Ca' Foscari
University
of Venice

Master's Degree
in Computer Science
Software Dependability and Cyber Security

Final Thesis

Automated usability analysis of secure QR code

Supervisors
Ch. Prof. Riccardo Focardi
Ch. Prof. Flaminia Luccio

Graduand
Andrea Cester
Matriculation Number 864008

Academic Year
2021 / 2022

Abstract

During the COVID19 pandemic in 2019-2020 a large number of states introduced a certification called EU Digital COVID Certification. The validation of these certificates was done by scanning a QR code whose verification was not always easy and usable. In this thesis we have simulated the typical cases in which a QR is scanned in order to analyze the usability of the scan based on the variation of the data contained and the level of error correction applied.

Contents

Introduction

A QR code, short for Quick Response code, is a two-dimensional barcode that can be scanned using a smartphone camera or a dedicated QR code reader. QR codes were first developed in 1994 by Denso Wave, a subsidiary of Toyota, for use in the automotive industry. However, they have since found a wide range of applications in various industrial context, including retail, transportation, and healthcare.

QR codes can store much more information than traditional linear barcodes, such as product information, website URLs, and contact information. They are a versatile and convenient technology that can be used for a wide range of applications, from providing product information to facilitating transactions and digital identification.

One of the biggest historical use was during the global pandemic called COVID-19. In the early months of this pandemic, a quarantine was imposed on the population to reduce the spread of the virus. With the introduction of vaccines, a secure digital certificate containing information regarding the vaccination status or recovery from SARS-Cov-2 of each citizen was introduced. To verify that each citizen was cured or vaccinated, confirmation applications were created and available to the entire population on each smartphone or tablet device.

The general purpose of this thesis is to analyze the performance of error correction on QR codes. First we decided to study two QR codes with the same data encoded, but with different error correction levels, the first one with the lower and second with the higher. In the second part, we analyze the differences between QRs with different data encoding by taking as samples the EU Digital COVID Certification (DCC) and the experimental version proposed by Marco Carfizzi [6] and Giacomo Arrigo [2] in their master thesis. In our study we investigate the relation between the level of error correction and the usability of the standard QR code and the new version proposed in [2, 6].
The thesis is structured as follow. The first chapter defines the QR code, giving an explanation of its structure and how data are encoded. The second chapter analyses the libraries used for the implementation of the tests. The third chapter describes the data structure utilized in the standard DCC and of the new DCC proposed in [2]. The fourth chapter gives an introduction about the structure of the tests. The last chapter analyses the data from the results of the tests. We conclude with future works directions.

Chapter 1

QR codes

QR Codes have gained widespread popularity in many areas, surpassing traditional barcodes in certain applications. This is often attributed to their ability to store much more data than a standard barcode, holding up to 7,089 characters in contrast to the maximum 20 digits of a barcode. Additionally, QR Codes are highly versatile and extensible, making them a more desirable option than barcodes. Furthermore, QR Codes have the ability to encode the same amount of data in a much smaller space, about one-tenth of a traditional barcode. Another key advantage of QR Codes is that they can be scanned from any angle, as the three specific squares positioned in the corners of the symbol and the alignment blocks allow the scanner to determine the correct orientation of the image (see Figure 1)[16].

Figure 1: QR

1.1 QR Data encoding

The data encoding of a QR code refers to the process of converting the data to be encoded into a specific pattern of black and white modules, which define the QR code. QR codes use a specific pattern of black and white modules, known as modules, to represent the data encoded in the code. The data is arranged in a specific way to make it possible to read the code with a QR code reader.

- **Numeric encoding** is used to encode digits from 0 to 9. It uses a more compact representation and is suitable for encoding numbers and short text.
- **Byte encoding**, by default, it is used to characters from the ISO-8859-1 character set. However, some QR code scanners can automatically detect if UTF-8 is used in byte mode instead.
- **Kanji encoding** is used to encodes Japanese characters. It is suitable

for encoding text in Japanese and other languages that use similar character sets.

- **Alphanumeric encoding** is used to encode the decimal digits, uppercase letter, special characters like $%*+-/: and the blank space.

1.2 QR data distribution

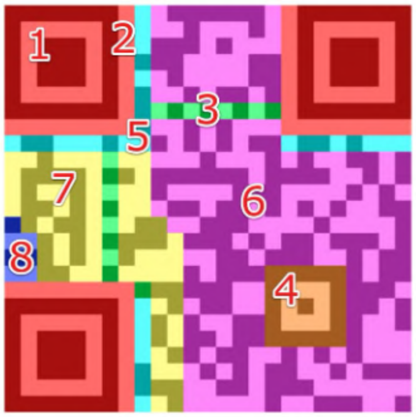


Figure 2: QR data distributin [16]

Detection markers

The detection markers (number 1 in Figure 2) are a crucial part of a QR code, as they help the QR code reader to locate the code and to determine its orientation, which is necessary for correctly reading the code. They are placed in the corners of the QR code and are made up of a specific pattern of black and white modules that can be easily distinguished from the rest of the code.

Each detection marker has a separator that are lines made up of light modules that serve to distinguish the finder patterns from the rest of the code (number 2 in Figure 2). They are 1 module wide and can only be found along the edges of the finder patterns that face the interior of the code.

Alignment markings

The alignment markers (number 4 on figure 2) are smaller then the detection markers, the scope of this little marks is to help straighten out the Qr code when it is drawn on a curved surface.

The number of these markers depends on the dimension of the Data stored in the QR. Version 1 does not use the alignment marker because there is not enough space to display the data and the marker.

Timing pattern

Combination of black/white modules on the QR code to support the configuration of the data grid. In base of the line the decoder determines the dimension of the data matrix (number 3 in figure 2).

Version information

The version information (number 8 in Figure 2) of QR Codes can have 40 different values, each of which has a varying number of modules. The first version, Version 1, has 21x21 modules, with a maximum of 133 of those modules being used to store encoded data. On the other hand, the largest QR Code version, Version 40, has 177x177 modules and can hold up to 23,648 data modules. The number of modules increases as the version number increases and the error correction level decrease [16].

Format information

The Formation Information section (number 5 in Figure 2), located next to the separators, contains 15 bits of information which includes details about the error correction level of the QR Code and the masking pattern selected [16].

Error correction and data

The data and the error correction's blocks (number 6 and 7 in Figure 2) contain the data and the error correction that permit a restoration of the missing or damage data.

The error correction bits are created by dividing a message polynomial by a generator polynomial, the specific steps are discussed forward in the section.

Quiet zone

The quiet zone is a blank area that surrounds the QR code, it is used to separate the code from other elements in the image and to make sure that the QR code reader can correctly locate the code and to distinguish it from other elements in the image. The size of the quiet zone can vary depending on the specific QR code reader and the environment in which the code will be used, but typically it should be at least 4 modules wide on all four sides of the QR code.

1.3 Error correction

QR codes can have different levels of error correction built in, which affects their robustness against damage or distortion. QR codes can be classified into four levels of error correction: L (low), M (medium), Q (quartile), and H (high) (see Table 1). The higher the level of error correction, the more data can be recovered even if the QR code is partially damaged or distorted. However, this also means that the QR code will be larger and may require more processing power to scan. To select error correction levels, various factors such as the operating environment and QR Code size need to be considered. Level Q or H may be selected for factory environment where QR Code get dirty, Level L and M may be selected for clean environment with the large amount of data.

Error correction level	**Error correction percentage recover**
Error correction level L	Recovers 7% of data
Error correction level M	Recovers 15% of data
Error correction level Q	Recovers 25% of data
Error correction level H	Recovers 30% of data

Table 1: Error correction table

The QR Code error correction feature is implemented by adding a Reed-Solomon Code to the original data.
The Reed-Solomon code is a powerful error-correcting code that is widely used in QR codes. The code works by using an algebraic structure and mathematical algorithm to generate check symbols, which are added to the original data, the algebraic structure used is the Galois field, and the algorithms are the Euclidean algorithm and the Forney's algorithm.
The Reed-Solomon uses the Galois Fields (Infinite Fields), to represent and perform arithmetic operations when encoding the data. In Reed-Solomon, the data are first divided into blocks according to the number of codewords, and represented as polynomials over a Galois field. The encoding is performed by evaluating the polynomials adding the evaluations to generate the

codeword. By expressing data as a vector in the Galois Field, mathematical operations can be performed with ease and efficiency to scramble data.
The elements of Galois Field $gf(p^n)$ is defined as

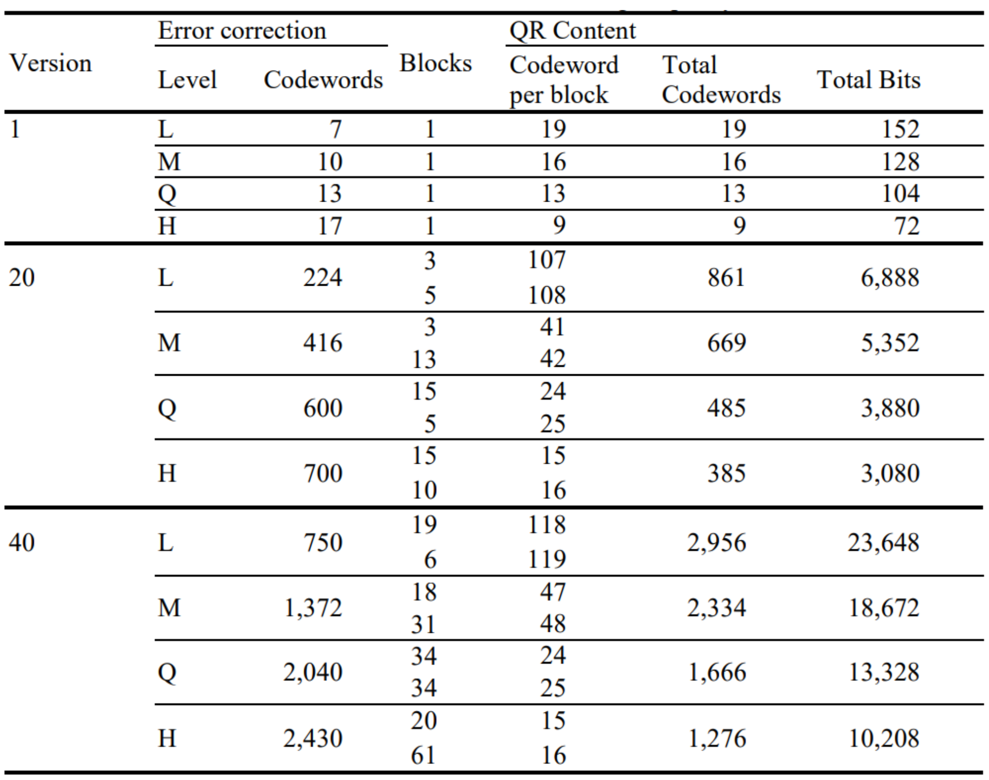
$$\begin{aligned} gf(p^n) =& (0, 1, 2, \cdots, p-1) \cup \\ & (p, p+1, p+2, \cdots, p+p-1) \cup \\ & (p2, p2+1, p2+2, \cdots, p2+p-1) \cup \cdots \cup \\ & (pn-1, pn-1+1, pn-1+2, \cdots, pn-1+p-1) \end{aligned}$$

where the polynomial $p \in P$ and $n \in Z^+$. The order of the field is given by p^n while p is called the characteristic of the field. Also note that the degree of polynomial of each element is at most $n-1$ [4].

The standard process of decoding Reed-Solomon codes involves the calculation of the identification of an error locator polynomial , and the resolution of error values. To find the syndromes, the received message is interpreted as the coefficients of a polynomial $S(x)$ (called *syndromes*), where x is a codeword. The Euclidean algorithm is employed to derive the error locator and error evaluator polynomials from the computed syndrome polynomial that will be used to find the error positions and the error values. The Forney algorithm uses the error locator and the error evaluator to correct the errors present on the codewords. It is used as part of the process in decoding Reed–Solomon code[14].
If the number of errors exceed the maximum number available, the Reed-Solomon cannot restore the data.

Table 3 shows the maximum data capacity under different error correction levels in versions 1, 20 and 40. The error correction capability of a QR code is determined by its error correction rate. A higher error correction rate means that more errors can be corrected, but it also means that less data can be stored. In different versions of QR codes, the stored data is divided into several blocks and corresponding error correction codewords are generated to ensure error correction capability.

For example, in QR code version 20-L, there are a total of 8 blocks, with 3 blocks containing 107 data codewords and 5 blocks containing 108 data codewords, adding up to a total of 861 data codewords. The 224 error correction codewords are also divided into 8 blocks to ensure the error correction capability for each corresponding data block [24].



Version	Error correction		Blocks	QR Content		
	Level	Codewords		Codeword per block	Total Codewords	Total Bits
1	L	7	1	19	19	152
	M	10	1	16	16	128
	Q	13	1	13	13	104
	H	17	1	9	9	72
20	L	224	3	107	861	6,888
			5	108		
	M	416	3	41	669	5,352
			13	42		
	Q	600	15	24	485	3,880
			5	25		
	H	700	15	15	385	3,080
			10	16		
40	L	750	19	118	2,956	23,648
			6	119		
	M	1,372	18	47	2,334	18,672
			31	48		
	Q	2,040	34	24	1,666	13,328
			34	25		
	H	2,430	20	15	1,276	10,208
			61	16		

Figure 3: Qr code capability [24]

1.4 QR code Masks

A masking process is used to avoid symbols that might confuse a scanner, such as misleading shapes that look like the locator patterns and large black or white areas. The masking process consists in inverting certain modules depending by the penalty score of each masks.

The **Distribution lines of same colors penalty** check the QR, first row by row, and then column by column. In order to evaluate the first condition, rows are checked individually. If there are five consecutive modules of the same color, a penalty of 3 is added. If there are additional modules of the same color, an additional penalty of 1 is added for each module. After this, the columns are checked in the same manner. The total of the horizontal and vertical penalties are then added to obtain the first penalty score (see Figures 4 and 5).

