Cryptography Lab 1

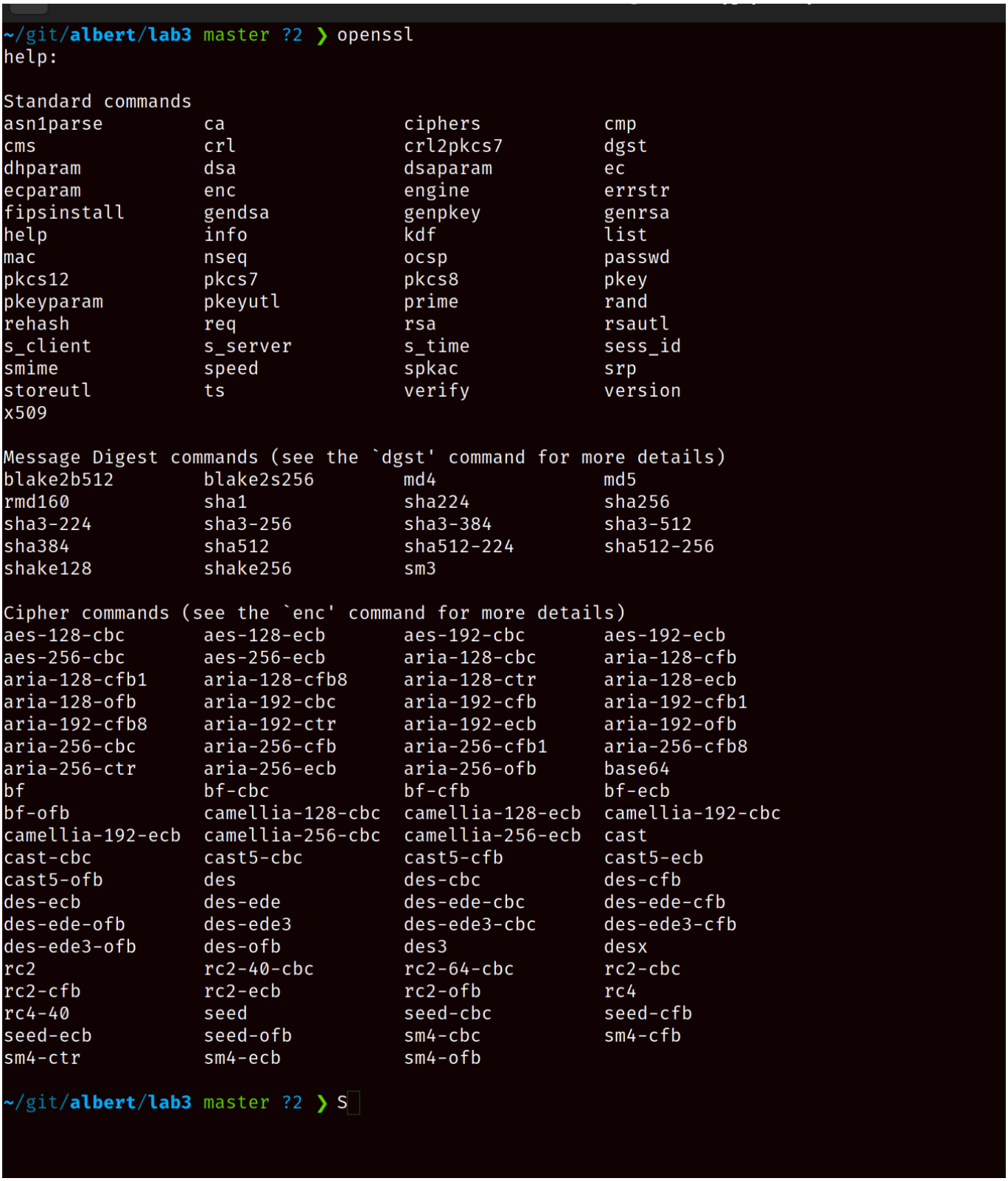
Name: Albert Ferguson **SID:** 13611165

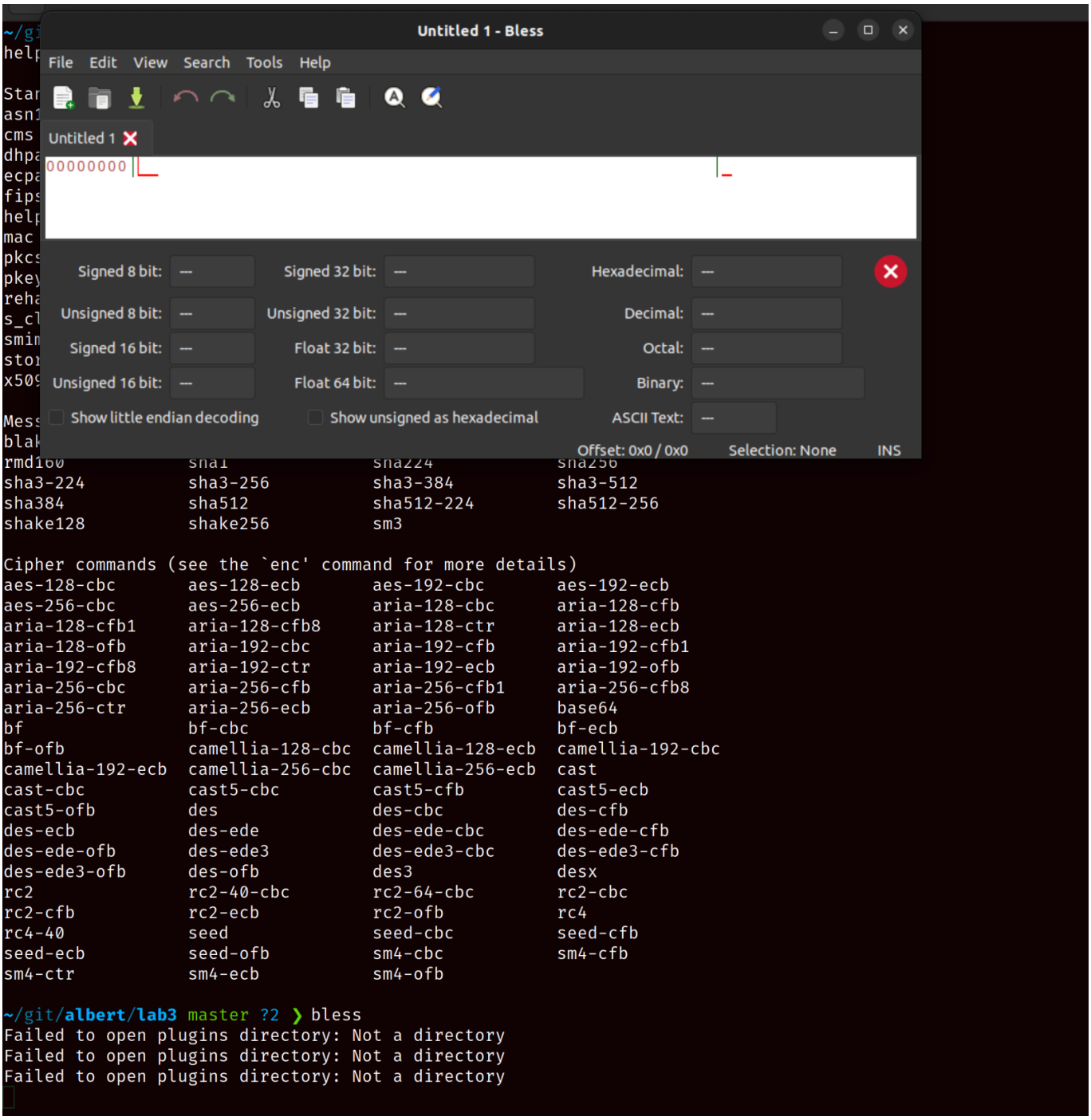
Setup

Installing dependencies, `openssl` and `bless`. I also needed to init some config for bless and install some further dependencies, `libcanberra-gtk-module libcanberra-gtk3-module`.

```
sudo apt-get install openssl bless
touch ~/.config/bless/plugins
touch ~/.config/bless/export_patterns
sudo apt install libcanberra-gtk-module libcanberra-gtk3-module -y
```

Testing it all out,


~/git/albert/lab3 master ?2 ❯ openssl
help:
Standard commands
asn1parse ca ciphers cmp
cms crl crl2pkcs7 dgst
dhparam dsa dsaparam ec
ecparam enc engine errstr
fipsinstall gendsa genpkey genrsa
help info kdf list
mac nseq ocsp passwd
pkcs12 pkcs7 pkcs8 pkey
pkeyparam pkeyutl prime rand
rehash req rsa rsautl
s_client s_server s_time sess_id
smime speed spkac srp
storeutl ts verify version
x509
Message Digest commands (see the `dgst' command for more details)
blake2b512 blake2s256 md4 md5
rmd160 sha1 sha224 sha256
sha3-224 sha3-256 sha3-384 sha3-512
sha384 sha512 sha512-224 sha512-256
shake128 shake256 sm3
Cipher commands (see the `enc' command for more details)
aes-128-cbc aes-128-ecb aes-192-cbc aes-192-ecb
aes-256-cbc aes-256-ecb aria-128-cbc aria-128-cfb
aria-128-cfb1 aria-128-cfb8 aria-128-ctr aria-128-ecb
aria-128-ofb aria-192-cbc aria-192-cfb aria-192-cfb1
aria-192-cfb8 aria-192-ctr aria-192-ecb aria-192-ofb
aria-256-cbc aria-256-cfb aria-256-cfb1 aria-256-cfb8
aria-256-ctr aria-256-ecb aria-256-ofb base64
bf bf-cbc bf-cfb bf-ecb
bf-ofb camellia-128-cbc camellia-128-ecb camellia-192-cbc
camellia-192-ecb camellia-256-cbc camellia-256-ecb cast
cast-cbc cast5-cbc cast5-cfb cast5-ecb
cast5-ofb des des-cbc des-cfb
des-ecb des-ede des-ede-cbc des-ede-cfb
des-ede-ofb des-ede3 des-ede3-cbc des-ede3-cfb
des-ede3-ofb des-ofb des3 desx
