Secure Data Transfer with Multi-Layer Security using Different Enc Standards and Steganography

Rohan Mode (rm5532)
Rakesh Puri (rp9708)

Abstract

Even with the present technology in cyber security, the protection of data is still a big challenge. Especially when the data is stored in advanced pictures such as QR codes and barcodes. The information stored in QR codes can easily be read with the help of barcode scanners. The weakness of correspondence of such advanced pictures has become a critical issue these days, specifically when the pictures are communicated through uncertain channels where the information in the advanced pictures is vulnerable to leak. Picture encryption procedure ought to be planned in a manner that works on the adequacy of correspondence and keeps up with the information safe and secure from an assault of adversaries over the insecure channel. However, encryption only suffices the purpose of confidentiality in this case, what if the data is altered and the receiver is not able to validate the integrity of information stored in the advanced picture.

In this project, we focus on adding multiple layers of security on QR codes to achieve confidentiality and integrity. Multiple layers of security involve a hash watermark and hiding the QR code in a Steganographic image. Further, encrypting the image with different encryption standards, then transferring the image over the cloud between servers in a more secure way. Our approach for multiple layers of security is based on symmetric encryption and steganographic theories.

Keywords – AES; DES; RC4; SHA-256; Steganography; AWS Elastic Compute Cloud (EC2); Secure Copy (SCP).

1 Introduction

With the advancement in technology use of advanced pictures is increasing day by day. Here we are referencing advanced pictures with QR codes and Barcodes. You can store a great deal of critical information in a QR code. The information like most of the companies like Uber, Zomato, Ola, and Swiggy are using QR codes for their payments. Most of the online payment platforms like PhonePe, Paytm, and Google Pay are using QR codes as their main source of transaction. Apart from payment details you can store any kind of information which is critical. The information stored in QR codes can be read easily just by scanning them with scanners. This functionality of QR codes make them more effective to be used for some instances, however with great powers comes great vulnerabilities, since transmitting these QR codes containing critical information through unsecure channel, say on internet, is more susceptible to attacks and leaking such important information would lead to serious damage. This is where we come up with a more secure solution for adding multi-layer security via combining the features of steganography [6] with encryption to provide confidentiality along with integrity check via watermarking the QR code.

Our approach explains that how we can improve the security of the information stored in QR codes and securely transmitting them over a cloud. We have used different symmetric key encryption standards such as AES, DES, and RC4 [3], with which the end-user could encrypt and decrypt the QR code. We planned to use symmetric key encryption standards as we tend to analyze the execution cost for faster implementations considering the implementation with less computing power. For securely transmitting the data over the cloud between two servers/machines, we are using Secure Copy (SCP) as it is based on two major protocols named

SSH (Secure Shell), providing both confidentiality via encryption and authentication via password requirement, and RCP (Remote Copy) which is basically providing features of file transfers.

We successfully added multi-layer security for our use case. Generated the QR code [4] with encoding hash in it to further check the integrity once the message is received on the other end by the receiver. For each step we added key passphrase so that only the user who is aware about the passphrase could perform action on top of it. Using steganography for watermarking and hiding the encrypted data became the most essential part of our security architecture and solution for our use case. Using AWS EC2 (Elastic Compute Service) happened to enhance the security for our solution on cloud for remote access of machines and performing cryptographic task on top of them.

Our results and analysis show that adding multi-layer security would be a better approach to solve the issue of data compromise, shared over the unsecured channel. We were able to demonstrate that we can successfully generate a QR code with a secret message stored in it, add watermark for integrity check, encrypt the image with multiple encryption standards and get a final output of encrypted data hidden in a final unimportant steganographic image ready to be transferred over the cloud remotely between two machine or servers. Successfully decrypting and checking the integrity of the message. For our main analysis of comparing the speeds of encryption standards performing the best and the worst for this use case, we noted the execution time for each process in the architecture for both encryption and decryption and successfully showed what algorithm performs the fastest and accordingly.

3 Our Approach

The security of critical information that is transmitted over an unsecure channel is really important, as encryption only suffices the purpose of confidentiality, what if the data is altered and the receiver is not able to validate the integrity of information received. This becomes a serious privacy issue, as in our case if the QR codes contains some crucial information like payment information, banking details, Personally Identifiable Information (PII) then leaking of such information could cause serious damage. Also, just encrypting the data and transmitting over an unsecured channel is vulnerable to multiple attacks as all the encryption algorithms are out open on internet.

Our solution completely focuses on improving the security of the critical information by adding multiple layers of security and securely transmitting the information from sender to receiver along with integrity check on the receiver side over the cloud. The following architectures explain the entire workflow of our executions:

- Client-Side Encryption
- Embedding Hash to QR code with LSB Steganography
- Secure Copy (SCP)
- Receiver Side Decryption and Integrity Check

3.1 Client-Side Encryption

In reference to figure 1, the complete workflow process of client-side encryption executes on the sender's machine. The sender basically must run the python script that we have written and

following the instruction accordingly and provide the user inputs with respect to the requirements. This way the sender would be able to generate a final steganographic image with the encrypted QR code containing a secret message or information with hash watermark embedded in it.

Once the user runs the script at first it will ask for QR code generation, for generating the end-user will have to provide a secret message to store in the QR code so that when one scans it would provide the same message stored in it. For integrity check, the console will prompt for the user to either generate a new hash with some input message string or if the end-user already has a hash with them then they can use the same for embedding. Here, the steganographic 1 process will embed the QR code with a hash watermark. The assumption is both the sender and receiver already either know the hash value or the input string to generate a new hash for the same string. Later, the console will prompt different symmetric encryption standards for the user to choose from what algorithm they want to encrypt their water marked QR code with. Further choosing the encryption standard they will require to provide a secret key for encryption, again the assumption here is both the sender and receiver knows the secret key.

Now, we get the encrypted file, but the issue is still the same that the adversary could still decrypt the message as all the algorithms are open on internet. So, to make it more secure we will now use Steganography to hide the encrypted file inside any random image so the end-user can choose whatever meaningless image he wants to hide the encrypted file. After the entire encryption process the console will display the total time taken by the entire process to run on the system. We saved these results for further analysis of speeds for the encryption and decryption by the standards for our use-case.

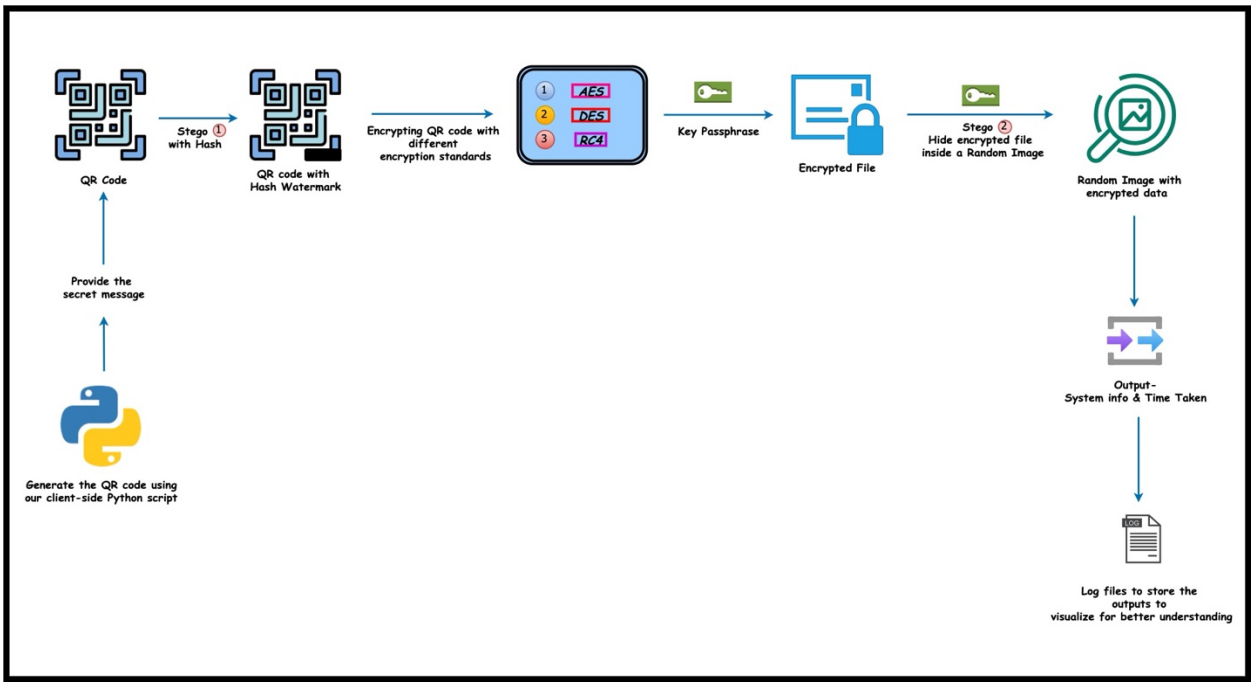


Figure 1 Client-Side Encryption Workflow

Now further we will see how to embed hash to the QR code using LSB Steganography.