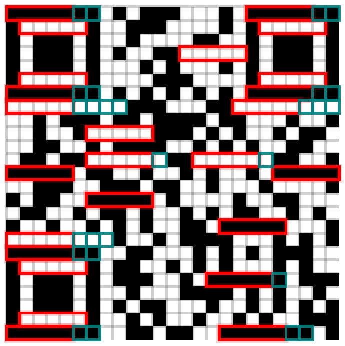


Figure 4: Distribution line penalty on row [12]

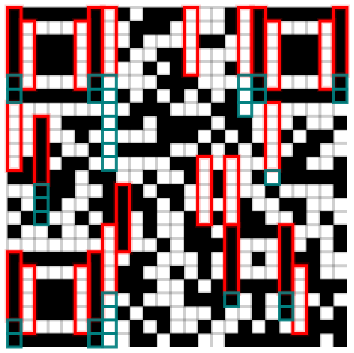


Figure 5: Distribution line penalty on column [12]

The **Data density penalty** method involves identifying blocks of the same color that are at least 2×2 modules or larger (see Figure 6). The QR code specification states that for a block of the same color with dimensions $m \times n$, the penalty score is calculated as $3 \times (m-1) \times (n-1)$. However, the specification does not provide guidance on how to calculate the penalty when there are multiple ways to divide up the solid-color blocks.

To simplify the calculation, a penalty of 3 is added for each 2×2 block of the same color in the QR code, including overlapping blocks. For example, a 3×2 block of the same color is counted as two 2×2 blocks, one overlapping the other. The final penalty score is the sum of all penalties from these blocks.

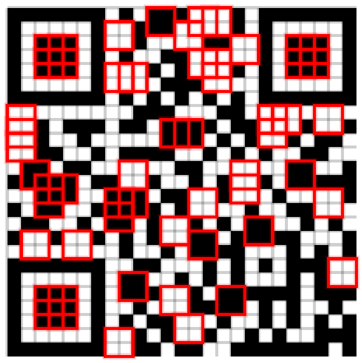


Figure 6: Data density penalty [12]

Block penalty evaluation condition searches for the pattern of dark-light-dark-dark-dark-light-dark with four light modules on either side. Whenever this pattern is detected, an additional 40 is added to the penalty score.

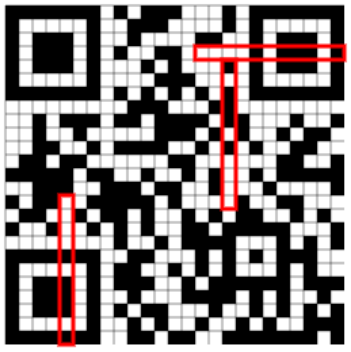


Figure 7: Block penalty [12]

The Proportion penalty gives the final evaluation condition assesses the ratio of light and dark modules in the QR code matrix. The process involves the following steps:

1 Count the total number of modules in the matrix.

2 Count the number of dark modules in the matrix.

3 Calculate the percentage of dark modules by dividing the number of dark modules by the total number of modules and multiplying the result by 100.

4 Determine the closest multiple of five to the calculated percentage, both lower and higher.

5 Take the absolute difference between each multiple of five and 50, divide the result by 5, and take the smaller of the two numbers.

6 Multiply the smaller number by 10 to get the final penalty score

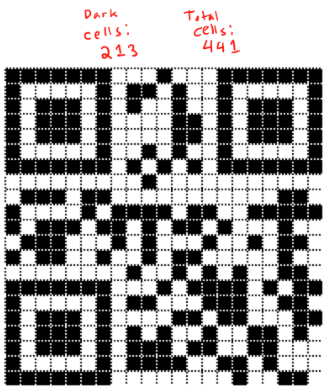


Figure 8: Proportion penalty [12]

These four penalty scores are used together to calculate the total penalty score, which is a measure of the quality of the QR code. The mask pattern with the lowest total penalty score is considered the best mask pattern for the QR code.

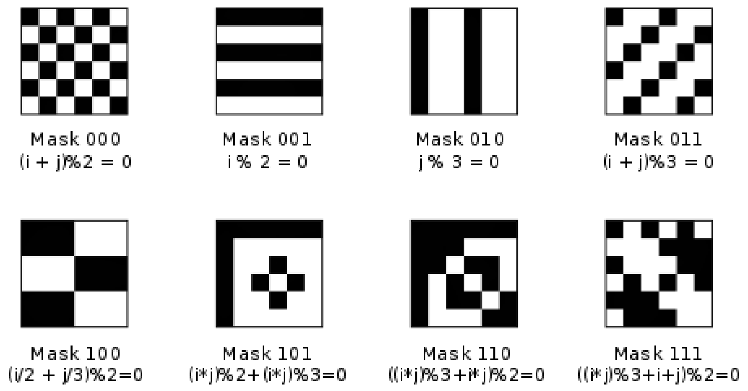


Figure 9: Data mask of a QR [1]

Conclusion In this Chapter we have introduced the concept of the QR codes by describing their structure. We then focused on explaining the error correction and how does it work. In the last chapter we discuss how the data mask is choose and apply. The importance of knowing these aspects lies in better understanding how works the QR code.

Chapter 2

Libraries

In this chapter, the libraries used in this thesis will be explained. The first one is the ZXing library, it was chosen as: it is an open source library, it contains several classes and functions that allow us to obtain the information regarding the various decoding steps of a QR; it was used for the creation of the Android app VerificaC19.
The second library instead is OpenCV, written in Python, with several methods useful for creating image distortions in the image.

2.1 The ZXing library

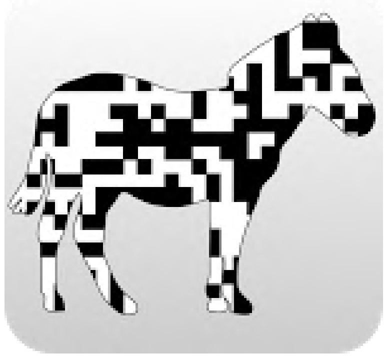


Figure 10: ZXing [20]

ZXing is an open-source library implemented by the team ZXing project in Java. Its scope is to provide a useful library that can be used to read 1D or 2D barcodes embedded website applications. Thanks to the great support of the community the library has stable ports to different languages [20].

1D product	**1D industrial**	**2D**
UPC-A	Code 39	QR Code
UPC-E	Code 93	Data Matrix
EAN-8	Code 128	Aztec
UPC/EAN Extension 2/5	ITF	MaxiCode, RSS-14, RSS-Expanded

Table 2: Format supported by the library [20]

As the Table 2 reports the library supports four 1D and four 2D dimensional barcodes.
Universal Product Code (UPC see Figure 11) is a bar code composed by twelve digits. The difference between the UPC-A and the UPC-E is the format of the information. The UPC-E is the compacted version of the UPC-A, obtaining a barcode smaller but with the same information more useful when the barcode is placed on small packages.

7 25272 73070 6

Figure 11: UPC barcode

The second 1D barcode of the library is the most used in the world and it is the European Article Number (EAN) (see Figure 12). Like the UPC it has multiple format but the most common are the EAN-13 for the majority of the utilization and the EAN-8 when the barcode is too big to be placed on the surface and it is needed a smaller version. In both formats the number indicates the digits used by the barcode.

9 780201 379624

Figure 12: EAN barcode

Even if the EAN is used as a standards for the GS1, the non-profit organiza-

tion that validate the standard used in the world, the UPC is acknowledged as a standard and can be used.
The difference between the two standard are minimal since the UPC is a subset of EAN and can be read from the 1D barcode without any problem. For the QR code format see Chapter 1. A Data Matrix (see Figure 13) is a bi-dimensional barcode with a higher information density than the majority of the barcode typologies. It has the capacity to include different error correction algorithms to improve the recovery property of the barcode. The Data Matrix can encode alphanumeric characters, symbols like kanji, and numeric digits in an efficient mode by using specific encoding schemes. The structure is a matrix composed by black and white modules, where each module represents a bit. All the modules of the matrix are stored in a frame. The detection marker is placed on the outermost square of modules in the frame. The dimension of the frame depends on the data size and all the code included in the error correction [9].

Figure 13: Data Matrix

The Aztec code is a 2D dimensional barcode (see Figure 14) defined by the standard ISO/IEC 24778 [13], it has the highest accuracy between all 2D barcode. What is most noticeable is the presence of the "eye-ball", called core, displayed at the center of the barcode. Each corner includes an orientation mark that allows the barcode reader to detect it even if the frame is rotated or mirrored. The Aztec Code uses the Reed-Solomon algorithm for error correction. Compared to the QR code the user can specify the percentage of the error correction (max 99%) of the data region codewords. The recommended percentage is 23%. This type of barcode can decode alphanumeric characters, numeric digits, byte data and symbols. Data are stored by a layer architecture with a clockwise direction starting form the first orientation pattern. It can go from 1 to a maximum of 32 layers. The smallest size is 15×15 and the greatest is 151×151. Since the presence of the core, the AZTEC does not need a quiet zone to help the detection of the code and that increase the displays of the frame on different location.

Figure 14: AZTEC code

The Maxicore (see Figure 15) is a barcode used to store information about packages. It is based on QR code and AZTEC code idea. Like AZTEC it has a "eye-ball" at the center of the frame. The disposition is a hexagonal grid instead of a square grid like in the QR code and instead of using square modules for the data it used dot. The structured portion of the message is stored in the inner area of the symbol, near the bull's-eye pattern. Maxicode use the Reed–Solomon error (ref section) correction to recover missing data. Even if it takes as reference the two most used barcodes it suffers of a poor presence of library and software in comparison with QR code and Aztec code.

Figure 15: Maxicode

ZXing library contains the following module:

Module	**Description**
core	The core image decoding library, and test code
javase	JavaSE-specific client code
android	Android client Barcode Scanner
android-core	Android-related code shared among *android*, other Android apps
zxingorg	The source behind *zxing.org*
zxing.appspot.com	The source behind web-based barcode generator at *zxing.appspot.com*

Table 3: Module of the library [20]

The core module is responsible for processing raw image data, extracting barcode information, and decoding the data encoded in the barcode supported see Table 3:

- *Aztec*: encoder, decoder and detector for Aztec code;
- *Common*: contains all the mathematically operation needed to compute the Galois Fields and Reed-Solomon operation;
- *Datamatrix*: encoder, decoder and detector for datamatrix barcode;
- *Maxicode*: contains only the decoder for maxicode barcode;
- *Multi*: module that permit the detection of multiple QR code in a single image;

- *Qr code*: encoder, decoder and detector for QR code;
- *Oned*: barcode reader and writer for each 1D barcode;
- *PDF417*: encoder, decoder and detector for PDF417 barcode.

The core module is built around a set of classes and interfaces that provide a common API for working with barcodes. The main classes include:

- *BarcodeReader*: this class is responsible for decoding barcode images. It takes an image file or in the case of a web app a stream video as input, and returns the decoded barcode data as output;
- *BarcodeWriter*: this class is responsible for encoding data into a barcode image. It takes in input a data string, a barcode format, the error correction level, and returns a barcode image as output;
- *reedsolomon*: it contains the two class for the Galois Fields, the Reed-Solomon decoder, the ReedSolomon encoder and the class for the exception error;
- *BarcodeFormat*: it is used to store Enumerates barcode formats known to this package ordered by alphabetic order;
- *Result*: it is a class that provide the output data of the decoded barcode data with extra information like the version etc...;
- *WriterOptions*: this class provides the base options for creating a barcode image, such as width, height, margin, error correction level, version, and encoding.

When the library is called to decode a QR, the following steps are computed, the same steps are applied for each 2D barcode:

1 *Detection of QR code markers*: the image is then analyzed the special markers used by the QR to provide the direction of the image. These markers include the three large squares located at the corners of the QR code and the smaller alignment patterns that are used when the QR contain a lot of data.

2 *Image perspective correction*: once the markers are located, the image is corrected for perspective distortion caused by the camera angle. This step is necessary to ensure that the QR code is in the correct orientation for decoding and also it is applied a cropping procedure on the image to remove the borders.

3 *Extraction of data modules*: before the extraction of data the image is reduced to a smaller image that represent a black or white square for each pixel. Then it is divided into small modules, each of them represents a single bit of data encoded in the QR code.

4 *Error correction*: the data blocks are passed through an error correction Reed-Solomon algorithm that used the Euclidean Algorithm, which corrects any errors caused by noise or damage to the QR code with the maximum percentage corresponding to the error correction level.

5 *Decoding of data*: after the application of the error-correction the result data blocks are then decoded to reveal the original data encoded in the QR code. The QR code decodes the data using the five encoding type: alphanumeric, numeric, byte, kanji and special symbol.

6 *Result extraction*: the decoded data are extracted and returned as the output.

When the library is called to encode data into a barcode image, it starts from the opposite order of the decoding.
The encoding data procedure of a QR code consist in the following steps:

1 *Data encoding*: the information to be encoded is transformed into a sequence of binary code words which will be represented by the modules of the QR code. These code words are produced using the suitable data mode like numerical, alphanumeric, and byte mode.

2 *Error correction*: the codewords are then passed through the Reed-Solomon algorithm encoder, which adds redundant information to the codewords using the Galois Fields.

3 *Module placement*: all the parts (detection marker, aligment marking etc.) of the QR are placed in the output image, the last operation is the placement of the error correction codewords and data codewords, then the masking is computed and the best mask is applied to obtain the better black and white blocks disposition.

4 *Quiet zone and final image creation*: a quiet zone is added around the QR code and the final QR code image is created. In the case of Aztec and Maxicode the quiet zone is not added.

Java SE (Standard Edition) is a standalone java application that can be used for encoding and decoding of a barcode performing image processing tasks; detecting and locating barcodes in images; reading and writing barcode data.

