UNIVERSITY OF SCIENCE AND TECHNOLOGY AT ZEWAIL CITY

COMMUNICATION AND INFORMATION ENGINEERING

Information Security

CIE 581

AES Encryption

Mohamed Mostafa 201600236

Contents

List of Figures

1 Introduction

An explanation of how the code is implemented can be found in the comments within the code itself. The user only needs to access three function to work with. First being *encrypt_img* which allows the user to input the image, a key and select the mode to work. The function is high level enough to allow non specialized users to work with the function easily. This is also paired with the *decrypt_img* function as well. Secondly and thirdly are *encrypt_cbc* and *encrypt_ecb* which needs only the input data in the form of a hex string or binary string to work along with a key to work although in the CBC case an IV is needed as well. They are also paired with their respective decrypt functions. The code allows the user to input data that does not complete a state as padding is implemented. The user can also view the image from the *encrypt_img* function by setting showImage to true. The code also allows the user to view the output of each layer from the encryption rounds that are applied to each state.

2 Problems Faced

The algorithm itself is pretty straightforward and after finally finishing it most of the problems faced were not in the algorithm itself but instead in technical diffucilties due to our implementation

2.1 Efficiency

The code at first was not optimized to handle large amounts of data. Several actions has been taken towards solving this.

After profiling the code the bottleneck was found to be at the mixColumns where we found that it took nearly 98% of the computation time.Thus,Instead of computing multiplications of mixColumns on the fly a dictionary was implemented to remove the need of multiplication which greatly improved the speed of the function and reducing the percentage of time taken by the function down to 50%

This dictionary approach is also implemented in the sBox function.

We also found that it is faster to open these dictionaries in a high level function and then passing them as arguments instead of a low level one which would be accessed a lot of times.

Another improvement that we implemented was that the mixColumns dictionary returns an int after being indexed to reduce the number of conversions.

We initially used The polynomial library in the first phase but after the above dictionary implementations we found no use for it so the code was refactored to deal with binary string instead of polynomial arrays. This change also increased the speed of the code.

Further improvements in the future could be parallelizing the code of the ECB function since it does not depend on previous values. An approach to this was already undertaken in this project by trying to use library numba, but some of the functions used by our implementation are not supported by it according to their github repo responses and did not work so another way of threading could be viewed in the future.

2.2 Technical diffuclties

After finishing the project, it is not recommended to implement AES on python a python for, due to that the language itself is high level and cannot compute a lot of operations with the speed of c++ of rust for example.

3 RunTime comparison

The 1Mb file is used between for comparison between the modes and key sizes but as the 10MB takes a lot of time (11mins), so the time for the other modes could be infered by comparison with the 1Mb runtime.

```
"""
AES-128 CBC on 1MB File
"""


with open("1MB", mode='rb') as file: # b is important -> binary
    fileContent = file.read().hex()
initialKey="00cc73c990d376b82246e45ea3ae2e37"
IV="e79026639d4aa230b5ccffb0b29d79bc"
cProfile.run('encfileContent=AES.encrypt_cbc(fileContent,initialKey,IV)',sort="tottime")
outputFile=open('1MBoutput_128_CBC', 'wb')
outputFile.write(bytearray.fromhex(encfileContent))
outputFile.close()
file.close()
         34420893 function calls (34420397 primitive calls) in 60.613 seconds
```

Figure 1: Runtime for encrypting 1MB with AES128 CBC

```
"""
AES-192 CBC on 1MB File
"""


with open("1MB", mode='rb') as file: # b is important -> binary
    fileContent = file.read().hex()
initialKey="80da652b1844dafe4fd4ca8ccc26b564b263711723b6cd48"
IV="e79026639d4aa230b5ccffb0b29d79bc"
cProfile.run('encfileContent=AES.encrypt_cbc(fileContent,initialKey,IV)',sort="tottime")
outputFile=open('1MBoutput_192_CBC', 'wb')
outputFile.write(bytearray.fromhex(encfileContent))
outputFile.close()
file.close()
         41630363 function calls (41629851 primitive calls) in 73.027 seconds
```

Figure 2: Runtime for encrypting 1MB with AES192 CBC

```
"""
AES-256 CBC on 1MB File
"""

with open("1MB", mode='rb') as file: # b is important -> binary
    fileContent = file.read().hex()
initialKey="2e39c585ce4900d323ce29713bebe73a1be08a0cb22e9f1310fcc14ad4b9b23e"
IV="e79026639d4aa230b5ccffb0b29d79bc"
cProfile.run('encfileContent=AES.encrypt_cbc(fileContent,initialKey,IV)',sort="tottime")
outputFile=open('1MBoutput_256_CBC', 'wb')
outputFile.write(bytearray.fromhex(encfileContent))
outputFile.close()
file.close()
         48840595 function calls (48840039 primitive calls) in 86.943 seconds
```

