**COME CREARE UN TOOL DI STEGANOGRAFIA?**

Per creare un tool di steganografia bisogna:

1. Leggere il file originale e trasformarlo in un formato adatto
2. Codificare il messaggio
3. Incorporare il messaggio nel file originale
4. Salvare il file modificato
5. In un secondo momento, estrarre il messaggio dal file

La codifica del messaggio avviene a seconda del formato del file originale. Nel nostro progetto utilizzeremo un’immagine come formato e useremo la tecnica dei bit meno significativi, in cui è necessario che sia il file che il messaggio siano in formato binario. Stiamo, di fatto, nascondendo il messaggio nei pixel dell’immagine. In un file .jpg, presumibilmente un’immagine a colori RGB, ogni pixel è codificato in 24 bit: 8 bit definiscono la gradazione del rosso, 8 del verde e 8 del blu.

Bisogna:

1. Suddividere il messaggio in segmenti
2. Per ogni segmento, esso dev’essere convertito in binario
3. Per ogni blocco di pixel, sostituire i bit meno significativi dei pixel con i bit del segmento (eventualmente si possono modificare pixel successivi)
4. Ripetere il processo fin quando tutti i segmenti del messaggio sono stati inseriti

I bit meno significativi sono le cifre che si trovano a destra del numero, cioè quelle che hanno meno peso nella gradazione del colore del pixel, per cui una loro modifica causa solo una leggera variazione di colore rispetto all’immagine originale. Inoltre, di solito si prendono le ultime due cifre come quelle meno significative, per cui bisogna suddividere il messaggio in segmenti da 2 bit ciascuno.

Come faccio a stabilire quali pixel devono essere modificati? Si possono usare vari metodi:

* Modificare pixel in modo casuale. Il problema è che il messaggio è più difficile da decodificare, perché non si conoscono a priori le coordinate dei pixel modificati (perché appunto sono stati scelti casualmente). Inoltre, c’è il rischio che siano modificati pixel in cui c’è poca variazione di colore, e quindi il pixel modificato salterebbe facilmente all’occhio.
* Metodo di dispersione, in cui si selezionano dei pixel predefiniti distribuiti in maniera omogenea. In questo modo la decodifica è più semplice e l’immagine risulta più simile all’originale. Tuttavia, garantisce un livello di sicurezza più basso rispetto agli altri due metodi.
* Metodo di selezione di pixel in base al contenuto, scegliendo pixel che, se modificati, non causerebbero una grande variazione dell’immagine originale. Il problema è che è un metodo molto costoso e difficilmente implementabile, se non tramite appositi algoritmi di AI.

Nel nostro tool sarà usato il secondo metodo.

La realizzazione del tool prevede un approccio bottom-up, in cui ciascuna funzionalità è scomposta in funzioni più piccole. Inoltre, si segue uno sviluppo incrementale, ossia si parte dallo sviluppo di un prototipo per poi estendere man mano le funzionalità.

Tutti i procedimenti eseguiti sono documentati a partire dalla pagina seguente.

**LEGGERE IL FILE ORIGINALE E TRASFORMARLO IN UN FORMATO ADATTO**

La prima cosa da fare è la lettura del file originale (nel nostro caso un’immagine .jpg) e trasformarla in un file binario, perché bisogna modificare i bit del file. In particolare, vogliamo un file contenente i pixel (ossia i 3 byte per ciascuno) che formano la foto. L’obiettivo è quindi di memorizzare in una matrice la griglia dei pixel dell’immagine.

In Python la libreria più usata per la manipolazione delle foto è PIL (Python Image Library). Bisogna usare il metodo open(path) della classe Image in modo da memorizzare l’immagine all’interno di una variabile:

img = Image.open(path)

L’oggetto img ha gli attributi height e width (rispettivamente l’altezza e la larghezza dell’immagine in pixel) e il metodo getpixel((x, y)) che restituisce i canali RGB del pixel di coordinate (x, y), dove x è la coordinata orizzontale e y la coordinata verticale.

r, g, b = img.getpixel((x, y))

Non ci resta che unire i canali RGB per ottenere i 3 byte. Per farlo, shifto r di 16 bit, g di 8 bit e infine sommo r, g e b. Se li avessi sommati senza shiftarli sarebbe stato un errore, perché non avrei avuto 24 bit e il pixel non sarebbe stato valido.