It can be integrated in some supported framework to create an application without rewriting all the classes and functions.
The Android module provides by all the functions and classes to create an application that can read and write different barcode in Android.
The ZXing android-integration module is a module which includes the necessary classes for the developer to launch the intent and handle the results, it also includes the classes for the UI customization, like the viewfinder and the overlays. This integration module is useful for developers that want to include the barcode scanning functionality in their app do not go through the hassle of integrating the library manually.
Zxingorg provides a web application to read or generate barcode without the necessary installation of the library or the Java software.
Zxing.appspot.com is the last module that gives the possibility to an user to read a QR code using the webcam of his/her device [20].

2.2 The OpenCV library

OpenCV (Open Source Computer Vision Library) is an open source computer vision and machine learning software library. The library is written in C++ and has interfaces for several programming languages, including C++, Python, and Java. OpenCV was built to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception in the commercial products. It contains a large collection of image processing and computer vision algorithms, including image filtering, feature detection, motion analysis, object recognition, and more. Being an Apache 2 licensed product, OpenCV makes it easy for businesses to utilize and modify the code [17].
For the goal of this thesis we used the following function: *GaussianBlur*, *imwrite*, *imread*, *findHomography* and *warpPerspective*.
GaussianBlur() this method apply obfuscation effect called blur (see Chapter 4) using a Gaussian kernel to distort the details of the image. It is done with the function

```
cv.GaussianBlur(src, kernel size(x,y), sigmaX, sigmaY)
```

This method has the following parameters:

- src: the input image to be smoothed.
- Kerne size: the kernel size of the Gaussian kernel.
- Sigma X: the standard deviation of the Gaussian function in the x-direction (must be positive).

- Sigma Y: the standard deviation of the Gaussian function in the y-direction (must be positive).

Taking as input an image by convolving it with a Gaussian kernel, which is a matrix of weights that is calculated based on the standard deviation (sigma) of the Gaussian function. Increasing the value of the sigma we will obtain a increase of the blur applied to the image, if the sigma Y is not defined the the method will use the same value of Sigma X on Sigma Y. If the kernel values are set to 0 the blur will be applied to all the image [17].

The **Imread()** function loads an image from the specified path and returns it as an array. The array dimension will be height, width, number of channels, where the number of channels will be 3 for RGB images and 1 for grayscale images. If the image cannot be read (because of missing file, improper permissions, unsupported or invalid format), the function returns an empty matrix [17].

```
cv.imread(filename[, flags])
```

Where the parameters are:

- filename: Name of file to be loaded;
- flag: values that can indicate a mode such as *IMREAD_COLOR* that convert image to the 3 channel BGR color image.

Supported image file format:

- **Windows bitmaps**: *.bmp, *.dib.

- **JPEG files**: *.jpeg, *.jpg, *.jpe.
- **JPEG 2000 files** : *.jp2.
- **Portable Network Graphics**: *.png.
- **WebP**: *.webp.
- **Portable image format**: *.pbm, *.pgm, *.ppm *.pxm, *.pnm.
- **Sun rasters**: *.sr, *.ras.
- **TIFF files**: *.tiff, *.tif.
- **OpenEXR Image files**: *.exr.
- **Radiance HDR**: *.hdr, *.pic.

Imwrite() function saves the image to the specified file. The image format is chosen based on the filename extension. In general, only 8-bit single-channel or 3-channel (with 'BGR' channel order) images can be saved using this function, with these exceptions:

```
cv.imwrite(filename, img[, params])
```

Where the parameters are:

- filename: Name of the file;
- img[, params]: Images to be saved with a possible format-specific parameters encoded.

The format image supported are:

- 16-bit unsigned images can be saved in the case of PNG, JPEG 2000, and TIFF formats
- 32-bit float images can be saved in PFM, TIFF, OpenEXR, and Radiance HDR formats.
- PNG images with an alpha channel can be saved using this function.
- Multiple images can be saved in TIFF format.

If the image format is not supported, the image will be converted to 8-bit unsigned and saved that way.

FindHomography finds a perspective transformation between two planes [17]. Taking in consideration an image captured with a camera, we can note that the picture can look distorted due to the prospective angle of the person that made the shot. This effect is called perspective distortion. If we take in consideration a subject in a photo taken form a perpendicular viewpoint, the four edges of the object are A,B,C and D (top-left, top-right, bottom-left and bottom-right). A photo of the same subject will be taken from a different angle the resulting image can be see as A',B',C' and D' that are the translated point of the original image [3].

```
cv.findHomography(srcPoints, dstPoints)
```

- **srcPoints**: coordinates of the points in the original plane, it is an image represented as a matrix containing all the dimension of the source picture.

- **dstPoints**: coordinates of the points in the target plane, matrix where the result are stored.

warpPerspective is used to apply a perspective transformation to an image [17]. The function warpPerspective transforms the source image using the specified matrix:

```
cv.warpPerspective( src, M, output size))
```

The function takes three parameters:

- The first parameter is the source image, which is an array representing the image to be transformed.
- The second parameter is the transformation matrix, which is a 3x3 floating-point array representing the perspective transformation to be applied.
- The third parameter is the size of the output image, which is a tuple of (width, height).

The transformation matrix is typically computed using the findHomography() function, which takes in a set of source and destination points and computes the perspective transformation matrix that maps the source points to the destination points.
The function returns an array that represents the output image after the perspective transformation has been applied. The size of the output image is specified by the third parameter.

Conclusion In this Chapter we have introduced libraries used in this thesis. We described which barcode supports the ZXing library and also illustrated the components of the core module. In the last section we discussed the function used from the OpenCV library.

Chapter 3

Digital Certifications

In this chapter we are going to explain the architectures used by the QR codes used in the experiments. In the first section, the architecture of the Digital Covid Certificate (DCC) will be explained. In the second section, on the other hand, the Digital Covid Certificate proposed in [2, 6].

3.1 Standard Certifications

The European Union eHealth Network, created guidelines and technical specifications for proof of vaccination for medical purposes. These guidelines prioritize simplicity, compatibility with existing national standards and strong protection of personal data, with the aim of promoting interoperability between EU Member States initiatives. The guidelines include a minimum dataset and a globally unique, verifiable Unique Vaccination Certificate/Assertion Identifier [8].

3.1.1 CBOR

CBOR (Concise Binary Object Representation) is a data format that aims to have compact code size, efficient message size and the ability to extend without requiring version negotiation. The data model of the format is based on the JSON format and its based on the following idea [15]. The encoding must have the capability to express most frequently used data formats in Internet standards in a clear and concise manner. The encoder or decoder code should be efficient to support systems that have limited memory, processing power, and instruction sets. The priority for the encoder and decoder should be compact code size, rather than compact data size, in the serialization process. The Decoding of data must be possible without the need for a schema description. The format should be appropriate for systems with limited resources as well as high-volume applications. Additionally, it should support the conversion of all JSON data types to and from JSON, and allow for expansion while still enabling earlier decoders to decode the extended data..

CBOR as the following data item:

- an integer in the inclusive range between -264 . . . 264-1;
- a sequence of zero or more bytes ("byte string");
- a sequence of zero or more Unicode code points ("text string");
- a sequence of zero or more data items ("array");
- a mapping from zero or more data items ("keys") each to a data item ("values"), ("map");

- a tagged data item (”tag”), comprising a tag number (an integer in the range between 0 and $2^{64} - 1$) and the tag content (a data item).

The CBOR encoding process involves converting a CBOR data item into a byte string that contains a well-formed encoded data item. The decoding must only return the data item if the input is a well-formed encoded CBOR data item. The first byte of each encoded data item includes information about the major type (represented by the 3 most significant bits) and additional information (represented by the 5 least significant bits). Taking in consideration the additional information's value describe how to load the argument value [5]:

- Less than 24: the argument value is encoded immediately after the additional information;
- 24, 25, 26, or 27: the data are encoded in 1, 2, 4, or 8 bytes after the first byte

The following list explain with an example the major type and the interaction that they have with other bytes associated with the type:

- **Major type 0**: the encoding of an unsigned integer within the range of 0 to $(2^{64} - 1)$ is included in CBOR representation. The encoded item holds the argument value itself;
- **Major type 1**: a negative integer ranging from -2^{64} to -1 is represented. The item's value is obtained by subtracting the argument from -1;

- **Major type 2**: a byte string is represented in CBOR. The length of the string is specified by the argument;
- **Major type 3**: a text string represented in UTF-8 format. The length of the string is specified by the argument;
- **Major type 4**: an array consisting of data items of various types, with the length of the array specified by the number of elements. The additional information bits in the initial byte denote the length. The rules for encoding arrays follow those for the major type 2;
- **Major type 5**: a map is comprised of pairs of data items, where each pair consists of a key followed by a value. The argument specifies the number of pairs in the map. As the items in a map come in pairs, their total number must always be even. A map that contains an odd number of items is considered not well-formed.

CBOR has a web token called CWT (CBOR Web Token) that provides an efficient way to convey claims between two parties. The claims in a CWT are encoded using the Concise Binary Object Representation (CBOR), and enhanced security is provided through the use of CBOR Object Signing and Encryption (COSE). Claims are pieces of information made about a subject, represented as a name-value pair consisting of a claim name and claim value. CWT is based on JSON Web Token (JWT) but uses CBOR instead of JSON [15].

3.1.2 COSE

The structure of COSE (CBOR Object Signing and Encryption) objects is created to allow for maximum code reuse while parsing and processing various types of security messages. All message structures are constructed using the CBOR array format [22]. The first three elements in the array are always uniform, consisting of:

- A bstr that encapsulates the protected header parameters;
- An unprotected header parameters map;
- The message content, either plaintext or ciphertext, wrapped in a bstr. When detached, the location is still used, but the content is represented as a nil value.

The elements beyond this point are determined by the specific message type. The design of COSE messages also employs the concept of layers to separate distinct cryptographic concepts.
COSE has two distinct signature structures, COSE_Sign and COSE_Sign1. COSE_Sign enables multiple signatures to be added to the same content, while COSE_Sign1 is limited to a single signer. Conversion between these two structures is not possible as the signature computation includes information identifying the structure used, leading to a failure in signature validation if converted. The DCC used the COSE_Sign1 to sign the data.

In this schema (see Figure 16), the signature is computed and verified by combining the contents of the protected header and the payload. The signature is then stored in the CBOR array as a byte string of major type 2.

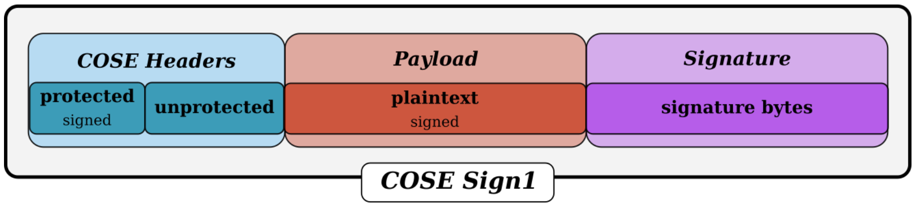


Figure 16: Sign1 schema [7]

To specify the algorithm used, a mandatory parameter named ”alg” with label 1 must be added to the protected header. This parameter restricts the algorithms that can be used to verify the signature that is the Elliptic Curve Digital Signature Algorithm (ECDSA) [22].
The ECDSA is a cryptographically secure digital signature scheme, based on the elliptic-curve cryptography [23]. The ECDSA keys are:

- private key: An integer of 256 bits in size (32 bytes) that is kept secret and known only to the person who generated it;
- public key: is generated from the private key and it is used to verify the signature

The signature is computed by following this steps:

- Compute the hash of the message using a cryptography hash function, such as SHA-256, to obtain the value h: h = hash(msg);
- Generate a random number k, in the range of [1 to n-1], securely. In the case of deterministic-ECDSA, the value of k is obtained by using HMAC derivation from the combination of h and privateKey;
- Calculate the x-coordinate, ”r”, of the random point ”R” generated by

multiplying a randomly generated number ”k” by the base point ”G”: r = R.x;

- Compute the signature proof: $s = k^{-1} * (\mathrm{h} + \mathrm{r} * \mathrm{privateKey})(\mathrm{mod\ n})$;
- Return the signature r, s.

The COSE specifications emphasize the use of a deterministic implementation of DSA to prevent collisions caused by biased random number generation, which has been a previous attack on the algorithm [19].

3.1.3 Base45

QR codes have limited capacity for storing binary data, which must be encoded as characters using a defined mode in the QR code standard. The simplest mode, Alphanumeric mode, uses 45 characters making Base32 or Base64 encoding less effective.
In QR codes, the Alphanumeric mode uses a 45-character subset of US-ASCII. This mode converts 2 bytes of data into 3 characters using the Base45 encoding scheme. This is different from Base64 encoding, which converts 3 bytes of data into 4 characters.
For encoding, two bytes $[a, b]$ are treated as a number n in base 256, represented as an unsigned 16-bit integer, resulting in $n = (a*256)+b$, the number n is transformed into base 45 $[c, d, e]$ such that $n = c + (d * 45) + (e * 45^2)$, with the order of c, d, and e arranged in a way that the leftmost $[c]$ holds the least significance. The values of c, d, and e are then used to find the corresponding characters in Table 4, forming a string of three characters. The reverse process is performed during decoding.

Value	**Encoding**	**Value**	**Encoding**
00	0	23	N
01	1	24	O
02	2	25	P
03	3	26	Q
04	4	27	R
05	5	28	S
06	6	29	T
07	7	30	U
08	8	31	V
09	9	32	W
10	A	33	X
11	B	34	Y
12	C	35	Z
13	D	36	Space
14	E	37	$
15	F	38	%
17	H	39	*
18	I	40	+
19	J	41	-
20	K	42	.
21	L	43	/
22	M	44	:

Table 4: Base 45 alphabet

To encode a byte string $[abcd \ldots xyz]$ of arbitrary content and length, the pairs of bytes are processed from left to right using the method described above. If the number of bytes is even, the resulting encoded string will have a length that is divisible by 3. If the number of bytes is odd, the last byte will be encoded using two characters. To decode a Base45 encoded string, the inverse operations are performed. If we take as example the string "AB", its byte sequence is [[65 66]]. If we look at all 16 bits, we get $65 * 256 + 66 = 16706$. 16706 is equals to $11 + (11 * 45) + (8 * 45 * 45)$, so the sequence in base45 is [11 11 8]. Taking the encoded value form the Table 4, we get the encoded string "BB8".

