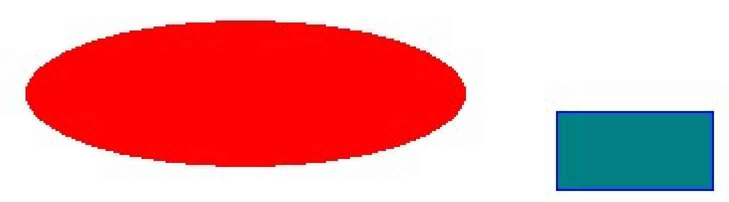
Comp 4580 A1

Katrina Dotzlaw [redacted]

Encryption Task 3



The original image:

To ensure an accurate comparison, I used the same key to encrypt with both CBC and ECB: `123456789`.

CBC

Process

First I encrypted the picture using the key defined above and the `aes-128-cbc` encryption method. I used `pic_original` as my input and the following command line argument produced the encrypted image `cbcpic_bmp`. `openssl enc -aes-128-cbc -e -in pic_original.bmp -out cbcpic.bmp`

Then I had to tell the encrypted image that it is actually a bmp image. I got the header from the original picture using the following command line argument and storing the results in `header`:
`head -c 54 pic_original.bmp > header`

Next I seperated out the body of the encrypted image and stored it in `body` using the following command line argument: `tail -c +55 cbcpic.bmp > body`

Lastly, I concatenated the header from the original image and the body from the encrypted image so the encrypted image could be viewed as a bmp. I used this command line argument: `cat header bodyy > cbc_img.bmp`

Observations

After applying CBC encryption, I got the following image as a result.

In this image, there are no clear patterns to be seen only 'noise'. Compared to ECB encryption, this is a more secure encryption since it doesn't show patterns.

ECB

Process

First, I encrypted the original image using the same key as CBC (123456789) and the `aes-128-ecb` encryption method. I used `pic_original.bmp` as my input and the following command encrypted the image into `ecb128`: `openssl enc -aes-128-ecb -e -in pic_original.bmp -out ecb128`

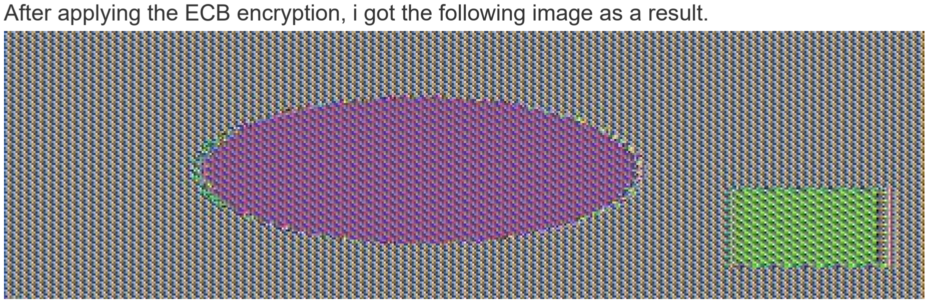
Next, I needed to tell the ECB encrypted image that it was a bmp so i got the header from the original image with: `head -c 54 pic_original.bmp > header`.

Then I seperated the body of the ECB encrypted image and stored it in `body2` using the following command: `tail -c +54 ecb128 > body2`.

Finally, I concatenated the header of the original image to the body of the ECB encrypted image using: `cat header body2 > ecb128_img.bmp`.

Observations

After applying the ECB encryption, i got the following image as a result.



In this image, we can see clear patterns that show the shapes and similar colours to the original image. This is expected since images are not suitable for ECB encryption because patterns are repeated in the encryption [1]. Compared to CBC, ECB encryption produces a less secure result because clear patterns are visible.

RSA Tasks 1-3

For all of the RSA tasks, I used the BIGNUM C library recommended in the lab instructions by creating a c file that included `stdio.h` and the BIGNUM library:

`#include <stdio.h>`

`#include <openssl/bn.h>`

I also used the print function (`printBN()`)they provided in their sample code to print out my BIGNUMs.

Task 1: Derive Private Key `d`

Process

First, I needed to define the context that would be used in association with the BIGNUM functions. I did this using the following code: `BN_CTX *ctx = BN_CTX_new();`.

Next, I defined `p`, `q`, `e`, `n`, and `d` as BIGNUMs using the following code:

```
BIGNUM *p = BN_new();
BIGNUM *q = BN_new();
BIGNUM *e = BN_new();
BIGNUM *n = BN_new();
BIGNUM *d = BN_new();
```

Then, I used the `BN_hex2bn(&addr, "hex")` method, I assigned `p`, `q`, and `e` to the hex values given to them in task 1:

```
BN_hex2bn(&p,"F7E75FDC469067FFDC4E847C51F452DF");
BN_hex2bn(&q,"E85CED54AF57E53E092113E62F436F4F");
BN_hex2bn(&e,"0D88C3");
```

Next, I needed to calulate `n = p*q` [2] but since all of the variables are BIGNUMs, regular operations cant be used on them. So I used `BN_mul(n,p,q,ctx)` to calculate `p*q`.

