Epita

First Defense Report

Not a Barcode

EpiCode

Authors:

Stanislas Sokolov

Jack Chouchani

Lucas Martin

Youness Suleiman

February 2018

Table of contents

1 The Project **3**

1.1 Project Synopsis . . . 3

1.2 Project goals . . . 4

1.3 Task distribution . . . 5

2 First Defense Goals **6**

3 Report **6**

3.1 Optical recognition . . . 6

3.1.1 Image processing . . . 8

3.1.2 Segmentation . . . 8

3.1.3 Image processing - Grayscale . . . 9

3.1.4 Image processing - Otsu's Thresholding Method . . . 9

3.1.5 Segmentation - Locate Finder Patterns . . . 10

3.1.6 Segmentation - Extract Bit Matrix, Format and Version 11

3.1.7 Segmentation - Unveiling Mask . . . 12

3.1.8 Segmentation - Matrix Traversal . . . 12

3.2 Encryption & Decryption . . . 14

3.2.1 Encryption . . . 14

3.2.2 Data analysis . . . 15

3.2.3 Data encoding . . . 16

3.2.4 Error Correction Coding . . . 19

3.2.5 Structure Message . . . 20

3.2.6 Data Marking . . . 21

3.2.7 Data Masking . . . 22

3.2.8 Informations Areas . . . 22

3.2.9 Decryption . . . 25

3.3 Reed-Solomon Error Correction . . . 26

3.3.1 Principles of error correction . . . 26

3.3.2 Finite field arithmetic . . . 26

3.3.3 Reed-Solomon Encoding/Decoding . . . 28

3.3.4 For later . . . 31

3.4 Interface . . . 32

3.5 Website 33

4 References **36**

1 The Project

1.1 Project Synopsis

EpiCode will be like a QRCode, it will be a bi-dimensional matrix, machine readable, that contains information about the item to which it is attached. The project aim will be to develop a software capable of both generating *EpiCodes* and reading them.

Generating an EpiCode will be the processing data part. The input data can be a web address, some text, numbers or any other information, and then it is morphed into an image. On the other hand, we have the reading part, which will involve image processing, as well as decrypting of the information contained in the code.

1.2 Project goals

By developing Epicode we plan to complete the following goals :

- Encoding and decrypting Epicode
- Read Epicode
- Read QrCode

Moreover doing such project will benefit us in the following ways :

- Lot of work on algorithm
- Use of image processing

The project will be developed under the following constraints :

- Language of development : **C**
- Work on : **Archlinux**
- Tolerance : **Zero warning or error**
- Target : **Everyone above 2 years old**

1.3 Task distribution

In order to complete the project as quickly and efficiently as possible, we need to have a solid organization. Thus we decided to divide the work, the following table is to show the task distribution.

	Stanislas	Lucas	Jack	Youness
Optical Rec	✓	✓		
Data Encryption / Decryption	✓	✓	✓	✓
Graphical				✓
Website	✓		✓	

2 First Defense Goals

- **Encoding Working**

  Conversion Converting input from numeric, alphanumeric or byte mode to binary, detecting encoding mode, setting basics indicators for encoding(mode indicator, characters counts, binary message and terminating bits). Using encoding indicators to create the QrCode matrix and translate message into the matrix. Draw the QrCode.

- **Decoding Working**

  Image Processing Basic image processing, Grayscale, Black and White thresholding, not optimized and not fully fleshed out in order to simplifiy for a basic segmentation in a "ideal case".

  Segmentation Locate Finder patterns and locate QrCode, no geometric restructuring, in order to work on "ideal cases", retrieve data matrix and the cyphered message in bits.

  Reed-Solomon Implementation of the basics mathematic functions, in order to complete and optimized by the Second Defense.

3 Report

3.1 Optical recognition

The optical recognition is a critical part of the project. In order to read a Qr-Code or EpiCode we will need to retrieve the code from an image. Such a process is alike what we did in the previous project, namely the Optical Character Recognition.

Thus in order to make a readable EpiCode or to retrieve the EpiCode from an image we need to establish some markings/patterns in the code to allow the program to recognize the portion of the image which contain the EpiCode (or the QrCode).

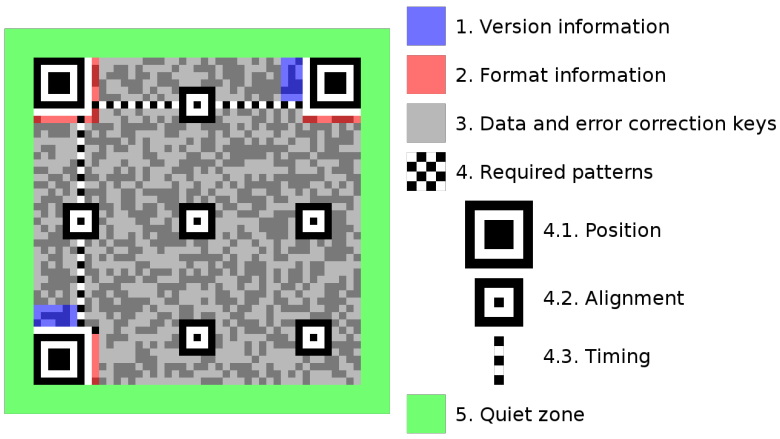


Figure 1: Structure of a normal Qr-Code

We plan to use the basic structure of a Qr-Code to make our own EpiCode's structure. Using this type of marking/pattern will simplify the process of retrieving the code.