rc2 rc2-40-cbc rc2-64-cbc rc2-cbc
rc2-cfb rc2-ecb rc2-ofb rc4
rc4-40 seed seed-cbc seed-cfb
seed-ecb seed-ofb sm4-cbc sm4-cfb
sm4-ctr sm4-ecb sm4-ofb
~/git/albert/lab3 master ?2 ❯ S




```
sha3-224          sha3-256          sha3-384          sha3-512
sha384            sha512            sha512-224        sha512-256
shake128          shake256          sm3

Cipher commands (see the `enc' command for more details)
aes-128-cbc       aes-128-ecb       aes-192-cbc       aes-192-ecb
aes-256-cbc       aes-256-ecb       aria-128-cbc      aria-128-cfb
aria-128-cfb1     aria-128-cfb8     aria-128-ctr      aria-128-ecb
aria-128-ofb      aria-192-cbc      aria-192-cfb      aria-192-cfb1
aria-192-cfb8     aria-192-ctr      aria-192-ecb      aria-192-ofb
aria-256-cbc      aria-256-cfb      aria-256-cfb1     aria-256-cfb8
aria-256-ctr      aria-256-ecb      aria-256-ofb      base64
bf                bf-cbc            bf-cfb            bf-ecb
bf-ofb            camellia-128-cbc  camellia-128-ecb  camellia-192-cbc
camellia-192-ecb  camellia-256-cbc  camellia-256-ecb  cast
cast-cbc          cast5-cbc         cast5-cfb         cast5-ecb
cast5-ofb         des               des-cbc           des-cfb
des-ecb           des-ede           des-ede-cbc       des-ede-cfb
des-ede-ofb       des-ede3          des-ede3-cbc      des-ede3-cfb
des-ede3-ofb      des-ofb           des3              desx
rc2               rc2-40-cbc        rc2-64-cbc        rc2-cbc
rc2-cfb           rc2-ecb           rc2-ofb           rc4
rc4-40            seed              seed-cbc          seed-cfb
seed-ecb          seed-ofb          sm4-cbc           sm4-cfb
sm4-ctr           sm4-ecb           sm4-ofb

~/git/albert/lab3 master ?2 ❯ bless
Failed to open plugins directory: Not a directory
Failed to open plugins directory: Not a directory
Failed to open plugins directory: Not a directory
```

