Digital Green COVID-19 Certificate QR optimizer

Marco Carfizzi
Giacomo Arrigo

November 2021

1 Introduction

This document has the goal of describing our web application, while at the same time to provide information on how the users data are treated and utilized.

A demo is available at this link.

Our target is simple: to provide for a slenderer Digital Green COVID-19 Certificate QR code, so that it does a quality of life improvement in everyday life, while at the same time respecting the privacy of the users that choose to utilise our service.

A user can input their *DGCC* in the form of a QR code and download its optimized version.

The resulting encoding is compatible with the Italian official verification app, but could still work with other European versions.

2 Optimizing the QR code

The first idea for this app came from studying the QR code that is given to document a COVID-19 certification, like a vaccination, a test or a disease recovery.

For privacy reasons, only fake certificates (not signed with real key) are shown as examples in this documet. In Figure 1 we have a simulated green certification : it's clear just by a first look that this QR is really big and dense compared to usual bar-codes.

Our first goal was trying to decrease the density of the generated QR code, since this hinders its practical usage in everyday life: especially in cases where the certification is printed in low density or the paper has been crumpled, or when the testing mobile phone has not a flagship camera, the validity check of a digital certification can either be slow or take multiple tentatives to eventually work out. The density of this QR becomes an issue for example when this code is printed we noticed that the official verification app (*VerificaC19*) would struggle to get it right and multiple attempts were necessary to get it accepted and verified.

Figure 1: Original green pass.

Figure 2: Optimized green pass.

Our optimizations also help when the QR is shown via a smartphone screen, by reducing the needed time of the readings.

Of course, the information encoded inside the QR is composed by multiple attributes so one should not expect a micro QR similar compression; still, we wanted to try and see if the encoding could be optimized or not.

An example of the optimized QR found in Figure 1 can be seen in Figure 2. The smaller size of the latter is given just by looking at the size of the aligning squares: we have a 3x3 grid against a 4x4 grid in the former.

2.1 Error correction

The QR format as defined in ISO/IEC 18004:2015 is used to encode information into a 2D bar-code, consisting in a graphical matrix composed by black dots on a white background.

The QR code utilize the Reed–Solomon error correction code and provides with four different levels of it: this translates into the fact that a QR can be made redundant to a level where maximum 30% of its content can be restored if damaged.

An error correction rate of 'Q' (around 25%) is recommended by design specifications (https://ec.europa.eu/health/ehealth/covid-19_en). We noticed that using the minimum level 'L' (7% correction) the density of the QR code greatly decreases, as expected. The question that hit us was really simple: is it necessary to have an error correction level so high?

Our intuition is the following: since that to perform the correct signature validation we need the totality of the bits of the QR content to be intact, it's a waste of space to increase the redundancy of the code.

```
public static let supportedPrefixes = [
  "HC1:"
]

static func parsePrefix(_ payloadString: String, errors: ParseErrors?) -> String {
  var payloadString = payloadString
  var foundPrefix = false
  for prefix in Self.supportedPrefixes {
    if payloadString.starts(with: prefix) {
      payloadString = String(payloadString.dropFirst(prefix.count))
      foundPrefix = true
      break
    }
  }
  if !foundPrefix {
    errors?.errors.append(.prefix)
  }
  return payloadString
}
```

Figure 3: VerificaC19 iOS app Health Certificate version check.

The fact of having a QR that is read with more reliability overweights, in terms of system usability, the possibility to have missing portions of the code. A level '*L*' still gives some basic level of redundancy while at the same moment it provides for smaller and more readable barcodes.

Another big optimization comes from the encoding of the *base45* string not as 'bytes', but as an alphanumeric string.

Since, in fact, the encoding is performed on a string there is really no need to not treat it as an alphanumeric.

2.2 Health Certificate version check

Decoding the QR gives a string that starts with a "HCx:" prefix, where x is a positive natural number.

As shown in the code snippet in Figure 3, the Health Certificate version check is basically useless and passes even if the given payload has no "*HCx:*" prefix.

By stripping it from the payload we can save 4 characters.

3 Application description

The application is designed as a web app. The functionalities are provided by the local browser via basic JavaScript, along with the utilization of the Bootstrap

framework for the graphical part.