3.1.1 Image processing

This part shall be oriented toward the improvement of the image's quality. We will apply multiple filters to make the Qr-Code stand out, similar to what we did to extract the text for the OCR.

Moreover the will check if the image is correctly oriented, if the 3 position patterns are correctly positioned on our image (one to the bottom left, one to the top left and one to the top right). If it is not then we will need to rotate the image and maybe skew it. If it's correctly aligned and there is no need of skewing then the image is ready to be segmented then decoded.

3.1.2 Segmentation

The segmentation will be necessary in order to locate the Qr-Code in an image, this section is not really complicated as Qr-Codes already have a format that eases it's recognition by code.

It will check if the image is correctly oriented, if the 3 position patterns are correctly positioned on our image (one to the bottom left, one to the top left and one to the top right). If it is not , it will be needed to do some geometric rectifications on the image. If it's correctly aligned and there is no need of skewing then the QrCode is ready to be extracted.

After having a correct image to work on, we will segment the image. The segmenting will result in the program retrieving several important informations. We will first extract the format of the Qr-Code which have a fixed position relative to the position pattern, the same for the version.

After retrieving the version and format we will extract the content which is almost all of what's left. To finish we decode it and return the result.

3.1.3 Image processing - Grayscale

Realized by Stanislas SOKOLOV

Grayscale image is one in which the value of each pixel is the value of the intensity of light. A quick computation following a basic reconversion formula into grayscale transforms the image into a image of shades of light.

Let red_v, $green_v$, $blue_v$ be the RGB respective values of a given pixel, $grayscale_v$ is the new RGB value for the given pixel.

$$grayscale_v = red_v * 0,3 + green_v * 0,59 + blue_v * 0,11$$

3.1.4 Image processing - Otsu's Thresholding Method

Realized by Stanislas SOKOLOV

Otsu's Method gives the best output when it comes to thresholding an image into black and white pixels, thus, reducing a graylevel image to a binary image, necessary for the segmentation.

Algorithm: Let L the bins of the histogram, t be the threshold, $\omega_{0,1}(t)$ the weights of the two classes separated by t, and ${\sigma^2}_{0,1}(t)$ the variances from these two classes, $\mu_{0,1}(t)$ the class mean.

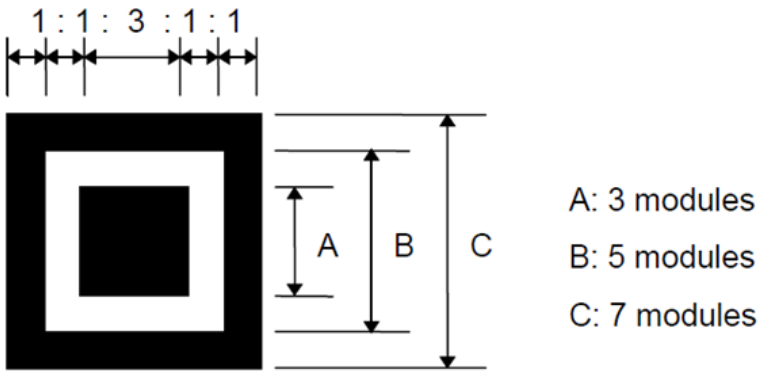
1. Compute histogram and probabilities of each intensity level
2. Set up initial $\omega_i(0)$ and $\mu_i(0)$
3. Step through all possible thresholds $t_1, t_2, ..., t_max$,
   (a) Update ω_i and μ_i
   (b) Compute ${\sigma^2}_b(t)$
4. $MAX({\sigma^2}_b(t))$ corresponds to the desired threshold t.

Compute ω_i, μ_i and ${\sigma^2}_b$:

$$\omega_0(t) = \sum_{i=0}^{t-1} p(i)$$
$$\omega_1(t) = \sum_{i=t}^{L-1} p(i)$$
$$\mu_0(t) = \sum_{i=0}^{t-1} \frac{ip(i)}{\omega_0}$$
$$\omega_1(t) = \sum_{t}^{L-1} \frac{ip(i)}{\omega_1}$$
$${\sigma^2}_b(t) = \omega_0(t)\omega_1(t)[\mu_0(t) - \mu_1(t)]^2$$

3.1.5 Segmentation - Locate Finder Patterns

Realized by Stanislas SOKOLOV



Structure of a finder pattern

Figure 2: Structure of finder pattern

There are free identical Finder Patterns located at the upper left, upper right and lower left corners of the symbol. Each finder pattern may be viewed as three superimposed concentric squares and has a size of 7x7 and 3x3 dark modules and 5x5 light modules. The ratio of module widths in each pattern is 1:1:3:1:1 as illustrated in the image above.