The Base45 encoding is recommended for storing binary data in a QR code, using 11 bits for 2 characters, with an ECI mode indicator of 0010. If the data is intended for another form of transport the other encoding like the Base64 have better performance on the data [18].

From the point of view of the security, the Base45, as all the encoding, can be attacked with a buffer overflow, that implies that the decoder must handle any input, including handling octet values from 0 to 255 and the NUL character.

The Base45 differs from other encoding like Base64, it avoids the padding and because of this it is always needed to take care when encoding an odd number of octets and when decoding a number of characters that is not divisible by 3.

Another possible attack is the covert channel attack ("type of attack that creates a capability to transfer information objects between processes that are not supposed to be allowed to communicate") where an attacker can corrupt the data inserting non alphabetic characters in order to exploit implementation error. To prevent this type of attack the implementation of the decoder must always reject non valid characters [18].

3.1.4 ZLIB

Zlib is a compression library free written by Jean-loup Gailly (compression) and Mark Adler (decompression). The compression method used by zlib, deflation, outputs compressed data in the form of blocks. One of the block types is ”stored blocks”, which consist of raw input data and a few header bytes. If other block types cause the data to expand, deflation defaults to using stored (uncompressed) blocks. With the default settings used by *deflateInit()*, *compress()*, and *compress2()*, the overhead is five bytes per 16 KB block (around 0.03%) plus an initial overhead of six bytes for the entire stream. This overhead applies even if the last or only block is smaller than 16 KB and it results in an overhead of 1100% for a single-byte input stream (eleven bytes of overhead, one byte of actual data). The overhead approaches the limiting value of 0.03% for larger stream sizes [21].

3.1.5 DCC payload and QR code architecture

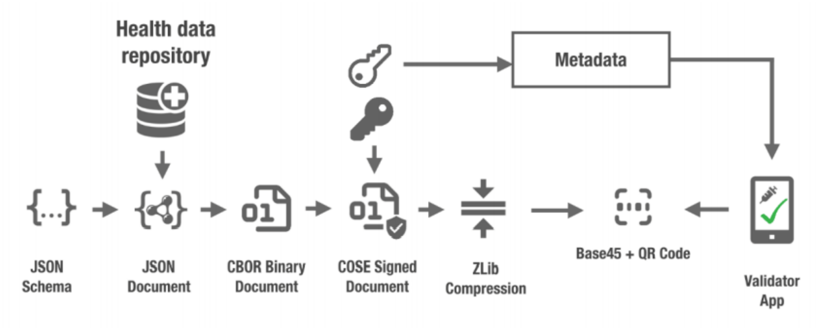


Figure 17: DCC creation process [11]

The DCC QR core is abtained by the computation of the process as depicted in Figure 17, the first step is to store the data about the user on a JSON following the data structure provided by the eHealt Networks guideline for the DCC-19 an example of a correct structure is showed below [10].

```
{
"ver": "1.2.1",
"nam": {
    "fn": "Musterfrau-G\u00f6\u00dfinger",
    "gn": "Gabriele",
    "fnt": "MUSTERFRAU<GOESSINGER",
    "gnt": "GABRIELE"
},
"dob": "1998-02-26",
"v": [
    {
        "tg": "840539006",
        "vp": "1119349007",
            "mp": "EU\/1\/20\/1528",
            "ma": "ORG-100030215",
            "dn": 1,
            "sd": 2,
            "dt": "2021-02-18",
            "co": "AT",
            "is": "Ministry␣of␣Health,␣Austria",
            "ci": "URN:UVCI:01:AT:10807843
   F94AEE0EE5093FBC254BD813#B"
        }
    ]
}
```

The next step in the process is to store the JSON payload as a CBOR (Section 3.1.1) (binary document) and build it into a COSE_Sign1 (Section 3.1.2). The appropriate header is added to the CBOR and the entire content is signed using the private key of the DSC and the selected encryption algorithm.

After the binarization of the JSON the data are compressed with the ZLIB library (Section 3.1.4). The compressed string is encoded in base45 (Section 3.1.3) and the last step consists in adding the prefix “HCx:” to state the Health Certificate version number.

In the end we will obtain a QR code valid (see Figure 18) with an error correction level Q (25%) that certificate the state of person that can be: vaccinated, infected or cured.

Figure 18: Standard DCC

3.2 Optimazed Certification

The optimized DCC is a proposal of improvement DCC provided by Giacomo Arrigo [2] and Marco Carfizzi [6] in their master thesis, the experimental QR is cross-language interface description language (IDL) to generate customizable data structures that used the buffer protocol standard for the security of the data form the different attacks.

3.2.1 Buffer Protocol

Protocol buffers are a serialization format for packets of structured data with types, which can range from a few megabytes in size. This format is appropriate for both short-term network communication and long-term data preservation. The addition of new information to protocol buffers can be

done without altering existing data or requiring software upgrades.
To define the message contained in the protocol buffer the user just need to provide the structure in the ".proto" file which will be executed by the compiler. An example of the structure can be see in the following code:

```
message Person {
  optional string name = 1;
  optional int32 id = 2;
  optional string email = 3;
}
```

From the example we can immediately notice that in the message before of each type there is a field specification *optional*, indeed the protocol buffer field can be:

- singular: this is the standard field regulation when no alternate field regulations have been designated for a specific field;
- optional: the same as a singular field, with the addition of the ability to determine if a value has been explicitly set. An optional field can either be set or unset;
- repeated: the repeated field type can appear zero or more times in a valid message and the order of the values will be maintained.
- map: this is a field type consisting of a key and a value pairing.

The encoding of a message transforms each key-value pair into a record composed of the field number, a wire type, and a payload. The wire type informs the parser of the size of the payload following it, enabling older parsers to bypass fields they are not familiar with.

The protocol buffer has six different wire types VARINT, I64, LEN, SGROUP, EGROUP and I32; the proposed DCC utilize the following two type:

- VARINT: that is identified by the number ”0” and it is used for the: int32, int64, uint32, uint64, sint32, sint64, bool and enum.
- LEN: defined by the number ”2” for the: string, bytes, embedded messages and packed repeated fields

The record's ”tag” is represented as a varint produced from the field number and wire type through the equation **(field_number ≪ 3) | wire_type**. In simpler terms, after decoding the varint related to a field, the bottom 3 bits give us the wire type and the rest of the integer number provides the field number.

3.2.2 DCC experimental architecture

The new data format uses location-based parsing, where the parser reads bytes sequentially and determines the field being read based on the order defined in an array in the IDL schema. This means that no key is saved in the data and every position is linked to a key-value pair specified in an external file, where each array element represents a field of the message and includes its metadata.

```
{"Test": [
      "bytes": 1,
      "id": "algorithm",
      "type": "u_int"
      ]
      "case":{
            "1":{
                data about the vacination
            }
            "2":{
                data about the user
            }
            .
            .
            .
            "10":{
                data about the provider of the vacination
            }
      }
}
```

With this new format it is facilitated the verification of the information for the reader of the QR code. The payload of the QR contains the JSON encoded using the base45 and the prefix verification of the DCC encoded before the JSON. Since its dimension is the shortest of the other QR code, only 183 byte in comparison with the 517 byte of the DCC, the experimental DCC does not need the compression given by the ZLIB.
From the point of view of the security, the experimental DCC is based on the Protocol Buffer that provide an efficient protection from the possible attacks to the payload of the QR.
The QR code structure consists in a QR code with error correction level ”L”, the version of the QR is ”6” providing an image of dimension 41x41 with low density as depicted in Figure 19 [2].

Figure 19: Experimental DCC

Conclusion

In this chapter we explained the architecture structure of the EU Digital COVID Certification and the experimental one proposed in [2]. The standard DCC architecture use CBOR to store the JSON file and the base45 to encode the data, but due to the dimension of the encoding data the standard DCC need to be compressed using ZLIB. The experimental DCC use a different structure obtaining an encoding of the data lower volume compered to the standard DCC. This new architecture does not need a compression of the encoded data and the resulting QR code has a modules density lower then the standard one.

Chapter 4

Test structure

This section will explain the structure of the tests. Different types of image distortion were applied to each QR test. The two testing cases being the moiré effect to reproduce a QR decoding from a screen and the second without the moiré effect to replicate a QR decoding from a piece of paper.
For each test the images are divided in the following directory structure:

- *Moiré/no Moiré*: directories containing the two principal test
  - *Noise levels*: directories containing the images with the three noise levels applied
    - *Distance range*: directories divided by the three distance applied
      - *Blur level from 0 to n*: directories the images with the previews effect and the blur effect

The population of each test is 1000 elements representing the different perspective distortion.

The distortion applied on each QR are:

- *Distance*: this parameter is used to define the distance from which the QR is seen that can be 0 (close), medium distance and long distance. Through this parameter we can simulate the attempt to read a QR code from different distances;

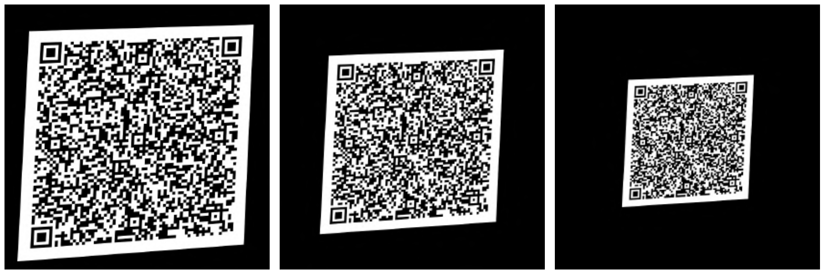


Figure 20: Example of QR with distance 0, 50 and 100

- *Perspective distortion*: refers to a change in the perceived size and shape of objects in an image as a result of their position relative to the camera. This happens when the camera is not perpendicular to the plane of the image and results in the objects in the foreground appearing larger than those in the background. Each test have a different perspective distortion to emulate the QR code read from various directions;

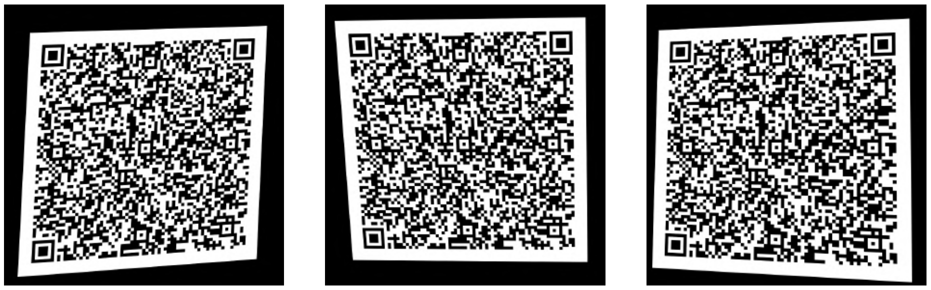


Figure 21: Example of QR difference perspective distortion applied

- *Moiré*: is an optical phenomenon that occurs when an image is captured or displayed with a pattern of repeating elements that are too close together. The Moiré effect can cause a number of visual artifacts, such as distorted lines, false colors, and other visual distortions. It can also make it difficult for a QR code reader to correctly read the code. The common to have a moiré distortion when taking a picture of the screen;

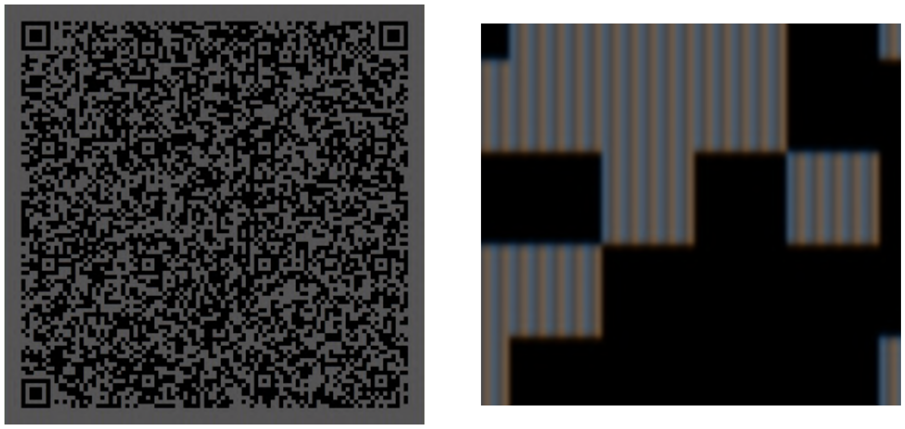


Figure 22: Moiré on QR

- *Noise*: in the context of images, thermal noise can appear as random speckles or graininess in the image, and it can be more pronounced

in low-light conditions. It can also affect the image quality, making it appear less sharp and more difficult to read. Thermal noise can also make it difficult for a QR code reader to correctly read the code.
The thermal noise value can have three levels that were chosen based on tests done over time on QRs, trying to reproduce effects that happen more easily;

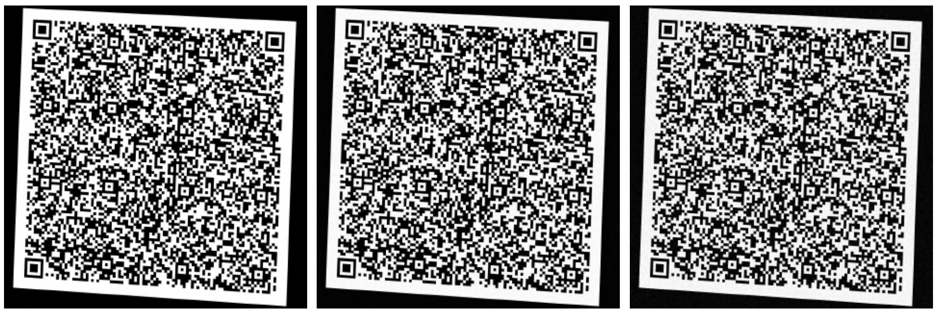


Figure 23: Example of QR with noise 0, 10 and 20

- *Blur*: it is a visual distortion that occurs when an image appears out of focus, or when its details are not sharp. This can happen for a variety of reasons, such as an image captured with a camera that is not properly on focus. Blur effect is applied from 0 (absent) to n, where the intensity of the blur is higher, as for noise the values were chosen based on tests over time.