3.2 Embedding Hash to QR code with LSB Steganography

There are multiple techniques to hide the data into an image using steganography but particularly we have used LSB technique. In this method, basically last few bits of a byte are used to hide the message. This method is more productive and easier to implement particularly for media files and images [10]. In an image the red, green, and blue (RGB) pixels are of one byte that sums up to eight bits. Thus, ranging from 0-255 and 00000000-11111111 decimals and binary values.

So, if we change only the last two bits in a green, red, or blue pixel from say 11111111 to 11111101 that is the values will change from 255 to 253, that will not create a significant change in the color which would be recognized by naked eyes.

Explained in figure 2 where we have shown how we stored the text WAR in the last few bits of the colors, and this would not change the original image.

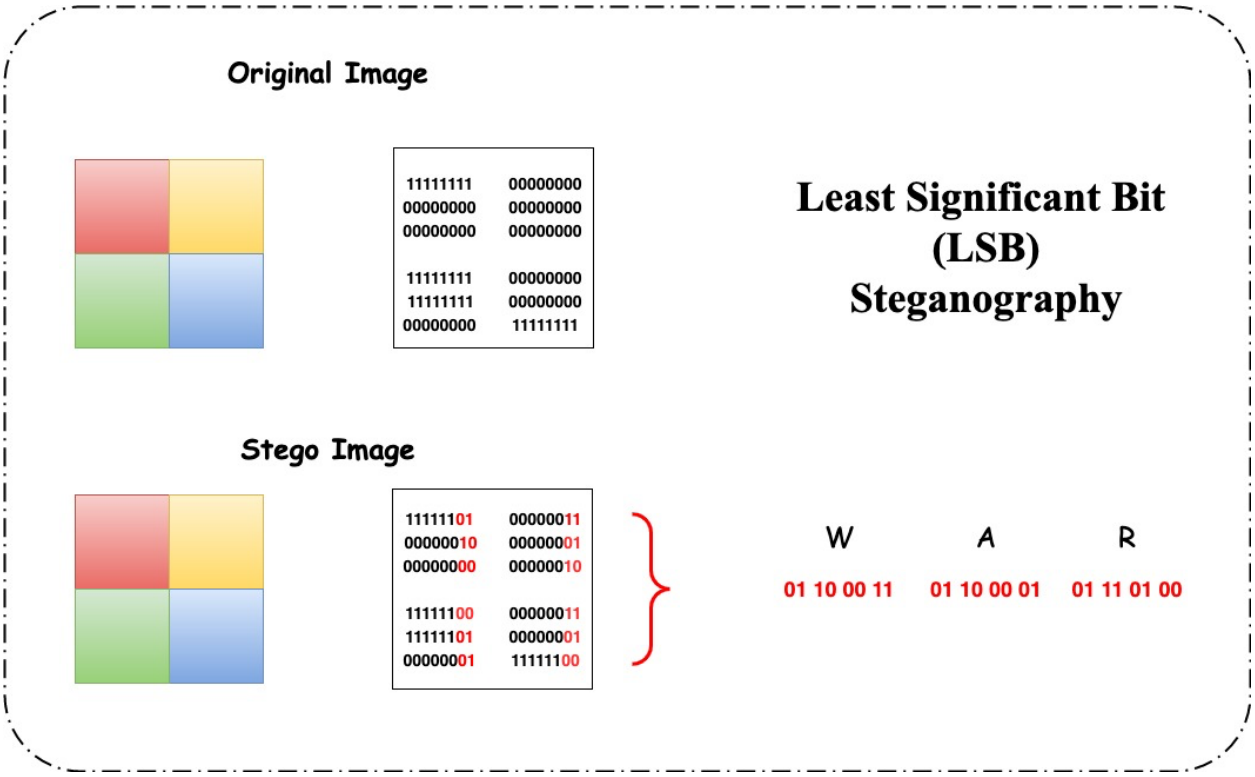


Figure 2 Least Significant Bit Steganography

Further, we will understand how to securely transfer the data from a local machine to remote or from remote-to-remote machines.

3.3 Secure Copy (SCP)

SCP basically stands for Secure Copy is as protocol for securely transferring files from local to remote or from remote-to-remote machines. As SCP happens to be based on both SSH and RCP (Remote Copy) Protocols it serves the purpose of file transfers and also provides authenticity and

confidentiality [9]. While using SCP (Secure Copy) it requires setting up password authentication for the receiver machine. The basic assumption for this approach is that the sender already knows the password for the authenticating itself to the receiver machine.

In reference to figure 3, for this implementation, we launched two AWS Kali-Linux servers in different regions, the first one in Mumbai region that is in India and the other one in Northern Virginia in USA. The client machine is the one in Mumbai region where the client-side encryption and steganographic process takes place and then the output is transferred to the Server machine here in our case at Northern Virginia on top of AWS Cloud Infrastructure.

For password authentication, we must set it up for the receiver machine here, we organized the Server machine in Northern Virginia. For this you must first change the user password and then change the sshd_config file in the machine to enable password authentication. This way if any local or remote server that want to establish a connection the receiver machine would just have to know the password for authentication.

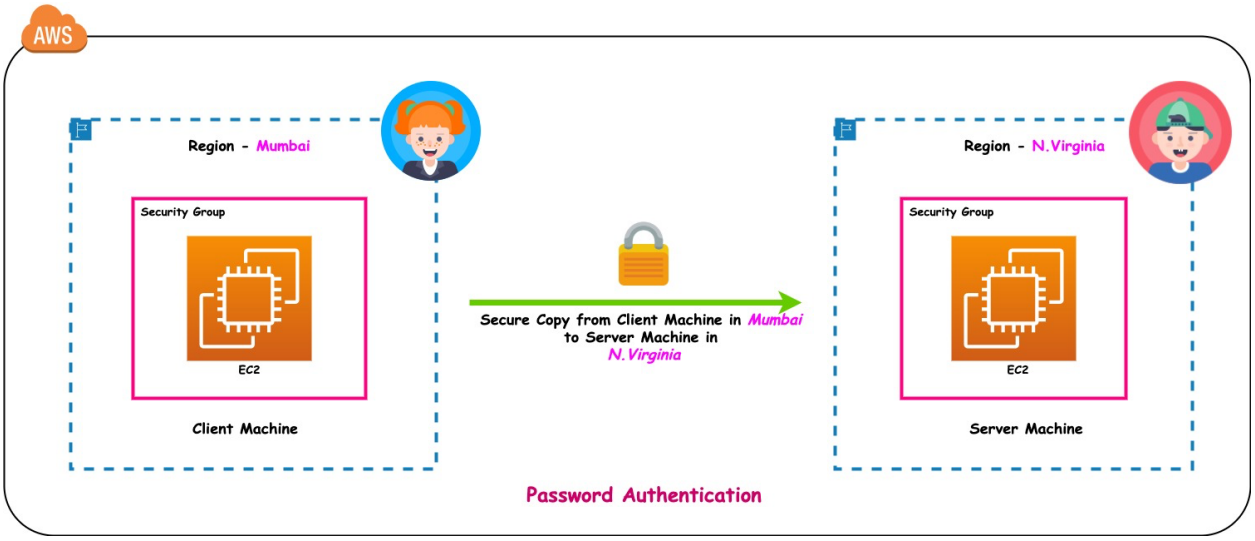


Figure 3 File Transfer with Secure Copy (SCP)

Once the data is securely transferred from the client machine to the server machine, we will see the decryption process further.

3.4 Receiver Side Decryption and Integrity Check

Here, on the receiver side once you get the Random Steganographic image with encrypted data in it from the client the process for decryption starts. The end-user will now run the python script on his machine and should follow the instructions accordingly to decrypt to original message. The script will first prompt to decode the Steganographic image to get the Original encrypted file out of it. Further, the console will prompt to choose the encryption standard to decrypt the encrypted file. Choosing the encryption standard to decrypt the console will ask for the secret key that was

used for encryption. Here, again the assumption is the receiver must know what standard which is used to encrypt the file along with the secret key.

Now we have the Original QR-code with us either we can scan the QR code directly and read the secret message stored in it or first do the integrity check, to validate whether the message has not been altered by an adversary and is the same that sender sent. For this the console with prompt two options to check integrity first whether the user already has the hash value if he does then directly can provide it and check the integrity. If he does not know the hash value but has the string on which the sender created a hash, can provide it by choosing the second option to create a new hash with the same string. This way the script will automatically compare both the hash values and if the hash values match exactly, it will display the integrity check successful message and if the hash value didn't match then will display the integrity check unsuccessful message.