The app can also be easily deployed in desktop environments thanks to the electron framework.

We chose to utilize plain JS so that the app would run in a local environment: by operating in this way, no personal information is sent to the web server.

There is also support for a rapid validity check to prove the validity of the signature of the new generated QR code against the set of available public keys.

3.1 Architecture description

Main components list:

- ./public/
  - /lib/
  - *index.html*
  - *core.js*

The *lib* directory has all the external libraries needed to perform the computations. We decided to store them locally to depend the least possible on external servers.

The *index.html* file has the html components of the page. It's a basic page with a file input form and some hidden buttons.

The *core.js* has all the JavaScript functions.

There is a listener on any changes performed on the file input form: when this is called then the app tries to scan the received QR image.

If the scan gives a positive response the *optimize* function is called, else an error is thrown and the user can repeat the upload. Note that the file is not reaching anything on the server side and it's only needed for local computations.

For the decoding of the QR we used the qr-scanner js library.

If the input file is a valid QR, then it is decoded using the dcc-utils library from the official Italian Ministero della Salute GitHub repository.

If the decoded payload is a valid COVID-19 certificate then we go on, else an error is thrown.

We pass the decoded base45 string to a QR generation library, qr-code-styling that renders the 2D image into a canvas in the html page. We chose this library for the multiple customization options, in particular for the possibility of setting the alphanumerical mode.

When the QR has been generated and shown in the page, a download and a verify button are shown: the first allows to download the QR code as a *JPEG* image with a random name, then second allows to perform the verification of the signature over a list of given public keys.

If an optimized QR code is given as input, the computation will fail and the app will return an error. This is done as an input sanification pass, since the *dcc-utils* library does a proper check even on the 'HCx:' prefix.

3.2 Privacy concerns

We designed this application with a clear idea: no personal data would be stored in the server side.

This is why we decided to implement it with basic HTML and JavaScript: a basic, easy to use, lightweight, multi-platform design that would run in a local environment.

If the app wasn't stored in a server, it could be even possibly work offline.

To provide a better layer of transparency, the source code is available at this link.

3.3 How to use

A simple guide to use the application:

1. Open app
2. Click "Choose file" and upload correct DGCC QR code (the library will find a QR code even in a image with other stuff in it) - Figure 4
3. Wait until the optimized code is generated and shown in the page - Figure 5
4. Click "Download QR code" and choose where to save the JPEG file.

Verification success is depicted in Figure 6.

Verification error is depicted in Figure 7, while an example error message (in this case when a wrong image is uploaded) is shown in Figure 8.

