ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

CAMPUS DI CESENA
SCUOLA DI INGEGNERIA E ARCHITETTURA

Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

SVILUPPO DI GNOME-KEYSIGN: FLATPAK DELL'APPLICAZIONE E AGGIUNTA DI METODI DI TRASFERIMENTO CON MAGIC WORMHOLE E BLUETOOTH

Tesi in:
Ingegneria dei Sistemi Software Adattativi Complessi

Tesi di Laurea di:
LUDOVICO DE NITTIS

Relatore:
Prof. MIRKO VIROLI

ANNO ACCADEMICO 2017–2018
SESSIONE II

PAROLE CHIAVE

Software Development

Python

Open Source

OpenPGP

Google Summer of Code

"Normal people . . . believe that if it ain't broke, don't fix it. Engineers believe that if it ain't broke, it doesn't have enough features yet." - Scott Adams

Indice

Introduzione

Le email che inviamo e riceviamo tutti i giorni non sono quasi mai né criptate end-to-end né firmate dal mittente semplicemente perché lo standard SMTP (Simple Mail Transfer Protocol) non lo prevede.

Per questo motivo, per poter ottenere ad esempio confidenzialità e autenticazione nelle email, bisogna utilizzare un software ad un livello più alto che permetta di firmare o criptare le email prima che vengano spedite.

Lo strumento attualmente più diffuso per tale scopo è il Pretty Good Privacy [1]. Il suo funzionamento è basato sull'utilizzo della crittografia asimmetrica, ovvero sull'utilizzo di una coppia di chiavi di cui una pubblica che può essere distribuita e una privata che deve rimanere segreta. Per ottenere la confidenzialità in un messaggio bisogna criptarlo con la chiave pubblica del destinatario, e in seguito solo chi sarà in possesso della relativa chiave privata potrà decriptarlo. Invece se si vuole ottenere l'autenticazione bisogna firmare il messaggio con la propria chiave privata e, utilizzando la relativa chiave pubblica, sarà possibile certificarne l'autenticità.

Essendo necessarie le chiavi pubbliche dei mittenti o destinatari dei messaggi, gli utenti in qualche modo devono essere in grado di ottenerle e verificarne l'effettiva appartenenza alle persone fisiche o entità dichiarate nelle chiavi stesse.

Per questo scopo PGP prevede l'utilizzo di *key server* per poter ottenere le chiavi pubbliche di altri utenti. Inoltre per poter controllare la validità di una chiave il metodo standard utilizzato da PGP è tramite l'utilizzo del *Web of Trust*, ovvero una rete di fiducia in cui gli utenti, verificando personalmente le chiavi pubbliche delle altre persone, possono rilasciare una firma certificando la validità di una data chiave.

[1] Con Pretty Good Privacy e PGP si intendono i software che seguono lo standard OpenPGP

Il processo di ottenimento di una chiave pubblica e la successiva firma non è per nulla semplice e richiede anche una certa attenzione da parte degli utenti nell'esecuzione dei vari passaggi richiesti. A causa di questo PGP fino ad ora è stato utilizzato solo da una piccola fetta di informatici, perché anche per loro l'intero processo di firma si rileva essere troppo lungo e pieno di complicazioni. [12]

Dato che il funzionamento di PGP si basa sul *Web of trust*, è fondamentale riuscire ad avere una rete di firme più ampia possibile perché, se non si fosse in grado di verificare la validità delle chiavi pubbliche, PGP non sarebbe utilizzabile.

Per cercare di risolvere tale situazione nel 2014 iniziò lo sviluppo di GNOME-Keysign, un software per Linux il cui obiettivo era quello di rendere lo scambio e firma di chiavi quasi alla portata di tutti. Il suo funzionamento di base prevede la possibilità di utilizzare il programma in modalità mittente, scegliendo una propria chiave pubblica da inviare, o in modalità ricevente, ottenendo una chiave di qualcun altro e consentendo di firmarla.

Questa tesi si prefigge l'obiettivo di migliorare GNOME-Keysign con lo scopo di renderlo più completo e facile da utilizzare, cercando così di ottenere indirettamente un incremento del *Web of trust*.

Dopo aver analizzato nel dettaglio GNOME-Keysign si è giunti alla conclusione che il software avesse diversi punti critici che potessero essere perfezionati. Essi vanno dall'unica modalità di trasferimento utilizzata alla difficoltà per gli utenti finali di ottenere il programma stesso.

Per questo motivo durante il progetto di tesi si sono aggiunte a GNOME-Keysign due ulteriori modalità di trasferimento delle chiavi che vanno ad affiancare il già esistente server locale: Bluetooth e Magic Wormhole. Grazie a Bluetooth è ora possibile effettuare il trasferimento delle chiavi anche in assenza di connettività locale, infatti basterà che entrambi gli utenti abbiano il Bluetooth disponibile sui propri Computer per poter stabilire una connessione. Con Magic Wormhole invece è possibile inviare le proprie chiavi passando per la rete Internet, quindi effettuare un trasferimento anche nel caso in cui i due Computer fossero collegati ad una rete WiFi guest o addirittura da remoto.

Per migliorare la facilità di reperimento di Keysign si è provveduto a creare un package AUR per la distribuzione Arch Linux [2], si è inoltre con-

[2] `https://aur.archlinux.org/packages/gnome-keysign/`

tattato il maintainer Debian aiutandolo a risolvere un bug che non consentiva di avere l'ultima versione su Debian e Ubuntu e infine si è realizzato un manifest Flatpak, distribuendolo su Flathub [3], rendendo l'applicazione installabile in modo immediato su un ampio numero di distribuzioni. Flathub rende pubblico il quantitativo di download che le varie applicazioni ricevono. In questo modo è possibile avere una stima grossolana sull'utilizzo del software che si è sviluppato. Ai primi di Ottobre, dopo 4 mesi dalla sua pubblicazione, GNOME-Keysign su Flathub ha ricevuto un totale di circa 400 download.

Per finire, GNOME-Keysign prevedeva solo il trasferimento delle chiavi pubbliche e l'invio per email delle relative firme, lasciando così agli utenti l'onere di dover importare manualmente le signature ricevute. Con l'obiettivo di migliorare l'esperienza d'uso, si è sviluppata un'applicazione companion a Keysign il cui scopo è quello di permettere agli utenti di poter automatizzare anche il processo di importazione delle firme, richiedendo semplicemente un drag and drop dell'allegato delle email che vengono ricevute tramite Keysign.

Questo progetto di tesi è iniziato tramite il Google Summer of Code 2017[4] con la supervisione di Tobias Mueller e Andrei Macavei, i due maintainer dell'applicazione GNOME-Keysign.

Nel primo capitolo vengono presentati i concetti di base necessari per comprendere gli argomenti di questa tesi. Inizialmente viene fatta una panoramica sulla crittografia arrivando a spiegare il funzionamento di PGP con le sue relative problematiche di utilizzo. Inoltre viene presentato il metodo con cui comunemente i package su Linux vengono creati e gestiti, mostrando infine le nuove alternative self-contained come Flatpak.

Nel secondo capitolo si parla di GNOME-Keysign spiegando nel dettaglio il suo funzionamento e individuando i suoi possibili miglioramenti.

Nel terzo capitolo vengono presentati i requisiti di questo progetto di tesi con i relativi casi d'uso e test plan.

Nel quarto capitolo vengono presentate le tecnologie utilizzate per la realizzazione del progetto quali Bluetooth e Magic Wormhole.

[3] `https://flathub.org/apps/details/org.gnome.Keysign`

[4] `https://summerofcode.withgoogle.com/archive/2017/projects/6637442190802944/`

Nel quinto capitolo si analizza il problema facendo una panoramica sull'architettura logica, i possibili rischi e le scelte progettuali che sono state prese.

Nel sesto capitolo viene descritta la fase di realizzazione vera e propria parlando dell'implementazione. Al fine di mostrare più nel concreto il lavoro svolto, sono stati inseriti anche degli snippet di codice Python.

Nel settimo capitolo si descrivono i vari tipi di test che sono stati effettuati per validare l'implementazione del progetto che è stata realizzata.

Nell'ottavo capitolo vengono presentate le conclusioni sul lavoro svolto, analizzando inoltre anche i possibili sviluppi futuri.

Capitolo 1

Backgroud

1.1 Crittografia

La crittografia può essere definita come la pratica e lo studio di tecniche per rendere una comunicazione sicura anche in presenza di un avversario. [17]

Diversi aspetti della sicurezza sono alla base della crittografia moderna, come la confidenzialità, l'integrità, l'autenticazione e il non-repudiation. [15]

La confidenzialità è spesso la prima caratteristica a cui si pensa parlando di crittografia. Essa consiste nel rendere i dati illeggibili a chiunque non sia a conoscenza della chiave/tecnica per decifrarli.

L'integrità garantisce l'accuratezza e consistenza dei dati. Lo scopo è quello di salvare i dati esattamente come ci si sarebbe aspettato e quando, in un secondo momento, li si andrà a legge nuovamente, si deve essere in grado di verificare che tali dati non siano stati alterati ne da modifiche accidentali [1] ne da possibili attacchi esterni.

L'autenticazione è la proprietà che garantisce sia l'integrità che l'identità della sorgente dati. I metodi più comuni per assicurare che i messaggi, o dati, siano autentici sono tramite l'utilizzo di un *message authentication code* (MAC) o con la firma digitale.

La non-repudiation [2] è la situazione in cui l'autore di una dichiarazione/messaggio non può metterne in dubbio la sua paternità o validità. È inoltre opportuno far notare che non tutti i meccanismi di autenticazione implicano anche il non ripudio. Ad esempio se si utilizza un meccanismo

[1] per esempio corruzione durante la trasmissione o difetto hardware

[2] chiamato anche non ripudio

di autenticazione a chiave simmetrica condivisa, non è possibile fare una distinzione tra i due attori in gioco perché entrambi possono generare un messaggio identico utilizzando la stessa chiave condivisa. [15, p. 361]

Uno dei più antichi e conosciuti crittogrammi è il cifrario di Cesare, in cui ogni lettera di un messaggio veniva sostituita con una lettera che si trovava ad un numero fisso di posizioni successive a quella di partenza nell'alfabeto. [3] Successivamente, grazie anche all'avvento dei Computer, le tecniche di criptazione divennero nettamente più complesse e sicure aumentandone così anche la loro diffusione.

1.2 Pretty Good Privacy

Pretty Good Privacy (PGP) è un software sviluppato inizialmente da Phil Zimmermann nel 1991 che rende possibile l'utilizzo di autenticazione e privacy nella trasmissione di dati. [21]

PGP fu rilasciato come un software proprietario, ma le specifiche vennero standardizzate sotto il nome di OpenPGP nel Novembre 2007 con l'RFC 4880, il successore del vecchio RFC 2440 del 1998. [10]

Alla base del funzionamento di PGP c'è l'uso della criptazione a chiave pubblica, cioè quel tipo di crittografia che richiede la generazione di una coppia di chiavi: una privata e una pubblica. La peculiarità di tale crittografia è il fatto di non poter dedurre in alcun modo la chiave privata partendo da quella pubblica. Utilizzando la propria chiave privata è possibile firmare il messaggio, ovvero certificarne l'autenticità e l'integrità. Oppure utilizzando la chiave pubblica del destinatario si può criptare un messaggio e assicurarne la confidenzialità.

Per questo motivo quando si vuole criptare un messaggio bisogna possedere la chiave pubblica del destinatario ed essere ragionevolmente sicuri che essa appartenga effettivamente al legittimo proprietario. Per un corretto funzionamento di PGP è dunque fondamentale riuscire ad ottenere in qualche modo le chiavi pubbliche autentiche delle altre persone.

Nella figura 1.1 viene mostrata la procedura utilizzata da PGP quando si vuole criptare e decriptare dei dati. La criptazione utilizza sia la cifrazione a chiave pubblica che quella simmetrica, questo perché cifrare con una chiave simmetrica è più efficiente rispetto ad usare la chiave pubblica. Quindi il messaggio criptato sarà formato dai dati cifrati con una chiave simmetrica

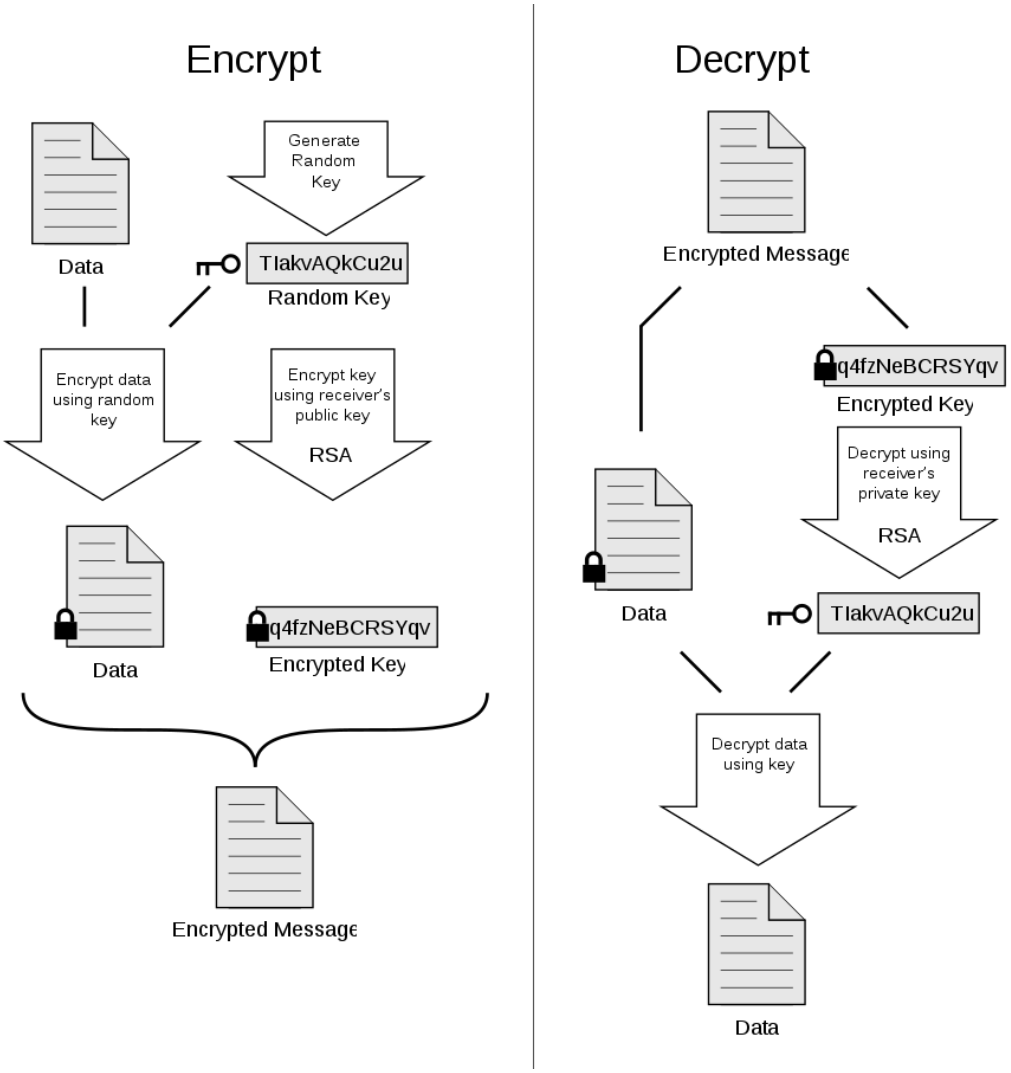

Figura 1.1: Funzionamento della criptazione e decriptazione con PGP. Immagine di xaedes & jfreax & Acdx / CC BY-SA

random generata sul momento e da tale chiave criptata a sua volta con la chiave pubblica del destinatario.

1.3 Web of Trust

Come menzionato in precedenza essere in grado di scambiarsi chiavi pubbliche e poterne accertare la loro autenticità è essenziale per il corretto funzionamento di PGP.

Effettuare lo scambio di persona è sicuramente il metodo più sicuro ma è anche molto scomodo da poter attuare.

Il metodo standard utilizzato da PGP per accertarsi che una data chiave pubblica appartenga effettivamente al presunto proprietario è tramite l'utilizzo del *Web of trust*, cioè di una rete di fiducia. Si tratta di un modello decentralizzato, opposto al modello centralizzato basato su *public key infrastructure* (PKI) in cui la fiducia viene basata esclusivamente su *Certificate Authority* (CA). Tutte le implementazioni conformi all'RFC 4880, lo standard OpenPGP, possono firmare digitalmente una chiave pubblica certificando di essere ragionevolmente sicuri che il legame chiave pubblica e persona fisica, elencati nel certificato, sia veritiero.

Quando si firma una chiave, OpenPGP permette di specificare il livello di fiducia che si vuole dare a tale firma, ovvero fino a che punto si è sicuri che tale chiave pubblica appartenga effettivamente alla presunta persona fisica. [1] I livelli possibili sono quattro:

- 0: Significa che non si fanno particolari asserzioni sul tipo di verifica effettuata. Questo è il livello che viene usato di default a meno che non venga specificato diversamente dall'utente.

- 1: Si usa quando si crede che la chiave appartenga effettivamente alla persona che è stata dichiarata ma non si è stati in grado di verificarlo. Questo livello è utile ad esempio quando si firmano chiavi appartenenti a pseudonimi.

- 2: È stata effettuata una verifica superficiale. Ad esempio quando si è solo verificato che il fingerprint della chiave fosse valido e si è confrontato l'user ID della chiave con un documento d'identità.

- 3: È stata effettuata una verifica approfondita. Ad esempio oltre ai passaggi del punto 2 si è anche verificato, scambiandosi email, che l'utente abbia effettivamente accesso all'indirizzo dichiarato nella chiave.

Gli utenti infine decideranno il numero minimo di firme, con i relativi livelli di fiducia, che riterranno necessari per potersi fidare di una data chiave pubblica.

