate di periodicamente utilizzando un metodo "lazy re-encryption": quando l'insieme dei membri del gruppo è modificato tutti i nuovi file, o file vecchi che sono poi stati modificati, saranno criptati con una nuova chiave alla quale l'utente che è stato espulso dal gruppo non potrà più accedere.
Permette inoltre una gestione dei gruppi, e perciò la creazione di folder criptati condivisi: Tresorit ha brevettato (codice US9563783) un protocollo per la condivisione di chiavi in maniera automatica tra gli utenti del gruppo senza che nessuno che abbia accesso al server o alla rete possa conoscerla.
La protezione dell'integrità lato client è garantita da un codice MAC applicato ad ogni file: in questo modo anche se il sistema di Tresorit fosse stato compromesso, nessuno sarebbe in grado di modificare le informazioni senza che l'utente lo sappia.
Il sistema di autenticazione che forniscono è zero-knowledge: la password dell'utente è salvata utilizzando una funzione hash crittografica con aggiunta di salt: ogni futuro accesso richiederà un challenge-response per confermare l'identità.
Utilizzano il protocollo TLS sul server, per autenticare il browser con il client

ma anche il viceversa: infatti un attaccante potrebbe impersonare un utente e richiedere la connessione con il server per ottenere le risorse contenute. In questo modo ogni client è sicuro dell'autenticità del server così come il server è sicuro del client.
Ogni client Tresorit invece riceve differenti certificati PKI per ogni device, in modo che la connessione dell'applicazione con il server sia molto più sicura.
Molti cloud provider utilizzano la "deduplicazione" per guadagnare spazio: prima di ogni upload il server controlla se l'oggetto che vuole essere inserito sia già presente, in caso affermativo non viene inserito nuovamente ma viene semplicemente creato un link al file.
Questo va contro la sicurezza semantica, poiché due oggetti uguali che sono stati cifrati due volte, dovrebbero avere due ciphertext differenti. Tresorit, preoccupandosi anche della sicurezza semantica, evita la deduplicazione:

> *We don't sacrifice security to save on storage costs.*

Fornisce inoltre una autenticazione forte che può essere impostata a piacere dell'utente: messaggi al telefono, email, o attraverso l'utilizzo di una app fornita da Tresorit.
È presente come client per ogni sistema operativo, sia Windows che Mac che Linux; inoltre è presente pure su Android, iOS, Windows Phone e Blackberry.
Non imposta dei limiti di velocità né di download né di upload, non ha una durata massima l'offerta di file version control e di ripristino di file eliminati, così come l'attività di log delle operazioni sui folder.
Permette l'invio di file anche ad utenti che non utilizzano Tresorit sempre in maniera protetta: sarà possibile impostare una password, un numero massimo di download e una data di scadenza.
Vi è pure la possibilità di condividere file e folder con altri utenti che utilizzano Tresorit: esattamente come altri cloud provider è possibile creare gruppi, condividere risorse con altri utenti che utilizzano il cloud o con altri gruppi e determinare i permessi che avranno sulla cartella condivisa.
Tresorit fornisce aggiuntivamente una piattaforma per autenticare gli utenti

su un proprio server e per criptare le informazioni ottenute chiamato Zero-Kit.
ZeroKit nasce come SDK per android, iOS e per Javascript: esso fornisce le primitive per gestire una cifratura end-to-end ed evitare di conservare le password degli utenti, limitando i danni di un eventuale furto.
Anche ZeroKit, come pCloud ha creato una sfida: chiunque fosse in grado di rompere il sistema da loro creato vincerebbe un premio da cinquanta mila dollari.
La sfida è aperta da 470 giorni ed è stata provata da più di 1050 persone, compresi studenti del MIT, Stanford e Harvard: nessuno però è riuscito finora a vincere il premio.
Parlando di prezzi, ZeroKit è gratuito fino a 100 utenti attivi al mese, dopo di che il costo aumenta a 100 euro mensili per ogni mille utenti.
Tresorit invece ha il prezzo maggiore tra tutti i prodotti presentati: con tutte le feature che presenta è normale che costi di più.
Non offre innanzitutto una versione gratuita, ma solamente versioni di prova; la prima differenza sul piano di acquisto è la volontà di usarlo in gruppo o se è per scopo personale: in quest'ultimo caso ovviamente non sono presenti tutte le feature create per la gestione dei gruppi.
La versione individuale base è di 8 euro al mese e si differenzia dalla versione da 20 euro mensili per varie feature: quella base ha uno storage di 200GB mentre la premium di 2TB; quella base permette l'accesso fino a 5 device e l'altra fino a 10; version recovery per un massimo di 10 versioni mentre quella pro controlla un numero illimitato di versioni; la feature aggiuntiva della versione da 20 euro mensili è inoltre la possibilità di condividere in maniera protetta i link anche ad utenti che non usano Tresorit.
Le versioni per team invece sono tre, a seconda del numero dell'azienda:16 euro mensili per 10 utenti, 20 euro fino a 100 utenti, 24 se si superano i 100. Tutte offrono 1TB di spazio cifrato per utente e la gestione dei gruppi, in aggiunta a tutte le feature comprese nei pacchetti personali elencati prima.
La versione da Business da 20 euro offre inoltre la possibilità di recuperare

le password da parte degli Admin, di integrarsi direttamente con l'active directory e di avere supporto telefonico.
La versione Enterprise aggiunge anche la feature di audit trail, di avere le Admin API e di una formazione personalizzata per il personale.

2.3 Analisi

La tecnologia che più si avvicina alla richiesta è Tresorit ma le complicazioni nel caso di adozione sarebbero molteplici. Innanzitutto ha un costo molto elevato, eccessivo se dovesse essere impiegato in una azienda di grosse dimensioni. In secondo luogo, si ritorna al problema della fiducia: non rilasciando pubblicamente l'implementazione di tutte le features che offrono, bisognerebbe fidarsi delle loro promesse. Notoriamente la security by obscurity non offre alcuna garanzia effettiva e non sarebbe possibile rilevare la presenza di una eventuale backdoor nel codice che potrebbe compromettere tutti i file fin ora inseriti. Per questo motivo si è deciso utilizzare solamente tecnologie opensource, decidendo così di adottare Cryptomator che è infatti l'unica opzione opensource tra quelle elencate e combacia perfettamente con l'idea di PwdFolder richiesta: una vault per Cryptomator è equivalente ad un PwdFolder per CryptoCloud.

Capitolo 3

Manuale dell'Applicazione: dall'Installazione all'Uso

3.1 Prerequisiti

Per poter utilizzare CryptoCloud è necessario avere installato sulla propria macchina una versione superiore alla 8 di Java, poiché è stato deciso di avvalersi di questo linguaggio; inoltre è necessaria la Java Cryptography Extension Unlimited Strength Policy Files, utilizzata per la creazione di chiavi di 256 bit. Un ulteriore software che deve preesistere alla creazione di CryptoCloud consiste nel sync-client di Dropbox, che servirà per decriptare e criptare i PwdFolder in maniera fluida, evitando di effettuare il download e poi l'upload di intere cartelle.

3.2 Tecnologie Utilizzate

Per la realizzazione di CryptoCloud si è deciso di sfruttare tre programmi già esistenti per evitare di dover costruire tutto dalle fondamenta: CryptoFS, Dropbox e BouncyCastle.

3.2.1 CryptoFS

CryptoFS implementa lo schema di cifratura di Cryptomator sotto forma di libreria Java. È totalmente opensource: "controllare di persona è meglio che fidarsi di belle parole". Utilizza *java.nio.file.FileSystem* per la gestione delle vault, in modo che sia facilmente incorporabile in applicazioni terze.

3.2.2 Dropbox

La seconda tecnologia utilizzata è Dropbox, di cui verranno impiegate due componenti: l'SDK scritta in Java e il sync-client da loro offerto. Innanzitutto è stato scelto Dropbox perché, oltre ad essere molto diffuso, ha una documentazione molto precisa e dettagliata. L'SDK è stata usata per avere una gestione delle API sotto forma di classi: tutte le operazioni di download, upload, rimozione e lettura verranno così implementate grazie alle primitive offerte.
Il motivo dell'utilizzo del sync-client è ben preciso: era necessario per sfruttare CryptoFS in maniera fluida. Questo infatti, data una cartella criptata, non permette in nessun modo di conoscerne il contenuto finché questa non viene decifrata. Diventerebbe perciò obbligatorio il download dell'intero PwdFolder prima di ogni decifrazione: operazione con un costo relativamente alto, soprattutto se si dovesse avere PwdFolder di grosse dimensioni. Per evitare ciò, si è pensato di inserire l'intera cartella Dropbox sul proprio filesystem, utilizzando quindi il client di Dropbox per avere costantemente il sync con la cartella remota, in maniera che CryptoFS possa lavorare con il mount e l'unmount in maniera veloce e senza peggiorare l'esperienza dell'utente finale.

3.2.3 BouncyCastle

La terza tecnologia consiste in BouncyCastle, libreria opensource fornita anche per il linguaggio Java che si occupa di offrire le primitive di sicurezza.

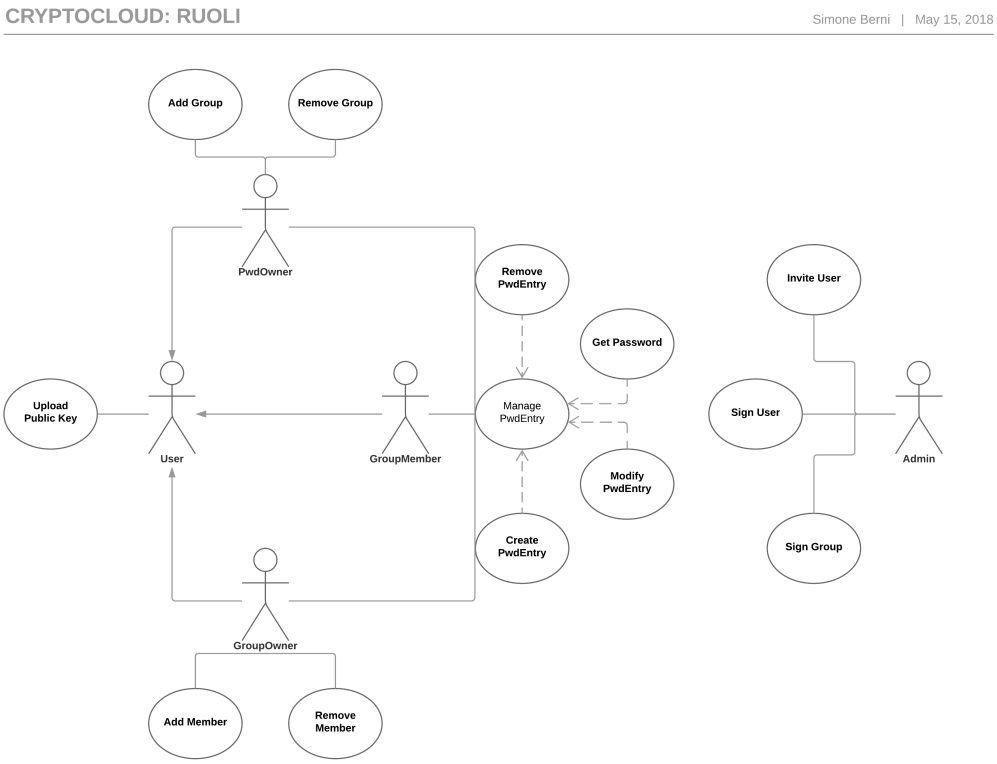
Permette la creazione di chiavi private e pubbliche fornendo inoltre all'utente la scelta dell'algoritmo preferito (come ad esempio RSA e la curva ellittica), esportazione e importazione di queste sotto forma di byte, firma e verifica di oggetti utilizzando le chiavi prima generate e infine cifratura e decifrazione. Della libreria verranno usate le primitive che riguardano l'algoritmo RSA: sarà necessario creare una coppia di chiavi per poter cifrare e decifrare i file, sarà implementata la gestione di firme e verifica per accertarsi dell'identità di un utente e della sua reale appartenenza ad un gruppo, saranno implementate sia l'esportazione che l'importazione di chiavi pubbliche per poterle inserire nel cloud e acquisirle.

