AES Design Documentation

Language Choice

For this AES implementation I chose to use Python 3.8.0 as my programming language. This was for a few reasons:

- The ‘bytes’ data type allows for easy manipulation of binary. A normal integer array would not throw an error if one of the bytes was set to 500, for example, while an element in a ‘bytes’ object is locked to the range [0, 255].
- List slicing (through the syntax array[start, end]) is extremely useful in many sections of this implementation for clarity. It returns the elements of ‘array’ between ‘start’ and ‘end’, including ‘start’ and excluding ‘end’.
- Iterator-based for-loops enable readability. ‘for index, byte in enumerate(state):’ is much clearer to read than ‘for(i = 0; i < state.len(); i++) { byte = state[i]; ... }’, the C++ equivalent.
- It is the language I have the most experience in.

Structure

I chose to structure the implementation into an ‘AES’ class that contains the internal logic for the algorithm, and outside cipher/decipher functions for ease of use. When executed, the program goes through a couple prompts for the information, and whether you want to encrypt or decrypt. The output of the program when encrypting is a hexadecimal representation of the ciphertext. The output of the program when decrypting is either a hexadecimal representation of the plaintext, or readable plaintext if possible.

Cipher / Decipher

The cipher / decipher functions are based on the pseudocode in the AES specification, FIPS 197. Key differences are that instead of a sequence of words, the key schedule is a sequence of bytes, and sliced into round keys of 4 words when necessary. I attempted both a sequence of words and a sequence of bytes as my key schedule, but the more readable of the two was the sequence of bytes because of Python's sequence slicing. I also keep the state as a sequence of bytes throughout the program, turning it into a state matrix and back when needed to improve code readability. Also to improve readability, I expanded Nb, Nr, and Nk into block_size, num_rounds, and key_size respectively. This makes the meaning of each variable much clearer. Beyond initializing an instance of the AES class, it should look similar to the standard's pseudocode.

SubBytes / InvSubBytes

These two were very straightforward to implement. Each byte '0xXY' is separated into 'X' and 'Y' using a combination of floor division and modulus with 0x10. Both the normal and inverse substitution boxes are located at the beginning of the program, encoded as matrices matching the specification.

ShiftRows / InvShiftRows

At the beginning of this function, the state sequence is turned into a state matrix by the helper function 'make_matrix(state)'. Each row of the matrix is then shifted right based on its index. To improve readability, I moved the sequence rotation section into its own method 'rotate(sequence, amount=1, reverse=False)'. The implementation of

‘rotate’ rotates the sequence left normally, so ‘reverse=True’ has to be passed in the inverse version to shift the rows to the right. The state matrix is then converted back into a sequence of bytes and passed along.

MixColumns / InvMixColumns

For this method, a helper function named ‘make_column(state)’ was created to get each column of the state matrix as a list to simplify the implementation. For the multiplication XOR acts as addition in GF(256), but finite-field multiplication is a bit more complicated. An xmult(a, b) method was created to perform the finite field multiplication for each element. The code is based on the Russian Peasant Multiplication Algorithm found at https://en.wikipedia.org/wiki/Finite_field_arithmetic. Some modifications were made to make it easier to read.

AddRoundKey

For add_round_key, the state and round key are both converted into a sequence of their columns, and each element of the state column is XORed with its corresponding round key value. The column sequence form was used to simplify the implementation. They state is then converted back into a sequence and passed along.

Key Expansion

Because of the choice to use byte sequences instead of, the key expansion method became the most difficult one to implement, due to the pseudocode’s reliance on words. In order to keep the implementation similar to the pseudocode, the key schedule is converted to a sequence of words for the expansion routine, then converted back to a sequence of bytes for use in the rest of the implementation. For the round

constant (Rcon[i/Nk] in the specification), I generate the sequence when the AES method is first initialized, in the '__init__()' method.

Since python doesn't have support for the XOR (^) operation between two lists, I created two helper functions. The 'word_xor' function takes two lists (or words) and xors their contents, then returns the result. The 'constant_word_xor' takes a word and the round constant. It converts the round constant to the form [const, 0, 0, 0], then passes it to the 'word_xor' function, as this is how the round constant is added in the AES specification.

Testing

A separate test suite was implemented, used to test every function in the implementation to make sure it runs correctly. The values used for testing were found from the AES specification, and a powerpoint located at https://kavaliro.com/wp-content/uploads/2014/03/AES.pdf. This sped up development time dramatically, as I could be confident that each change did not break a different part of the program, and I could know exactly what parts of the implementation were not working.

Running

This program was tested using Python 3.8.0 on Linux. It can be run with 'python3 aes.py' for the main program, and 'python3 test_aes.py' for the test suite. The test suite is found inside of the zip file, as is a copy of the AES specification and assorted project files.