Tasks

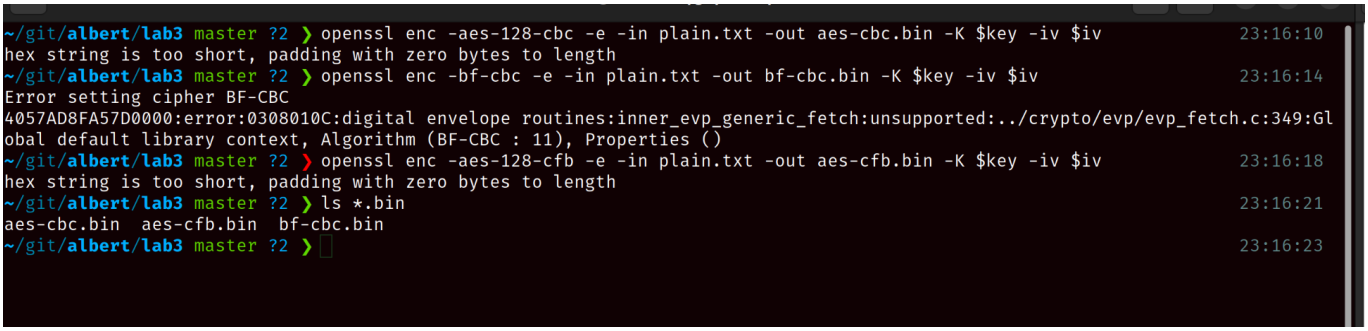
3.1 Encryption using different ciphers and modes

Before running, checkign the help and available ciphers,

```
 -K val              Raw key, in hex
 -S val              Salt, in hex
 -iv val             IV in hex
 -md val             Use specified digest to create a key from the passphrase
 -iter +int          Specify the iteration count and force use of PBKDF2
 -pbkdf2             Use password-based key derivation function 2
 -none               Don't encrypt
 -*                  Any supported cipher

Random state options:
 -rand val           Load the given file(s) into the random number generator
 -writerand outfile  Write random data to the specified file

Provider options:
 -provider-path val  Provider load path (must be before 'provider' argument if required)
 -provider val       Provider to load (can be specified multiple times)
 -propquery val      Property query used when fetching algorithms
~/git/albert/lab3 master ?2 ❯ openssl enc --ciphers
Supported ciphers:
-aes-128-cbc               -aes-128-cfb               -aes-128-cfb1
-aes-128-cfb8              -aes-128-ctr               -aes-128-ecb
-aes-128-ofb               -aes-192-cbc               -aes-192-cfb
-aes-192-cfb1              -aes-192-cfb8              -aes-192-ctr
-aes-192-ecb               -aes-192-ofb               -aes-256-cbc
-aes-256-cfb               -aes-256-cfb1              -aes-256-cfb8
-aes-256-ctr               -aes-256-ecb               -aes-256-ofb
-aes128                    -aes128-wrap               -aes192
-aes192-wrap               -aes256                    -aes256-wrap
-aria-128-cbc              -aria-128-cfb              -aria-128-cfb1
-aria-128-cfb8             -aria-128-ctr              -aria-128-ecb
-aria-128-ofb              -aria-192-cbc              -aria-192-cfb
-aria-192-cfb1             -aria-192-cfb8             -aria-192-ctr
-aria-192-ecb              -aria-192-ofb              -aria-256-cbc
-aria-256-cfb              -aria-256-cfb1             -aria-256-cfb8
-aria-256-ctr              -aria-256-ecb              -aria-256-ofb
-aria128                   -aria192                   -aria256
-bf                        -bf-cbc                    -bf-cfb
-bf-ecb                    -bf-ofb                    -blowfish
-camellia-128-cbc          -camellia-128-cfb          -camellia-128-cfb1
-camellia-128-cfb8         -camellia-128-ctr          -camellia-128-ecb
-camellia-128-ofb          -camellia-192-cbc          -camellia-192-cfb
-camellia-192-cfb1         -camellia-192-cfb8         -camellia-192-ctr
-camellia-192-ecb          -camellia-192-ofb          -camellia-256-cbc
-camellia-256-cfb          -camellia-256-cfb1         -camellia-256-cfb8
-camellia-256-ctr          -camellia-256-ecb          -camellia-256-ofb
-camellia128               -camellia192               -camellia256
-cast                      -cast-cbc                  -cast5-cbc
-cast5-cfb                 -cast5-ecb                 -cast5-ofb
-chacha20                  -des                       -des-cbc
-des-cfb                   -des-cfb1                  -des-cfb8
-des-ecb                   -des-ede                   -des-ede-cbc
-des-ede-cfb               -des-ede-ecb               -des-ede-ofb
-des-ede3                  -des-ede3-cbc              -des-ede3-cfb
-des-ede3-cfb1             -des-ede3-cfb8             -des-ede3-ecb
-des-ede3-ofb              -des-ofb                   -des3
-des3-wrap                 -desx                      -desx-cbc
-id-aes128-wrap            -id-aes128-wrap-pad        -id-aes192-wrap
-id-aes192-wrap-pad        -id-aes256-wrap            -id-aes256-wrap-pad
-id-smime-alg-CMS3DESwrap  -rc2                       -rc2-128
-rc2-40                    -rc2-40-cbc                -rc2-64
-rc2-64-cbc                -rc2-cbc                   -rc2-cfb
-rc2-ecb                   -rc2-ofb                   -rc4
-rc4-40                    -seed                      -seed-cbc
-seed-cfb                  -seed-ecb                  -seed-ofb
-sm4                       -sm4-cbc                   -sm4-cfb
-sm4-ctr                   -sm4-ecb                   -sm4-ofb

~/git/albert/lab3 master ?2 ❯
```

```
key=00112233445566778889aabbccddeeff
iv=0102030405060708
openssl enc -aes-128-cbc -e -in plain.txt -out aes-cbc.bin -K $key -iv $iv
# don't provide the IV for the BF-CBC mode, as it isn't needed
openssl enc -bf-cbc -e -in plain.txt -out bf-cbc.bin -K $key
openssl enc -aes-128-cfb -e -in plain.txt -out aes-cfb.bin -K $key -iv $iv
```