Algorithm Procedure to locate finder patterns

- Horizontal scan of the image in search of the ratio 1:1:3:1:1.
- When the ratio was found, do:
  - Get the center of the found segment.
  - Do a vertical scan from the center of the segment to verify if the pattern persists, update center.

- Do a horizontal scan from the center of the segment to verify if the pattern persists, update center.
- Do a diagonal scan from the center of the segment to verify if the pattern persists.
- If all checks have been passed, check if a similar center has been found before, if not, then declare new Finder pattern at with the computed center.

As we are only working with "ideal" cases for the first defense, when we found the 3 Finder patterns, the bounds of the QrCode are easily found. Thus the process continues directly to bit retrieval.

3.1.6 Segmentation - Extract Bit Matrix, Format and Version

Realized by Stanislas Sokolov

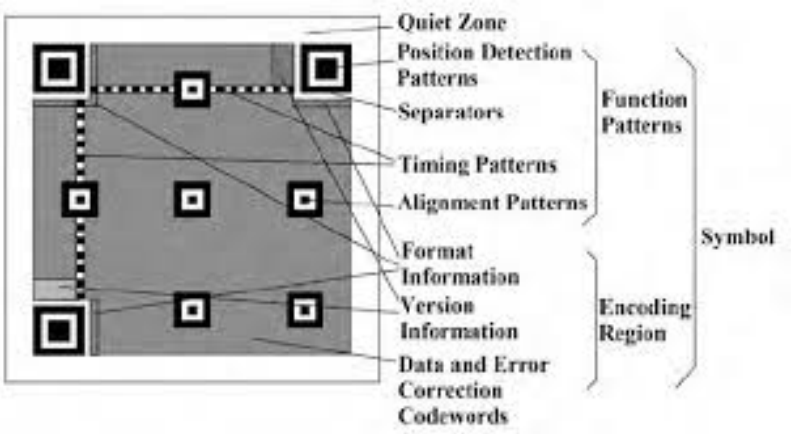


Figure 3: Data location in QrCode

Version and Module size estimation Version is first estimated using the estimated module size, computed while locating the finder patterns, and the total size of the QrCode. Let v be the version, s_p the QrCode size in pixels, s_m the QrCode size in modules, ems the estimated module size, m_s the module size.

$$s_m = \frac{s_p}{ems}$$
$$v = \frac{s_m - 17}{4}$$
$$m_s = \frac{ems * \frac{s_m}{v}}{2}$$

we can then compute more precisely the *ems* and correct the the QrCode coordinates.

Extracting QrCode into a Matrix With the estimated version we can now rescan the image with m_s and get the bit matrix of the QrCode.

Version retrieval When the QrCode is v7 or above, the QrCode has version bits located on the left of the upper right Finder pattern, and above the bottom left Finder pattern. After retrieval of the version and re-transcription into it's decimal form , we can compare it to v_s in case we have an error with the version or the segmentation.

Format retrieval With the new found matrix, the 2 format string are retrieved (and compared in case we are having an error), then it is XORed with 101010000010010 and then we retrieve the error correction level, the mask and the error correction bits.

$$XORed[0:1] = error.correction.level$$
$$XORed[2:4] = mask$$
$$XORed[5:14] = error.correction.bits$$

3.1.7 Segmentation - Unveiling Mask

Realized by Stanislas

Before extracting the bits, the mask must be removed.

3.1.8 Segmentation - Matrix Traversal

Realized by Stanislas SOKOLOV

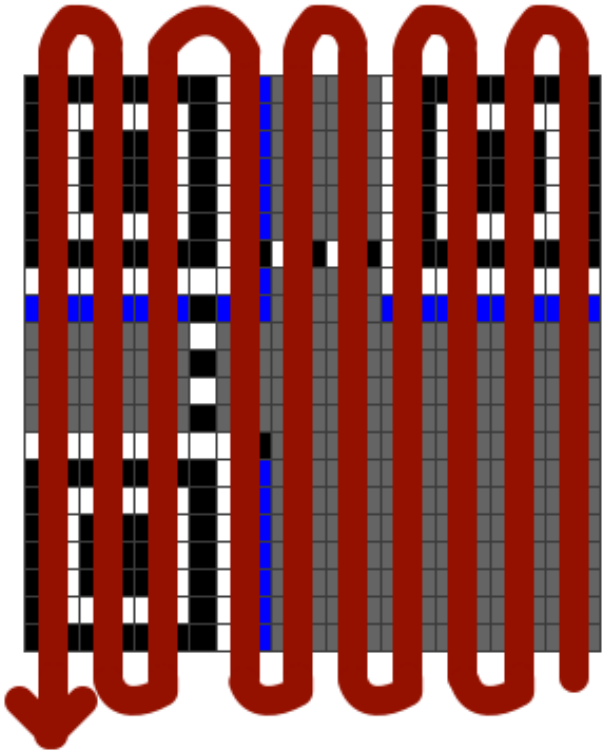


Figure 4: Matrix Traversal in QrCode

In order to retrieve the cyphered message and pass it out through the Reed-Solomon decode algorithm, a traversal through the matrix is necessary. However it is a special traversal that is described in the figure above.