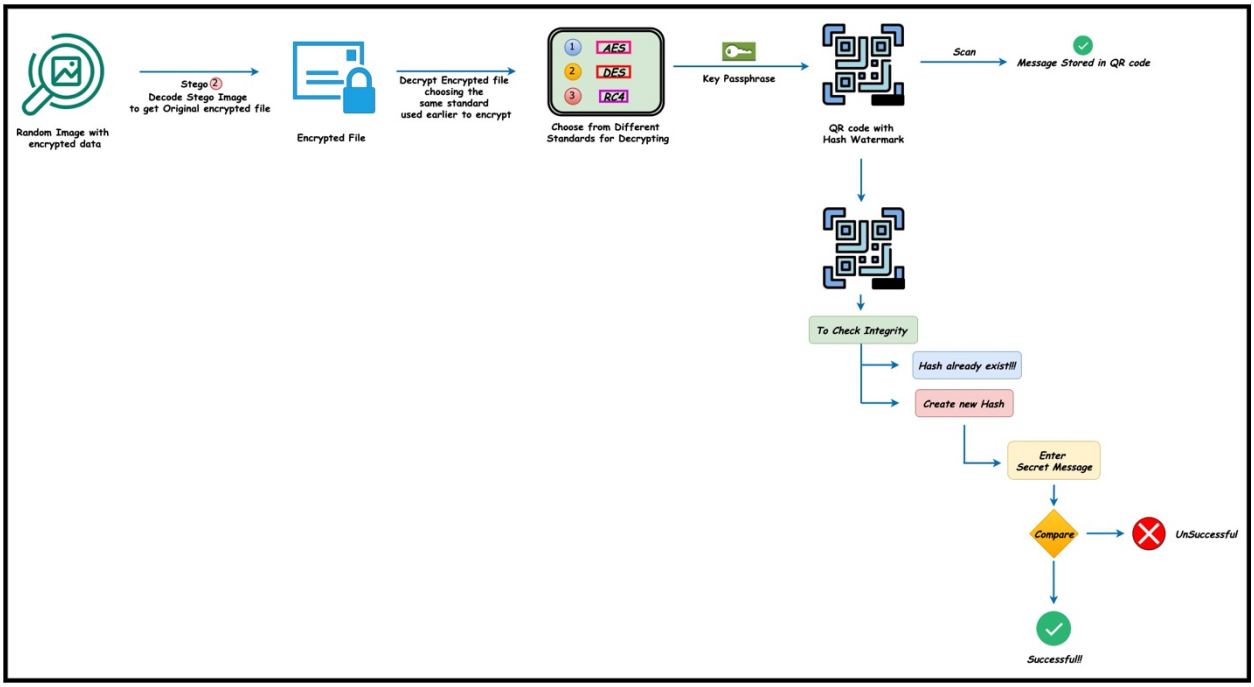


Figure 4 Receiver-Side Decryption Workflow

Thus, we were successful to validate the integrity of the message after decrypting and now we can scan the QR code and get the same message that the sender would have encoded in the QR code.

4 Implementation Details

4.1 Server Configuration on AWS EC2

For our implementation over the cloud, we are using AWS Elastic Compute Cloud (EC2) service to launch our client and server remotely in two different regions. AWS EC2 is basically a compute service using which you can launch machine with different AMI (Amazon Machine Image) that are

nothing but preconfigured machine with all the dependencies for all kinds of Operating Systems that you just have to launch and start using. For our implementation we are using Kali-Linux AMIs (Amazon Machine Images) which comes with a lot of dependencies pre-installed.

The first server that we are using as an Encryption-Client is launched in Mumbai Region. On this server the encryption process will execute and the final steganographic image with encrypted data will be generated. Then using Secure Copy (SCP) this final steganographic file will be transferred to the Decryption-Client.

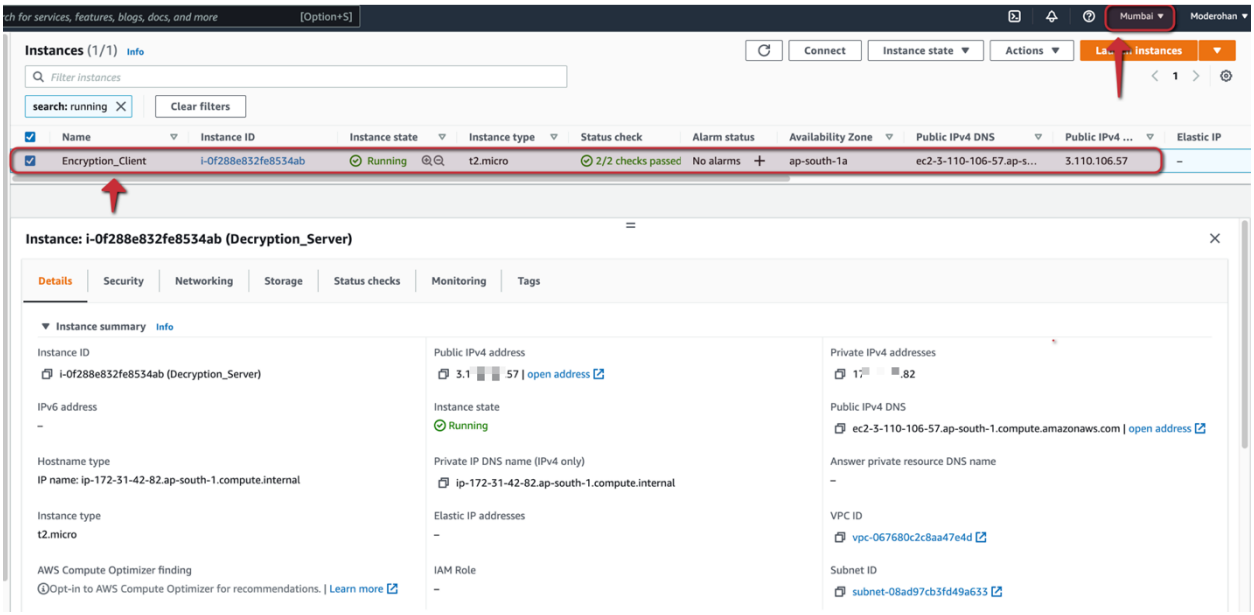


Figure 5 Encryption-Client Machine (Mumbai)

The Decryption-Client machine is basically launched in the Northern Virginia Region. This is where the decryption process will execute, and the receiver would be able to access the legitimate message send by the sender after checking the integrity and scanning the QR code.

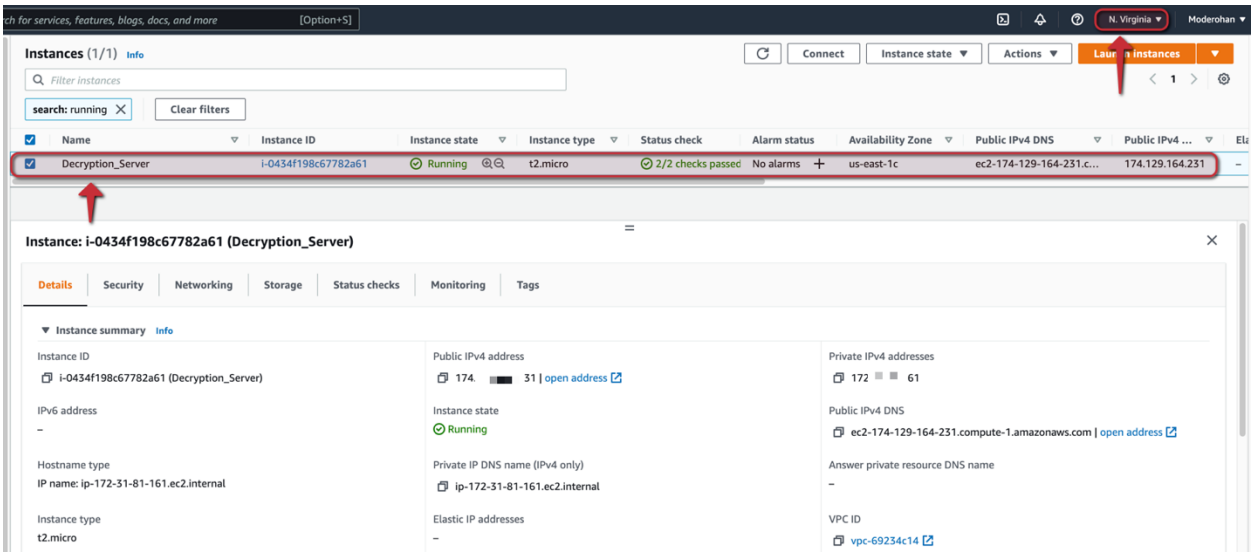


Figure 6 Decryption-Client Machine (N.Virginia)

4.2 Configuring SCP Password Authentication

For Secure Copy between two machines either local to remote or remote-to-remote we always have to configure password authentication. As SCP provides authenticity, for which the receiver machine should be configured to allow traffic only from authenticated machines. In our implementation the Northern Virginia machine is the receiver machine, so we have to first set up the password for the current user on the machine other than root user, here which is kali in our case. To update the password for the current user you must require root privilege so first switch to root privilege and then change the password for the current user.