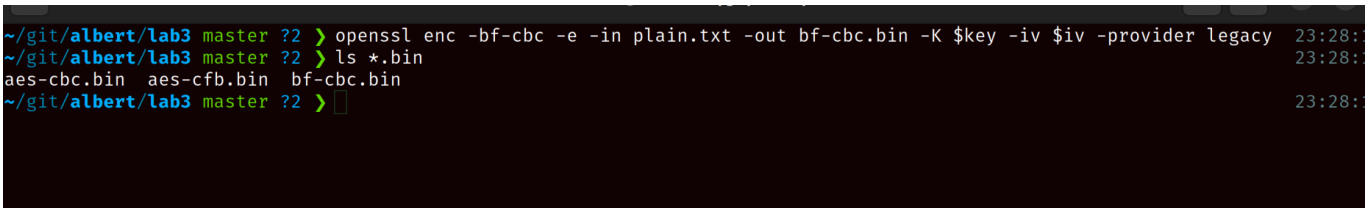
This resulted in the output,



Notably I was given a warning that the "hex string was too short" for AES in both CBD and CFB modes. This would be because the key and IV are the wrong length. OpenSSL looks to be automatically padding these with zeroes to ensure they are the correct length before proceeding.

Also, the `bf-cbd` algorithm throws an error while setting up the cipher. I imagine this is due to OpenSSL not supporting insecure algorithms by default. Checking the OpenSSL docs (https://www.openssl.org/docs/man3.0/man7/OSSL_PROVIDER-legacy.html), this would be because I'm using the default provider instead of the legacy provider. Swapping to this provider resolves the issue,

```
openssl enc -bf-cbc -e -in plain.txt -out bf-cbc.bin -K $key -provider
legacy
```



3.2 encryption mode ECB vs. CBC

Similar to above, I run OpenSSL with the legacy provider for ECB mode and the default for CBC mode,

```
openssl enc -bf-ecb -e -in uts.bmp -out uts-ecb.bmp -K $key -provider
legacy
openssl enc -aes-128-cbc -e -in uts.bmp -out uts-cbc.bmp -K $key -iv $iv
```

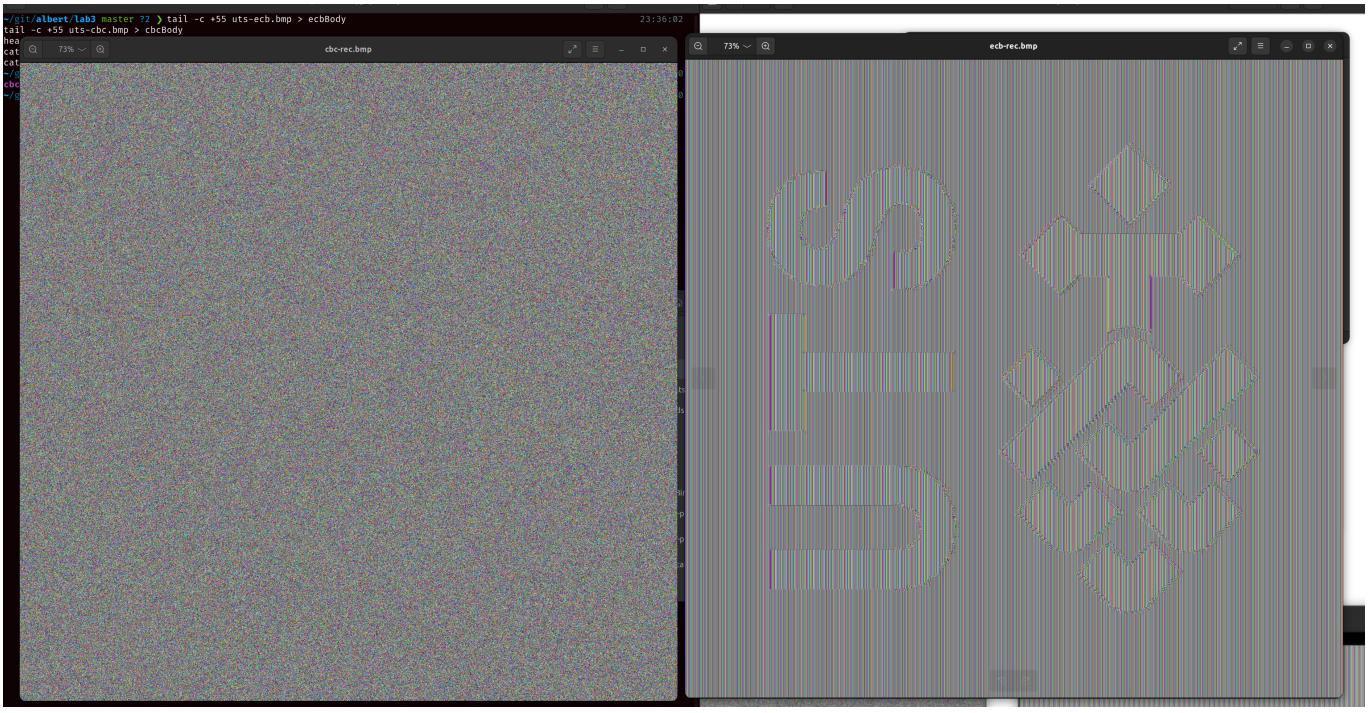
```
~/git/albert/lab3 master ?2 ❯ openssl enc -bf-ecb -e -in uts.bmp -out uts-ecb.bmp -K $key -provider legacy   23:29:36
~/git/albert/lab3 master ?2 ❯ ls *.bmp                                                                      23:29:40
uts.bmp  uts-ecb.bmp
~/git/albert/lab3 master ?2 ❯ openssl enc -aes-128-cbc -e -in uts.bmp -out uts-cbc.bmp -K $key -iv $iv       23:29:44
hex string is too short, padding with zero bytes to length
~/git/albert/lab3 master ?2 ❯ ls *.bmp                                                                      23:31:04
uts.bmp  uts-cbc.bmp  uts-ecb.bmp
~/git/albert/lab3 master ?2 ❯                                                                               23:31:07
```