pixel\_value = (r << 16) + (g << 8) + b

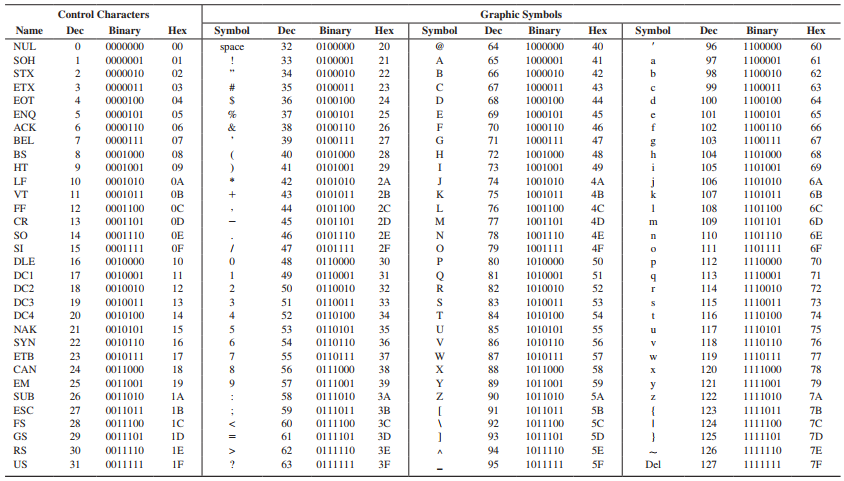
Utilizzando iterativamente questo metodo, è quindi possibile ottenere una griglia di pixel che dovrà essere memorizzata in una matrice.

**CODIFICARE IL MESSAGGIO**

Una volta ottenuta la griglia dei pixel dell’immagine, bisogna codificare il messaggio da nascondere. Quindi, si deve trasformare il messaggio (eventualmente criptato) in una stringa binaria. Per farlo usiamo il metodo seguente:

bin = message.encode('utf-8')

Questo metodo trasforma ogni carattere della stringa nel corrispondente codice ASCII. Infatti, ogni carattere corrisponde a un certo numero, come si può osservare dalla tabella ASCII:



Questa corrispondenza carattere-numero è necessaria per tutti gli elaboratori, perché al livello del linguaggio macchina si possono memorizzare e processare solo dati in formato binario.

Da notare che il metodo utilizzato in precedenza non restituisce il corrispondente numero binario per ogni carattere della stringa, ma restituisce quello in sistema decimale. Quindi c’è bisogno di convertire ogni valore ottenuto in binario:

bin = message.encode('utf-8')

encodedMessage = list()

    for byte in bin:

        encodedMessage.append('0' + format(byte, 'b'))

        if len(encodedMessage[-1]) == 7:

            encodedMessage[-1] = '0' + encodedMessage[-1]

Per dividere ciascun carattere in quattro segmenti da due bit ciascuno, modifichiamo l’algoritmo in modo da aggiungere uno 0 all’inizio di ogni stringa binaria corrispondente a un carattere, così da ottenere stringhe formate da otto bit (e quindi divisibili per due). Per cui ‘abcd’ sarà convertito in [‘01100001’, ‘01100010’, 01100011’, ‘01100100’].

Se però la stringa corrispondente a un carattere inizia già con uno 0 (come per il carattere ‘?’), la funzione format() restituisce una stringa di 6 caratteri, non 7, perché ovviamente lo 0 preso come cifra più significativa è ignorato, per cui bisogna aggiungere due zeri all’inizio, non uno solo.

# INCORPORARE IL MESSAGGIO NEL FILE ORIGINALE

In questa fase bisogna inserire il messaggio all’interno del file, modificando i bit meno significativi dei pixel dell’immagine stessa.

Questo processo si suddivide in tre sottofasi:

1. Stabilire in che modo deve avvenire la segmentazione del messaggio
2. Stabilire quali pixel modificare, a seconda della tecnica usata
3. Inserire ciascun segmento del messaggio nell’immagine

Innanzitutto, si deve definire in che modo avviene la segmentazione del messaggio, ossia quanti segmenti realizzare e quanti bit devono esserci in ciascun segmento.