Figure 3: Runtime for encrypting 1MB with AES256 CBC

```
"""
AES-256 CBC on 10MB File
"""

with open("10MB", mode='rb') as file: # b is important -> binary
    fileContent = file.read().hex()
initialKey="2e39c585ce4900d323ce29713bebe73a1be08a0cb22e9f1310fcc14ad4b9b23e"
IV="e79026639d4aa230b5ccffb0b29d79bc"
cProfile.run('encfileContent=AES.encrypt_cbc(fileContent,initialKey,IV)',sort="tottime")
outputFile=open('10MBoutput_256_CBC', 'wb')
outputFile.write(bytearray.fromhex(encfileContent))
outputFile.close()
file.close()
         488259475 function calls (488258919 primitive calls) in 842.412 seconds
```

Figure 4: Runtime for encrypting 10MB with AES256 CBC

```
"""
AES-128 ECB on 1MB File
"""

with open("1MB", mode='rb') as file: # b is important -> binary
    fileContent = file.read().hex()
initialKey="00cc73c990d376b82246e45ea3ae2e37"
cProfile.run('encfileContent=AES.encrypt_ecb(fileContent,initialKey)',sort="tottime")
outputFile=open('1MBoutput_128_ECB', 'wb')
outputFile.write(bytearray.fromhex(encfileContent))
outputFile.close()
file.close()
```

```
         34158747 function calls (34158251 primitive calls) in 61.146 seconds
```

Figure 5: Runtime for encrypting 1MB with AES128 ECB

```
"""
AES-192 ECB on 1MB File
"""

with open("1MB", mode='rb') as file: # b is important -> binary
    fileContent = file.read().hex()
initialKey="80da652b1844dafe4fd4ca8ccc26b564b263711723b6cd48"
cProfile.run('encfileContent=AES.encrypt_ecb(fileContent,initialKey)',sort="tottime")
outputFile=open('1MBoutput_192_ECB', 'wb')
outputFile.write(bytearray.fromhex(encfileContent))
outputFile.close()
file.close()
```

```
         41368217 function calls (41367705 primitive calls) in 70.080 seconds
```

Figure 6: Runtime for encrypting 1MB with AES192 ECB

```python
"""
AES-256 ECB on 1MB File
"""

with open("1MB", mode='rb') as file: # b is important -> binary
    fileContent = file.read().hex()
initialKey="2e39c585ce4900d323ce29713bebe73a1be08a0cb22e9f1310fcc14ad4b9b23e"
cProfile.run('encfileContent=AES.encrypt_ecb(fileContent,initialKey)',sort="tottime")
outputFile=open('1MBoutput_256_ECB', 'wb')
outputFile.write(bytearray.fromhex(encfileContent))
outputFile.close()
file.close()
```

```
48578449 function calls (48577893 primitive calls) in 85.870 seconds
```

Figure 7: Runtime for encrypting 1MB with AES256 ECB

The conclusion from these times is that the CBC takes marginally more time as it is a little bit more complex. The Larger the key size, the longer the computation drastically. It can be infered that the implementation has a linear complexity by comparison between 1MB and 10MB. when using openssl on these files the runtime is much much lower scoring only 0.2 secs for the whole 100MB file and 0.033 secs for the 10MB one.

4 Verification

4.1 Encryption Decryption using AES – 128

4.1.1 CBC

```
#testcase for AES128 CBC
inputData="58c8e00b2631686d54eab84b91f0aca1"
initialKey="00000000000000000000000000000000"
IV="00000000000000000000000000000000"
cipherText="08a4e2efec8a8e3312ca7460b9040bbf"
cipher=AES.encrypt_cbc(inputData,initialKey,IV)
print(f"Correct Encryption:{cipher[:-2]==cipherText}")
print(f"Correct Decryption:{AES.decrypt_cbc(cipher,initialKey,IV)==inputData}")

Correct Encryption:True
Correct Decryption:True
```