To view these in an image viewer, we need to reconstruct the headers. Otherwise the image viewer will complain that the header data is bogus (which it is at the moment). To recreate them I followed the lab prompt,

```
tail -c +55 uts-ecb.bmp > ecbBody
tail -c +55 uts-cbc.bmp > cbcBody
head -c 54 uts.bmp > header
cat header ecbBody > ecb-rec.bmp
cat header cbcBody > cbc-rec.bmp
```

```
~/git/albert/lab3 master ?2 ❯ tail -c +55 uts-ecb.bmp > ecbBody                                             23:36:02
tail -c +55 uts-cbc.bmp > cbcBody
head -c 54 uts.bmp > header
cat header ecbBody > ecb-rec.bmp
cat header cbcBody > cbc-rec.bmp
~/git/albert/lab3 master ?2 ❯ ls *-rec.bmp                                                                  23:36:20
cbc-rec.bmp  ecb-rec.bmp
~/git/albert/lab3 master ?2 ❯                                                                               23:36:30
```

Then in an image viewer (CBC left, ECB right),



> Q: Can you derive any useful information about the original picture from the encrypted picture? Please explain your observations.

> A: yes (in ECB mode)

ECB mode was only considered for short-length data with non-repeating bits, as repeating bits will always result in the same ciphertext. This weakness is obvious for the sample image, as it is 7.7MB (`du -sh uts.bmp`). Simply, although the repeated bits are encrypted they are the same, so patterns and colors from the original image are still apparent.

However, CBC mode is superior, as the resultant ciphertext is not obviously based on the plaintext.

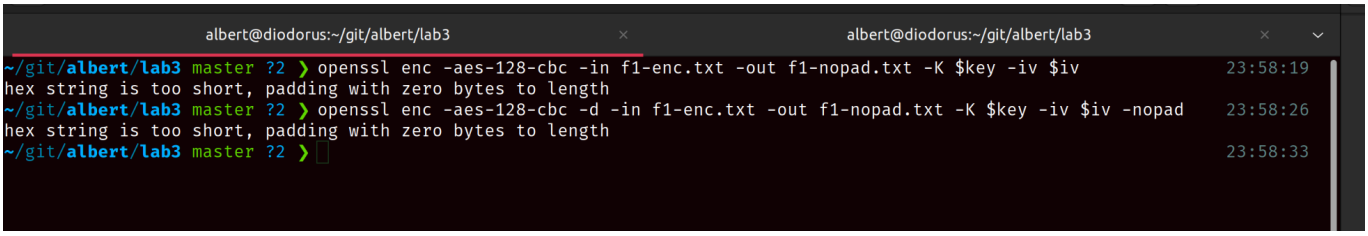
3.3 Padding

The task requests that I check ECB, CBC, CFB, and OFB modes and assert which use padding. Any of the above modes which use block modes will require padding to ensure the plaintext blocks match the expected size. The stream based approaches do not require this padding.

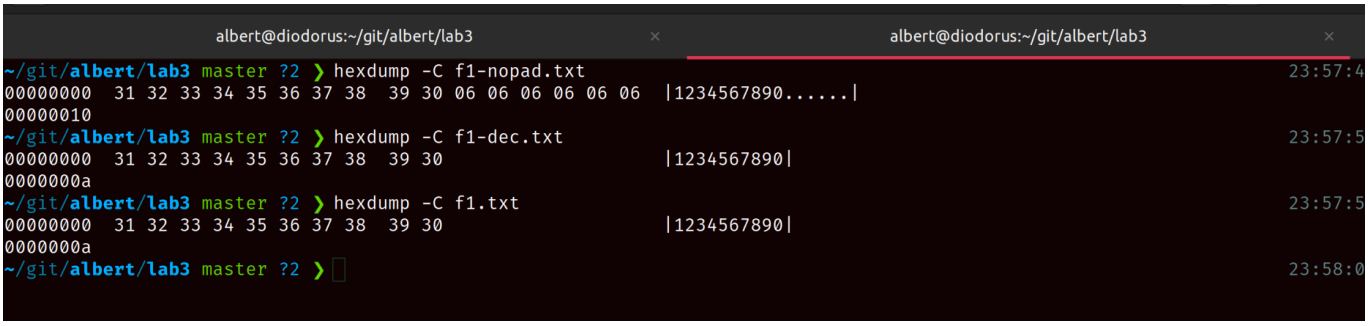
ECB and CBC are block-wise algorithms, so will require padding. CFB will not, as it is stream-based. Since OFC (and CTR) are similar implementations, they also don't require padding.

