Ca' Foscari
University
of Venice

Master's Degree programme

in Computer
Science
LM-18

Final Thesis

Proposal of improvements for the Digital COVID-19 Certificate

Supervisor
Prof. Riccardo Focardi

Graduand
Marco Carfizzi
Matriculation Number 860149

Academic Year
2020 / 2021

Index

1 Introduction **1**
1.1 Scenario . . . 1
1.2 Project objectives . . . 1
1.3 Work structure and achievements . . . 2

2 Background **4**
2.1 QR code . . . 4
2.2 JSON . . . 5
2.3 CBOR . . . 7
2.4 COSE . . . 8
2.4.1 Signature . . . 8
2.5 Digital COVID-19 Certificate . . . 10
2.5.1 System architecture overview . . . 10
2.5.2 Certificate serialization process . . . 11
2.5.3 Payload structure . . . 12
2.5.4 Header content . . . 15
2.5.5 Payload content . . . 16
2.6 Business rules . . . 17
2.7 Programming technologies . . . 17
2.7.1 HTML . . . 17
2.7.2 JavaScript . . . 18
2.7.3 JsonLogic . . . 20

3 Design and practical implementation of results **23**
3.1 Design of the applications and systems architecture . . . 23
3.1.1 Optimization app . . . 23
3.1.2 Light payload architecture . . . 29
3.2 Practical implementation of the apps and systems architecture . . . 34
3.2.1 Optimization app implementation . . . 34
3.2.2 Verification app implementation . . . 38

4 Conclusions **48**
4.1 Relevant aspects . . . 48
4.2 Next steps . . . 48

5 References **50**

1 Introduction

1.1 Scenario

During the first half of 2021 countries from the European Union have developed a system in an effort to return a normal life to its citizens, after months of lockdowns and restrictions due to COVID-19.

The approved and final solution is what is called the "*Digital COVID-19 Certificate*", *DCC* or *DGCC* (Green) from now on: the DCC is defined as a digital documentation that testifies a full vaccination, a negative test or a recovery from COVID-19.

This certification is provided in the form of a QR code and it has to be checked, via apposite applications together with a valid ID, when the law requires it (for example when entering a restaurant or taking a flight), attesting that a person is not infected or is immunized against COVID-19.

The DCC goal is to restore the travelling freedom that COVID-19 countermeasures took away from people, while at the same time providing a certain degree of safety from the disease.

The system has been adopted by EU Countries and the following non-member States: Albania, Andorra, Armenia, Cabo Verde, El Salvador, Faroe Islands, Georgia, Israel, Iceland, Lebanon, Liechtenstein, Moldova, Monaco, Montenegro, Morocco, New Zealand, North Macedonia, Norway, Panama, San Marino, Serbia, Singapore, Switzerland, Taiwan, Thailand, Tunisia, Togo, Turkey, Ukraine, United Arab Emirates, United Kingdom and the Crown Dependencies (Jersey, Guernsey and the Isle of Man), Uruguay and The Vatican.

1.2 Project objectives

The COVID-19 Certificate has to be provided for a verification when needed, according to local national laws, for example when entering a crowded event. The QR code is to be presented to an officer and it will be scanned by an official verification application, alongside an ID document to attest the correct ownership of the certificate.

The first goal of this thesis is to propose a way to decrease the density of the 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 the QR becomes an issue for example, as just mentioned, when this code is printed; during our analysis it has been noticed that the official

Italian verification app (*VerificaC19*) could struggle to get a QR code right and multiple attempts were necessary to get it accepted and verified.

The main focus of our work group is to propose improvements to the DCC QR code and in order to achieve that two possible ways are presented in this thesis.

The first proposal is to provide a slenderer Digital Green COVID-19 Certificate QR code, without any modifications to its content but using a more efficient graphical representation.

The second proposal is to provide a new different DCC payload, with less fields, while at the same time trying to be compliant to the verification rules that are given in the official European specifications documentation.

In particular, the first approach can be applied on top of the second to optimize the final result, resulting in the possible smallest QR code.

1.3 Work structure and achievements

The group is made up of the author of this thesis, Marco Carfizzi, and Giacomo Arrigo, under the guidance and supervision of Professor Riccardo Focardi.

We began our investigation by studying the official guidelines on the DCC architecture and the DCC QR code itself.
Our first research objective was to perform a security analysis on the system, in particular we studied the applications that compose the whole system architecture and tried different manipulations on the DCC content, but performing more experiments on the payload our work focus shifted on a usability analysis, mainly on the QR density and how to improve it.

Our studies focus went initially to the structure behind the DCC.
The comprehension of the different data formats that have been used is crucial to figure out how to manipulate the fields of the payload: the work started with the building of software tools that are capable of decoding the QR code into a readable list of its fields.
Further studies continued in the QR code standard, understanding how it is produced and how it is able to represent information.
Being able to decipher the content of the certificate, handle it and being capable of re-encoding it as a new QR code had a pivotal role in the final proposals.

To solve the first objective of providing a slenderer QR code, our work group built a web application that gives improvement to the quality of life by re-encoding the DCC into a more readable one, while at the same time respecting the privacy of the users that choose to utilise our service.

Regarding the second objective, consisting of shrinking the DCC payload, our

group built a Proof of Concept architecture to generate, sign and validate these new certificates.

In particular two web applications were developed: one that generates and signs our new DCCs and one that verifies them.

2 Background

This section has the goal of giving basic background information about the concepts needed to understand the content of this thesis.

2.1 QR code

The QR format, as defined in “ISO/IEC 18004:2015” [11], is a graphical matrix composed by black dots on a white background arranged in a geometrical square shape; it is used to encode information into a 2D bar-code.

An example is shown in Figure 1: the depicted image is an encoding of the URL of our application demo.

From the provided QR it is possible to define two components of it.
The first one (identified by the red color in the picture) is made up by the three outlined black squares in the top left and right and bottom left corners: these symbols are used to define the position of the whole code, helping also with the identification of the correct size and inclination.
The second component (identified by the green color) is called alignment pattern and is observable near the bottom right corner: the bigger and denser the QR code, the more alignment blocks will be present.
The rest of the internal black squared dots are used to encode the desired information.

The possible black or white squares inside the QR matrix are called *modules*.
It is possible to classify QRs differentiating by size; this grouping is named *version*.
Versions go from 1 to 40, starting to a dimension of 21×21 modules, up to 177×177 modules.

Encoding modes For our use case we need to define two of the possible encoding modes:

- Byte: data is encoded as 8 bit per character;
- Alphanumeric: data is encoded using a set of 45 character. 10 numerical digits (0-9), 26 alphabetical characters (A-Z) and 9 special symbols (SP, $, %, *, +, -, ., /, :).
  In this case 11 bits of space are used for the encoding of two characters.

Error correction The QR code utilizes the *Reed–Solomon* error correction and provides with four different levels of it:

1. L: 7%;
2. M: 15%;

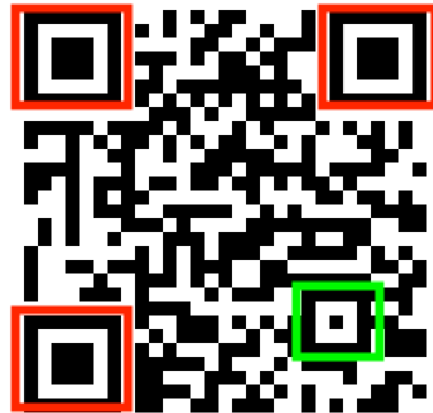


Figure 1: QR containing an URL.

3. *Q*: 25%;
4. *H*: 30%.

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.

2.2 JSON

As defined in ECMA-404, *JSON* (JavaScript Object Notation) "is a lightweight, text-based, language-independent data interchange format". [10]

Among its pros we have the ease with which it can be read by humans and the simplicity with which it can be produced and analyzed by computers.

It is possible to define two structure types inside JSON:

1. objects: obtained by enclosing a set of named values between curly brackets;
2. array: obtained by enclosing an ordered list of named values between square brackets.

Elements that compose these structures are described with a *name* followed by a colon and its *value*.
A value can be a *string* (contained inside quotation marks), a number, a *boolean* (true or false), *null*, an object or an array. [10]

A JSON can be accompanied by its own *Schema*: a schema is a definition of the JSON structure, implemented to know which fields with which value types must be present.

A JSON can be then validated against a given schema to assert if it has a valid and acceptable structure; an example of a JSON Schema is shown in the following snippet, taken from "Understanding JSON Schema" [4]:

```
{
  "type": "object",
  "properties": {
    "first_name": { "type": "string" },
    "last_name": { "type": "string" },
    "birthday": { "type": "string", "format": "date" },
    "address": {
      "type": "object",
      "properties": {
        "street_address": { "type": "string" },
        "city": { "type": "string" },
        "state": { "type": "string" },
        "country": { "type" : "string" }
      }
    }
  }
}
```

Of course a JSON object like the following will not be accepted, since its fields do not match with the ones provided by the schema:

```
{
  "name": "George Washington",
  "birthday": "February 22, 1732",
  "address": "Mount Vernon, Virginia, United States"
}
```

While the next JSON is to be accepted, since it provides correct fields and value types:

```
{
  "first_name": "George",
  "last_name": "Washington",
  "birthday": "1732-02-22",
  "address": {
    "street_address": "3200 Mount Vernon Memorial Highway",
    "city": "Mount Vernon",
    "state": "Virginia",
```

```
9       "country": "United States"
10    }
11  }
```

2.3 CBOR

CBOR (Concise Binary Object Representation) is a format essentially based on JSON, from which it differs mainly for the use of binary encoding, saving space and allowing faster processing.
A JSON can be fully translated into a CBOR.

As stated in *RFC 8949* [3], “a CBOR data item is encoded to or decoded from a byte string carrying a well-formed encoded data item”.
Moreover, “the initial byte of each encoded data item contains both information about the major type (the high-order 3 bits) and additional information (the low-order 5 bits)”.

The value of the additional information gives information about how to load the actual argument value:

- **less than 24**: the value is encoded directly in the additional information spot;
- **24, 25, 26, or 27**: the argument value is to be read in the next 1, 2, 4 or 8 bytes.

For the scope of this thesis only major types 1, 2, 4 and 5 will be described as in RFC 8949 [3]:

- **1**: A negative number in the range $-2^{64}..-1$, extremes included. The result is given by -1 minus the decoded value (*e.g.* Integer -500 given by *0b001_11001*, meaning that we have major type 1 and additional information 25, followed by *0x01f3* bytes that convert into decimal number 499);
- **2**: A byte string, whose length is given as additional information (*e.g.* data header *0b010_00101* states major type 2 and string length of 5 bytes. If we had a string of length 500 then we would have *0b010_11001*, stating major type 2 and additional information 25, followed by the two bytes *0x01f4* and finally followed by the 500-bytes string);
- **4**: An array of data items. Elements can have different types and the additional information gives the length of the array (*e.g.* data header *0b100_01010* states major type 4 with an array of 10 elements and it must be followed by the 10 data items);

◇ **5**: A map of pairs of data items, in particular key-value couples. The pair number is given as additional information (*e.g.* a map with 9 couples is described by the data header *0b101_01001* stating major type 5 and additional information 9 and will expect the 18 items as next, with the following structure: first item is first key, second item is first value, third item is second key, and so on).

2.4 COSE

The *COSE* format (*CBOR Object Signing and Encryption*) is the one used to store and sign the full DCC payload; in particular, we refer to the *COSE_Sign1* signature structure.

“The COSE_Sign1 signature structure is used when only one signature is going to be placed on a message”. [12]

The COSE_Sign1 structure is a CBOR array. The fields of the array in order are defined in eHealth Network guidelines on “Technical Specifications for Digital Green Certificates Volume 3 Interoperable 2D Code” [16]:

◇ protected, major type 2 (binary string);

◇ unprotected, major type 2 and empty;

◇ payload, major type 2, containing health information;

◇ signature: this field contains the computed signature value, major type 2.

The signature is computed on the entirety of the bits of the COSE, meaning that it uses also the headers and not only the payload.

