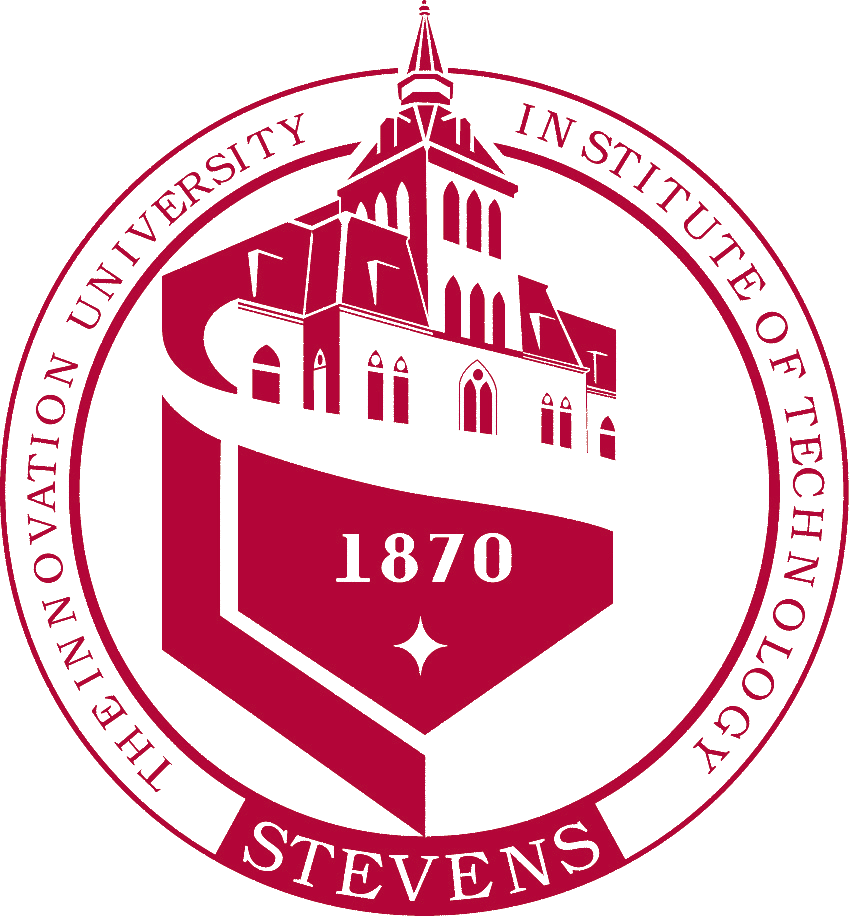
CPE-593 Term Project

Stevens Institute of Technology



**Big Integer Implementation of**

**AES Encryption**

Group 14

Fall Semester 2020

# Abstract:

As the final project for CPE-593, applied data structures and algorithms, Group 14 has chosen to implement a successful encryption/decryption algorithm. The Advanced Encryption Standard (AES), otherwise known by its original name of the Rijndael algorithm, has been adopted by the United States government and is currently the worldwide standard for encryption. Other encryption systems do exist, such as the Data Encryption Standard (DES) or the Rivest-Shimir-Adleman (RSA) algorithms. However, due to the most practical use stemming from AES encryption, the group has decided to take that route for implementation into their final project. A successful implementation of an AES encryption system would ultimately require the group to research and implement several well known algorithms. These algorithms include the Miller-Rabin primality test, Diffie-Hellman key exchange and the Rijndael algorithms. In addition to these, the project required the use of hash tables and other various skills developed in CPE-593. The following report describes the process and results for the group’s big integer implementation of AES encryption in the C++ language.

# Introduction:

The implementation of encryption in technology has existed since the 1970’s in the USA. Back then, the standard was known as Data Encryption Standard, or DES for short, and was implemented normally within many computers in order to protect customer data. However, it was deemed inadequate starting in the late 1990’s. The primary reason for this was because the length of the key for the algorithm was considered too short to be secure enough for the technology and the cracking methods at the time.