We can test this assumption by using the `-nopad` option in OpenSSL then checking the padding manually,

```
# create a simple plaintext file
echo -n "1234567890" > f1.txt
# encrypt it with CBC (blockwise algorithm) which should apply padding
openssl enc -aes-128-cbc -in f1-enc.txt -out f1-nopad.txt -K $key -iv $iv
# decrypt without automatically removing padding
openssl enc -aes-128-cbc -d -in f1-enc.txt -out f1-nopad.txt -K $key -iv
$iv -nopad
# again but as per usual
openssl enc -aes-128-cbc -d -in f1-enc.txt -out f1-nopad.txt -K $key -iv
$iv
```



Then with hexdump to compare the padding we can see that the following hex was adding as padding to the plaintext, `06 06 06 06 06 06`,



3.4 Error propagation - corrupted ciphertext

First we need a 64 byte input. We can achieve this using dd,

```
dd if=/dev/urandom of=./sample.plaintext bs=1 count=64
# > 64 bytes copied, 0.000254261 s, 252 kB/s
# verify with du
du -bhs ./sample.plaintext
# > 64  ./sample.plaintext
```

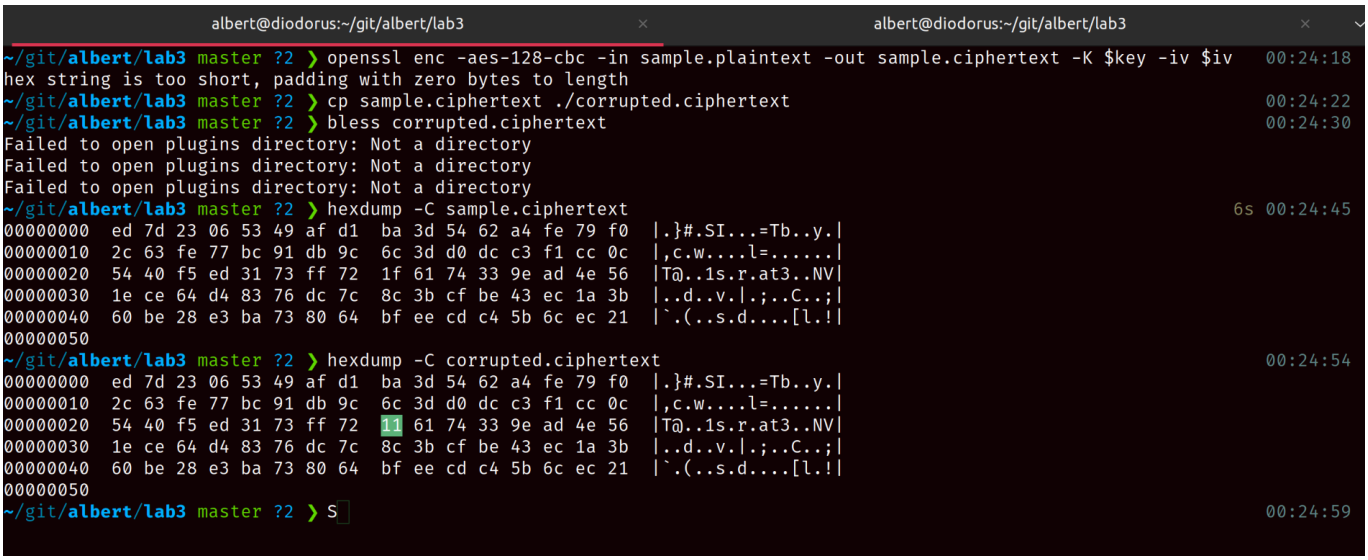
A quick hexdump shows we have a fairly random sample,

```
00000000  76 53 9b cb 48 18 e1 3f  b0 7b 58 5c f2 6d 38 10  |vS..H..?.
{X\.m8.|
00000010  4a a9 53 fc 72 0c 88 1f  82 9e f8 8c 5c 9f 3b 73
|J.S.r.......\.;s|
00000020  d5 c3 f0 03 01 45 bd 7a  d9 1e 4b a8 ae 1f 1a f0
|.....E.z..K.....|
00000030  42 be 5d 72 8e 2d e7 bc  42 4e 03 28 ff 96 83 70  |B.]r.-..BN.
(...p|
00000040
```

We need to encrypt it as before, then create a copy before "corrupting" the copy using bless,

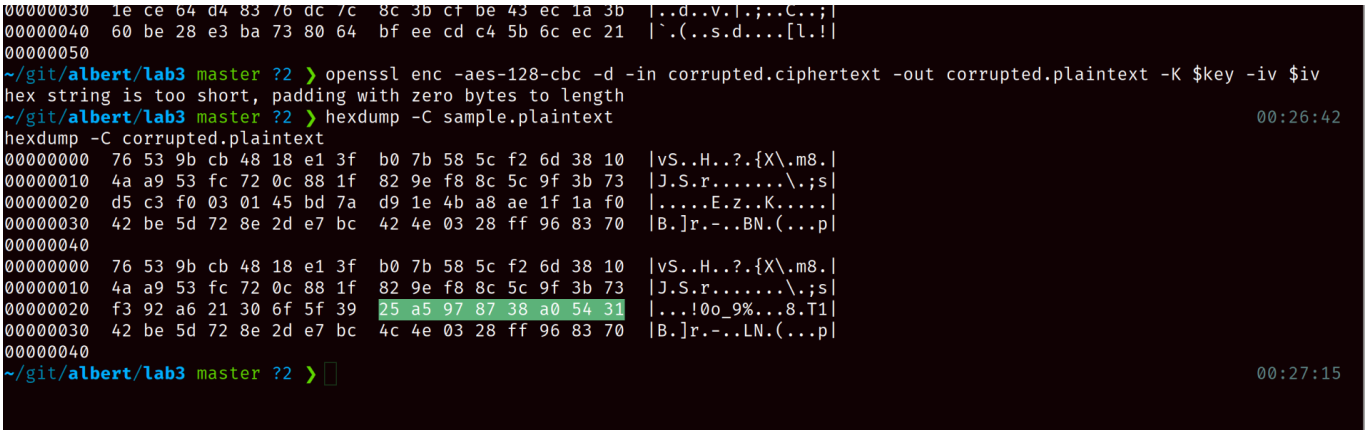
```
openssl enc -aes-128-cbc -e -in sample.plaintext -out sample.ciphertext -K
$key -iv $iv
cp sample.ciphertext ./corrupted.ciphertext
bless corrupted.ciphertext
```

Then using bless, I changed a single hex-value. The output with hexdump after saving compares the original and corrupted,



Decrypting the corrupted file and comparing with the original

```
openssl enc -aes-128-cbc -d -in corrupted.ciphertext -out
corrupted.plaintext -K $key -iv $iv
hexdump -C sample.plaintext
hexdump -C corrupted.plaintext
```



The highlighted block is now corrupted in plaintext. The remaining blocks are correct.