2.4.1 Signature

The signature algorithm scheme is called signature with appendix.
In this scheme, defined in [12], “the message content is processed and a signature is produced; the signature is called the appendix.
This is the scheme used by algorithms such as *Elliptic Curve Digital Signature Algorithm* (ECDSA) and the RSA Probabilistic Signature Scheme (RSASSA-PSS)”.

The signature functions are the following:

◇ signature = Sign(message content, key)

◇ valid = Verification(message content, key, signature)

ECDSA ECDSA is a variant of the *Digital Signature Algorithm* (*DSA*), which uses elliptic curve cryptography.
It's the *FIPS* (*Federal Information Processing Standards*) standard for digital signature, proposed by *NIST* (*National Institute of Standards and Technology*) in 1991, based on public-key cryptography.
Since this is the primary, and so far only, algorithm for the DCC signature generation and verification, it is the only one that is defined in this thesis.

A few concepts:

- Private key: A single unsigned 256-bit integer (32 bytes) known only to the person that generated it.
- Public key: A number computed from the private key (can't compute the private key from it). It is used to verify if signature is genuine.
- Signature: A number that proves that a signing operation took place.
  A signature is mathematically generated from a hash of something to be signed, plus a private key.
  The signature itself is made up of a couple of two numbers, known as *r* and *s*.

Steps to compute the signature:

1. Generate a random number k such that $0 < k < q$.
2. Compute $r = (g^k \mod p) \mod q$.
3. Compute $s = (k^{-1}(H(m) + x * r)) \mod q$, where $H(m)$ is a SHA-d hash function applied to message m.
4. If $r = 0$ or $s = 0$, recompute the signature.
5. Signature is the pair (r, s).

Usually the *extended Euclidean algorithm* is used to compute $k^{-1} \mod q$ to have best performance.

COSE specifications point out to use a deterministic implementation of DSA, such as [22], so that the generation of the value of k is not relying on random number generation to avoid collisions: the biased generation of the random value is an attack on this algorithm and has already been exploited in the past.

Steps to verify the signature:

1. Reject signature if $0 < r < q$ and $0 < s < q$ is not verified.
2. Compute $w = s^{-1} \mod q$.
3. Compute $u_1 = (H(m) * w) \mod q$.

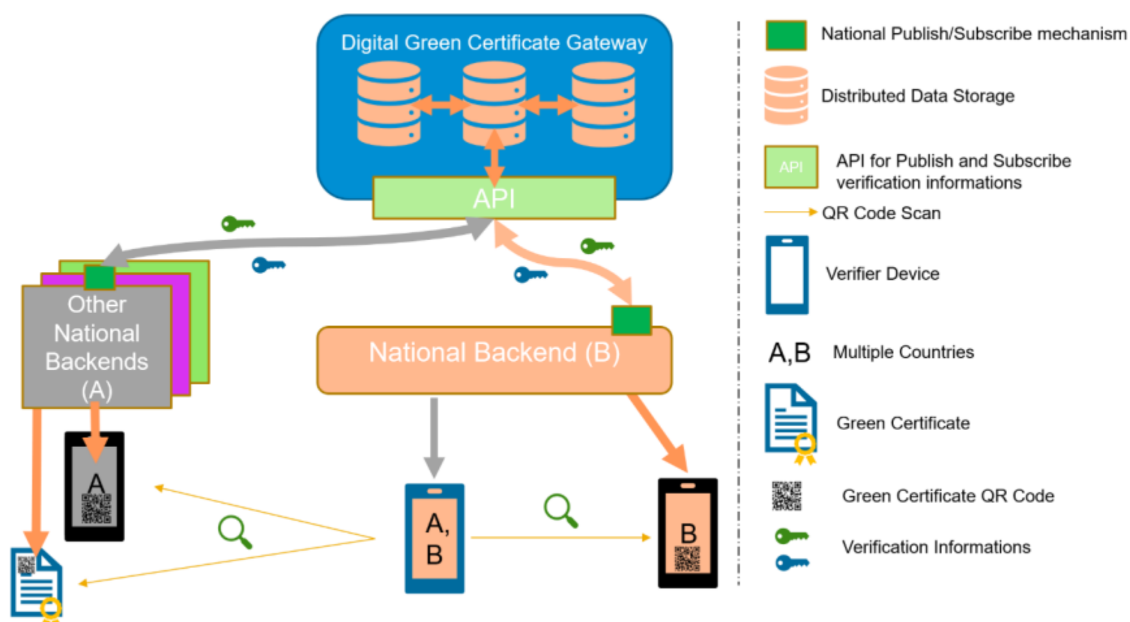


Figure 2: DCC architecture overview.

4. Compute $u_2 = (r * w) \mod q$.
5. Compute $v = ((g^{u_1} * y^{u_2}) \mod p) \mod q$.
6. Signature is verified if $v = r$.

2.5 Digital COVID-19 Certificate

2.5.1 System architecture overview

An overview of the DCC system architecture, taken from Volume 2 of guidelines [15], is depicted in Figure 2.
On the top level, colored in blue, it is possible to notice a component called *Digital COVID-19 Certificate Gateway* (DCCG): the gateway purpose is to give a common interface between countries back-ends, to share the signature keys in order to let each nation to be able to perform verification of the DCCs.

The DCCG is implemented following a *Publish/Subscribe* design pattern: each country back-end subscribes to the gateway dispatcher, receiving updates regarding the public keys (*Trust lists*) from other member nations.

By doing so it is possible for each country to provide certificates verification services, for member countries that adhere to the network, via the validation apps.

The connection between national back-ends and the gateway uses *Transport Layer Security* (TLS) with mutual authentication to create an encrypted com-

munication channel between the two.
This means that both the DCCG and the back-ends must hold respectively a server and a client TLS certificate, provided by a publicly trusted certificate authority as specified in Volume 5 of guidelines (Public Key Certificate Governance) [18].

Following the definition of two entities [15]:

- *CSCA*: Country Signing Certificate Authority certificate;
- *DSC*: Document Signer Certificates, certificate which the member state is using to sign documents.

The CSCA is the authority that generates the DSCs, giving the following Certificate Authority hierarchy: Root CA → CSCA → DSCs.
Having multiple DSCs means that each member state can have multiple public keys to validate (and compute from the corresponding private ones) signatures.

2.5.2 Certificate serialization process

The serialization process needed to produce a DCC QR code is depicted in Figure 3, taken from Volume 3 of guidelines [16].

First, the user health data is stored in a *JSON* document; this is done only to provide a JSON Schema as specified in eHealth Network "Guidelines on Technical Specifications for EU Digital COVID Certificates JSON Schema Specification" [19].

The JSON payload is then stored as a *CBOR* (binary document), to be built into a *COSE_Sign1* as next step: the correct header is added to the CBOR and the whole content is signed with the DSC private key, choosing the selected cipher algorithm.

The binary stream that constitutes the COSE is then compressed with the open source software library *ZLib*: this pass reduces the payload size without losing any content (loss-less data compression). [20]

The compressed string is encoded as *base45*: this standard, defined in IETF "The Base45 Data Encoding" [21], transforms an input string into a new one defined only over a 45-character subset of the *US-ASCII*, encoding 2 bytes in 3 characters.
This translation is useful when working with QR codes, since it works on the same character subset that is employed in QR alphanumeric mode.

The next step is to add, as a prefix, the string "*HCx:*" (where *x* is an integer number) to state the Health Certificate version number.

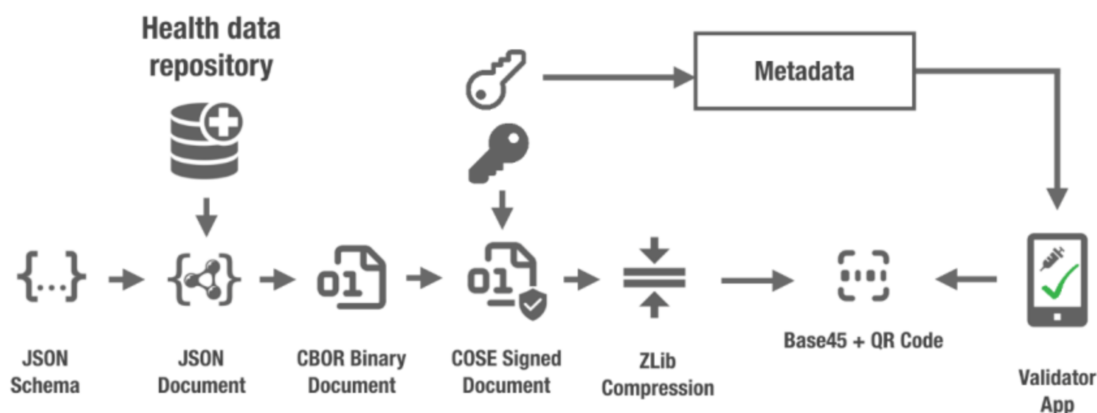


Figure 3: Serialization process.

The last step is to encode the final string as a QR: this code is the one that is given to user that received a negative test, a vaccination or has recovered from COVID-19 and it is the one to be provided when the DCC must be verified.

2.5.3 Payload structure

Following the serialization process (point 2.5.2) steps backwards it is possible to extract the DCC payload from a QR.

The following listing is an example of a decoding of a DCC QR code (a base45 string, prefixed with the Health Certificate version 1, composed by 600 characters) signed with ECDSA and taken from test-suite [2]:

HC1:NCFRW2EM75VOP10DGHKX30R8C:3-U22329RF.+3IX9UT60AN67C5
KM$QWV4-$9J$I14N2E9O51XWUWBH5+9+JTKBL$01Y$2-:2U/DNSG3UNTB3-
DUAE73ZV6UUZKJRX5:\%82WJ6BMHZRMY6 ODJV8IHVR.HPMTHDUJ6W9$3.
4LIMHTNT +NLJCZ 9HHLSOIQE3R49NI95-8*20O51OLNDHPR13PVHQ44FNU-
VH0LVEG4Z\%RMWIK$A679VWI:C73ZODZ2D60/XG3BBI5TCSQ7:A0LMF/VF$
N8NPQVPXD646JL5IWI76ITXZIMN0\%20HXUXFGCKDK L79C8T0L65IDHDQE.
ZOT8B**0--4*126*HU71455Q\%N:E4N1MYH9JS2EBW J3$CLPFD.S7GSI.6
SK U44UZ.I2\%VUK1\% 2ARJK8D1MQM\%DC*CSEP 8P9ND8P17ZS2\%L7959
YJZ/4RR21D3TQTXU2/MLKGUZ$6Z1LSPOZ-EZ+CL0
D9:P**VTS7XY1D\%3S4N93F97VWE2P6OT:LJTUOWE7+NA$I4P7LU9+4JEK
I6 TKP7FHN-CE5-O32W\%-1.GN: U4NP204ONIW0

Removing the version prefix, decoding the result from base45 to hexadecimal and decompressing it, results in the following COSE payload (786 characters):

d2844da201260448d919375fc1e7b6b2a0590133a401624154041a63c196
3c061a61e062bc390103a101a4617681aa62636978315524e3a55564349

```
3a30313a41543a3130383037383433463934414545304545353039334642
43323534424438313323426263 6f62415462646e016264746a323032312d
30322d3138626973781b4d696e6973747279206f66204865616c74682c20
41757374726961626d616d4f52472d3130303033303231356 26d706c4555
2f312f32302f3135323862736402627467693834303533393030366 27670
6a3131313933343930303763646f626a313939382d30322d3236636e616d
a462666e754d75737465726672 61752d47c3b6c39f696e67657262676e68
4761627269656c6563666e74754d55535445524652 41553c474f45535349
4e47455263676e746847414252 49454c456376657265312e322e3158402b
73787641e3181365112cb97e7bed805e820549383f0bd070c4c58598925d
c058078354211f6c23383afa20b3835eb3f8752c686a6efad22c37332999
d99414
```


Extrapolating the content, keeping out the signature, we get the *CWT*: “the payload is structured and encoded as a CBOR with a COSE digital signature. This is commonly known as "CBOR Web Token" (CWT), and is defined in ‘IETF RFC 8392’.