3.3 Implementazione delle Specifiche

3.3.1 Ruoli

All'interno di CryptoCloud ci sono principalmente due ruoli che una persona può assumere: quello di Admin o quello di user.
La prima posizione è occupata dalla persona che ha il compito di amministrare la corretta gestione del sistema attraverso tre funzionalità: invito di un utente a far parte di CryptoCloud, firma o rimozione della chiave pubblica di un utente, firma o rimozione della associazione gruppo-owner. Con queste tre primitive un Admin è in grado di autorizzare e controllare tutti i suoi utenti all'utilizzo di CryptoCloud.
L'user invece è colui che all'interno dell'azienda potrà usufruire della maggior parte delle funzioni offerte da CryptoCloud, assumendo di volta in volta ruoli temporanei a seconda del compito che deve svolgere. I ruoli che può attribuirsi momentaneamente un utente sono tre: group member, PwdFolder's owner, group's owner.
Ad esempio quando un user vuole accedere ad un PwdFolder, potrà farlo per due motivi: o perché è l'owner del PwdFolder in questione, o perché è membro di un gruppo che ne ha ricevuto l'accesso.

In maniera più metodica, quando un utente sta assumendo il ruolo di membro di un gruppo ottiene la funzionalità aggiuntiva di gestire tutte le PwdEntry contenute dentro a tutti i PwdFolder condivisi al gruppo di cui fa parte.
Se invece è anche owner di un gruppo potrà effettuare l'aggiunta o la rimozione di membri di questo, continuando a mantenere la gestione di tutte le PwdEntry contenute dentro a tutti i PwdFolder condivisi con il gruppo.
L'ultimo caso invece consiste nell'essere owner di un PwdFolder, guadagnando la possibilità di condivisione il suo PwdFolder con i gruppi, o in caso rimuoverli, e di gestire a proprio piacimento le PwdEntry al suo interno.

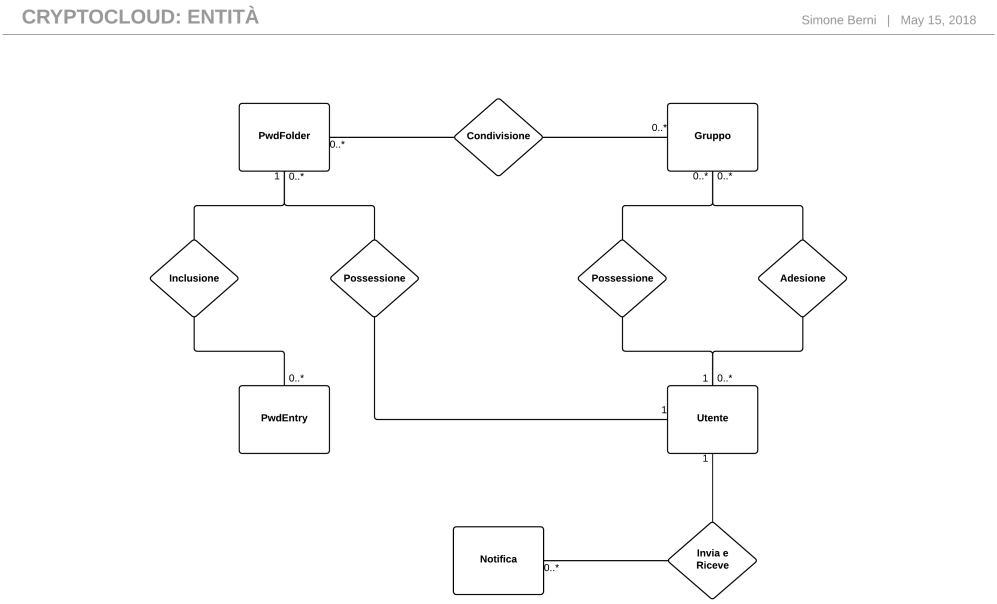


3.3.2 Entità

Le entità implementate sono le stesse richieste nelle specifiche della Sezione 1.2: è stato aggiunto solamente l'oggetto `Notification`.
L'utente è rappresentato da un'identità virtuale composta dai dati inseriti per la registrazione su Dropbox quali la email, il nome e il cognome. La

email è perciò una chiave che identifica l'utente. Ognuno possiede inoltre una coppia di chiavi, una privata salvata localmente e una pubblica inserita nel cloud. Per essere autenticato un utente deve prima di tutto essere autorizzato: per fare ciò l'Admin firma con la propria chiave privata la chiave pubblica dell'utente e inserisce la firma in una apposita struttura.
Il gruppo è rappresentato per la maggior parte delle volte da un file JSON avente i seguenti parametri: Nome, Owner e lista degli utenti che ne fanno parte. La composizione del gruppo viene poi firmata dall'owner stesso per evitare che altri utenti modifichino il file inserendosi al suo interno guadagnando diritti a loro non concessi. Ma questo livello di sicurezza non è sufficiente, poiché per un utente sarebbe possibile fingersi owner del gruppo e successivamente firmare con la propria chiave privata la composizione: per impedire ciò l'Admin a sua volta firma l'associazione tra gruppo e proprietario, risolvendo il problema.
Un gruppo inoltre viene anche rappresentato da un folder omonimo nel quale saranno inseriti i dati dei PwdFolder condivisi con questo.
Il PwdFolder invece si mostra in tre modalità diverse: un file JSON posseduto solamente dal suo creatore e avente come parametri Nome, Owner, Password, lista dei gruppi con cui è stato condiviso e lista dei permessi di questi ultimi; un folder vero e proprio, criptato e condiviso con tutti i membri di tutti gruppi inseriti precedentemente, e l'ultima possibilità con cui ci si può riferire al PwdFolder è un ulteriore file JSON posseduto solamente da tutti gli utenti con cui è stato condiviso: esso viene conservato nel gruppo (inteso come folder) grazie al quale gli utenti ne hanno ottenuto la condivisione, con parametri Nome, Password, Owner e Gruppo.
Per rappresentare il file sensibile vero e proprio si è utilizzato l'oggetto PwdEntry: essa è semplicemente un file JSON con i seguenti parametri: Nome, PwdFolder in cui è stata inserita, Sistema, Username, Password e Data di modifica.
La notifica invece è rappresentata da un file JSON con parametri diversi a seconda del tipo di messaggio che si vuole inviare: possono essere di ti-

po PwdFolderShared, PwdFolderDeleted, GroupDeleted, MemberAdded o MemberRemoved.



3.3.3 Composizione Cloud

Ogni utente nel proprio folder Dropbox dovrà avere tutti i folder necessari per il corretto funzionamento di CryptoCloud. Quelli che dovranno essere in comune a tutti gli utenti che utilizzano CryptoCloud saranno creati all'inizializzazione del sistema da parte dell'Admin; quelli strettamente personali invece verranno creati al primo accesso di ogni utente e saranno visibili ovviamente soltanto a loro. Ogni utente avrà perciò la seguente struttura di cartelle:

`System` in cui tutti hanno permessi di scrittura.

`SignedKeys` con permesso di lettura per gli utenti e di scrittura per l'Admin: all'interno solamente quest'ultimo potrà inserire i file che rappresenteranno la firma sulla chiave pubblica di ogni utente.

`SignedGroupsOwner` con permessi di lettura per gli utenti e di scrittura per l'Admin: all'interno quest'ultimo inserirà i file JSON che rappresenteranno l'associazione tra un gruppo e un suo owner.

`PersonalFolder` uno per utente: esso è criptato e perciò protetto grazie ad una password scelta dall'utente; essa verrà salvata in un file locale per velocizzare tutte le future richieste. Questa cartella è utilizzata per due funzioni: permettere all'utente di sapere quali PwdFolder ha creato e quali gli sono stati condivisi.

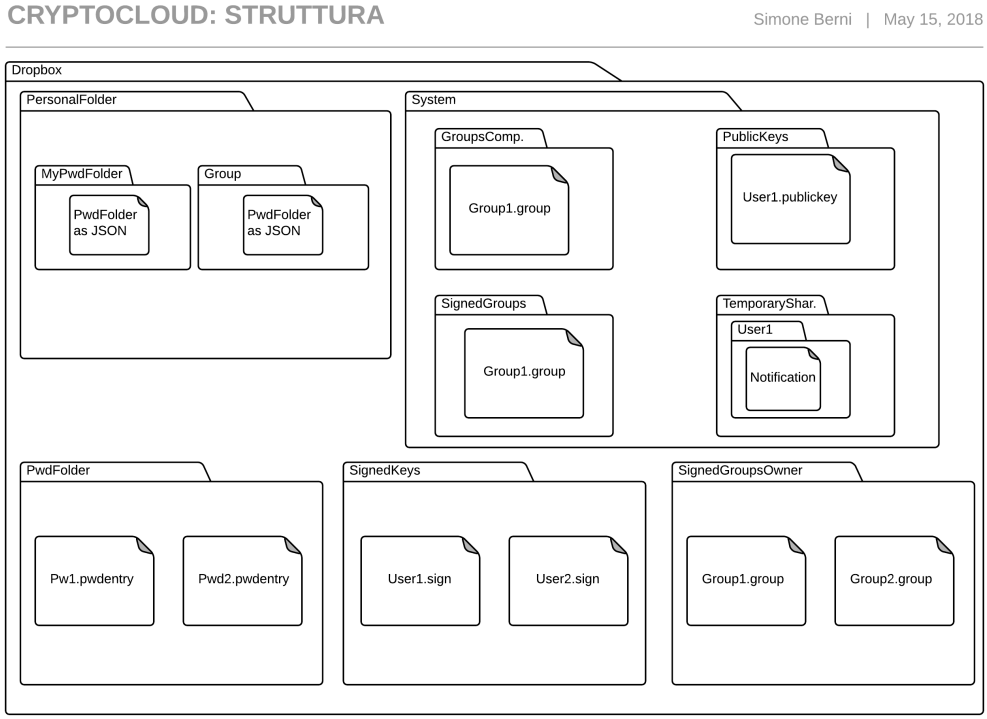
`GroupsComposition` interno a *System*: qui gli utenti inseriranno i gruppi da loro creati sotto forma di file JSON.

`SignedGroups` interno a *System*: gli owner dei gruppi inseriranno un file rappresentante la loro firma del gruppo.

`PublicKeys` interno a *System*: ogni utente inserirà la propria chiave pubblica, utilizzata poi per criptare messaggi.

`TemporarySharing` interno a textitSystem: ogni utente ha al suo interno una directory denominata come il proprio indirizzo email nel quale tutti gli altri utenti potranno inserire file JSON rappresentanti notifiche.

`MyPwdFolder` interno a *PersonalFolder*: qui vengono creati i file JSON rappresentanti i PwdFolder per il suo owner.