La possibilità di trovare un collegamento tra due persone può essere giustificato dalla teoria denominata *del mondo piccolo*, ipotesi per cui ogni persona può essere collegata ad un'altra con una catena di amici di amici formata da un numero relativamente piccolo di collegamenti. [11] Comunque anche nel caso in cui si riesca a trovare un collegamento tra le due persone, il mittente deve fidarsi di ognuna delle persone della catena, cioè che ognuna sia onesta e che abbia seguito le linee guida per assicurarsi dell'identità del soggetto prima di rilasciare la firma. Quindi ad ogni step la fiducia complessiva che bisogna porre aumenta. L'unica soluzione possibile a tale problema è quella di aumentare il numero di firme presenti nel web of trust, rendendo in questo modo più facile la verifica sull'autenticità delle chiavi.

1.3.1 Key Server

Un *key server* è un sistema che riceve, per poi mettere a disposizione, chiavi crittografiche. I key server svolgono un ruolo centrale nella crittografia a chiave pubblica, visto che sono il punto di riferimento per il *bootsrapping*, ovvero sono utilizzati per ottenere una chiave pubblica di cui non si era precedentemente in possesso.

I key server sono solitamente gestiti da organizzazioni o individui che volontariamente mettono a disposizione server e banda Internet. Molti di questi server pubblici utilizzano il *Synchronizing Key Server* (SKS): key server il cui obiettivo è quello di permetterne una facile installazione, offrire un sistema decentralizzato dei server e garantire una sincronizzazione affidabile. [3] Questo per l'utente finale significa che basta inviare una propria chiave ad un singolo server SKS ed essa verrà automaticamente distribuita su tutti i restanti server.

[3] `https://bitbucket.org/skskeyserver/sks-keyserver/wiki/Home`

1.3.2 Caso d'uso pratico

Di seguito verrà presentato un esempio di un comune caso d'uso che può verificarsi quando due utenti utilizzano PGP:

Alice ha nel proprio keyring solo la chiave pubblica di Bob, già verificata e firmata. Un giorno Alice vuole inviare una email crittografata a Carol. Procede quindi cercando online in uno dei key server pubblici disponibili la chiave pubblica di Carol. Una volta trovata controlla le firme che tale chiave ha ricevuto e scopre che tra esse c'è anche la firma di Bob. Per questo motivo Alice, fidandosi di Bob, può essere ragionevolmente sicura che la chiave pubblica di Carol sia anche essa autentica.

Questo esempio evidenzia molto bene il fatto che il Web of Trust si basa, come dice appunto il nome, sulla fiducia. Se non ci fossero stati collegamenti tra le firme di Carol e Alice, quest'ultima non avrebbe avuto alcun modo di poter verificare l'autenticità di tale chiave pubblica.

1.4 Problematiche con l'utilizzo di PGP

1.4.1 Key server

Se si vuole ottenere una copia di una chiave pubblica, ad esempio perché si vuole contattare il destinatario crittografando le email oppure perché si vuole firmare la sua chiave pubblica, si può usare uno dei tanti *key server* pubblici disponibili.

Purtroppo però ci sono diversi problemi a cui si può andare in contro, tra cui:

- Leaks dei dati e MITM: i server possono essere contattati utilizzando HTTP, comportando una perdita di privacy, visto che un possibile *eavesdropper* potrebbe trivialmente leggere l'intera conversazione tra l'utente e il server essendo essa totalmente in chiaro. Inoltre un possibile *man-in-the-middle* potrebbe effettuare anche un attacco attivo e ad esempio sostituire la chiave richiesta dall'utente con una forgiata da lui sul momento.

- Alterare le informazioni: un server potrebbe essere esso stesso malevolo, magari a seguito di un attacco subito non ancora rilevato dagli amministratori del sistema. In questa ipotesi, anche se un utente

contattasse il server utilizzando TLS potrebbe venire in contro ad alterazioni della chiave richiesta come:

- Una completa sostituzione della chiave con una non veritiera
- Restituire la chiave senza alcuni degli UID [4] validi che conteneva
- Rimuovere dalla chiave alcune delle firme che aveva ricevuto, cercando in questo modo di screditarne l'autenticità
- Nel caso una chiave fosse stata revocata, tale informazione potrebbe essere omessa del server semplicemente evitando di inviare il certificato di revoca

1.4.2 Scambio e firma di chiavi pubbliche

Per firmare le chiavi pubbliche di altri utenti il metodo più utilizzato è tramite i *key signing party*. La procedura più comune (ad esempio utilizzata anche al FOSDEM [5]) prevede di:

1. Fare l'upload preventivamente della propria chiave su un *key server* dell'evento
2. Scaricare le fingerprint di tutti i partecipanti e stamparle su un foglio di carta
3. Recarsi al *key signing party* con un proprio documento di riconoscimento e il foglio stampato precedentemente
4. Disporsi su due file e scorrere di volta in volta verificando l'identità della persona di fronte e che il fingerprint della sua chiave coincida con quello stampato in proprio possesso
5. Una volta finito l'evento bisogna ottenere una copia autentica di ciascuna chiave
6. Firmare a inviare le chiavi utilizzando caff con un *mail transfer agent* (MTA)

[4] User IDs

[5] `https://archive.fosdem.org/2018/keysigning/`

Controllare i fingerprint di tutti i partecipanti ad un *key signing party* è un lavoro tedioso e molto lungo. Bisogna leggere a voce alta, ogni volta, il fingerprint della chiave che si vuole controllare, quindi 40 caratteri esadecimali ripetuti per il numero di persone presenti.

Successivamente per ottenere una copia autentica della chiave ci sono diversi aspetti a cui bisogna fare attenzione. Oltre a tutti i possibili problemi elencati nel paragrafo 1.4.1 è inoltre opportuno menzionare le *short key id*. Attualmente ci sono tre modi per riferirsi ad una chiave PGP:

- Short key ID: ultimi 8 caratteri esadecimali del fingerprint
- Long key ID: ultimi 16 caratteri esadecimali del fingerprint
- Intero fingerprint

Almeno dal 2011 è noto che usare uno *short key id* è insicuro perché è computazionalmente facile generare due chiavi GPG che abbiano gli stessi ultimi 8 caratteri. [14] Con una moderna scheda grafica è possibile trovare una collisione in appena 4 secondi. [13]

Purtroppo nonostante questo molte persone e organizzazioni ancora utilizzano le *short key id* rendendo molto difficile riuscire ad accorgersi se una chiave scaricata sia effettivamente quella corretta. [5] [19]

Infine il software comunemente utilizzato per automatizzare la firma delle chiavi si chiama caff[6]. Software che necessita di una configurazione tramite file testuale con sintassi PERL e della presenza di un *mail transfer agent* funzionante sul sistema in uso. Questi prerequisiti però rendono l'utilizzo di caff fuori dalla portata di molte persone.

1.5 Packaging tradizionale di software su Linux

Su Linux per gestire i packages, ovvero i software archiviati per una determinata tipologia di distribuzione, vengono utilizzati i così detti *package manager* come ad esempio *apt* o *pacman*. Essi consentono di automatizzare

[6]caff sta per "CA - Fire and Forget". `https://salsa.debian.org/stappers/pgp-tools/tree/master/caff`

il processo di installazione, aggiornamento, configurazione e disinstallazione rendendo l'intera procedura consistente.

La quasi totalità dei software più utilizzati sono presenti nei *software repository* della specifica distribuzione in utilizzo, permettendo agli utenti di installare nuove applicazioni senza dover effettuare ricerche manuali su Internet. I repository, sotto questo punto di vista, assomigliano ai vari store disponibili nei sistemi operativi mobile, infatti essi sono il luogo principale, se non unico, in cui si va a controllare per installare nuovo software.

Se si vuole includere la propria applicazione nei repository delle varie distribuzioni Linux si deve seguire una procedura che si diversifica da distribuzione a distribuzione. Ad esempio per Debian bisogna fare richiesta aprendo un bug report per la così detta *Request for Package (RFP)*. [6]

Oppure su Arch Linux la procedura standard prevede la creazione di un package description chiamato *PKGBUILD* che consente di compilare il proprio software per Arch. Tale package description deve essere poi inserito nell'*Arch User Repository (AUR)*, cioè un repository gestito dagli utenti e non dai developer di Arch. Per poter essere incluso nei repository ufficiali il programma deve essere utilizzato da un numero considerevole di utenti e un developer di Arch deve esprimere la propria volontà nel migrare e gestire tale software. [2]

1.6 Flatpak

La più valida alternativa all'impacchettamento manuale per ogni distribuzione è tramite l'utilizzo di sistemi con *self-contained packages* come Flatpak, Snappy o AppImage. Flatpak è relativamente recente, con la prima versione rilasciata nel 2015 e solo in Agosto 2018 si è arrivati alla versione 1.0 dichiarando che la release si potesse considerare *"feature complete"*. [8]

Flatpak rispetto ai metodi di distribuzione tradizionali offre diversi vantaggi come:

- *Build for every distro*: creando una applicazione con Flatpak sarà possibile distribuirla sull'intero Linux Desktop market. Al momento della stesura di questa tesi, Agosto 2018, Flatpak funziona out of the box, o con una minima configurazione, su 16 distribuzioni Linux. [7]

- Ambiente consistente: quando si testa e sviluppa la propria applicazione si avrà a disposizione un ambiente identico a quello degli utenti finali.

- future-proof build: le applicazioni create con Flatpak rimarranno compatibili anche in futuro con le nuove release delle distribuzioni Linux. Questo grazie al fatto che Flatpak richiede la compilazione di un manifest in cui vengono elencate le dipendenze necessarie con le relative versioni e quindi l'ambiente di funzionamento dell'applicazione rimarrà sempre invariato.

Per contro gli svantaggi di Flatpak sono:

- Maggiore spazio richiesto: quando si installano applicazioni utilizzando Flatpak lo spazio di archiviazione utilizzato sarà maggiore rispetto ad installare la stessa applicazione utilizzando il package manager della distribuzione in uso.

- Numero di software disponibili limitato: nei repository delle varie distribuzioni Linux di solito sono presenti diverse migliaia di applicazioni, mentre a Settembre 2018 Flathub, sito e repository [7] per distribuire le applicazioni Flatpak, ne conta solo circa 400 [8].

1.6.1 Concetti base

In figura 1.2 è mostrata la struttura con cui le varie applicazioni sono organizzate quando vengono installate utilizzando Flatpak.

1.6.1.1 Runtime

I runtime offrono una serie di dipendenze pronte all'uso. Il loro scopo è quello di rendere più facile la gestione delle dipendenze per una applicazione, visto che i runtime contengono già tutte quelle più comuni, e di offrire una base stabile su cui far girare le applicazioni, non dipendendo da alcuna particolare versione delle distribuzioni.

[7] `https://flathub.org/home` - Creato dagli stessi developer di Flatpak

[8] `https://web.archive.org/web/20180909183648/https://ahayzen.com/direct/flathub.html`

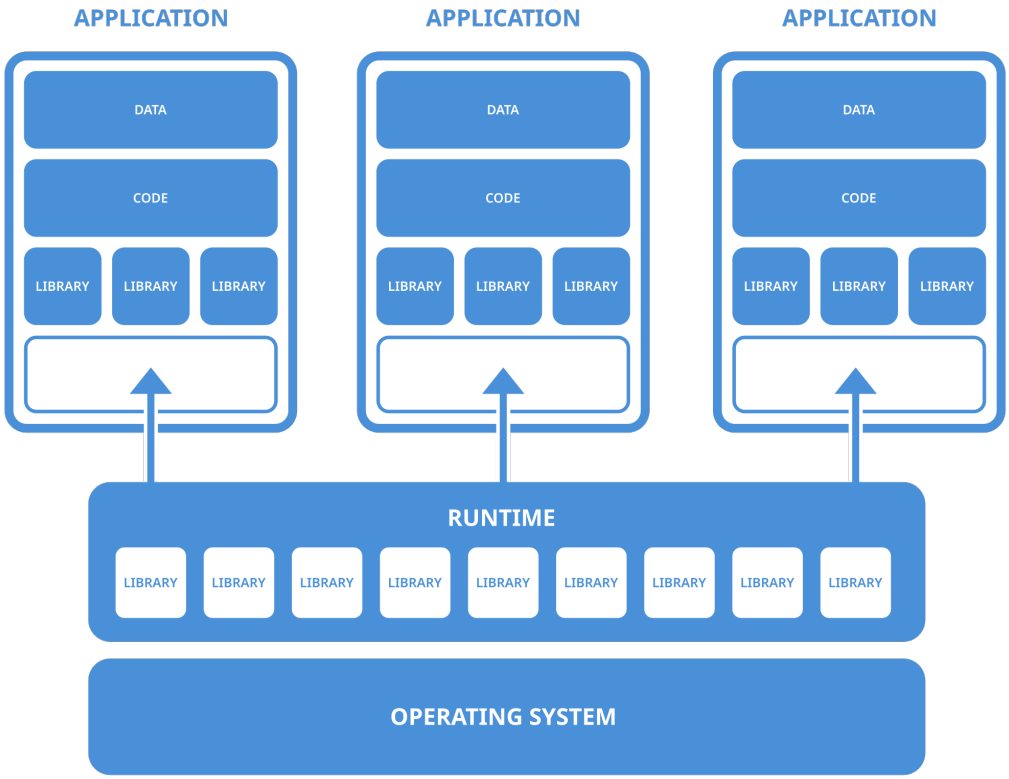

Figura 1.2: Diagramma di Flatpak. Immagine di Flatpak Team / CC-BY 4.0

I runtime devono essere installati sul sistema host su cui si vuole eseguire una determinata applicazione e diversi runtime possono essere installati allo stesso tempo.

1.6.1.2 Bundled libraries

Se una applicazione ha bisogno di dipendenze che non siano presenti nei runtime, è possibile inserirle manualmente dichiarandole, con il relativo numero di versione, al momento della creazione del Flatpak. Questa possibilità garantisce di poter utilizzare qualsiasi tipo di librerie, anche se ad esempio non fossero disponibili nemmeno nei repository della distribuzione in uso.

1.6.1.3 Sandbox

Con Flatpak ogni applicazione viene eseguita in una *sandbox*, ovvero in un ambiente isolato dal sistema host. Le applicazioni di default possono solo accedere al contenuto della sandbox. Nel caso in cui, per il corretto funzionamento, sia necessario che un software acceda ad esempio a determinate cartelle sul file system o che utilizzi la rete bisogna garantire esplicitamente tali permessi nel momento della creazione del Flatpak.

1.6.1.4 Portali

I portali sono un meccanismo per permettere alle applicazioni di compiere determinate azioni all'esterno della sandbox senza la necessità di dichiararne il permesso. Per non rendere vana la funzione della sandbox, i portali funzionano solo quando è l'utente che, interagendo con l'applicazione, permette l'accesso a funzioni dell'ambiente host come ad esempio garantire la lettura di un file tramite un *file chooser dialog* in cui è l'utente che selezione il singolo elemento a cui la sandbox potrà accedere.

1.6.2 Manifest

Ogni applicazione Flatpak richiede la creazione di *manifest*, ovvero un file di testo in JSON o YAML in cui vengono elencati i parametri richiesti per la compilazione dell'applicazione.

Ad esempio il file YAML postato qui sotto è un semplice manifest per uno script bash chiamato *hello.sh*.

Per le applicazioni reali di solito i manifest sono più complessi, ad esempio è comune trovare un elenco dei permessi richiesi o delle varie dipendenze necessarie non incluse nel runtime.

```
#!/bin/sh
echo "Hello world, this is a Flatpak example"
```

hello.sh

```
app-id: org.flatpak.Hello
runtime: org.freedesktop.Platform
runtime-version: '1.6'
sdk: org.freedesktop.Sdk
```

```
command: hello.sh
modules:
  - name: hello
    buildsystem: simple
    build-commands:
      - install -D hello.sh /app/bin/hello.sh
    sources:
      - type: file
        path: hello.sh
```

org.flatpak.hello.yaml

1.6.3 Alternative simili

Le alternative più diffuse simili a Flatpak attualmente sono:

- Snappy: software sviluppato da Canonical. Quando Flatpak si concentra più che altro su applicazioni Desktop, Snappy ha come obiettivo quello di incorporare anche tool da linea di comando e servizi che girano in backgroud. Il lato negativo di Snappy è il fatto di essere legato ancora molto ad Ubuntu, rendendo la sua installazione e configurazione non semplice sulle altre distribuzioni Linux. [20]

- AppImage: software il cui sviluppo iniziò nel lontano 2004 sotto il nome di *klik*. Rispetto a Flatpak e Snappy non richiede alcun tipo di configurazione ne software aggiuntivo sul sistema host. Le applicazioni sono distribuite come immagini montabili e direttamente eseguibili senza bisogno di installazione. Su AppImage l'utilizzo della sandbox è optionale e non obbligatoria.

1.7 Google Summer of Code

Il *Google Summer of Code* (GSoC) è un evento internazionale con cadenza annuale iniziato nel 2005 in cui Google premia gli studenti universitari che riescono a completare un dato progetto open source durante il periodo estivo. I fondatori di Google, Sergey Brin e Larry Page, idearono questa iniziativa con lo scopo di avvicinare gli studenti allo sviluppo in open source.

[4] I partecipanti sono affiancati da un mentor che li introduce alla community open source e li segue nello sviluppo del proprio progetto fornendo giudizi e commenti durante tutta la durata dell'evento.

Chris DiBona, open source program manager a Google, ha dichiarato che il *summer of code* fu progettato in modo da far trarre beneficio a tutti i partecipanti. Agli studenti sarà data la possibilità di lavorare a progetti reali, invece che accademici, ricevere una borsa di studio e fare conoscenze con persone del settore. Le organizzazioni invece avranno nuovo codice e funzionalità nei loro progetti e una chance di ottenere nuovi developer.

Figura 1.3: Logo ufficiale del Google Summer of Code. Immagine di Google / CC BY 3.0

Gli studenti universitari di tutto il mondo possono inviare fino a 3 richieste di partecipazione con i dettagli sul progetto a cui intendono lavorare. Le domande vengono valutate da parte dei vari mentor assegnati da ogni organizzazione accettata in questa iniziativa. I mentor valutano tutte le proposte ricevute per poter poi decidere quali progetti, con i relativi studenti, accettare. Le organizzazioni non possono comunque eccedere il numero massimo di slot che Google assegna ad ognuna di esse.