Repeating with ECB, CFB, OFB, and CTR:

```
dd if=/dev/urandom of=./ecb.plaintext bs=1 count=64
dd if=/dev/urandom of=./cfb.plaintext bs=1 count=64
dd if=/dev/urandom of=./ofb.plaintext bs=1 count=64
dd if=/dev/urandom of=./ctr.plaintext bs=1 count=64

openssl enc -bf-ecb -e -in ecb.plaintext -out ecb.ciphertext -K $key -
provider legacy
openssl enc -aes-128-cfb -e -in cfb.plaintext -out cfb.ciphertext -K $key -
iv $iv
openssl enc -aes-128-ofb -e -in ofb.plaintext -out ofb.ciphertext -K $key -
iv $iv
openssl enc -aes-128-ctr -e -in ctr.plaintext -out ctr.ciphertext -K $key -
iv $iv

cp ecb.ciphertext ecb-corrupted.ciphertext
cp cfb.ciphertext cfb-corrupted.ciphertext
cp ofb.ciphertext ofb-corrupted.ciphertext
cp ctr.ciphertext ctr-corrupted.ciphertext

# replacing the third character in each file
bless ecb-corrupted.ciphertext
bless cfb-corrupted.ciphertext
bless ofb-corrupted.ciphertext
bless ctr-corrupted.ciphertext

openssl enc -bf-ecb -d -in ecb.ciphertext -out ecb.plaintext -K $key -
provider legacy
openssl enc -bf-ecb -d -in ecb-corrupted.ciphertext -out ecb-
corrupted.plaintext -K $key -provider legacy
```

```
openssl enc -aes-128-cfb -d -in cfb.ciphertext -out cfb.plaintext -K $key -
iv $iv
openssl enc -aes-128-cfb -d -in cfb-corrupted.ciphertext -out cfb-
corrupted.plaintext -K $key -iv $iv

openssl enc -aes-128-ofb -d -in ofb.ciphertext -out ofb.plaintext -K $key -
iv $iv
openssl enc -aes-128-ofb -d -in ofb-corrupted.ciphertext -out ofb-
corrupted.plaintext -K $key -iv $iv

openssl enc -aes-128-ctr -d -in ctr.ciphertext -out ctr.plaintext -K $key -
iv $iv
openssl enc -aes-128-ctr -d -in ctr-corrupted.ciphertext -out ctr-
corrupted.plaintext -K $key -iv $iv

# compare hexdumps
echo ECB
hexdump -C ecb.plaintext
hexdump -C ecb-corrupted.plaintext

echo CFB
hexdump -C cfb.plaintext
hexdump -C cfb-corrupted.plaintext

echo OFB
hexdump -C ofb.plaintext
hexdump -C ofb-corrupted.plaintext

echo CTR
hexdump -C ctr.plaintext
hexdump -C ctr-corrupted.plaintext
```

```
~/git/albert/lab3 master ?2 ❯ echo ECB
hexdump -C ecb.plaintext
hexdump -C ecb-corrupted.plaintext

echo CFB
hexdump -C cfb.plaintext
hexdump -C cfb-corrupted.plaintext

echo OFB
hexdump -C ofb.plaintext
hexdump -C ofb-corrupted.plaintext

echo CTR
hexdump -C ctr.plaintext
hexdump -C ctr-corrupted.plaintext
ECB
00000000  de cf 8d a7 1f 68 d4 cd  0a 95 ac 17 1a e4 59 ea  |.....h........Y.|
00000010  91 6b 5d 05 8c f1 91 51  5f 2a d4 3f 2c 28 0a 43  |.k]....Q_*.?,(.C|
00000020  3b 97 cc b0 8f b7 0d 11  c9 2d a8 24 c1 fe cd 7c  |;........-.$...||
00000030  96 6b ca e0 4b 20 50 b9  cd 19 03 3c 94 e1 37 65  |.k..K P....<..7e|
00000040
00000000  19 05 52 42 c2 fc a9 b6  0a 95 ac 17 1a e4 59 ea  |..RB..........Y.|
00000010  91 6b 5d 05 8c f1 91 51  5f 2a d4 3f 2c 28 0a 43  |.k]....Q_*.?,(.C|
00000020  3b 97 cc b0 8f b7 0d 11  c9 2d a8 24 c1 fe cd 7c  |;........-.$...||
00000030  96 6b ca e0 4b 20 50 b9  cd 19 03 3c 94 e1 37 65  |.k..K P....<..7e|
00000040
CFB
00000000  ea 76 a5 66 ef 3c 1d 98  c4 3f 2e fa 3a 18 06 5a  |.v.f.<...?..:..Z|
00000010  c2 f1 99 cf 06 cd 09 7d  52 bb 45 c4 9f e2 a0 1b  |.......}R.E.....|
00000020  86 f0 64 93 0e 0a 31 18  48 c8 10 78 51 c6 34 85  |..d...1.H..xQ.4.|
00000030  5a 88 67 5c 30 3f 19 fc  dc 43 1d dd a4 1a aa fe  |Z.g\0?...C......|
00000040
00000000  ea 76 9e 66 ef 3c 1d 98  c4 3f 2e fa 3a 18 06 5a  |.v.f.<...?..:..Z|
00000010  d4 22 b1 43 49 d0 4f 74  29 35 92 43 5c 7e 75 62  |."CI.Ot)5.C\~ub|
00000020  86 f0 64 93 0e 0a 31 18  48 c8 10 78 51 c6 34 85  |..d...1.H..xQ.4.|
00000030  5a 88 67 5c 30 3f 19 fc  dc 43 1d dd a4 1a aa fe  |Z.g\0?...C......|
00000040
OFB
00000000  08 7b 64 2e fe a9 cd 37  50 4e a6 42 27 00 a2 94  |.{d....7PN.B'...|
00000010  d1 20 54 9f c3 52 9b b3  53 27 9f 72 58 55 70 18  |. T..R..S'.rXUp.|
00000020  af 50 dd 7e 65 a0 8a a7  a3 55 6b b7 dd 66 1c 97  |.P.~e....Uk..f..|
00000030  25 73 4e 17 74 7c 6b 84  62 a5 6b 95 4c 84 26 53  |%sN.t|k.b.k.L.&S|
00000040
00000000  08 7b 9e 2e fe a9 cd 37  50 4e a6 42 27 00 a2 94  |.{.....7PN.B'...|
00000010  d1 20 54 9f c3 52 9b b3  53 27 9f 72 58 55 70 18  |. T..R..S'.rXUp.|
00000020  af 50 dd 7e 65 a0 8a a7  a3 55 6b b7 dd 66 1c 97  |.P.~e....Uk..f..|
00000030  25 73 4e 17 74 7c 6b 84  62 a5 6b 95 4c 84 26 53  |%sN.t|k.b.k.L.&S|
00000040
CTR
00000000  fb 07 2c b3 1b ef 4c a2  1b db b6 f4 eb 3f 17 d6  |..,...L......?..|
00000010  0e 47 a5 bb be 22 34 a3  69 ba 3c 3a 87 78 0e 86  |.G..."4.i.<:.x..|
00000020  be d6 fb 9f 43 46 f8 8f  db a0 ee ce ac 4b 79 35  |....CF.......Ky5|
00000030  c4 ab a5 f2 ba 56 07 96  fb 7c 54 38 37 3b 32 99  |.....V...|T87;2.|
00000040
00000000  fb 07 9e b3 1b ef 4c a2  1b db b6 f4 eb 3f 17 d6  |......L......?..|
00000010  0e 47 a5 bb be 22 34 a3  69 ba 3c 3a 87 78 0e 86  |.G..."4.i.<:.x..|
00000020  be d6 fb 9f 43 46 f8 8f  db a0 ee ce ac 4b 79 35  |....CF.......Ky5|
00000030  c4 ab a5 f2 ba 56 07 96  fb 7c 54 38 37 3b 32 99  |.....V...|T87;2.|
00000040
~/git/albert/lab3 master ?2 ❯
```