3.3.4 Problemi e Soluzioni

Tutto quello che si è voluto creare ha un unico scopo: non permettere a chiunque non sia il corretto destinatario di poter accedere ai dati sensibili. La cartella *PersonalFolder*, così come ogni PwdFolder in generale, è criptata per non consentire ai gestori di Dropbox di leggerne il contenuto; le notifiche che vengono salvate in *TemporarySharing*, quando contengono le password per accedere ai PwdFolder, sono criptate con la chiave pubblica del destinatario; l'integrità dei gruppi e degli utenti è protetta dalla firma dell'Admin. L'unica situazione in cui i dati sensibili sono leggibili è quando si trovano all'interno del computer del destinatario: di una notifica criptata dovrà essere effettuato il download, poi sarà decriptata, letta e infine cancellata.
Quando è necessario conoscere la password di un PwdFolder a cui si vuole accedere, è necessario recuperare il file corrispondente contenuto dentro a *PersonalFolder*: il folder viene montato esattamente come se fosse un filesy-

stem da parte dell'utente (quindi in cloud resta criptato, ma localmente si ha la versione decriptata) e il suo contenuto diventa facilmente accessibile. Dopo averne effettuata la consultazione è possibile smontarlo per permettere al PwdFolder di essere montato, poiché ne è stata recuperata la password, e di leggerne il contenuto.
Con questa suddivisione della posizione delle informazioni è garantita anche la privacy di ogni utente: si può sapere la composizione di tutti i gruppi che sono stati creati nel sistema, come richiesto dalle specifiche, ma non a quali PwdFolder un gruppo ha accesso finché non ne fa parte anche lui. Questo perché la composizione dei gruppi è stata inserita in un folder pubblico mentre la descrizione dei PwdFolder è salvata localmente dentro a *PersonalFolder*.
È necessario brevemente spiegare perché siano realmente necessari i folder *SignedKeys* e *SignedGroupsOwner* ma per fare ciò bisogna descrivere cosa avrebbe potuto compiere un utente malintenzionato che è stato inserito nel sistema per errore. Poiché tutto ciò che è interno a *System* è modificabile da tutti, un utente potrebbe facilmente fingersi qualcun altro. È sufficiente che modifichi la chiave pubblica di un altro utente con la propria e, se non ci fosse il controllo con la firma da parte dell'Admin, nessuno potrebbe essere sicuro che la chiave pubblica non sia stata alterata. Inoltre l'owner di un gruppo firma la composizione del gruppo stesso: nessun utente può aggiungersi ad un gruppo perché non potrebbe imitare la firma dell'owner. Potrebbe però assumere il comando del gruppo, sostituendosi all'owner e firmando lui stesso il gruppo e avrebbe a tutti gli effetti acquisito il controllo del gruppo stesso. Per questo motivo è necessario che l'Admin firmi l'associazione gruppo-owner: per garantirne la proprietà.
Resta di fatto il problema che un utente malintenzionato, anche se non potrà ottenere informazioni altrui, possa comunque provocare danni: potrà arrivare a cancellare file in tutte le cartelle contenute in *System* creando disagi ai membri che utilizzano CryptoCloud. In un certo senso potrebbe creare un attacco DOS, dal quale purtroppo non sarà possibile difendersi.

L'unica ipotetica protezione consiste nella prevenzione, cioè nel non invitare utenti per errore, o eliminare un utente non considerato affidabile. Una soluzione di difesa potrebbe essere quella di fare in modo che ogni utente dentro a *PublicKeys* possa leggere tutti i file ma modificare solamente quello corrispondente alla propria chiave pubblica, che possa dentro a *GroupsComposition* leggere tutti i file ma aggiornare solamente quelli che crea, e infine all'interno di *TemporarySharing* creare file nelle cartelle omonime agli utenti a cui vuole inviare notifiche ma che non possa eliminare file se non quelli nel proprio folder. Avendo questi vincoli iniziali, un utente non potrebbe fare nessun tipo di danno alla struttura del sistema perché non ne avrebbe i permessi, e sarebbe inoltre possibile rimuovere le cartelle *SignedKeys*, *SignedGroupsOwner* e *SignedGroups*.
Purtroppo però non è possibile implementare questo meccanismo per una limitazione di Dropbox: è possibile assegnare i permessi solamente al folder radice condiviso, e non è possibile modificarli per le sue cartelle interne. Avendo questo forte vincolo dato da Dropbox è necessaria la creazione dei file contenenti le firme e l'accettazione dell'esistenza di un possibile problema interno a CryptoCloud.
Rimane infine da spiegare l'utilizzo del folder *TemporarySharing* e quando due utenti avrebbero la necessità di comunicare tra loro: si supponga per esempio che un PwdFolder P sia stato condiviso con un gruppo G. Se G aggiunge un nuovo membro M, anche M dovrà poter accedere a P. Per fare ciò l'owner di G invierà una notifica all'owner di P di aggiungere anche l'utente M. Questa notifica non è presa per assoluta verità poiché sarebbe facile coniarla: quello che realmente accade è che dopo aver ricevuto la notifica l'owner di P controlla che sia realmente cambiato G e in caso affermativo, aggiunge X a P con gli stessi permessi di tutti gli altri utenti di G.

3.4 Implementazione delle Operazioni

Le operazioni vengono suddivise in due categorie, poiché possono essere compiute da due categorie diverse di utenti: un Admin potrà svolgere solamente le operazioni per la gestione delle autorizzazioni, mentre gli utenti avranno quelle per amministrare i gruppi e i PwdFolder.

3.4.1 Operazioni dell'Admin

Le seguenti operazioni permettono ad un Admin di svolgere il suo compito in maniera adeguata:

`signUser()` Elenca gli utenti in attesa di una approvazione e dopo che l'Admin ne ha selezionato uno, firma la chiave pubblica dell'user selezionato.

`signGroup()` Elenca i gruppi in attesa di una approvazione e dopo che l'Admin ne ha selezionato uno, crea il file JSON contenente l'associazione tra il gruppo e il suo owner.

`removeSignUser()` Rimuove la firma precedentemente creata per un utente.

`removeSignGroup()` Rimuove la firma precedentemente creata per un gruppo che nel frattempo è stato cancellato.

`addUsers()` Aggiunge degli ulteriori nuovi utenti al sistema CryptoCloud dell'Admin. Ciò consiste nell'aggiungere la condivisione agli utenti delle cartelle *System*, *SignedKeys*, *SignedGroupsOwner*.

`removeUsers()` Rimuove degli utenti tra quelli appartenenti al sistema CryptoCloud dell'Admin. Ciò consiste nel rimuovere la condivisione agli utenti delle cartelle *System*, *SignedKeys*, *SignedGroupsOwner*.

3.4.2 Operazioni dell'User

Tutte le prossime operazioni elencate sono eseguibili soltanto da un utente che è stato autorizzato da un Admin. Ogni operazione di aggiunta o rimo-

zione di un gruppo, controlla che il gruppo scelto sia verificato; se non fosse così l'operazione non verrà portata a termine.
È facile notare che state implementate tutte le operazioni richieste nella fase di specifica.

`mountSystem()` Permette all'utente di inserire sul proprio account le cartelle *System*, *SignedKeys* e *SignedGroupsOwner*.

`listUsers()` Stampa a video tutti gli utenti che sono stati aggiunti da un Admin al sistema.

`listGroups()` Stampa a video tutti i gruppi creati e che quindi compaiono nella cartella *GroupsComposition*.

`listMyGroups()` Stampa a video tutti i gruppi a cui l'utente partecipa e con i quali è stato condiviso almeno un PwdFolder.

`listPwdFolders()` Stampa a video tutti i PwdFolder che sono stati condivisi con l'utente. Viene eseguita controllando quali cartelle condivise esistono nel Dropbox dell'utente escludendo *System.SignedKeys* e *SignedOwnerGroup*.

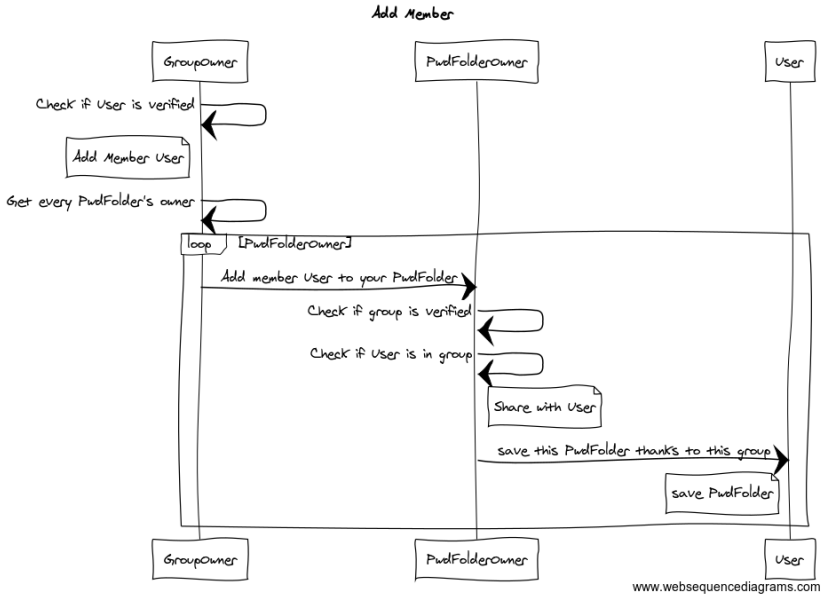
`listOwningPwdFolders()` Stampa a video tutti i PwdFolder di cui l'utente è owner. Viene eseguita elencando i PwdFolder contenuti dentro a *MyPwdFolder*

`listPwdEntries()` Stampa a video tutte le PwdEntry interne ai propri PwdFolder.

`createGroup()` Crea un gruppo inserendo l'utente che invoca il metodo come suo owner, permette l'aggiunta a piacimento di membri all'interno del gruppo e infine lo firma. Il gruppo in formato JSON sarà inserito in *GroupsComposition* e la sua firma in *SignedGroups*.
L'Admin, essendo in polling nella cartella *GroupsComposition* verrà avvisato della creazione di un nuovo gruppo e digitando un unico carattere potrà approvare o meno la creazione del gruppo. In questo

modo, nessun utente può impersonarsi owner del gruppo in quanto non potrebbe modificare il file creato dall'Admin.

`addMembers()` Aggiunge nuovi utenti ad un gruppo di cui l'utente è owner. Consiste nella modifica del JSON che rappresenta il gruppo presso *GroupsComposition* e della sostituzione della sua firma con una nuova per confermare i componenti del gruppo. Vengono inoltre notificati tutti gli owner di tutti i PwdFolder a cui il gruppo aveva ricevuto accesso, in modo che anche i nuovi utenti possano consultarli inviando una notifica di tipo AddMember. In maniera automatica, una volta ricevuta la notifica, si accerterà che il gruppo abbia veramente i nuovi membri e, in caso affermativo, recuperando dal PwdFolder in formato JSON i permessi che erano stati assegnati al gruppo che ha fatto la richiesta vengono conferiti anche ai nuovi membri. Infine vengono condivise le credenziali di accesso al PwdFolder a tutti i nuovi membri.
Una sintesi del flusso è il seguente:



`removeMembers()` Rimuove degli utenti da un gruppo di cui l'utente chiamante è owner. Non è possibile rimuovere sé stessi con questa operazione. Consiste nella modifica del JSON che rappresenta il gruppo

presso *GroupsComposition* e della sostituzione della sua firma con una nuova per confermare i componenti del gruppo. È inoltre necessario notificare tutti gli owner di tutti i PwdFolder a cui il gruppo aveva ricevuto accesso, in modo che ne rimuovano la condivisione con gli utenti eliminati.

Viene inoltre inviata a tutti gli owner di tutti i PwdFolder che erano stati condivisi con il gruppo una notifica di tipo MemberRemoved poiché dovranno toglierne la condivisione con i membri rimossi. Automaticamente una volta ricevuta la notifica, si accerterà che il gruppo non abbia veramente i membri elencati nella notifica e che il gruppo stesso faccia parte di quelli con cui ha effettuato la condivisione del PwdFolder: in caso positivo, rimuove l'accesso agli utenti. Infine vengono notificati i singoli utenti affinché rimuovano le credenziali del PwdFolder a cui, in ogni caso, non potranno più accedere.

`deleteGroup()` Elimina un gruppo se vengono rispettate due condizioni: l'utente che vuole effettuare l'operazione è l'owner ed è l'unico membro del gruppo. In caso positivo il gruppo e la sua firma verranno eliminati. Sarà inoltre cancellato all'interno di *PersonalFolder* il folder omonimo al gruppo in questione: i file JSON rappresentanti i PwdFolder ottenuti attraverso quel gruppo non hanno più necessità di rimanere salvati, poiché anche i PwdFolder intesi come cartelle vengono rimossi. Prima di questo però è stata inviata una notifica a tutti gli owner dei PwdFolder, in maniera che eliminino il gruppo e che tolgano la condivisione con l'ex owner del gruppo.