Figure 8: Verification for AES128 CBC 1

```
#testcase for AES128 CBC
inputData="807bc4ea684eedcfdcca30180680b0f1ae2814f35f36d053c5aea6595a386c1442770f4d7297d8b91825ee7237241da8925dd594ccf676aecd46ca2068e8d37a3a0ec8a7d5185a201e663b5ff36ae19
initialKey="89a553730433f7e6d67d16d373bd5360"
IV="f724558db3433a523f4e51a5bea70497"
cipherText="406af1429a478c3d07e555c5287a60500d37fc39b68e5bbb9bafd6ddb223828561d6171a308d5b1a4551e8a5e7d572918d25c968d3871848d2f16635caa9847f38590b1df58ab5efb985f2c66cfaf8
cipher=AES.encrypt_cbc(inputData,initialKey,IV)
print(f"Correct Encryption:{cipher[:-2]==cipherText}")
print(f"Correct Decryption:{AES.decrypt_cbc(cipher,initialKey,IV)==inputData}")

Correct Encryption:True
Correct Decryption:True
```

Figure 9: Verification for AES128 CBC 2

4.2 ECB

```
#testcase for AES128 ECB
inputData="58c8e00b2631686d54eab84b91f0aca1"
initialKey="00000000000000000000000000000000"
cipherText="08a4e2efec8a8e3312ca7460b9040bbf"
cipher=AES.encrypt_ecb(inputData,initialKey)
print(f"Correct Encryption:{cipher[:-2]==cipherText}")
print(f"Correct Decryption:{AES.decrypt_ecb(cipher,initialKey)==inputData}")

Correct Encryption:True
Correct Decryption:True
```

Figure 10: Verification for AES128 ECB 1

```
#testcase for AES128 ECB
inputData="37a1205ea929355d2e4ee52d5e1d9cda279ae01e640287ccb153276e7e0ecf2d633cf4f2b3afaecb548a2590ce0445c6a168bac3dc601813eb74591bb1ce8dfcd740cdbb6388719e8cd283d9cc7e736
initialKey="00cc73c990d376b82246e45ea3ae2e37"
cipherText="c88e03383ba9da6f982c057fe92c0bb3ed5b9cd18295a100e13a4e12d440b919bbb8b221abead362902ce44d30d0b80e56bee1f66a7d8de0b1e1b4dbf76c90c1807a3bc5f277e9814c82ab120f7e10
cipher=AES.encrypt_ecb(inputData,initialKey)
print(f"Correct Encryption:{cipher[:-2]==cipherText}")
print(f"Correct Decryption:{AES.decrypt_ecb(cipher,initialKey)==inputData}")

Correct Encryption:True
Correct Decryption:True
```

Figure 11: Verification for AES128 ECB 2

4.3 Encryption Decryption using AES – 192

4.3.1 CBC

```
#testcase for AES192 CBC
inputData="1b077a6af4b7f98229de786d7516b639"
initialKey="000000000000000000000000000000000000000000000000"
IV="00000000000000000000000000000000"
cipherText="275cfc0413d8ccb70513c3859b1d0f72"
cipher=AES.encrypt_cbc(inputData,initialKey,IV)
print(f"Correct Encryption:{cipher[:-2]==cipherText}")
print(f"Correct Decryption:{AES.decrypt_cbc(cipher,initialKey,IV)==inputData}")

Correct Encryption:True
Correct Decryption:True
```

Figure 12: Verification for AES192 CBC 1

```
#testcase for AES192 CBC
inputData="6abcc270173cf114d44847e911a050db57ba7a2e2c161c6f37ccb6aaa4677bddcaf50cad0b5f8758fcf7c0ebc650ceb5cd52cafb8f8dd3edcece55d9f1f08b9fa8f54365cf56e28b9596a7e1dd1d341
initialKey="f9c27565eb07947c8cb51b79248430f7b1066c3d2fdc3d13"
IV="2bd67cc89ab7948d644a49672843cbd9"
cipherText="ca282924a61187feb40520979106e5cc861957f23828dcb7285e0eaac8a0ca2a6b60503d63d6039f4693dba32fa1f73ae2e709ca94911f28a5edd1f30eaddd54680c43acc9c74cd90d8bb648b4e544
cipher=AES.encrypt_cbc(inputData,initialKey,IV)
print(f"Correct Encryption:{cipher[:-2]==cipherText}")
print(f"Correct Decryption:{AES.decrypt_cbc(cipher,initialKey,IV)==inputData}")

Correct Encryption:True
Correct Decryption:True
```