In modern day, newer methods with larger keys are being used. Two types in particular, RSA (Rivest-Shimir-Adleman) and AES (Advanced Encryption Standard), have risen up to be the global standards for encryption. In fact, AES has been adopted by the United States government and is currently the worldwide standard for encryption. One defining aspect that makes these algorithms much better than DES is that RSA and AES use a higher total of bits in order to make a key. Three different variants of AES encryption exist. These methods are known as AES-128, AES-192 and AES-256, where each uses key lengths of 128, 192 and 256 bits, respectively. This makes the decryption process much more difficult to crack or to randomly guess and check with simple computer programs. For this project, the AES-256 encryption will be the primary focus. As a result, adjustments will need to be made as the largest value currently allowed in C++ has a length of only 64 bits. AES encryption is a symmetric-key algorithm, where the same key will be used for both encryption and decryption of messages. This differs completely from RSA, as this method uses two separate keys for encryption and decryption. Therefore, the group believed that it would be a much more feasible project to implement an AES encryption system over RSA encryption. The following describes the methods of how the group utilized various algorithms and ultimately implemented a successful working encryption/decryption algorithm.

# Diffie-Hellman Key Exchange:

In order to reproduce this particular algorithm for a key exchange, we decided to create a class in order to help simulate a real example of private vs. public information. To help explain what information will be held private vs. be put out to the public, we can refer to the standard example of Alice, Bob, and Eve. Alice and Bob represent the two people exchanging messages and Eve is someone who is listening in on Alice and Bob's exchange. The following information in Table 1 represents the data needed to be generated through the Diffie-Hellman key exchange, along with who has access to each piece of information.

*Table 1: Diffie-Hellman Information Table*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Value** | **Calculated By** | **Known By:** | | |
|  |  | **Alice** | **Bob** | **Eve** |
| Public Prime Base (g) | rand() + MillerRabin() | Yes | Yes | Yes |
| Public Prime Modulus (p) | rand() + MillerRabin() | Yes | Yes | Yes |
| Alice Private Key (a) | rand() | Yes | No | No |
| Bob Private Key (b) | rand() | No | Yes | No |
| Alice Public Key (A) | ga mod p | Yes | Yes | Yes |
| Bob Public Key (B) | gb mod p | Yes | Yes | Yes |
| Shared Private Message (s) | Ba mod p  OR  Ab mod p | Yes | Yes | No |

All random generation of values was done using the built in <rand> library and the respective rand() function. For the values of p and g, we needed to generate a random prime number, where p is generally greater than g. To do this, we implemented a function to randomly generate numbers and check if it’s prime using the Miller-Rabin primality test, which has a time complexity of O(k\*(log n)2), where k is the number of trials tested. In the group’s implementation, 2 trials are used. The main advantage of using the Miller-Rabin test is that this test has the capability of testing for Charmichael numbers. These numbers are values that may result as prime in other prime number tests, even though the numbers themselves are composite. If the randomly generated number passed the primality test, it was the prime number that we’d use. For values a and b, we also randomly generated a number. These numbers, however, were not required to be prime.

In order to generate the results for both public keys and the final shared key, we used the powermod function, which generates values according to the formula “ab mod c”. This function has a time complexity of O(log n). When implementing the person class to simulate information as public vs. private, two values were created as private. These values were the person’s private key as well as the final private message. The rest of the values, including p, g, and public keys were created as public, and thus anyone would have access to these values. All functions regarding generating and decoding keys, as according to Table 1, are also built in functions within the public section of this person class.