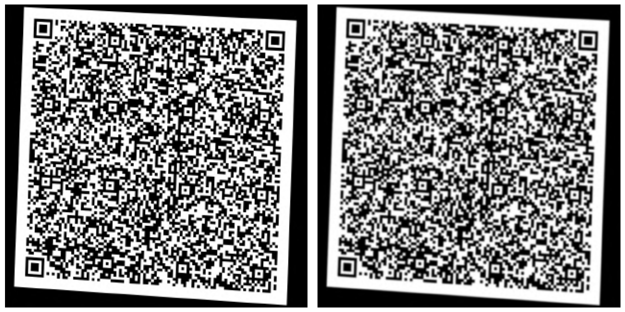


Figure 24: Example of QR with blur 0 and 2

To implement the error correction tests, the ZXing library has been modified. The modified class is *DecoderResult()* from *com.google.zxing.common*, which serves to encapsulate the result of decoding a matrix of bits. A new parameter was added in the class signature representing the byte value of the codowords before the Reed-Solomon algorithm is applied to compare how many bytes have been corrected, so that we know whether error correction has been applied.

```
DecoderResult ( byte [] rawBytes , byte [] notCorrectedBytes
   , String text , List < byte [] > byteSegments , String
   ecLevel ,int saSequence , int saParity , int
   symbologyModifier )
```

The *DecoderResult* object occurs as a result of the *Decoder* method from the *com.google.zxing.common* package, where a copy of the parsed the raw byte array and the corrected codewords are given in input.

```
private DecoderResult decode ( BitMatrixParser parser , Map <
   DecodeHintType ,? > hints )
```

it has been created a JAVA file containing three functions for running all the QRs tests:

- *Main*: function that calls the *getFiles* function to take all the QRs to be tested and saves the result of *test_single_QR* method on each image. The results data are stored in one of the six possible results variables: correctly read, not correctly read, correctly read with the use of the error correction, notFound, notDecoded and error correction fail. For the variables notFound, notDecoded and error correction fail, catching the exceptions in the Java file. Once all the QRs have been tested, it saves the results to the appropriate files. Figure 25 shows the diagram of the java file;

- *getFiles*: function that takes a directory in input and gives in output all the files having as extension *.jpg* or *.png*;

- *test_single_QR*: function that performs a *tryHard* decode on the input image and returns the result using the *DecoderResult* method.

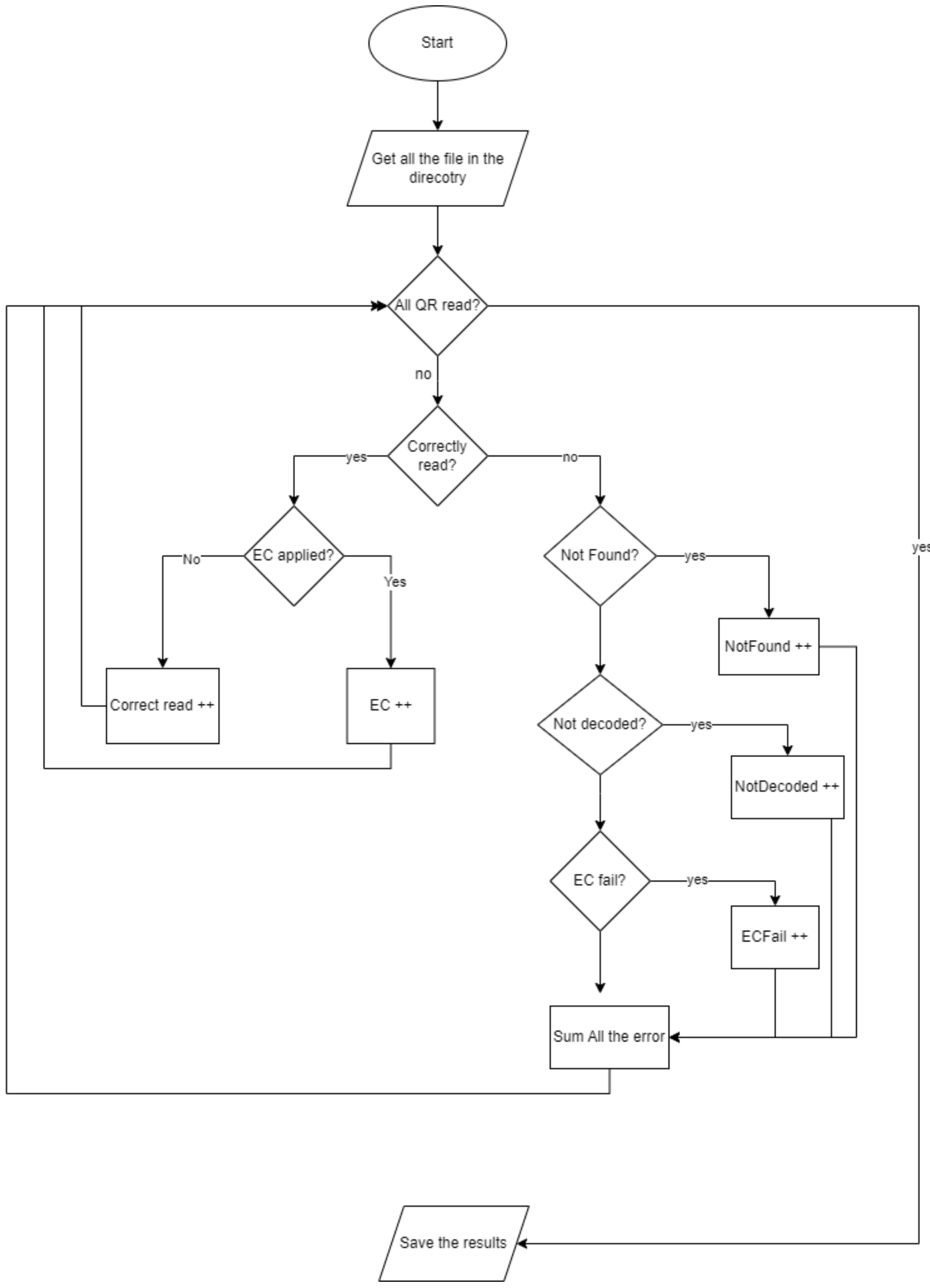


Figure 25: Java code's diagram

Conclusion

In this chapter, the structure of the tests was explained in general. In the first part, the effects applied to images were explained specifying the reason behind the values chosen. The second part explained the structure of tests done with JAVA, describing the modification applied to the library and how the data were extracted.

Chapter 5

Case Studies

In this chapter we are going to analyze the data extracted from the tests by explaining the results obtained from the graphs. In the first section we will analyse the data from the study between two QRs with the same data but different error correction level. In the second section instead we will compere the performances of the standard DCC with the experimental DCC.

5.1 Methodology

The first part of the experiment concerns the utility of error correction in barcodes having a large density of data, for these types of tests the Green Pass was considered because of its immense use during the COVID19 pandemic and the problems encountered by users when verifying the certificate.
The seconds part of the experiment, on the other hand, consists in comparing the performance of the DCC proposed by the EHB and the experimental green pass proposed by Arrigo and Carfizzi.

With each test, the results should be viewed as the probability of reading a QR. A probability tending toward 100 will indicate that the QR is more clear to read compared to one with a probability tending toward 0.
In all the data study it is taken into consideration the data from the medium level noise, because it represents the average value between the low level and high level of noise. The first data representation used in this thesis is a line chart where each line indicates the percentage of decoding of each population of QRs(Y-axis) over different levels of blur (X-axis). Each line represents different levels of scaling applied to the QRs, b indicate the zero and f indicate the farthest. The second chart is a stacked area chart that represents the number of QRs (Y-axis) over different levels of blur (X-axis).

5.2 DCC - Low High error correction

The DCC Low - High data analysis concerns QRs with low level of error correction (maximum 7% of recovery) and high level of error correction (maximum 25% of recovery). Both QRs are analyzed with and without the distortion created by the Moiré as depicted in Figures 26 and 27 (DCC low on the left and DCC High on the right) and applying all the distortion described in chapter 4 .

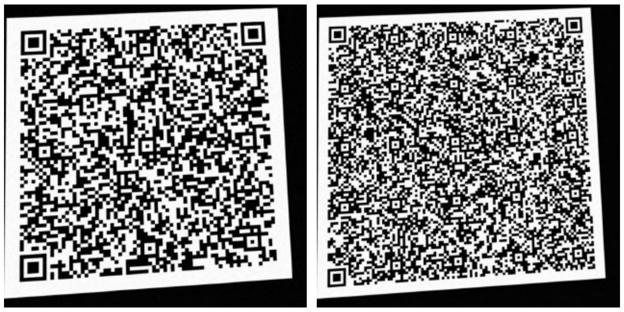


Figure 26: Example of QR without the moiré applied

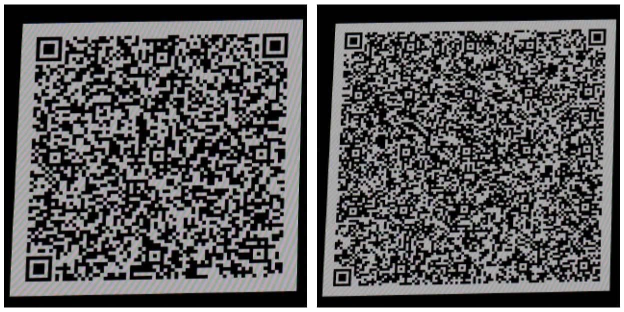


Figure 27: Example of QR with the moiré applied

5.2.1 Analysis DCC Low - High without Moiré

From the first experimental tests, we can see that as the distance varies and the blur increases, the QRs decoding rate decreases. This occurs because as the distance and blur value increases, the QR will become more and more difficult to read because of the library's difficulty in detecting the QR due to the distance and the increased noise produced by the blur. Comparing the DCC low with the DCC high both with a medium level of noise (see Figure 28), we can easily see that the QR with lower error correction has a higher readability than the QR with higher error correction.

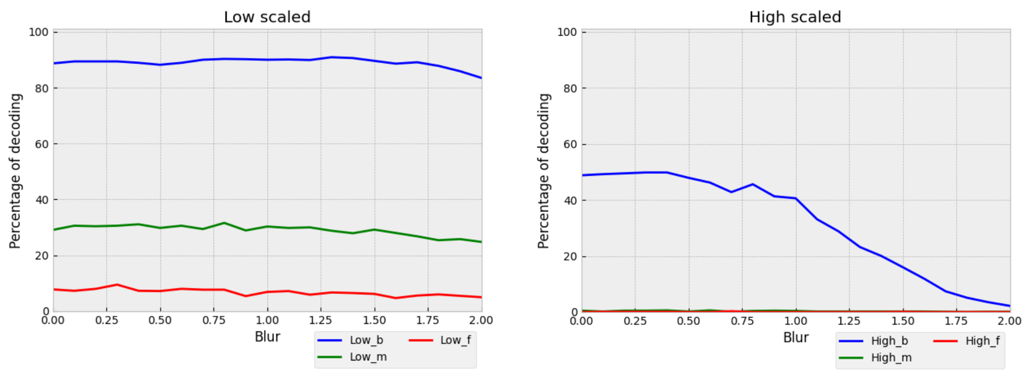


Figure 28: QR Low and High without moiré

By comparing each distance graph we understand why there is this gap. Beginning by analyzing the graphs in Figure 29 we can see that for zero distance, the DCC with low level has a success rate of about 90% and as the blur level increases the reading still remains optimal, on the other hand, for the DCC with High level the performance initially is about 50% but as the blur level increases, the reading decreases to almost 0%. An important finding that can be seen from the High level is the use of error correction, which as the blur of the image increases is applied but fails due to the excessive amount of error above the threshold guaranteed by the High level. Taking into consideration the graphs present in Figures 30 and 31 we can see that as the distance increases the Low level begins to have difficulties in correctly decoding the QR and to compensate for the amount of disruption caused by increasing the distance between the QR and the camera and increasing the blur, it uses error correction but it is never applied due to the amount of errors. Assuming that the High level has a higher threshold than the Low level, it is expected that as the disruption increases the performance will improve. Contrary to predictions, the high level of error correction does not improve decoding, reducing the percentage of success to almost 0%. Those fail on average are for the 40% in both cases (see Figures 30 and 31) due to the error correction. This is because the Reed-Solomon code detected the errors but they exceed the error threshold. From the comparison of the QRs we obtain that the percentage of success without moiré is bounded to the density of the QRs. This is because as error correction increases, there is a greater range of correction at the expense of QR density.