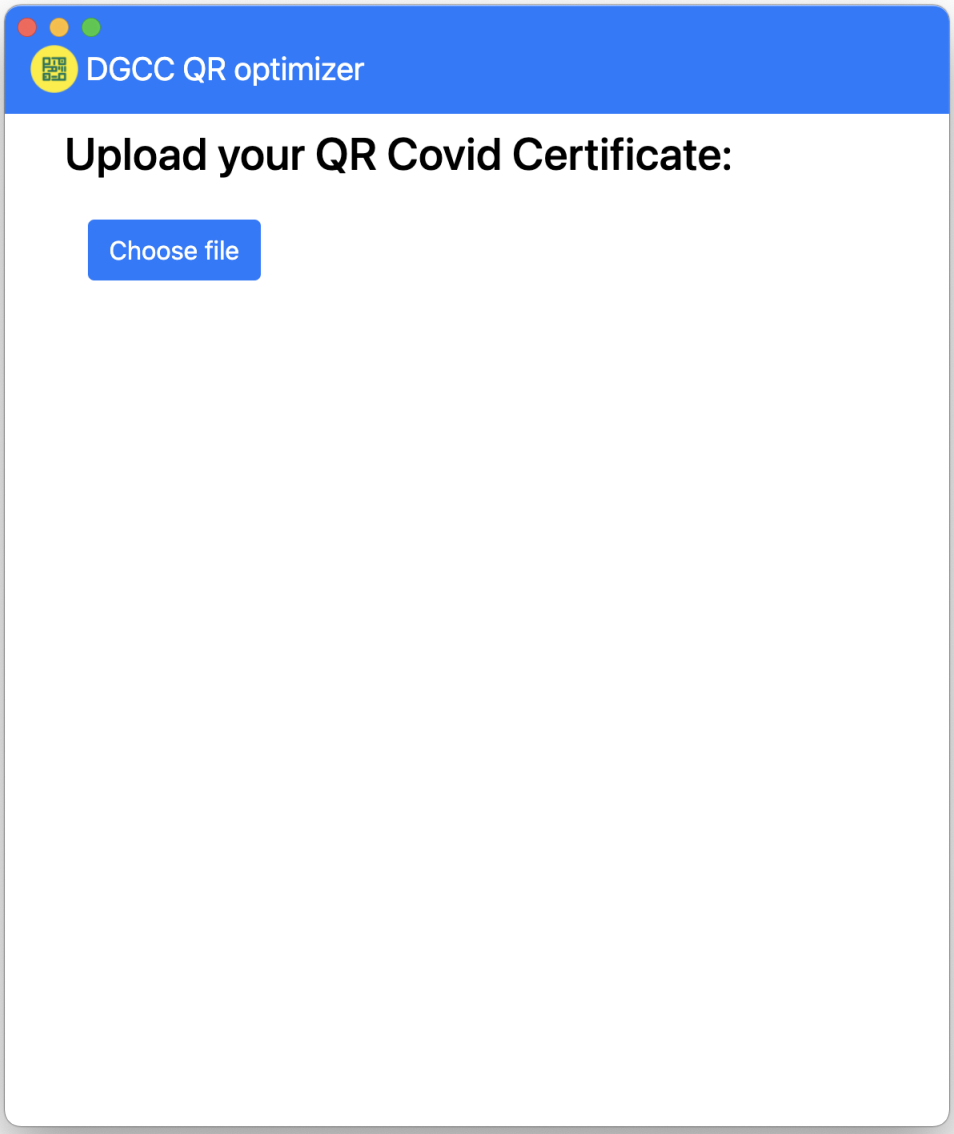

Figure 4: DGCC QR optimizer starting screen.