Figure 13: Verification for AES192 CBC 2

4.3.2 ECB

```
#testcase for AES192 ECB
inputData="1b077a6af4b7f98229de786d7516b639"
initialKey="000000000000000000000000000000000000000000000000"
cipherText="275cfc0413d8ccb70513c3859b1d0f72"
cipher=AES.encrypt_ecb(inputData,initialKey)
print(f"Correct Encryption:{cipher[:-2]==cipherText}")
print(f"Correct Decryption:{AES.decrypt_ecb(cipher,initialKey)==inputData}")

Correct Encryption:True
Correct Decryption:True
```

Figure 14: Verification for AES192 ECB 1

```
#testcase for AES192 ECB
inputData="1e49e9541ed0f97023f8205e71fe75668680f12fb5902d7a11066d108c1adef960321cefaef8f71365b077de66c91e59e6b16c9113eea945fa6bdee3a80f25b38f9422512c97d260e7eb837d3b33247
initialKey="80da652b1844dafe4fd4ca8ccc26b564b263711723b6cd48"
cipherText="a47e7d0d8266595792bc980bf3c47822229e4428cc91aa1b15519823a133dd479a4664ec8408b4d9301760454ed2037279d996eb1ccc98df77660aa97422dca1c54d7bfac37223d0caee6a6b34d760
cipher=AES.encrypt_ecb(inputData,initialKey)
print(f"Correct Encryption:{cipher[:-2]==cipherText}")
print(f"Correct Decryption:{AES.decrypt_ecb(cipher,initialKey)==inputData}")

Correct Encryption:True
Correct Decryption:True
```

Figure 15: Verification for AES192 ECB 2

4.4 Encryption Decryption using AES – 256

4.4.1 CBC

```
#testcase for AES256 CBC
inputData="91fbef2d15a97816060bee1feaa49afe"
initialKey="0000000000000000000000000000000000000000000000000000000000000000"
IV="00000000000000000000000000000000"
cipherText="1bc704f1bce135ceb810341b216d7abe"
cipher=AES.encrypt_cbc(inputData,initialKey,IV)
print(f"Correct Encryption:{cipher[:-2]==cipherText}")
print(f"Correct Decryption:{AES.decrypt_cbc(cipher,initialKey,IV)==inputData}")

Correct Encryption:True
Correct Decryption:True
```

Figure 16: Verification for AES256 CBC 1

```
#testcase for AES256 CBC
inputData="cf52e5c3954c51b94c9e38acb8c9a7c76aebdaa9943eae0a1ce155a2efdb4d46985d935511471452d9ee64d2461cb2991d59fc0060697f9a671672163230f367fed1422316e52d29eceacb8768f56d9
initialKey="9ddf3745896504ff360a51a3eb49c01b79fccebc71c3abcb94a949408b05b2c9"
IV="e79026639d4aa230b5ccffb0b29d79bc"
cipherText="34df561bd2cfebbcb7af3b4b8d21ca5258312e7e2e4e538e35ad2490b6112f0d7f148f6aa8d522a7f3c61d785bd667db0e1dc4606c318ea4f26af4fe7d11d4dcff0456511b4aed1a0d91ba4a1fd6cd
cipher=AES.encrypt_cbc(inputData,initialKey,IV)
print(f"Correct Encryption:{cipher[:-2]==cipherText}")
print(f"Correct Decryption:{AES.decrypt_cbc(cipher,initialKey,IV)==inputData}")
```

```
Correct Encryption:True
Correct Decryption:True
```

Figure 17: Verification for AES256 CBC 2

4.5 ECB

```
#testcase for AES256 ECB
inputData="91fbef2d15a97816060bee1feaa49afe"
initialKey="0000000000000000000000000000000000000000000000000000000000000000"
cipherText="1bc704f1bce135ceb810341b216d7abe"
cipher=AES.encrypt_ecb(inputData,initialKey)
print(f"Correct Encryption:{cipher[:-2]==cipherText}")
print(f"Correct Decryption:{AES.decrypt_ecb(cipher,initialKey)==inputData}")
```

```
Correct Encryption:True
Correct Decryption:True
```

Figure 18: Verification for AES256 ECB 1

```
#testcase for AES256 ECB
inputData="e199899e1e12cffcb28909aec51b36c2f96fab49ef32b9650cc38aa37d2f4c8b785f9176c590f6a07e04037e13f7535290d5f5fc23aa1113d9dacf34a812749ab27ecfefc83dd3622d1285fa9d5c192
initialKey="2e39c585ce4900d323ce29713bebe73a1be08a0cb22e9f1310fcc14ad4b9b23e"
cipherText="2c4998428e72f6d96e982a316f73bf2a7da81730909b65403489ab92ada6de11882d08742f90f0f109d3420b00b8abe6873f4fdd1474923da2c5bdea2e4523ffca2132015ecf7c9cac9de2f956b112
cipher=AES.encrypt_ecb(inputData,initialKey)
print(f"Correct Encryption:{cipher[:-2]==cipherText}")
print(f"Correct Decryption:{AES.decrypt_ecb(cipher,initialKey)==inputData}")
```

```
Correct Encryption:True
Correct Decryption:True
```

