Symmetric Ciphers: Stream and Block Ciphers

Annabel Su

December 2020

1 Introduction

In class, we learned about the RSA encryption scheme, which was asymmetric in that it used different keys (public and private) to encrypt and decrypt. Here, we will discuss another family of ciphers, **symmetric ciphers**. A key that is used to both encrypt and decrypt is called a **symmetric key**, and ciphers that use symmetric keys are known as symmetric ciphers. Two of the most famous kinds of symmetric ciphers are stream and block ciphers. We will go over both of these ciphers and how they work, as well as study examples of each.

There are Python implementations of the examples here that you can use to check your understanding:
`https://github.com/annabelsu/15251-crypto-project`

2 Stream Ciphers

2.1 What are stream ciphers?

Given a message/plaintext M, stream ciphers use a pseudo-random keystream K to encrypt one digit of M at a time. In practice, digits of M are generally bits, and each bit is encrypted by XOR-ing it with the corresponding bit of K. "One time pad" is an example of a stream cipher. A popular example of another stream cipher is RC4, created by Ron Rivest (from RSA) and noted for its speed and simplicity.

2.2 RC4

RC4 differs from one time pad in that it generates its own keystream, but the user still has to supply a secret key that is used to generate the keystream. The keystream is generated in two parts, the Key Scheduling Algorithm (KSA) and the Pseudo-random Byte Generation Algorithm (PRGA).

2.2.1 KSA

The user gives a plaintext `M` and a key `key`.

We initialize an array `S` of 256 numbers, 0 to 255, in that order.

Initialize an index `j = 0`.

As we iterate through each index `i` for `S`, we update `j` by adding `S[i]` and `key[i % length(key)]`, then modding by 256. Then we swap the value of `S` at `i` and the new `j`.

At the end of KSA, the contents of `S` should appear random.

2.2.2 PRGA

Each iteration of PRGA outputs one byte of the keystream, so the number of iterations depends on the number of bytes in the input plaintext, since the length of the keystream needs to be the same.

We first initialize two index variables, `i = 0` and `j = 0`. For each iteration:

Keep incrementing `i` by one (mod 256).

Increment `j` by `S[i]` (also mod 256).

Swap the values of `S` at indices `i` and `j`.

Add the values of `S` at `i` and `j` (mod 256) to get a new index, `t`.

Output the byte at `S[t]`.

Once the full keystream is generated, it is XOR-ed with the plaintext to produce the ciphertext. Decryption is done in the same manner by XOR-ing the ciphertext with the keystream.

2.2.3 Check Your Understanding

Based on the specification above, try implementing RC4 encryption and decryption. Some things to keep in mind:

- You may want to consider using the `yield` operator in the PRGA step.
- If you want to encrypt/decrypt strings, you will want to convert your plaintext and key to numbers before doing any numeric operations with them. You can do this with the `ord()` function.

You can check your implementation with the code in `rc4.py` in the Github repository.

3 Block Ciphers

3.1 What are block ciphers?

While stream ciphers encode one bit at a time, block ciphers encode in fixed-size chunks called blocks. Suppose you have a message M with n bits (n is your block size). To use a block cipher, you want to encrypt M with a symmetric key K with k bits. A block cipher will use K to permute M. Thus, another name for block ciphers are Psuedorandom Permutations (PRPs).

One property of PRPs is that it provides a one-to-one mapping. Thus, since there exists an efficient encryption algorithm with a PRP, there also exists an efficient decryption algorithm, the inversion function.

There are several different designs for block ciphers, including an iterated design, substitution-permutation networks, and Feistel ciphers, among others, but here we will focus on these three.

3.2 Iterated Block Ciphers

Most block ciphers follow an iterative design. In order to encrypt a plaintext M with a key K, a PRP called the round function is applied multiple times, with each iteration being called a "round." The number of rounds depends on the specific cipher you are using. So, given a round function R that takes in an M and a K, iterated block ciphers work as follows.

The original key K is expanded into multiple round keys K_i, typically one for each round. How the key is expanded depends on the specific cipher, but this process is called the **key schedule**. The round function takes in the plaintext M_0 and the first round key K_1 and produces a modified plaintext M_1. It then takes the new plaintext and encrypts it with the second round key. This continues until every round is complete. More formally, $M_i = R(M_{i-1}, K_i)$.

