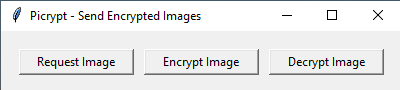
George Marché

CS 563

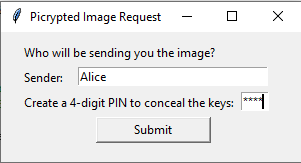
Programming Project

I originally had confidence in the idea for the project I wanted to implement, but underestimated how much patience and effort it would take. For this project, I implemented a program that would take an image, break it down into pixels, encrypt each of the pixel values (the “rgb”) using Elgamal encryption through the cipher block chaining mode (CBC), place those pixel values into a CSV, then have the other side decrypt the pixel values in the original.

The main interface of this program consists of 3 buttons: the “Request Image” button, the “Encrypt Image” button, and the “Decrypt Image” button.

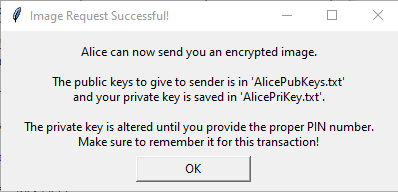


The person that would like an image sent to them (“Bob”) will click on the “Request Image” button, where a new window will appear. In this window, the user will add the image sender’s name and set a 4-digit PIN. After submitting this request, the program will handle errors regarding a sender name being empty or non-alphabetic and a PIN that is empty or non-numeric. The program will then produce a random prime number *p* larger than 256 and smaller than 787 and calculate a list of primitive elements. It randomly selects one of the primitive elements to be *α* (“alpha”). The private key *d* is then selected to be between 2 and 2 less than *p*, and *β* (“beta”) is calculated as *αd* mod *p*.

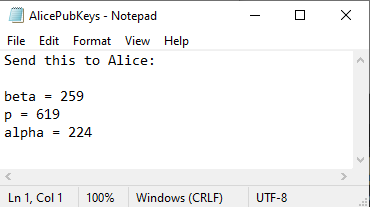


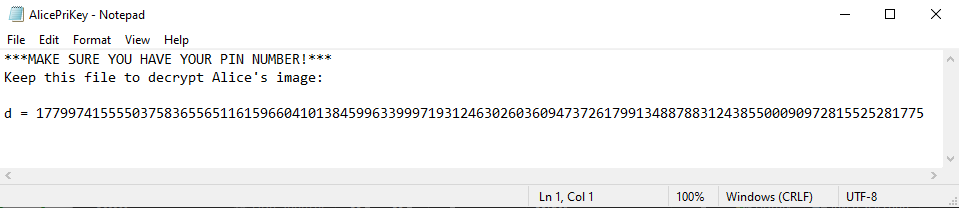
The values of *β*, *p*, and *α* are saved into a text file that should be sent to the sender (named “*[SenderName]PubKeys.txt*”). This file can be sent by any means that the user decides. Another text file is created to save the private key *d*, but, since *d* is a private key, it was decided that it would be converted into a much larger number based on the PIN number provided, which the user should remember for a later time. The private key would first be multiplied by the PIN number, then raised to the power of the PIN number integer divided by 300, then reduced by the PIN modulo 100 (the last 2 digits). Though not related to the cryptographic concepts we’ve learned, I thought it would be a decent way to protect the private key from snooping eyes.

After the submission of the request is complete, a message letting the user know the request was successful will direct the user to send the “*[SenderName]PubKeys.txt*” file, protect the “*[Sendername]PirKey.txt*” file, and remember the PIN number for the rest of the transaction.

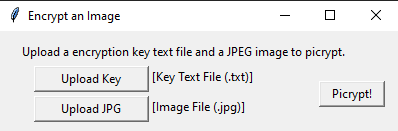


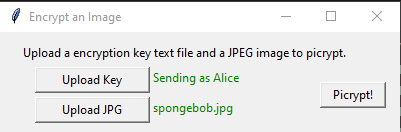
The text files have the parameters saved in them, with a few guiding directions at the top of the file. Below is an example of how each of the text files look when they are opened. The top file is the one that must be sent, with the bottom one being the private key that must be saved. Notice how long the number is representing *d*, though *d* is originally far smaller than that.



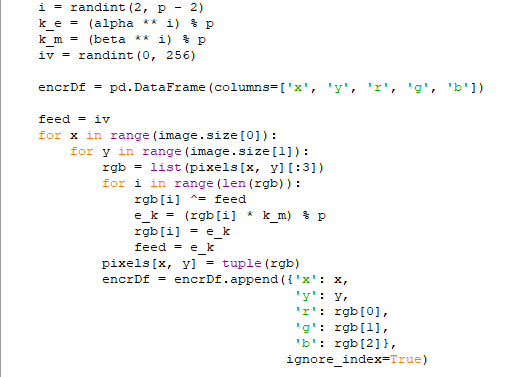


On the sender’s (“Alice’s”) side, they will get the text file with the public keys from the image recipient. It is then their turn to send the picture. The sender will press on the “Encrypt Image” button, where they will be prompted to upload the public key text file they received from the image recipient and the JPG file that is to be “picrypted”. The program will display the name of the sender (“Sending as *[SenderName]*”) and the image file name. If the upload was unsuccessful, it will display that as well. The sender then presses the button “Picrypt!”, where the program will then parse the public key values from the text file to start the process.