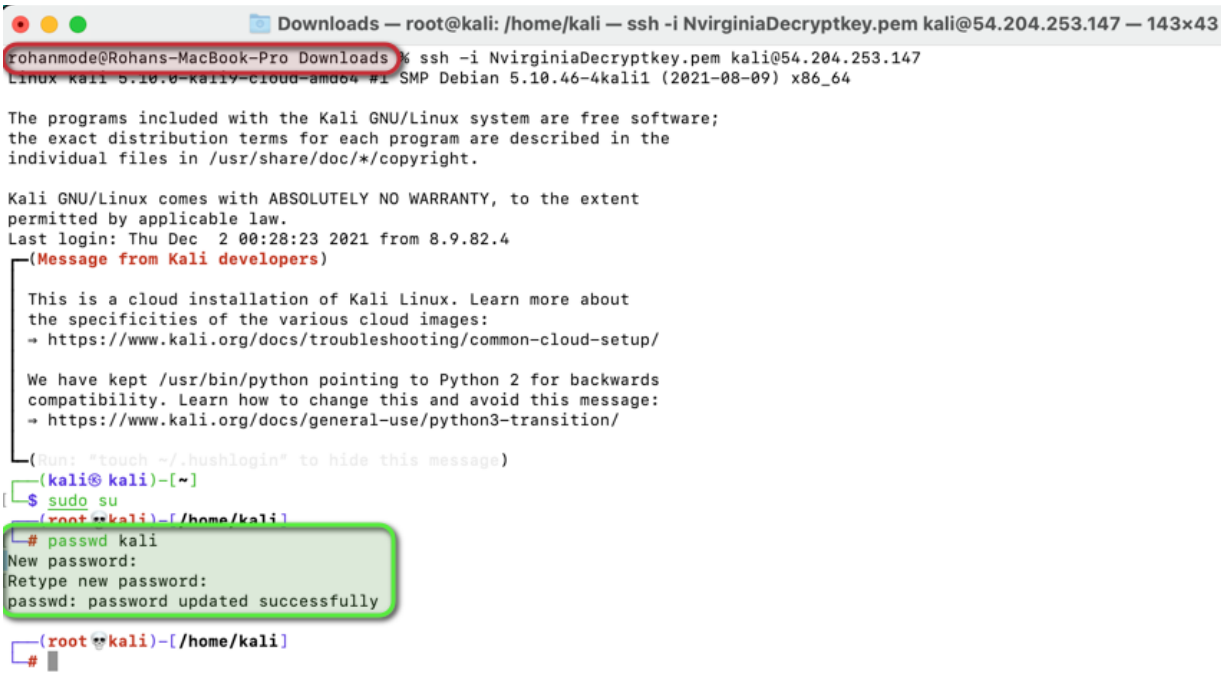


Figure 7 Configure password for receiver machine

Once the password for the current user of the receiver machine is changed, we are required to enable the password authentication on the machine from the sshd_config file [9]. The sshd_config file is basically located at /etc/ssh/sshd_config, however you can change the location for this file via -f command line functionality. The sshd_config file is nothing but the default configuration file of OpenSSH server. For any configuration changes regarding to OpenSSH you will always have to refer to sshd_config file and update it.

Here, for our use case we must enable password authentication by editing the default configuration file just by changing the password authentication instruction from no to yes as shown in figure 8. Once you have edited the sshd_config file you will have to restart the sshd service, without restarting the service the changes won't take into effect.

Figure 8 Enabling Password Authentication

Now, you can successfully authenticate any remote machine to this decryption-client machine which knows the password that was configured for the receiver machine earlier. Here, the assumption is that the sender must already be aware of the password of the receiver machine to authenticate itself to it. Once authenticated the sender can securely transfer all the files that it wants to share.

4.3 Code Organization

For implementation we have used python3 installing all the dependencies for the python on both the sender and receiver machines. The dependencies here required are ‘pypng’, ‘opencv-python’, and ‘numpy’

For QR code generation we happened to use ‘pyqrcode’ module, to save the QR code in image format like PNG or SVG we have used ‘pypng’ Here the usage of pyqrcode is very easy all we need to do is insert the content that we want to be displayed while scanning the QR code. ‘pyqrcode’ is well written which guesses all the necessary parameters depending on the input given to generate the QR code.

```
 99  def qr_code(secret_text):
100      #Generating the QR code
101      code = pyqrcode.create(secret_text)
102      #saving the PNG file.
103      code.png("secret_qr.png", scale = 6)
104  #--------------------
```

Figure 9 Generating QR code using pyqrcode

With reference to figure 9, we have defined the function qr_code, that ingest an input, further this input is used in line 101 which generates the QR code and line 103 converts it into an image file.

We have used LSB steganography for encoding hash into the QR code, the code that we used for this use case of steganography can be viewed at [8].

```
34  def encode_data(image_file_name, secret_evidence_data):
35      image = cv2.imread(image_file_name)               #we are reading the image file
36      n_of_bytes = image.shape[0] * image.shape[1]*3//8               #capacity
37      print("[+] Maximum bytes that can be encoded:", n_of_bytes)
38      if len(secret_evidence_data) > n_of_bytes:
39          raise ValueError("[!] Image Byte stream is too small, need bigger image_file or less secret_evidence_data.")
40      else:
41          print("[+] Encoding secret_evidence_data into Image_File...")
42      secret_evidence_data += "======="
43      initial_data_index = 0
44      bin_secret_evidence_data = bin_convert(secret_evidence_data)               #Converting secret_evidence_data to binary
45      secret_evidence_data_len = len(bin_secret_evidence_data)               #Size of secret_evidence_data to encode
46      for row in image:
47          for pixel in row:
48              red,green,blue = bin_convert(pixel)               #RGB to binary format
49
50              if initial_data_index < secret_evidence_data_len:
51                  pixel[0] = int(red[:-1] + bin_secret_evidence_data[initial_data_index], 2)               #red pixel as least significant bit
52                  initial_data_index += 1
53
54              if initial_data_index < secret_evidence_data_len:
55                  pixel[1] = int(green[:-1] + bin_secret_evidence_data[initial_data_index], 2)               #green pixel as least significant bit
56                  initial_data_index += 1
57
58              if initial_data_index < secret_evidence_data_len:
59                  pixel[2] = int(blue[:-1] + bin_secret_evidence_data[initial_data_index], 2)               #blue pixel as least significant bit
60                  initial_data_index += 1
61
62              if initial_data_index >= secret_evidence_data_len:
63                  break               #once data is encoded, break out.
64
65      return image
```

Figure 10 Encoding hash with LSB Steganography

We have created a function called encode_data(), this function requires two inputs, one is the image file and other is the hash value. Once the input is provided, {code_line:35} reads the image file and {code_line:36} maximum data bits that can be stored in the pixels of the image are calculated. Using function bin_convert() the hash value is converted into binary [8]. Here, we used a for-loop to run through the pixel bits of the image to replace the LSB bits [2] with the binary data of our hash value. Once all the binary data have been written, the code exits the for loop and this function returns the image with hash embedded to it.

We have used OPENSSL to encrypt and decrypt image files, it contains all the current encryption/ decryption standards used in the industry. Also, we have used a library called subprocess, which helps in executing shell commands. {code_line:112} Here, we have defined a function enc_AES(), which uses a module called “subprocess” to execute shell command. {code_line:113} AES-256-CBC is used as our encryption standard with a salt and pbkdf2 (it reduces the chances of brute-force attacks.) then in the same command we provide the input image file as: “encode_image_file.png” and output image file as: “encoded.enc”, once the command here is executed, it will ask for a pass key for AES encryption once the key is provided it will successfully encrypt the image file and generate the output. {code_line:116} similarly we have defined another function dec_AES(), which decrypts the AES encryption using the Pass key [5]. Similar Methodology and logic is used for DES [1] as well RC4 encryption and decryption process.

```
112  def enc_AES():
113      subprocess.call("openssl aes-256-cbc -a -pbkdf2 -salt -in encoded_image_file.png -out encoded.enc", shell=True)
114      print("[*] Encrypting Image File...\n[+] Encryption Success!\n")
115  #--------------------
116  def enc_DES():
117      subprocess.call("openssl des-cbc -a -pbkdf2 -in encoded_image_file.png -out encoded.enc",shell=True)
118      print("[*] Encrypting Image File...\n[+] Encryption Success!\n")
119
120  #--------------------
121  def enc_RC4():
122      subprocess.call("openssl rc4 -a -salt -pbkdf2 -in encoded_image_file.png -out encoded.enc", shell=True)
123      print("[*] Encrypting Image File...\n[+] Encryption Success!\n")
124
125  #--------------------
126  def dec_AES():
127      subprocess.call("openssl aes-256-cbc -a -pbkdf2 -salt -d -in encoded.enc -out QR_IMAGE.PNG",shell=True)
128      print("[*] Decrypting Image File...\n[+] Decryption Success!")
129
130  #--------------------
131  def dec_DES():
132      subprocess.call("openssl des-cbc -a -pbkdf2 -d -in encoded.enc -out QR_IMAGE.PNG",shell=True)
133      print("[*] Decrypting Image File...\n[+] Decryption Success!")
134
135  #--------------------
136  def dec_RC4():
137      subprocess.call("openssl rc4 -a -salt -pbkdf2 -d -in encoded.enc -out QR_IMAGE.PNG",shell=True)
138      print("[*] Decrypting Image File...\n[+] Decryption Success!")
139
```

Figure 11 Encryption and Decryption process for Standards

Steghide is a tool which is used for steganography, it hides data file as well as encrypts them with a pass key. {code_line:141} we defined a function Steghide(), which uses subprocess module for executing shell command. Steghide needs three inputs. First is the data file that we are going to hide, second file is the cover file, and the third input is the output file. {code_line:145} we defined another function Steghide_decode(), this function calls Steghide to extract the encrypted secret file, it needs two inputs first is the image file with secret_image hidden inside it, second is the output image file name.

```
141  def steghide():
142      subprocess.call("steghide embed -ef encoded.enc -cf Stego2_IMAGE.JPG -sf Final_Stego.JPG", shell=True)
143
144  #--------------------
145  def steghide_decode():
146      print("[+]: Enter Key for Steghide decode:")
147      subprocess.call("steghide extract -sf Final_Stego.JPG -xf QR_IMAGE.PNG", shell=True)
148
```

Figure 12 Encoding encrypted file using Steghide

5 Results and Analysis

5.1 Testing

5.1.1 Client-Side Encryption

To start with implementation of the code we were successful in generating a QR code with a secret message hidden in it. So now when you scan the QR code you could see the secret message stored in it refer to figure 14. Along with execution of the code to generate the QR code we also calculated the time of execution, which we can use to analysis further that you can see in figure 13.