After all the rounds are done, we are left with M_r, where r is the number of rounds. M_r is also known as the ciphertext.

3.3 Substitution-Permutation Networks (SPNs)

Substitution-permutation networks are a type of iterated block cipher, where we alternate between substitution (S-boxes) and permutation (P-boxes).

Individually, S-boxes and P-boxes are not very powerful or complicated. In fact, they do exactly as their names suggest. An S-box takes an input plaintext and substitutes bits to produce the output. In some designs, the key is also used to substitute bits. It is also common to use multiple S-boxes to encrypt each sub-block. In general, however, you want your S-box to have the avalanche effect (i.e. a slight change in input can lead to a drastic change in output).

Likewise, a P-box takes the outputs of the S-boxes and permutes them. A good P-box will have the property that the output bits of any S-box are distributed across as many other S-boxes as possible.

In each round, the round key is applied to the current plaintext, perhaps through an operation like XOR. It is important to note that all of these operations (substitution, permutation, and applying the round key) must be invertible to allow for decryption.

Note that the combination of the S and P-boxes through multiple rounds causes SPNs to satisfy Shannon's confusion (substitution hides the relationship between the ciphertext and the key) and diffusion (permutation hides the relationship between the ciphertext and the plaintext) properties.

Below, in section 3.6, we will look at one of the most famous SPNs, the Advanced Encryption Standard (AES).

3.4 Feistel Ciphers

Feistel ciphers, also known Feistel networks, are another kind of iterated block cipher. Within each round, the plaintext is split into two halves, L_i and R_i. The round function F is run on one half with the round key, and then the output of F is XOR-ed with the other half. Finally, the two halves are swapped for the next round. More formally, and $L_{i+1} = R_i$ and $R_{i+1} = F(R_i, K_i) \oplus L_i$.

When L_i and R_i are not the same length, it is called an unbalanced Feistel cipher.

Decryption is essentially the same process, but the order of the round keys is reversed. A big advantage that a Feistel network has over an SPN is that the round function F need not be invertible. Remarkably, however, even if F is not invertible, the entire process remains invertible. This means that F can be made arbitrarily complicated, and that implementing a Feistel network is much simpler.

Fun fact: Although not a direct successor to RC4, RC5 (another cipher designed by Ron Rivest), uses a Feistel structure.

3.5 Tiny Encryption Algorithm (TEA)

TEA uses a Feistel-like structure block cipher. It normally has 64 rounds, implemented in pairs called cycles, and uses a 128-bit key with 64-bit blocks.

The key schedule for TEA is very simple: it breaks the key into four 32-bit parts and simply cycles through them, two in each round. Due to rounds being implemented in pairs, all four keys are used in every cycle, in the same order.

TEA also makes use of a magic number, usually called delta. Delta is `0x9E3779B9`, or the floor of $2^{32}/\phi$, where ϕ is the golden ratio.

TEA first splits the plaintext into `(L,R)` and initializes a running sum variable `sum = 0` before iterating over the cycles. Here is the encryption algorithm for TEA in Python:

```
def encrypt(M, K):
    delta = int("9E3779B9", 16)
    (L, R) = M[0], M[1]
    sum = 0
    for i in range(0, 32):
        sum += delta
        L += ((R << 4) + K[0]) ^ (R + sum) ^ ((R >> 5) + K[1])
        R += ((L << 4) + K[2]) ^ (L + sum) ^ ((L >> 5) + K[3])

    return [L, R]
```

3.5.1 Check Your Understanding

Given the encryption algorithm, try implementing the decryption algorithm for TEA. You can check your work with the `tea.py` file in the GitHub repository.

3.6 Advanced Encryption Standard (AES)

AES is a subset of the Rijndael ciphers, using a substitution-permutation network. It has a fixed block size of 128 bits, and a key size of 128, 192, or 256 bits. Practically speaking, there is no reason to use 192-bit over 128-bit or 256-bit, except for negligible speed gains. Most AES applications today use 128 or 256 bits.

Depending on the size of the key, the number of rounds are also different: 10 rounds for 128-bit, 12 rounds for 192-bit, and 14 rounds for 256-bit. Here, we will work with a 128-bit key.