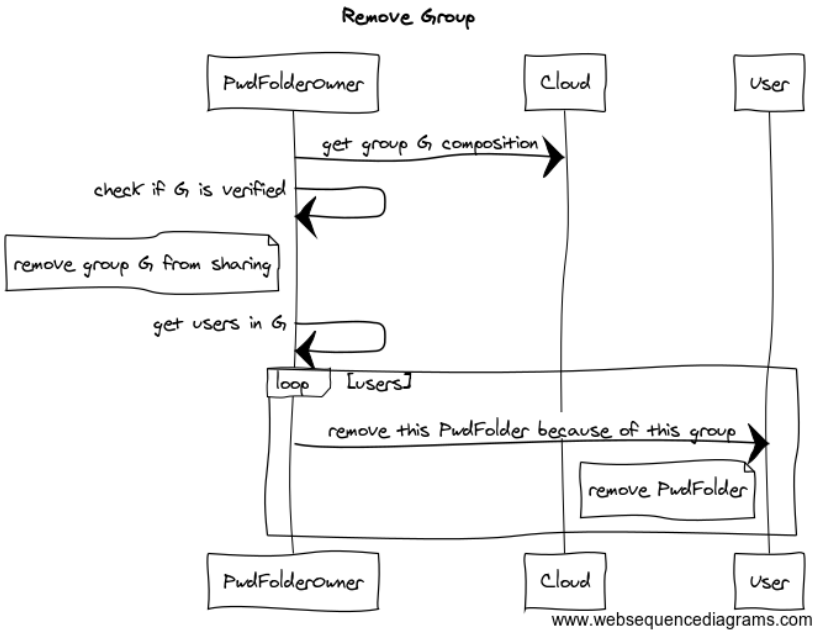
La notifica è di tipo GroupDeleted e una volta ricevuta CryptoCloud si accerterà che il gruppo non esista veramente più e dopo aver tolto la condivisione con quello che ormai è l'ex-owner del gruppo, modificherà il file JSON corrispondente al PwdFolder.

`createPwdFolder()` Crea un nuovo PwdFolder permettendo all'utente di sceglierne un nome, una password e un insieme di gruppi con associati

i relativi permessi che avranno nel PwdFolder: o di lettura o di scrittura. Vengono prodotti due oggetti: un file JSON che verrà inserito nel *MyPwdFolder* avente per parametri tutte le informazioni prima inserite e un folder. Quest'ultimo viene condiviso con tutti i membri dei gruppi scelti e successivamente criptato con la password scelta dal suo owner. Infine viene inviata a tutti gli utenti aggiunti una notifica PwdFolderShared in modo che possano salvare la password del PwdFolder per un futuro accesso.

`addGroups()` Aggiunge nuovi gruppi ad un PwdFolder di cui l'utente è owner. Consiste nel condividere con tutti i membri di tutti i gruppi desiderati il PwdFolder, ovviamente con il permesso desiderato, e nel modificare il file JSON che lo rappresenta contenuto in *MyPwdFolder*. Infine vengono notificati tutti gli utenti aggiunti, anche in questo caso con la notifica di tipo PwdFolderCreate, in modo che possano salvare la password del PwdFolder per un futuro accesso.

`removeGroups()` Rimuove dei gruppi da un PwdFolder di cui l'utente è owner. Consiste nel togliere dalla condivisione tutti i membri di tutti i gruppi desiderati dal PwdFolder e nel modificare il file JSON che lo rappresenta contenuto in *MyPwdFolder*. Infine è inviata una notifica di tipo PwdFolderDelete tutti gli utenti dei gruppi rimossi in modo che possano cancellare le informazioni relative al PwdFolder.
Un esempio di cosa la chiamata comporta è il seguente:



`deletePwdFolder()` Elimina un PwdFolder di cui l'utente è owner, togliendone l'accesso a tutti i gruppi. Ciò consiste nella rimozione del file JSON contenuto in *MyPwdFolder* e nell'annullamento della condivisione del PwdFolder. Infine vengono notificati tutti gli utenti dei gruppi rimossi, utilizzando sempre una notifica di tipo PwdFolderDelete, in modo che possano cancellare le informazioni relative al PwdFolder.

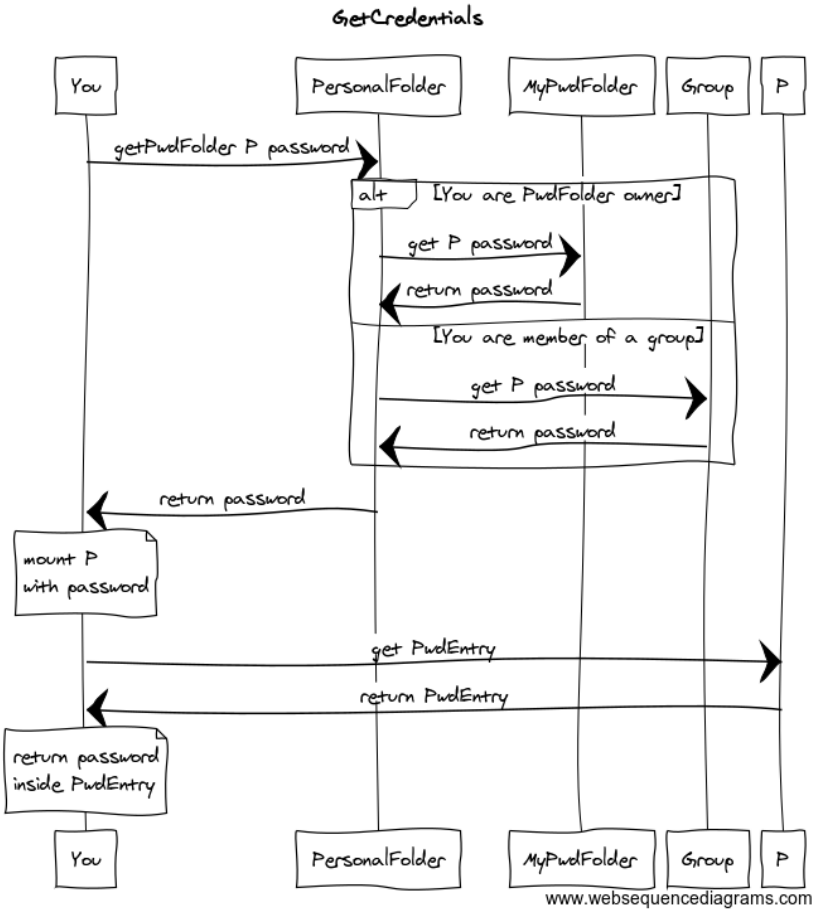
`createPwdEntry()` Crea una nuova PwdEntry nella quale l'utente può decidere i seguenti campi: PwdFolder in cui inserirlo, il nome univoco che identifica la PwdEntry, il sistema in cui dovrà essere usato, l'username di accesso e la password di accesso. Il file JSON che corrisponde alla PwdEntry verrà infine inserito nel PwdFolder selezionato.

`modifyPwdEntry()` Modifica una PwdEntry già esistente nella quale l'utente può scegliere una nuova password.

`deletePwdEntry()` Elimina una PwdEntry già esistente.

`getCredentials()` Stampa a video l'username e la password contenuti in una PwdEntry.

Uno schema del flusso complessivo per ricavare le credenziali è il seguente:



Capitolo 4

Architettura dell'Applicazione: punto di vista per Sviluppatori

Il software è stato sviluppato utilizzando come linguaggio Java, adottando perciò il metodo della programmazione ad Oggetti.
Poiché il codice è opensource, l'intero sorgente è reperibile su github presso https://github.com/0ssigeno/cryptocloud, così come la documentazione di ogni classe sotto forma di Javadocs. CryptoCloud è formato da sei classi principali: `User`, `Admin`, `Group`,`PwdFolder`,`PwdEntry` e `Notification` che rispondono agli oggetti individuati durante la fase di progettazione.
È inoltre presente l'oggetto `ProjectHandler` che implementa tutte le primitive di utility utilizzate dentro l'applicazione, come ad esempio la corretta gestione dell'input dell'utente, la trasformazione di un oggetto in JSONObject. Altre tre classi necessarie per il corretto funzionamento di CryptoCloud sono: `DropboxFactory` che produce il client di Dropbox, `KeysFactory` che genera la coppia di chiavi pubblica-privata e `Cryptomator` che si occupa di generare e decriptare filesystem protetti.
Da questo punto verranno presentate brevemente le classi utilizzate per l'implementazione, fornendo note sulle scelte implementative che riguardano i loro costruttori e i loro metodi principali.

4.1 Classe ProjectHandler

Tutto ciò che riguarda le costanti, come le estensioni dei file che si verranno a creare o i path per raggiungere i vari folder interni a Dropbox, è reperibile da questa classe. Sono inoltre presenti metodi statici per la cancellazione di file e directory, trasformazione di JSONObject in file e il viceversa in aggiunta alla funzione `inputUser()` che si occupa della corretta gestione dell'input stream.

4.2 Classe KeysFactory

La classe ha come obiettivo creare o prelevare la coppia di chiavi dell'utente e per questo motivo possiede due attributi statici:

`PublicKey` la chiave pubblica dell'utente

`PrivateKey` la chiave privata dell'utente

Costruttore

Se non è possibile recuperare le chiavi dalla cartella locale in quanto è la prima volta che l'utente esegue CryptoCloud è necessario che la coppia di chiavi sia creata. Utilizzando l'algoritmo RSA e il provider fornito da *BouncyCastle*, le chiavi sono generate e assegnate ai due attributi della classe per essere facilmente recuperabili.

```
if (public1key.exists() && private1key.exists()) {
  publicKey =
     KeysFunctions.importPublic(public1key.getCanonicalPath());
  privateKey =
     KeysFunctions.importPrivate(private1key.getCanonicalPath());
} else {
  KeyPair keypair = generateKeyPair();
  KeysFunctions.exportKeys(keypair);
```

```
  publicKey = keypair.getPublic();
  privateKey = keypair.getPrivate();
}
```

Dopodiché le chiavi vengono esportate in locale dentro la cartella *CryptoCloud*, situata nella home dell'utente, in modo da poterle recuperare per le future esecuzioni del software.

4.3 Classe KeysFunctions

La classe è formata solamente da funzioni statiche e vi sono racchiuse tutte le utility necessarie per utilizzare le chiavi generate dalla classe `KeysFactory`.

Metodi

È possibile suddividere i metodi in tre categorie: quelli che si occupano della gestione delle chiavi, quelli che effettuano la cifratura o la decifrazione di byte, quelli che si occupano della generazione e della verifica di firme.

exportKeys()

Data la coppia di chiavi pubblica e privata esse vengono innanzitutto codificate utilizzando i metodi forniti dalle classi `PublicKey` e `PrivateKey` del pacchetto *java.security*. Infine vengono salvate nella cartella *CryptoCloud* nella home dell'utente.

generatePkcs1Signature()

Crea l'array di byte che rappresenta la firma fatta dalla propria chiave privata su un input. La difficoltà nella creazione della firma è delegata alla classe `Signature`, anche essa offerta dal pacchetto *java.security*.

```
Signature signature = Signature.getInstance("SHA384withRSA", "BC");
signature.initSign(rsaPrivate);
```

```
signature.update(input);
return signature.sign();
```

encrypt()

Si occupa di cifrare un blocco di byte utilizzando la propria chiave pubblica: come nei casi precedenti, la maggior parte della difficoltà è delegata ad un'altra classe, `Cipher`, che fa parte del pacchetto *java.crypto*.

```
Ciptherr cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
cipher.init(Cipher.ENCRYPT_MODE, key);
cipherText = cipher.doFinal(text);
```

4.4 Classe DropboxClientFactory

La classe nasce per risolvere una necessità ben precisa: avere un *Singleton* dell'oggetto `DbxClientV2`, generato dalla SDK, per collegarsi al cloud. Per questo motivo l'attributo `client` è statico, in modo che non sia necessario istanziare la classe per recuperare l'oggetto.

Costruttore

Precisamente in questa fase vengono generati due client, uno utilizzato normalmente da tutto CryptoCloud, il secondo solamente dalla classe `Dropboxpolling`. Per la creazione del client è necessario possedere il token di accesso: l'utente verrà reindirizzato alla pagina web di Dropbox dove potrà effettuare il login e consentire a CryptoCloud di accedere ai suoi dati personali.

```
DbxAppInfo appInfo = new DbxAppInfo(KEY_INFO, SECRET_INFO);
DbxWebAuth webAuth = new DbxWebAuth(requestConfig, appInfo);
DbxWebAuth.Request webAuthRequest = DbxWebAuth.newRequestBuilder()
```

```
.withNoRedirect()
.build();
String authorizeUrl = webAuth.authorize(webAuthRequest);
```

Una volta inserito il token nella console, esso verrà esportato localmente nella cartella *CryptoCloud* per evitare che venga richiesto ad ogni nuovo avvio del software.

```
String code = ProjectHandler.inputUser();
DbxAuthFinish authFinish = webAuth.finishFromCode(code);
client = new DbxClientV2(requestConfig,
    authFinish.getAccessToken());
authInfo = new DbxAuthInfo(authFinish.getAccessToken(),
    appInfo.getHost());
File output = new File(url.toString());
output.getParentFile().mkdirs();
DbxAuthInfo.Writer.writeToFile(authInfo, output);
```

4.5 Classe DropboxFunctions

In questa classe formata solamente da funzioni statiche sono racchiuse tutte le utility invocate per utilizzare le potenzialità offerte dal client di Dropbox generato dalla classe `DropboxClientFactyory`.