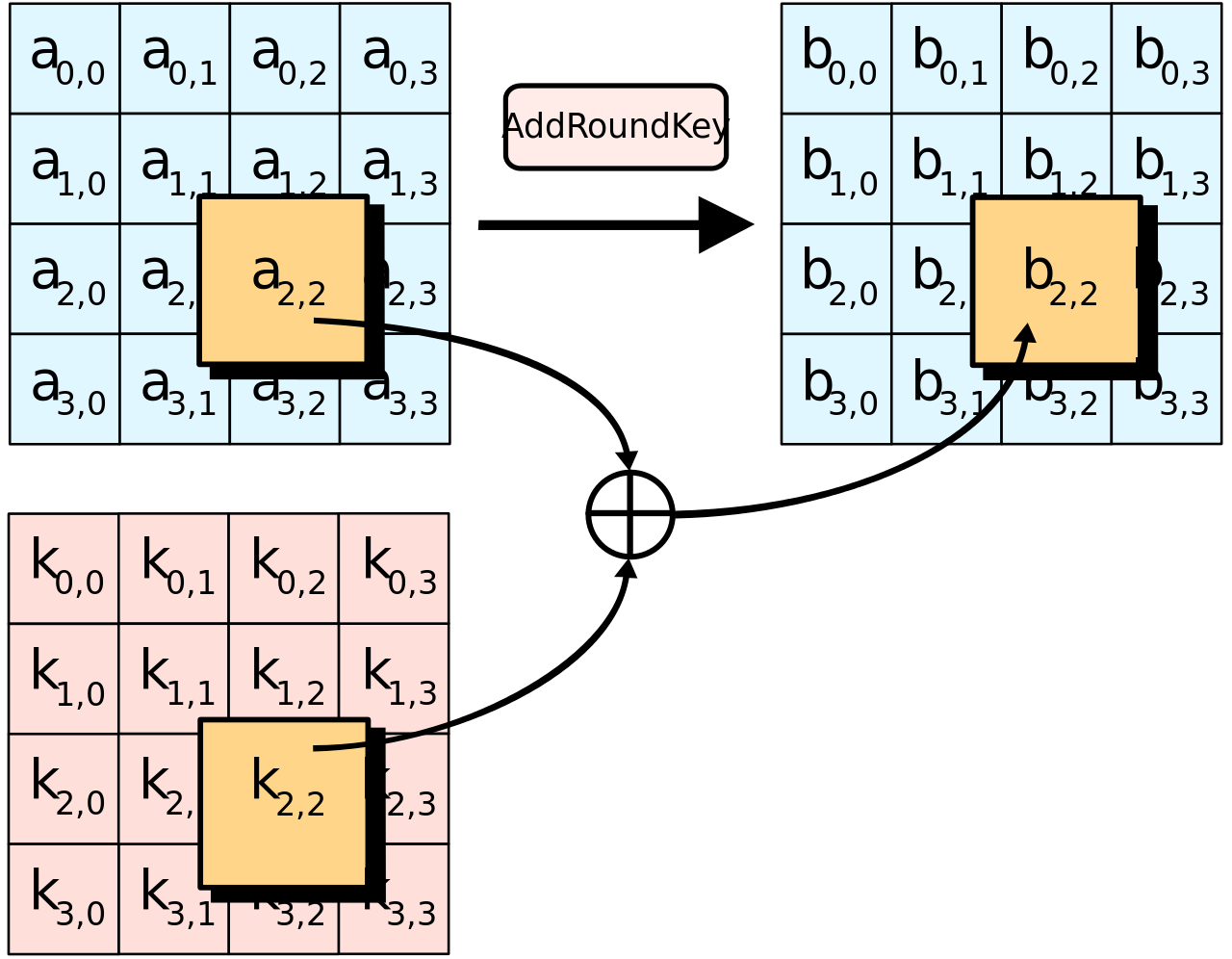
# AES Algorithm:

## Encryption:

The AES algorithm is made up of five separate functions, which are titled as Expand\_Keys(), Add\_Round\_Key(), Sub\_Bytes(), Shift\_Rows() and Mix\_Columns(). The basic idea is that the algorithm segments its encryption key and plaintext message into separate 128 bit blocks and encrypts the message into ciphertext block by block. The biggest aspect of what makes AES encryption work so nicely is that every function operates on a fixed grid of 128 bits, otherwise known as the state array. Since each function only iterates once per element of the state array, they all have a time complexity of O(1), making the process very fast for both encryption and decryption. These functions each play a role in somehow manipulating the state array in a manner that does not increase the overall size of the grid. So to explain the algorithm’s process, we will first start with the Expand\_Keys() function, whose purpose is to transform the larger 256 bit encryption key into something that can be used within the state array. The process breaks down the larger 256 bit key into smaller 128 bit “round keys'' to be used in the state array during each step of the encryption process. In essence, it is a key scheduling algorithm that determines what keys will be used throughout the algorithm. This process was not meant to be complicated, but rather fast in order to improve the speed of the overall problem.