The payload, as defined below, is transported in a *hcert* claim.”[14]

“The CWT is a compact means of representing claims to be transferred between two parties.
The claims in a CWT are encoded in the Concise Binary Object Representation (CBOR), and CBOR Object Signing and Encryption (COSE) is used for added application-layer security protection.
A claim is a piece of information asserted about a subject and is represented as a name/value pair consisting of a claim name and a claim value.
CWT is derived from JSON Web Token (JWT) but uses CBOR rather than JSON”. [24]

Following an overview of the CWT structure [14], along its content and CBOR data information [16]:

◇ Protected Header: contains the used algorithm and the key identifier;

- Signature Algorithm (*alg, label 1* major type 1, values: -7/-37, Algorithm Field)
- Key Identifier (*kid, label 4* major type 4, value: first 8 bytes of the SHA256 value)

◇ Payload

- Issuer (iss, claim key 1, optional, ISO 3166-1 alpha-2 of issuer, major type 2)
- Issued At (iat, claim key 6, major type 2)

- Expiration Time (exp, claim key 4, major type 2)
- Health Certificate (hcert, claim key -260, , major type 5)
- EU Digital Green Certificate v1 (eu_dgc_v1, claim key 1)

◇ Signature

Here is a vaccination payload, human readable, example (no header and signature):

```
4(Expiration): 1683849600,
6(Issued at): 1627989097,
1(Issuer): "IT",
-260(Health Certificate): {
   1(DGCv1): {
        v: [
            {
              dn: 3,
              ma: 'ORG-100030215',
              vp: '1119349007',
              dt: '2021-12-21',
              co: 'IT',
              ci: 'ABC01ITREDACTED123#4',
              mp: 'EU/1/20/1528',
              is: 'Ministero della Salute',
              sd: 3,
              tg: '840539006'
            }
        ],
        nam: { fnt: 'CARFIZZI', fn: 'CARFIZZI', gnt: 'MARCO', gn:
    'MARCO' },
        ver: '1.3.0',
        dob: '1994-12-07'
    }
}
```

This payload can be read with the help of the following legend:

◇ 4: QR expiration date in UNIX TimeStamp format;

◇ 6: QR issuing date in UNIX TimeStamp format;

◇ 1: QR issuer country;

◇ v: array containing vaccination data;

- ◇ dn: last received dose;
- ◇ ma: marketing authorization holder;
- ◇ vp: vaccine or prophylaxis;
- ◇ dt: date of vaccination;
- ◇ co: country of vaccination;
- ◇ ci: unique certificate identifier;
- ◇ mp: vaccine medical product;
- ◇ is: certificate issuer;
- ◇ sd: total series of doses;
- ◇ tg: disease or agent targeted;
- ◇ nam: full name field;
- ◇ fnt, fn: (standardized) surname;
- ◇ gnt, gn: (standardized) forename;
- ◇ ver: schema version;
- ◇ dob: date of birth.

With the introduction of the third dose and the "super green pass", the version field has been updated to 1.3.0. from the initial 1.0.0. for Italian DCCs.

2.5.4 Header content

Key identifier The Key Identifier (*kid*) is a string stored in one of the headers that allows the verifier applications to fetch the correct public key from a list of Issurer keys (*iss*).

The *kid* may be placed both in the unprotected or the protected header since it is not a security-critical field, as described in Volume 1 [14].

"The *kid* is calculated when constructing the list of trusted public keys from DSC certificates and consists of a truncated (first 8 bytes) SHA-256 fingerprint of the DSC encoded in DER (raw) format". [14]

Signature algorithm The *alg* field in one of the headers defines the name of the employed signature algorithm.

In eHealth Network specifications Volume 1 [14] is stated that two algorithms are defined to be used for the signature generation and verification: one primary and one secondary.

The secondary is to be used only if issues arise from the primary.

The chosen algorithms, in order and already defined in point 2.4.1, are:

1. *Elliptic Curve Digital Signature Algorithm* (ECDSA) with P-256 curve and SHA-256 hash algorithm, corresponding to COSE algorithm parameter *ES256* and *alg* value -7;

2. *RSA Probabilistic Signature Scheme* (RSASSA-PSS) with a modulus (key length) of 2048 bits and SHA-256 hash-algorithm, corresponding to COSE algorithm parameter *PS256* and *alg* value -37.

An example of header content:

```
{4: h'349A42B0C2D0728E', 1: -7}
```

Indicates the bytes of the *kid* in label 4 and ECDSA signature algorithm in label 1 with value -7.

2.5.5 Payload content

In point 2.5.3 it has already been given an overview about the main components of the COSE payload; this section has the goal of describing more in detail the claims in it.

Issuer The *iss* claim (key 1) contains the ISO 3166-1 alpha-2 of the country that is emitting the DCC.

This field can be used by verifier apps to check the correct sub-set of DSCs to use in the validation phase.

Expiration time The *exp* claim (key 4) is a timestamp that states for how long the signature is to be considered valid: it is indicated as a *NumericDate* format (number of seconds since Epoch).

Issued at The *iat* claim (key 6) is a timestamp that states the time when the certificate was issued.

Health certificate claim The *hcert* claim (key -260) is the core of the DCC: it contains a JSON object, encoded with CBOR format, that stores the health information about its owner.

The JSON format is used only for schema purposes; as specifics from Volume 1, “application developers may not actually ever de-, or encode to and from the JSON format, but use the in memory structure”. [14]

2.6 Business rules

As stated in the "EU DCC Validation Rules" document [13], "the EU DCC Validation Rules are applied on the payload of the HCert".

A payload must be checked against the set of the correct business rules to state its content validity: an example of check could be a date validation (certificate valid before a given number of months from emission) or that the vaccination is coming from a European Medicines Agency approved manufacturer. In particular, rules that check fields in a similar way as the last example must fetch the values from given *value sets*: as stated in the guidelines about validation rules [13], "the *valuesets* will be placed as JSON on the server and will be published by a public route", meaning that it is available as a public API.

Countries can have different rules from each other, yet they must maintain the same structure, depicted in Table 1 as per specifications [13], to be evaluated by a software component called *Rule Engine*.

In Table 2 we can see the EU validation rules with their descriptions [1], with the following legend:

- *GR*: General rule, valid for each category;
- *VR*: Vaccination rule;
- *TR*: Test rule;
- *RR*: Recovery rule.

Technical details about rules logic are discussed in points 2.7.3 and 3.2.2.

2.7 Programming technologies

The goal of this section is to define and give basic information about the technologies employed to design our applications.

2.7.1 HTML

The *HyperText Markup Language* (HTML) is a markup language, defined by the “*World Wide Web Consortium*” (W3C), used to build web pages.

HTML is used to statically define the structure and graphical layout of a web document via different markup *tags*.

Even if HTML itself is not a programming language, it is possible to insert *scripts* inside it.

Field	Description
Identifier	The unique Rulename
Type	Type of the Rule
Version	Version of the Rule
SchemaVersion	Version of the used Schema
Engine	Type of the RuleEngine
EngineVersion	Version of the used Engine
CertificateType	Type of the certificate
Description	Array of Human readable description of the rule
ValidFrom	Start Validity of the Rule
ValidTo	End Validity of the Rule
AffectedFields	Fields of the payload which are used by the rule
Logic	The logic payload in JSON

Table 1: Rule format.

2.7.2 JavaScript

JavaScript (JS) is an object and event oriented programming language, mainly deployed in web applications client side.

It's a weakly typed programming language, interpreted by the browser on the client side: this means that is not executed on the server side, but on the user machine directly.

JS code can be inserted into a HTML page to extend a web app functionalities, adding dynamic components to it.

Electron Electron is a open-source framework, developed and hosted by GitHub Inc., that combines the Chromium rendering engine and Node.js runtime to allow web applications to run as if they were native desktop applications.

Electron apps are structured in processes: one "*browser*" process and multiple "*renderers*". The browser is the component that runs the application logic and renders the result in windows that are presented to the user.

Electron apps can be, simplistically, viewed as web-sites that are contained in a Chromium instance.

The pro of using such framework is the possibility of building web-apps into desktop cross-platform applications, with little to no effort.
The cons consist in having apps that are a bit heavier on the space side (both on primary and permanent memory), since they are bundled with Chromium

Code	Description
GR-EU-0000	Exactly one type of event.
GR-EU-0001	The "disease or agent targeted" must be COVID-19 of the value set list.
VR-EU-0000	At most one v-event.
VR-EU-0001	EMA must approve allowed vaccines.
VR-EU-0002	The vaccination course must be completed to provide enough protection.
VR-EU-0003	The full vaccination protection starts up 14 days after vaccination and is valid for 365 days.
VR-EU-0004	The number of doses must be positive.
TR-EU-0000	At most one t-event.
TR-EU-0001	The test type (tt) can be RAT or NAA.
TR-EU-0002	If the test type is "RAT" then the test must be in the list of accepted RAT tests.
TR-EU-0003	The test must be performed in the previous 72 hours.
TR-EU-0004	Test result must be negative ("not detected").
RR-EU-0000	At most one r-event.
RR-EU-0001	The certificate must be valid.
RR-EU-0002	The validity period of the certificate must be checked.

Table 2: EU validation rules and their business descriptions.

functionalities, and a lack of integration with the operating system, since that Electron apps are not native.

2.7.3 JsonLogic

JsonLogic [25] is a format that supports comparisons between data with support to variables, and not only literals.

JsonLogic defines a syntax for writing rules in JSON notation: each rule results in one decision.
The structure of a rule logic always follows the following schema:

```
{"operator" : ["values" ... ]}
```

It is possible to have nested operators (as "*values*" in the operands array), also no unsafe *eval()* function is called.

A simple example of a JsonLogic rule:

```
jsonLogic.apply( { "==" : [1, 1] } );
```

The value returned by this rule is "*true*", since it evaluates $1 == 1$.

Let's now introduce and list the main operators for this thesis use case:

- **Accessing data:**
  - *var*: access a variable value, whose identifier is the operand;
    e.g. $"var" : ["a"]$, with data $"a" : 1$, returns 1.
- **Logic and Boolean operations:**
  - *if* usually has three arguments: a guard, a *then* branch and an *else* branch. If the guard evaluates to true the *then* branch is executed, otherwise the *else* branch is picked.
    If more than 3 arguments are provided then they will be paired up as if-then elseif-then else;
  - $==$ and $===$ are the equality comparisons between two operands; in the first case (abstract equality) type conversion will happen if necessary, while in the second case (strict equality) no conversion is done, meaning that operands should already be of comparable type;
  - $!=$ and $!==$ are the counterpart of last point, for inequality comparison;
  - $!$ takes only one operand and performs its negation;

Value	JsonLogic boolean
0	false
non-zero number	true
empty array	false
non-empty array	true
empty string	false
non-empty string	true
"0"	true
null	false

Table 3: JsonLogic boolean values.

- *or* and *and* take two operands and perform the logical operations of “OR” and “AND” on them;

◇ **Numeric operations:**

- *>*, *>=*, *<* and *<=*;
- *+*, *-*, ***, */* and *%*;

◇ **Array operations:**

- *reduce*: used to combine the elements of an array into one value. It takes two arguments: the array and a lambda function (provided in the form of $"current" :< current >, "accumulator" :< accumulator >$, where “*current*” contains the proper item of the array and “*accumulator*” contains the result, so far, of the left-folding);
- *in* takes two arguments and checks if the first argument is contained inside the second argument (that has to be an array).

To ensure that boolean values are mapped correctly between different platforms, since verification logic can be applied in multiple programming languages, JsonLogic specifies how they are evaluated as shown in Table 3.

CertLogic CertLogic, specified in “*EU DCC Validation Rules*” [13], defines a subset of JsonLogic syntax: CertLogic is compatible with JsonLogic, but it expands where necessary (*i.e.* for correct handling of dates, needed to check if a DCC has expired).