Metodi

Di seguito sono spiegati solamente i metodi che hanno qualche particolarità degna di nota.

uploadPersonalFile()

La funzione è strutturata in due parti, a seconda della dimensione del file di cui si vuole effettuare l'upload: se non supera la dimensione massima, viene

utilizzata la classe `UploadBuilder` che gestirà autonomamente l'upload. Se invece il file è troppo grande per essere inviato come una unica entità, l'upload viene eseguito dalla funzione `chunkedPersonalUploadFile()`.

```
if (size >= ProjectHandler.CHUNKED_UPLOAD_CHUNK_SIZE) {
   System.err.println("File too big, uploading the file in
       chunks.");
   chunkedPersonalUploadFile(client, localFile, dropboxPathNoName);
} else {
   InputStream in = new FileInputStream(localFile);
   client.files().uploadBuilder(dropboxPathNoName)
   .withMode(WriteMode.ADD)
   .withClientModified(new Date(localFile.lastModified()))
   .withMute(notification)
   .uploadAndFinish(in);
}
```

downloadPersonalFile()

Effettua il download di un file da Dropbox alla propria cartella *tmp*, mantenendo invariato il nome del file. Per la parte di download vero e proprio ci si affida alla classe `DownloadBuilder` fornita dall'SDK di Dropbox.

```
FileOutputStream out = new FileOutputStream(localFile);
client.files().downloadBuilder(dropboxPathAndFileName)
.download(out);
return localFile;
```

addMembersToFolder()

Condivide un folder con una lista di `User` al folder passato come parametro: l'SDK di Dropbox richiede che l'utente sia in realtà una istanza della

classe `AddMember` per poterne effettuare la condivisione. È necessario perciò un cast che viene eseguito dalla funzione `usersToAddMember()`.

```
MemberSelector newMember = MemberSelector.email(user.getEmail());
AddMember newAddMember = new AddMember(newMember, accessLevel);
```

4.6 Classe DropboxPolling

Il compito principale della classe è quello di effettuare polling su una cartella per ricevere aggiornamenti. La particolarità consiste nel mettersi in ascolto su un folder attraverso le api fornite da Dropbox e non direttamente dal filesystem, e per questo motivo utilizza tutte le classi fornite dal SDK di Dropbox e i relativi metodi.
Il metodo migliore per effettuare la sua mansione senza bloccare l'utente che vuole utilizzare CryptoCloud, è utilizzare un nuovo thread per ogni cartella in cui si vuole essere in ascolto: per questa ragione la super classe è proprio `Thread` e di questa effettua l'override del metodo `run`. L'oggetto contiene quattro attributi:

`Client` un oggetto `DbxClientV2` utilizzato per effettuare le operazioni sui file dentro a Dropbox

`LongPollClient` un oggetto `DbxClientV2` utilizzato per controllare l'aggiunta o la rimozione di file in un determinato path

`Type` il tipo di utente che ha creato la classe

`DropboxPath` il path in cui deve essere eseguito il polling

Costruttore

L'oggetto verrà creato durante l'utilizzo del metodo `setup()`, e può essere invocato con tre valori dell'attributo `type` diversi: *User*, *Admin1* o *Admin2*. Se è stato creato dalla classe `User` allora il valore di `type` sarà *User*, e il polling

dovrà essere eseguito sul folder dell'user in *TemporarySharing*; se invece è stato generato durante dalla classe `Admin`, il costruttore verrà invocato due volte, una con parametro *Admin1* per essere in polling sul folder *PublicKeys* in modo da sapere quando avviene l'aggiunta di nuove chiavi, l'altra con valore *Admin2* per essere notificati nella cartella *GroupsComposition* quando nuovi gruppi vengono creati e sono in attesa dell'autorizzazione.

Metodi

run()

Il metodo è ereditato dalla classe `Thread` e permette di controllare quando un nuovo file è inserito nella cartella di cui si sta effettuando il polling. Ogni volta che viene individuata una modifica nella cartella, viene invocato il metodo `answerChanges()` che si occuperà di gestire la notifica.

```
while(true){
  ListFolderLongpollResult result = longPollClient.files()
  .listFolderLongpoll(cursor, longpollTimeoutSecs);
  if (result.getChanges()) {
      cursor = answerChanges(cursor);
   }
}
```

answerChanges()

Grazie a questo metodo, in associazione alla creazione della giusta notifica, è possibile automatizzare molte funzioni all'interno di CryptoCloud. In particolare gestisce automaticamente tutte le notifiche che arrivano: se però l'attributo `type` ha valore diverso da *User* si richiedrà all'Admin di confermare l'azione per evitare impersonamenti, mentre nell'altro caso delega la responsabilità della corretta gestione della notifica alla classe `Notification`.

Una notifica è considerata valida se ha la giusta estensione ed è stata posizionata nel folder corretto; se non è cosi il file non sarà considerato degno di essere analizzato e verrà subito cancellato.

4.7 Classe Cryptomator

La finalità di questa classe è unica: aprire e poi utilizzare filesystem protetti da password. Con il termine protetti si intende che chi non conosce la password di accesso non solo non può leggere i file al suo interno ma non può neanche sapere dell'esistenza del contenuto.
Gli attributi della classe per questo motivo sono il filesystem che si vuole rendere sicuro e la password per sbloccarlo.
Ogni filesystem verrà poi rappresentato, mentre è criptato, come un folder: dentro CryptoCloud l'entità che deve essere protetta è il `PwdFolder` e il suo contenuto, e per questo motivo si farà un uso intenso della classe Cryptomator quando bisognerà cifrarlo, decifrarlo o leggere le `PwdEntry` al suo interno.
Sono inoltre presenti all'interno tutti path che verranno utilizzati per trovare in quale punto, all'interno del proprio filesystem, è stata inserita la cartella *Dropbox* che gestisce l'operazione di sync con il cloud.
Non sono stati utilizzati `Singleton` in questa classe per evitare problemi di sicurezza: si vuole infatti che ogni filesystem rimanga aperto, e quindi decriptato, per il minor tempo possibile, anche se questo comporta un overhead delle operazioni di cifratura e di decifrazione di questi.

Costruttore

Il costruttore prende come parametri la posizione del filesystem che si vuole andare a decriptare e la sua password. Se questa ultima è corretta allora il filesystem viene sbloccato e diventa possibile eseguire tutte le operazioni che erano effettuabili in un normale filesystem.

Metodi

Sono stati creati metodi banali che provocano modifiche nel filesystem, come `deleteFile()`, `copyFileFromExternal()` e `copyFileFromVault()` che facilitano la gestione dei path nel filesystem una volta decifrato.

setStandardPassword()

Metodo invocato quando la cartella che si vuole decriptare è *PersonalFolder* in modo da settare l'attributo *Password* correttamente e senza richiederla come input all'utente. Durante la creazione del folder in questione, è stato infatti richiesto di digitare una password e di non dimenticarsene: è stata salvata localmente in un file denominato *PersonalPassword* e ogni volta che si cercherà di decriptare il folder esso verrà letto ed utilizzato. Se non è più possibile trovare il file, verrà richiesto all'utente di digitare nuovamente la password e verrà nuovamente salvata in locale per facilitare le consultazioni future.

```
Path path = Paths.get(ProjectHandler.MY_PERSONAL_PATH + "/" +
    "PersonalPassword");
try (BufferedReader reader = Files.newBufferedReader(path)) {
   return (reader.readLine());
} catch (IOException e) {
   System.out.println("Couldn't get your PersonalFolder password
       automatically, please enter it");
   String pwd = ProjectHandler.inputUser();
   try {
      Files.write(path, pwd.getBytes(), StandardOpenOption.CREATE);
   } catch (IOException e1) {
      System.out.println("Couldn't save your personal password");
   }
   return pwd;
}
```

4.8 Classe User

Un utente rappresenta un individuo che utilizza CryptoCloud e si occupa di unire due concetti differenti: l'account che la persona ha utilizzato per collegarsi su Dropbox, di cui è importante l'Email poiché diventerà la chiave per riconoscere l'utente, e la chiave pubblica che ha generato. Per ottenere questo risultato, sono stati implementati i seguenti attributi:

`Email` la stringa che rappresenta l'email dell'utente

`PublicKey` la chiave pubblica dell'utente

`Verified` il booleano che rappresenta se l'utente è verificato o meno

In aggiunta all'interno della classe è presente un oggetto statico `User` denominato *yourself*: viene inizializzato al momento dell'avvio del software e rappresenta l'utente che sta eseguendo CryptoCloud. Poiché l'oggetto è statico, tutte le altre classi potranno recuperare le informazioni su questo preciso `User` in ogni momento e senza doverlo creare da zero: in poche parole la classe possiede anche un *Singleton* del chiamante.

Costruttore

La costruzione dell'oggetto inizialmente consiste soltanto nell'assegnamento della sua email e si è deciso di non inizializzare al momento della creazione gli attributi `PublicKey` e `Verified` per un motivo ben preciso: il settaggio di questi due valori richiederebbe infatti il download di tre file, che si vorrebbe evitare se non realmente necessario, in modo da non avere rallentamenti inutili durante l'esecuzione di CryptoCloud.
L'implementazione adottata effettua il set degli attributi dentro le funzioni `get`:

```
PublicKey getPublicKey() {
   if (publicKey == null) {
      setPublicKey();
```

```
    }
    return publicKey;
}
```

Metodi

setup()

Il metodo è invocato ad ogni avvio di CryptoCloud. Se non è necessario creare la propria struttura di cartelle perché già esistente, viene controllata la presenza o meno di notifiche nel proprio *TemporarySharing*. In caso positivo, vengono eseguite le notifiche in maniera automatica creando l'oggetto `Notification`.

```
Notification notification = new Notification();
if (notification.getNotifications().isEmpty()) {
    System.out.println("There isn't any new notification");
} else {
    notification.doNotifications();
}
```

Se invece è la prima volta in assoluto che si avvia il sistema verranno eseguite in sequenza le seguenti operazioni per permettere di conservare i propri dati sensibili su Dropbox e di comunicare con gli altri utenti invitati dall'Admin:

- Accettare l'invito della condivisione, o mount del folder nel proprio account Dropbox, delle cartelle *System*, *SignedKeys* e *SignedOwnerGroup*.
- Upload della propria chiave pubblica nel folder *PublicKeys*.
- Creazione del folder *PersonalFolder*.
- Creazione del folder *MyPwdFolder*.