Figure 19: Verification for AES256 ECB 2

4.6 Key Scheduling testing

```
initialKey="3c4baaf4bdfa1e3de57891cfde4522ab"
subKeys=KS.genKeyState(initialKey)
subKeys

['3c4baaf4bdfa1e3de57891cfde4522ab',
 '53d8c8e9ee22d6d40b5a471bd51f65b0',
 '91952fea7fb7f93e74edbe25a1f2db95',
 '1c2c05d8639bfce6177642c3b6849956',
 '4bc2b496285948703f2f0ab389ab93e5',
 '391e6d31114725412e682ff2a7c3bc17',
 '377b9d6d263cb82c085497deaf972bc9',
 'ff8a4014d9b6f838d1e26fe67e75442f',
 'e29155e73b27addfeac5c23994b08616',
 '1ed512c525f2bf1acf377d235b87fb35',
 '3fda84fc1a283be6d51f46c58e98bdf0']
```

Figure 20: Sub keys generated from a 128 bit initial key

```
initialKey="3c4baaf4bdfa1e3de57891cfde4522ab3c4b6fff21a3b5ed"
subKeys=KS.genKeyState(initialKey)
subKeys

['3c4baaf4bdfa1e3de57891cfde4522ab',
 '3c4b6fff21a3b5ed379eff098a64e134',
 '6f1c70fbb15952508d123dafacb18842',
 'fd5ad398773e32ac18224257a97b1007',
 '24692da888d8a5ea985c545cef6266f0',
 'f74024a75e3b34a07a521908f28abce2',
 'ee39ccd5015baa25f61b8e82a820ba22',
 'd272a32a20f81fc8bff92462bea28e47',
 '48b900c5e099bae732eb19cd12130605',
 'e2964fab5c34c1ec148dc129f4147bce',
 'c6ff6203d4ec64066cd520e330e1e10f',
 '246c2026d0785be8168739ebc26b5ded',
 '939975c6a37894c98714b4ef576cef07']
```

Figure 21: Sub keys generated from a 192 bit initial key

```
initialKey="3c4baaf4bdfa1e3de57891cfde4522ab3c4b6fff21a3b5ed56e11a99a0cd4566"
subKeys=KS.genKeyState(initialKey)
subKeys

['3c4baaf4bdfa1e3de57891cfde4522ab',
 '3c4b6fff21a3b5ed56e11a99a0cd4566',
 '802599143ddf8729d8a716e606e2344d',
 '53d3771c7270c2f12491d868845c9d0e',
 'c87b324bf5a4b5622d03a3842be197c9',
 'a22bffc1d05b3d30f4cae55870967856',
 '5cc7831aa9633678846095fcaf810235',
 'db2788570b7cb567ffb6503f8f202869',
 'e3f37a694a904c11cef0d9ed6171dbd8',
 '348431363ff88451c04ed46e4f6efc07',
 '6c43bfed26d3f3fce8232a118952f1c9',
 '938490ebac7c14ba6c32c0d4235c3cd3',
 '06a8d9cb207b2a37c8580026410af1ef',
 '10e33134bc9f258ed0ade55af3f1d989',
 'e79d7ec6c7e654f10fbe54d74eb4a538']
```