The complexity here relies in avoiding the zones that are not part of the message, that is the version string, format string, finder pattern, alignment pattern, timing pattern...

With the extracted data, we can proceed to the Reed-Solomon decoding.

3.2 Encryption & Decryption

3.2.1 Encryption

In order to create our **Epicodes** we need to have a way to encode data into an image. This process will be called *the encryption*. A powerful algorithm will be needed in order to translate/encrypt our data into a readable code.

To encrypt the data we will need to establish a "language" with which we will translate the data. That language shall be used for the encryption and the decryption process.

To encode the data we will have to perform various step. The first one will be the **Data Analysis** which will determine the most efficient method to handle the data. Then we proceed to encode the data, it's the **Data Encoding** that will segment the data and convert it in the appropriate format.

The third step will be the coding of the **Error Correction** using the Reed-Solomon Error Correction algorithm which will provide error correction code-words to make sure that the program read the data correctly.

The fourth step will be the creation the QR-Code, meaning creating the structure according to the Qr-Code Structure. This process will be called the **Structure Message**.

The fifth step will be the moment when the marks are put down, those mark will direct the reading of the Qr-Code, those patterns/marks are common to all Qr-Codes. We will call this process the **Data Marking**.

The sixth step will be make sure the Qr-Code is readable and to find the best way to make it readable, we use masks, the Qr-Codes have 8 masks, to improve the Qr-Code, this step is known as the **Data Masking**. The last step will be add the **format and version information**. Format version will make the reader able to identify the error correction level and the mask pattern used. The version is used to encode the size of the Qr matrix but are used for rather large Qr-Codes.

3.2.2 Data analysis

Realized by Lucas MARTIN

The data analysis is the most fundamental part of the encoding. During this process we need to determine the type of data we're being given. In our project we will use the following type :

- Numeric encoding
- Alphanumeric encoding
- Byte encoding

Numeric This encoding is the set of the number from 0 to 9. It is useful for encoding "pure" numbers.

Alphanumeric This encoding is the set of the upper case letter (A-Z), 0 to 9 digits and 9 symbols : (space) $ % * + - . / :

Byte This encoding is the set of all upper and lower characters and symbols of the ISO 8859-1 encoding.

During this stage we retrieve the correction parameter and the encoding type. The correction has 4 level which are **LOW**, **MEDIUM**, **QUARTILE** and **HIGH**, each can recovers respectively 7%, 15%, 25% or 30% of the data.

Now that we have the correction level and the encoding type we can proceed to start encoding.

3.2.3 Data encoding

Realized by Lucas MARTIN

As we have the correction level and the encoding type we know need to define the version of the QrCode, the version is ranging from 1(21 * 21 modules) to 40(177 * 177 modules), each being augmented by 4 modules compared to the previous version allowing a bigger capacity.

Find the smallest version When looking for the version we look for the smallest one to optimize the encoding. Having a Qrcode version 40 for encoding "HELLO WORLD" is way to wasteful. To find the version we first count the number of characters we need to encode. For example in "HELLO WORLD" there is 11 characters. According to the character capacities table, the version 1 will be suitable in the correction level Q (Quartile) if the character count is inferior or equal to 16.

Indicators From this we can already create a mode indicator as well as a character count indicator. The mode indicator will be created following the table hereunder :

Mode Name	Mode Indicator
Numeric Mode	0001
Alphanumeric Mode	0010
Byte Mode	0100

Then we encode the character count indicator following the next dimension :

- Version 1 through 9
  - Numeric mode: 10 bits
  - Alphanumeric mode: 9 bits
  - Byte mode: 8 bits
- Version 10 through 26
  - Numeric mode: 12 bits
  - Alphanumeric mode: 11 bits
  - Byte mode: 16 bits
- Version 27 through 40
  - Numeric mode: 14 bits
  - Alphanumeric mode: 13 bits
  - Byte mode: 16 bits

For example in the case of "HELLO WORLD" which is a alphanumeric and if the correction level is Q then we will have the following indicator :

Mode indicator	Character count
0010	000001011

Encoding the string(input) To encode we need to see which encoding mode we have to following a certain way.

Numeric mode If the input is a string of numbers then we need to break up the string in group of three digits. The break should start from the left and finish with the unit. If the string length is not a multiple of 3 then the last might will be composed of 1 or 2 digits. After this convert the group into binary.

Alphanumeric mode To encode from alphanumeric we need to break up the input into pairs of 2 letter, if the length is not a multiple of 2 then we will have a last group of 1 letter. Then convert each letter to its decimal value (based on the hexadecimal (example : A = 10 & B = 11)). After that multiply by 45 the first value and add the second. Convert the obtained value in binary. If the number of character is odd we will encode the last one has a 6 bits value.

Byte mode To encode from byte mode we will have to convert each character into its hexadecimal value and then to convert it to 8 bit binary string. If necessary, we will use 0 to fix the string (from the left).

Still using the example of "HELLO WORLD" with correction level Q and alphanumeric encoding :

Mode indicator	Character count
0010	000001011

converted string
01100001011 01111000110 10001011100 10110111000 10011010100 001101

After encoding the string we need to see if the whole data block(indicators + encoded string) match the size specified in the specification of the QrCode. If not we can add up to 4 terminating '0'. If adding the terminating '0' is not sufficient we will add some special byte in the string which are : "11101100" and "00010001". Those value can be used as much as needed to fill the space.

