Lab 1: Exploring Symmetric Key Cryptography

Revised: January 19, 2016

Please read through the whole lab before starting it.

Background

In this lab, you will explore symmetric key encryption by using the One Time Pad and the AES block cipher in different modes of operation. You will explore the properties of these schemes, seeing a visual representation of the state of the ciphertext, and investigate how error propagation during decryption can be used to attack a system. You will also explore the properties of cryptographic hash functions, and find collisions on a truncated range.

Environment

You are free to implement the solutions to this lab using the programming language and cryptographic library of your choosing. However, the instructor's strong recommendation is that you use Python and PyCrypto (or some other high-level language with decent crypto support).

Task I. Warm It Up

Write a function that XOR's two arbitrary-length bit strings together. To check your work, the ASCII string:

```
Darlin dont you go
```

XOR'd with the ASCII string:

```
and cut your hair!
```

should produce the hex value:

```
250f164c0a1b54441601015259071449154e
```

Be sure to throw an error if the two string are of unequal length.

Congratulations! You've just implemented the encryption and decryption algorithm for the One Time Pad!

Task II. Implement OTP

The One Time Pad (OTP) algorithm XOR's the bits of a plaintext with the bits of key to produce a ciphertext. In order to maintain "perfect secrecy" the key must be generated perfectly at random, be as long as the message, and never reused (we'll see the dangers of misuse soon!). Write a program that takes as an input a file to be encrypted and generates a random key (using your system's `/dev/urandom` device will be sufficient), XOR's the key with the plaintext, and writes the resulting ciphertext to a new file. Within the same program, try to XOR the key with the ciphertext and verify that you get the original plaintext back again.

Task III. Encrypt an Image Using OTP

Modify your program from Task II to encrypt a Windows bitmap (BMP) image. Download the two bitmap images (`cp-logo.bmp` and `mustang.bmp`) from the class website. Encrypt both files using the One Time Pad, creating two different ciphertexts. It will be helpful if you name your ciphertexts with a .bmp extension. We will visualize the ciphertexts, tricking an image viewer into thinking they are images again. To do this, you need to do a little pre-processing to make these files viewable.

BMP images have a 54 byte header that tells the image viewer that the file is a legitimate BMP image. You will need to replace the encrypted BMP header in the ciphertext you created with the valid BMP header from the original plaintext. It will be easiest if you do this programmatically (rather than by hand).

Finally, use the 54 bytes of header from one of the images, and fill the remaining bytes with random data from the /dev/urandom device (up to the size of the original images).

View and compare the three images. *What do you observe? Are you able to derive any useful information about from either of the encrypted images? What are the causes for what you observe?*

Task IV. The Two-Time Pad

The One Time Pad is only secure under the conditions given above (the single use of a perfectly random key as long as the message), and becomes grossly insecure if any of those criteria are not met. In this task, you will see how the reuse of the key dissolves confidentiality.

Further modify your program from Task III to encrypt both bitmaps images (`cp-logo.bmp` and `mustang.bmp`) under the same key. Perform the same plaintext header preservation as before.

View the encrypted images. Both images should appear indistinguishable from random.

Now, XOR both encrypted images together, preserving and re-appending one of the headers (either will work). Consider the following property of XOR as you do this:

$$(A \otimes K) \otimes (B \otimes K) = A \otimes B$$

View the output. *Are you able to now derive any useful information about the original plaintexts from the resulting image? What are the causes for what you observe?*

Task V. Modes of Operation

In this task, you will explore the differences in security attained by the ECB and CBC modes of encryption.

Using the AES-128 primitive provided by your cryptographic library[1], implement the ECB and CBC modes of operations (don't cheat and use built-in methods!). Like before, your program should take a (plaintext) file, generate a random key (and random IV, in the case of CBC), and write the encryption of the plaintext to a new file.

Keep in mind, by itself, AES is a block cipher, and expects a 128-bit plaintext message and produces a 128-bit ciphertext – nothing more, nothing less. Putting a block cipher, like AES, into a "mode of operation" is the way we are able to encrypt data larger than 128-bits using a single key. However, a problem arises in certain modes of operation if the files are not evenly divisible by 128-bits. To account for plaintexts that are not an integral size of AES's block size, implement PKCS#7 padding. Details of this scheme can be found here: http://tools.ietf.org/html/rfc5652#section-6.3 but perhaps a more easily understood description is here: http://en.wikipedia.org/wiki/Padding_(cryptography)#PKCS7

Encrypt one of the BMP files using your ECB and CBC implementations, creating two different ciphertexts. (Be sure to the preserve and re-append the plaintext BMP headers!)