L'evento prevede una prima fase chiamata *community bonding* dalla durata di un mese in cui gli studenti prendono confidenza con l'organizzazione open source con cui dovranno lavorare. Inoltre questo è un periodo in cui

ci si aspetta che gli studenti studino il codebase con cui dovranno lavorare e le best practices del progetto. In seguito ci sarà la fase di sviluppo vera e propria dalla durata di 12 settimane.

Nel 2017 le organizzazioni accettate furono 201 e gli studenti che si registrarono al programma furono 20.651 provenienti da 144 Paesi. Dopo la prima fase di selezione 1.318 proposte di studenti, appartenenti a 575 Università diverse, furono accettate. [18]

Capitolo 2

Caso di Studio: GNOME-Keysign

2.1 Introduzione

GNOME-Keysign è un'applicazione Linux che nasce con l'obiettivo di migliorare il processo necessario per firmare chiavi OpenPGP rendendolo il più semplice e intuitivo possibile. Lo sviluppo iniziò nel 2014 con il rilascio della prima versione 0.1 nel Novembre dello stesso anno.

Il programma ha una grafica minimale con due tab tra cui scegliere: invio e ricezione. Su "invio" viene mostrata la lista delle chiavi private disponibili nel PC su cui è in esecuzione. 2.1 Selezionando una di queste chiavi la si potrà inviare utilizzando la rete LAN.

Per l'invio verrà mostrato il fingerprint della chiave e un QR code. 2.2

Lato ricezione l'utente potrà scansionare il QR code con la webcam del proprio PC o in alternativa inserire manualmente il fingerprint. 2.3 Quest'ultima possibilità è considerata di fallback, nel caso in cui l'utente per esempio non abbia una webcam funzionante da utilizzare.

Il trasferimento della chiave avverrà utilizzando la rete LAN e, una volta ricevuta, *Keysign* provvederà a firmarla. In seguito il client email dell'utente verrà automaticamente aperto con la possibilità di inviare le signatures generate direttamente al mittente.

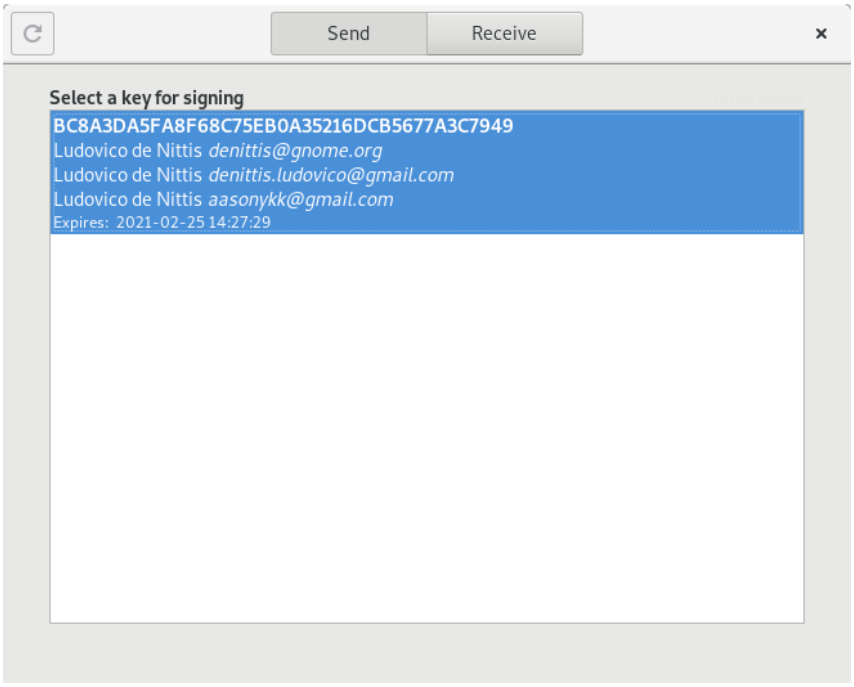

Figura 2.1: Pagina principale di GNOME-Keysign in cui è possibile scegliere la propria chiave da inviare.

Figura 2.2: Schermata mostrata dopo aver scelto la chiave che si intende inviare.

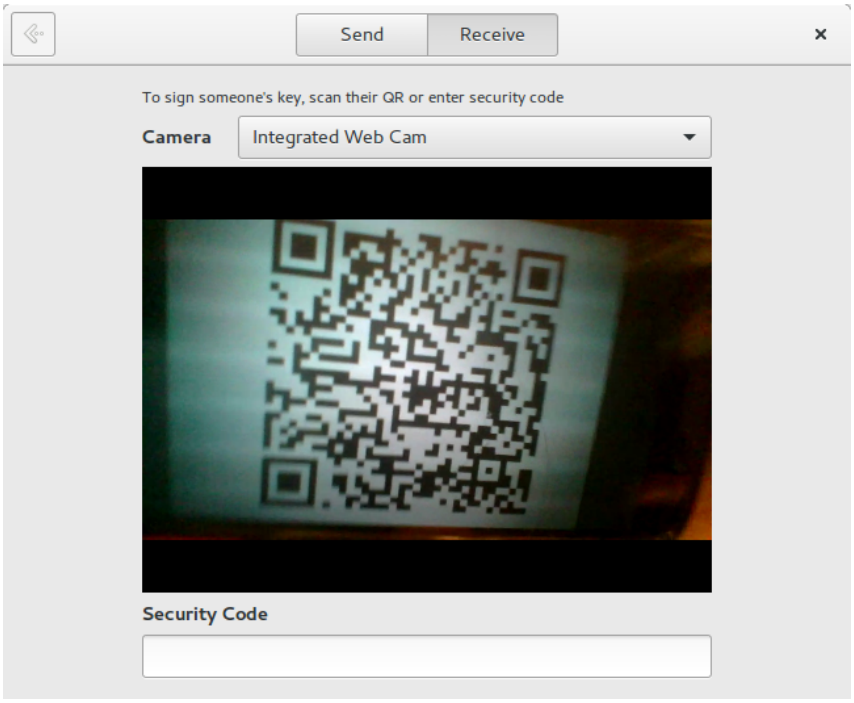

Figura 2.3: Pagina di ricezione di una chiave in cui l'utente sta scansionando un QR code con la webcam.

2.2 Funzionamento nel dettaglio

Le due modalità di esecuzione possibili sono:

- Server (invio): la modalità server consente di inviare una propria chiave pubblica per permettere ad un altro utente di firmarla. All'apertura l'applicazione ottiene la lista delle chiavi private attualmente presenti sul sistema e le mostrerà all'utente. Selezionandone una si procederà alla pagina successiva in cui sarà possibile visualizzare le informazioni sulla chiave scelta. Inoltre un server locale HTTP verrà attivato, utilizzando *Avahi*, per consentire ad altri utenti di poter scaricare la chiave pubblica relativa alla privata selezionata. Nella pagina dei dettagli sulla chiave sarà mostrato un *codice di sicurezza* in forma testuale e un QR code. Condividendo uno di questi due codici l'altra parte sarà in grado di ottenere e firmare tale chiave pubblica.

- Client (ricezione): la modalità client consente di firmare le chiavi pubbliche di qualcun altro. In questa modalità viene mostrato lo stream video dalla propria webcam, con cui è possibile scansionare il QR code dell'altra persona, e inoltre si ha a disposizione un campo in cui è possibile inserire manualmente il *codice di sicurezza*. Dopo aver scansionato un QR code, o aver inserito il codice a mano tramite tastiera, l'applicazione controllerà nella rete locale se fosse presente un'altra istanza di *keysign* in modalità server che sta offrendo la chiave che si sta cercando. Nel caso in cui fosse disponibile la chiave corretta, essa verrà scaricata e all'utente saranno mostrati i dettagli di tale chiave. Una volta che l'utente ha effettuato un ultimo controllo, per verificare che la chiave sia effettivamente quella desiderata, può proseguire premendo il pulsante *continua*. A questo punto il programma firmerà la chiave e aprirà il client email primario del sistema con una mail precompilata destinata all'indirizzo specificato nella chiave e con in allegato la firma.

2.2.1 Controllo di autenticità

Quando si sceglie di inviare una propria chiave pubblica, nel QR code che viene generato sarà presente oltre al codice di sicurezza [1] viene generato

[1] Il codice di sicurezza corrisponde al fingerprint completo della chiave

anche un *hash-based message authentication code* (HMAC). Tale HMAC viene calcolato utilizzando il fingerprint e i dati della chiave pubblica.

Questa ulteriore misura di sicurezza è utilizzata per prevenire un possibile attacco *man in the middle* con il fine di sostituire o alterare la chiave pubblica nel momento del trasferimento.

Nel caso in cui gli utenti non utilizzino il QR code ma digitino a mano il codice di sicurezza, il controllo tramite HMAC verrà ignorato e sarà eseguito solo il controllo sul fingerprint. Questo è un compromesso per permettere agli utenti un sistema di fallback alternativo al QR code ma che al tempo stesso che non richieda di inserire manualmente codici troppo lunghi.

2.2.2 Trasferimento firma via email

Alla fine del processo di firma la signature creata dal ricevente verrà trasferita al mittente tramite una email. Questo avviene aprendo in automatico il proprio client email predefinito con la funzione *xdg-email* offerta da linux.

La scelta di inviare la firma tramite email, invece che ad esempio utilizzando lo stesso canale con cui si è ricevuta la chiave pubblica, è stata presa perché solo così sarà inoltre possibile verificare che l'utente sia effettivamente in possesso dell'indirizzo email dichiarato sulla chiave PGP. Infatti nel caso in cui un utente si voglia far firmare una chiave con un indirizzo di cui lui non abbia accesso, quando la firma verrà recapitata lui non sarà in grado di reperire tale email e quindi la firma.

Se una chiave pubblica ha diversi UID associati, *Keysign* li firmerà separatamente creando tante email quanti sono gli UID della chiave.

2.3 Metodi per ottenere Keysign

Per ottenere GNOME-Keysign i metodi disponibili sono:

- Scaricare i sorgenti da Github o Gitlab e installare tutte le dipendenze richieste prima di eseguire il programma.

- Se si utilizza Debian Sid (unstable) come distribuzione si può installare direttamente dal package manager *apt* visto che l'ultima versione è presente nei repository

- Con Ubuntu 17.04 nei repository è disponibile la vecchia versione 0.6 quando attualmente l'ultima è la 0.9

Questa situazione rende complicato per un utente poter installare *Keysign* sul proprio sistema visto che, a parte per Debian Sid, l'unico metodo per ottenere l'ultima versione è quella di dover scaricare i sorgenti e controllare manualmente tutte le dipendenze richieste prima di poter utilizzarlo.

2.4 Possibile miglioramento

Avendo fatto una panoramica sullo stato attuale di *GNOME-Keysign* è possibile valutarne dei possibili miglioramenti che renderebbero il programma più completo e aumenterebbero la sua facilità di utilizzo.

- Una delle maggiori limitazioni è il singolo sistema utilizzato per il trasferimento delle chiavi, ovvero l'utilizzo di un server locale in LAN. Purtroppo, a parte per le reti domestiche, è pratica comune isolare i dispositivi connessi con le comunemente chiamate *reti guest*. Questo significa che, anche se si connettono due computer ad una stessa rete come quella di una università o di un bar, i dispositivi saranno in grado di navigare su internet ma non di comunicare localmente tra di loro, rendendo impossibile il trasferimento delle chiavi tra due istanze di *GNOME-Keysign*. Per questo motivo l'aggiunta di metodi di trasferimento alternativi al server locale *avahi* potrebbero incrementare i casi di utilizzo del programma stesso.

- *Keysign* pone molta enfasi sull'utilizzo del QR code per poter inizializzare il trasferimento, con l'utilizzo del campo per l'inserimento manuale del codice di sicurezza solo come una alternativa di fallback. Resta il fatto che i casi in cui il ricevente non sia in grado di utilizzare il QR code non sono così rari. Infatti se ad esempio si sta utilizzando un computer Desktop la probabilità di non avere a disposizione una webcam sono molto alte. Il risvolto negativo nell'utilizzare il codice di sicurezza è che esso è composto da 40 caratteri esadecimali. Dettare e trascrivere tale codice è una operazione lunga e soggetta anche a possibili errori di battitura. Non potendo forzare gli utenti ad utilizzare esclusivamente i QR code, l'alternativa sarebbe cercare di

rendere i codici di sicurezza più corti e semplici da trascrivere senza compromettere però la sicurezza del trasferimento stesso.

- Attualmente il bacino di utenti che si riesce a raggiungere è abbastanza limitato a causa della difficoltà che essi incontrano cerando di ottenere *GNOME-Keysign*. Come scritto in 2.3 solo con Debian Sid è possibile ricevere l'ultima versione del software utilizzando il package manager. Una possibile soluzione è quella di contattare i vari maintainer per le principali distribuzioni convincendoli a inserire *Keysign* nei loro repository ufficiali, magari aiutandoli con la compilazione del pacchetto software necessario. In alternativa si potrebbe optare per la creazione di un manifest *Flatpak* per garantire l'installazione del programma in qualsiasi distribuzione Linux.

- Il processo gestito da *GNOME-Keysign* si ferma con la ricezione della firma tramite email. A questo punto è lasciato all'utente il compito di importare tale signature nel proprio keyring personale. Per completare il processo di firma, *Keysign* potrebbe prevedere anche la possibilità di scegliere una signature dal proprio sistema e provvedere lui stesso ad importarla nel keyring.

Capitolo 3

Analisi

3.1 Visione

Considerando le diverse difficoltà, elencate nella sezione 1.4, che gli utenti devono affrontare per poter utilizzare PGP, questa tesi si pone l'obiettivo di migliorare la situazione attuale andando a rendere GNOME-Keysign più facile da utilizzare e più completo dal punto di vista delle features.

3.2 Requisiti

3.2.1 Requisiti Funzionali

- Aggiungere un metodo di trasferimento alternativo che consenta di far comunicare due istanze di Keysign anche se collegate ad una rete guest.

- Aggiungere un metodo di trasferimento aggiuntivo che consenta di far comunicare due istanze situate fisicamente a breve distanza anche non avendo alcuna connessione a reti LAN/Internet.

- Aumentare i canali con cui gli utenti possono ottenere l'applicazione.

- Consentire di importare nel proprio keyring le firme ricevute.

3.2.2 Requisiti non Funzionali

- Aggiungere i metodi di trasferimento aggiuntivi modificando la UI solo se strettamente necessario.
- Evitare di porre l'utente davanti a scelte non necessarie, come il chiedere quale metodo di trasporto utilizzare per il download.
- Aggiungere il minor numero possibile di elementi e passaggi richiesti, mantenendo quindi il più possibile lo stile minimale esistente. Ogni elemento aggiuntivo incrementa la complessità del programma sia dal punto di vista dell'usabilità che della mantenibilità.
- Informare l'utente durante tutta la durata del processo di trasferimento.

3.3 Analisi dei requisiti

3.3.1 Glossario

- GNOME-Keysign (Keysign): software che consente a due utenti di firmare le proprie chiavi OpenPGP.
- Utente: persona che interagisce con il sistema.
- Mittente: utente con una istanza di *Keysign* avviata in modalità server.
- Destinatario: utente con una istanza di *Keysign* avviata in modalità client.
- Chiave PGP/OpenPGP: coppia di chiavi di cui una pubblica e una privata, appartenenti ad un utente.
- Firma/signature: file criptato da un utente con la propria chiave privata che certifica la chiave pubblica di un altro utente.
- Metodo di trasferimento: protocollo o libreria che consente di trasmettere dati tra due parti
- Keyring: raccolta locale di chiavi con le relative firme.

- Rete guest: detta anche rete isolata, ovvero che non consente a due dispositivi di comunicare direttamente tra loro.

3.3.2 Casi D'uso

- Alice e Bob si incontrano all'Università e decidono di scambiarsi gli indirizzi email con le relative chiavi OpenPGP, firmandole inoltre l'un l'altro. Entrambi accendono i propri computer portatili, si connettono alla rete WiFi ed avviano GNOME-Keysign. Prima Alice invia la propria chiave a Bob e successivamente Bob farà lo stesso con Alice. Anche se la rete WiFi universitaria non permette una comunicazione diretta tra due computer, Keysign deve riuscire a stabilire una comunicazione sicura ed effettuare il trasferimento.

- Alice e Bob si incontrano al parco e decidono di scambiarsi gli indirizzi email con le relative chiavi OpenPGP firmandole inoltre l'un l'altro. Entrambi accendono i propri computer portatili ed avviano GNOME-Keysign. Anche se non è presente una connessione ad Internet o LAN, Keysign deve riuscire a stabilire una connessione sicura senza fili ed effettuare il trasferimento. Quando Alice e Bob torneranno a casa potranno inviare l'email con la firma generata al parco.

- Alice e Bob una volta utilizzato GNOME-Keysign e inviato le firme generate tramite email vogliono poterle importare in modo semplice e intuitivo senza dover utilizzare il terminale.

3.3.3 Test Plan

I test servono ad analizzare il software per cercare gli errori presenti e permettere agli sviluppatori di correggerli. Inoltre sono utilizzati anche come parametro per dare valore alla qualità del codice che si è sviluppato.

Si prevede di scrivere e validare i test appena possibile in modo da poter verificare subito la correttezza del codice che si andrà a scrivere e cercando così di evitare bug nelle fasi iniziali che potrebbero rivelarsi complicati da risolvere se si trovassero solo in una fase di sviluppo avanzato.