Dato N il numero di caratteri del messaggio, bisogna inserire N\*8 bit all’interno dell’immagine. Poiché bisogna modificare le due cifre meno significative del pixel, è necessario dividere il messaggio in (N\*8)/2 = N\*4 segmenti da due bit ciascuno.

Ad esempio, se un messaggio è formato da 20 caratteri, vuol dire che è codificato in 160 bit ed è frammentato in 80 segmenti da due bit ciascuno. Inoltre, un singolo carattere corrisponde a quattro frammenti sequenziali.

Seguendo i calcoli fatti in precedenza, bisogna inserire N\*4 segmenti nell’immagine: vuol dire che devono essere modificati N\*4 pixel.

Occorre quindi verificare che l’immagine sia abbastanza grande da permettere l’inserimento del messaggio, ossia bisogna controllare che dati M pixel che formano l’immagine, si verifichi che M >= N\*4. È evidente che più N è piccolo, maggiore sarà l’efficacia della steganografia, perché le modifiche apportate sarebbero notevolmente meno visibili.

Per distribuire uniformemente i segmenti fra i vari pixel, bisogna definire il ratio (numero pixel totali)/(numero segmenti) ossia M/N\*4. In questo modo si stabilisce quanti pixel “saltare” fino a raggiungere quello che dev’essere modificato.

È importante notare che questo metodo è dinamico, cioè si adatta alle dimensioni dell’immagine di partenza, per cui è tendenzialmente più efficace rispetto a una distribuzione statica dei segmenti.

Un’altra nota da fare è che se M = N\*4 e quindi il ratio è 1, vuol dire che tutti i pixel dell’immagine devono essere modificati, rendendo il processo steganografico particolarmente evidente.

In Python:

ratio = (height\*width)//(len(message)\*4)

if ratio < 1:

print("ERRORE: il messaggio è troppo grande per poter essere inserito nell'immagine")

    return ratio

Una volta stabilito il ratio, bisogna memorizzarlo in un pixel all’interno dell’immagine in modo da poter ricostruire il messaggio in un secondo momento.

Si sceglie un pixel predefinito (nel nostro tool il primo) e si modificano gli ultimi tre bit, indicati come XYZ. Si nota che se il ratio è maggiore di 7, è impossibile memorizzare questo valore in 3 cifre binarie. Per cui si può procedere in un altro modo: le cifre X e Y stabiliscono effettivamente qual è il ratio, mentre la cifra X stabilisce se il ratio continua del pixel successivo oppure no.

Ad esempio, se il ratio è 12 (ossia 1100 in binario), allora esso verrà distribuito nei primi due pixel dell’immagine. Le ultime tre cifre del primo pixel sono 111 e quelle del secondo pixel sono 000. Questo perché le cifre YZ del primo pixel sono 11 e la cifra X è 1: vuol dire che il ratio continua nel secondo pixel, le cui cifre YZ sono 00 e la cifra X è 0, cioè la ricerca del ratio si interrompe. Concatenando le cifre YZ del primo pixel con le cifre YZ del secondo pixel si otterrà 1100 (12 in decimale), cioè il ratio.

In questo modo, quando bisogna estrarre il messaggio dall’immagine, il tool capisce quali sono i pixel modificati ed è in grado di ricostruire i segmenti nascosti.

Per definire quanti pixel devono essere modificati per inserire il ratio è sufficiente dividere per due il numero di cifre nel ratio convertito in binario. Se esso è dispari, occorre aggiungere uno 0 per renderlo pari.

In Python il codice per eseguire questo processo è il seguente:

binRatio = format(ratio, 'b')

if len(binRatio) % 2 == 1:

    binRatio = '0' + binRatio

pixelsToSet = len(binRatio)//2

    for i in range(pixelsToSet):

        newPixel = str(grid[0][i])

        newPixel = newPixel[:-3] + '1' + binRatio[i\*2:(i\*2)+2]

        grid[0][i] = newPixel

lastPixel = grid[0][pixelsToSet-1]

grid[0][pixelsToSet-1] = lastPixel[:-3] + '0' + lastPixel[-2:]

Una volta inserito il ratio all’interno dell’immagine, non resta che inserire il messaggio. Nella fase di codifica del messaggio, esso è stato trasformato in una matrice di stringhe binarie di cui ciascuna è corrispondente a un certo carattere. Bisogna quindi inserire due bit alla volta ogni N pixel, dove N è proprio il ratio.