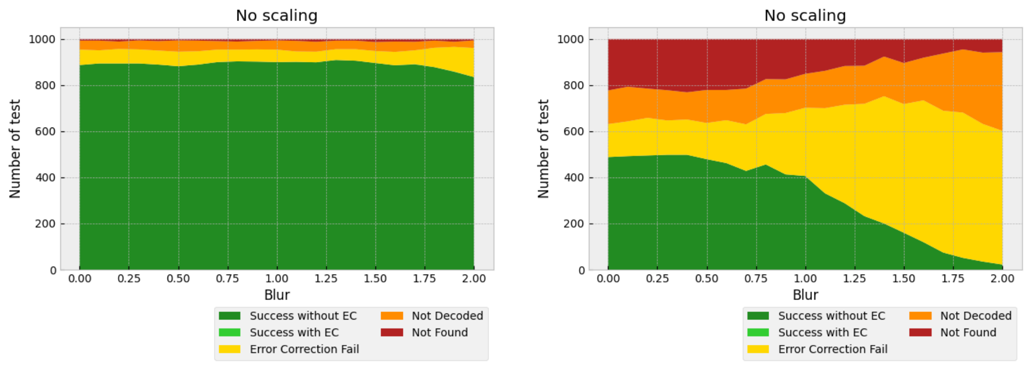


Figure 29: DCC Low-High zero distance charts

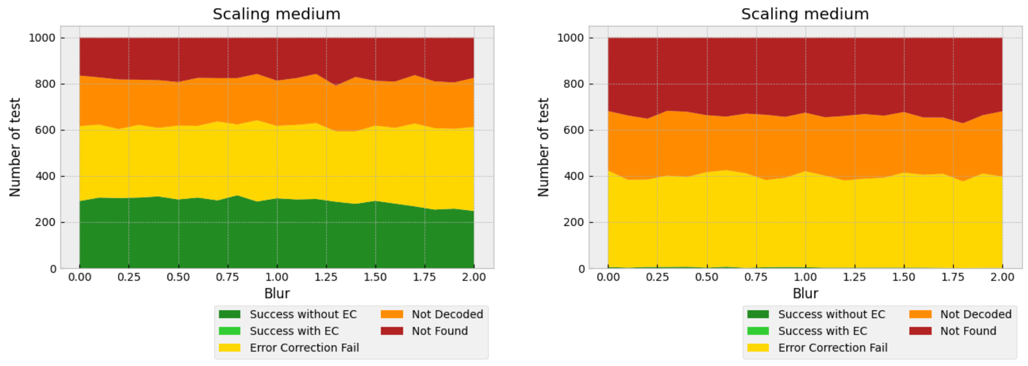


Figure 30: DCC Low-High medium distance charts

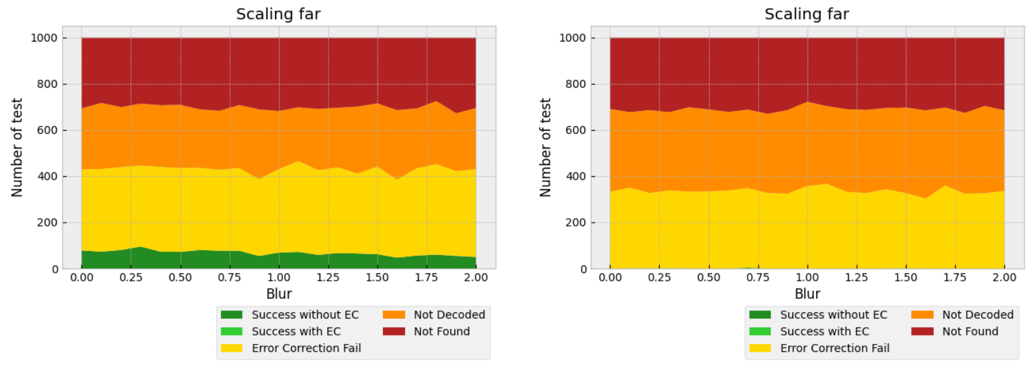


Figure 31: DCC Low-High far distance charts

5.2.2 Analysis DCC Low - High with Moiré

After completing the analysis of the two DCCs without Moiré, we analyzed the same data structures with the moiré effect applied. Like the previews analysis, starting from the charts representing the decoding success rates with average noise level. We notice (see Figure 32) that the performances are similar to those without the moiré effect for the QR with error correction Low. Instead, for error correction High the performance begin to decrease earlier (see Figure 32) than the other version previously analyzed. Since they have similar graphs, we expect to have analogous decoding success and fail rates between the two studies.

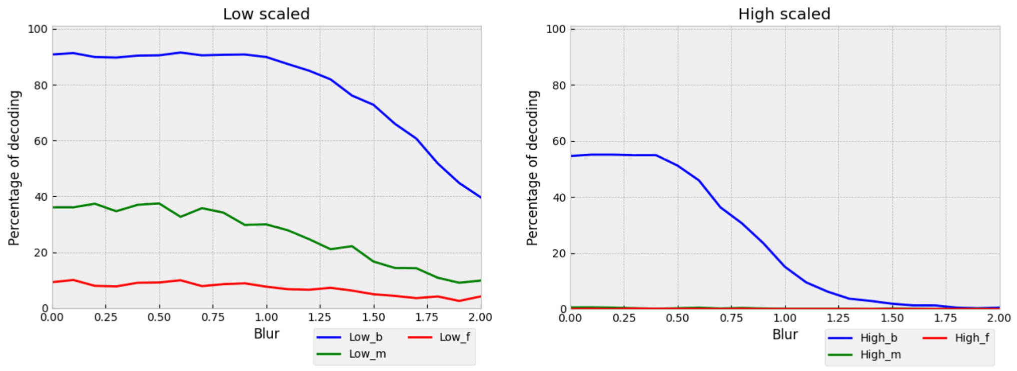


Figure 32: QR Low and High with moiré

Contrary to our assumptions by looking at the charts of the three distance, we notice that with the presence of the moiré distortion, the decoding success rate is prevalently dependent on the application of the error correction (see Figure 33). This applies to both DCC. the application of error correction is due to the distortion created by the moiré effect, which from the lowest blur value makes the image more difficult to decode, denoting that without the presence of error correction would make the QRs unreadable. Another important finding that is noted, is that as the blur increases, the use of error correction decreases. This happen for the correlation between moiré and blur, at the increase of the blur on the image, the moiré distortion decrease. From this analysis, we realized that the use of error correction is important when decoding is done in the presence of distortions, such as moiré, that lead an increment of noise in the image. While being useful, from the charts in Figures 33, 34 and 35, the decoding success rate is low and this is due to the payload architecture used in the DCC. In the next section we will compare two DCCs with different architectures to see whether with lower density error correction remains useful for successful decoding or not.

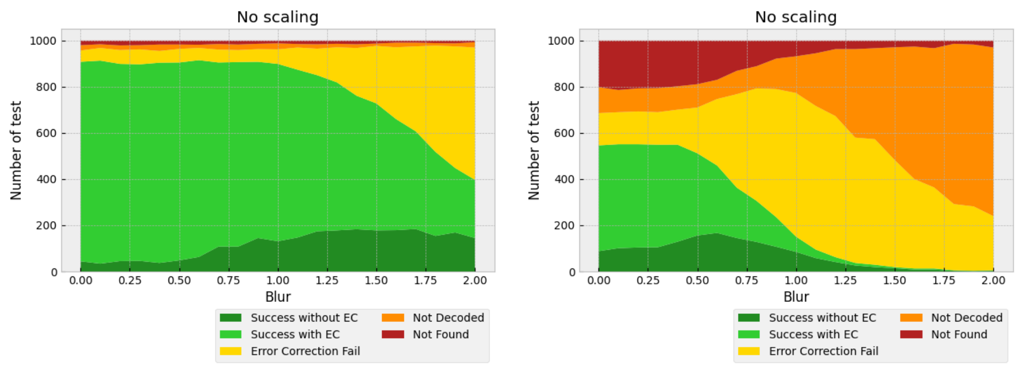


Figure 33: DCC Low-High zero distance charts

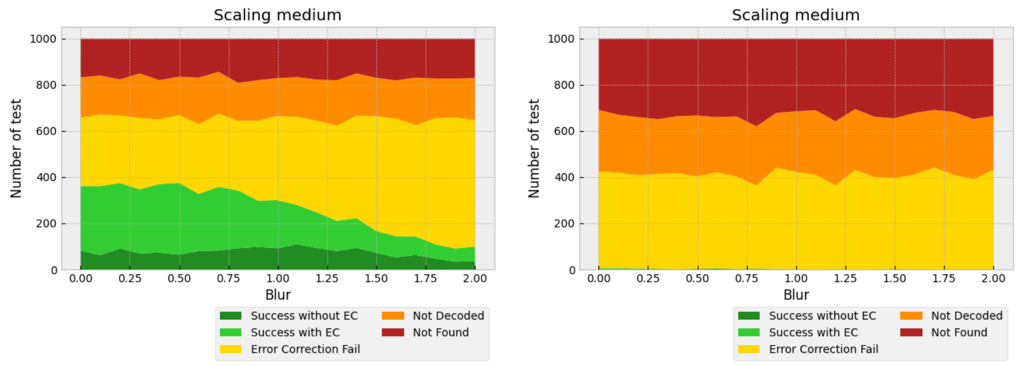


Figure 34: DCC Low-High medium distance charts

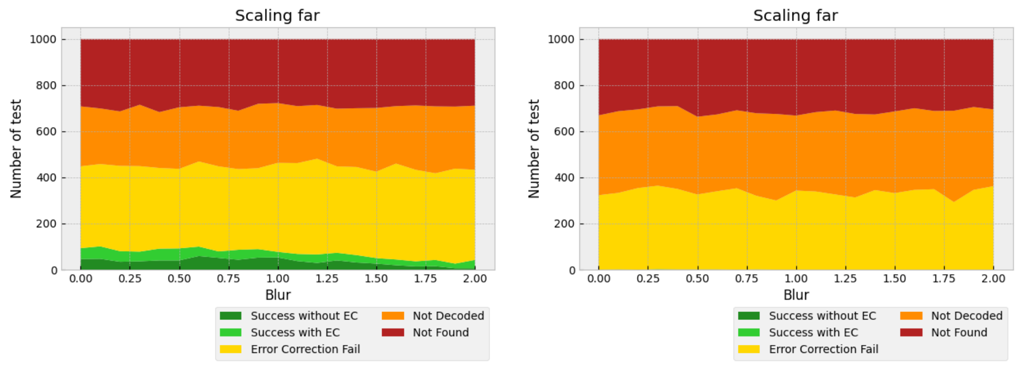


Figure 35: DCC Low-High far distance charts

5.3 DCC and Optimized DCC analysis

In this case study we are going to compare the performance of the DCC used during the pandemic and the experimental DCC proposed by Giacomo Arrigo. Applying the same tests carried out in the previous section (see Figures 36 and 37). Medium and high noise levels will be compared since, unlike the basic QR with low and high, the performance of the experimental DCC remains high even in the worst case. The low noise level was not be showed since the QR optimized in the tests has a success rate of 100% without moiré and 90% with moiré.

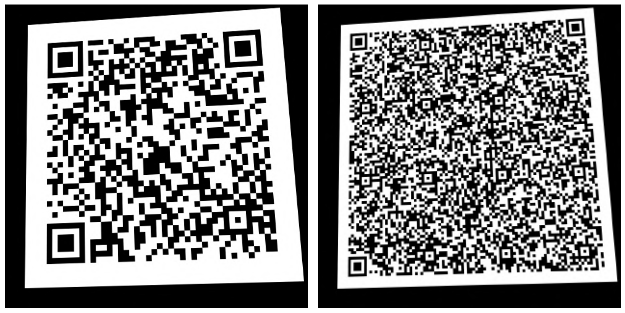


Figure 36: Example of DCC optimezed and base without the moiré applied

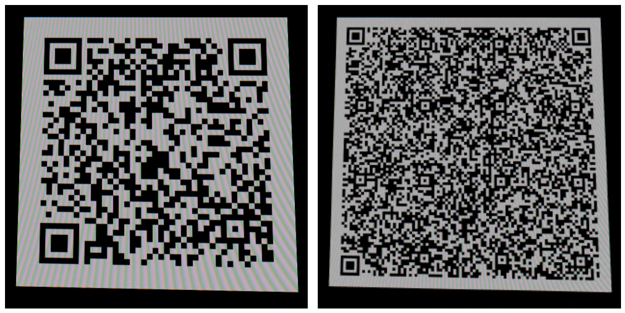


Figure 37: Example of DCC optimezed and base with the moiré applied

5.3.1 Analysis DCC Base - Optimized wihtout Moiré

From the graphs of decoding rates, it is immediately noticeable that the optimized DCC has significantly better success rates than the original DCC on both medium and high noise level (see Figures 38 and 39). These better performances for the medium noise level are not due to error correction since it is applied on average 1% of the times (see Figures 40, 41 and 40). Instead, in the case with high noise level the error correction is applied. The decode reach the 100% success rate on base distance with an application of the 18% (see Figure 43), 90% success rate on the medium distance using the error correction on average 12% of the times (see Figure 44) and a 70% decode rate for the high distance, with the majority of utilization of the error correction where the level of blur is higher (see Figure 45). From this comparison, it is immediately clear that by using an improved architecture the decoding performance of the DCC improves even in situations where, as in the case of maximum distance, the noise level is so high that normal DCC cannot be read. Furthermore, in situations where the noise level is high, the optimized DCC takes advantages of error correction to maintain decoding rates almost similar to those present in the analysis with a lower noise level. In conclusion, we can say that the experimental DCC has a very good performance due to its low density and the use of error correction.