Gli unit test permettono inoltre di facilitare i cambiamenti come ad esempio le rifattorizzazioni del codice. E i test possono essere considerati

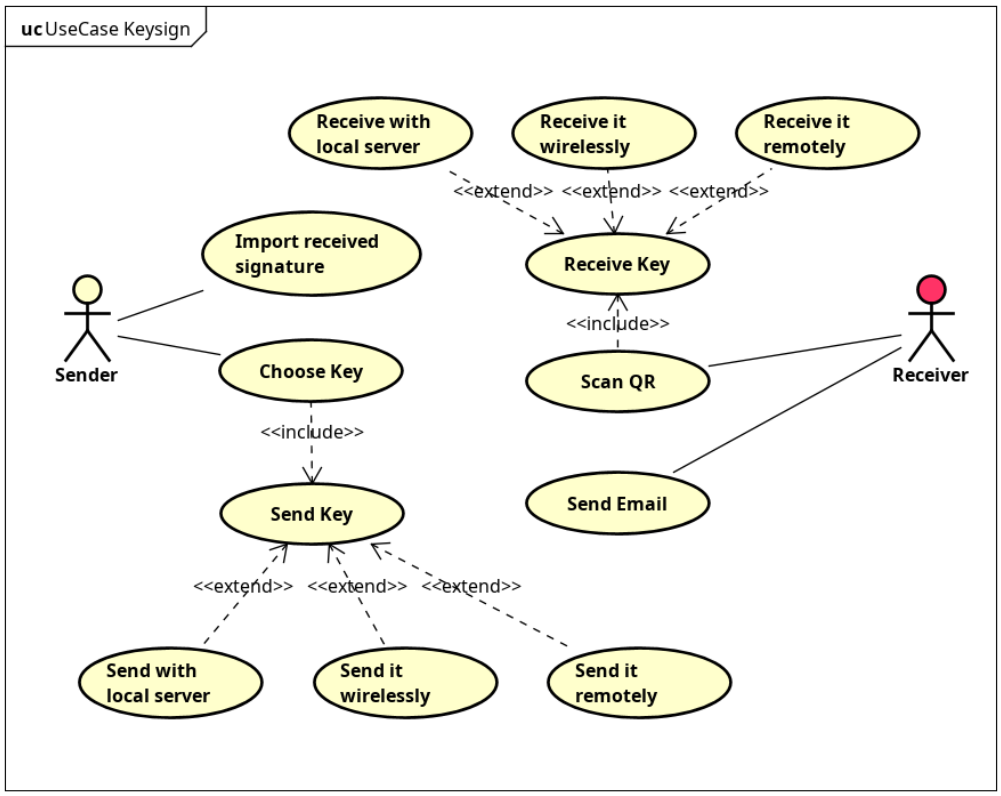

Figura 3.1: Diagramma dei casi d'uso relativo allo scambio di chiavi e importazione di una firma.

inoltre come una sorta di documentazione del codice. Chi volesse capire quali funzionalità siano offerte da un determinato modulo e come usarle potrebbe essere facilitato guardando come i test utilizzano tale modulo.

I test che si andranno a realizzare saranno unit test, per controllare le singole parti di codice che si svilupperanno e modificheranno, integration test, controllando l'interazione di diversi moduli per verificare che il loro comportamento sia corretto, e anche validation test per validare appunto il lavoro svolto, cioè controllare che si stia realizzando il prodotto corretto rispettando i bisogni degli utenti.

Capitolo 4

Tecnologie

4.1 Bluetooth

Bluetooth è una tecnologia wireless che permette di trasmettere dati tra dispositivi posizionati a breve distanza. La nascita di questa tecnologia risale al 1994 quando Ericsson volle sostituire i propri cavi RS-232 con una soluzione a radio frequenze, senza la necessità di fili. Circa nello stesso periodo anche altre società erano interessate ad una tecnologia simile per poter permettere a telefoni e computer di connettersi in modalità wireless. Invece che realizzare diversi progetti concorrenti alcune società come Intel e Nokia nel Dicembre del 1996 decisero di unirsi ad Ericsson per formare un Bluetooth *Special Interest Group* (SIG) cercando di garantire al progetto interoperabilità con i dispositivi disponibili al tempo e futuri. [16]

Figura 4.1: Logo ufficiale di Bluetooth

4.1.1 Caratteristiche Tecniche

Bluetooth opera sulle frequenze libere *industrial, scientific and medical* (ISM) ad una frequenza compresa tra 2402 e 2480 MHz. I trasferimenti avvengono utilizzando la tecnologia chiamata *frequency-hopping spread spectrum* che consente di inviare segnali radio cambiando rapidamente la frequenza di trasmissione aumentando così la larghezza di banda utilizzata.

I dispositivi Bluetooth sono divisi in classi, variando la quantità di corrente utilizzata e la loro portata. Come mostrato in tabella 4.1 esistono attualmente 3 classi di dispositivi, con la classe 3 avente una portata fino a 1 metro, la classe 2, trovata di solito nei dispositivi mobile, con una portata di circa 10 metri e infine la classe 1, principalmente utilizzata in ambito industriale, con una portata di circa 100 metri. Il range di comunicazione reale che è possibile raggiungere dipende non solo dalla qualità dei due dispositivi che intendono connettersi ma anche da fattori esterni come le condizioni atmosferiche o il numero di ostacoli presenti.

Classe	Massima potenza consentita		Range (m)
	(mW)	(dBm)	
1	100	20	∼100
2	2.5	4	∼10
3	1	0	∼1

Tabella 4.1: Classi esistenti di Bluetooth [9]

4.1.2 Protocolli

Bluetooth utilizza una quantità considerevole di protocolli divisi in due categorie:

- *Controller stack*: contengono le interfacce radio che dipendono dal timing con cui vengono eseguite le azioni. Generalmente si implementano in dispositivi al silicio a basso costo contenenti il modulo Bluetooth e un microprocessore.
- *Host stack*: contengono i dati di più alto livello. Di solito vengono implementati nel sistema operativo o come libreria esterna.

Il *Controller stack* prevede due tipi di connessioni possibili:

- *Asynchronous Connection-Less* (ACL): protocollo di comunicazione asincrono in cui i pacchetti vengono ritrasmessi automaticamente nel caso in cui non venissero confermati dal ricevente, permettendo di correggere i dati inviati in un canale radio disturbato o non stabile. ACL viene utilizzato sopratutto per l'invio di dati.

- *Synchronous connection-oriented* (SCO): protocollo di comunicazione sincrono che non richiede la ritrasmissione dei pacchetti. Viene utilizzato specialmente quando è richiesto un flusso dati continuo e real-time come ad esempio per una chiamata vocale.

Nell'*Host stack* alcuni dei protocolli più importanti e utilizzati sono:

- *Logical link control and adaptation protocol* (L2CAP): offre la possibilità di fare multiplexing dei dati tra diversi protocolli di livelli superiori, di segmentare e riassemblare i pacchetti, di creare una trasmissione multicast verso un gruppo di device Bluetooth e anche di garantire il *quality of service* (QOS).

- *Radio frequency communication* (RFCOMM): è un set di protocolli di trasporto basati su L2CAP ed emulano una porta seriale RS-232. RFCOMM, come avviene ad esempio con TCP, offre un canale dati affidabile.

- *Service discovery protocol* (SDP): permette di ottenere la lista dei servizi supportati dagli altri dispositivi e quali sono i parametri da utilizzare per poter stabilire una connessione con essi.

4.1.3 Implementazione su Linux

Bluez è uno stack sviluppato per il kernel Linux che fornisce una implementazione delle specifiche Bluetooth. Nel 2006 Bluez ha aggiunto il supporto a tutti i layer e profili core di Bluetooth. Lo sviluppo è iniziato da parte di Qualcomm ed è stato aggiunto al kernel Linux a partire dalla versione 2.4.6 risalente al 2001. Inoltre sono stati sviluppati anche i pacchetti *bluez-utils* e *bluez-firmware* che contengono utilies di basso livello per poter gestire i dispositivi Bluetooth collegati al sistema.

Per poter interagire con Bluetooth, anche dai programmi in esecuzione sul sistema, sono presenti diversi binding per i più diffusi linguaggi di programmazione come ad esempio *PyBluez*[1] che permette di far comunicare un programma Python con i device Bluetooth.

4.2 Magic Wormhole

Magic Wormhole è una libreria, e tool da linea di comando, che permette di inviare file e cartelle tra due computer in maniera sicura. L'obbiettivo è quello di riuscire a stabilire un canale di comunicazione sicuro in maniera più facile rispetto agli altri tool esistenti, sopratutto se i due computer ad esempio si trovano in reti locali separate.

I due endpoints sono identificati da un *"wormhole code"* generato solitamente dal mittente e inserito manualmente dal ricevente. I codici sono di default formati da un numero, rappresentante il canale su cui avverrà la comunicazione, e due parole foneticamente distinte corrispondenti ad una password di sessione. Questi wormhole codes sono generati ad ogni trasferimento, quindi non sono riutilizzabili e non c'è la necessità di memorizzarli.

4.2.1 Esempio di trasferimento da CLI

Nello snippet che si trova qui sotto è mostrato un esempio di trasferimento da linea di comando utilizzando il tool wormhole. Il mittente esegue il programma con il parametro *send* e il file che intende inviare. Successivamente il codice generato lato mittente dovrà essere comunicato al ricevente, il quale, eseguendo *wormhole receive*, lo inserirà per poter iniziare il trasferimento del file.

```
1 $ wormhole send mySecret.txt
2 Sending 71 Bytes file named 'mySecret.txt'
3 Wormhole code is: 5-replica-fallout
4 On the other computer, please run:
5
6 wormhole receive 5-replica-fallout
7
8 Sending (<-192.168.10.200:53378)..
9 100%|============================| 71.0/71.0 [00:00<00:00, 109kB/s]
```

[1] https://github.com/pybluez/pybluez

```
File sent.. waiting for confirmation
Confirmation received. Transfer complete.
```

Mittente

```
$ wormhole receive
Enter receive wormhole code: 5-replica-fallout
Receiving file (71 Bytes) into: mySecret.txt
ok? (y/N): y
Receiving (->tcp:192.168.10.200:37779)..
100%|============================| 71.0/71.0 [00:00<00:00, 401B/s]
Received file written to mySecret.txt
```

Destinatario

4.2.2 Design

Wormhole utilizza il *"Password-Authenticated Key Exchange"*, detto anche PAKE, cioè una famiglia di algoritmi crittografici che utilizzano una password corta a bassa entropia per stabilire una chiave condivisa sicura ad alta entropia. L'algoritmo, richiedendo una interazione tra le due parti, evita tutti i possibili attacchi offline come ad esempio i brute-force. L'unico modo per un attaccante di scoprire la chiave condivisa è quella di effettuare un *man-in-the-middle* durante il primo tentativo di connessione tra le due parti e cercare di indovinare il codice utilizzato. Di default il codice è lungo 16 bit [2], quindi un attaccante avrà una chance su 65536. Questo perché quando si userà un codice errato al mittente verrà notificato tale tentativo interrompendo automaticamente l'invio e quando il mittente tenterà nuovamente di inviare il file, un nuovo codice verrà generato lasciando sempre l'attaccante con una chance su 65536.

4.2.3 Relays

Wormhole per poter omettere agli utenti gli indirizzi IP e numeri di porta utilizza un *rendevous server*, ovvero un semplice websocket relay che spedisce i messaggi tra due client. Inoltre è presente un *transit relay*, cioè un ulteriore server che mette in comunicazione due connessioni TCP inbound.

[2] Quando si invia un file è possibile richiedere la generazione di un codice più lungo.

Di default vengono utilizzati due server pubblici mantenuti dagli sviluppatori del progetto, ma in alternativa è anche possibile utilizzarne di propri visto che il codice per farlo è rilasciato con la libreria stessa.

Quando il *rendevous server* comunica ai client l'IP e il numero di porta della controparte essi cercheranno di stabilire una comunicazione diretta tra loro. Nel caso fallissero, perché ad esempio si trovano dietro NAT, allora utilizzeranno il *transit relay* come metodo di fallback.

Capitolo 5

Analisi del Problema

5.1 Architettura Logica

Potendo utilizzare Magic Wormhole per soddisfare il requisito di aggiunta di un metodo di trasferimento alternativo che funzionasse anche in presenza di una rete guest, si sono considerate due possibilità riguardo al come affiancarlo al già esistente Avahi:

- Far scegliere di volta in volta agli utenti se utilizzare Wormhole come metodo di trasferimento.
- Nascondere agli utenti la scelta del tipo di trasferimento da utilizzare e usare sempre sia Wormhole che Avahi.

Anche se Wormhole utilizza un trasferimento criptato end-to-end, gli utenti potrebbero preferire di utilizzare solo il server locale se sono sicuri di poter stabilire un canale comunicativo anche senza l'utilizzo di Internet, ottenendo al tempo stesso una maggiore velocità di inizializzazione nel caso in cui la rete Internet fosse particolarmente lenta o non del tutto stabile. Per questo motivo si è deciso di lasciare la possibilità agli utenti di abilitare e disabilitare il trasferimento tramite Wormhole.

Per quanto riguarda il trasferimento diretto a corto raggio si è optato per l'utilizzo di Bluetooth visto che soddisfa i requisiti richiesti ed ha il vantaggio di essere largamente diffuso. Per la coesistenza con i restati metodi di trasferimento si è scelto di aggiungere il Bluetooth senza l'utilizzo di ulteriori pulsanti o switch. Keysign cercherà di inviare automaticamente la chiave

	LAN Only	Guest LAN+Internet	LAN+Internet	Remote
Avahi	✓		✓	
Wormhole		✓	✓	✓

Tabella 5.1: In quali occasioni si può usare Avahi e Magic-Wormhole

anche attraverso il Bluetooth nel caso in cui fosse presente e utilizzabile. A differenza di Magic Wormhole, l'inizializzazione del trasferimento tramite Bluetooth dovrebbe essere quasi istantanea e quindi, anche nel caso in cui fosse stata data la possibilità di disabilitarlo manualmente, non sarebbero state visibili differenze nelle tempistiche per la generazione del QR code e del codice di sicurezza per l'invio di una chiave.

Quando Keysign viene avviato in modalità ricezione e si scansiona un QR code, il client cercherà di scaricare la chiave utilizzando i vari metodi di trasferimento disponibili uno dopo l'altro fino a quando il download non vada a buon fine o non si avranno più metodi a disposizione da provare. Per decidere l'ordinamento con cui il client debba tentare i vari trasferimenti si è usato come parametro di riferimento la velocità necessaria a stabilire una connessione. Inoltre si è cercato di lasciare per ultimo l'utilizzo della rete Internet perché se fosse stata possibile una connessione diretta non avrebbe avuto senso scomodare Internet per effettuare il trasferimento. Dopo alcuni test effettuati si è potuto constatare che l'utilizzo della rete locale fosse più veloce rispetto al Bluetooth sia per quanto riguarda il tempo necessario per stabilire una connessione che per il trasferimento vero e proprio. Detto ciò, si è scelto di ordinare i tentativi di connessioni in: server locale Avahi, Bluetooth e Magic Wormhole.

In figura 5.1 viene mostrato il diagramma di sequenza relativo allo scambio e firma di chiavi da parte di due utenti. In figura 5.2 è presente un diagramma di sequenza che entra nel dettaglio di come avviene il trasferimento nei casi in cui il ricevente possa contattare il mittente solo tramite Wormhole, o quando anche il Bluetooth fosse funzionante e infine con anche il server locale Avahi. Per quanto riguarda la struttura, in figura 5.3 viene presentato il diagramma delle classi, con cui è possibile ottenere una visione d'insieme relativa all'organizzazione in classi del progetto.

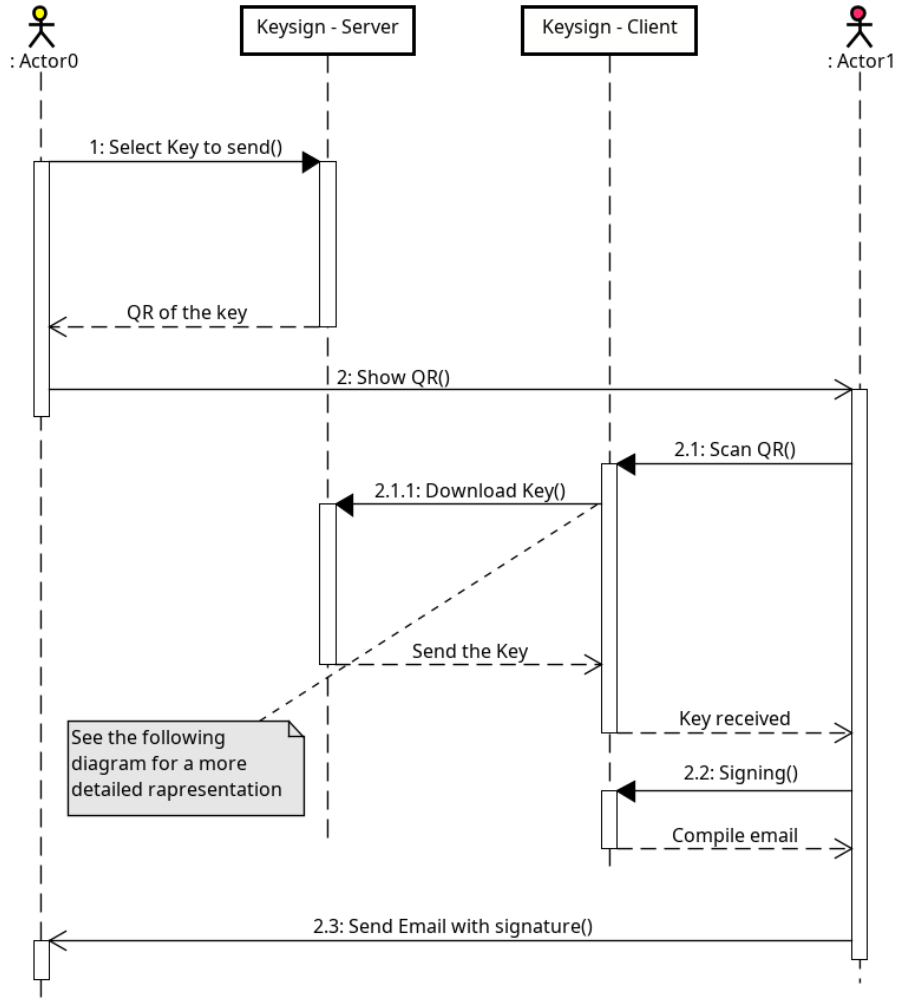

Figura 5.1: Diagramma di sequenza relativo allo scambio di chiavi e firme.