3.2.4 Error Correction Coding

Realized by Lucas MARTIN & Jack CHOUCHANI

Before encoding the correction codes we need to break up the data into codewords. The codewords will be split in groups and blocks following specification (relative to the version and correction level). After obtaining groups/blocks/codewords we will start computing correction codes. To do so we will use the **reed-Solomon algorithm**. Using polynomials we will be able to obtain codes which will act as life saver in case of data corruption in the QrCode.

3.2.5 Structure Message

Realized by Lucas MARTIN

After computing the error correction codes we will now create the final message (data to put in the QrCode matrix). For that we need to interleave the blocks (normals and error corrections blocks (which are the values computed earlier)) following this procedure :

1. Interleave normal block
    - take the first data codewords of the first block
    - take the first data codewords of the second block
    - take the second data codewords of the first block
    - take the second data codewords of the second block
    - .... continue until all blocks and codewords have been used
2. Interleave error correction block
    - take the first error correction codewords of the first block
    - take the first error correction codewords of the second block
    - take the second error correction codewords of the first block
    - take the second error correction codewords of the second block
    - .... continue until all blocks and codewords have been used

When this is done we convert the string to binary. We might need to add remainder 0 following the version.

3.2.6 Data Marking

Realized by Lucas MARTIN & Stanislas SOKOLOV

Now that we have the final data string we need to make the array of the QrCode in order to create a image of the QrCode later. We need to place the finder pattern as well as the timing pattern and the alignment pattern without forgetting the various informations area.

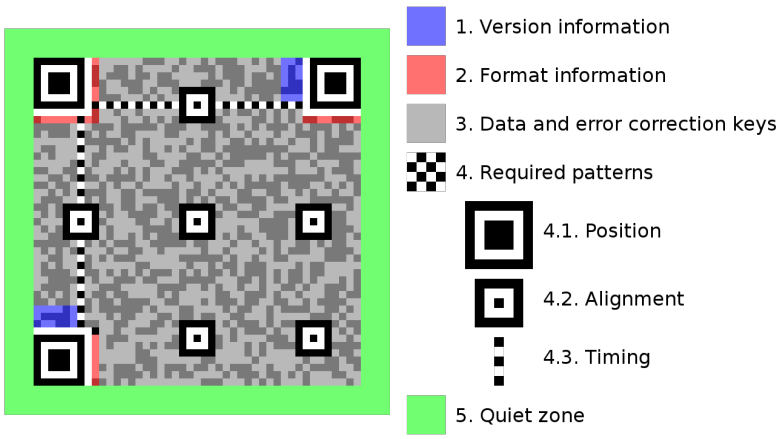


Figure 5: Structure of a normal Qr-Code

Those pattern being placed, we can now write our data and mask the Qrcode.

3.2.7 Data Masking

Realized by Lucas MARTIN & Stanislas SOKOLOV

Definition of Masking from thonky.com *If a module in the QR code is "masked", this simply means that if it is a light module, it should be changed to a dark module, and if it is a dark module, it should be changed to a light module. In other words, masking simply means to toggle the color of the module.*

The QrCode has a total of 8 possible mask, it is possible to use several but at the moment we're only using one. To test which mask is the best (which one improve the best the readability of the QrCode) we perform 4 tests.

- The first rule gives the QR code a penalty for each group of five or more same-colored modules in a row (or column).
- The second rule gives the QR code a penalty for each 2x2 area of same-colored modules in the matrix.
- The third rule gives the QR code a large penalty if there are patterns that look similar to the finder patterns.
- The fourth rule gives the QR code a penalty if more than half of the modules are dark or light, with a larger penalty for a larger difference.

By counting the total penalty of each mask we are able to determine which one has the lowest penalty and thus is the best to use in our case.

3.2.8 Informations Areas

Realized by Lucas MARTIN

At this point we just need to put format string into the matrix and the version informations if the version is greater than 6 to be finished.

To retrieve the format string we can compute it using the polynomial generator and the error correction bits or directly fetch it from a tabular. There is a total of 28 formats string which won't change. Following our the correction level and the mask used we will find our corresponding format string. After that we put it in the format string area which are located near the finder pattern (2 at the top left, 1 at the bottom left and 1 at the top right).

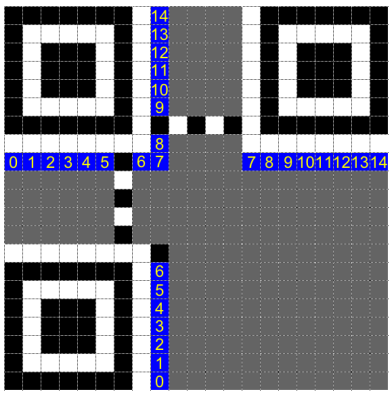


Figure 6: Format string area and their value (in blue)

The format information string is placed below the topmost finder patterns and to the right of the leftmost finder patterns, as shown in the image below. The number 0 in the image refers to the most significant bit of the format string, and the number 14 refers to the least significant bit.