Then I needed to calculate `phi n` which meant I needed to define `phin` as a BIGNUM. Here's where I ran into my first problem during this process. Computing `phin = (p-1)*(q-1)` wouldn't work with the `BN_mul()` method. This was because 1 is not a BIGNUM and cant be used in BIGNUM functions. So I created 2 more BIGNUMS, `pone` and `qone`, to hold the hex values of `p` and `q` with 1 subtracted. The definitions of `phin`, `pone`, and `qone` as BIGNUMs are as follows:

```
BIGNUM *phin = BN_new();
BIGNUM *pone = BN_new();
BIGNUM *qone = BN_new();
```

And the assignment of `pone` and `quone` are:

```
BN_hex2bn(&pone,"F7E75FDC469067FFDC4E847C51F452DE");
BN_hex2bn(&qone,"E85CED54AF57E53E092113E62F436F4E");
```

Then I calculated `phin` using `BN_mul(phin, pone, qone, ctx)`. To calculate the private key `d`, I needed to compute a unique `d` such that `e*d` was identical to `1%phin` [2]. Using this definition of `d` we can see that `d` is the multiplicative modular inverse of `e` and `phin`, which allowed me to avoid the previous issue I had with 1 not being a BIGNUM. I calculated `d` using the following code:

```
//Task 1: Derive Private Key
//n = p*q
  BN_mul(n,p,q,ctx);
  //calcualte phin -- phin = (p-1)*(q-1)
  BN_mul(phin,pone,qone,ctx);
  //private key d: unique st ed is identical to 1%phin
  //d is multiplicative modular inverse of e & phin so
  BN_mod_inverse(d,e,phin,ctx);
  printBN("Private key=",d);
```

I got the following results for `d`:
`3587A24598E5F2A21DB007D89D18CC50ABA5075BA19A33890FE7C28A9B496AEB`

Results & Verification

To verify that `d` is correct, I decided to encrypt and decrypt the message "A top secret!" (4120746f702073656372657421 in hex). I created 2 new BIGNUMs `res` and `a` where `a` is a message in hex, and `res` will store the result of encrypting the message using the public key `(e,n)`:

```
//test
BIGNUM *res = BN_new();
BIGNUM *a = BN_new();
BN_hex2bn(&a,"4120746f702073656372657421");
```

I encrypted the message in `a` using the public key `(e,n)`, and `res = a^e % n` [2]. Next, I decrypted the encrypted message in `res` using private key `d` that I found and `res = res^d % n` [2]. The following code illustrates this process:

```
BN_mod_exp(res,a,e,n,ctx);
printBN("Encrypt=",res);

BN_mod_exp(res,res,d,n,ctx);
printBN("decrypt w d=",res);
```

The output is as follows:

`Encrypt= 90A81343DFE08415EDF79337CDE00457BAB56AFFA1B0CE5647BF9025665B396A`
`decrypt w d= 4120746F702073656372657421`

The resulting decrypted message was: `4120746F702073656372657421`. After decoding this value with python:

> > > bytes.fromhex("4120746F702073656372657421").decode()

> > > 'A top secret!'

Since the original message and the message decrypted with `d` match, `d` is correct.

Task 2: Encrypting a Message

Process

Because I used the same message as the lab, "A top secret!", I didnt have to convert it to hex since the lab already did that.

First, I defined `n2`, `e2`, `msg`, and `d2` as BIGNUMs and used `BN_hex2bn(&addr, hex)` to assign their hex values:

```
  BIGNUM *n2 = BN_new();
  BIGNUM *e2 = BN_new();
  BIGNUM *d2 = BN_new();
  BN_hex2bn(&n2,"DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
BN_hex2bn(&e2, "010001");
BN_hex2bn(&d2,"74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");
```

Then, I needed to encrypt the message using `res = msg^e % n` [2]. I used the public key `(e2, n2)` to encrypt the message and store it in `res`:

```
  BIGNUM *msg = BN_new();
  BN_hex2bn(&msg,"4120746f702073656372657421");
  //encrypt
  BN_mod_exp(res,msg,e2,n2,ctx);
  printBN("Encrypted msg= ",res);
```

The resulting encrypted message is:
`6FB078DA550B2650832661E14F4F8D2CFAEF475A0DF3A75CACDC5DE5CFC5FADC`

Results & Verification

To verify that this encryption is correct, I decryped it using `d2` and `res = res^d2 % n2` [2]:

```
  //verify
BN_mod_exp(res,res,d2,n2,ctx);
  printBN("Decrypted msg=",res);
```

The decrypted message is `4120746F702073656372657421` and after decoding it using python:

> > > bytes.fromhex("4120746F702073656372657421").decode()

> > > 'A top secret!'

It's the same message as the original one which shows that it was encrypted correctly.

Task 3: Decrypt a Message

Process

First, I defined a cipher `c` as a BIGNUM and assigned its value as the one provided in the lab using `BN_hex2bn(&addr, hex)`:

```
 //Task 3: Decrypt Msg
 BIGNUM *c = BN_new(); //cipher
 //use same n2,e2,d2 as task 2
BN_hex2bn(&c,"8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBDFC7DCB67396567EA1E2493F");
```

Then I decrypted this message using the private key in Task 2, `d2`, and `n2` from Task 2 by following the formula `res = c^d2 % n2` [2].

```
 //Task 3: Decrypt Msg
 BIGNUM *c = BN_new(); //cipher
 //use same n2,e2,d2 as task 2
BN_hex2bn(&c,"8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBDFC7DCB67396567EA1E2493F");
 BN_mod_exp(res,c,d2,n2,ctx);
 printBN("Decrypted msg=",res);
```

The decrypted message is: `50617373776F72642069732064656573`

After decoding with python:

> > > bytes.fromhex("50617373776F72642069732064656573").decode()

> > > 'Password is dees'

References

[1] Dr. Noman Mohammed. Cryptography II Slide 15. 30/01/24.

[2] Dr. Noman Mohammed. Cryptography III Slide 16. 30/01/24.