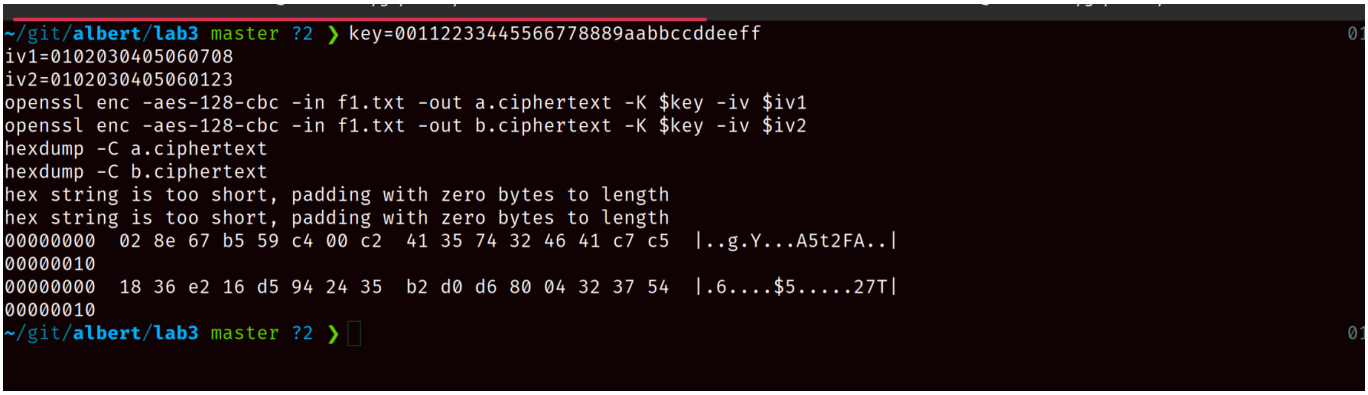
From the above results,

- ECB: can recover all plaintext except for the corrupted block
- CFB: self-heals and recovers all plaintext except the corrupted character and the next block (third row from CFB result screenshot).
- OFB and CTR: recover the best, as only the affected character is corrupted and all other plaintext is recovered.

3.5 Initial Vector (IV)

Setting up some variables and then running the encryption twice for two different keys,

```
key=00112233445566778889aabbccddeeff
iv1=0102030405060708
iv2=0102030405060123
openssl enc -aes-128-cbc -in f1.txt -out a.ciphertext -K $key -iv $iv1
openssl enc -aes-128-cbc -in f1.txt -out b.ciphertext -K $key -iv $iv2
hexdump -C a.ciphertext
hexdump -C b.ciphertext
```



> Q: What properties the IV should have? Why?

> A: In all cases, the IV should always be unique. The IV should also be unpredictable by 3rd parties and secret (until encryption has completed).

Regarding secrecy, the IV will be made public (typically as a prefix) once the ciphertext is transmitted.

An IV should be both unpredictable (to a 3rd party) as well as unique. If it is predictable, the cryptanalysis techniques can potentially provide knowledge of the plaintext. This is an issue when the IV is the same as the plaintext, as the resultant ciphertext will XOR to `0`.

If the IV is randomly unique but predictable, then there is potential for a chosen-plaintext attack. Whereby a 3rd party provides the encoder the original IV as well the predicted IV. By passing the predicted IV before encryption, the algorithm will cancel out the predicted IV and leave re-use the original IV. The encoder would then regenerate the original ciphertext.

Summary and discussion

This lab provided examples and problems regarding various cipher modes. by progressively exploring the encoding process, padding, and error propagation I've come to understand the iterative improvements and reasoning behind each mode.

This week's content builds on previous weeks, which have previously discussed IVs and mentioned CBC in relation to various WiFi spec's. Expanding on this content in detail after understanding an implementation gives me an excellent 'hook' to remember the content more effectively.

Further, having now broken down the vulnerabilities in CBC in detail, previous content regarding vulnerabilities in WEP and WPA make more sense to me, as I am now familiar with the "why" as well as the "what".