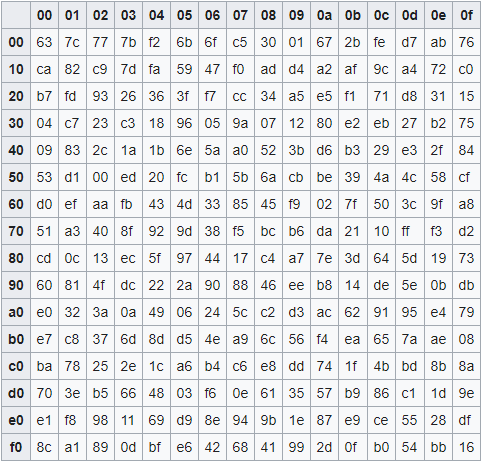
The next function of the encryption process is the function Add\_Round\_Key(). As the name suggests, this function simply adds in the next round key that has been produced by the key scheduling algorithm. However, rather than actually using addition, the process uses the bitwise operator exclusive or (XOR) in order to bring the next scheduled key into our state array. The diagram below in Figure 1 demonstrates this process, where grid A represents the original state array, grid K represents the round key and grid B represents the resulting state array after XOR operation.

*Figure 1: Add\_Round\_Key() Function*

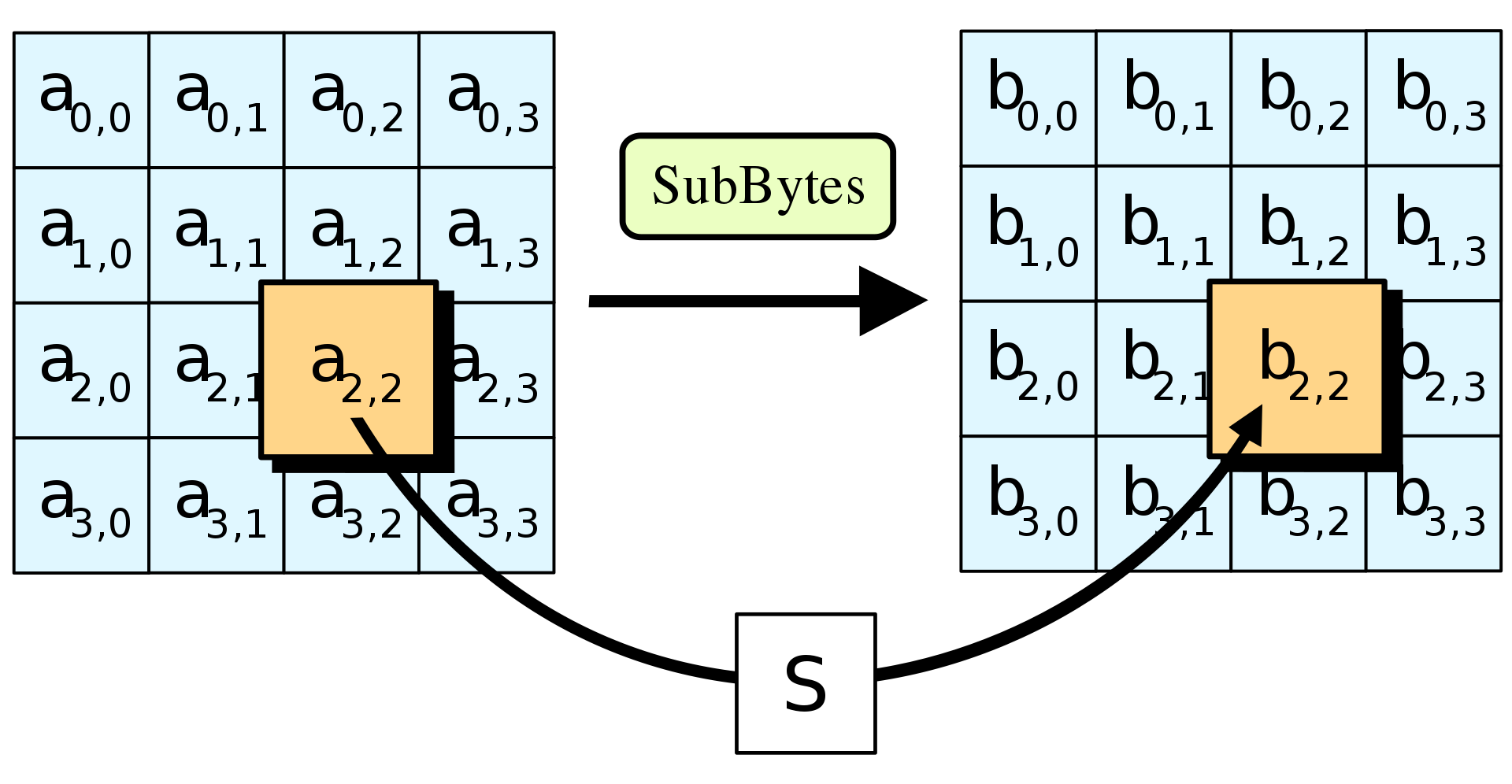