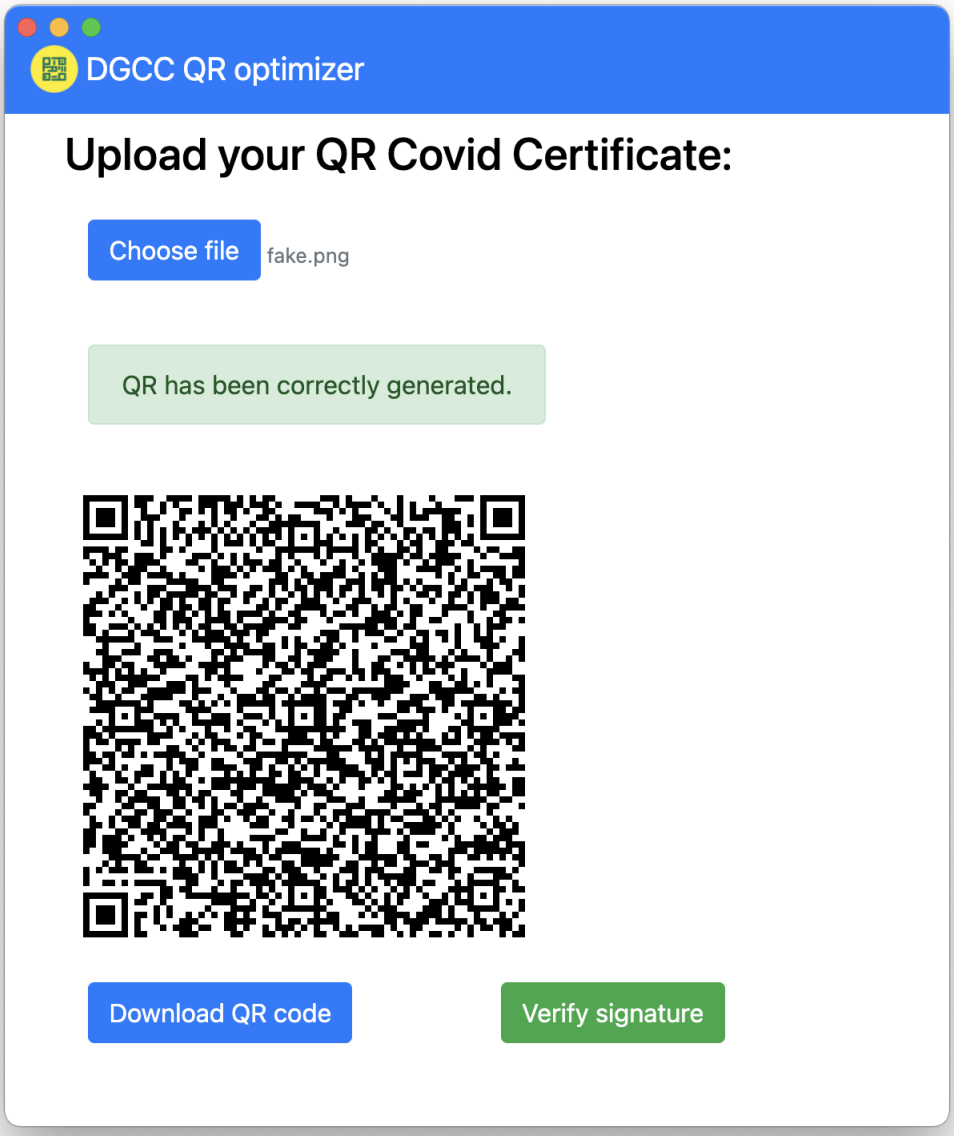

Figure 5: DGCC QR optimizer successful generation.

