Project 3:
DES

CSCI 360

October 11, 2017

In this project we will learn to use real ciphers from the command line. We will also compile and run real code for DES. Lastly we will implement a simplified CBC mode encryption using a provided DES encryption function.

1 Using DES

Suppose you want to encrypt your diary using DES so that your sister can't read it. How do you do this? One natural way is to use `openssl`, an open source implementation of many cryptographic protocols. This beloved utility has been in the news a lot for the last couple of years because of a few high profile security failures.[1]

Most often openssl is used through library calls in code, but it can also be used from the command line in Linux. This website provides some instructions and examples:

https://wiki.openssl.org/index.php/Enc

Have a look at that page and take in the prototype usage. Then scroll down to the EXAMPLES section (or click the EXAMPLES link at the top of the page). All of these will work from a terminal window (on cocalc or on your local machine).

The first thing you will see are references to Base64. The purpose of Base64 is to allow encrypted files to be represented as ASCII text. Normally this is not possible, because certain byte values fall outside the range of printable ASCII characters.[2] But Base64 encoding allows ciphertext (or anything digital) to be pasted into instant messages or email. There is a nice Wikipedia article on the topic of Base64 encoding which you should probably read if you want to use crypto in real life.

Let me send you a secret message. Here is a DES encrypted sentence represented in Base64 encoding:

iQ/nqbqZxbr3UQbsqxOmFpYSkZ/oO3Sv/in4BTfv3hM=

[1] The most famous of these was the "Heartbleed" bug. As a result of the failures there are now several forks of openssl (e.g. libressl) under active development. Here is a list of other implementations of common cryptographic algorithms: `http://en.wikipedia.org/wiki/Comparison_of_TLS_implementations.`

[2] Remember that you can type `man ascii` at the terminal prompt to see which bytes correspond to keyboard symbols.

Copy this text and paste it into a file. Save the file and name it `A.b64`. Now, to decrypt you need to get this back into regular binary encoding. Use this command from a terminal (do not type the $):

```
$ openssl base64 -d -in A.b64 -out A.des
```

You can do a `cat A.dex` to verify that the printable characters are gone. If you want to see exacly what bytes `A.dex` contains you can execute `xxd A.des` in the terminal.

Of course, you want the plaintext and we still have only the ciphertext, now in a binary format. The file is encrypted using DES with the password `abababababababab`

To decrypt the file, do this:

```
openssl des -d -in A.des -out msg.txt -K abababababababab -iv 1234567876543210
```

The above command writes the plaintext to the file `msg.txt`. You can read it now, if you want. As with Base64, the `-d` option means "decrypt". The `-K` option in the command specifies the key in hexadecimal. Verify that the key is given as 8 bytes (of course only 56 bits are used). The `-iv` option gives the initialization vector for CBC mode, which is the default block cipher mode in openssl for DES. It is also 8 bytes long, because DES blocks are 64 bits. The values given for these options (e.g. abababababababab) are just things that I made up.

A more natural way to use openssl encryption is not to specify a key. In this case openssl will prompt you for a password which it will then translate into a key using a key generation algorithm. Try encrypting and decrypting a couple of messages of your own. Send them to your friends for fun. Here's an example:

```
echo "ATTACK PEARL HARBOR DEC 6" > msg.txt
openssl des -in msg.txt -out msg.cipher
```

You can now write your own diary encrypting program, but be careful: the algorithms really work. If you lose the key, you can't recover the plaintext!

Note that all of the above works for ciphers other than `des`, for example the more secure `aes`.

2 Implementations of DES

In the folder for this lab I've included two implementations of DES. One of them is short and highly optimized. The other is long and designed to be easily recognizable as the DES algorithm. The optimized code has attribution included in the comments (copyright Stuart Levy, 1988). The "simple" implementation is from the book *Implementing SSL/TLS: Using Cryptography and PKI* by Joshua Davies. The book has a lot of prose describing the code which is unfortunately not included in the comments, but hopefully I will be able to scan it for you.

2.1 Simple implementation

In the simple implementation, read the code for the function `des_block_operate`. It should be easily identifiable with the algorithm presented in class. Some parts of it (for example the S-box substitutions) are still a little arcane, even though the code is written for pedagogic use. Note that the only difference between encryption and decryption is an alteration of the key schedule.

The implementation does CBC block cipher mode whether the user wants it or not. This is not a bad thing – you might have absorbed from the textbook that not allowing the user to chose weak security options (such as ECB mode) is good policy.

2.2 Optimized implementation

The optimized implementation is better commented, but more difficult to follow. Like the simple implementation, it is about 300 lines long with comments removed.

This implementation does not have a mode, and provides only a single block encryption in the `fencrypt` method. If the user desires to use a block cipher mode, he or she must write the code. Unlike the simple implementation, the key schedule is pre-computed, not determined on the fly. It is stored in a special data type defined in `des56.h`.

While the simple implementation has a workhorse method called `permute` which performs the various permutations required by DES, this code is more subtle. There is a preprocessing function called `build_tables` which translates the permutations specified by the NIST (aka NBS) standards in such a way that they can be implemented with bitwise operations. For example, consider the implementation of the IP (initial permutation box) which is done with this code in `fencrypt` (line 464).

```
L = R = 0;
i = 7;
ap = wL_I8;
do {
    register int v;
    v = block[i]; /* Could optimize according to ENDIAN */
    L = ap[v & 0x55] | (L << 1);
    R = ap[(v >> 1) & 0x55] | (R << 1);
} while(--i >= 0);
```

This code exploits the pattern in the IP (it is a very non-random permutation) to implement the initial shuffling and partitioning as a series of bit-shifts and bitwise disjunction operations (aka `|`). This is done via the ingenious structure of the `wL_I8` array, and presumably speeds up the algorithm.

You see this same kind of thing happening with the key schedule permuted choice operations (line 413) and in the combined (!) E-box, subkey xor, S-box and P-box actions implemented on lines 490-515 (the heart of the algorithm).

2.3 Challenge

We will first get the code in `des56.c` running and understand its workings. I will help with this in class. The comments at the beginning of `des56.c` explain how to use the code. Encrypt a block of plaintext (for example, 8 ASCII characters) and compare the resulting ciphertext to what you get using `openssl` as described above. Once the outputs match you can be sure that the code is running correctly.

The goal of the project is to expand the code by implementing the following functions:

```
void cbc_enc_three_blocks(char block1[8],char block2[8], char block3[8], char IV[8])
```

The input to this function is three blocks of plaintext, and an initialization vector.

The key is not given as input to this function, because it will already have been used in a separate call to build the key schedule.

The function returns nothing, but the three input blocks will be altered. The blocks of input will become three blocks of ciphertext which are the result of running DES in CBC mode with the provided IV. You can check the correctness of this function using `openssl` from the command line.

Please also implement the reverse operation:

```
void cbc_dec_three_blocks(char block1[8],char block2[8], char block3[8], char IV[8])
```

The input to this function is three blocks of ciphertext, and an initialization vector.

Again, there is no output. But as a side effect of running this method, the input blocks should be transformed into three blocks of plaintext which are the result of running DES decryption in CBC mode with the provided IV. Just as before it is not necessary to specify the key here because it has already been used to build the key schedule.

You should be able to achieve the above functions with a few calls to `fencrypt`.

You can verify the correctness of your code by comparing the output to what you get using the command line `openssl` tool. Also check that encrypting and then decrypting results in the original plaintext.