Che sia o meno il primo avvio, viene inoltre effettuata ulteriormente un'operazione:

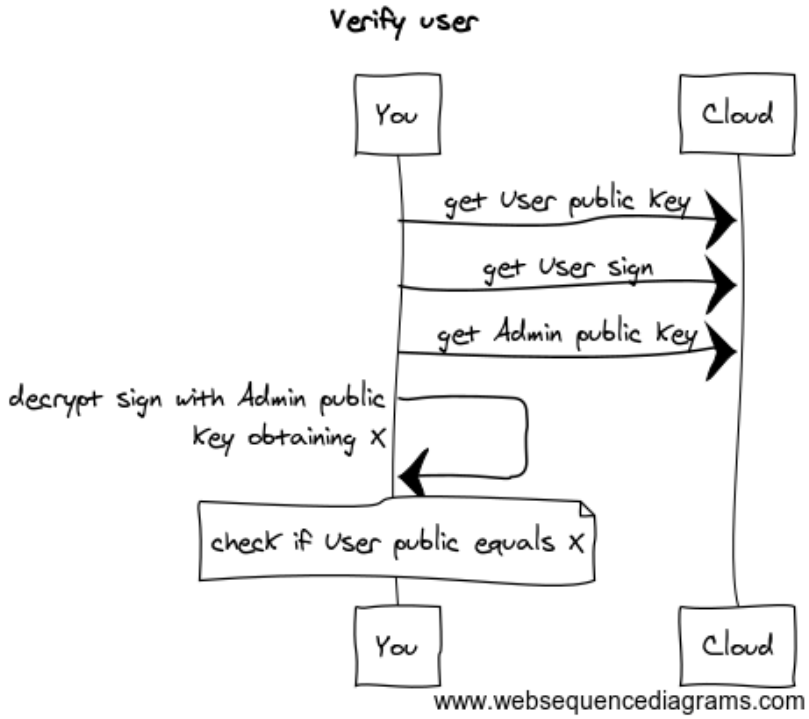
- Creazione della classe `DropboxPolling` su un nuovo thread che si occupa di ascoltare cambiamenti nel proprio folder *TemporarySharing*: se un altro utente invia una `notifica` CryptoCloud necessita di essere avvertito per poterla eseguire istantaneamente e automaticamente.

setVerify()

Il metodo è invocato quando alla classe si vuole assegnare il valore dell'attributo `Verified`. Questo può accadere in diversi casi, ad esempio prima di potere aggiungere un utente in un gruppo. Visto che tutti gli altri oggetti non lavoreranno mai con utenti singoli ma bensì con gruppi, si potrà dare per scontato che l'utente è verificato.
Il valore è settato a `True` se la firma dell'Admin sulla chiave dell'utente è autenticata. Per effettuare il controllo prima di tutto bisogna ottenere la chiave dell'user, il file che rappresenta la firma da parte dell'Admin e sua la chiave pubblica. Una volta ottenuti questi file, si può controllare che tutto combaci.
Se il file contenente la chiave pubblica è equivalente alla decifrazione del file rappresentante la firma attraverso la chiave dell'Admin, allora l'utente è verificato.

```
verified=(KeysFunctions.verifyPkcs1Signature(adminKey,
    publicKeyByte, sign));
```

Uno schema che semplifica la comprensione della verifica dell'utente è il seguente:



createGroup()

Crea un nuovo oggetto `Group` del quale il chiamante è suo owner e può decidere il nome, controllando che non esista già un omonimo, per poi scegliere tutti gli utenti che si vogliono aggiungere attraverso l'invocazione del metodo `addMembers()` della classe `Group`.
Dopo averne settato i parametri, l'oggetto è trasformato in un file JSON e viene effettuato l'upload sia del file JSON che della firma dell'owner.

createPwdFolder()

L'idea alla base è la stessa utilizata da `createGroup()`: il chiamante assume il ruolo di owner del `PwdFolder` e potrà sceglierne il nome univoco, una password per cifrarne il contenuto e avrà la possibilità di aggiungere gruppi con associati i permessi di accesso. Questo ultimo punto è gestito dal co-

struttore del `PwdFolder`, che invocherà il metodo `addGroups()` della classe `GroupsPermissions`.
L'oggetto deve essere poi trasformato in due entità: un file JSON che sarà posizionato nel folder *PersonalFolder* dopo averlo decriptato grazie alla classe `Cryptomator`

```
File file = ProjectHandler.jsonToFile(json,
    ProjectHandler.values.VAL_PWDFOLDER);
Cryptomator cryptomator = new Cryptomator();
cryptomator.moveFileFromExternal(file.getAbsolutePath(),
    Cryptomator.pathMyPwdFolder + "/" + file.getName());
```

e una vera e propria cartella nel cloud protetta dalla password scelta:

```
DropboxFunctions.createSharedFolder(name);
Cryptomator.createStandardStorage(Cryptomator.pathBase + "/" +
    name, password);
cryptomator.exit();
```

Infine il folder è reso accessibile a tutti i membri di tutti i gruppi a cui si è garantito l'accesso e ne vengono condivise le informazioni fondamentali per accedervi, quali nome, password e il gruppo grazie al quale ne hanno ottenuto l'accesso, attraverso la creazione di una notifica `PwdFolderShared` effettuata dalla funzione `shareToMember()` della classe `PwdFolder`.

createPwdEntry()

L'utente dopo aver selezionato in quale PwdFolder inserirla, deciderà il nome univoco che l'oggetto `PwdEntry` dovrà possedere:

```
PwdEntry pwdEntry = new PwdEntry(ProjectHandler.inputUser());
while (pwdFolder1.getPwdEntries().contains(pwdEntry)) {
   System.out.println("Name already exists: try again");
```

```
    pwdEntry = new PwdEntry(ProjectHandler.inputUser());
}
```

Dopodiché si settano i vari attributi e infine si effettua l'upload all'interno del folder prescelto.

4.9 Classe Notification

La classe ha due obiettivi: data una classe e un problema, creare un JSONObject che li rappresenti, e data una notifica eseguire la corretta operazione per risolverla.
Tra i metodi della classe infatti ne sono presenti alcuni *statici* che si occupano della costruzione dei vari tipi di notifiche che si è deciso di implementare dentro a CryptoCloud, grazie ai quali è stato possibile creare il sistema di message passing:

`PwdFolderShared` indica che il ricevente ha ottenuto la condivisione di un nuovo PwdFolder

`PwdFolderDeleted` indica che il ricevente ha perso la condivisione di un PwdFolder

`GroupDeleted` indica che un gruppo a cui il ricevente aveva condiviso un PwdFolder è stato rimosso

`AddMember` indica che un gruppo a cui il ricevente aveva condiviso un PwdFolder ha aggiunto almeno un nuovo membro

`RemoveMember` indica che un gruppo a cui il ricevente aveva condiviso un PwdFolder ha eliminato almeno un membro

La notifica è rappresentata dentro a Dropbox come un file JSON posizionato nel folder *TemporarySharing*; Essa verrà poi riconosciuta in base alla

sua estensione: ogni tipologia ha la propria, ad esempio la notifica `AddMember` avrà come estensione una volta trasformata in file `.addM`.

Costruttore

I costruttori della classe sono due, a seconda che si voglia gestire una notifica sola o una lista: se si sceglie la prima opzione, bisogna passare il file che rappresenta la notifica come oggetto `Metadata`, classe fornita dal SDK di Dropbox che rappresenta il path di un file nel cloud.
Se si preferisce la seconda modalità il costruttore si occupa di trovare automaticamente tutte le notifiche che devono ancora essere lette, e quindi ancora risolte, all'interno del folder *TemporarySharing* del chiamante.

Metodi

doNotification()

La funzione è invocata ogni volta che un utente si ritrova un file nel proprio *TemporarySharing* e si deve leggerne il contenuto per poi eseguire l'operazione adeguata: il suo compito è riassumibile in quello di controllare che tipo di notifica è stata ricevuta e invocare il corretto metodo per risolverla.

savePwdFolder()

Il metodo è chiamato quando la notifica è di tipo `PwdFolderShared`: il file è prelevato da Dropbox e decriptato con la propria chiave privata per poterne leggere il contenuto.
Viene successivamente fatto il mount del PwdFolder, in questo modo la cartella è inserita tra quelle raggiungibili dall'utente, vengono salvate le informazioni per accederci, cioè il nome del folder e la sua password, nel proprio *PersonalFolder*.

addNewMemberToPwdFolder()

Funzione eseguita quando la notifica è di tipo `AddMember`: inizialmente si controlla che il gruppo sia verificato e che gli utenti inseriti in un campo della notifica siano realmente membri del gruppo. Se i controlli vengono superati, viene prelevato dal file che rappresenta la classe `PwdFolder` l'oggetto `AccessLevel` che era stato assegnato al gruppo. Successivamente il tipo di accesso lo si applica anche ai nuovi membri. Infine viene inviata una notifica di tipo `PwdFolderShared` in modo che possano salvarsi le informazioni del `PwdFolder`.

```
PwdFolder pwdFolder1 = new PwdFolder(pwdFolder);
AccessLevel accessLevel = pwdFolder1.getGroupsPermissions()
    .getAccessLevelGroup(realGroup);
DropboxFunctions.addMembersToFolder(accessLevel, pwdFolder, new
    ArrayList<>(Collections.singletonList(user)));
pwdFolder1.shareToMember(realGroup, user);
```

removeOwnerGroupFromPwdFolder()

Metodo eseguito con notifiche di tipo `GroupDelete`: poiché questo implica che il gruppo inserito dentro la notifica è stato eliminato, è necessario controllare innanzitutto che effettivamente non esista più e che fosse tra quelli con cui il PwdFolder era stato condiviso.
Se i controlli sono superati si elimina dalla condivisione del folder quello che era l'ex owner del gruppo e si modifica il file JSON che rappresenta la classe `PwdFolder` in maniera adeguata alla eliminazione appena effettuata.

removeMemberFromPwdFolder()

La funzione viene invocata se la notifica ricevuta è di tipo `MemberRemove`. I diversi controlli che è necessario eseguire per autorizzare l'operazione sono i seguenti: il gruppo deve essere verificato, ovviamente deve essere tra quelli con cui il PwdFolder era stato condiviso, e di ogni utente si controlla che non

appartenga più al gruppo né che sia l'owner del PwdFolder stesso.
Come ultimo controllo si verifica che ogni `User` non faccia parte di altri gruppi che hanno ancora accesso al PwdFolder: questo perché anche se l'utente può decidere di condividere una cartella con un gruppo, e non con un utente preciso, l'implementazione consiste nella condivisione con tutti i suoi membri, e non con il gruppo inteso come entità a sé stante; perciò se un utente viene rimosso da uno ma fa ancora parte di gruppi con cui è stato condiviso il PwdFolder, deve mantenerne l'accesso.
Questo ultimo controllo è effettuato dal metodo `checkIfRemoveUser()`

```
if(pwdFolder1.checkIfRemoveUser(realGroup,user))
```

Se tutte queste condizioni sono rispettate, gli ex membri perdono la condivisione del PwdFolder e ricevono una notifica di tipo `PwdFolderDeleted` in modo che possano cancellare i dati di accesso che non potrebbero più utilizzare.

4.10 Classe Group

Un gruppo rappresenta l'unione di tanti `User`, e la classe è composta da quattro attributi:

`Name` la stringa che rappresenta il nome del gruppo

`Owner` l'`User` che è il capo del gruppo

`Verified` il booleano che rappresenta se il gruppo è verificato o meno

`Members` l'ArrayList di `User` che fanno parte del gruppo

Costruttore

Come per la classe `User` anche in questo caso si è cercato di adottare il pattern *Builder*: i parametri `Owner`, `Verified` e `Members` vengono settati successivamente, evitando così il download di file non necessari .

In particolare, per recuperare uno di qualsiasi dei tre parametri è necessario prima di tutto invocare il metodo `setAttributes()`. Inoltre per poter effettuare la `get` di`Owner` o di `Members` il valore dell'attributo `Verified` deve essere settato a `True` in modo da assicurarsi che il gruppo non sia stato alterato.