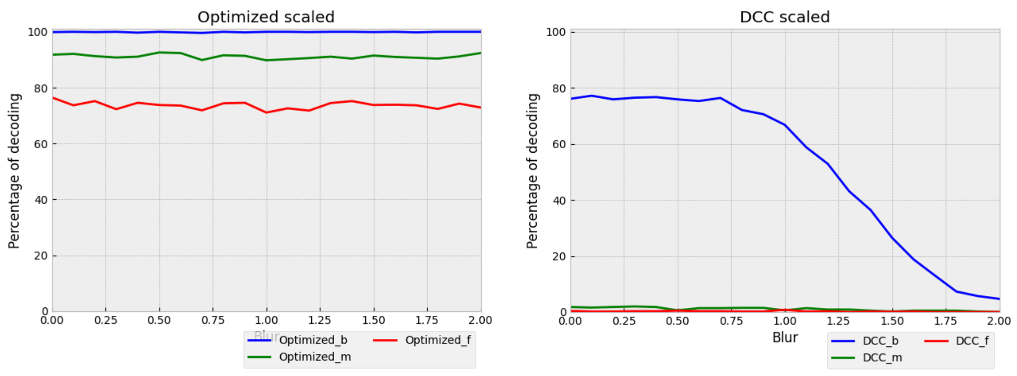


Figure 38: DCC optimized and base DCC with medium noise level

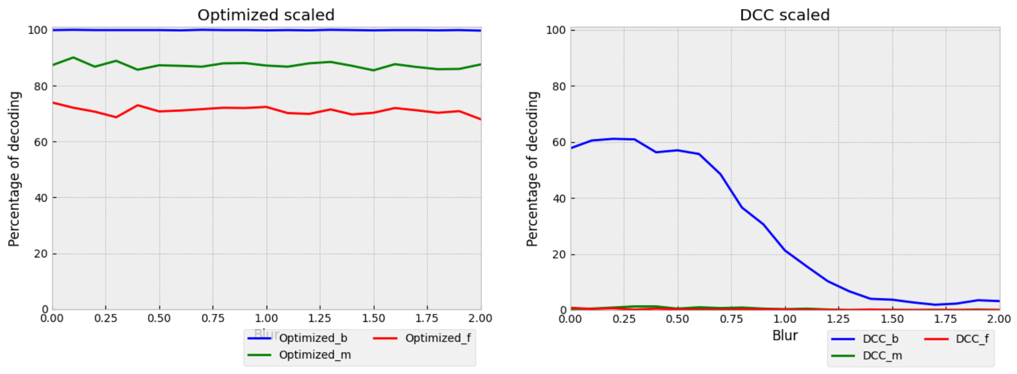


Figure 39: DCC optimized and base DCC with high noise level

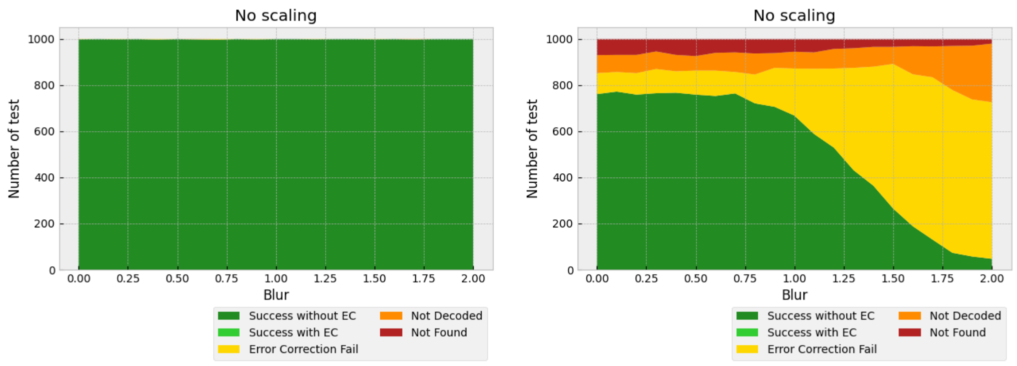


Figure 40: Optimized and base DCC with medium noise level charts

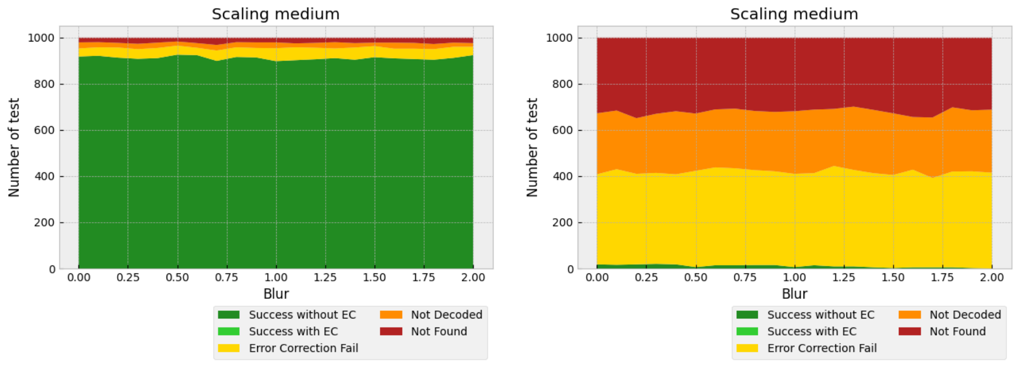


Figure 41: Optimized and base DCC with medium noise level charts

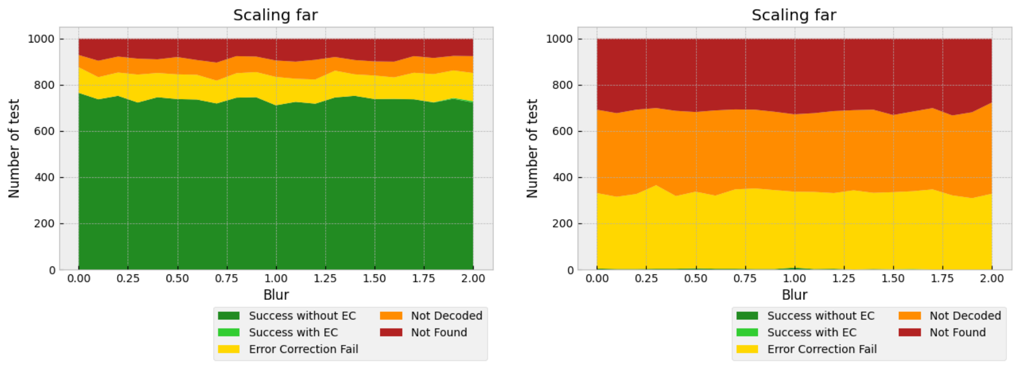


Figure 42: Optimized and base DCC with medium noise level charts

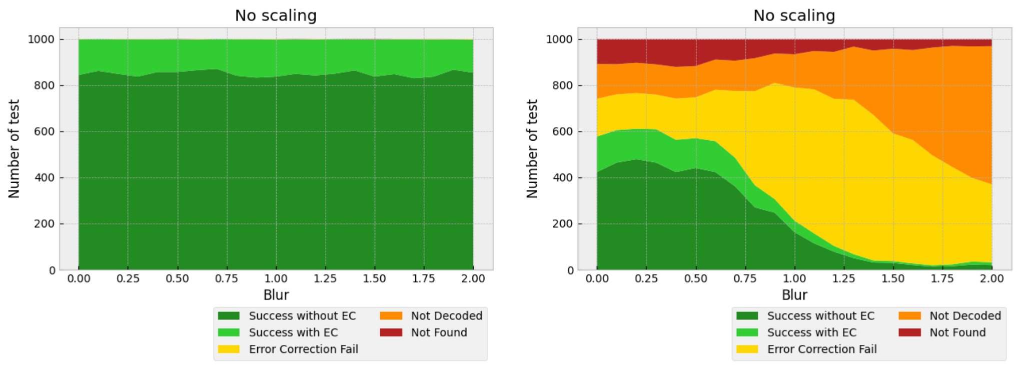
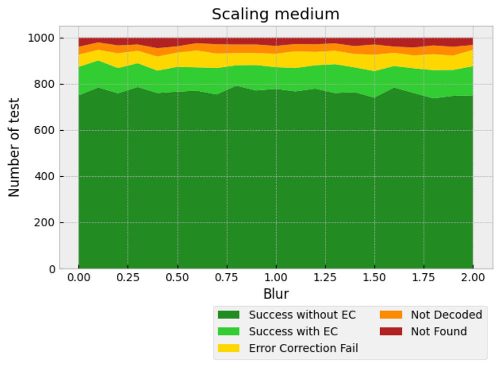


Figure 43: Optimized and base DCC with high noise level charts



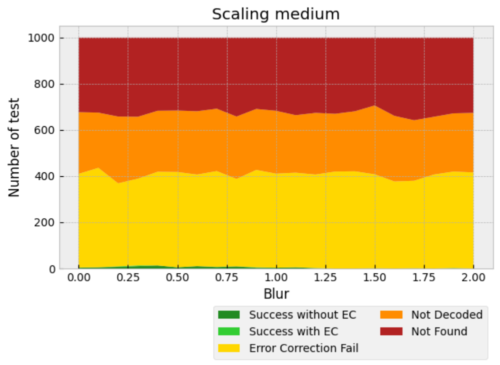
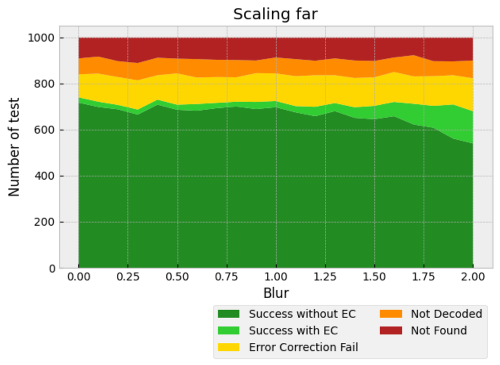


Figure 44: Optimized and base DCC with high noise level charts



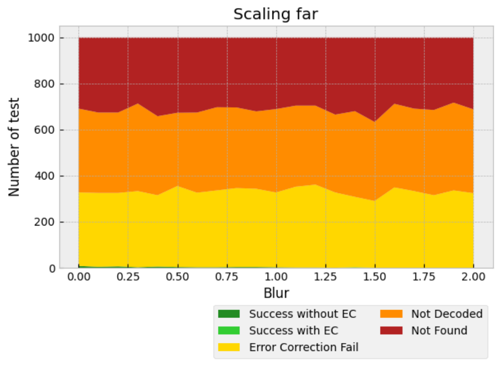


Figure 45: Optimized and base DCC with high noise level charts

5.3.2 Analysis DCC Base - Optimized wiht Moiré

From the previous analysis of data on QRs with moiré (see Section 5.2.2), we realized that the effect damages the images increasing the difficulty ion the detection of the QR code. From the charts representing the decoding rates we can notice that the experimental DCC is better than the basic DCC (see Figures 46 and 47). Analysing the medium noise level, we can see that the success rates on average among the distances are 100% for the closer, 90% for the medium and 78% for the far (see Figures 48, 49 and 50). This good performances are due to the application of error correction that is used in the majority of the decoding increasing the success rate. For the higher noise level, on the other hand, the rate across distances on average are 90% for the closer, 80% for the medium and 60% for the far (see Figures 51, 52 and 53). In contrast to the medium noise level, the success in decoding the QR code in the high noise level is exclusively due to error correction. From this analysis, we can deduce that in the presence of high noise levels, such as moiré, it is not enough to have a low density QR code but the use of error correction is necessary for the successful reading of a QR. From the charts on Figure 50, it can be seen that as the distance increases, the need of error correction is reduced. This happens because in addition to the correlation with the blur, increasing the distance reduces the moiré distortion.