In AES, we express the plaintext as a 4x4 matrix of bytes, called the state array. Since the plaintext is 128-bits, we can break it down into 16 bytes, and write it into a matrix like follows:

$$\begin{bmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_1 & b_5 & b_9 & b_{13} \\ b_2 & b_6 & b_{10} & b_{14} \\ b_3 & b_7 & b_{11} & b_{15} \end{bmatrix}$$

We also expand the key into 44 words (each 4 bytes) through the AES key schedule, which we will go deeper into later. Before the iteration of the rounds begin, we XOR the state array with the first 4 words. The remaining 40 words are used as round keys, 4 words per round. The first 9 rounds consist of four steps: `SubBytes`, `ShiftRows`, `MixColumns`, and `AddRoundKey`. The last round does not include the `MixColumns` step.

3.6.1 `SubBytes`

The first step of a round is the substitution step. For each byte, we will substitute it out with another byte from a 16x16 lookup table called the S-box. Each entry of the S-box is a byte represented in hexadecimal. The S-box remains the same across all rounds, and is generated as follows.

Since the row and column indices of the table go from 0 to 15, we can use hexadecimal to represent the indices. For the cell at row R and column J, the value in the cell is initialized to IJ. For example, the cell at (5, A) would have value 0x5A.

After initializing the S-box, we replace each entry with its multiplicative inverse in the Galois field $GF(2^8)$. The number 0x00 stays the same, since it does not have an inverse.

The final step is to perform some bit-scrambling on each entry. Each entry byte $b_7b_6b_5b_4b_3b_2b_1b_0$ is transformed through the following:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \times \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix}$$

Note that the addition operations are actually XOR operations. You may also notice that there seems to be an arbitrary vector representing 01100011, or 0x63, thrown in. This vector is commonly noted as c.

In order for `SubBytes` to be invertible, we need to create a one-to-one mapping, which the multiplicative inverses give us. We do not want any byte to map to itself, since that would be a weakness for AES. Unfortunately, since we leave 0x00 unchanged in the first step, if we simply bit scrambled without c, 0x00 would still map to itself. With c, 0x00 no longer maps to itself, but the one-to-one mapping is still preserved.

3.6.2 `ShiftRows`

In this step, we leave the first row of the state array unchanged, shift the second row to the left by one, shift the third row to the left by two, and shift the last row to the left by three. Visually, it looks like this:

$$\texttt{ShiftRows}\left(\begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix}\right) \Longrightarrow \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,1} & s_{1,2} & s_{1,3} & s_{1,0} \\ s_{2,2} & s_{2,3} & s_{2,0} & s_{2,1} \\ s_{3,3} & s_{3,0} & s_{3,1} & s_{3,2} \end{bmatrix}$$

3.6.3 `MixColumns`

Each byte in a column is updated by a function of the whole column. More specifically, each byte is updated by 2 times that byte plus 3 times the next byte plus the next two bytes. The concept of the "next" byte wraps around the column, i.e. the next byte after the last byte in the column is the first byte. You can update each column of the state array using the following matrix:

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \times \begin{bmatrix} s_{0,j} \\ s_{1,j} \\ s_{2,j} \\ s_{3,j} \end{bmatrix} = \begin{bmatrix} s'_{0,j} \\ s'_{1,j} \\ s'_{2,j} \\ s'_{3,j} \end{bmatrix}$$

It is important to note that the multiplication and addition operations are done in $GF(2^8)$ arithmetic. We will not go into details about this, since finite field arithmetic was not covered in 15-251, but essentially what this means is that addition is XOR and multiplication is modulo $x^8 + x^4 + x^3 + x + 1$, and the matrix entries are treated as coefficients of a 7th degree polynomial.

3.6.4 AES Key Expansion

Before we discuss the last operation, `AddRoundKey`, we need to first look at how round keys are derived and how they are expressed. Like how we rewrote our plaintext as a 4x4 matrix, we can also do the same for our key:

$$\begin{bmatrix} k_0 & k_4 & k_8 & k_{12} \\ k_1 & k_5 & k_9 & k_{13} \\ k_2 & k_6 & k_{10} & k_{14} \\ k_3 & k_7 & k_{11} & k_{15} \end{bmatrix}$$

We take each of the columns in this matrix to create our first four words: k_0 to k_3 make up w_0, k_4 to k_7 make up w_1, k_8 to k_{11} make up w_2, and k_{12} to k_{15} make up w_3. These four words are XOR-ed with the state array before the first round.