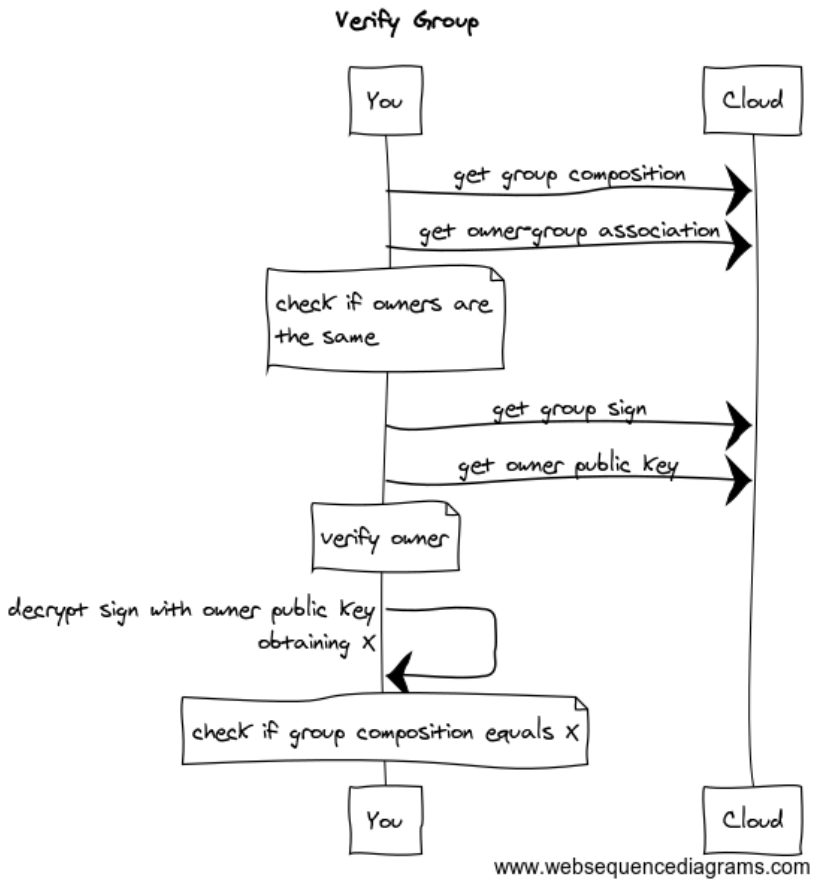
Metodi

setAttributes()

Prima di tutto è effettuato il download del file JSON che descrive la composizione del gruppo in modo da settare gli attributi `Owner` e `Members`.

```
this.owner=(new User((String) json.get("Owner")));
ArrayList<String> strings = (ArrayList<String>) json.get("Users");
this.members = new ArrayList<>();
for (String string : strings) {
   members.add(new User(string));
}
```

Successivamente è effettuato il download del file che ne rappresenta la firma e del JSON rappresentante l'associazione owner-gruppo creato dall'Admin. Una volta ottenuti questi file è possibile controllare che l'attributo`owner` dei due JSON combaci: in questo modo si è sicuri che non vi sia stata impersonazione. Dopodiché si ottiene la chiave pubblica del capogruppo, controllando prima che lui stesso sia verificato, e si decripta il file che rappresenta la sua firma sul gruppo. Se il file decriptato è uguale al JSON che rappresenta la composizione del gruppo, allora questo è verificato.
Uno schema che semplifica la comprensione della verifica del gruppo è il seguente:



shareToPwdFoldersOwner()

Metodo invocato sia dalla funzione che aggiunge membri al `Group` che dalla sua inversa, quella che ne rimuove. Lo scopo della funzione è duplice: prima di tutto rimuovere i file che rappresentavano l'oggetto e inserire i nuovi utilizzando la funzione `upload()`:

```
File file = ProjectHandler.jsonToFile(createJsonGroup(),
    ProjectHandler.values.VAL_GROUP);
DropboxFunctions.uploadPersonalFile(ProjectHandler.GROUPS_COMPOSITION,
    file, false, false);
File sign = sign(file.getCanonicalPath());
```

```
DropboxFunctions.uploadPersonalFile(ProjectHandler.SIGNED_GROUPS,
    sign, true, false);
```

Il secondo obiettivo è avvertire tutti gli owner di tutti i PwdFolder a cui il gruppo ha accesso che la composizione è cambiata e dovranno apportare delle modifiche ai loro `PwdFolder` attraverso la creazione di una notifica di tipo `Add` o `Remove` a seconda che sia stata chiamata rispettivamente dal metodo `addMember()` o `removeMember()`. La notifica sarà inserita nel folder del destinatario presso *TemporarySharing*. L'owner del PwdFolder gestirà poi automaticamente la richiesta inviata.

```
Cryptomator cryptomator = new Cryptomator();
for (String vault : vaults) {
   File file1 = cryptomator.copyFileFromVault("/" + this.name + "/"
       + vault, ProjectHandler.MY_TEMP_PATH + "/" + vault);
   String ownerVault = (String)
       ProjectHandler.fileToJson(file1).get("Owner");
   JSONObject jsonRemove = Notification.GroupChanged(
       vault.replace(ProjectHandler.END_SHARED_PWDFOLDER, ""),
       users, this);
   File file2 = ProjectHandler.jsonToFile(jsonRemove, val);
   DropboxFunctions.uploadPersonalFile(
       ProjectHandler.TEMPORARY_SHARING_FOLDER + "/" + ownerVault,
       file2, true, true);
}
cryptomator.exit();
```

delete()

È stata fatta la scelta di una rigida condizione per poter eseguire correttamente il metodo: il chiamante deve essere l'owner e unico membro del gruppo. Questo perché sarebbe complicato per gli owner dei PwdFolder recuperare la lista dei membri, per poi rimuoverne l'accesso, se il gruppo non

esiste più.
L'idea è la stessa presentata nel metodo `shareToPwdFoldersOwner()` solamente che i file JSON non devono essere aggiornati ma semplicemente cancellati, e la notifica che riceveranno gli owner dei PwdFolder è tipo `GroupDelete`.

4.11 Classe Admin

La classe estende `User`, avendo perciò tutti gli attributi della sua superclasse e, in aggiunta, la chiave privata che utilizzerà per firmare le chiavi pubbliche degli utenti.
Un'altra particolarità della classe è l'essere un *Singleton*: come per l'oggetto `yourself`, non c'è motivo che esistano più di un oggetto `Admin` istanziati.

```
public static Admin getAdmin() {
   if (admin == null) {
      admin=new Admin();
   }
   return admin;
}
```

Metodi

setup()

Il metodo è un override dell'omonimo della classe `User` ed è invocata ad ogni avvio di CryptoCloud. Se non è necessario creare la struttura, vengono elencati sia gli utenti che i gruppi in attesa della sua firma. L'operazione di firma non può essere eseguita in automatico, a differenza della risoluzione delle notifiche, in quanto, per sicurezza, bisogna che sia l'Admin a prendersi la responsabilità di firmare l'identità dell'utente.

```
ArrayList<User> usersToSign = toSignUser();
```

```
if (usersToSign.isEmpty()) {
   System.out.println("No users are waiting to be signed");
} else {
   System.out.println("These are the users waiting to be signed:");
   for (User user : usersToSign) {
      System.out.println(ProjectHandler.ConsoleColors.GREEN +
          user.getEmail() + ProjectHandler.ConsoleColors.RESET);
   }
}
```

Se invece è la prima volta in assoluto che si avvia CryptoCloud, verranno eseguite in sequenza le seguenti operazioni per la creazione della struttura di folder che gli utenti andranno poi ad utilizzare:

- Creazione del folder *System* e condivisione con tutti gli utenti che l'admin vorrà: essi saranno gli utenti che potranno partecipare all'applicazione.
- Creazione del folder *SignedKeys* e condivisione in lettura con tutti gli utenti prima decisi.
- Upload della chiave pubblica nel folder *SignedKey* sotto forma di `admin.publickey`.
- Creazione del folder *SignedGroupsOwner* e condivisione in lettura con tutti gli utenti prima decisi.
- Creazione del folder *GroupsComposition*.
- Creazione del folder *SigedGroups*.
- Creazione del folder *PublicKeys*.
- Creazione del folder *TemporarySharing*.

Che sia o meno il primo avvio, vengono effettuate ulteriormente due operazioni:

- Creazione della classe `DropboxPolling` su un nuovo thread che si occupa di ascoltare cambiamenti in *PublicKeys*: se un utente inserisce la propria chiave pubblica l'Admin deve poter essere avvisato e deve poterlo eventualmente firmare.

- Creazione della classe `DropboxPolling` su un nuovo thread che si occupa di ascoltare cambiamenti in *GroupsComposition*: se un gruppo viene creato, l'Admin deve poter essere avvisato e deve poter in caso approvare l'associazione owner-gruppo.

```
startPolling(ProjectHandler.values.VAL_ADMIN2);
startPolling(ProjectHandler.values.VAL_ADMIN1);
```

signUser()

Dopo aver elencato gli utenti che attendono la sua firma, e ciò sarà realizzato effettuando il confronto tra gli utenti con cui è stata condivisa la cartella *System* e quelli che hanno la firma in *SignedKeys*, si preleva la chiave pubblica dell'utente, lo si firma con la propria chiave privata e si inserisce il risultato in *SignedKeys*.

signGroup()

Il ragionamento è simile a quello presentato in *signUser()*: la differenza principale è che non è più una vera firma, bensì un file JSON contenente il nome del `Group` e quello del suo owner inserito nella cartella *SignedGroupsOwner*. In questo modo nessuno può modificare il file se non l'Admin stesso poiché solo lui ha i permessi di scrittura, ma chiunque può verificare il gruppo. Non è stato implementato lo stesso metodo utilizzato per firmare un `User` per un motivo ben preciso: la composizione del gruppo può modificarsi nel corso del tempo, la chiave pubblica dell'`User`, se non in casi veramente rari, non viene mai modificata. Per questo motivo se venisse creata una firma sarebbe necessaria ricrearla ad ogni alterazione della composizione del `Group`.

4.12 Classe GroupsPermissions

Classe interna a `PwdFolder` e utilizzata solamente da questa, avente come attributi due liste, una contenente i `Group` e l'altra il relativo permesso di ogni gruppo con il `PwdFolder` sotto forma di oggetto `AccessLevel`.

addGroup()

Metodo invocato da due funzioni, o dal costruttore della classe `PwdFolder` come `setter` dell'attributo `GroupsPermissions`, o dal metodo omonimo della classe `PwdFolder`.
La funzione prima di tutto crea un nuovo oggetto `GroupsPermissions`: questo perché si vorrà avere una lista dei `Group` aggiunti per poterne condividere poi le informazioni di accesso al PwdFolder. Viene permesso all'utente di scegliere tra l'elenco dei gruppi esistenti nel sistema purché non siano stati già aggiunti e che essi siano verificati da un Admin. Superata questa fase si può selezionare se fornire accesso di sola lettura oppure anche di scrittura.
Il valore di ritorno sarà l'insieme dei `Group` nuovi e il relativo `AccessLevel`

4.13 Classe PwdFolder

Un PwdFolder è una cartella criptata utilizzando `Cryptomator` del quale l'utente creatore può deciderne una password e con che gruppi condividerla.
Gli attributi sono i seguenti:

`Name` la stringa che rappresenta il nome

`Owner` l'`User` che ne è il proprietario

`Password` la stringa che rappresenta la password

`GroupsPermissions` un oggetto `GroupsPermissions` contenente le liste di`Group` e relativo `AccessLevel`

`PwdEntries` la lista delle `PwdEntry` interne al folder

Costruttore

La classe ha una particolarità rispetto a quelle presentate fino ad adesso: i valori degli attributi anche in questo caso ottenuti dal file che rappresenta l'oggetto inserito su Dropbox, ma non è scontata la locazione del JSON nel cloud né i parametri del file stesso. Infatti, se l'utente è owner del PwdFolder avrà il file `.pwdFolder` salvato nella propria cartella *MyPwdFolder*, mentre un utente che ne ha solamente ricevuto l'accesso, lo troverà dentro al folder del gruppo che ne ha permesso la condivisione. Non solo: l'owner è anche a conoscenza dei `Group` con cui ne ha condiviso l'accesso così come il tipo di `AccessLevel`.
Per questo motivo la struttura del costruttore del PwdFolder è suddivisibile in due parti: la prima consiste nel controllare se il chiamante è anche il suo owner e, per fare questo passo, si prova a recuperare le informazioni dal folder sopra citato.

```
File file =
    cryptomator.copyFileFromVault(Cryptomator.pathMyPwdFolder + "/"
+ pwdFolder + ProjectHandler.END_PWDFOLDER,
    ProjectHandler.MY_TEMP_PATH + "/"
+ pwdFolder + ProjectHandler.END_PWDFOLDER);
JSONObject json = ProjectHandler.fileToJson(file);
ProjectHandler.deleteLocalFile(file);
jsonToPwdFolder(json);
cryptomator.exit();
```

Se non è possibile recuperare i valori da impostare al `PwdFolder` in questo modo, significa che l'utente non è il suo owner, e si cercano le informazioni tra tutti i gruppi di cui l'utente è membro.

```
ArrayList<String> groups = Group.listMyGroupsAsArrayString();
for (String group : groups) {
```

```
    ArrayList<String> pwdFolders = listPwdFoldersInGroup(group);
    if (pwdFolders.contains(pwdFolder)) {
        cryptomator = new Cryptomator();
        File file = cryptomator.copyFileFromVault("/" + group + "/" +
            pwdFolder + ProjectHandler.END_SHARED_PWDFOLDER,
            ProjectHandler.MY_TEMP_PATH + "/" + name +
            ProjectHandler.END_SHARED_PWDFOLDER);

        JSONObject jsonObject = ProjectHandler.fileToJson(file);
        this.name = pwdFolder;
        this.password = (String) jsonObject.get("Password");
        this.owner = new User((String) jsonObject.get("Owner"));
        cryptomator.exit();
        break;
    }
}
```

Se l'oggetto è creato in questo ultimo modo l'attributo `GroupsPermissions` resterà vuoto in quanto un utente non potrà mai ottenere questa informazione.
Un altro costruttore è quello che prende come parametri `Owner`, `Nome` e `Password`: non ottiene però direttamente il valore dell'attributo `GroupsPermissions`. Questo perché quando si decide di utilizzare questo costruttore i nuovi `Gruoup` che si vogliono aggiungere verranno scelti dal chiamante. Questa modalità è infatti usata dal metodo `createPwdFolder()` della classe `User`.
Come in precedenza si è deciso di evitare di effettuare download non strettamente necessari nel costruttore: la lista di PwdEntry verrà settata solamente quando è invocata la *get* dell'attributo.