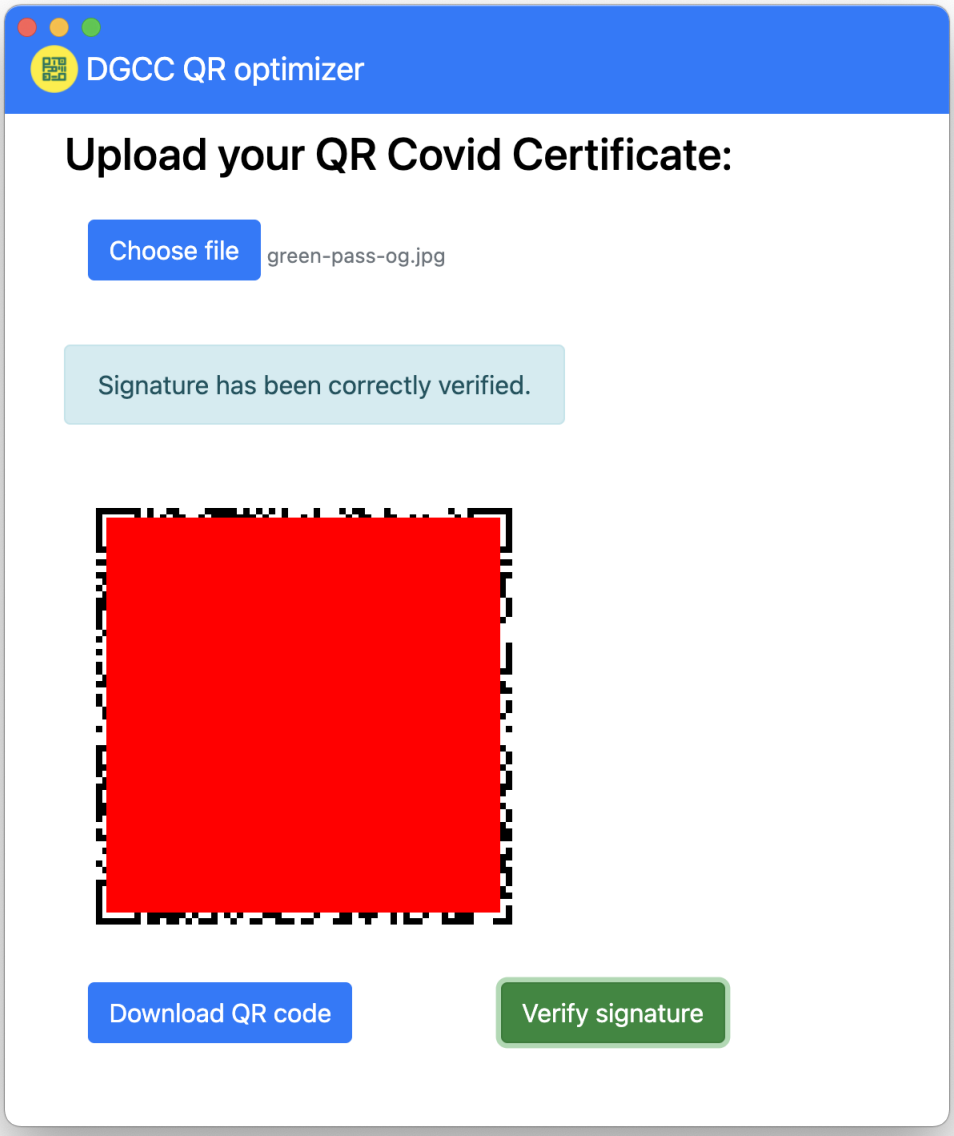

Figure 6: DGCC QR optimizer successful verification.

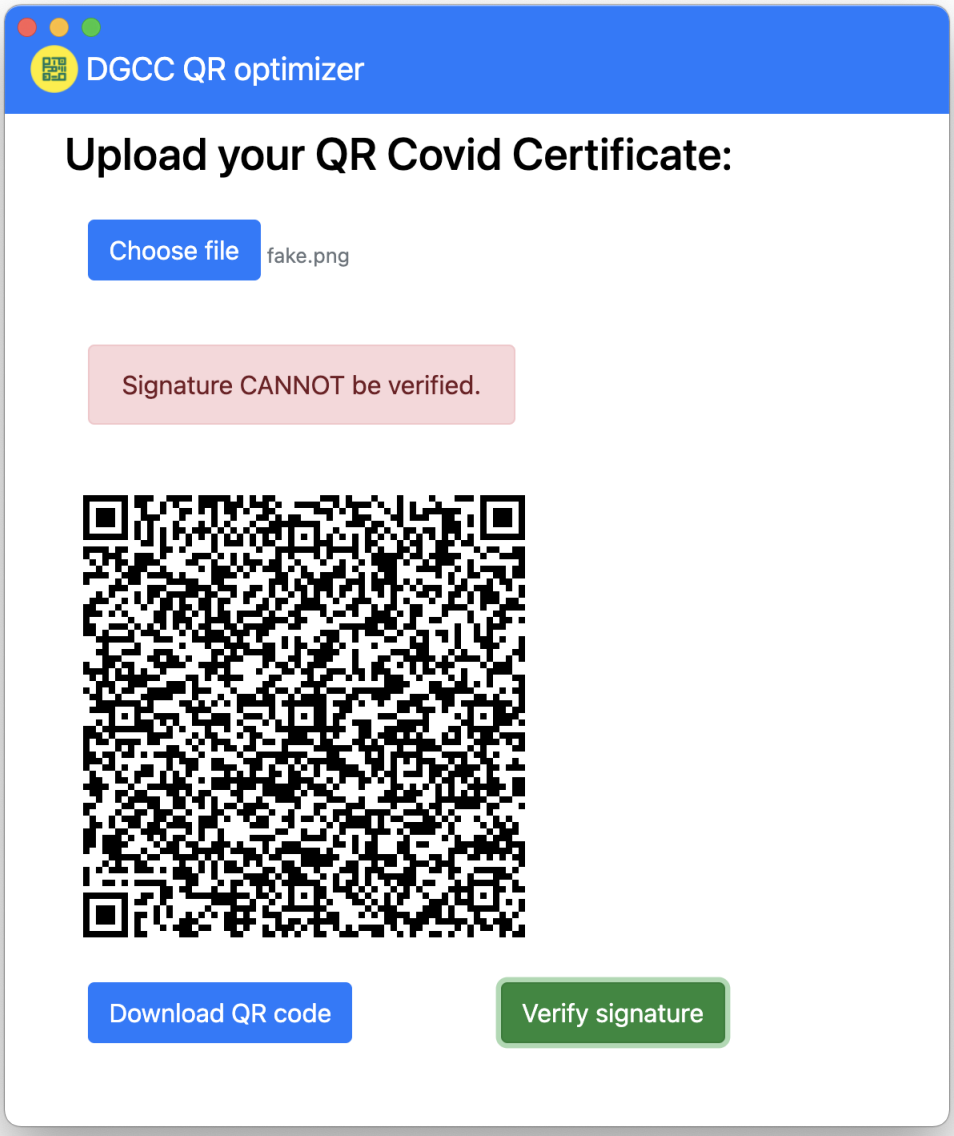

Figure 7: DGCC QR optimizer error in verification.