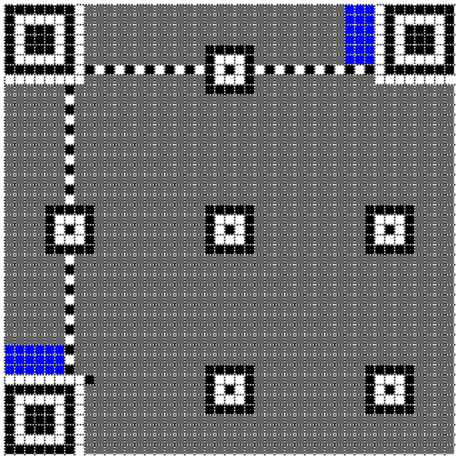


Figure 7: Version information areas (in blue)

After adding everything the encoding and drawing of the QrCode is done.).

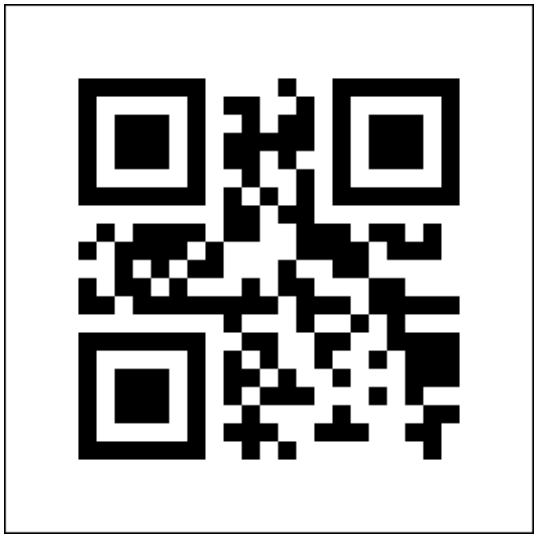


Figure 8: "HELLO WORLD" QrCode

3.2.9 Decryption

The decryption process will be easier than the encryption because it will be a reverse operation. By using the same language we will be able to retrieve the data. We will be using SDL to retrieve the Qr-Code and then we'll reverse the encryption process to get the data.

By retrieving the format, the version and the patterns in the Qr-Code we will be able to decode the data from the Qr-Code.

3.3 Reed-Solomon Error Correction

Realized by Jack CHOUCHANI

Error correcting codes are a signal processing technique to correct errors. They are nowadays ubiquitous, such as in communications (mobile phone, internet), data storage and archival (hard drives, optical discs CD/DVD/BluRay, archival tapes), warehouse management (barcodes) and advertisement (Qr codes).

Reed–Solomon error correction is a specific type of error correction code. It is one of the oldest but it is still widely used, as it is very well defined and several efficient algorithms are now available under the public domain.

3.3.1 Principles of error correction

The main insight of error correcting codes is that, **instead of using a whole dictionary of words, we can use a smaller set of carefully selected words, a "reduced dictionary", so that each word is as different as any other.** This way, when we get a message, we just have to lookup inside our reduced dictionary to **1) detect** which words are corrupted (as they are not in our reduced dictionary); **2) correct** corrupted words by finding the most similar word in our dictionary.

3.3.2 Finite field arithmetic

We'd like to define addition, subtraction, multiplication, and division for 8-bit bytes and always produce 8-bit bytes as a result, so as to avoid any overflow. Naively, we might attempt to use the normal definitions for these operations, and then mod by 256 to keep results from overflowing. So we decided to use the **uint8_t** type for our integers. Here's a brief introduction to Galois Fields: a finite field is a set of numbers, and a field needs to have six properties: Closure, Associative, Commutative, Distributive, Identity and Inverse. More simply put, using a field allow to study the relationship between numbers of this field, and apply the result to any other field that follows the same properties. Here we will define the usual mathematical operations that you are used to doing with integers, but adapted to GF(2^8), which is basically

doing usual operations but modulo 2^8.

Addition and Subtraction Both addition and subtraction are replaced with exclusive-or in Galois Field base 2. This is logical: addition modulo 2 is exactly like an XOR, and subtraction modulo 2 is exactly the same as addition modulo 2. This is possible because additions and subtractions in this Galois Field are carry-less.

Thinking of our 8-bit values as polynomials with coefficients mod 2:

$$0101 + 0110 = 0101 - 0110 = 0101 \wedge 0110 = 0011$$

Multiplication with logarithms The procedure to implement Galois Field multiplication is special, the traditional way for the multiplication of 2 numbers takes up to eight iterations of the multiplication loop, followed by up to eight iterations of the division loop. However, we can multiply with no looping by using lookup tables. One solution would be to construct the entire multiplication table in memory, but that would require a bulky 64k table. The solution described below is much more compact.

First, notice that it is particularly easy to multiply by 2=00000010 (by convention, this number is referred to as α or the generator number): simply left-shift by one place, then exclusive-or with the modulus 100011101 if necessary.