Figure 22: Sub keys generated from a 256 bit initial key

4.7 Bitmap Encryption

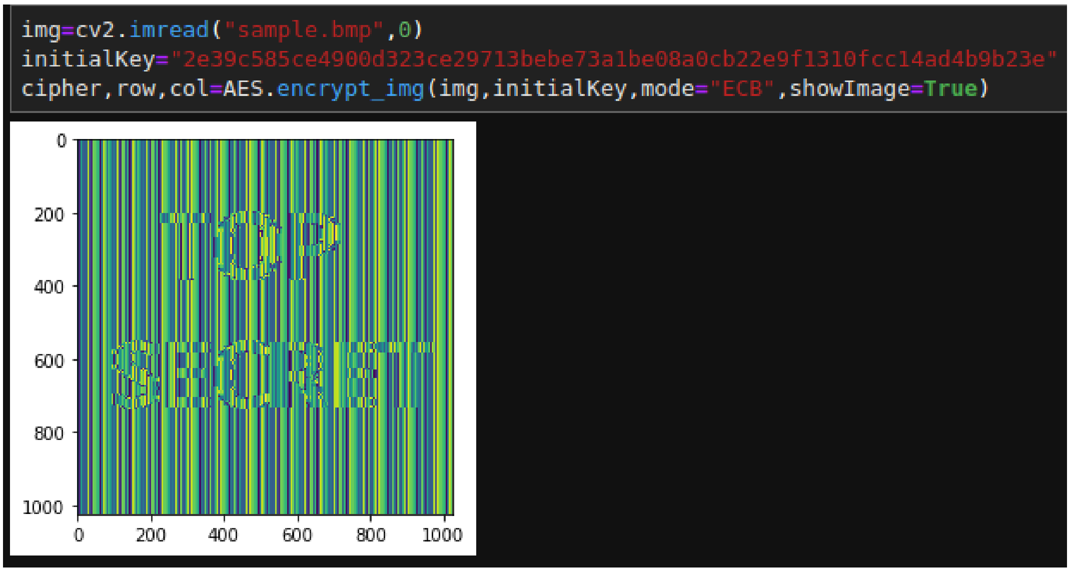

Figure 23: Bitmap encryption using AES256 ECB

Leakage is very apparent when using ECB with images. The image general structure can be seen.

Figure 24: Bitmap decryption using AES256 ECB

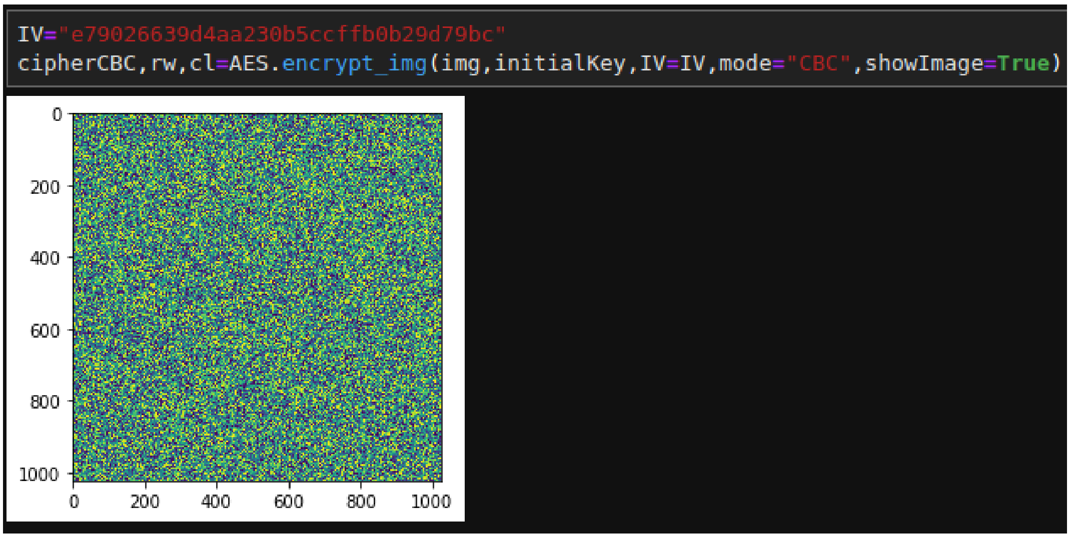

Figure 25: Bitmap encryption using AES256 CBC

The whole image is garbled when using CBC with images. Nothing can be seen.

```
clearCBC=AES.decrypt_img(cipherCBC,initialKey,rw,cl,IV=IV,mode="CBC",showImage=True)
```

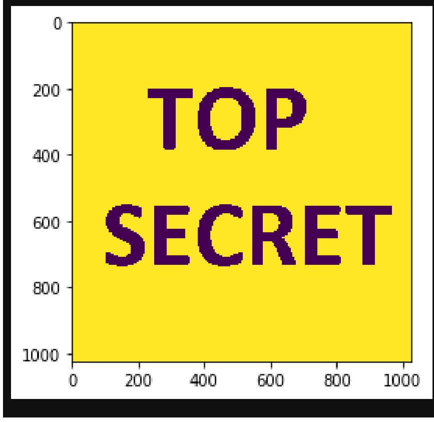

Figure 26: Bitmap decryption using AES256 CBC