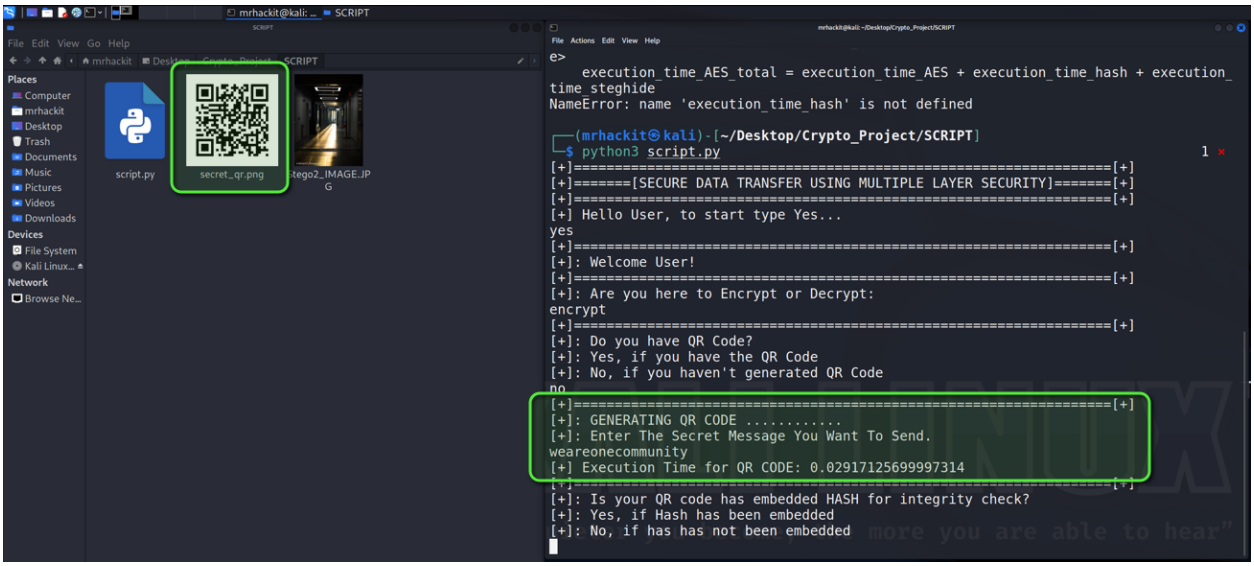


Figure 13 Generating QR code with hidden secret message

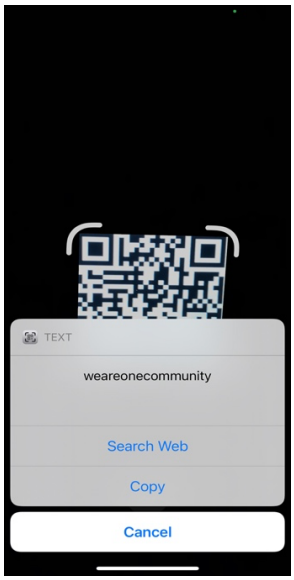


Figure 14 Retrieved Secret Message when QR code scanned

Once the QR code is generated we started testing for embedding [7] a hash value in the QR code that we will use for the sake of integrity check at the receiver side. With custom code we generated a hash value, and we successfully embedded the hash inside a random image with steganographic technique of Least Significant Bit (LSB) refer to figure 15.

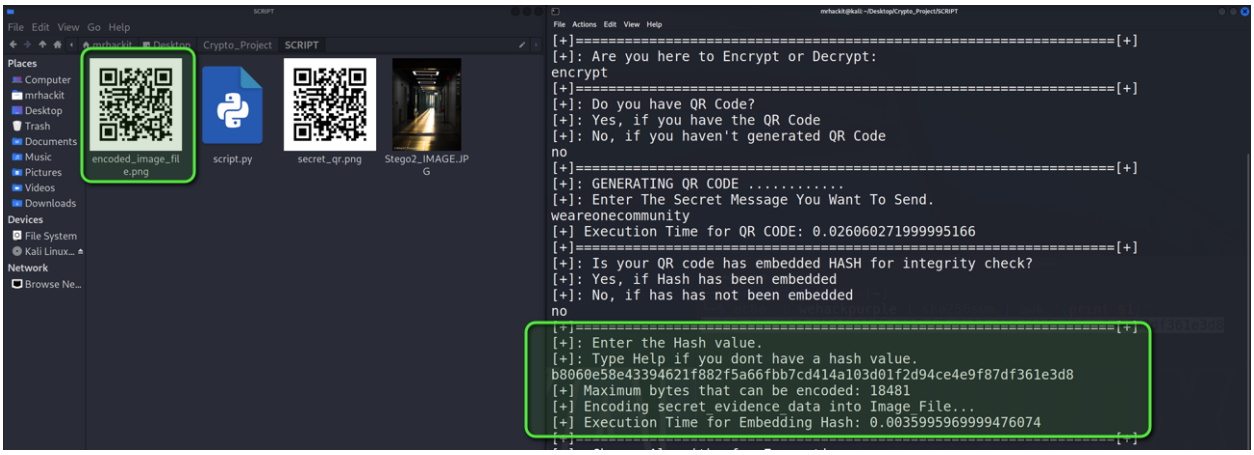


Figure 15 Embedding Hash with QR code

Now, after the hash is embedded, for the sake of multiple security we considered encryption would provide better results. So, we added symmetric algorithms for encrypting the QR code. For our use case we implemented encryption with AES, DES and RC4. Also, we focused on how all the algorithms perform for our use case. We successfully implemented providing different encryption standards for encrypting the QR code with hash for reference check figure 16.

Figure 16 Encrypting QR code with hash

Presently, we have an encrypted file containing the QR code. Even with encrypted file if shared over an unsecured channel would be a risk as any adversary could still decrypt the file with brute force as all the algorithms are out in open. So, for resolving this issue, we thought to again take help of steganographic technique but this time we hide the entire encrypted file in a random unimportant image. However, we were successful to hide the complete encrypted file inside a random image for reference check figure 17.

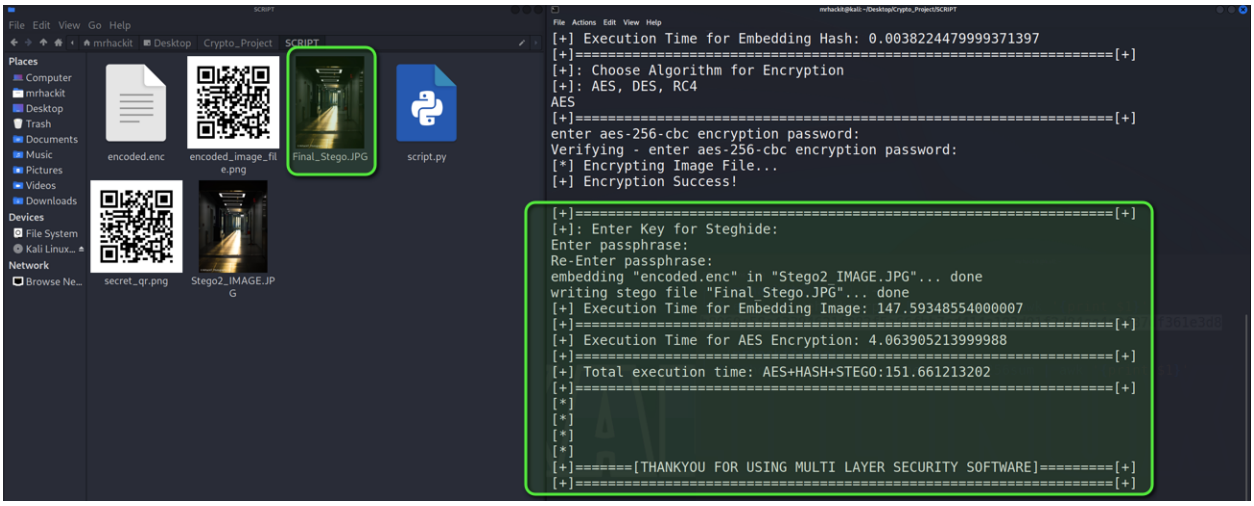


Figure 17 Embedding the Encrypted File in a Random Image

5.1.2 Secure transmission of the Final Steganographic Image over the Cloud.

For this, we configured both of our server/machines on AWS EC2 in two different regions. The encryption process was performed on the machine in Mumbai, the output final steganographic image needs to be securely transmitted to the receiver machine that resides in Northern Virginia. We succeeded in configuring password authentication on the receiver machine for Secure Copy (SCP) refer to figure 8. Further, using the command for scp we transferred the final steganographic image over the channel to receiver machine successfully refer to figure 18, so that the decryption process can now initiate on the other side.