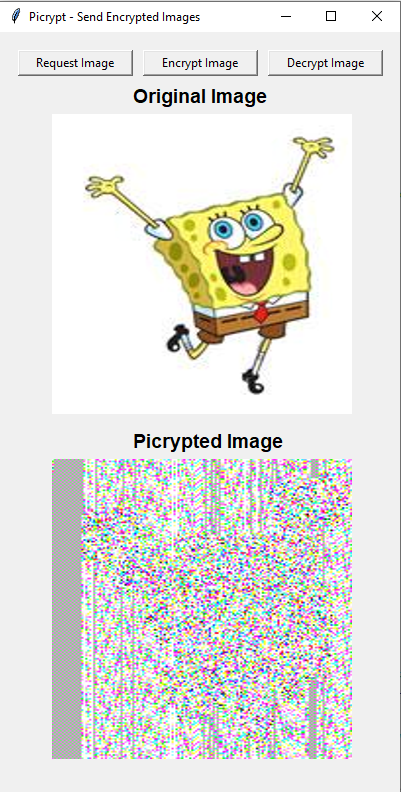


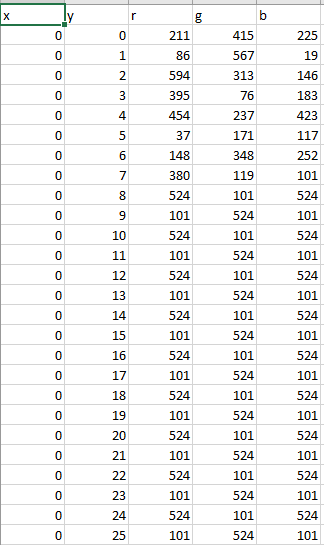


After the key values have been established from the file, the program will then produce the sender’s private key *i* to be a random number between 2 and 2 less than *p*. It then calculates the ephemeral key *KE* and the masking key *Km* and selects a random number less than 256 to be the initialization vector *IV*. The image will then be parsed into its individual pixels. Picrypt will then iterate through each pixel, then iterate through the 3 color codes (*r*, *g*, and *b*). Each color code is processed through an output feedback mode and XORed with *IV* initially and then the result of the previous encrypted color code from thereon forth. It is then encrypted by multiplying the color code with *KM* and setting it to modulo *p*.

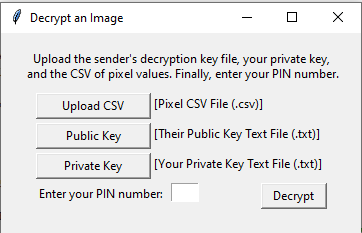


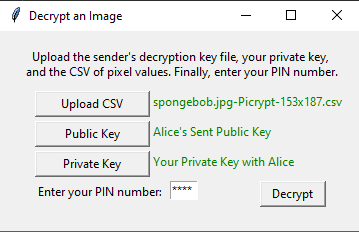
In each iteration, the pixel is set to the new encrypted value and a dataframe with the columns *x*, *y*, *r*, *g*, and *b* is populated with the pixel position (*x*, *y*) and the new color values (*r*, *g*, *b*). The original image and “picrypted” image are displayed on top of each other so the sender can see the transformation. The complete dataframe is placed in a new CSV file specific to the image (“*[JPGFileName]-Picrypt-[Width]x[Height].csv*”). This CSV file is to be sent to the recipient. Also, the ephemeral key *KE* and initialization vector *IV* is saved in a text file that must be sent to the recipient as well. The CSV file being the communication of the image rather than the encrypted image itself was decided because the image’s pixels were altered in the process of communication for some odd reason. The encrypted pixel values were different after the image was saved when the encrypted image was uploaded again. It was perplexing to me. However, the CSV method may have the advantage of being more secure, since having the communication of an image, though encrypted, could mean having it at risk of an attacker getting a vague idea of what the image originally looked like, despite the encryption. The encryption of images is not fast, however; performing Elgamal with CBC mode on a 153 pixel by 187 pixel image, which is fairly small compared to most others, took nearly a minute.





Once the recipient receives the CSV and text file with the sender’s ephemeral key and initialization vector, the sender presses on the “Decrypt Image” button. A window pops up that prompts the user to upload the CSV file with the “picrypted” pixel values, the text file with the values required for decryption sent by the sender (“*SentKey-[SenderName]PubKeys.txt*”), and the private key that was supposed to be saved. Finally, the user will have to fill in their 4-digit PIN number in order for the private key *d* to be recalculated to its original value.





Once the sender presses “Decrypt”, the program will process the Elgamal decryption using the reverse CBC mode. Oddly enough, the decryption process turned out to be far faster than the encryption process; this may likely be because of the fact that the encryption process consisted of both producing an image and populating a growing dataframe. The result comes back with the original image. However, it’s odd that the image has a few blemishes in it, where a few pixels did not come out correctly. I’m not honestly sure why that is so; it may be because of the large number of calculations that are required to output the decrypted image or some alteration that occurred in the loading or calculation process. I have checked my code several times and am still unsure as to why this occurred. This doesn’t occur when the CBC mode calculations were commented out, strangely enough. Despite this, the image came out nearly identical to the original image, meeting the objective.

Both the “picrypted” image and original image are displayed below the buttons in the main window, and the decrypted image is also downloaded as “*Decrypted-[ImageFileName.jpg*”. Below are images of the transformation after decryption, the left one with CBC mode and right one without it. Notice that the image isn’t decrypted enough without a blockchain mode.