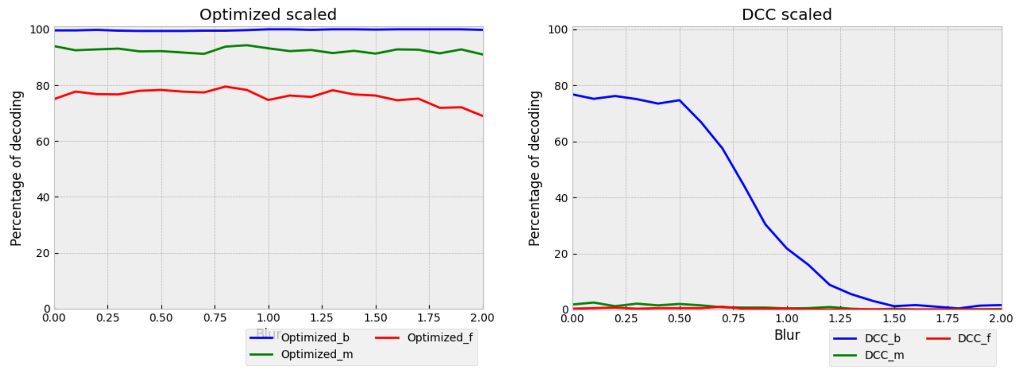


Figure 46: DCC optimized and base DCC with medium noise level

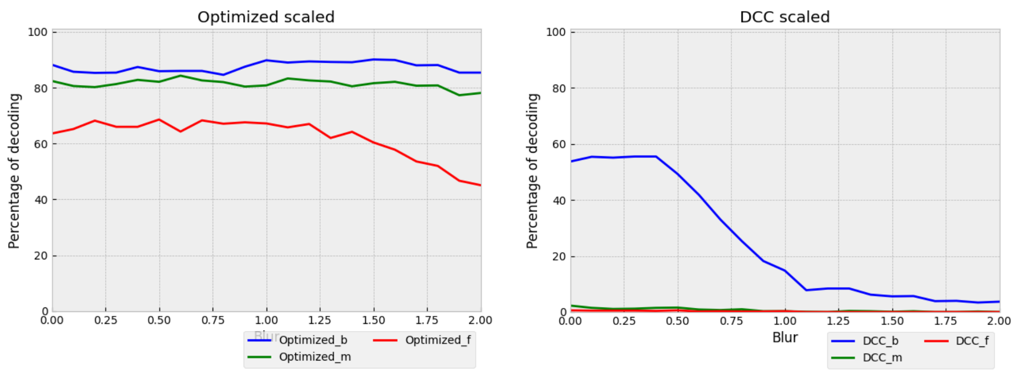


Figure 47: DCC optimized and base DCC with high noise level

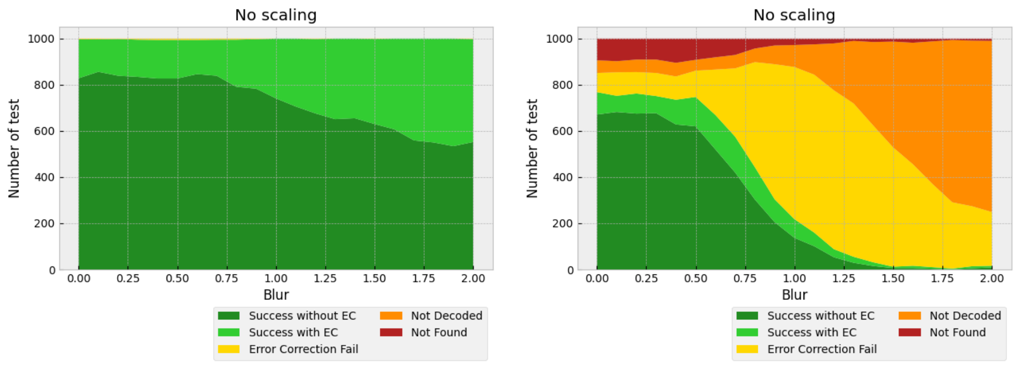


Figure 48: Optimized and base DCC with medium noise level charts

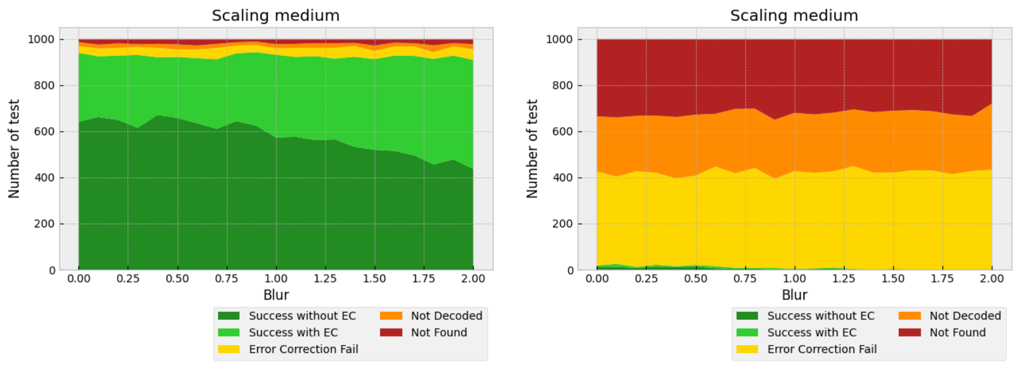


Figure 49: Optimized and base DCC with medium noise level charts

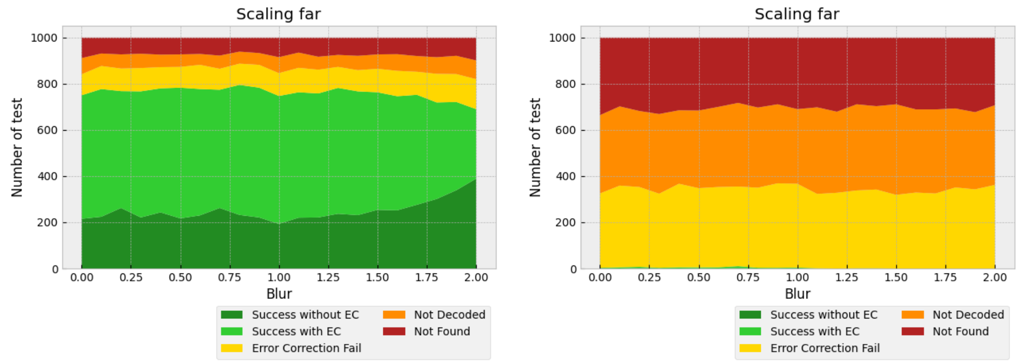


Figure 50: Optimized and base DCC with medium noise level charts

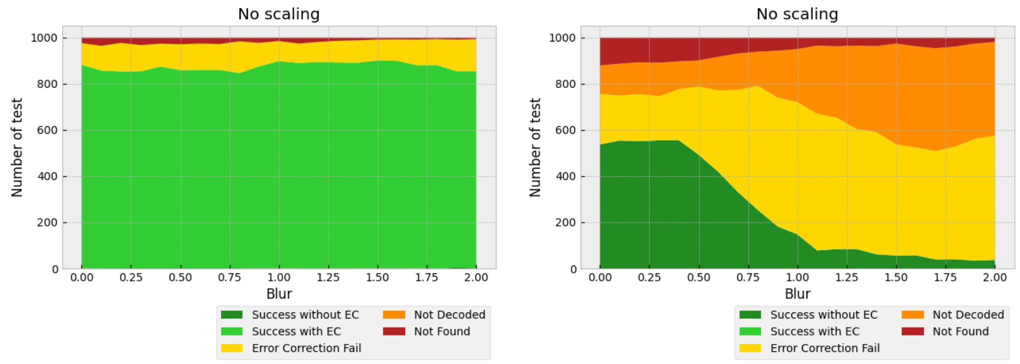


Figure 51: Optimized and base DCC with high noise level charts

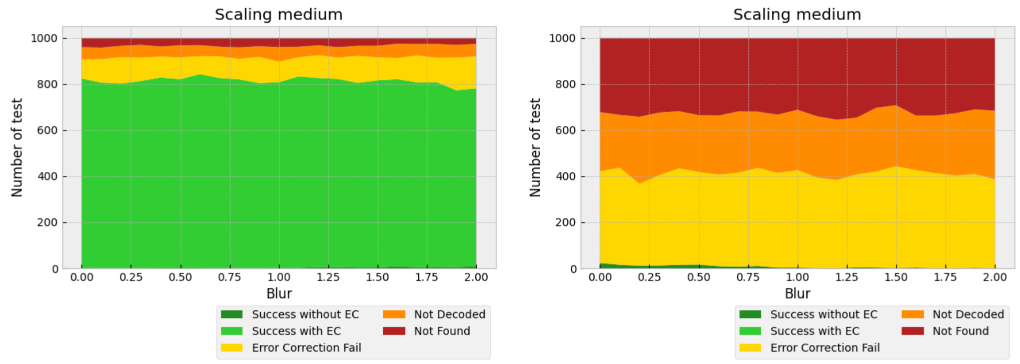


Figure 52: Optimized and base DCC with high noise level charts

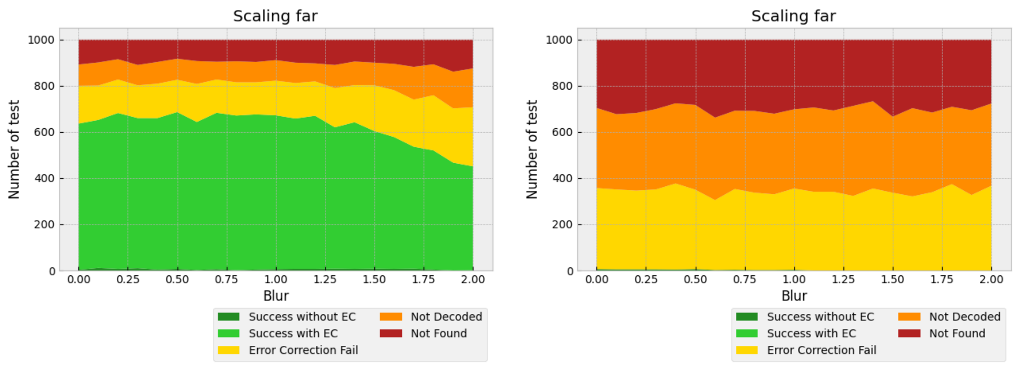


Figure 53: Optimized and base DCC with high noise level charts

Conclusion

This thesis presented the analysis carried on the EU Digital COVID Certificate system and the COVID Certificate system proposed in [2]. The analysis involved the study of the impact of error correction on the decoding rates of the two systems when different image distortions are applied.

In the first tests, two QRs containing the same data and having the same structure were compared, one with error correction level low (7% recoveries) and the other with error correction high (30% recoveries). Different distortions were applied to the images to both tests. From the study done on QRs without moiré effect, we could conclude that the use of error correction is not very important for the correct decoding of the barcode. Instead from the study on QRs with moiré effect, we concluded that the use of error correction is crucial in the successful barcode decoding. Furthermore, an important finding discovered is that, contrary to expectations, error correction high does not improve decoding as the applied distortion levels increase.

In the second study, the basic DCC and the experimental DCC proposed by Arrigo were compared. From the data analysis, it was noticed that both with and without moiré, the experimental DCC has an excellent decoding rate compared to the EU digital COVID certification. These excellent results

are due to both the density of the QR and the use of error correction, which allows it to be read even in cases where the image has been distorted. This huge difference between the two QRs is due to the excellent data structure presented by Arrigo and the choice to use an error correction level low instead of an error correction level quartile as in the case of the DCC.
In conclusion, taking the results of both studies into consideration, we can say that error correction helps the decoding of a QR code, but in the case of dense barcodes such as DCC it is always recommended to use a better data structure and an error correction level low.
From the second study done in this thesis, it was proven that the experimental DCC has better performance than the current DCC. A possible future work can further implement the standard for optimized QR code since at the moment it is only an experimental protocol, thus it has no recognized standard yet.
Starting from the functions we developed, two more studies could be done. The first study could focus on the masks currently used by QR. The study would consists of taking QRs of different densities and, by modifying the existing libraries, applying each type of mask to each QR to see if the mask chosen by penalty score is the most efficient mask at a practical level. The second study consists of finding and applying an improved mask to a QR, comparing its performance with those currently in use. In both studies it is necessary that the library has to be modified or created, because by default all libraries use the pattern provided by the mask to facilitate decoding.
From the studies done, we also noticed that in the presence of certain distortions, as distance and blur increases, the disturbance on the readout de-

creases. Using the libraries available for reading QRs, one could create a library that upon detecting a distortion applies a filter to reduce the damage on the QR or advises the user to move the camera closer or afar from the barcode.

Bibliography

[1] Mask patterns are used to break up solid blocks in a qr code. `https://commons.wikimedia.org/wiki/File:QR_Code_Mask_Patterns.svg`, September 2011.

[2] G. Arrigo. Analysis of the eu digital covid certificate system and proposal of design improvements. Master's thesis in computer science, Ca'Foscari University of Venice, 2021.

[3] K. Arulmani and S. Murali. Automatic rectification of perspective distortion from a single image using plane homography. *International Journal on Computational Science and Applications*, 3:47–58, 10 2013.

[4] C. J. Benvenuto. Galois field in cryptography. `https://sites.math.washington.edu/$\sim$morrow/336$_$12/papers/juan.pdf`, May 2012.

[5] C. Bormann and P. Hoffman. Concise binary object representation (CBOR). Proposed standard, Internet Engineering Task Force (IETF), December 2020.

[6] M. Carfizzi. Proposal of improvements for the digital covid-19 certificate. Master's thesis in computer science, Ca'Foscari University of Venice, 2021.

[7] T. Claeys. pycose. `https://github.com/TimothyClaeys/pycose/blob/master/images/sign1.png`, October 2020.

[8] A. A. Corici, T. Hühnlein, D. Hühnlein, and O. Rode. Towards interoperable vaccination certificate services. In *Proceedings of the 16th International Conference on Availability, Reliability and Security*, ARES 21, New York, NY, USA, 2021. Association for Computing Machinery.

[9] A. de A. Formiga, R. D. Lins, S. J. Simske, G. Dispoto, and M. Thielo. An assessment of data matrix barcode recognition under scaling, rotation and cylindrical warping. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, page 266–267, New York, NY, USA, 2011. Association for Computing Machinery.

[10] E. eHealth Network. ehealth certificate service examples generator. `https://dgc.a-sit.at/ehn/`, June 2021.

[11] E. eHealth Network. Technical specifications for eu digital covid certificates. `https://health.ec.europa.eu/system/files/2022-02/eu-dcc$_$validation-rules$_$en.pdf`, February 2022.

[12] Z. Henry. Set of image the show the penalty score selection. `https://observablehq.com/@zavierhenry/encoding-qr-codes`, April 2021.

[13] ISO. Information technology — automatic identification and data capture techniques — aztec code bar code symbology specification. Stan-

dard 35.040.50, ISO, https://www.iso.org/standard/41548.html, February 2008.

[14] L. Jeng-An and F. Chiou-Shann. 2D barcode image decoding. *Mathematical Problems in Engineering*, (Article ID 848276), December 2013.

[15] M. Jones, E. Wahlstroem, S. Erdtman, and H. Tschofenig. CBOR Web Token (CWT). `https://www.rfc-editor.org/info/rfc8392`, May 2018.

[16] P. Kieseberg, M. Leithner, M. Mulazzani, L. Munroe, S. Schrittwieser, M. Sinha, and E. Weippl. Qr code security. In *Proceedings of the 8th International Conference on Advances in Mobile Computing and Multimedia*, (MoMM '10), page 430–435, New York, NY, USA, 2010. Association for Computing Machinery.

[17] OPenCV. Opencv library. https://opencv.org/about/, October 2022.

[18] F. Patrik, L. Fredrik, and v. G. Dirk-Willem. The Base45 Data Encoding. RFC 9285, Aug 2022.

[19] T. Pornin. Deterministic usage of the digital signature algorithm (DSA) and elliptic curve digital signature algorithm (ECDSA). `https://www.rfc-editor.org/info/rfc6979`, Aug 2013.

[20] Z. Project. Zxing project library. https://github.com/zxing/zxing, October 2022.

[21] G. Roelofs. Zlib manual. `https://www.zlib.net/`, October 2022.

[22] J. Schaad. CBOR object signing and encryption (COSE). `https://www.rfc-editor.org/info/rfc8152`, July 2017.

[23] M. S. Svetlin Nakov and M. Shideroff. *Practical Cryptography for developers.* ISBN: 978-619-00-0870-5. SoftUni (Software University), Sofia, November 2018.

[24] R.-Z. W. Yin-Jen Chiang, Pei-Yu Lin and Y.-H. Chen. Blind qr code steganographic approach based upon error correction capability. *KSII Transactions on Internet and Information Systems*, vol.7(10):17 pages, October 2013.