Once again, view the resulting ciphertexts. *What do you observe? Are you able to derive any useful information about from either of the encrypted images? What are the causes for what you observe?*

[1] In PyCrypto you can get the primitive object with a call to AES.new() passing in only a key.

Task VI. The Limits of Confidentiality

In this task, you will explore some of the limits of block ciphers in their use within a secure system.

Start by using the PKCS#7 padding and CBC code from Task V to write two "oracle" functions that emulate a web server that wants to use cryptography to protect access to a site administration page.

First, at the start of your program generate a random AES key and IV, which will be used in both of functions, keeping it constant for the execution of your program; *i.e.* don't generate a new key or IV for every encryption and decryption.

The first function, called `submit()`, should take an arbitrary string provided by the user, and prepend the string:

```
userid=321;userdata=
```

and append the string:

```
;session-id=31337
```

For example, if the user provides the string:

```
You're the man now, dog
```

`submit()` would create the string:

```
userid=321;userdata=You're the man now, dog;session-id=31337
```

In addition, `submit()` should: (1) URL encode any ';' and '=' characters that appear in the user-provided string, (2) pad the final string (using PKCS#7), and (3) encrypt the padded string using AES-128-CBC. `submit()` should return the resulting ciphertext.

The second function, called `verify()`, should: (1) decrypt the string, (2) parse the string for the pattern "`;admin=true;`" and, (3) return true or false based on whether that string exists. If you've written `submit()` correctly, it should be impossible for a user to provide input to `submit()` that will result in `verify()` returning true.

Now the fun part: use your knowledge of the way CBC mode works to modify the ciphertext returned by `submit()` to get `verify()` to return true.

Hint: Flipping one bit in ciphertext block c_i will result in a scrambled plaintext block m_i, but, will flip the same bit in plaintext block m_{i+1}.

Why was this attack possible? What would this scheme need in order to prevent such an attack?

Task VII. Exploring Cryptographic Hash Functions

In this task, you will investigate the properties of cryptographic hash functions. You might find Python's hashlib module quite useful.

One-wayness

Start by writing a program that uses SHA256 to hash arbitrary inputs and print the resulting digests to the screen in hexadecimal format. Next, hash two strings (of any length) whose Hamming Distance is exactly 1 bit (*i.e.* differ in only 1 bit). Repeat this a few a times.

What do you observe? How many of the bytes are different between the two digests?