The data types are the same of JsonLogic, with the following modifications:

◇ **truthy** accepted values are *true*, *any non-empty array or object*;

◇ **falsy** accepted values are *false* and *null*.

◇ **datetime**, or *timestamp*, used to handle the date format: to transform a

string into a *datetime* one must perform a *plusTime* operation adding 0 days.
This solution is adopted to allow a consistent cross-platform date type, without relying on native types.

In order to be able to operate with *datetime* data, the following operations have been defined:

- **plusTime** adds time to a string value in an accepted date format, returning a *datetime*.
  It takes three operands: the first is the string to be evaluated, the second is the integer representing the quantity of time to be added (can be negative) and the last one is a string with a time unit (*i.e.* "hour", "day");
- **after**: equivalent to > but date-time specific, operands must be *datetime*;
- **before**: equivalent to < but date-time specific, operands must be *datetime*;
- **not-after**: equivalent to <= but date-time specific, operands must be *datetime*;
- **not-before**: equivalent to >= but date-time specific, operands must be *datetime*;

3 Design and practical implementation of results

3.1 Design of the applications and systems architecture

3.1.1 Optimization app

This section has the goal of describing the optimization web application, while at the same time providing information on how the users data are treated and utilized.

A demo is available at this link: https://mcarfiz.github.io/dgcc-optimizer-js/.

A user can input their *DCC* in the form of a QR code, both as an image file or by scanning it with the webcam, and download its optimized version.

The resulting encoding is compatible with the Italian official verification app (*VerificaC19*) and should theoretically be compliant with every other European app.

In particular these are the countries that our work group tested, alongside pictures for proof of correctness: Belgium (Figure 4a), Czech Republic (Figure 4b), Ireland (Figure 5a), United Kingdom (Figure 5b), Austria (Figure 6a), Portugal (Figure 6b), Swiss Confederation (Figure 7a) and Germany (Figure 7b).

Optimizing the QR code For privacy reasons, only fake and not valid certificates (not signed with a real key) are shown as examples in this thesis [2].

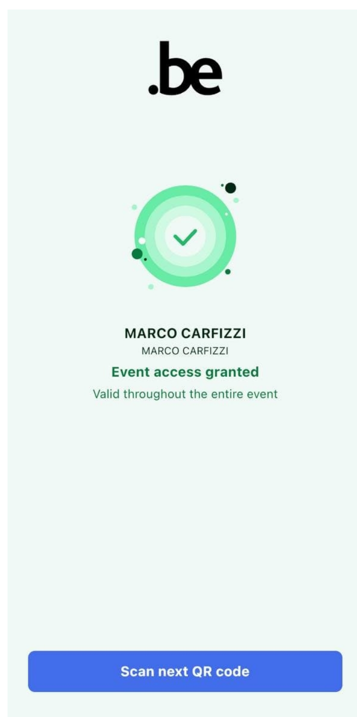
In Figure 8a it is possible to notice a simulated green certification: it's clear just by a first look that this QR is really big and dense compared to usual bar-codes.

The proposed optimizations also help when the QR is shown via a smartphone screen, and not only when it is physically printed on paper, by reducing the needed time of the readings.

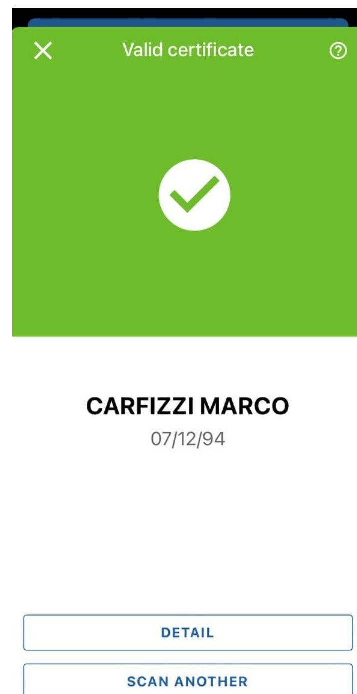
Of course, the information encoded inside the QR is composed by multiple attributes so it is not possible expect a micro QR-like compression.

An example of the optimized QR found in Figure 8a can be seen in Figure 8b.
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.
The graphical comparison between the two DCCs underlines the useless density of the original one.

Error correction An error correction rate of '*Q*' (around 25%) is recommended by design specifications [14], without giving a proper explanation.
The reason behind this decision can be hypothesized in a false sense of "safety"

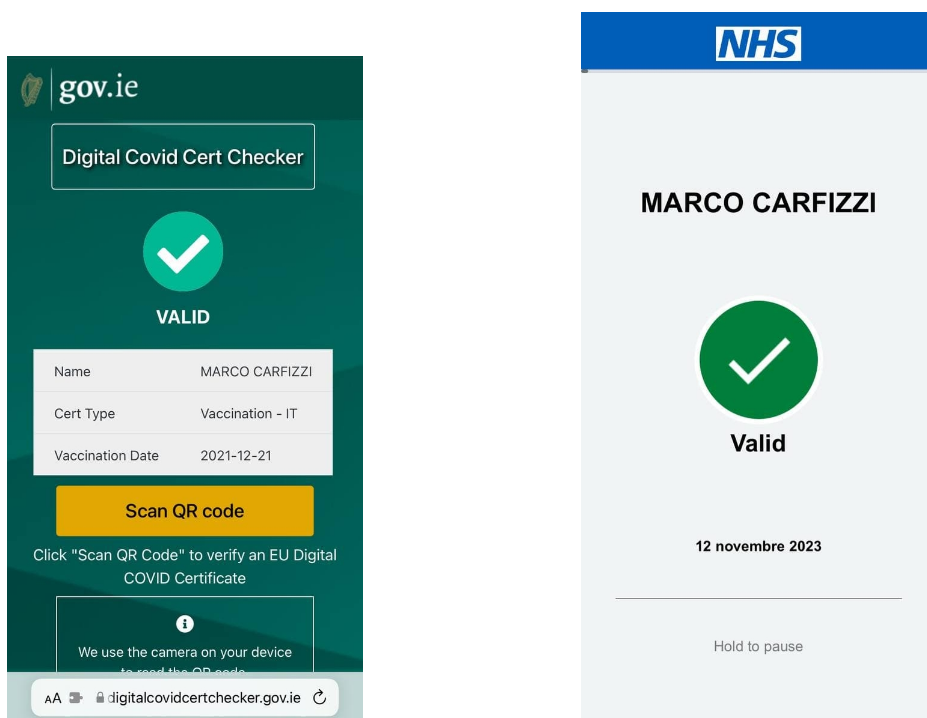


(a) Belgium DCC verification app result.



(b) Czech Republic DCC verification app result.

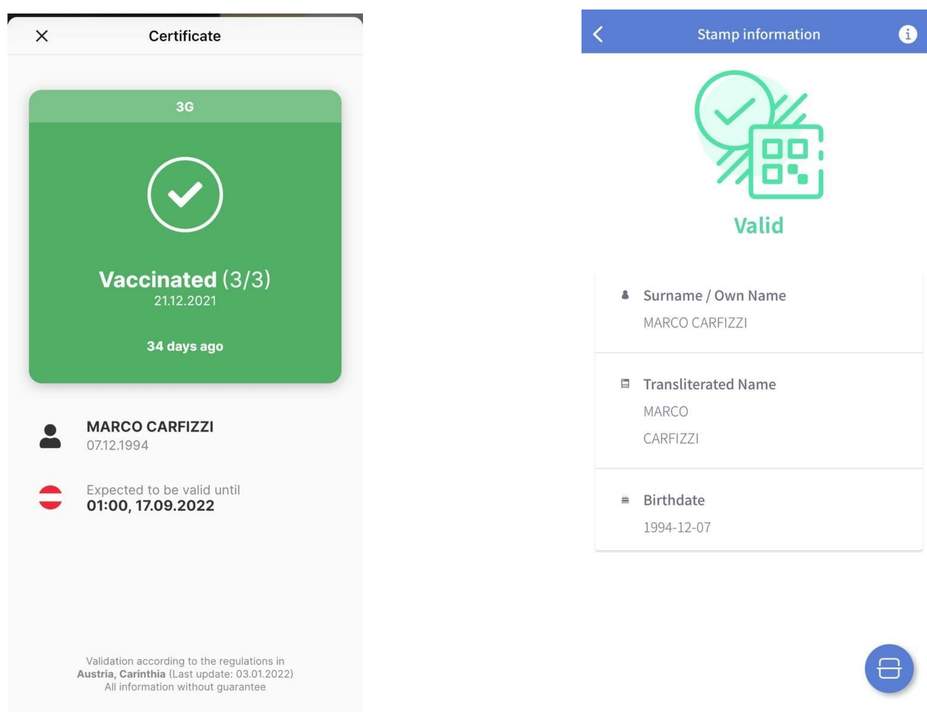
Figure 4: Belgium and Czech Republic DCC verification app results.



(a) Ireland DCC verification app result.

(b) United Kingdom DCC verification app result.

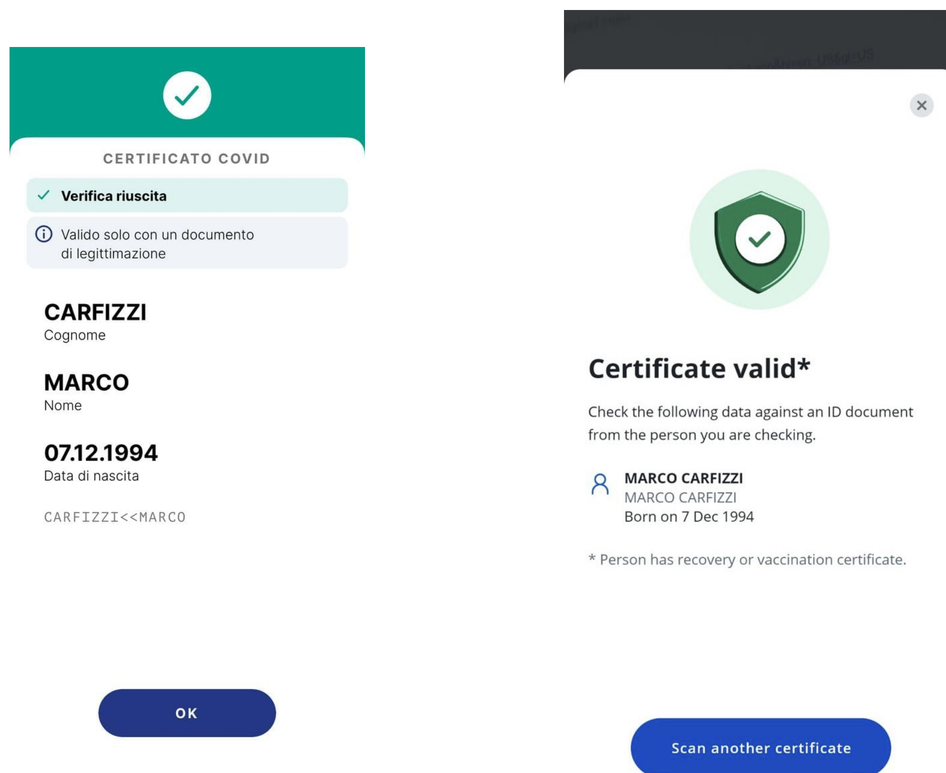
Figure 5: Ireland and UK DCC verification app results.



(a) Austria DCC verification app result.

(b) Portugal DCC verification app result.

Figure 6: Austria and Portugal DCC verification app results.



(a) Swiss DCC verification app result.

(b) Germany DCC verification app result.

Figure 7: Switzerland and Germany DCC verification app results.

(a) Original DCC.

(b) Optimized DCC.

Figure 8: Example of DCC optimization difference.

in employing a higher correction rate: of course, the higher the correction level, the more QR code can be recovered from an unreadable one at expenses of having a denser image.

The use cases of the DCC are mainly only two:

1. presenting the QR from a smartphone (or a similar electronic device) screen;
2. printing the QR code on paper.