We also use these four words to generate 40 more words.

Suppose we have a set of four words w_i, w_{i+1}, w_{i+2}, and w_{i+3} and we want to use them to generate the next set of four words, w_{i+4}, w_{i+5}, w_{i+6}, and w_{i+7}. We first take w_{i+3} and apply the following steps to it:

- Rotate left (circular) by one byte
- Substitute each byte using the same lookup table from the `SubBytes` step
- XOR the bytes with the round constant for that round

A **round constant** is a 4-byte word whose rightmost three bytes are all zeroes. The round constant for the ith round is usually denoted as $Rcon[i] = (RC[i], \texttt{0x00}, \texttt{0x00}, \texttt{0x00})$, where $RC[i]$:

- $RC[1] = \texttt{0x01}$
- $RC[i] = \texttt{0x02} \times RC[i-1]$

After applying these three steps to w_{i+3}, we XOR the resulting word with w_i to produce w_{i+4}. The remaining three words of the new set are created as follows:

- $w_{i+5} = w_{i+1} \oplus w_{i+4}$
- $w_{i+6} = w_{i+2} \oplus w_{i+5}$
- $w_{i+7} = w_{i+3} \oplus w_{i+6}$

Given that we have just derived 40 words for our 10 rounds, each round key will use 4 words. Since each word has 4 bytes, each round key has 16 bytes, so we can express it as a 4x4 matrix, like how we did for our plaintext and original key.

3.6.5 `AddRoundKey`

Since we express both the state array and round keys as a 4x4 matrix, applying the round key is simple: we just XOR each entry in the state array with the corresponding entry in the round key.

This concludes the encryption process of AES. You may have noticed that it is by far the most complex of the examples we have looked at, for good reason. AES is still used as an industry standard for data encryption, and has been adopted worldwide after the U.S. government selected it as the successor to the previous standard, **Data Encryption Standard (DES)**.

Fun fact: Ron Rivest created another cipher, RC6, that was also a candidate to become DES' successor.

3.7 AES Decryption

Althought AES decryption is a similar process to encryption, it is not as simple as just doing things in reverse.

For decryption, the round keys remain the same, but we reverse the order in which we use them. This means that before starting our rounds, we XOR the ciphertext with the last four words, w_{40}, w_{41}, w_{42}, and w_{43}.

One decryption round consists of the following steps: `InvShiftRows`, `InvSubBytes`, `InvAddRoundKey`, then `InvMixColumns`. Similar to encryption, the last round does not include the `InvMixColumns` step.

3.7.1 `InvShiftRows`

We shift the rows of the state array in the same way, leaving the first row unchanged, and incremementing the offset for each of the following rows, but this time shifting right instead of left.

3.7.2 `InvSubBytes`

In `InvSubBytes`, we first bit scramble, and then find the multiplicative inverses. The bit scrambling is as follows:

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \times \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix}$$

Again, the addition operations are all XOR operations. Additionally, instead of c, we now use a vector called d, which is 00000101 or 0x05.

After bit scrambling, we simply find the multiplicative inverse for each cell, again in $GF(2^8)$.

3.7.3 `InvAddRoundKey`

Like `AddRoundKey`, we simply XOR the state array with the round key in this step. Remember that the order of the round keys is reversed.

3.7.4 `InvMixColumns`

The matrix used for the inverse is:

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \times \begin{bmatrix} s_{0,j} \\ s_{1,j} \\ s_{2,j} \\ s_{3,j} \end{bmatrix} = \begin{bmatrix} s'_{0,j} \\ s'_{1,j} \\ s'_{2,j} \\ s'_{3,j} \end{bmatrix}$$

3.7.5 Check Your Understanding

One round of encryption is `SubBytes`, `ShiftRows`, `MixColumns`, and `AddRoundKey`. However, one round of decryption is not just the reverse of these steps, `InvAddRoundKey`, `InvMixColumns`, `InvShiftRows`, and `InvSubBytes`, but instead `InvShiftRows`, `InvSubBytes`, `InvAddRoundKey`, then `InvMixColumns`. Why is this the case?