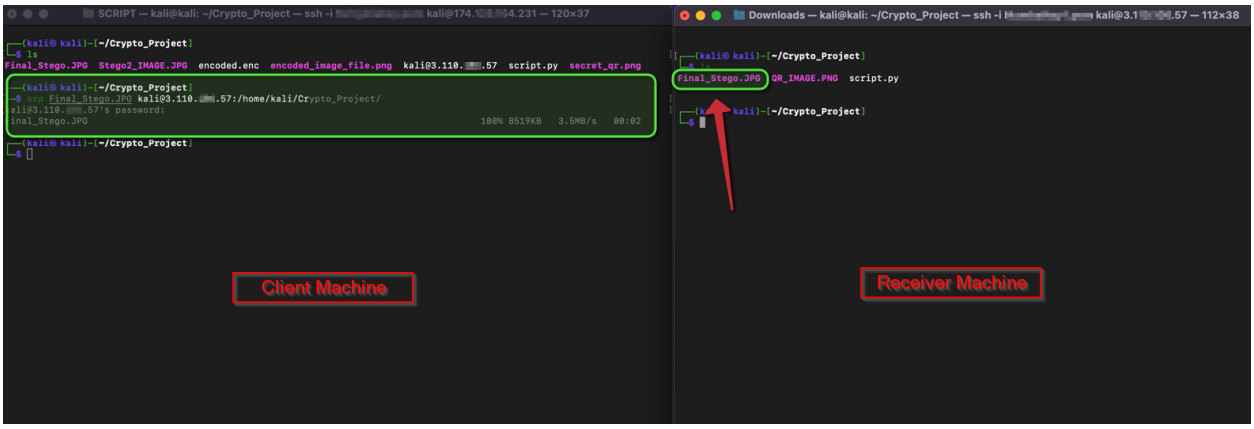


Figure 18 Secure transfer of file via SCP

Thus, we were successful to securely transmit the data in practical between the servers/machines. Further we need to decode and implement the decryption process on the receiver machine to get the original message with integrity check.

5.1.3 Decryption Process with Integrity check

Once we have received the final steganographic image on the receiver machine, the receiver will now run our python script to decode the steganographic image at first using steghide, here you will have to provide the same passphrase that the sender used to hide, the assumption is both

the sender and receiver should know the passphrase to encode and decode. Now you will receive the QR image which you cannot open because it is still encrypted.

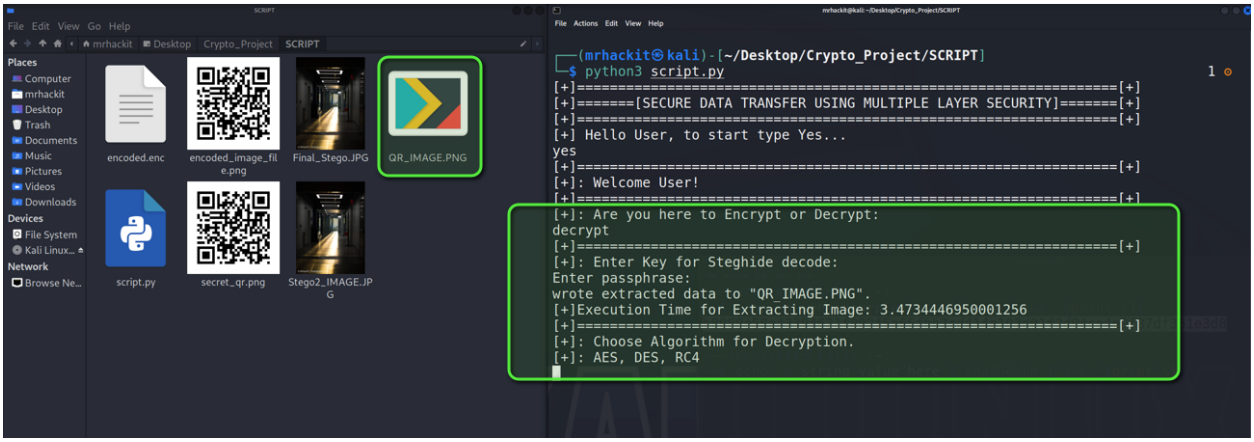


Figure 19 Decode the Received Steganographic image with Steghide

Now, choose the same algorithm to decrypt the encrypted file with which it was encrypted by the sender. The assumption here is the receiver should be aware of the algorithm by which the data was encrypted. We successfully decrypted the encrypted image and extracted the QR code from it.

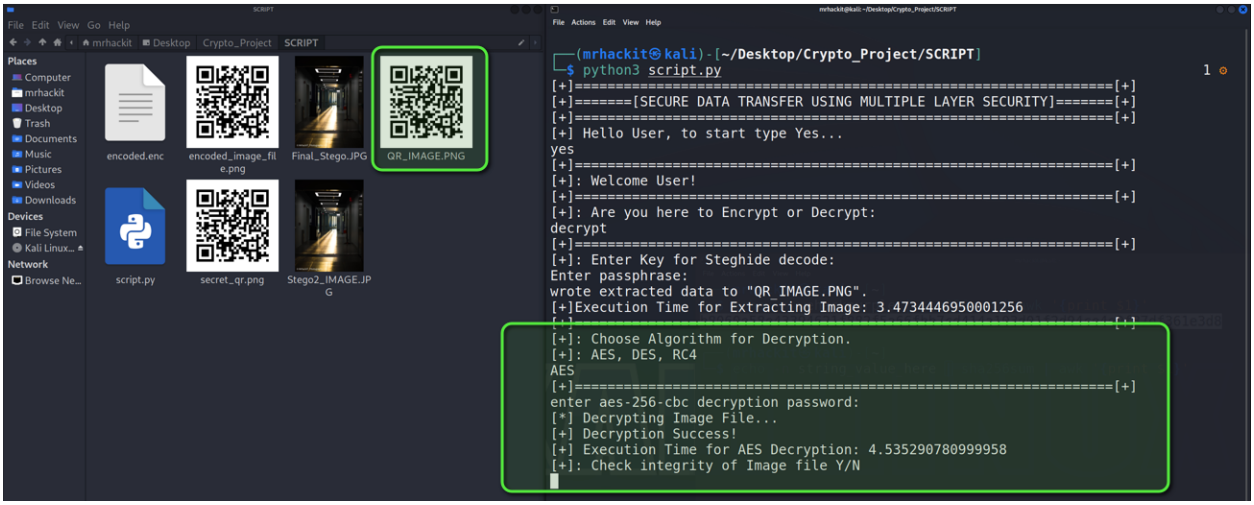


Figure 20 Decrypt the encrypted QR code with Decryption Algorithms

Now you can directly scan the QR code and get the secret message, but the issue here is how the receiver will know if the information store in QR code was not altered and it's the same. For this we have water marked the QR code with hash in the first step. Now the console will prompt whether to check the integrity, for this either you already have the hash or the message string to generate a new hash for the same. Once you provide the hash value it will compare with the water mark if it remains exact same then the console will display Integrity check successful message and if not identical will display integrity check unsuccessful.