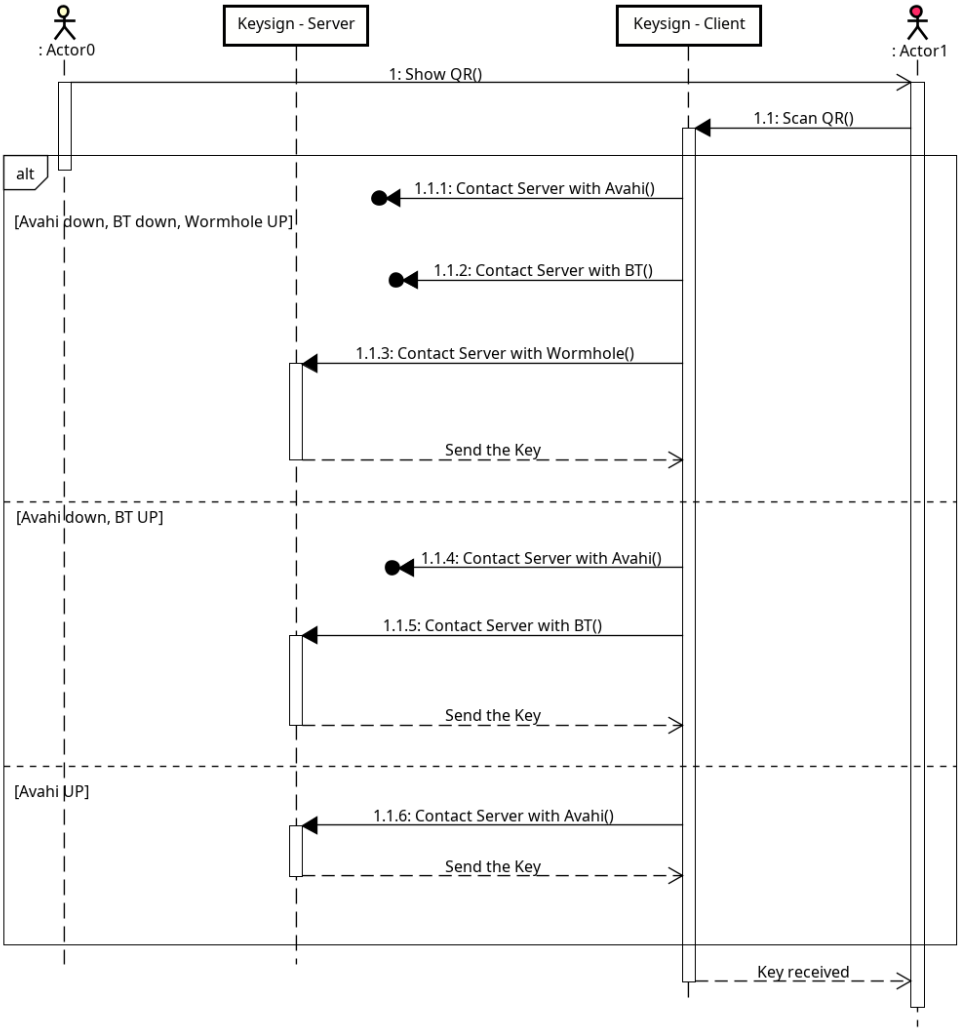

Figura 5.2: Vista di dettaglio sullo scambio delle chiavi con un diagramma di sequenza.

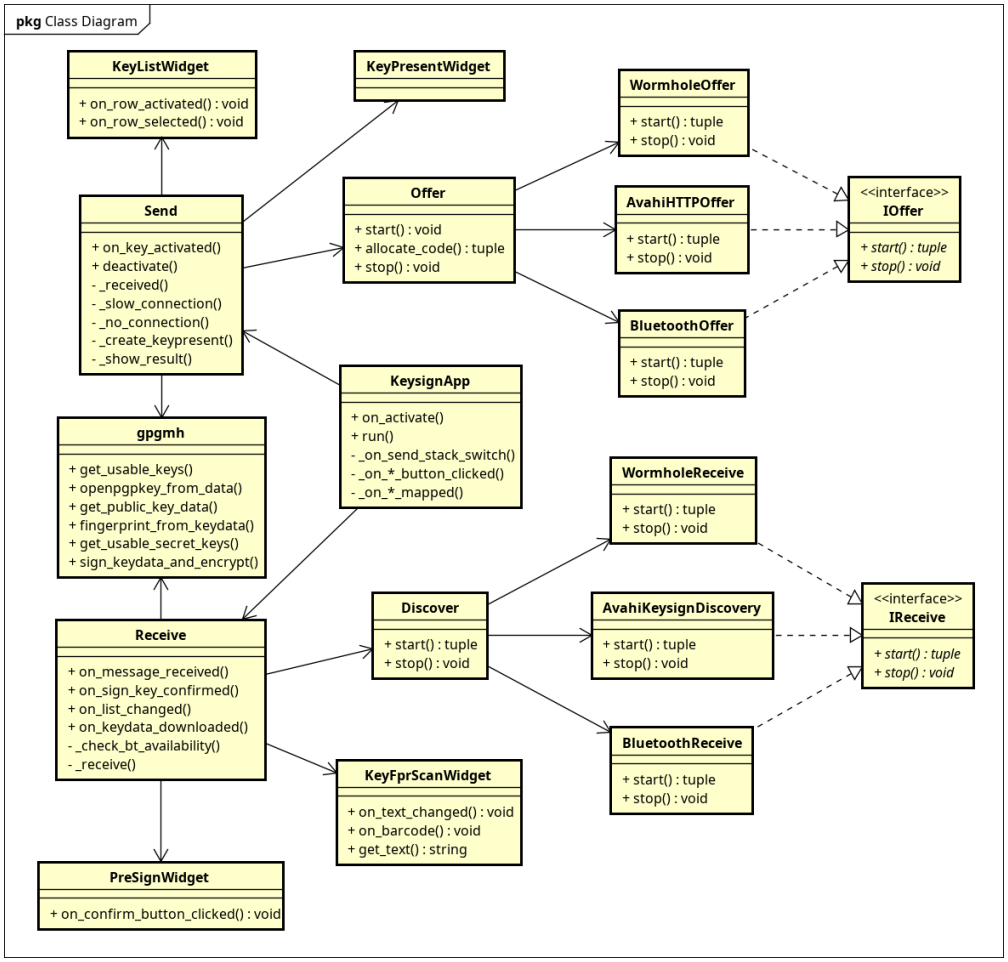

Figura 5.3: Diagramma delle classi generale del progetto.

5.2 Analisi dei Rischi

Per questo progetto si può utilizzare la famosa citazione di Adam Savage: "Failure is *always* an option".

Le problematiche che potrebbero presentarsi durante la realizzazione di questo progetto sono molteplici e le più concrete sono:

- Rendere troppo complicata l'interfaccia utente quando si aggiungono i metodi alternativi di trasferimento. Questo vanificherebbe l'obiettivo di GNOME-Keysign, ovvero quello di rendere lo scambio di chiavi semplice e intuitivo. Per evitare tale scenario si cercherà di mantenere i contatti con gli utenti finali durante tutta la fase di sviluppo ottenendo continuamente feedback tentando di evitare di far prendere al progetto una direzione indesiderata.

- Non riuscire ad incorporare in upstream le modifiche che verranno proposte in questa tesi. Anche nel caso in cui si riuscissero ad implementare tutte le funzioni prefissate, c'è sempre la possibilità che tali modifiche vengano rigettate, ad esempio perché non considerate appropriate o perché lo stile dell'implementazione non rispecchi quello adottato dal resto del progetto. Nel caso in cui questo dovesse avvenire l'intero lavoro svolto sarebbe quasi reso vano visto che gli utenti finali non potrebbero mai beneficiare di tali migliorie. La tecnica che verrà adottata per evitare che questo possa accadere è di tenere sempre informati i maintainer dell'applicazione sull'andamento e le scelte che verranno adottate in questo progetto di tesi.

5.3 Design

Il mittente quando inviava una chiave tramite il server locale non informava l'utente del fatto che qualcuno avesse scaricato con successo la sua chiave pubblica e il server Avahi rimaneva attivo per consentire altri download senza che l'utente dovesse fare nulla. Con Magic Wormhole invece non è possibile consentire connessioni multiple senza cambiare il codice di trasferimento perché questo costituirebbe una elevata diminuzione per la sicurezza. Per questo motivo è stata introdotta una pagina finale indipendentemente dal metodo di trasferimento utilizzato in cui viene mostrato all'utente

il risultato dell'invio lato mittente. In questo modo è possibile informare l'utente anche di eventuali tentativi di download fraudolenti.

In figura 5.4 viene mostrato il diagramma di attività relativo all'invio di una propria chiave pubblica, in cui è possibile notale l'opzionalità di Wormhole e l'aggiunta della pagina finale con il risultato dell'invio.

In figura 5.5 è presente il diagramma di attività per la parte di ricezione. Nel caso in cui non fosse possibile scaricare la chiave con i codici presenti nel QR code, si potrà scansionarne un altro e ritentare il download.

5.4 Trasferimento tramite Magic Wormhole

Il codice per il trasferimento tramite Wormhole viene generato contattando il relay server utilizzando la rete Internet. Quindi se si ha a disposizione una rete Internet lenta si può dover attendere fino a qualche secondo prima che il codice venga generato. Per questo motivo le due possibili tecniche di integrazione in *Keysign* che sono state prese in considerazione sono:

- Presentare all'utente direttamente il codice per il trasferimento locale e solo in un secondo momento, quando sarà a disposizione, aggiungere il codice per Wormhole.

- Far transitare l'utente alla pagina di invio solo quando il codice Wormhole sarà pronto.

Nel secondo caso gli utenti dovranno aspettare, magari mostrando uno spinner di caricamento, la disponibilità del codice Wormhole prima di poter visualizzare la pagina di invio, la quale però sarà già completa di tutte le informazioni. Al contrario nel primo caso quando un utente sceglierà di inviare una propria chiave potrà procedere alla pagina di invio quasi istantaneamente, ma magari senza visualizzare ne il QR code ne il codice per Wormhole, i quali saranno presentati solo in un secondo momento. Anche se magari con il primo metodo l'applicazione potesse sembrare più veloce, si è optato di utilizzare il secondo metodo per evitare di mostrare agli utenti una pagina che possa cambiare dinamicamente dopo un intervallo variabile di tempo.

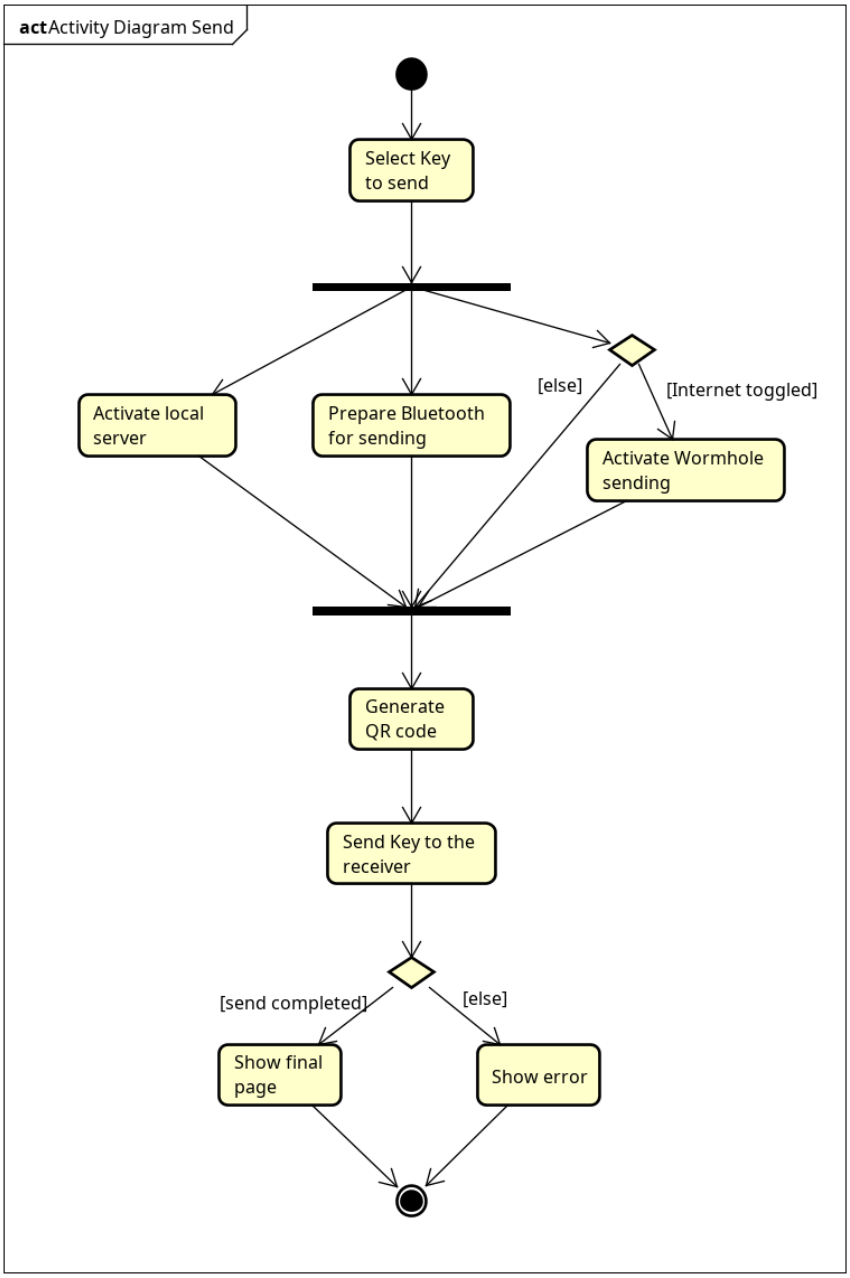

Figura 5.4: Diagramma di attività relativo alla parte di invio.

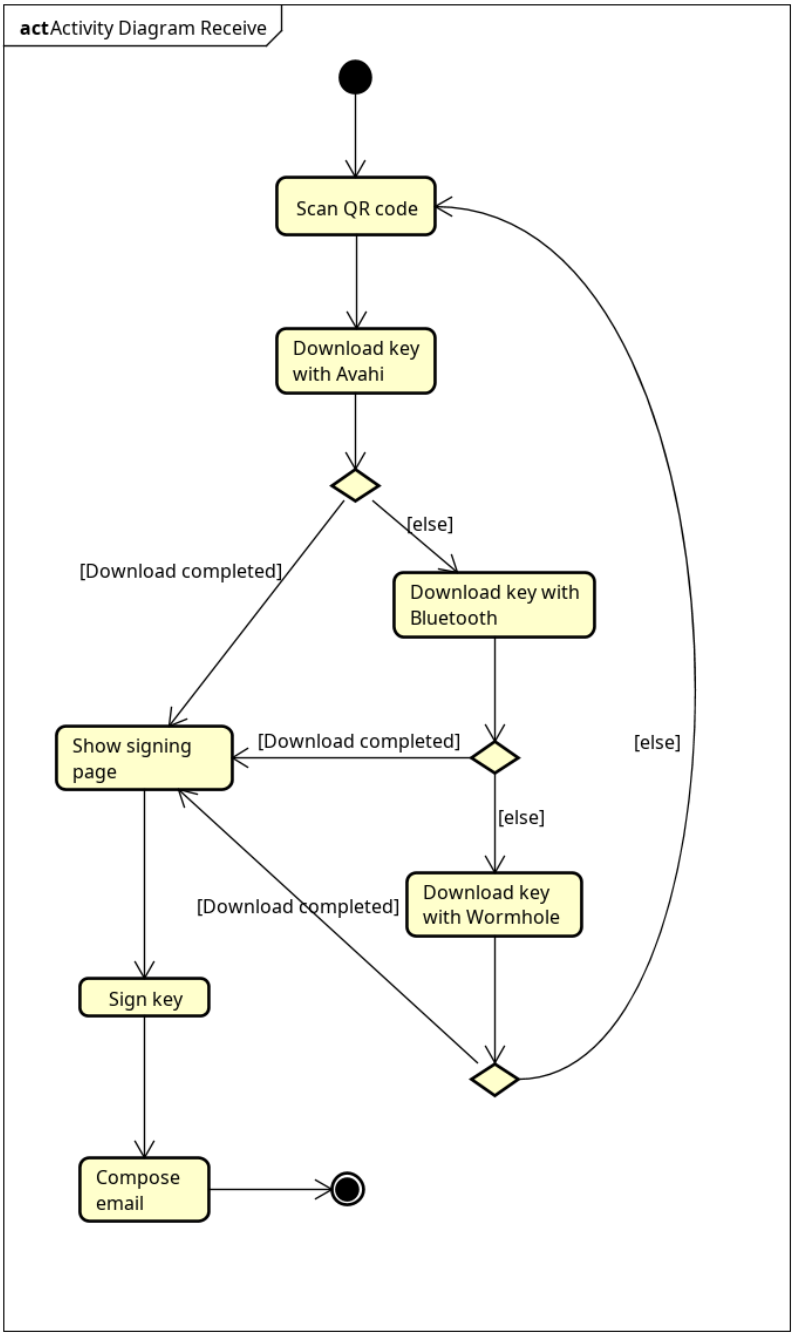

Figura 5.5: Diagramma di attività relativo alla ricezione di una chiave.

5.5 Trasferimento tramite Bluetooth

Per consentire a due istanze di Keysign di trovarsi utilizzando il Bluetooth (BT) diverse possibilità si possono prendere in considerazione:

- Cambiare l'alias del BT per fare matching con il fingerprint della chiave che si vuole inviare: essendo il fingerprint già utilizzato per il server locale Avahi, nel QR code non ci sarebbe bisogno di inserire alcun tipo di codice extra. Il ricevente inserisce il fingerprint o scansiona il QR code, Keysign effettua una ricerca dei dispositivi Bluetooth nelle vicinanze e se si rileva un disponitivo con il nome uguale a quello che si sta cercando si inizia la procedura per il trasferimento della chiave.