I pixel devono essere modificati a partire dalla seconda riga della griglia, perché la prima riga è già stata utilizzata per memorizzare il ratio.

In Python:

gridRow, gridColumn = 1, 0

for i in range(len(message)):

    for j in range(4):

        segment = message[i]

        segment = segment[j\*2:(j\*2)+2]

        newPixel = str(grid[gridRow][gridColumn])

        newPixel = newPixel[:-2] + segment

        grid[gridRow][gridColumn] = newPixel

        gridColumn += ratio

        while gridColumn >= len(grid[0]):

            gridColumn -= len(grid[0])

            gridRow += 1

Mediante questo script si inseriscono i segmenti del messaggio in maniera omogenea all’interno dell’immagine. In generale, bisognerebbe evitare tre cicli iterativi interni, ma in questo caso l’aggiunta del terzo ciclo (quello in while) è necessaria, perché se il ratio è elevato bisogna saltare più righe della griglia, in modo da permettere una distribuzione omogenea dei segmenti. In caso contrario, i segmenti sarebbero disposti soltanto nelle prime righe dell’immagine, rendendo più evidente il processo steganografico.

Ora che il messaggio è stato nascosto all’interno dell’immagine, bisogna ricostruire l’immagine stessa per poterla salvare in memoria. Prima, però, bisogna porsi una domanda: quando bisogna ricostruire il messaggio, come si fa a capire quanti pixel analizzare?

Potremmo inserire la dimensione del messaggio (ossia il numero di segmenti, e quindi il numero di pixel modificati) subito dopo i pixel in prima riga dedicati alla memorizzazione del ratio, creando così una vera e propria “intestazione” del messaggio.

Esiste un altro metodo più semplice ed efficiente: poiché ogni carattere è convertito in 7 bit a cui noi aggiungiamo uno 0 all’inizio, vuol dire che i pixel multipli di 4 all’interno della successione dei pixel modificati hanno sempre ‘0’ come penultima cifra, proprio perché è il primo bit di un determinato carattere.

Per far capire all’algoritmo di estrazione quando termina una successione, possiamo inserire un ulteriore segmento di flag, successivo all’ultimo, che abbia un ‘1’ come penultimo bit. All’interno della successione, questo segmento è multiplo di 4, perché virtualmente i suoi ultimi 2 bit sono le prime due cifre binarie di un carattere.

L’algoritmo di estrazione sa che all’interno della successione ogni segmento multiplo di 4 deve sempre iniziare con ‘0’, per cui quando esamina il segmento di flag (che, ricordiamo, è multiplo di 4 nella successione) e scopre che inizia con ‘1’, capisce che deve fermarsi e che non rimane nessun altro segmento da esaminare all’interno dell’immagine.

Nel nostro tool sarà usato il secondo metodo.

In Python:

flagSegment = str(grid[gridRow][gridColumn])

grid[gridRow][gridColumn] = flagSegment[:-2] + '1' + flagSegment[-1]

La presenza del segmento di flag e dei segmenti di ratio influiscono sulle precondizioni per l’inserimento del messaggio nell’immagine. La nuova precondizione è:

(numero pixel immagine esclusa la prima riga)/(numero segmenti messaggio + quattro segmenti) >= 1.

I quattro segmenti da aggiungere al denominatore corrispondono al carattere di flag, che inizia con ‘1’.

In Python:

ratio = ((height-1)\*width)//((len(message)\*4)+4)

**SALVARE IL FILE MODIFICATO**

In questa fase bisogna ricomporre l’immagine a partire dai pixel della griglia, per poi salvarla in memoria.

Per farlo, bisogna creare un nuovo oggetto immagine e inserire in esso i pixel, uno dopo l’altro, 8 bit alla volta, mediante il metodo putdata().