We notice that the binary form of a number in the Galois Field starts with $\alpha^0 = 00000001$ and ends with $\alpha^{254} = 11111111$. Thus, every element of the field except zero is equal to some power of α. The element α, that we define, is known as a primitive element or generator of the Galois field.

The problem is, how do we find the power of α that corresponds to 10001001? This is known as the discrete logarithm problem, and no efficient general solution is known. However, since there are only 256 elements in this field, we can easily construct a table of logarithms. While we're at it, a corresponding table of antilogs (exponentials) will also be useful. So we can have the multiplication by performing this operation: **gf_mul(x,y) = gf_exp[gf_log[x] + gf_log[y]]**

Division Another advantage of the logarithm table approach is that it allows us to define division using the difference of logarithms. In the code below, 255 is added to make sure the difference isn't negative. And the operation is almost the same: **gf_div(x,y) = gf_exp[(gf_log[x] + 255 - gf_log[y]) % 255]**

Power and Inverse The logarithm table approach will once again simplify and speed up our calculations when computing the power and the inverse:
gf_pow(x,y) =gf_exp[(gf_log[x] * power) % 255]
gf_inv(x,y) =gf_exp[(255 - gf_log[x])]

Now that we have all the basic operations, we develop the operations on polynomials in the Galois Field and then we go on to the Reed-Solomon Encoding/Decoding.

3.3.3 Reed-Solomon Encoding/Decoding

Encoding outline Like BCH codes, Reed-Solomon codes are encoded by dividing the polynomial representing the message by an irreducible generator polynomial, and then the remainder is the RS code.

To summary, with an approximated analogy to encryption: our generator polynomial is our encoding dictionary, and polynomial division is the operator to convert our message using the dictionary (the generator polynomial) into a RS code. The encoding goes as follow, we start by generating the irreducible generator polynomial, then we perform a polynomial division on the message to encode with the polynomial. And then the remainder from the division is our RS code. It is very simple and straightforward.

Rs Decoding Reed-Solomon decoding is the process that, from a potentially corrupted message and a RS code, returns a corrected message. In other words, decoding is the process of repairing your message using the previously computed RS code.

Although there is only one way to encode a message with Reed-Solomon, there are lots of different ways to decode them, and thus there are a lot of different decoding algorithms.

However, we can generally outline the decoding process in 5 steps:

1. Compute the syndromes polynomial. This allows us to analyze what characters are in error using Berlekamp-Massey (or another algorithm), and also to quickly check if the input message is corrupted at all.

2. Compute the erasure/error locator polynomial (from the syndromes). This is computed by Berlekamp-Massey, and is a detector that will tell us exactly what characters are corrupted.

3. Compute the erasure/error evaluator polynomial (from the syndromes and erasure/error locator polynomial). Necessary to evaluate how much the characters were tampered (ie, helps to compute the magnitude).

4. Compute the erasure/error magnitude polynomial (from all 3 polynomials above): this polynomial can also be called the corruption polynomial, since in fact it exactly stores the values that need to be subtracted from the received message to get the original, correct message (i.e., with correct values for erased characters). In other words, at this point, we extracted the noise and stored it in this polynomial, and we just have to remove this noise from the input message to repair it.

5. Repair the input message simply by subtracting the magnitude polynomial from the input message.

In addition, decoders can also be classified by the type of error they can repair: erasures (we know the location of the corrupted characters but not the magnitude), errors (we ignore both the location and magnitude), or both errors-and-erasures.

Syndrome Calculation Decoding a Reed-Solomon message involves several steps. The first step is to calculate the "syndrome" of the message. Treat the message as a polynomial and evaluate it at α^0, α^1, α^2, ..., α^n. Since these are the zeros of the generator polynomial, the result should be zero if the scanned message is undamaged (this can be used to check if the message is corrupted, and after correction of a corrupted message if the message was completely repaired). If not, the syndromes contain all the information necessary to determine the correction that should be made.

Erasure correction It is more convenient to correct mistakes in the code if the locations of the mistakes are already known. This is known as erasure correction. It is possible to correct one erased symbol (ie, character) for each error-correction symbol added to the code. If the error locations are not known, two EC symbols are needed for each symbol error (so you can correct twice less errors than erasures). This makes erasure correction useful in practice if part of the QR code being scanned is covered or physically torn away. It may be difficult for a scanner to determine that this has happened, though, so not all Qr code scanners can perform erasure correction. We will use the the Forney algorithm to calculate the correction values (also called the error magnitude polynomial).

Error correction In the more likely situation where the error locations are unknown (what we usually call errors, in opposition to erasures where the locations are known), we will use the same steps as for erasures, but we now need additional steps to find the location. The Berlekamp-Massey algorithm is used to calculate the error locator polynomial, which we can use later on to determine the errors locations.
Then, using the error locator polynomial, we simply use a brute-force approach called trial substitution to find the zeros of this polynomial, which identifies the error locations (ie, the index of the characters that need to be corrected). A more efficient algorithm called Chien search exists, which avoids recomputing the whole evaluation at each iteration step, and this algorithm may be implemented later on for optimization.