Preimage Resistance

Write a second program that calculates the number of inputs required to find a SHA256 hash digest that is less than each of the following hexadecimal target values[2]:

```
0x0FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
0x00FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
0x000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
0x0000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
0x00000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```

For each target, your program should output the number of inputs, the target value, and the computed digest.

What do you observe about the number of inputs required?

Collision Resistance

Now, let's try to find two strings that create the same digest (called a collision). Because SHA256 is a secure cryptographic hash function (as far as we know), it's not feasible to use its full 256-bit output. Instead, you will limit its domain to between 8 and 50 bits.

[2] These targeted collision problems are exactly how Bitcoin's "proof of work" algorithm works for miners.

Modify your program to compute SHA256 hashes of arbitrary inputs, so that it is able to truncate the digests to between 8 and 50 bits (it doesn't matter which bits of the output you choose, so long as you're consistent). Once you have completed the above, try to find a collision in your truncated hash domains (*i.e.* two different inputs, that create the same, truncated digest) . There are at least two different ways of doing this (you need only do one):

1. Find a target hash collision (second preimage resistance): Given m_0, find m_1 such that $H(m_0) = H(m_1)$ and $m_0 \neq m_1$. This is easier to code, but will take "longer" to execute[3].
2. Maximize your chances of finding a collision by relying on the Birthday Problem: For *any* two messages m_0, m_1 where $m_0 \neq m_1$, find $H(m_0) = H(m_1)$. This requires a little more code (and memory usage), but will find a collision more quickly. Consider using a hashtable or dictionary, but be careful about efficiency as finding collision on 50-bit outputs is right at the edge of what's feasible by an average computer.

For multiples of 2 bits (*i.e.* for digests sized 8, 10, 12, ..., 48, 50 bits), measure both the *number of inputs* and the *total time* for a collision to be found. Create two graphs: one which plots digest size (along the x-axis) to collision time (y-axis), and one which plots digest size to number of inputs. *Include these graphs in your report.*

What is the maximum number of files you would ever need to hash to find a collision on an n-bit digest? Given the Birthday Bound, what is the expected number of hashes before a collision on an n-bit digest? Is this what you observed? Given the data you've collected, speculate on how long it might take to find a collision on the full 256-bit digest.

Given an 8-bit digest, would you be able to break the one-way property (i.e. can you find any *pre-image)? Do you think this would be easier or harder (i.e. more or less work) than finding a collision? Why or why not?*

Submission

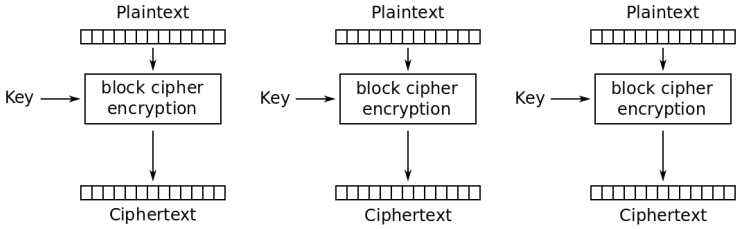
Write a brief report describing what you did and what you observed. Include any code that you wrote, as well as answers to any questions (these are in italics). Please include any explanations of the surprising or interesting observations you made. Write at a level that demonstrates your technical understanding, and do not shorthand ideas under the assumption that the reader already "knows what you mean." Think of writing as if the audience was a smart colleague who may not have taken this class. Describe what you did in sufficient detail that it can be reproduced. Please do not use screenshots of the VM to show the commands and code you used, but rather paste (or carefully re-type) these into your report from your terminal. Submit your completed write up to PolyLearn in PDF format.

[3] Back of the envelope calculations put finding a collision in this way at around 156,000 hours at 1,000,000 hashes per second.

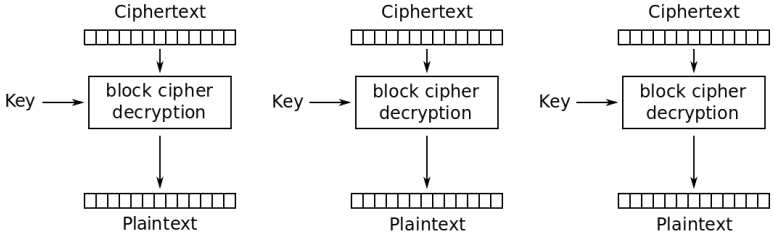
Modes of Operations

The modes of operation explored in this lab (ECB and CBC) are all NIST standards. We summarize their block diagrams (from Wikipedia).

Electronic Codebook Mode (ECB)

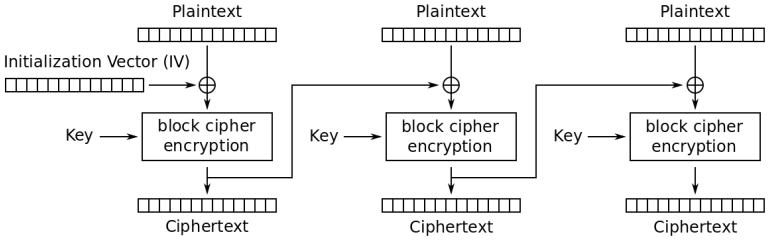


Electronic Codebook (ECB) mode encryption

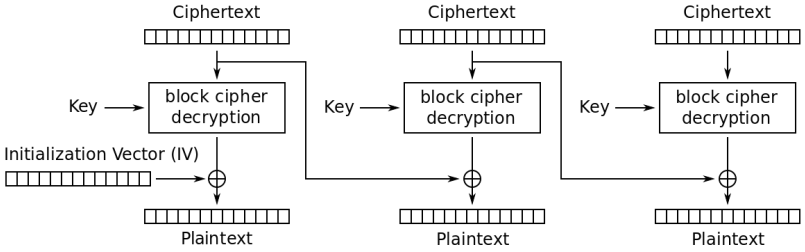


Electronic Codebook (ECB) mode decryption

Cipher Block Chaining Mode (CBC)



Cipher Block Chaining (CBC) mode encryption



Cipher Block Chaining (CBC) mode decryption