In Python:

img = Image.new("RGB", (width, height))

binaryList = list()

for i in range(height):

    for j in range(width):

        pixel = str(grid[i][j])

        binaryList.append(int(pixel[:8], 2))

        binaryList.append(int(pixel[8:16], 2))

        binaryList.append(int(pixel[16:], 2))

img.putdata([(r, g, b) for r, g, b in zip(binaryList[0::3], binaryList[1::3], binaryList[2::3])])

L’immagine dev’essere poi salvata grazie al metodo .save(path).

Nota: l’immagine dev’essere salvata in formato .png perché è un tipo di formato lossless, ossia senza perdita di dati. Se invece fosse salvata in .jpg l’algoritmo di estrazione del messaggio non funzionerebbe.

**ESTRAZIONE DEL MESSAGGIO DAL FILE**

Infine, occorre estrarre il messaggio dal file modificato. Bisogna:

1. Convertire l’immagine in una griglia di pixel
2. Estrapolare il ratio
3. Estrapolare i segmenti del messaggio
4. Ricomporre il messaggio

La conversione dell’immagine in una griglia è una funzionalità già implementata nella prima fase della steganografia.

Per estrapolare il ratio, è sufficiente esaminare i primi pixel della prima riga: le ultime due cifre definiscono il ratio, la terzultima cifra stabilisce, quando è 1, che il ratio continua nel pixel successivo.

Una volta ottenuto il ratio, bisogna convertirlo in sistema decimale.

In Python:

ratio = ""

for i in range(len(grid[0])):

    pixel = str(grid[0][i])

    ratio += pixel[-2:]

    if pixel[-3] == '0':

        break

return int(ratio, 2)

In seguito, si devono estrapolare i segmenti. Possiamo usare un array in cui ciascun elemento è formato da quattro segmenti da due bit, ossia un determinato codice ASCII. L’algoritmo di estrazione si interrompe quando un elemento aggiunto alla lista ha ‘1’ come bit iniziale: è il segmento di flag.

I segmenti del messaggio devono essere cercati a partire dalla seconda riga, perché la prima è occupata dai segmenti di ratio.

In Python:

gridRow, gridColumn = 1, 0

message = list()

while True:

    segment = ""

    for i in range(4):

        pixel = str(grid[gridRow][gridColumn])

        segment += pixel[-2:]

        gridColumn += ratio

        while gridColumn >= len(grid[0]):

            gridColumn -= len(grid[0])

            gridRow += 1

    if segment[0] == '1':

        break

    message.append(segment)

Non resta che convertire ciascun codice ASCII nel corrispondente carattere:

output = ''

for i in range(len(message)):

    binaryStr = message[i]

    output += chr(int(binaryStr, 2))

**CRITTOGRAFIA DEL MESSAGGIO**

Un messaggio nascosto all’interno di un’immagine è molto difficile da individuare per un ente esterno, ma con l’utilizzo di terminali con un’elevata potenza di calcolo, è possibile procedere per tentativi e nel giro di giorni (o anche settimane) si riesce ad individuare il pattern di distribuzione dei segmenti del messaggio, e quindi la sua ricombinazione.

Se però si utilizza la crittografia in combinazione con la steganografia, è molto difficile che l’ente esterno riesca ad intercettare la comunicazione perché anche se è riuscito a estrarre tutti i segmenti, mettendoli insieme si ottiene una stringa che sembra non avere senso, proprio perché è criptata.

Nel nostro tool, si è deciso di utilizzare anche la crittografia del messaggio, che avviene prima che esso sia convertito in binario.

In Python sarà utilizzato il modulo *cryptography* in modo da poter crittografare il messaggio tramite una chiave appositamente generata. La chiave deve quindi essere inserita all’interno dell’immagine, in modo che durante il processo di estrazione del messaggio, la chiave possa essere anch’essa estratta e poi utilizzata per il decoding.

Questo script è usato per ottenere il messaggio criptato:

key = Fernet.generate\_key()

fernet = Fernet(key)

encMessage = fernet.encrypt(message.encode())

La chiave dev’essere convertita in una stringa binaria per poi essere inserita all’interno dell’immagine. La posizione di inserimento della chiave è arbitraria, per cui decidiamo di porla in seconda riga. Per cui modifichiamo il codice della funzione *hideMessage* in modo da poter nascondere il messaggio a partire dalla riga successiva rispetto a quella modificata per la chiave (ossia la terza), e non dalla seconda, come era stato stabilito precedentemente.

