Final Project Report

Zach Stecher, Thusharika Nuthalapati, Bhargavi Madhunula

5/3/16

For our final project, our group decided to expand the already created 128-bit AES algorithm to include 192- and 256-bit encryption, plus accomodate any size plaintext message and the decryption of that message. In order to handle this, modifications had to be made to the existing AES program, as well as the writing of the decryption algorithm and modification to the driver file to handle different sized plaintexts.

Inside our AES file, modifications were made to remove the original function of printing the ciphertext straight from the encryption method. This is a twofold improvement as the original intention was simply to demonstrate that it encrypted correctly. The current version allows us to pass the result back to the driver file for further manipulation and decryption. Additionally, the original program contained hardcoded values for 128-bit encryption, such as the number of rounds, size of the expanded key matrix, and secret key matrix. These were modified to accomodate keys up to 256-bit in length by initializing the matrices to the highest possible necessary space needed, and restricting the use to only what was necessary. A variable for "rounds", "columns" and "keySize" each was initialized, and used to store the number of columns needed and number of rounds needed based on the keySize, which was computed as simply the bit length of the hex key (passed as a String) divided by 2. No other major modifications were made to this file as we decided to focus primarily on getting the decryption and block cipher mode to work.

For our decryption file, we mostly re-used the methods from AES, as the decryption process boils down to just being the encryption process in reverse with only three major differences. The first is that the NibbleSub method now utilizes Rijndael's inverse S-box for lookups, though the program still required the original S-box as well for the round key generation. The

second is that the finite Galois field used for MixColumns contains different values. As such we needed to instantiate different lookup tables and switch some numbers around in the MixColumns method, but no major structural changes were necessary. The final difference was probably the most difficult to solve. Because decryption runs in reverse, modifications were necessary so that the round keys would be read from their matrix in reverse. While simple in an abstract sense, it was more tricky than originally thought to accomplish in code.

Lastly, the driver file needed to be updated to accomodate messages of any size. For the purposes of time restrictions and having a working program to demo, we decided to settle on ECB. While we would never use this mode in a real-world setting, it seemed sufficient enough to demonstrate a functional padding and cipher-mode procedure. Two methods - ECBencrypt and ECBdecrypt - were added to the driver file to handle this. The ECBencrypt method measures the length of the plaintext, determines if padding is necessary, and if so appends the necessary number of padded bytes onto the end of the message before encrypting. The resulting ciphertext will always be a multiple of the block size (16 bytes) and the use of non-zero padding helps to prevent repetitions within the blocks. ECBdecrypt takes that ciphertext and runs it through our decryption algorithm. This will result in a plaintext with the plain padding appended onto the end as a multiple of the 16-byte block size. The driver program then reads the size of the original message and trims bytes off the end until the padding has been removed.

While the code is functional on its face, there are numerous improvements to be made in any future attempts at implementation. The initial speed issue with continuous parsing between integer and String data types was never resolved, though not for lack of trying. We tried replacing the instances of String storage within the program with integer storage, as integers were the data type most being manipulated anyway, and then parsing the end result into hex values. Unfortunately, this did not give us the same result, and we could not figure out the issue within the time constrant. There are also areas of unnecessary code repetitions or code bloat that should be refactored to flow more elegantly. The more important unresolved issue lies within the block cipher mode decryption handling. We were unable to come up with a way to handle arbitrary padding - I.E. our code could be padded but could also not be padded - in such a way that the decryption would always be accurate. We tried reading the last byte of the last block, but that would not account for no padding. The idea to pad an entire block onto the end should no padding

be required (so that there would always be padding, but would pad a full block if no padding were actually required) was also discussed, but we felt that would cause repetitions in the ciphertext.

Some other improvements should be made to this code in future iterations, such as appropriate error handling and checks within the driver file to prevent incorrect key inputs. As of now, if a user attempted to use a key with a length other than 128-, 192- or 256-bit, the program will error out. This can be solved with a simple check, but ideally the program would generate its own pseudorandom key based on the user's key size selection. The program could also be modified to utilize inheritence and polymorphism to further refactor; since the encryption and decryption procedures use many of the same methods, they should be placed in a parent class to be used by both rather than have two copies of each method.

Overall, despite the improvments that can and should be made, our group was able to successfully implement AES encryption and decryption for 128-, 192- and 256-bit keys, handle messages of arbitrary sizes, and successfully implement a padding strategy more secure than zero padding.