Metodi

addGroups()

Il metodo aggiunge nuovi gruppi alla classe modificando l'attributo `GroupsPermissions`. Il valore di ritorno sarà utilizzato per poi condividere le informazioni di accesso del PwdFolder solamente con i nuovi `Group` aggiunti, evitando così di inviare nuovamente notifiche a chi le aveva già salvate.

```
GroupsPermissions added=this.groupsPermissions.addGroups();
shareToGroups(added);
```

Viene inoltre rimpiazzato il file JSON che rappresentava l'oggetto aggiungendone uno che contiene pure i nuovi `Group` e il relativo `AccessLevel`. Infine invocando `shareToGroups()` vengono condivise le informazioni di accesso.

shareToMember()

Il metodo permette di condividere i dati di accesso di un PwdFolder con un `User` singolo: il primo passaggio consiste nel creare una notifica `PwdFolderShared` e successivamente in file per poi poterne fare l'upload dentro *TemporarySharing*.

```
File file = ProjectHandler.jsonToFile(
    Notification.PwdFolderShared(this, group.getName()),
    ProjectHandler.values.VAL_VAULT);
```

Poiché il file contiene dati sensibili, quale la password di accesso del PwdFolder, prima di poter essere inserito nella cartella viene cifrato utilizzando la chiave pubblica del destinatario

```
ProjectHandler.byteToFile(KeysFunctions.encrypt(
    ProjectHandler.fileToByte(file.getCanonicalPath()),
    user.getPublicKey()),
```

```
file.getCanonicalPath());
DropboxFunctions.uploadPersonalFile(
ProjectHandler.TEMPORARY_SHARING_FOLDER + "/" + user.getEmail(),
file, true, true);
```

unshare()

Il metodo può essere invocato con due valori diversi del parametro `type`: `DELETEPWDFOLDER` o *REMOVE*. Il primo indica che il PwdFolder è in fase di cancellazione e quindi è necessario togliere ogni utente dalla condivisione di questo, indipendentemente dal gruppo a cui apparteneva. Il secondo invece denota che soltanto ad alcuni gruppi è stata revocata la condivisione.
In quest'ultimo caso bisogna controllare se l'utente debba perdere veramente l'accesso al PwdFolder: se infatti fa parte anche di un altro gruppo il quale ha accesso al PwdFolder, l'utente deve poter continuare ad accedere alla cartella. Questo controllo è effettuato nel seguente modo dal metodo `checkIfRemoveUser()`:

```
if(group.getOwner().equals(user)){
   return false;
}
boolean flag = true;
for (Group group2 : getGroupsPermissions().getGroups()) {
   System.out.println(group2.getName());
   if (group2 != group) {
      if (group2.getMembers().contains(user)) {
         flag = false;
         break;
      }
   }
}
return flag;
```

4.14 Classe PwdEntry

La classe `PwdEntry` è un esempio di dato sensibile che deve essere protetto da occhi indiscreti e per questo motivo è protetto dentro ad un PwdFolder. Esso contiene diversi attributi:

`Name` il nome univoco che individua l'oggetto

`Author:` l'utente che ha per ultimo modificato la PwdEntry

`Date` la data di ultima modifica

`Username` informazione sensibile da proteggere

`Password` informazione sensibile da proteggere

`System` indica in quale sistema le credenziali debbano essere usate

`Cryptomator` l'oggetto `Cryptomator` utilizzato per accedere alla PwdEntry

Costruttore

Il costruttore è molto semplice: si seleziona la PwdEntry dall'elenco che viene mostrato in console e tutti i parametri vengono settati recuperando l'oggetto grazie alla funzione `getPwdEntry()` della classe `PwdFolder`.
In seguito il dettaglio della funzione `getPwdEntry()`

```
if (pwdEntries.contains(pwdEntry)) {
   Cryptomator cryptomator = new Cryptomator(Cryptomator.pathBase +
       "/" + name, password);
   File file = cryptomator.copyFileFromVault("/" + nameEntry +
       ProjectHandler.END_PWDENTRY,
   ProjectHandler.MY_TEMP_PATH + "/" + nameEntry +
       ProjectHandler.END_PWDENTRY);
   JSONObject json = ProjectHandler.fileToJson(file);
   ProjectHandler.deleteLocalFile(file);
   pwdEntry.fromJSON(json, cryptomator);
```

```
}
return pwdEntry;
```

È possibile notare che questa funzione è l'unica che non esegue il metodo `exit()` di `Cryptomator`: questo perché si occuperanno le funzioni della `PwdEntry` a invocarla.

Conclusioni

La tesi mostra che è possibile realizzare un sistema per conservare dati sensibili in cloud utilizzando tecnologie opensource che ne cifrano il contenuto, evitando di avere un rallentamento così elevato da rendere il software utile solo a livello teorico.
È stato realizzata una gestione di gruppi autonoma e non dipendente dal cloud provider scelto: per l'implementazione di CryptoCloud infatti è stato utilizzato Dropbox, ma l'idea di base è replicabile utilizzando un qualsiasi provider che proponga una SDK o eventualmente un'API, anche se potrebbe essere necessario dover eseguire un numero maggiori di modifiche al codice sorgente.
Ci possono essere ulteriori margini di miglioramento dell'applicazione: è possibile, ad esempio, l'integrazione delle notifiche con un servizio di E-mail in modo che l'utilizzatore possa essere sempre aggiornato, anche senza dover per forza avviare il programma.
Per servirsi di CryptoCloud a livello aziendale, è inoltre necessario che l'Admin o si fidi dei gestori del provider scelto per l'autenticità della sua chiave pubblica, oppure che consegni ad ogni utente una copia fisica della chiave per garantirne l'integrità: magari questa soluzione che prevede una interazione tra due persone potrà essere sostituita da una feature che risolve in maniera diversa ma con lo stesso risultato, e solamente via software, il problema.
Sarebbe inoltre utile effettuare un test con un numero maggiore di utenti di quello effettuato fin'ora per vedere la scalabilità del codice nel momento in cui si viene a creare una elevata concorrenza di operazioni dentro alla cartella

System.

Utilizzando CryptoCloud è anche possibile implementare una versione base di *Single sign-on* per l'azienda: un utente non avrebbe più bisogno di conoscere tutte le credenziali di ogni macchina, ma sarebbe sufficiente recuperare i dati di accesso salvati sotto forma di `PwdEntry` attraverso la funzione `getCredentials()`.

Il codice infine è molto dipendente dalle tecnologie utilizzate: se in futuro `Cryptomator` dovesse presentare dei bug in grado di compromettere la sua sicurezza, queste falle sarebbero presenti anche dentro a CryptoCloud.

Il sorgente presenterà quasi sicuramente bug e falle di sicurezza che non sono state ancora rilevate, in quanto purtroppo la totalità del sorgente è stata scritta, verificata e debuggata da solo una persona: questo è uno dei tanti motivi per cui il codice è stato pubblicato online, in modo da avere, si spera, anche opinioni di programmatori più esperti.

Ringraziamenti

A Teresa, la mia ragazza PERFETTA che oltre a consolarmi nei momenti di crisi e controllare la stesura della tesi, mi ha obbligato a redigerla al posto di passare il pomeriggio a giocare al computer.
Ringrazio Angelo Neri, responsabile dell'Area IT&DP del Cineca, che ha permesso la nascita dell'idea di CryptoCloud durante l'attività di tirocinio curricolare. Grazie al mio relatore, il professore Renzo Davoli, per avermi insegnato l'Arte della programmazione ed essere sempre stato una fonte inesauribile di conoscenza e consigli.
Ringrazio i miei genitori, che oltre a finanziare la mia cultura e correggermi la grammatica sono stati moderatamente bravi a non interrompere i miei flussi di pensieri. Sotto sotto vi voglio bene. Ai miei coinquilini, Fabio, Daniele e Radu, che oltre ad insegnarmi i congiuntivi sono riusciti pure a non avvelenarmi durante questa convivenza, cosa per cui li ringrazio profondamente e non del tutto scontata. Ammetto di non essere il miglior coinquilino del mondo. Ringrazio Marco, collega di università e di studio matto e disperato, grazie al quale è stato possibile superare in maniera ottima ogni esame che richiedeva un progetto.
Ringrazio Cristian, ormai unico compagno di videogame e di infinite risate, spesso a suo discapito. Ai Manzi, i miei gemelli preferiti, che hanno bloccato i miei attacchi di ira dovuti alla pubblicità di Spotify acquistandomi un account premium. Li posso chiamare quasi la mia seconda famiglia: non per altro condivido l'account family di Netflix con loro. A tutti i residenti dello studentato Irnerio, che in questo anno hanno sopportato la mia presenza e

le mie critiche su come si cucini correttamente la piada. Non si cuoce nella padella. Ringrazio inoltre tutti i miei amici di Rimini, molti dei quali li conosco da tutta la vita, che mi hanno sopportato per tutti questi anni e sono sempre stati al mio fianco nel momento del bisogno. A tutti voi, un grazie dal profondo del mio cuore.

Bibliografia

[1] Cryptomator, URL: *https://github.com/cryptomator/cryptomator*

[2] BoxCryptor, URL: *https://www.boxcryptor.com/it/technical-overview/*

[3] Sookasa, URL: *https://www.sookasa.com/security/*

[4] Odrive, URL: *https://www.odrive.com/features*

[5] Mega, URL: *https://mega.nz/sourcecode*

[6] pCloud, URL: *https://www.pcloud.com/it/features/security.html*

[7] Tresorit, URL: *https://tresorit.com/security*

[8] T. Bevis, *Java Design Pattern Essential*, Second Edition, 2012.

[9] B Eckel, *Thinking in Java*, Fourth Edition, 2006.

[10] D. Cameron, *Java 8: The Fundamentals*, First Edition, 2014.

[11] J. Bloch, *Effective Java*, Second Edition, 2008.

[12] CryptoFS, URL: *https://github.com/cryptomator/cryptofs*

[13] Dropbox SDK, URL: *https://github.com/dropbox/dropbox-sdk-java*

[14] Dropbox, URL: *https://www.dropbox.com/developers*

[15] BouncyCastle, URL: *https://www.bouncycastle.org/java.html*

[16] H. Dogra, S. Verma, N. Hubballi, M. Swarnkar, *Security service level agreement measurement in cloud: A proof of concept implementation*, IEEE International Conference on Advanced Networks and Telecommunications Systems, 2018

[17] A. R. Wani, Q. P. Rana, N. Pandey, *Cloud security architecture based on user authentication and symmetric key cryptographic techniques*, 6th International Conference on Reliability, Infocom Technologies and Optimization, 2017

[18] G. Himanshu, K. D. Afewou, *A trust model for security and privacy in cloud services*, 6th International Conference on Reliability, Infocom Technologies and Optimization, 2017

[19] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts (Author), *Refactoring: Improving the Design of Existing Code* , First Edition, 1999.