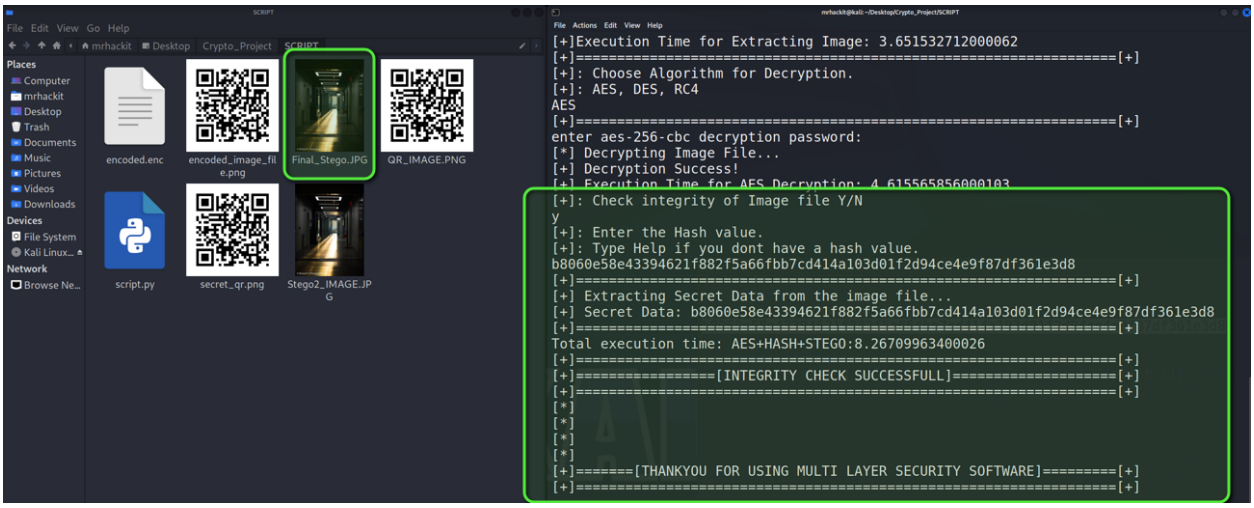
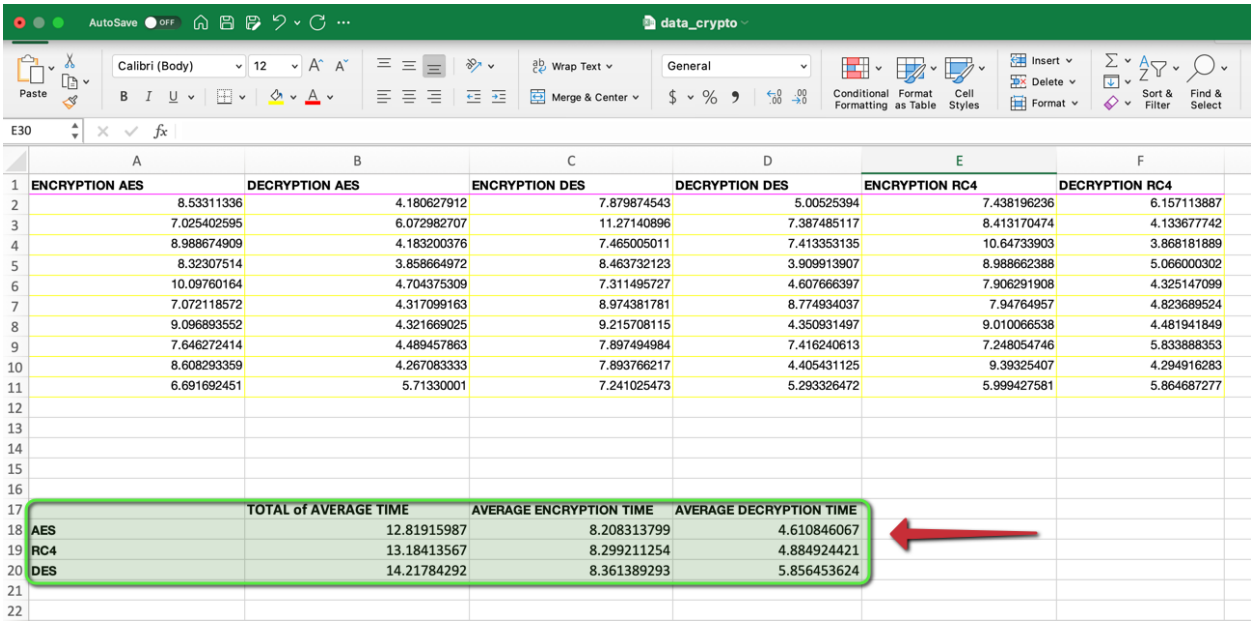


Figure 21 Integrity Check for the Received QR code

5.2 Analysis

The main aim of our entire implementation was to calculate and compare the time taken by each of the process and the algorithm for our use case. For this we executed the code multiple times and make track of each value in a excel sheet refer to figure 22. We executed each time with a new image for steganographic implementation, also the hash value. We followed a pattern that with one image and one hash we executed the entire code for all the algorithm and then calculated the time so that all the algorithms for encryption and decryption would have same values to execute. This way we would be able to get exact execution time for each of the algorithms on identical parameters.



ENCRYPTION AES	DECRYPTION AES	ENCRYPTION DES	DECRYPTION DES	ENCRYPTION RC4	DECRYPTION RC4
8.53311336	4.180627912	7.879874543	5.00525394	7.438196236	6.157113887
7.025402595	6.072982707	11.27140896	7.387485117	8.413170474	4.133677742
8.988674909	4.183200376	7.465005011	7.413353135	10.64733903	3.868181889
8.32307514	3.858664972	8.463732123	3.909913907	8.988662388	5.066000302
10.09760164	4.704375309	7.311495727	4.607666397	7.906291908	4.325147099
7.072118572	4.317099163	8.974381781	8.774934037	7.94764957	4.823689524
9.096893552	4.321669025	9.215708115	4.350931497	9.010066538	4.481941849
7.646272414	4.489457863	7.897494984	7.416240613	7.248054746	5.833888353
8.608293359	4.267083333	7.893766217	4.405431125	9.39325407	4.294916283
6.691692451	5.71330001	7.241025473	5.293326472	5.999427581	5.864687277

	TOTAL of AVERAGE TIME	AVERAGE ENCRYPTION TIME	AVERAGE DECRYPTION TIME
AES	12.81915987	8.208313799	4.610846067
RC4	13.18413567	8.299211254	4.884924421
DES	14.21784292	8.361389293	5.856453624

Figure 22 Time taken for execution by each Algorithm

In reference to figure 23, for the data that we documented in the excel sheet, we performed some mathematical operations like took the Total Average time of execution based on all the values

that we mentioned in the excel sheet. Also, we calculated Average Encryption time and Average Decryption time for all the algorithms. To have more visibility for the data that we documented we visualized it in terms of performance comparison charts.

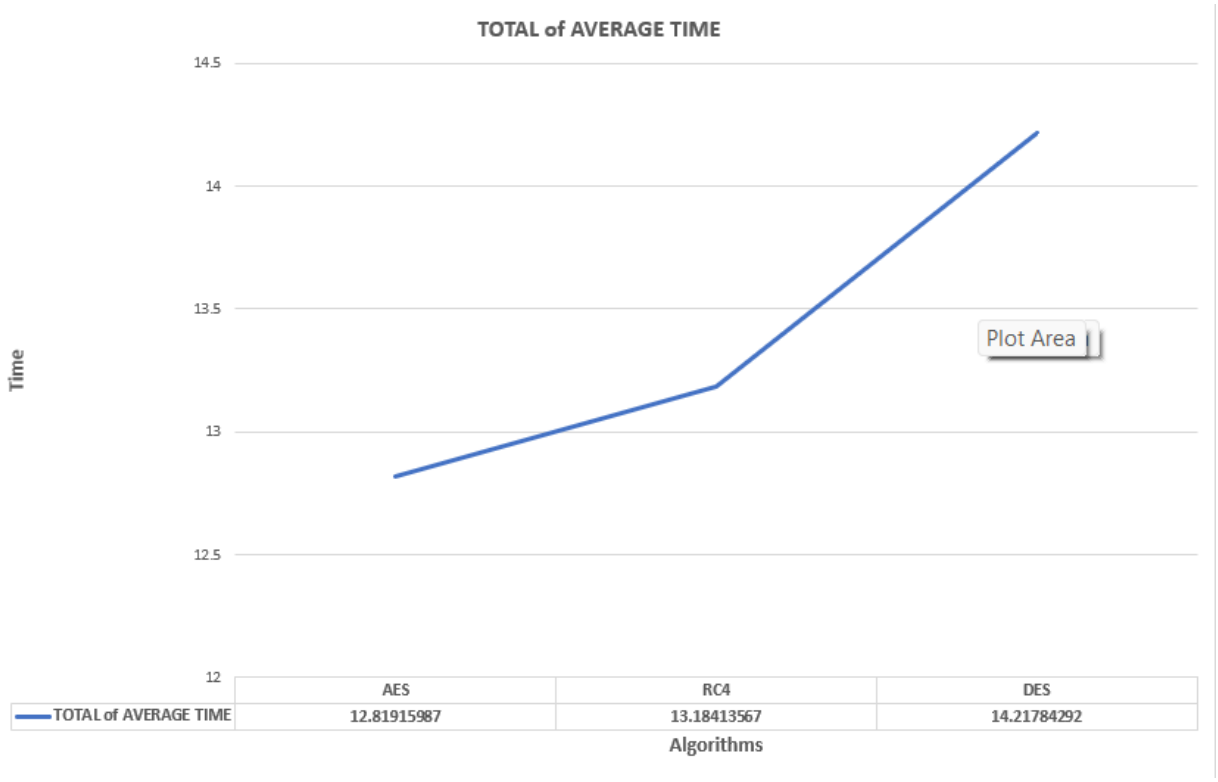


Figure 23 Total Average Time for Execution

Now for Average encryption and Average decryption time taken by all the algorithms we generated a performance comparison Bar chart. This made it clearer that the total average encryption time for all the three algorithm was almost close to each other. However, the change in the analysis happened based on the decryption times. The decryption time for DES remained the most as compared to other algorithms.