Solution: Instead of thinking in terms of rounds, we can think about the sequence of steps for encryption one by one, including the first XOR of the state array (which is just `AddRoundKey` with the first four keys). If we do the same for decryption, also including the first XOR (which is just `AddRoundKey` with the last four keys), we see that the decryption sequence is the reverse of the encryption sequence.

3.7.6 Check Your Understanding

Implement AES encryption and decryption. Some notes:

- Since we haven't talked about modes of operation or padding, you want your blocks and keys to be exactly 128 bits.
- If you want to work with strings, it is a good idea to convert them to a 16-byte hex number, and then encrypt/decrypt that hex number.
- The substitution boxes and round constants are given to you in `aes.py`, so no need to generate them.

You can check your implementation with the `aes.py` file in the Github repository.

3.8 Modes of Operation

At this point, you may have realized that having a fixed block size only allows you to work with a very limited amount of information. What if we want to be able to encrypt data beyond a certain number of bits? In order to encrypt a variable length message, we use different **modes of operations**, which act as guidelines on how to repeatedly apply a block cipher to encrypt multiple blocks of data.

Simply partitioning the data into separate blocks and encrypting each one individually is not very secure, since identical blocks will result in identical ciphertexts, thus exposing patterns in the plaintext. We want to randomize the ciphertexts, and most modes of operation achieve this through something called an initialization vector (IV).

An **initialization vector**, also called a starting variable, is a non-repeating, sometimes random, binary sequence. In order to randomize the ciphertexts, we randomize the input data. After encrypting one block, most modes of operations will take the ciphertext for that block and apply it somehow to the next block of plaintext, before encrypting that next block. For the first block of plaintext, since we lack a block of prior ciphertext, we use the IV instead.

3.8.1 Padding

Since we are now working with variable length plaintexts, when partitioning into blocks, the final block may not be exactly the block size. Some modes require that a block is exactly the block size, in which case padding is needed.

There are several ways to pad, dependent on which mode is in use, but some examples include filling in zeroes at the end or filling in a one and then all zeroes at the end. These kinds of methods can result in extra, unnecessary ciphertext.

Another method that does not create extra ciphertext is something called **ciphertext stealing**. Ciphertext stealing takes a portion of the ciphertext from the second to last block and uses it to pad the last block.

3.8.2 Common Modes

There are numerous modes of operation, but since we just covered AES, we will go over some common modes of operation for AES.

Electronic Codebook (ECB): simply splits the plaintext up into blocks and encrypts them individually using the same key. Because it exposes patterns in the plaintext, it is not recommended.

Cipherblock Chaining (CBC): XORs a block of plaintext with the ciphertext from the previous block (or the IV for the first block) before encryption. Generally recommended.

Cipher Feedback (CFB): Encrypts the ciphertext from the previous block (or the IV for the first block), then XORs the result with the plaintext to create the ciphertext. Generally recommended.

Counter (CTR): Uses a nonce (an arbitrary number) combined with a counter. Encrypts this number and XORs the result with the plaintext to create ciphertext. Blocks are encrypted individually. Generally recommended.

3.8.3 Check Your Understanding

Why do we use an IV? What benefit does this have over encrypting with multiple keys?

What do all the recommended modes of operation have in common?

What benefit does ciphertext stealing have over simply adding in bits at the end?

What is a possible problem with simply padding in zeroes at the end? How does adding in a one bit before the zeroes overcome this?

4 Sources

`https://en.wikipedia.org/wiki/RC4`

`https://www.cs.purdue.edu/homes/ninghui/courses/Fall05/lectures/355_Fall05_lect13.pdf`

`https://www.coursera.org/lecture/crypto/what-are-block-ciphers-t4JJr`

`https://en.wikipedia.org/wiki/Block_cipher`

`https://en.wikipedia.org/wiki/Tiny_Encryption_Algorithm`

`https://engineering.purdue.edu/kak/compsec/NewLectures/Lecture8.pdf`

`https://en.wikipedia.org/wiki/Advanced_Encryption_Standard`