Anche il calcolo del ratio dev’essere modificato, in quando adesso lo spazio disponibile per inserire il messaggio non è più (altezza\_immagine-1)\*larghezza\_immagine, ma sarà (altezza\_immagine-2)\*larghezza\_immagine perché la prima riga è usata per nascondere il ratio e la seconda riga per nascondere la chiave.

Per inserire la chiave usiamo lo stesso metodo che è stato utilizzato per il ratio, cioè modificando gli ultimi tre bit di N pixel consecutivi.

Per inserire i dati nell’immagine, bisogna quindi:

1. Generare una chiave
2. Criptare il messaggio utilizzando la chiave
3. Convertire la chiave e il messaggio criptato in utf-8
4. Inserire la chiave nella seconda riga, e poi inserire il messaggio criptato a partire dalla terza riga

Per estrarre un messaggio, invece, bisogna:

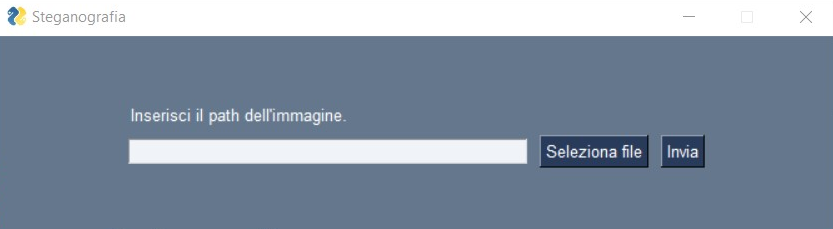
1. Estrarre i segmenti della chiave e fermarsi quando il terzultimo bit è 1
2. Estrarre i segmenti del messaggio e fermarsi al segmento di flag
3. Combinare la chiave e il messaggio, sotto forma di stringhe
4. Convertire sia chiave che messaggio in formato utf-8
5. Utilizzare la funzione di decoding del messaggio tramite la chiave

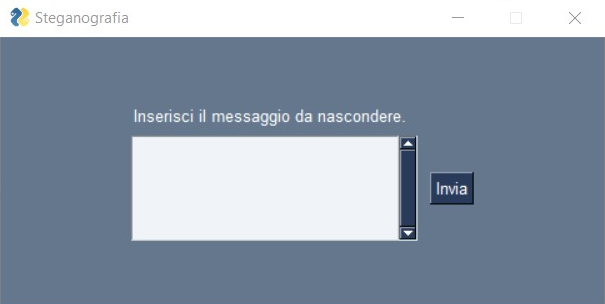
**INTERFACCIA GRAFICA**

Per rendere più piacevole l’utilizzo del tool di steganografia è opportuno realizzare un’interfaccia grafica. In questo modo, è possibile effettuare l’upload dell’immagine in cui nascondere/estrarre i dati senza scriverne il path, ma semplicemente selezionandolo tramite il file explorer del sistema operativo.

Per realizzare l’interfaccia grafica è stato utilizzato il modulo *PySimpleGUI* che permette di mostrare su schermo delle finestre di output e di poter inserire degli input mediante delle caselle di testo o cliccando sugli appositi bottoni.

Ecco alcuni esempi di finestre realizzabili con PySimpleGUI:







**UTILIZZO DEI FILE DI TESTO**

Fino a questo punto, il tool permette di nascondere/estrarre del testo all’interno di un’immagine. È utile aggiungere quindi due nuove funzionalità:

* Nascondere del testo di un file .txt
* Salvare il testo estratto dall’immagine in un file .txt

Per implementare la prima funzionalità è sufficiente utilizzare la seguente sintassi:

with open(values["Seleziona file"], "r") as file:

    message, key = cryptMessage(file.read())

In Python, il comando *with open(path) as file* permette di aprire il file presente nel percorso dato come parametro e di eseguire il codice indentato sottostante, per poi chiudere il file senza bisogno di esplicitare il comando close(). La lettura del contenuto di un file testuale avviene grazie alla funzione read().

Per implementare la seconda funzionalità il comando si utilizza la funzione write():

with open(path, "w") as file:

    file.write(decMessage)