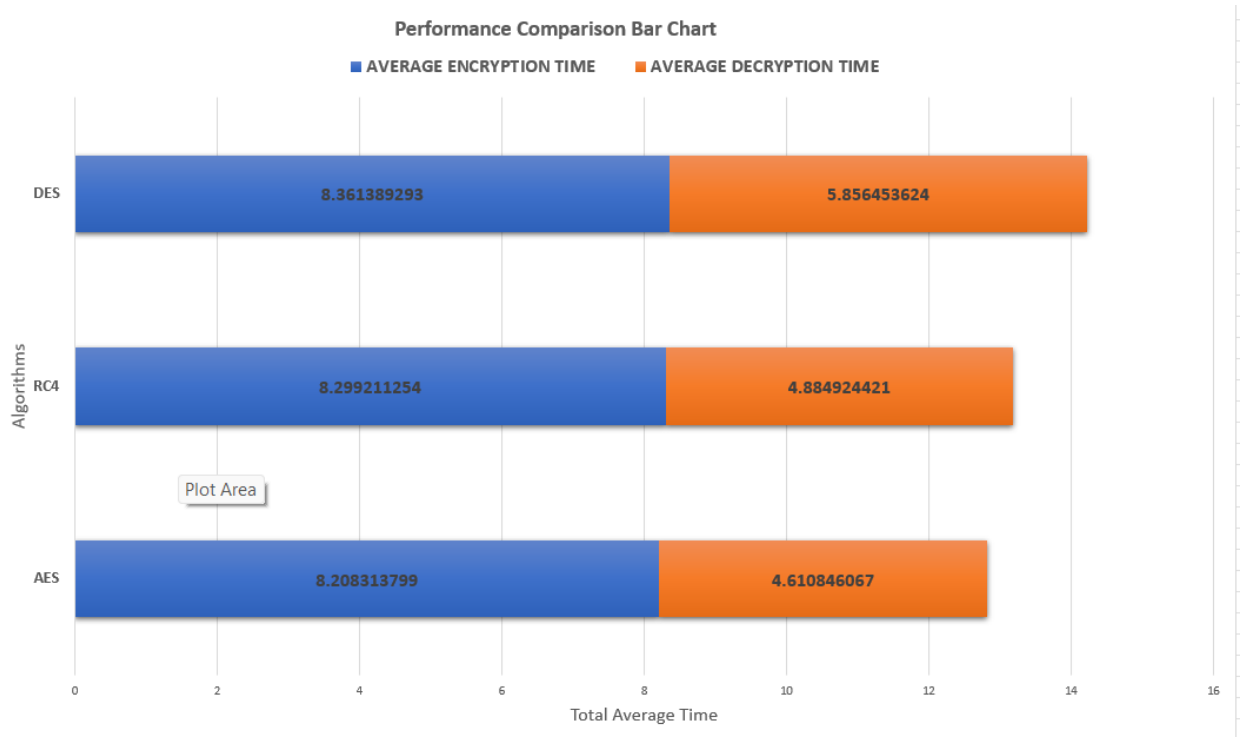


Figure 24 Performance Bar chart for Average Encryption and Decryption

For the final analysis, based on the values and performance comparison charts we found that the encryption time for each of the algorithm was close to each other but the variation in the results came with the decryption times and values. DES required slightly higher time than that of RC4 and AES to decrypt refer to figure 25.

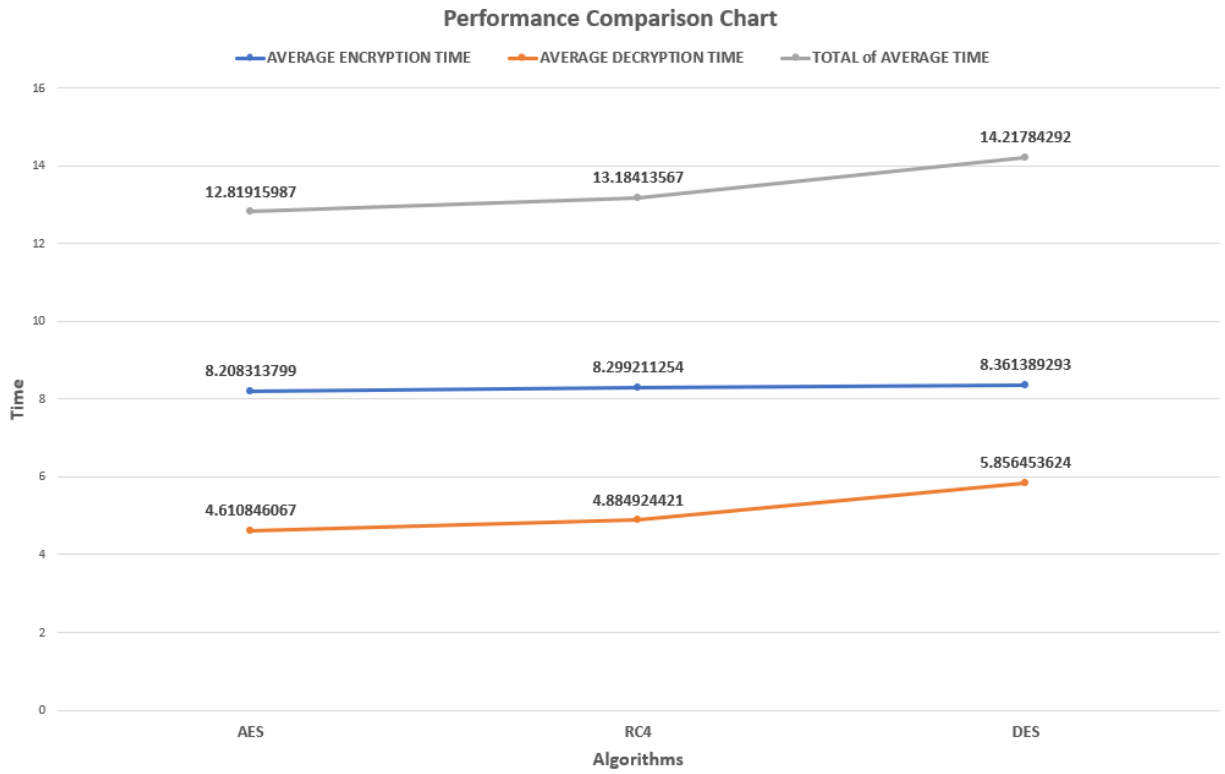


Figure 25 Performance Comparison chart for Complete Values

6 Conclusions and Future Work

Based on the implementation and the analysis report using the following methods we established multiple layers of security to transmit data over the cloud. From the analysis report it is clear that for real world use case, AES is the fastest followed by RC4 and DES as slowest in that order. The execution of multi-layer security with different algorithms was relatively simple, however implementing entire secure architecture was difficult. Initially before writing the code, we had to draw, multiple architecture and flow diagrams for how exactly we are going to perform our use case and how we shall compare the performance based on execution time and parameters.

For our implementation, we focused on speed of the execution which is why we implemented only for symmetric key algorithms. However, for better security we could have used Asymmetric key algorithms and tested for the speed and accuracy of encryption and decryption process for our use case. Asymmetric key execution being more secure would-be time consuming as it involves complex execution and calculations for encryption and decryption.

Also, for our implementation, we used only one hashing algorithm that is SHA-256. For more clearer view we could have used multiple hash algorithms like MD5, SHA-1, SHA-3, etc., and then analyzed the results for timely execution and accuracy for the same QR code use case that we have implemented earlier.

The other security measure that we could have implemented was to improve the secure transfer of the data between the machines by restricting the traffic coming in and out of the servers/machines. For implementing this, we could have used security group feature, which is nothing but a semi-permeable membrane or say, a firewall, which allows users to set rules for inbound and outbound traffic. For improving the security, we could have created a security group rule for the receiver machine that all the traffic that is allowed in, must only be from the security group of the sender machine. This way, no other machine can talk to the receiver machine other than our sender machine residing in Mumbai.

7 References

[1] R. Shivhare, R. Shrivastava and C. Gupta, "An Enhanced Image Encryption Technique using DES Algorithm with Random Image overlapping and Random key Generation," *IEEE,* 2019.

[2] A. Moumen and H. Sissaoui, "Images Encryption Method using Steganographic LSB Method, AES and RSA algorithm," *Nonlinear Engineering,* 2016.

[3] R. U. Ginting and R. Y. Dillak, "Digital color image encryption using RC4 stream cipher and chaotic logistic map," *IEEE,* 2013.

[4] P.-Y. Lin and Y.-H. Chen, "QR code steganography with secret payload enhancement," *IEEE,* 2016.

[5] R. B. K. S. M. a. N. R. Nooka Saikumar, "An Encryption Approach for Security Enhancement in Images using Key Based Partitioning Technique," *International Conference on Circuit Power and Computing Technologies [ICCPCT],* 2016.

[6] A. C. K. a. S. S. G. B Karthikeyan, "Enhanced Security in Steganography using Encryption and Quick Response code," *International Conference on Wireless Communications Signal Processing and Networking (WiSPNET),* 2016.

[7] X. P. Z. a. S. Z. Wang, "Efficient steganographic embedding by exploiting modification direction," *IEEE,* 2006.

[8] A. Rockikz, "How to Use Steganography to Hide Secret Dara in Images in Python - Python Code," 2021. [Online]. Available: https://www.thepythoncode.com/article/hide-secret-data-in-images-using-steganography-python.

[9] H. James, "SCP Linux – Securely Copy Files Using SCP examples," [Online]. Available: https://haydenjames.io/linux-securely-copy-files-using-scp/.

[10] B. Slash, "How to Hide Secret Data Inside an Image or Audio File in Seconds," [Online]. Available: https://null-byte.wonderhowto.com/how-to/steganography-hide-secret-data-inside-image-audio-file-seconds-0180936/.