In most cases users that prefer the second solution are older people (in Italy for example it's possible to get the certificate in a paper format directly from pharmacies) or people that are not quite tech experts and do not "trust" electrical devices to carry sensitive data: there is a high probability that the printer hardware employed in the process is not high-performant (*i.e.* print with a small dot-per-inch ratio).
In this scenario, employing a high level of error correction is detrimental to the usability of the system, since the low quality printed QR code really hinders the reading process employed in the DCC verification step by slowing it down or even incapacitating it altogether.

The high density QR code is also not ideal when operating in the first use case, since smartphone have a limited screen size and the presented image will be reduced to respect the physical size of the device; by doing so, smaller portions of the black dots of the image can stop being visible.
For example when displays employ a low pixel resolution, combined with particular sub-pixel dispositions such as Samsung PenTile Matrix, graphical artifacts (*i.e.* moiré effect and blockiness on image) might arise and smaller parts of the image will be displayed incorrectly.

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 work group intuition is the following: given the fact that we need the totality of the bits of the QR content to be intact in order to perform the correct signature validation, it is a waste of space to increase the redundancy of the code, even without taking into account the usability argument.

The fact of having a QR that is read with more reliability overweighs, in terms of system usability, the possibility to have significant 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

```
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 9: VerificaC19 iOS app Health Certificate version check.

'bytes', but as an alphanumeric string.

Since the encoding is performed on a string, as a matter of fact, there is really no need to treat it as any other format.

As per official specifications of the QR code, levels Q or H should be used for industrial conditions (*e.g.* if there is the danger of staining the QR), while level L should be picked for clean conditions with data of big dimensions; this particular last case corresponds to the one that describes the DCC use cases.

To give a numerical measure of the provided improvements coming from the lowering of the error correction, it is possible to look at the versions of the different QR codes in the vaccination case.
The original DCC is using a version 19, since it is what is needed to represent 600 characters with a correction level Q (it can hold at most 644 characters), with a matrix size of 93×93 modules.
By lowering the level to L, the version drops to 13 (at most 619 characters) with a matrix size of 69×69, which is about 55% of the original size.

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 9, the Health Certificate version check in the *VerificaC19* iOS app is basically useless and passes even if the given payload has no "*HCx:*" prefix.
A similar check function is applied in the Android version, but details will be covered in Giacomo Arrigo's part.

By stripping it from the payload it is possible to save 4 characters on the final string.

Of course, this choice precludes the correct verification from other applications that may perform different checks on the input: to prevent this, choice is given to the users, in the form of an advanced setting toggle, to prefer maximum compression or compatibility.

To masquerade this level of complexity to the average user it has been decided to set the best compatibility option as the default one, hiding the possible choice selection behind a detail HTML element.
This design has been picked to not scare inexperienced users with options that they might not fully understand at first glance, keeping the app as simple as possible.

3.1.2 Light payload architecture

Our work group designed a proposal of a new version of the DCC architecture, where the COSE payload is stripped down to the bare minimum, in order to improve the QR density and consequently the system usability.

Our idea is to remove fields from the payload that are not taken into consideration when the validity rules check is performed.
Doing so, the final result will still be compliant with what is expected by the official EU DCC specifications [13].

In order to show the viability of the idea we implemented a Proof of Concept composed by the two main components:

- *Issuer*: it comprises two parts, a front-end and a back-end written in Python.
  The issuer task is to take user data as text input from a web page and to generate the new DCC QR code. To sign the COSE payload we used our own generated private keys with the ECDSA algorithm.
  This part is covered in Giacomo Arrigo's Master's Degree thesis.
- *Verifier*: it is a web-app that simulates the behaviour of a national verification app.
  A DCC QR is scanned from the camera, if it has valid structure and signature then the set of validity rules taken from the official EU DCC specifications [13] is applied against the payload content.
  The returned result gives information about the validity of the provided DCC.

Payload COSE design The new COSE has been designed with the same structure as the original one, but without some fields that have been selected

as non-necessary for the verification process.

In addition, for the proposed PoC, the signature process has been implemented to work only with ECDSA since it provides for smaller output compared to RSA.
More in detail, an ECDSA signature produced on the P-256 curve has a size of 64 bytes, while an RSA signature on a 2048 bit key results in 256 bytes, corresponding to the size of the modulus.

The “Signature Algorithm” *alg* field has not been removed from the header, so that it may support additional signature algorithms in the future (in case ECDSA proves to be too weak, or a better version becomes available).

The protected header of the COSE has the same structure as the original one, except that the key identifier has been designed to work in a different way.

Since the key identifier is used to quickly identify a public key from a list, the set of the public keys has been implemented as an array and the *kid* is now an integer index that is used to access directly the correct public key, needed to verify the signature.

The original *kid* is made up of 8 bytes, while the one that is provided in this design can be smaller: the value depends on the total number of keys, for example with the 1 byte that has been implemented in the proposed PoC it is possible to represent up to $2^8 = 256$ entries.

From the COSE payload the following fields have been identified to be removed, without compromising the business rules verification as per European specifications:

◇ Vaccination:

- *vp*, 14 bytes;
- *co*, 6 bytes;
- *is*, 26 bytes;
- *ci*, 43 bytes;

◇ Test:

- *nm*, 45 bytes;
- *tc*, 32 bytes;
- *co*, 6 bytes;
- *is*, 26 bytes;
- *ci*, 43 bytes;

◇ Recovery:

- *co*, 6 bytes;
- *is*, 26 bytes;
- *ci*, 43 bytes.

The measurements given in the last listing are relative to the examples that our group worked on: since each DCC may have different field sizes, the final size could be slightly diverse from the numbers that are provided in this thesis. Nevertheless, the provided measurements give a solid idea about the order of the improvements to the payload size.

The whole customized COSE (including headers, payload and signature) weights became like indicated in the next listing:

◇ Vaccination: old 344 bytes and new 248 byte ($\sim$72% of the original);

◇ Test: old 406 bytes and new 253 bytes ($\sim$62% of the original);

◇ Recovery: old 310 bytes and new 235 bytes ($\sim$82% of the original).

The work group explored the business rules that are to be applied against a payload in order to correctly verify it and kept only the needed information inside it.

The following snippets illustrate some examples of the new payload, one for each DCC category:

◇ Vaccination:

```
1  {
2      v: [
3          {
4              dn: 2,
5              dt: '2021-01-01',
6              ma: 'ORG-100030215',
7              mp: 'EU/1/20/1528',
8              sd: 2,
9              tg: '840539006'
10         }
11     ],
12     dob: '1990-12-07',
13     nam: { fn: 'CARFIZZI', gn: 'MARCO', fnt: 'CARFIZZI', gnt:
       'MARCO' },
14     ver: '1.0.0'
15 }
```

◇ Test:

```
{
    t: [
        {
            ma: 1232,
            sc: '2021-12-12T23:59:00+01',
            tg: '840539006',
            tr: '260415000',
            tt: 'LP217198-3'
        }
    ],
    dob: '1990-05-18',
    nam: { fn: 'ARRIGO', gn: 'GIACOMO', fnt: 'ARRIGO', gnt: 'GIACOMO' },
    ver: '1.0.0.'
    }
```

◇ Recovery:

```
{
    r: [
        {
            df: '2021-08-15',
            du: '2022-01-01',
            fr: '2021-08-01',
            tg: '840539006'
        }
    ],
    dob: '1990-05-18',
    nam: { fn: 'ARRIGO', gn: 'GIACOMO', fnt: 'ARRIGO', gnt: 'GIACOMO' },
    ver: '1.0.0.'
}
```

To give a numerical measure of the provided improvements coming from the smaller payload, it is possible to look at the versions of the different QR codes in the vaccination case.
The final base45 string has a length of 373 characters meaning that, with error correction L, it has version 10 (up to 395 characters) with matrix size of 57×57 modules: about 38% of the official DCC size and 68% of the proposed original optimized version.

Issuer The *issuer* is the component designed to generate and sign new DCCs.

A web page where the user can insert health data in text forms is presented: the inputted information is then passed to a python back-end that builds the corresponding CBOR payload.
The CBOR is then signed with ECDSA and a private key, generating the final COSE.
The final steps of the serialization process are then applied, meaning that the COSE is compressed with *zlib* and then encoded in *base45*.

The final QR is generated with the same parameters that have been applied to the optimization app, to ensure the smallest result possible.

Verifier The verifier consists in a publicly accessible web app that scans a DCC QR code via camera and then returns whether it is an authentic and valid certificate or not.

The application is quite simple since it only provides for a camera viewpoint, a button to repeat a scan and finally the result of the verification.

The verification pipeline structure that has been implemented is the one depicted in *"eHealth Network Guidelines on Technical Specifications for Digital Green Certificates Volume 4 - European Digital Green Certificate Applications"* [17]:

1. Scanning of QR code;
2. Base45 decoding;
3. COSE signature extraction;
4. CBOR-to-JSON decoding;
5. Fetching the signature's public key;
6. Verification of COSE signature with public key;
7. Verification of CBOR content;
8. Comparison with deny-list*.

The last step, identified with the asterisk, has not been implemented yet, since it is not strictly necessary to prove the PoC.
The provided verificator app does not work totally offline in the current stage of works, but will be made compliant to this specification in the future: a support to caching rules and signatures list must be implemented, with the fetching that happens only each 24 hours.

The verification process must ascertain the following concepts:

1. the QR content is a proper DCC, anything else cannot be accepted;

2. the signature is correct, meaning that the data inside the DCC is consistent and has been signed by a valid Certificate Authority;
3. the business rules, relative to the content type (vaccination, test or recovery), must be all valid, meaning that the certificate validity has not expired yet.

The business rules have been implemented in CertLogic format, by our work group, starting from the descriptions given in the official specifications.
Since the pandemic situation can variate quite rapidly and changes to the rules can be enforced on criteria that ranges on a member states basis, our group took the decision to implement the business rules as a generic European standard and not to customize them basing our work only on one state.
Nonetheless, new rules or modification to existing ones (*e.g.* changing the validity length) can be added without any trouble at any given time, due to the modular structure of the rules design.

3.2 Practical implementation of the apps and systems architecture

3.2.1 Optimization app implementation

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.
The benefit of having such functionality is to be able to have the new QR code displayed on a secondary monitor, giving the possibility of being able to scan it with a smartphone to be inserted in a wallet application (*e.g.* Apple iOS 15.4 implemented functionalities to officially add the EU DCC into the integrated Wallet by directly scanning the QR with the camera app).

The application has been designed to utilize plain JS so that it could run in a local environment: by operating in this way, no personal information is sent to the web server.

The graphical aspects of the app have been implemented via the Bootstrap 5 framework, locally imported. This means that while its version might not be instantly updated when a new one comes out, it is possible to fully deploy and utilize the app in an offline environment.
The described implementation has been chosen to give further proof that the app does not send any sensitive data to a server: a test of this can be achieved by launching and running the app in a local desktop while in airplane mode.

Architecture description Main components list and directory structure:

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

The *lib* directory has all the external libraries needed to perform the computations. As already stated, it has been decided to store them locally to depend the least possible on external servers.
Some of the libraries needed in the project that were implemented to work only in Node.js (server side) where bundled with *browserify* to be able to use a node-style *require()* with the *<script>* tag [5].

In the *lib* path is also stored the *lang.js* file that stores the strings for different languages; since the app can be utilized in a multitude of countries, the language selection has been implemented in a modular way. To insert support to new languages it is just needed to translate the strings in a correct way and add a new button to the HTML page. At the current time only support to English and Italian languages is present.

The *index.html* file has the HTML components of the page. It is a basic page with a file input form to let the user to provide the QR image, a button to activate the camera scanning functionalities to import the QR directly, an advanced settings section and the buttons needed to let the user operate with our app.
A *FAQ* section has been inserted, to provide the user with the most possible information about the app and how it operates. Advanced users that want more details can consult the *Advanced FAQ* sections or directly inspect the source code from GitHub.

The *core.js* has all the JavaScript functions to perform the following operations:

- load the selected language;
- listen on a image file input form;
- scan a QR directly from camera;
- optimize the read DCC into a new smaller image;
- download the optimized QR;
- show FAQ content.

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.

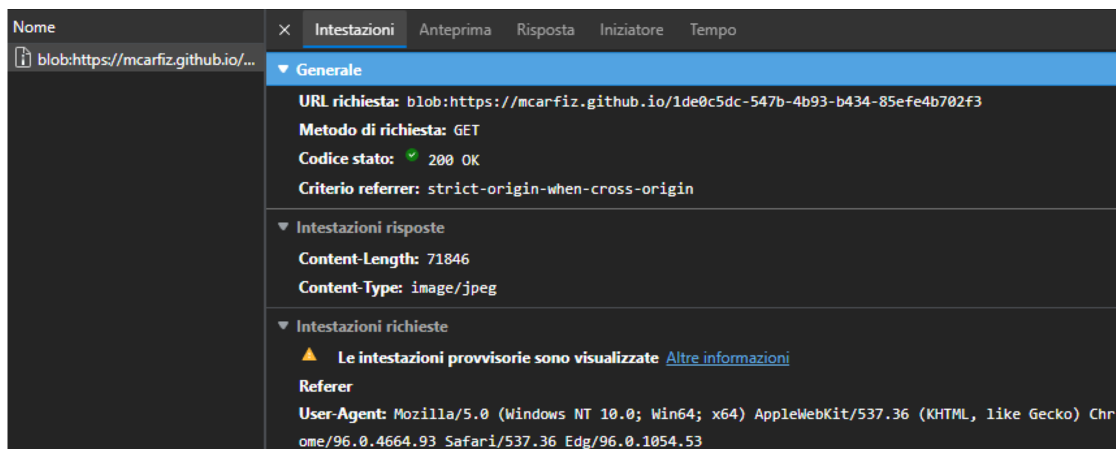


Figure 10: Blob creation request.

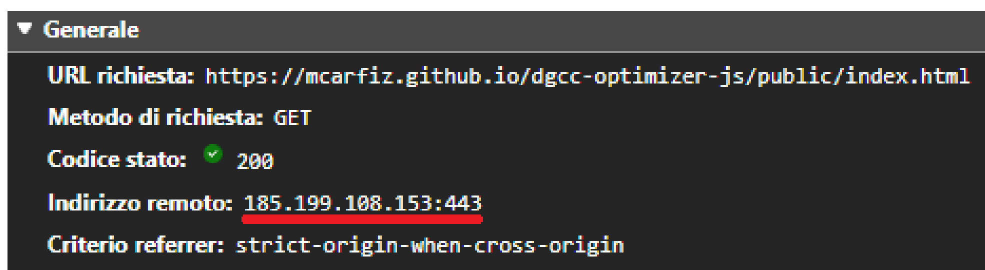


Figure 11: Local address for the blob request.

In alternative, there is an option to scan a QR directly from the camera; the scanner will default to the rear camera when available.
Of course, the user must give explicit consent to access its camera and this can only happen over a https connection.

If a 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 only needed for local computations: the whole app can be launched offline and would still work.

In the case of manually loading a file, JavaScript requires the creation of a URL object in order to access the user-supplied image; this process is considered a network request by some browsers, as shown in Figure 10.
Nevertheless, this request is not actually sent over the network and proof of this is the fact that generation can be done offline and the request does not have a remote address as depicted in Figure 11.

For the decoding of the QR from an image file we used the *qr-scanner* [9] JS library; there is also the option to scan the code directly from a camera thanks to the *Html5-QRCode* [8] library.

If the input file is a valid QR, then it is decoded using the *dcc-utils* [23] library

from the official Italian Ministero della Salute GitHub repository.

If the decoded payload is a valid COVID-19 certificate then the computations go on, else an error is thrown.
If the "Support only VerificaC19 Italian app" option has been checked, then the string is stripped from the first four characters that constitute the health certificate version prefix.

The decoded base45 string is then passed to the QR generation library, *qr-code-styling* [7] that renders the 2D image into a canvas in the HTML page. This library has been chosen 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 button is shown: it allows to download the QR code as a *JPEG* image.

If a QR code optimized for the Italian app 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 cut of the first four characters that should contain the Health Certificate version.

Privacy concerns This application has been designed with a clear idea: no personal data would be stored in the server side.