  Il vantaggio di questo metodo è il fatto di non dover usare alcun tipo di codice aggiuntivo visto che sarà necessario solo lo stesso fingerprint già usato da Avahi. Per contro dover cambiare il nome del Bluetooth è un metodo un po invasivo e inoltre il processo di ricerca, per ottenere i MAC e gli alias dei dispositivi nelle vicinanze, è abbastanza lungo.

- Usare direttamente il MAC address del BT: lato mittente si inserisce nel QR code anche il MAC address del Bluetooth in uso e il ricevente, una volta scansionato il QR, effettua un tentativo di connessione diretta con il dispositivo specificato nel QR.

  I vantaggi in questo caso sono la maggiore velocità dovuta al non dover effettuare alcuna ricerca di dispositivi Bluetooth, visto che essendo già a conoscenza del MAC è possibile tentare una connessione direttamente, e inoltre non si deve cambiare il nome dell'adattatore Bluetooth in uso. Lo svantaggio è la necessità di utilizzare un nuovo codice aggiuntivo, ovvero l'indirizzo MAC.

Considerando troppo invasivo il cambiamento del nome dell'adattatore Bluetooth si è deciso di optare per l'utilizzo del MAC address per consentire alle due istanze di Keysign di stabilire un canale di comunicazione.

Inoltre invece che mostrare all'utente un ulteriore codice di sicurezza si è optato per rilegare il trasferimento tramite Bluetooth solo al QR code lasciando il metodo di fallback, tramite digitazione manuale del codice di sicurezza, al solo trasferimento con server locale. Questa scelta consente di evitare di mostrare agli utenti codici ancora più lunghi e difficili da digitare.

Per rendere il trasferimento tramite Bluetooth semplice ed immediato si è scartata la necessità di effettuare il pairing tra i dispositivi. Ovviamente il compromesso qui sarà l'assenza di garanzie sui dati che verranno ricevuti, non potendo creare un canale comunicativo criptato end-to-end. Ma anche questo scenario è accettabile dal punto di vista della sicurezza visto che sarà comunque possibile garantire l'autenticità della chiave ricevuta utilizzano l'HMAC presente nel QR code.

Infine considerando che non tutti i sistemi hanno a disposizione la possibilità di utilizzare Bluetooth, come la maggior parte dei computer Desktop, si è deciso di lasciare il Bluetooth come opzionale in modo da non forzare gli utenti ad installare dipendenze per funzioni che non sarebbero comunque in grado di usare.

5.6 Flatpak dell'Applicazione

Per soddisfare il requisito di aumento dei canali con cui gli utenti potessero ottenere GNOME-Keysign si è deciso di utilizzare Flatpak.

A seguito di una fase preliminare di studio le maggiori criticità rilevate sono state:

- Flusso video dalla webcam: dopo aver controllato lo stato di Flatpak si è scoperto che attualmente non è disponibile un portale che supporti le webcam.[1] Probabilmente quindi si dovrà ripiegare chiedendo un permesso nel manifest per l'utilizzo dei device collegati al computer.

- Invio di email: attualmente Keysign per aprire il client email presente sul sistema esegue il comando *xdg-email*. Però dalla sandbox non è possibile utilizzare questa tecnica visto che non è consentita l'esecuzione di software sul sistema host. Per risolvere questo problema Flatpak mette a disposizione un portale che consente di aprire il client email del sistema host. Per poterlo utilizzare sarà richiesta una modifica ai sorgenti di Keysign.

- Utilizzo del Bluetooth: a seguito di una fase preliminare di studio e testing si è scoperto che dalla sandbox non era consentito utilizzare dispositivi Bluetooth dell'host. Fortunatamente, dopo aver contattato

[1] `https://github.com/flatpak/xdg-desktop-portal/issues/38`

il maintainer di Flatpak, è stata aggiunta tale possibilità a partire dalla versione 0.11.8 [2]

Una volta realizzato il Flatpak si dovrà cercare di rilasciarlo su Flathub per consentire agli utenti di trovare ed installare l'applicazione con facilità. Nel caso in cui si riuscisse a creare un Flatpak completamente funzionante, la pubblicazione su Flathub non dovrebbe rilevarsi un problema.

5.7 Importazione di una firma

La funzionalità di importazione di una nuova firma può essere presentata agli utenti attraverso:

- L'aggiunta di un nuovo tab nella schermata principale.
- Permettere di importare una chiave solo dopo aver inviato con successo una propria chiave. Il vantaggio di questo approccio è il fatto di rendere i vari passaggi più lineari, ovvero nascondere la possibilità di importare una firma se l'utente, non avendo ancora inviato la propria chiave, molto probabilmente non ne abbia bisogno. Per contro però se il ricevente tardasse ad inviare l'email con la signature e il mittente chiudesse *Keysign* non si avrà più la possibilità di importarla. Ed essendo l'intero processo di invio della firma non real time, questa è una possibilità concreta.
- Creare una applicazione minimale companion a GNOME-Keysign con l'unico scopo di permettere agli utenti di importare le proprie firme. Questo approccio permette di sperimentare da subito il processo di importazione, non dovendo aspettare che l'implementazione venga inclusa in upstream. In seguito si può valutare se distribuire tale applicazione companion insieme a *Keysign* o se fosse il caso di includere tale funzionalità direttamente nell'applicazione principale.

Tenendo in considerazione i vari risvolti positivi e negativi dei differenti approcci, inizialmente si produrrà una applicazione companion separata in modo da poter testare la funzionalità da subito. In seguito, dopo aver raccolto anche i feedback degli utenti, si potrà optare se sia il caso di integrarla nell'applicazione principale.

[2] `https://github.com/flatpak/flatpak/commit/7739209`

Capitolo 6

Implementazione

6.1 Libreria asincrona per Keysign

Su Python per poter scrivere codice concorrente ci sono diverse librerie tra cui poter scegliere, ad esempio: Twisted, Asyncio, Trio, Curio e molte altre. Attualmente GNOME-Keysign supporta sia Python 2 che Python 3, quindi per la scelta della libreria da utilizzare si è cercato di considerare prima le librerie che fornissero la compatibilità anche con Python 2 e, solo nel caso in cui non ci fossero state soluzioni valide, si sarebbe optato per una libreria che supportasse solo Python 3.

Tenendo in considerazione che Magic Wormhole basa il suo funzionamento su Twisted e che quindi tale libreria diventerà una dipendenza indiretta anche di GNOME-Keysign, Twisted è stata la prima libreria presa in considerazione. Essa supporta Python 2, ha una codebase stabile e matura, consente l'utilizzo di callback e inline callback per gestire le chiamare asincrone e infine si integra con gli event loop di GTK[1]. Coprendo tutte le necessità richieste per questo progetto, si è deciso di scegliere Twisted senza il bisogno di investigare anche le altre librerie esistenti.

6.2 Deferred, programmazione asincrona

Il framework Twisted utilizza la programmazione asincrona basata su callback, ovvero i blocchi di codice bloccanti non dovranno essere eseguiti di-

[1]La libreria grafica utilizzata da GNOME-Keysign

rettamente ne essere delegati ad ulteriori thread, ma si utilizzeranno solo funzioni la cui esecuzione è istantanea e il loro tipo di ritorno è una promise che prima o poi conterrà il risultato della funzione che era stata invocata.

Il meccanismo utilizzato da Twisted per controllare il flusso di codice asincrono si chiama *deferred*. Un *deferred* è un oggetto che contiene una promessa. È possibile legare ad esso i callback delle funzioni e, una volta che i risultati saranno disponibili, i relativi callback verranno chiamati.

6.2.1 Callback e Inline Callback

Twisted prevede inoltre l'utilizzo di un decorator chiamato *inlineCallbacks* che permette di lavorare con i deferred senza usare funzioni di callback. Per poterlo utilizzare bisogna scrivere il codice come generators con un *yield* del risultato invece che legare i callback.

Nel primo esempio qui sotto viene presentato un blocco di codice che per ottenere e controllare le password in asincrono utilizza due callback evitando di bloccare l'esecuzione del codice con chiamate che richiedano un tempo di esecuzione lungo.

Nel secondo esempio invece viene mostrato lo stesso codice, questa volta però scritto utilizzando le inline callback. L'enorme vantaggio che si nota immediatamente è la possibilità di scrivere codice asincrono ma che comunque mantenga le istruzioni in ordine evitando salti difficili da seguire.

Considerando il fatto che gli inline callback rendono il codice più facile da leggere e seguire, per il progetto si è scelto di utilizzarli al posto dei callback classici.

```
1  from twisted.internet import reactor
2
3  def on_failure(error):
4      print("An error occurred: %s", error)
5      reactor.stop()
6
7  def on_success():
8      print("Success!")
9      reactor.stop()
10
11 def check_passwords(passwords):
12     d = async_check_passwords(passwords) # This returns a Deferred
13
14     d.addCallback(on_success)
15     d.addErrback(on_failure)
16
17 def get_passwords():
```

```
18      d = makeRequest("GET", "/passwords") # This returns a Deferred
19
20      d.addCallback(check_passwords)
21      d.addErrback(on_failure)
22
23  if __name__ == '__main__':
24      get_passwords()
25      reactor.run()
```

Twisted con callback

```
1   from twisted.internet import reactor
2   from twisted.internet.defer import inlineCallbacks
3
4   @inlineCallbacks
5   def check_passwords():
6       try:
7           # makeRequest still returns a Deferred, but after the yield it
8           # contains the result
9           passwords = yield makeRequest("GET", "/passwords")
10          yield async_check_passwords(passwords)
11          print("Success!")
12      except Exception as e:
13          print("An error occurred: %s", error)
14      finally:
15          reactor.stop()
16
17  if __name__ == '__main__':
18      check_passwords()
19      reactor.run()
```

Twisted con inline callback

6.2.2 Invio e ricezione di chiavi

Per prima cosa si è notato che la ricezione tramite il server locale Avahi era sincrona e bloccante. Avendo ora a disposizione la possibilità di utilizzare la programmazione asincrona di Twisted si è evitato di effettuare una chiamata diretta al metodo Avahi `find_key` utilizzando un `deferToThread`.

In seguito è stato aggiunto il trasferimento tramite Magic Wormhole e Bluetooth. Nella loro implementazione si è cercato di mantenere il più possibile una struttura simile al già esistente Avahi.

Per quanto riguarda Magic Wormhole si implementa la parte di invio e ricezione attenendosi alle specifiche ufficiali riuscendo così ad ottenere interoperabilità anche con la versione CLI di Wormhole. Questo significa

che se un utente utilizza GNOME-Keysign per inviare la propria chiave, il ricevente potrebbe utilizzare Wormhole da riga di comando e ricevere con successo la sua chiave pubblica. Oppure se il mittente invia la chiave pubblica utilizzando direttamente Wormhole da CLI, il ricevente può usare GNOME-Keysign per riceverla.

```
@inlineCallbacks
def start(self):
    log.info("Wormhole: Trying to receive a message with code: %s",
        self.code)

    self.stop()
    self.w = wormhole.create(self.app_id, RENDEZVOUS_RELAY, reactor)
    # The following mod is required for Python 2 support
    self.w.set_code("%s" % str(self.code))

    try:
        message = yield self.w.get_message()
        m = decode_message(message)
        key_data = None
        offer = m.get("offer", None)
        if offer:
            key_data = offer.get("message", None)
        if key_data:
            log.info("Message received: %s", key_data)
            if self._is_verified(key_data.encode("utf-8")):
                log.debug("MAC is valid")
                success = True
                message = ""
                # send a reply with a message ack, this also ensures
                # wormhole cli interoperability
                reply = {"answer": {"message_ack": "ok"}}
                reply_encoded = encode_message(reply)
                self.w.send_message(reply_encoded)
                returnValue((key_data.encode("utf-8"), success, message))
            else:
                log.warning("The received key has a different MAC")
                self._reply_error(_("Wrong message authentication code"))
                self._handle_failure(WrongPasswordError())
        else:
            log.info("Unrecognized message: %s", m)
            self._reply_error("Unrecognized message")
            self._handle_failure(TransferError())
    except WrongPasswordError as wpe:
        log.info("A wrong password has been used")
        self._handle_failure(wpe)
    except LonelyError as le:
        log.info("Closed the connection before we found anyone")
        self._handle_failure(le)
```

Snippet relativo alla ricezione tramite Wormhole

L'invio di una chiave tramite Bluetooth avviene aprendo una server socket e rimanendo in attesa di ricevere una connessione. In questo caso sia l'apertura della socket che l'invio dei dati vengono eseguiti in asincrono per evitare di bloccare l'interfaccia grafica durante l'esecuzione di tali operazioni.

```
@inlineCallbacks
def start(self):
    self.stopped = False
    message = "Back"
    success = False
    try:
        while not self.stopped and not success:
            # server_socket.accept() is not stoppable. So with select we
            # can call accept() only when we are sure that there is
            # already a waiting connection
            ready_to_read, ready_to_write, in_error = yield
                threads.deferToThread(
                select.select, [self.server_socket], [], [], 0.5)
            if ready_to_read:
                # We are sure that a connection is available, so we can
                # call accept() without deferring it to a thread
                client_socket, address = self.server_socket.accept()
                key_data = get_public_key_data(self.key.fingerprint)
                kd_decoded = key_data.decode('utf-8')
                yield threads.deferToThread(client_socket.sendall,
                    kd_decoded)
                log.info("Key has been sent")
                client_socket.shutdown(socket.SHUT_RDWR)
                client_socket.close()
                success = True
                message = None
    except Exception as e:
        log.error("An error occurred: %s" % e)
        success = False
        message = e

    returnValue((success, message))
```

Snippet relativo all'invio tramite Bluetooth

I vari metodi di invio disponibili sono stati raggruppati e gestiti in una nuova classe chiamata `Offer` che mette a disposizione un metodo per inizializzare l'invio delle chiavi e un metodo che consente di fermare tale operazione. In questo modo quando si deve effettuare l'invio di una chiave basterà eseguire il metodo offerto da `Offer` senza doversi preoccupare di inizializzare manualmente tutti i metodi di trasferimento disponibili.

Anche per la ricezione è stata messa a disposizione una classe chiamata

`Discover` che provvede a fare il parsing del contenuto presente nell'eventuale QR code scansionato, estraendo i codici per i vari metodi di trasferimento. Inoltre grazie al metodo `start`, consente di tentare la ricezione di una chiave pubblica gestendo in autonomia i vari metodi di trasferimento disponibili, ordinandoli secondo quanto previsto nella sezione 5.1.

6.3 Interfaccia grafica

Nella figura 6.1 è possibile notare l'aggiunta di un toggle chiamato `Internet` che consente di abilitare o disabilitare l'invio tramite Magic Wormhole.

In figura 6.2 viene mostrata l'aggiunta di una barra di informazioni, chiamata infobar, che consente di notificare l'utente nel caso in cui Internet non fosse disponibile o se la connessione fosse molto lenta e la generazione del codice richiedesse più del previsto.

La figura 6.3 mostra la schermata di invio dopo aver scelto una chiave. Come si può notare, a differenza della figura 2.2, se si sceglie un invio tramite Internet, il codice di sicurezza mostrato sarà solo relativo a Magic Wormhole, lasciando il codice del server locale solo all'interno del QR code.

Infine in figura 6.4 è presente la pagina con il risultato dell'invio. Quella mostrata è la schermata visualizzata quando un trasferimento va a buon fine. In alternativa verrà mostrato un messaggio di errore nel caso in cui l'invio non si fosse concluso correttamente. In alto a sinistra è stato inserito un pulsante per consentire agli utenti di tornare alla schermata di invio, nel caso debbano effettuarne altri.

6.4 Flatpak

Come detto nella sezione 5.6, per permettere una maggiore diffusione di GNOME-Keysign si è deciso di creare un manifest per Flatpak per poi poterla pubblicare su Flathub. [2] [3]

Una delle parti più rilevanti del manifest che si è realizzato è il tag *finish-args* che contiene i permessi richiesti per poter funzionare correttamente. Tra essi abbiamo:

[2] `https://github.com/flathub/org.gnome.Keysign`
[3] `https://flathub.org/apps/details/org.gnome.Keysign`

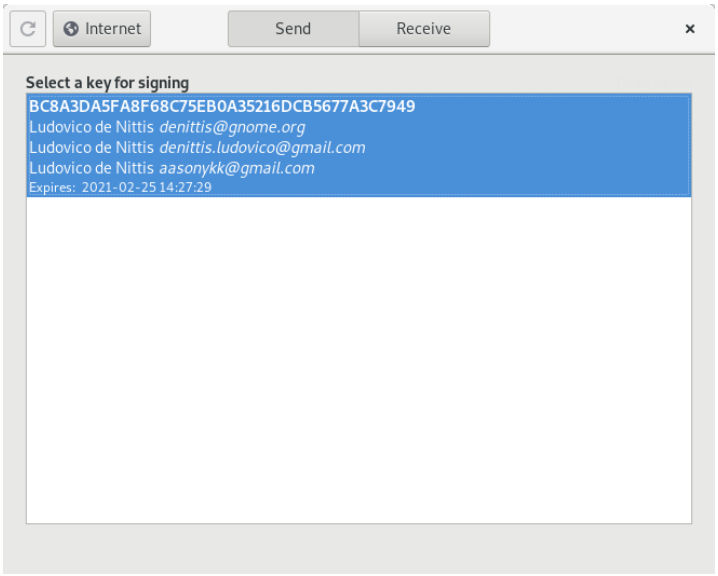

Figura 6.1: Schermata principale con possibilità di abilitare l'invio tramite la rete Internet.