Error and erasure correction It is possible for a Reed-Solomon decoder to decode both erasures and errors at the same time, up to a limit (called the Singleton Bound) of 2*e+v ¡= (n-k), where e is the number of errors, v the number of erasures and (n-k) the number of RS code characters (called nsym in the code). Basically, it means that for every erasures, you just need one RS code character to repair it, while for every errors you need two RS code characters (because you need to find the position in addition of the value/magnitude to correct). Such a decoder is called an errors-and-erasures decoder, or an errata decoder.

3.3.4 For later

One immediate issue that you may have noticed is that we can only encode messages of up to 256 characters. This limit can be circumvented by several ways, the three most common being:

1. Using a higher Galois Field, for example 2^{16} which would allow for 65536 characters, or 2^{32}, 2^{64}, 2^{128}, etc. The issue here is that polynomial computations required to encode and decode Reed-Solomon become very costly with big polynomials (most algorithms being in quadratic time, the most efficient being in n log n such as with number theoretic transform).

2. By "chunking", which means that you simply encode your big data stream by chunks of 256 characters.

3. Using a variant algorithm that includes a packet size such as Cauchy Reed-Solomon.

3.4 Interface

The **EpiCode** UI will be minimalist and user-friendly. We will have a window for reading **Epicodes** and another for when we are generating them. We plan on using SDL for the Graphical User Interface, and develop an Android application using Android NDK, which let us develop an Android app using the C language.

The GUI allows the user to have access to the features by a 'graphical' way. Thus the user has the opportunity not to use the shell but to use a special window with some different buttons linked with different specific functions.

Actually, in this project, the GUI is not crucial for this project, because it is still usable in the shell. Nevertheless, a such interface permits to make the program easier to run.

It could permit the users to interact with the program, because it shows the different options in a single simple window.

When we thought about how we could organize the different buttons on our window. So we made inquiries about the most typical tips to elaborate such interfaces:

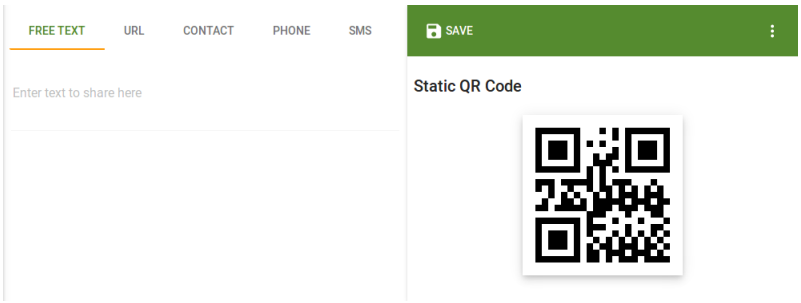


Figure 9: QR code generator template

GTK & Glade In order to realize our GUI (to make the program use more pleasant), we chose to use the library GTK with a software called Glade. GTK is a library which allows us to display different widgets such as text, images, etc... It offers us the ability to have an efficient and a pleasurable interface.

***Bonus:* Android NDK** The Android NDK is a toolset that lets you implement parts of your app in native code, using languages such as C and C++. For certain types of apps, this can help you reuse code libraries written in those languages. The Android NDK will let us develop an Android app using the C language and thus will let us have a mobile application. We might look for other solutions in the future or even choose not to opt for such program after all.

Figure 10: Android NDK

3.5 Website

Realized by Jack CHOUCHANI and Stanislas SOKOLOV

In order to better present our project, there is a need for a website to showcase all the functionalities. A static web page will be developed, it will contain information about the project, the team and a download page to try **Epicode**.

We will try to keep the website updated with every feature we are adding. The website will be hosted on GitHub and will be freely available for anyone who wants to check about a thing or report an issue.

Figure 11: GitHub Pages

GitHub Pages, as we found that it was the most suitable technology to meet our needs. GitHub Pages is designed to host static web sites directly from a GitHub repository. Alongside GitHub Pages, Jekyll will help to create the base of our page and then the page will be modified to our needs and style.

Figure 12: Jekyll

Jekyll is a simple, blog-aware, static site generator. It takes a template directory containing raw text files in various formats, runs it through a converter (like Markdown) and the Liquid renderer, and spits out a complete, ready-to-publish static website suitable for serving with your favorite web server. Everything is editable which lets us adapt it to **Epicode**.

List of Figures

1 Structure of a normal Qr-Code 7
2 Structure of finder pattern 10
3 Data location in QrCode 11
4 Matrix Traversal in QrCode 13
5 Structure of a normal Qr-Code 21
6 Format string area and their value (in blue) 23
7 Version information areas (in blue) 23
8 "HELLO WORLD" QrCode 24
9 QR code generator template 32
10 Android NDK 33
11 GitHub Pages 34
12 Jekyll 34

4 References

- https://pages.github.com
- https://jekyllrb.com
- https://www.thonky.com/qr-code-tutorial/introduction
- http://blog.qartis.com/decoding-small-qr-codes-by-hand/
- https://developer.android.com/ndk/index.html
- http://http://libsdl.org
- https://en.wikiversity.org/wiki/Reed–Solomon_codes_for_coders
- https://en.wikipedia.org/wiki/Finite_field_arithmetic