This is why the implementation was achieved only with HTML and JavaScript technologies: a basic, easy to use, lightweight, multi-platform design that could run in a local environment.

If the app wasn't stored in a server, it could be even possibly work offline.
It is also possible to run it on desktop as a standalone multi-platform app, thanks to the Electron framework.

To provide a better layer of transparency, the source code is available at this link https://github.com/mcarfiz/dgcc-optimizer-js.

How to use A simple guide to use the application:

1. Open app
2. Click “Choose file” or “Scan QR Code” and upload a DCC image (the library will find a QR code even in a image with other stuff in it) - Figure 12;
3. Wait until the optimized code is generated and shown in the page - Figure 13;
4. Click “Download QR code” and choose where to save the JPEG file.

An example error message (in this case when a wrong image is uploaded) is shown in Figure 14. In Figure 15 it is possible to see the "Advanced settings" menu opened and shown as described in point 3.1.1.

3.2.2 Verification app implementation

The verificator is a web-app written in JavaScript and HTML with the use of Bootstrap 5 framework.

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

Similar considerations to those that were made for the optimization app regarding privacy issues are still valid for this implementation, with the side note that the app is not yet completely offline-compatible, as described in the following points.

The source code can be found at this link https://github.com/mcarfiz/dgcc-verifier-js-custom, while a demo is available at this link https://mcarfiz.github.io/dgcc-verifier-js-custom/.

Architecture description Main components list:

- *index.html*
- ./data/
  - */rules/*
  - *public_keys.json*
- ./js/
  - *core.js*

The *js* 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 camera scan component.
When a reading is complete its result will be shown in a green or red box, depending on the outcome, at which point a button to perform a new scan will appear.

The *core.js* has all the JavaScript functions, in particular it allows to:

- read a QR from camera;
- fetch the correct business rules and the value sets;

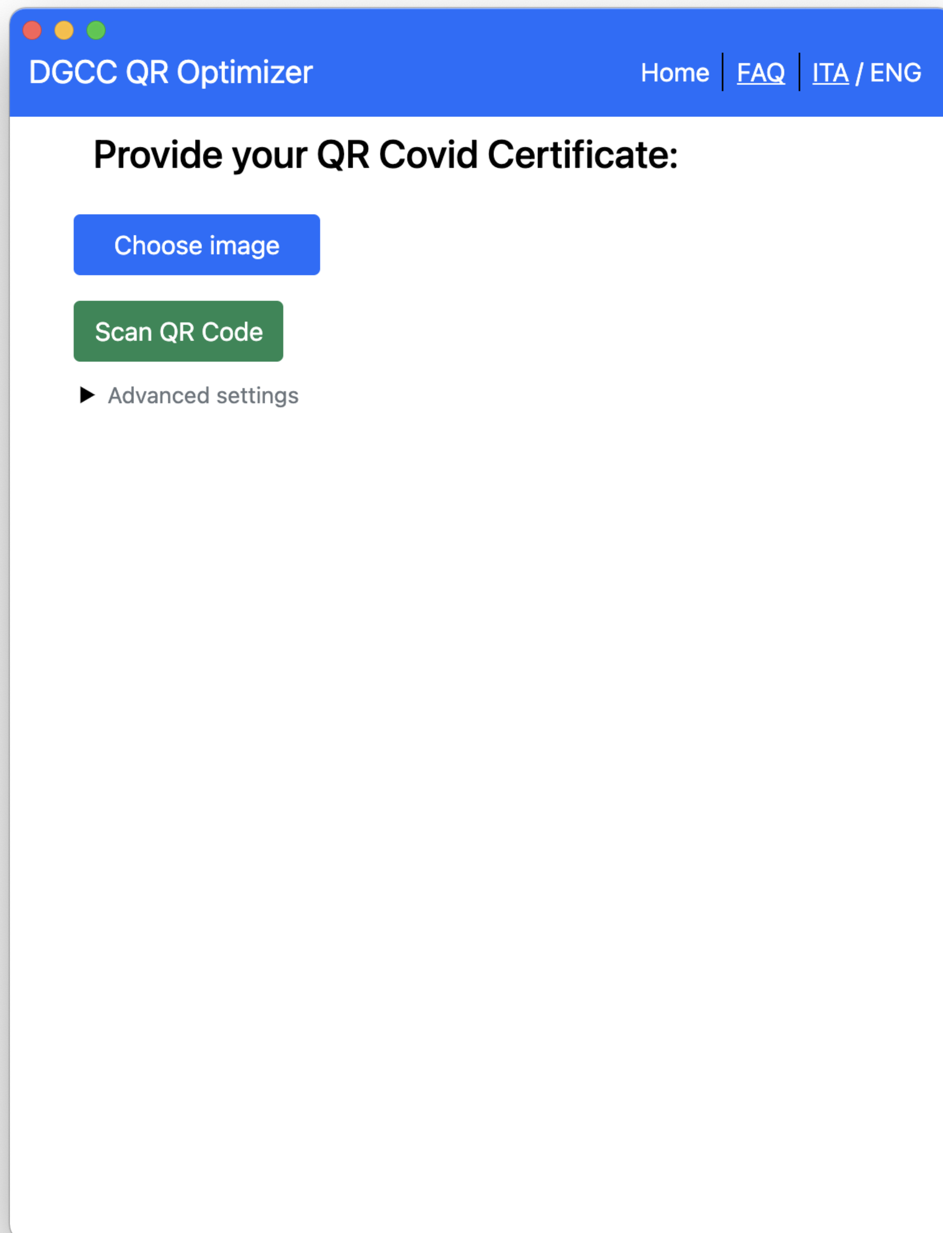


Figure 12: DCC QR optimizer starting screen.

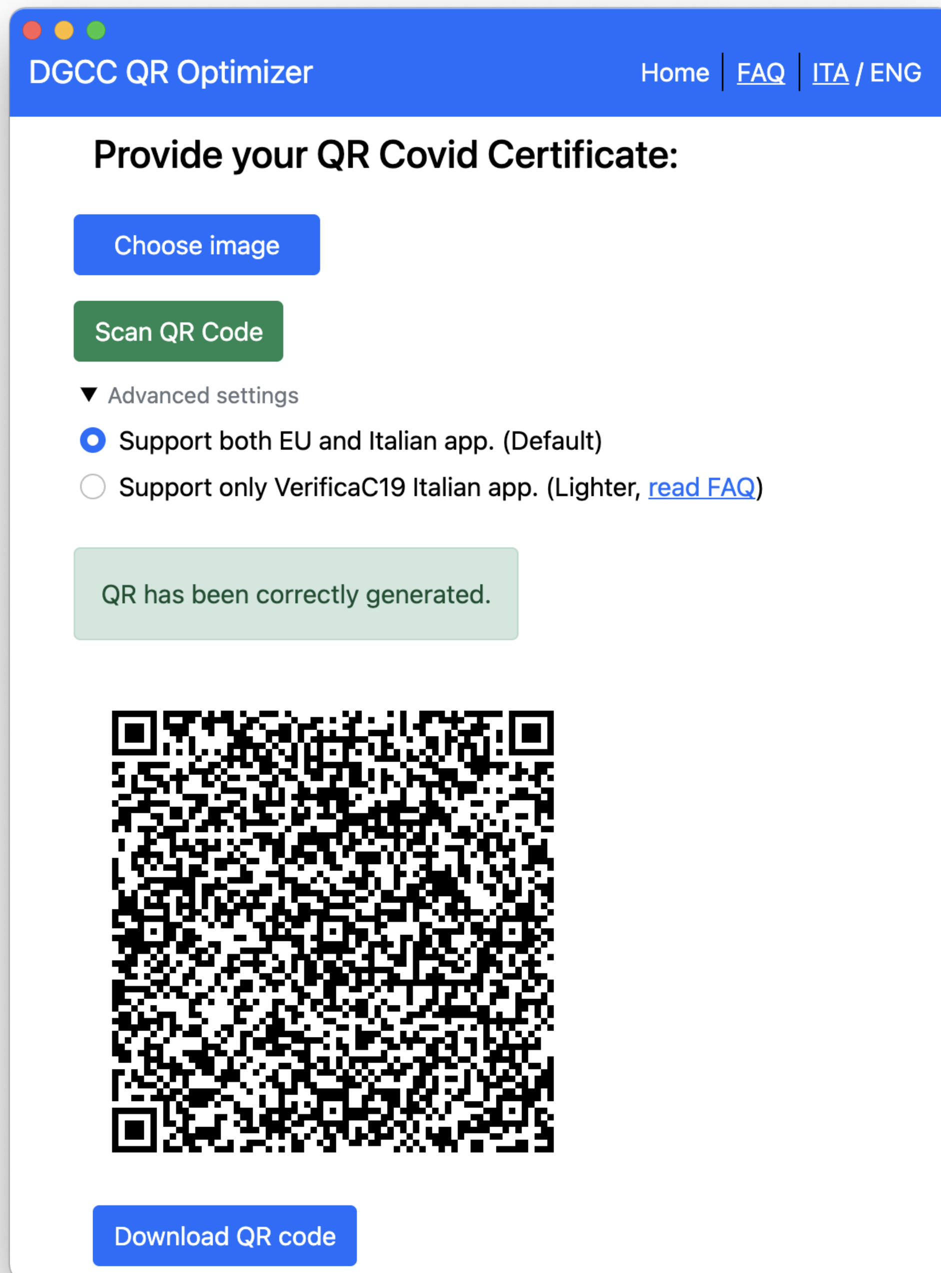


Figure 13: DGCC QR optimizer successful generation.

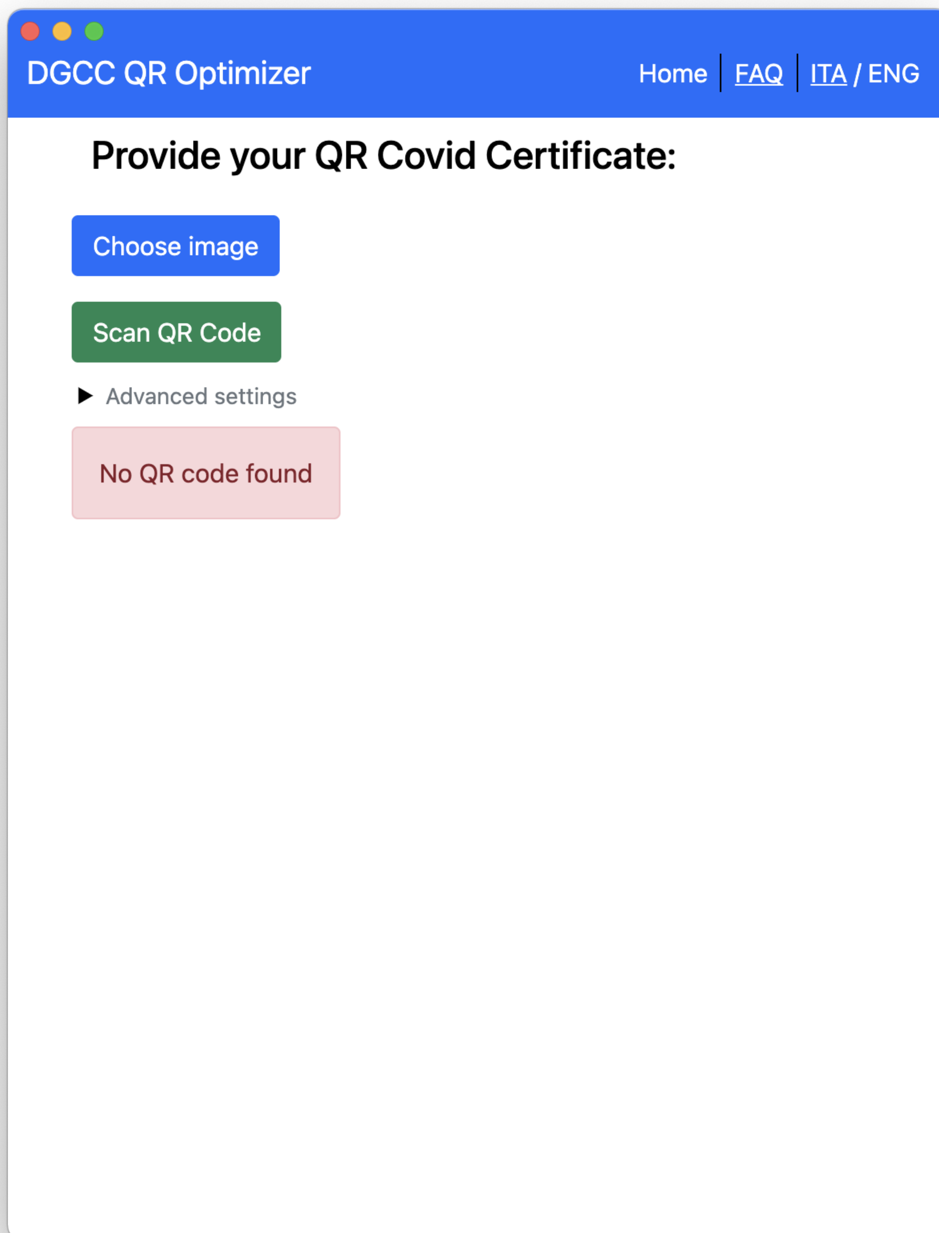


Figure 14: DGCC QR optimizer error message.

▼ Advanced settings

- Support both EU and Italian app. (Default)
- Support only VerificaC19 Italian app. (Lighter, read FAQ)

Figure 15: DGCC QR optimizer advanced settings detail.

◇ fetch the public keys list;

◇ verify the DCC signature;

◇ verify the DCC content against the CertLogic business rules.

The *data* folder contains two important elements: first the “public_key.json” file, which contains each DSC public key organized in an array, and the *rule* folder, which holds the *.json* rules files (an example of rule is shown at point 3.2.2).

The app is designed to have a QR camera scanner: thanks to the “Html5-QRCode” library [8] the cam form starts at the opening of the application, defaulting to the back camera if available.

The user must explicitly give consent to access camera the first time the app is run to utilize the functionalities and this can only happen over a https connection.

The library necessitates a callback function called *onScanSuccess* that is invoked when a correct QR is read: the scanning is then halted and the resulting base45 string is deserialized into a COSE structure via the “dcc-utils” library [23].

If the DCC building fails, then an error (red colored) with no private information is returned and shown to the user.
Whenever a response, be it positive or negative, is given, then a button to restart the QR camera scanning is shown.

If the DCC has been built successfully, the rule fetching begins.
A rule is a *JSON* file that contains what is depicted in point 2.6, in particular in Table 1.

First of all, the app fetches the general rules and pushes them into an array, basically used as a “queue” (First In First Out) data structure; then a check is performed to ensure whether the payload contains data about a vaccination, a test or a recovery.
This is performed by checking if the payload, which is itself a *JSON*, contains a field called “v”, “t” or “r”.
When the type has been attested, the fetching of the correct rules happens. Note that discriminating the type of DCC content is done only for performance reason, since no useless fetches are performed; in any case, if all the rules were to be fetched the check would still perform as expected, considering that if a rule finds an “incorrect” payload it returns *true* (*e.g.* all the VR rules provide *true* as response when checking a recovery payload).

The app then fetches from the file “*public_keys.json*” the public keys for each country (note that each country can have multiple *DSC*).

When all the data has been collected it is time to perform the check on the signature field of the *COSE*: it is necessary to examine if it is consistent with the payload content (ensuring that it has not been tampered) and that it is verifiable against a Certificate Authority public key.

To perform the signature validation, the correct public key (the point on the curve, described by its x and y coordinates) is selected by using the *kid* header field and the "cose-js" library [6] is called to perform the check.
As stated in point 3.1.2, our implementation only utilizes *ECDSA* with curve *P-256* to ensure the least space needed for the final DCC.

The public keys list object has been structured as an array, meaning that each key is now stored in a position given by its index and not by an external identifier.
This solution improves the key search time performance from an O(n), with n keys, to an O(1) since we directly access the needed position (position = kid). If no entry is available for a given key, then the signature verification must fail.

If the signature cannot be verified with the given key identifier, then the verifier cannot prove the validity of the certificate content and therefore an error with no user information, since there is no guarantee on what is inside the payload fields, is displayed (Figure 17).
This error can be thrown when a payload content has been tampered, since even one bit change in the CBOR fields of the payload leads to a totally different signature generation. This last example in particular underlines the futility of having a high level of error correction when even one wrong bit can invalidate the whole verification process.
Another possible mismatch case is the one happening when the DCC is signed with a private key that is not valid. The user could be trying to craft its own DCC, impersonating a Certificate Authority: no corresponding public key should verify the signature and therefore the verification must fail.

When the rule array is ready and the signature has been established as valid, each element of the vector will be dequeued, following a *FIFO* order, and thanks to the Rule Engine implemented in the *dcc-utils* library [23] every rule is applied to the payload.

If the signature check returns a positive result, then it is time to perform the rules check.
A function that ensures that each rule in the array is valid, calling the rule engine implemented in the *dcc-utils* library, performs the check and if it finds a rule that is not verified on the payload content, it halts the loop and returns a negative result.
When all the rules in the array have passed, a positive result is given back.

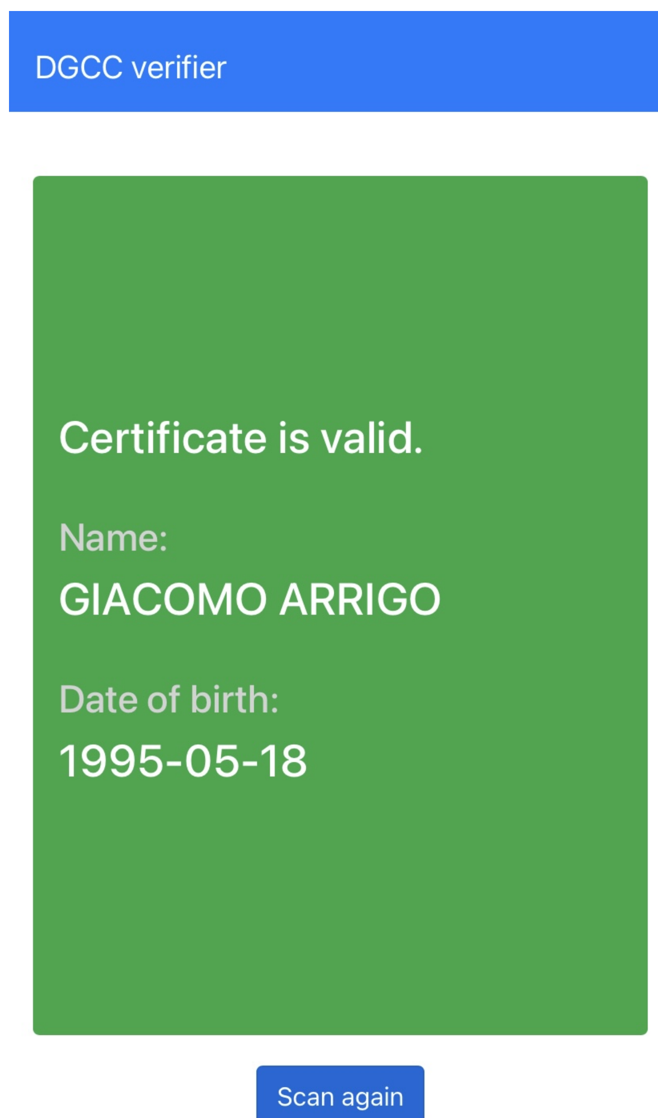


Figure 16: Verifier "success" message.

If the rules check returns a positive response, a green confirmation screen containing name, surname and date of birth, is presented to the user (Figure 16): showing basic personal data is needed so that the person who is trying to verify a DCC can cross-check the shown data with a valid ID to ensure that no impersonation is taking place.

If the rules check fails, then an error result is given back to the user along name, surname and date of birth contained in the payload as shown in Figure 18.
In this case the payload has correct and valid structure (it passed the *dcc-utils* checks), a valid signature meaning that the payload is consistent and has not been tampered, but problems arise when the business rules were checked (*e.g.* a test that is too old to be relevant and acceptable).

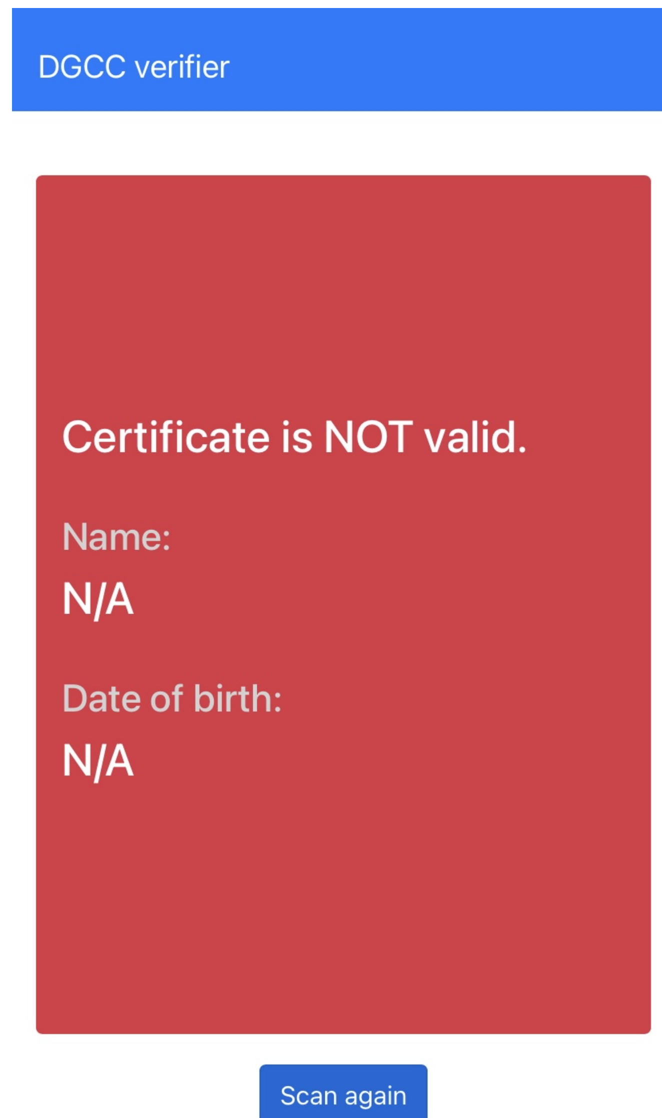


Figure 17: Verifier "signature verification error" message.

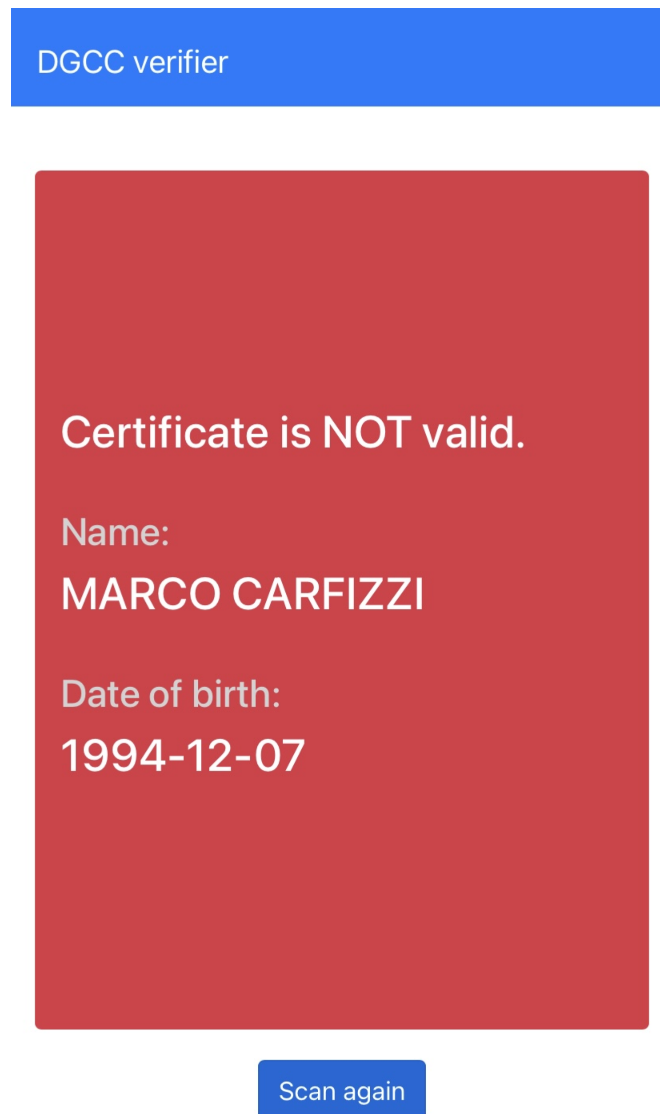


Figure 18: Verifier "certification not valid error" message.

Rules logic example This paragraph has the goal of illustrating a practical example of a rule logic implementation.

The following code snippet is an example of the *VR-EU-0002* rule logic, remembering its description “The vaccination schedule must be complete (e.g., 1/1, 2/2)”:

```
"Logic": {
    "if": [
      {
        "var": "payload.v.0"
      },
      {
        ">=": [
          {
            "var": "payload.v.0.dn"
          },
          {
            "var": "payload.v.0.sd"
          }
        ]
      },
      true
    ]
}
```

The logic of the last example can be unrolled as follows: if the payload has a vaccine ("v") element, then return the boolean result of “dn (last received dose) $\geq$ sd (total series of doses)”; if no *v* element is found, then return *true* (meaning that the payload is not a vaccination and there is no need to check the rule validity).

4 Conclusions

4.1 Relevant aspects

In this section the main points of the practical results that have been described are summarized:

- a web application to shrink and reduce the QR code density has been designed, created and published, and is available to public usage.
  The optimizations that our work group applied to the original QR have been proven to be working with a number of real European verification apps, yet it should theoretically be fine with each one of them: this thesis proposes that the listed optimization can be used and implemented in the real world;
- a PoC of an architecture that employs a DCC with a smaller payload, composed by two applications (an issuer and a verifier), has been designed and created.
  Since this system is a demo it is not intended to be submitted as an official proposition, yet.
  With further work and improvements we will be able to provide a definitive proposal in the near future.

In addition to the practical results it is important to underline the results that have been achieved, in particular the ones regarding the QR optimizations: it has been described in this thesis why the proposed results should be a preferred solution to be adopted, opposed to the inefficient current one.
More work will follow to finalize a definitive proposal to the payload compression problem, with further details that will be covered in Giacomo Arrigo's thesis.

4.2 Next steps

The future developments of the work will follow both main paths that have been covered in this thesis.

In the first place, the group will work on improvements of the proposal of the lighter DCC; supplemental investigations and studies about the fields size, encoding optimization and further compression will be pursued.
Examples of upcoming attempts would be to use UNIX TimeStamp as date format, or try to substitute long string values with magic numbers.
The verificator will be improved to fully match the Volume 4 specifications [17], meaning that it must support full offline verification process, with rules and signature fetch that happens only after 24 hours and is cached locally, in addition to a blacklist system to support for public keys revocation.

The optimization app has reached a stage that satisfies the work group in terms of functionalities, so the next steps will consist in trying to research about user satisfaction with the improved system: more in detail, an investigation about the user content level, regarding both the application and the resulting less dense QR code.
A first proposal for a test would be to ask students of the scientific campus of Ca' Foscari University of Venice to evaluate the application and to try to scan both a regular DCC and our improved QR: later they will be asked to fill a survey about their experience and if they find any tangible improvement with the utilization of our optimizations.

5 References

References

[1] European eHealth network - Digital Covid Certificate. *Digital COVID Certificates: Business Rules*. URL: https://github.com/ehn-dcc-development/dgc-business-rules.

[2] A-SIT. *A-SIT ENH test suite*. URL: https://dgc.a-sit.at/ehn/testsuite.

[3] P. Hoffman C. Bormann. *RFC 8949 Concise Binary Object Representation (CBOR)*. 2020.

[4] Michael Droettboom. *Understanding JSON Schema*. URL: https://json-schema.org/understanding-json-schema/about.html.

[5] http://browserify.org. *browserify*. URL: https://github.com/browserify/browserify.

[6] https://github.com/erdtman. *cose-js*. URL: https://github.com/erdtman/COSE-JS.

[7] https://github.com/kozakdenys. *QR Code Styling*. URL: https://github.com/kozakdenys/qr-code-styling.

[8] https://github.com/mebjas. *Html5-QRCode*. URL: https://www.npmjs.com/package/html5-qrcode.

[9] https://github.com/nimiq. *QR Scanner*. URL: https://github.com/nimiq/qr-scanner.

[10] ECMA International. *The JSON Data Interchange Format*. 2013.

[11] ISO. *ISO/IEC 18004:2015 - QR Code bar code symbology specification*. 2015.

[12] August Cellars J. Schaad. *CBOR Object Signing and Encryption (COSE)*. URL: https://datatracker.ietf.org/doc/html/rfc8152.

[13] European Commission - eHealth Network. *EU DCC Validation Rules*. URL: https://ec.europa.eu/health/sites/default/files/ehealth/docs/eu-dcc_validation-rules_en.pdf.

[14] European Commission - eHealth Network. *Guidelines on Technical Specifications for Digital Green Certificates Volume 1*. URL: https://ec.europa.eu/health/sites/default/files/ehealth/docs/digital-green-certificates_v1_en.pdf.

[15] European Commission - eHealth Network. *Guidelines on Technical Specifications for Digital Green Certificates Volume 2 - European Digital Green Certificate Gateway*. URL: https://ec.europa.eu/health/sites/default/files/ehealth/docs/digital-green-certificates_v2_en.pdf.

[16] European Commission - eHealth Network. *Guidelines on Technical Specifications for Digital Green Certificates Volume 3 - Interoperable 2D*

Code. URL: https://ec.europa.eu/health/sites/default/files/ehealth/docs/digital-green-certificates_v3_en.pdf.

[17] European Commission - eHealth Network. *Guidelines on Technical Specifications for Digital Green Certificates Volume 4 - European Digital Green Certificate Applications*. URL: https://ec.europa.eu/health/sites/default/files/ehealth/docs/digital-green-certificates_v4_en.pdf.

[18] European Commission - eHealth Network. *Guidelines on Technical Specifications for Digital Green Certificates Volume 5 - Public Key Certificate Governance*. URL: https://ec.europa.eu/health/sites/default/files/ehealth/docs/digital-green-certificates_v5_en.pdf.

[19] European Commission - eHealth Network. *JSON Schema Specification*. URL: https://ec.europa.eu/health/sites/default/files/ehealth/docs/covid-certificate_json_specification_en.pdf.

[20] J-L. Gailly P. Deutsch. *ZLIB Compressed Data Format Specification version 3.3*. URL: https://datatracker.ietf.org/doc/html/rfc1950.

[21] D. van Gulik P. Faltstrom F. Ljunggren. *The Base45 Data Encoding*. 2021. URL: https://tools.ietf.org/pdf/draft-faltstrom-base45-07.pdf.

[22] T. Pornin. *RFC 8152 - Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)*. URL: https://datatracker.ietf.org/doc/html/rfc6979.

[23] Ministero della Salute. *DCC Utils*. URL: https://github.com/ministero-salute/dcc-utils.

[24] M. Jones - E. Wahlstroem - S. Erdtman - H. Tschofenig. *CBOR Web Token (CWT)*. URL: https://datatracker.ietf.org/doc/html/rfc8392.

[25] Jeremy Wadhams. *JsonLogic*. URL: https://jsonlogic.com/.