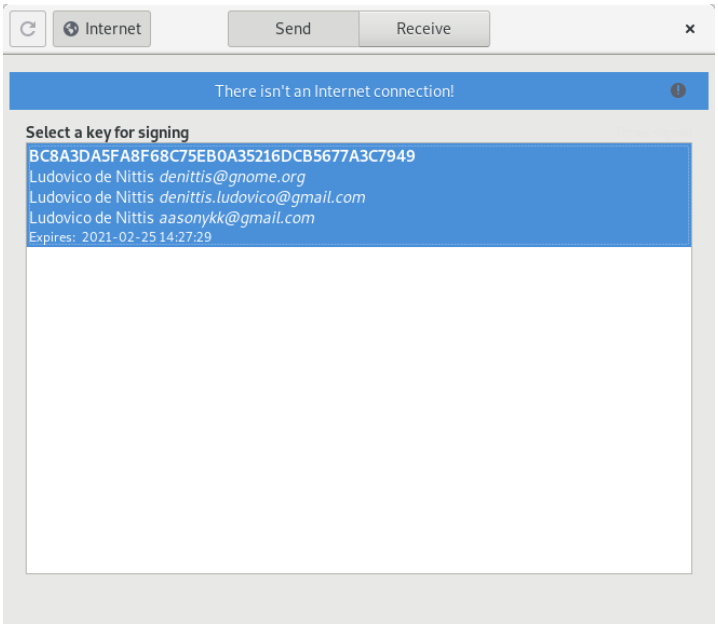

Figura 6.2: Barra di informazioni che notifica l'utente riguardo l'assenza di connessione ad Internet.

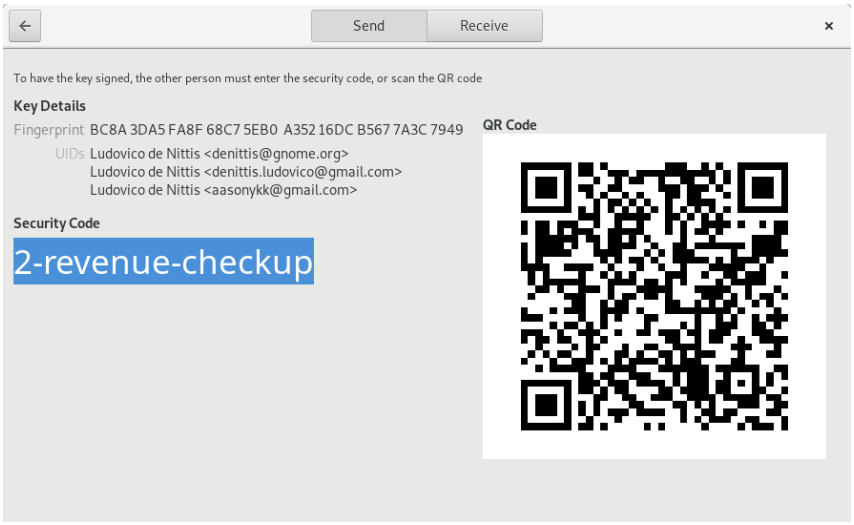

Figura 6.3: Schermata di invio di una chiave utilizzando la rete Internet.

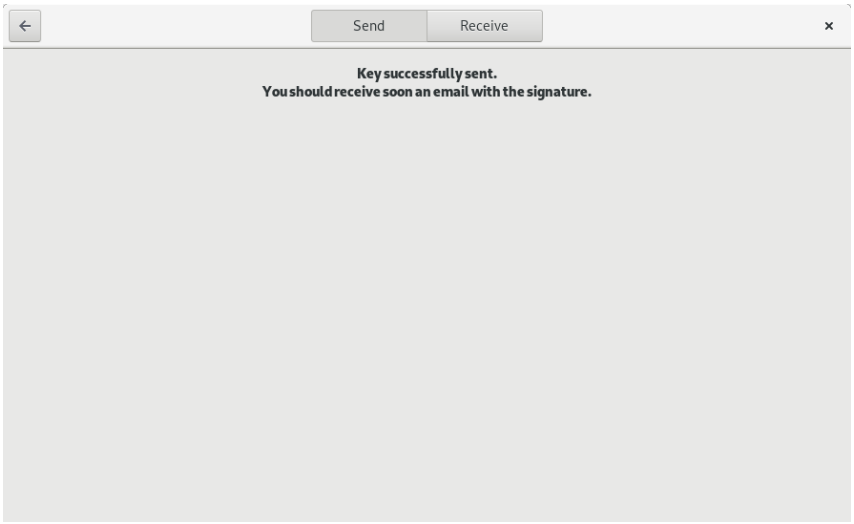

Figura 6.4: Schermata mostrata a seguito di un invio.

- *share=network* che consente di poter interagire con la rete locale e Internet.
- *system-talk-name=org.freedesktop.Avahi* per poter comunicare con il DBus di Avahi, consentendo di inviare e ricevere file utilizzando la rete LAN.
- *system-talk-name=org.bluez* e *allow=bluetooth* che consentono di utilizzare il Bluetooth passando per il driver Bluez di Linux.
- *device=all* questo permesso è necessario per poter utilizzare la webcam. Questa è una soluzione temporanea fino a quando il portale specifico non venga realizzato. [4]
- *filesystem= /.gnupg:ro* e *filesystem=xdg-run/gnupg:ro* che consentono di accedere alle due cartelle specificare in modalità read-only (ro). La cartella *gnupg* contiene le chiavi OpenPGP di un utente e quindi Keysign deve poterci accedere per poterle inviare. Sono state elencate due cartelle perché, in base alla versione di gnupg in uso, la cartella di default su cui vengono salvate le chiavi differisce.

```
id: org.gnome.Keysign
runtime: org.gnome.Platform
runtime-version: '3.28'
sdk: org.gnome.Sdk
command: gnome-keysign
copy-icon: true
finish-args:
  - --share=ipc
  - --socket=x11
  - --socket=wayland
  - --share=network
  - --system-talk-name=org.freedesktop.Avahi
  - --system-talk-name=org.bluez
  - --allow=bluetooth
  - --device=all
  - --filesystem=~/.gnupg:ro
```

[4] https://github.com/flatpak/xdg-desktop-portal/issues/38

```
17    - --filesystem=xdg-run/gnupg:ro
18  # We're waiting for webcam support:
      https://github.com/flatpak/xdg-desktop-portal/issues/38;
19  # we use --device=all meanwhile. We do network, because we're
      opening a port.
20  build-options:
21    cflags: -O2 -g
22    cxxflags: -O2 -g
23    env:
24      V: '1'
25  cleanup:
26    - /include
27    - /lib/pkgconfig
28    - /share/pkgconfig
29    - /share/aclocal
30    - /man
31    - /share/man
32    - /share/gtk-doc
33    - "*.la"
34    - "*.a"
35  modules:
36    - name: pycairo
37      buildsystem: simple
38      ensure-writable:
39        - easy-install.pth
40        - setuptools.pth
41      build-commands:
42        - python3 setup.py build
43        - python3 setup.py install
            --prefix=${FLATPAK_DEST}
44      sources:
45        - type: archive
46          url: https://github.com/pygobject/pycairo/releases/
               download/v1.13.4/pycairo-1.13.4.tar.gz
47          sha256: ae1eb50d4b61167cfde585261d93fde6ecda08f0f6c0
               136d3cf92abc5eb893ed
48
```

```
49    - name: pygobject
50      build-options:
51        env:
52          PYTHON: python3
53      ensure-writable:
54        - easy-install.pth
55        - setuptools.pth
56      sources:
57        - type: archive
58          url:
                http://ftp.acc.umu.se/pub/GNOME/sources/pygobject/
                3.28/pygobject-3.28.2.tar.xz
59          sha256: ac443afd14fcb9ff5744b65d6e2b380e70510278404f
                b8684a9b9fb089e6f2ca
60
61  # [...] Long list of dependency cropped for readability
62
63    - name: gnome-keysign
64      no-autogen: true
65      buildsystem: simple
66      build-commands:
67        - pip3 install --prefix=${FLATPAK_DEST} .
68      sources:
69        - type: archive
70          url: https://gitlab.gnome.org/GNOME/gnome-keysign/-/
                archive/0.9.9/gnome-keysign-0.9.9.tar.gz
71          sha256: e4b32675664a32bef9386f2f6bc187c0506f54a1c6ce
                ff5b07377f73a85e3e39
```

Manifest Flatpak per GNOME-Keysign

Per poter comporre una email allegandoci la signature creata alla fine del processo di firma Keysign utilizzava xdg-email. Non essendo utilizzabile all'interno della sandbox di Flatpak si è implementata la composizione di una email utilizzando il portale *org.freedesktop.portal.Email*. Oltre all'utilizzo vero e proprio del portale è stato necessario anche modificare il path della signature che viene passato al client email, perché all'interno della sandbox l'applicazione vedrà il path come */var/tmp/signature.asc* ma questo è un

path relativo alla sandbox e visto che il client email che verrà utilizzato sarà eseguito sul sistema host, in /var/tmp non sarà presente tale file. Fortunatamente */var/tmp/* è l'unica cartella speciale che è visibile anche all'esterno della sandbox ma solo se acceduta dal suo path "reale". Per questo motivo se Keysign si rende conto di essere all'interno di una sandbox modificherà il path dell'allegato prima di lanciare il client email.

```
def _email_portal(to, subject=None, body=None, files=None):
    # The following two checks are to ensure Python 2 compatibility
    if not hasattr(os, 'O_PATH'):
        os.O_PATH = 2097152
    if not hasattr(os, 'O_CLOEXEC'):
        os.O_CLOEXEC = 524288
    name = "org.freedesktop.portal.Desktop"
    path = "/org/freedesktop/portal/desktop"
    bus = dbus.SessionBus()
    try:
        proxy = bus.get_object(name, path)
    except dbus.exceptions.DBusException:
        log.debug("Desktop portal is not available")
        return None
    iface = "org.freedesktop.portal.Email"
    email = dbus.Interface(proxy, iface)
    parent_window = ""
    attrs = []
    if files:
        for file in files:
            fd = os.open(file, os.O_PATH | os.O_CLOEXEC)
            attrs.append(dbus.types.UnixFd(fd))
    opts = {"subject": subject, "address": to, "body": body,
        "attachment_fds": attrs}
    try:
        ret = email.ComposeEmail(parent_window, opts)
        return ret
    except TypeError:
        log.debug("Email portal is not available")
        return None

def _using_flatpak():
    """Check if we are inside flatpak"""
    return os.path.exists("/.flatpak-info")


def _fix_path_flatpak(files):
    """In Flatpak the only special path visible also from outside is
        /var/tmp/ . To be able to use the files from the host we change
        the path to the absolute one. This fix in the future may not be
        necessary because the portals should be able to automatically
        handle it."""
    part_1 = os.path.expanduser("~/.var/app/")
    app_id = "org.gnome.Keysign"
```

```
40      part_2 = "cache/tmp/"
41      flatpak_path = os.path.join(part_1, app_id, part_2)
42      fixed_files = []
43      if files:
44          for file in files:
45              base = os.path.basename(file)
46              new_path = os.path.join(flatpak_path, base)
47              shutil.move(file, new_path)
48              fixed_files.append(new_path)
49      return fixed_files
50
51
52  def send_email(to, subject=None, body=None, files=None):
53      """Tries to send the email using firstly the portal, then the
            xdg-email and as a last attempt the mailto uri"""
54      if _using_flatpak():
55          files = _fix_path_flatpak(files)
56
57      if _email_portal(to, subject, body, files):
58          return
59
60      try:
61          _email_file(to=to, subject=subject, body=body, files=files)
62          return
63      except FileNotFoundError:
64          log.debug("xdg-email is not available")
65
66      if _email_mailto(to, subject, body, files):
67          return
68
69      log.error("An error occurred trying to compose the email")
```

Utilizzo del portale per inviare le email

6.5 PKGBUILD per Arch Linux

Per permettere agli utenti della distribuzione Arch Linux di poter installare più facilmente GNOME-Keysign, è stato creato il relativo PKGBUILD postandolo nell'Arch User Repository (AUR) [5]

La sua struttura, tra le altre cose, prevede la lista delle dipendenze divise in hard, cioè le obbligatorie, opzionali e quelle necessarie solo durante la fase di build. Inoltre il PKGBUILD prevede due fasi principali: il *build* in cui ad esempio vengono estratte le traduzioni delle stringhe presenti sia

[5] `https://aur.archlinux.org/packages/gnome-keysign/`

per l'interfaccia grafica che per il file desktop, e il *package* in cui avviene l'installazione vera e propria.

```
pkgname=gnome-keysign
pkgver=0.9.9
pkgrel=1
pkgdesc="An easier way to sign OpenPGP keys over the local
    network or Bluetooth."
arch=('any')
url="https://github.com/gnome-keysign/gnome-keysign"
license=('GPL3')
install=gnome-keysign.install
depends=('python' 'python-cairo' 'python-dbus'
    'python-future' 'python-gobject' 'python-gpgme'
    'python-qrcode' 'python-requests' 'python-twisted' 'avahi'
    'dbus' 'gst-plugins-good' 'gst-plugins-bad'
    'magic-wormhole' 'zbar')
optdepends=('python-pybluez: Bluetooth support')
makedepends=('git' 'python-setuptools' 'python-lxml')
_commit=2a67b564057ceb7e4a9cb20b1d15669b80ce16d3 #
    tags/0.9.9
source=("git+https://gitlab.gnome.org/GNOME/
         gnome-keysign.git#commit=$_commit")
sha256sums=('SKIP')

build() {
    cd $pkgname
    python setup.py build
}


package() {
    cd $pkgname
    python setup.py install --root="${pkgdir}"
        --prefix="/usr" --optimize=1
}
```

PKGBUILD di GNOME-Keysign

6.6 Applicazione companion

La schermata principale dell'applicazione companion è mostrata in figura 6.5. Per importare una firma gli utenti dovranno fare un drag and drop del file, oppure dovranno selezionarlo con un classico file chooser. Una volta scelta la signature verrà visualizzata una schermata con il risultato dell'operazione di importazione. Se gli utenti avessero ulteriori chiavi da importare potranno utilizzare il pulsante *back* in alto a sinistra per tornare alla pagina principale e ripetere l'operazione con le successive chiavi.

Per l'operazione di importazione è stata utilizzata la libreria *python-gpgme* come mostrato nello snippet di codice seguente.

```
1  import gpg
2
3  def on_drag_data_received(self, widget, drag_context, x, y, data, info,
       time):
4      filename = data.get_text()
5      logger.info("Received file: %s" % filename)
6      # remove file://, \r\n and NULL
7      filename = filename[7:].strip('\r\n\x00')
8      try:
9          result = self.import_key(filename)
10     except gpg.errors.GPGMEError as e:
11         logger.error(e)
12         result = "An error occurred, please try again"
13     self.go_to_result(result)
14
15 @staticmethod
16 def import_key(filename):
17     ctx = gpg.Context()
18     with open(filename, "rb") as fh:
19         decrypted = ctx.decrypt(fh)
20     ctx.op_import(decrypted[0])
21     result = ctx.op_import_result()
22     if len(result.imports) > 0:
23         return "Signed key successfully imported"
24     else:
25         raise gpg.errors.GPGMEError
```

Importazione di una firma

Alternativamente era possibile utilizzare le API offerte da Seahorse tramite D-Bus. La principale differenza tra i due metodi è che con il primo è direttamente l'applicazione che importa la firma nel keyring dell'utente, invece con il secondo si demanda questa operazione a Seahorse. Il vantaggio nell'utilizzare Seahorse è il fatto di poter importare chiavi senza dover

richiedere il permesso di scrittura sulla cartella *gnupg* del sistema, visto che sarà Seahorse a provvedere all'importazione. Per contro però l'applicazione sarebbe resa dipendente da un programma terzo.

Per questo motivo attualmente si è scelto di usare *python-gpgme*, ma in futuro si potrebbe anche prevedere di utilizzare entrambi i metodi e utilizzare il più appropriato in base alle circostanze.

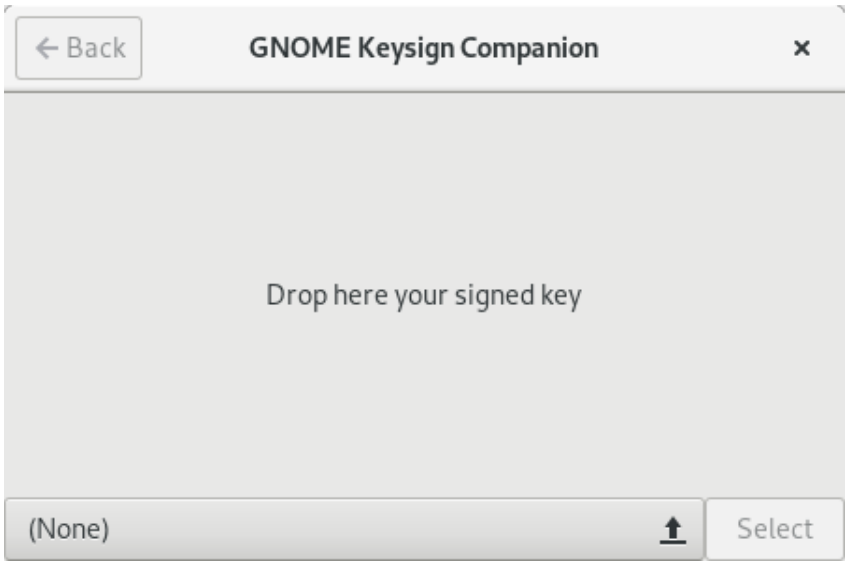

Figura 6.5: Schermata principale di GNOME-Keysign-companion.

Capitolo 7

Testing

7.1 Test automatici

GNOME-Keysign per i propri test utilizza la libreria nose[1] di Python. Per questo motivo anche per i nuovi test relativi alle funzionalità sviluppate in questa tesi è stato utilizzato nose.