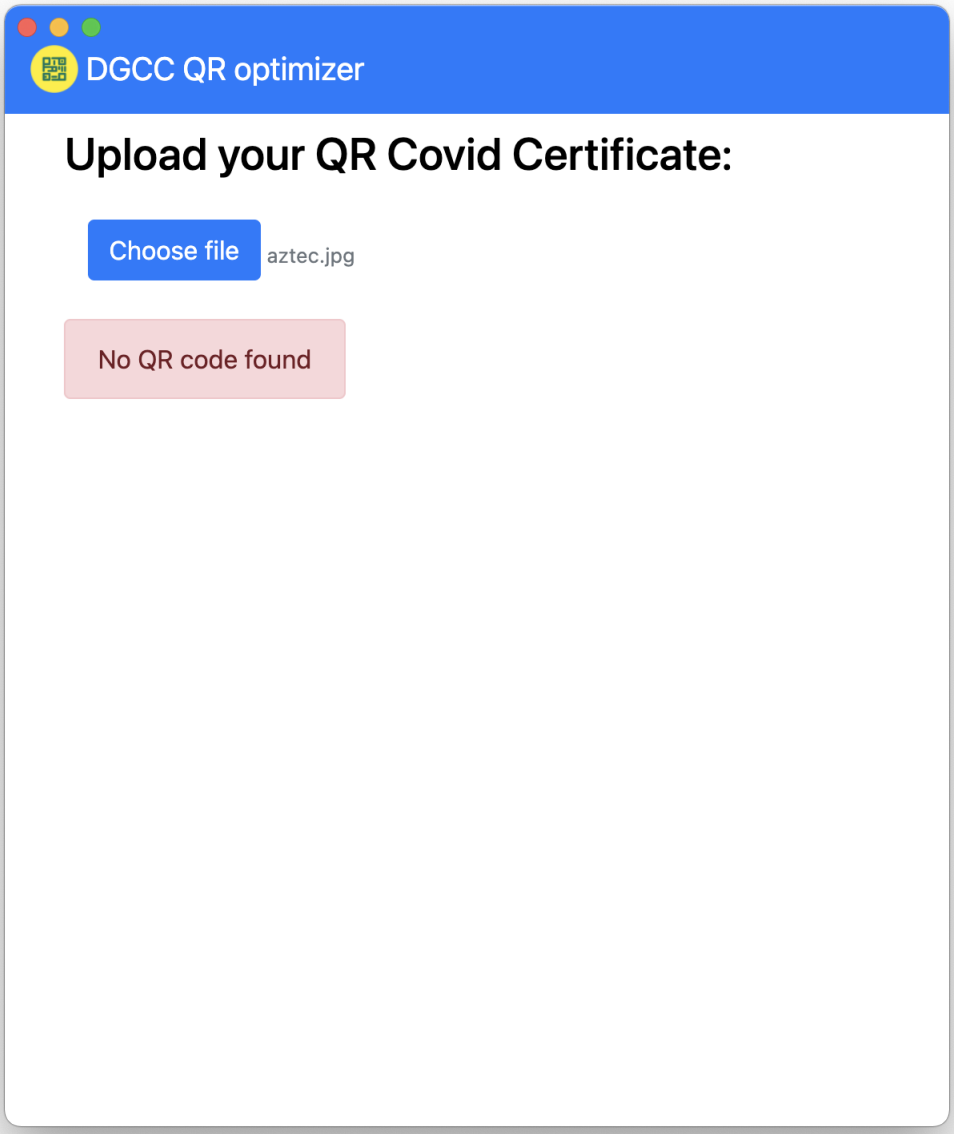

Figure 8: DGCC QR optimizer error message.