Per quanto riguarda il trasferimento tramite Magic Wormhole sono stati previsti tre tipi di test:

- Un trasferimento corretto in cui il ricevente inserisce il codice Wormhole mostrato lato mittente. In questo scenario si controlla che i dati ricevuti debbano essere uguali a quelli di partenza.
- Anche se attualmente su Keysign il codice Wormhole viene generato random lato server, dalle specifiche è prevista anche la possibilità di utilizzare un codice custom concordato dagli utenti in offline. In questo secondo test si controlla la correttezza di un trasferimento utilizzando un codice Wormhole generato in offline.
- Il tentativo di utilizzo di un codice errato, controllando che effettivamente non venga inizializzato il trasferimento della chiave.

```
1 @deferred(timeout=10)
2 @inlineCallbacks
3 def test_wrmhl():
```

[1] https://nose.readthedocs.io/en/latest/testing.html

```
4       data = read_fixture_file("seckey-no-pw-1.asc")
5       key = openpgpkey_from_data(data)
6       file_key_data = get_public_key_data(key.fingerprint)
7       log.info("Running with key %r", key)
8       # Start offering the key
9       offer = WormholeOffer(key)
10      info = yield offer.allocate_code()
11      code, _ = info
12      offer.start()
13      receive = WormholeReceive(code)
14      msg_tuple = yield receive.start()
15      downloaded_key_data, success, _ = msg_tuple
16      assert_true(success)
17      log.info("Checking with key: %r", downloaded_key_data)
18      assert_equal(downloaded_key_data, file_key_data)
19
20
21  @deferred(timeout=10)
22  @inlineCallbacks
23  def test_wrmhl_offline_code():
24      data = read_fixture_file("seckey-no-pw-1.asc")
25      key = openpgpkey_from_data(data)
26      file_key_data = get_public_key_data(key.fingerprint)
27      # We assume that this channel, at execution time, is free
28      code = "5556-penguin-paw-print"
29      # Start offering the key
30      offer = WormholeOffer(key)
31      offer.allocate_code(code)
32      offer.start()
33      receive = WormholeReceive(code)
34      msg_tuple = yield receive.start()
35      downloaded_key_data, success, _ = msg_tuple
36      assert_true(success)
37      log.info("Checking with key: %r", downloaded_key_data)
38      assert_equal(downloaded_key_data, file_key_data)
39
40
41  @deferred(timeout=10)
42  @inlineCallbacks
43  def test_wrmhl_wrong_code():
44      data = read_fixture_file("seckey-no-pw-1.asc")
45      key = openpgpkey_from_data(data)
46      log.info("Running with key %r", key)
47      # Start offering the key
48      offer = WormholeOffer(key)
49      info = yield offer.allocate_code()
50      code, _ = info
51      offer.start()
52      receive = WormholeReceive(code+"-wrong")
53      msg_tuple = yield receive.start()
54      downloaded_key_data, success, message = msg_tuple
55      assert_false(success)
56      assert_is_not_none(message)
```

```
57      assert_equal(message, WrongPasswordError)
```

Testing per Magic Wormhole

I test relativi al trasferimento con Bluetooth sono quattro:

- Verificare la correttezza di un trasferimento tramite Bluetooth in cui il ricevente scansiona il QR code.
- Utilizzare un HMAC errato, come nel caso in cui qualcuno cerchi di sostituire la chiave corretta. In questo caso si controlla che effettivamente l'HMAC ricevuto non corrisponda a quello calcolato e che la chiave scaricata venga rigettata.
- Verificare il fallito tentativo di download quando l'host che si cerca di contattare non è disponibile.
- Simulare una connessione non stabile con il caso in cui venga ricevuta una chiave non completa.

```
1   @deferred(timeout=15)
2   @inlineCallbacks
3   @unittest.skipUnless(HAVE_BT, "requires bluetooth module")
4   def test_bt():
5       """This test requires two working Bluetooth devices"""
6       data = read_fixture_file("seckey-no-pw-1.asc")
7       key = openpgpkey_from_data(data)
8       file_key_data = get_public_key_data(key.fingerprint)
9       log.info("Running with key %r", key)
10      hmac = mac_generate(key.fingerprint.encode('ascii'), file_key_data)
11      # Start offering the key
12      offer = BluetoothOffer(key)
13      data = yield offer.allocate_code()
14      # getting the code from "BT=code;...."
15      code = data.split("=", 1)[1]
16      code = code.split(";", 1)[0]
17      port = int(data.rsplit("=", 1)[1])
18      offer.start()
19      receive = BluetoothReceive(port)
20      msg_tuple = yield receive.find_key(code, hmac)
21      downloaded_key_data, success, _ = msg_tuple
22      assert_true(success)
23      log.info("Checking with key: %r", downloaded_key_data)
24      assert_equal(downloaded_key_data.encode("utf-8"), file_key_data)
25
26
27  @deferred(timeout=15)
28  @inlineCallbacks
```

```
29 @unittest.skipUnless(HAVE_BT, "requires bluetooth module")
30 def test_bt_wrong_hmac():
31     """This test requires two working Bluetooth devices"""
32     data = read_fixture_file("seckey-no-pw-1.asc")
33     key = openpgpkey_from_data(data)
34     log.info("Running with key %r", key)
35     hmac = "wrong_hmac_eg_tampered_key"
36     # Start offering the key
37     offer = BluetoothOffer(key)
38     data = yield offer.allocate_code()
39     # getting the code from "BT=code;...."
40     code = data.split("=", 1)[1]
41     code = code.split(";", 1)[0]
42     port = int(data.rsplit("=", 1)[1])
43     offer.start()
44     receive = BluetoothReceive(port)
45     msg_tuple = yield receive.find_key(code, hmac)
46     downloaded_key_data, success, _ = msg_tuple
47     assert_false(success)
48
49
50 @deferred(timeout=15)
51 @inlineCallbacks
52 @unittest.skipUnless(HAVE_BT, "requires bluetooth module")
53 def test_bt_wrong_mac():
54     """This test requires one working Bluetooth device"""
55     receive = BluetoothReceive()
56     msg_tuple = yield receive.find_key("01:23:45:67:89:AB", "hmac")
57     downloaded_key_data, success, error = msg_tuple
58     assert_is_none(downloaded_key_data)
59     assert_false(success)
60     assert_equal(error.args[0], "(112, 'Host is down')")
61
62
63 @deferred(timeout=15)
64 @inlineCallbacks
65 @unittest.skipUnless(HAVE_BT, "requires bluetooth module")
66 def test_bt_corrupted_key():
67     """This test requires two working Bluetooth devices"""
68
69     @inlineCallbacks
70     def start(bo):
71         success = False
72         try:
73             while not success:
74                 ready_to_read, ready_to_write, in_error = yield
                      threads.deferToThread(
75                     select.select, [bo.server_socket], [], [], 0.5)
76                 if ready_to_read:
77                     client_socket, address = bo.server_socket.accept()
78                     key_data = get_public_key_data(bo.key.fingerprint)
79                     kd_decoded = key_data.decode('utf-8')
80                     # We send only a part of the key. In this way we can
```

```
                    simulate the case
                # where the connection has been lost
                half = len(kd_decoded)/2
                kd_corrupted = kd_decoded[:half]
                yield threads.deferToThread(client_socket.sendall,
                    kd_corrupted)
                client_socket.shutdown(socket.SHUT_RDWR)
                client_socket.close()
                success = True
        except Exception as e:
            log.error("An error occurred: %s" % e)

    data = read_fixture_file("seckey-no-pw-1.asc")
    key = openpgpkey_from_data(data)
    log.info("Running with key %r", key)
    file_key_data = get_public_key_data(key.fingerprint)
    hmac = mac_generate(key.fingerprint.encode('ascii'), file_key_data)
    # Start offering the key
    offer = BluetoothOffer(key)
    data = yield offer.allocate_code()
    # getting the code from "BT=code;...."
    code = data.split("=", 1)[1]
    code = code.split(";", 1)[0]
    port = int(data.rsplit("=", 1)[1])
    start(offer)
    receive = BluetoothReceive(port)
    msg_tuple = yield receive.find_key(code, hmac)
    downloaded_key_data, result, error = msg_tuple
    assert_false(result)
    assert_equal(type(error), ValueError)
```

Testing per Bluetooth

7.2 Test manuali

Manualmente sono stati effettuati tutta una serie di test che non era possibile automatizzare. Per lo scopo è stato installato GNOME-Keysign su due computer e sono stati effettuati i seguenti test:

- Dopo aver collegato i due computer a reti WiFi separate si è tentato un trasferimento utilizzando il Bluetooth. In seguito si è disabilitato il Bluetooth dal sistema operativo e si è tentato nuovamente un trasferimento, questa volta utilizzando Magic Wormhole
- Scollegare entrambi i computer dalle reti WiFi ed effettuare un trasferimento con Bluetooth

- Tentare di utilizzare Wormhole, selezionando il pulsante *Internet*, con il computer scollegato da qualsiasi rete. In questo caso ci si aspettava di vedere una *infobar* che informasse l'utente dell'assenza di connessione ad Internet.

- Dopo aver avviato due istanze di Keysign in modalità invio si è tentato di immettere in rapida successione entrambi i codici di sicurezza in una terza istanza in modalità ricezione. Questo test ha permesso di verificare la bontà della funzione di stop, perché dopo l'inserimento del primo codice parte la ricerca e il download della chiave, ma prima che finisca viene immesso un secondo codice, anch'esso valido. Quindi il programma deve essere in grado di annullare l'operazione attualmente in corso e iniziare un tentativo di download con il nuovo codice.

Capitolo 8

Conclusioni

8.1 Futuri sviluppi

Attualmente l'applicazione companion può essere considerata completa relativamente alle funzioni che mette a disposizione. In futuro si potrebbe valutare nuovamente la possibilità di incorporarla direttamente nella principale GNOME-Keysign. Ad esempio aggiungendo un terzo tab in cui sarà possibile importare le firme ricevute, oppure inserirlo come una pagina aggiuntiva raggiungibile solo dopo aver inviato una propria chiave.

Inoltre, se due persone desiderano inviarsi email crittografate, non possono farlo fino a quando non si saranno scambiate le relative firme. Le email però non hanno una tempistica di invio affidabile e a volte ci possono volere diversi minuti, se non ore, prima di poterle ricevere. Si può pensare quindi di generare anche una firma locale con una durata breve, nell'ordine di qualche ora, per consentire ai due utenti di inviare email crittografate fin da subito.

Infine GNOME mette a disposizione un sistema di continuous integration (CI) [1] che, eseguendo automaticamente i test disponibili, si rivela essere molto utile per poter accorgersi dell'eventuale aggiunta di bug o regressioni nella scrittura di nuovo codice. Aggiungendo tale funzione al repository di GNOME-Keysign renderebbe più robusti gli sviluppi futuri, riducendo gli errori che potrebbero passare inosservati.

[1] `https://wiki.gnome.org/Projects/GnomeContinuous`

8.2 Valutazioni finali

Come presentato nella sezione 1.4 attualmente gli utenti si trovano davanti a diverse difficoltà nel caso in cui vogliano utilizzare PGP per firmare e criptare le proprie email. Nella sezione 2 è stato presentato il software GNOME-Keysign il cui obbiettivo è quello di semplificare l'intero procedimento di scambio chiavi e firma.

Considerando che molte persone potessero beneficiare dall'utilizzo di un software come GNOME-Keysign, si è deciso di analizzarlo più nel dettaglio per poterne identificare i punti in cui fosse possibile intervenire per migliorarlo, sia dal punto di vista della sua flessibilità e versatilità che per la sua user-friendliness.

Per quanto riguarda il trasferimento è stato rilevato che il solo utilizzo di un server locale potesse risultare limitate in diverse occasioni e quindi si è cercato di migliorare la situazione aggiungendo metodi di trasferimento aggiuntivi quali Bluetooth e Magic Wormhole. Con questa aggiunta attualmente GNOME-Keysign è diventato utilizzabile anche quando due computer siano collegati a reti WiFi di tipo guest o, grazie al Bluetooth, anche in totale assenza di rete. Le modifiche apportate sopra elencate sono state incluse upstream e rese disponibili agli utenti finali a partire dalla versione 0.9.5[2] per quanto riguarda il Bluetooth e dalla 0.9.9[3] per Magic Wormhole.

Inoltre si è constatato che GNOME-Keysign conclude le sue funzioni con l'invio della firma tramite email, lasciando così agli utenti il compito di importare le signature ricevute utilizzando la riga di comando o programmi terzi. Allo scopo di ottenere una esperienza utente più completa si è quindi creata una applicazione companion che permettesse agli utenti anche di importare le chiavi una volta ricevute.

Infine avendo constatato l'attuale difficoltà per gli utenti nel reperire l'ultima versione del software si è provveduto a rilasciare GNOME-Keysign nei repository della distribuzione Arch Linux[4] e si è realizzato un Flatpak dell'applicazione rendendolo disponibile successivamente su Flathub[5]. Su Flathub, avendo a disposizione il numero di download effettuati, è stato possibile, a posteriori, verificare l'effettivo beneficio nell'aver rilasciato l'ap-

[2] `https://github.com/gnome-keysign/gnome-keysign/releases/tag/0.9.5`
[3] `https://github.com/gnome-keysign/gnome-keysign/releases/tag/0.9.9`
[4] `https://aur.archlinux.org/packages/gnome-keysign/`
[5] `https://flathub.org/apps/details/org.gnome.Keysign`

plicazione anche tramite questo canale. Il rilascio di GNOME-Keysign su Flathub è avvenuto verso la metà di Maggio 2018 e, ai primi di Ottobre dello stesso anno, il numero di download è stato di circa 400. Prendendo atto della gioventù sia di Keysign che di Flathub (attivo solo da Aprile 2017), questo può essere considerato un risultato più che soddisfacente. Per il futuro ci si aspetta che i numeri di GNOME-Keysign possano aumentare, seguendo il trend di incremento nell'utilizzo generale di Flatpak.

Solo avendo a disposizione un programma facile da installare, usare e che funzioni in ogni occasione gli utenti saranno invogliati a creare firme per le persone che conoscono e incontrano, riuscendo così ad aumentare il *Web of Trust*.

Ringraziamenti

Ringrazio la mia famiglia per essermi sempre stata vicino ed avermi sostenuto durante tutto il percorso universitario. Ivi compreso Ciak, il mio affettuosissimo Labrador.

Ringrazio gli amici che ho conosciuto in questa facoltà con cui ho potuto condividere la passione per l'informatica.

Ringrazio tutta la GNOME community per avermi accettato al GSoC e permesso di lavorare a questo progetto. In particolare ringrazio Tobias Mueller per avermi seguito durante tutto lo svolgimento di questo progetto di tesi, dispensando sempre ottimi consigli.

Infine ringrazio il prof. Mirko Viroli per l'aiuto e il tempo dedicatomi per la stesura della tesi.

Elenco delle figure

Bibliografia

[1] *gpg(1) - Linux man page*, Luglio 2018.

[2] Arch Linux contributors. Arch user repository. `https://wiki.archlinux.org/index.php/Arch_User_Repository`, 2018. [Online; controllata il 5-Settembre-2018].

[3] N. L. Biggs. *Codes: An Introduction to Information Communication and Cryptography*. Springer, 2008.

[4] B. Byfield. Google's summer of code concludes. `https://www.linux.com/news/googles-summer-code-concludes`, 2005. [Online; controllata il 16-Settembre-2018].

[5] Debian contributors. Keysigning. `https://www.debian.org/events/keysigning`, 2018. [Online; controllata il 27-Agosto-2018].

[6] Debian contributors. Request for package (rfp). `https://wiki.debian.org/RFP`, 2018. [Online; controllata il 5-Settembre-2018].

[7] Flatpak Developers. Flatpak - quick setup. `https://web.archive.org/web/20180826150105/https://www.flatpak.org/setup/`, 2018. [Online; controllata il 26-Agosto-2018].

[8] Flatpak Developers. Flatpak 1.0 released, ready for prime time. `https://www.flatpak.org/press/2018-08-20-flatpak-1.0/`, 2018. [Online; controllata il 5-Settembre-2018].

[9] B. S. I. Group. Bluetooth core specifications. `https://www.bluetooth.org/DocMan/handlers/DownloadDoc.ashx?doc_id=421043`, 2016.

[10] N. W. Group. Openpgp message format. `https://www.ietf.org/rfc/rfc4880.txt`, 2007.

[11] F. Karinthy. Chain-links. `https://djjr-courses.wdfiles.com/local--files/soc180%3Akarinthy-chain-links/Karinthy-Chain-Links_1929.pdf`, 1929.

[12] M. King. Pgp: Still hard to use after 16 years. `https://www.whitehatsec.com/blog/pgp-still-hard-to-use-after-16-years`, 2015. [Online; controllata il 15-Settembre-2018].

[13] R. Klafter and E. Swanson. Evil 32: Check your gpg fingerprints. `https://evil32.com`, 2014. [Online; controllata il 27-Agosto-2018].

[14] A. Laroia. Short key ids are bad news (with openpgp and gnu privacy guard). `http://www.asheesh.org/note/debian/short-key-ids-are-bad-news.html`, 2011. [Online; controllata il 27-Agosto-2018].

[15] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[16] Nordic Semiconductor. A short history of bluetooth. `https://www.nordicsemi.com/eng/News/ULP-Wireless-Update/A-short-history-of-Bluetooth`, 2014. [Online; controllata il 9-Settembre-2018].

[17] R. L. Rivest. *Handbook of Theoretical Computer Science*. Elsevier, 1990.

[18] S. Taylor. Google summer of code 2017 statistics part 2. `https://opensource.googleblog.com/2017/06/google-summer-of-code-2017-statistics.html`, 2017. [Online; controllata il 16-Settembre-2018].

[19] Ubuntu contributors. Gnuprivacyguardhowto. `https://help.ubuntu.com/community/GnuPrivacyGuardHowto`, 2018. [Online; controllata il 27-Agosto-2018].

[20] P. Wolfer. Snap support for peek screen recorder discontinued. `https://www.reddit.com/r/Ubuntu/comments/870bcn/snap_support_for_peek_screen_recorder_discontinued/`, 2018. [Online; controllata il 9-Settembre-2018].

[21] P. Zimmermann. Why i wrote pgp. `https://www.philzimmermann.com/EN/essays/WhyIWrotePGP.html`, 1991.