The next function in the encryption process is the Sub\_Bytes() function. During this step, each element of the state array is simply being substituted with a value from a predefined array known as the “sBox”. The sBox array is standard for all AES algorithms and was designed in such a way that when substitution occurs, no value will ever be substituted with itself, nor will a value be substituted with its opposite, meaning every bit is inverted. The entirety of the sBox can be seen in Figure 2 and the process of SubBytes can be represented in Figure 3, where grid A represents the state array prior to substitution and grid B represents the state array after substitution with the corresponding sBox value.

*Figure 2: Standard sBox Array*

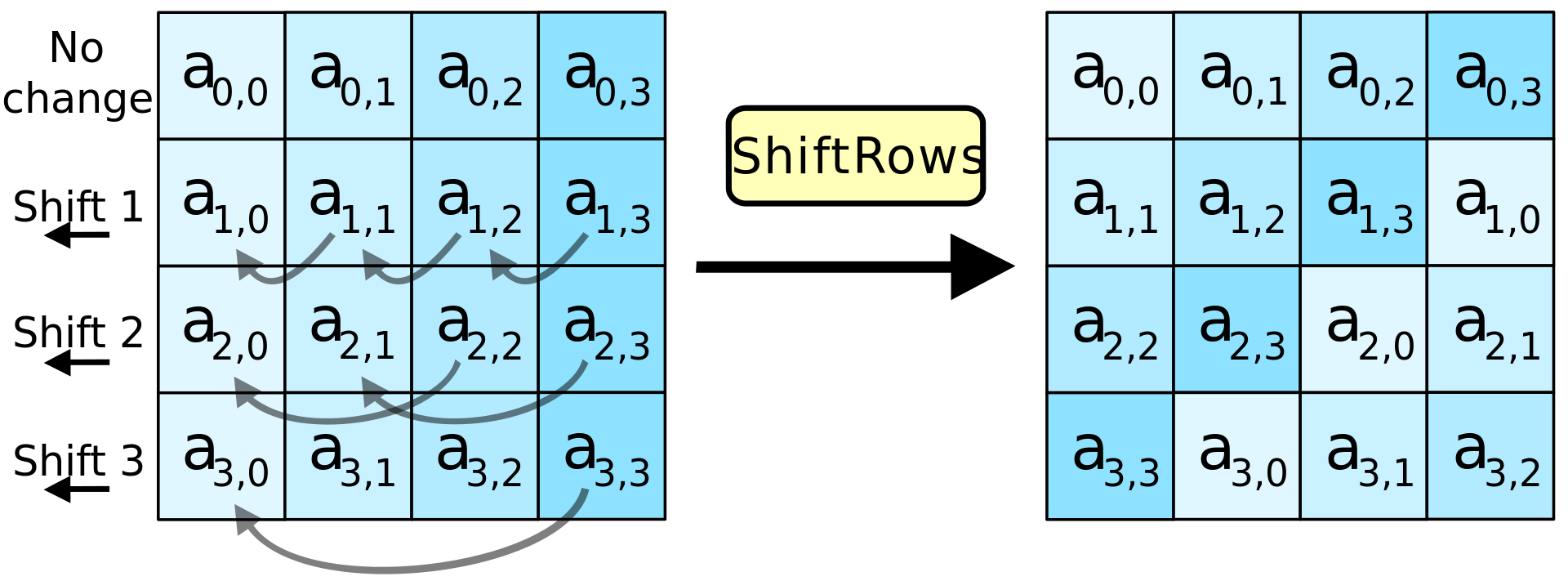


*Figure 3: Sub\_Bytes() Function*



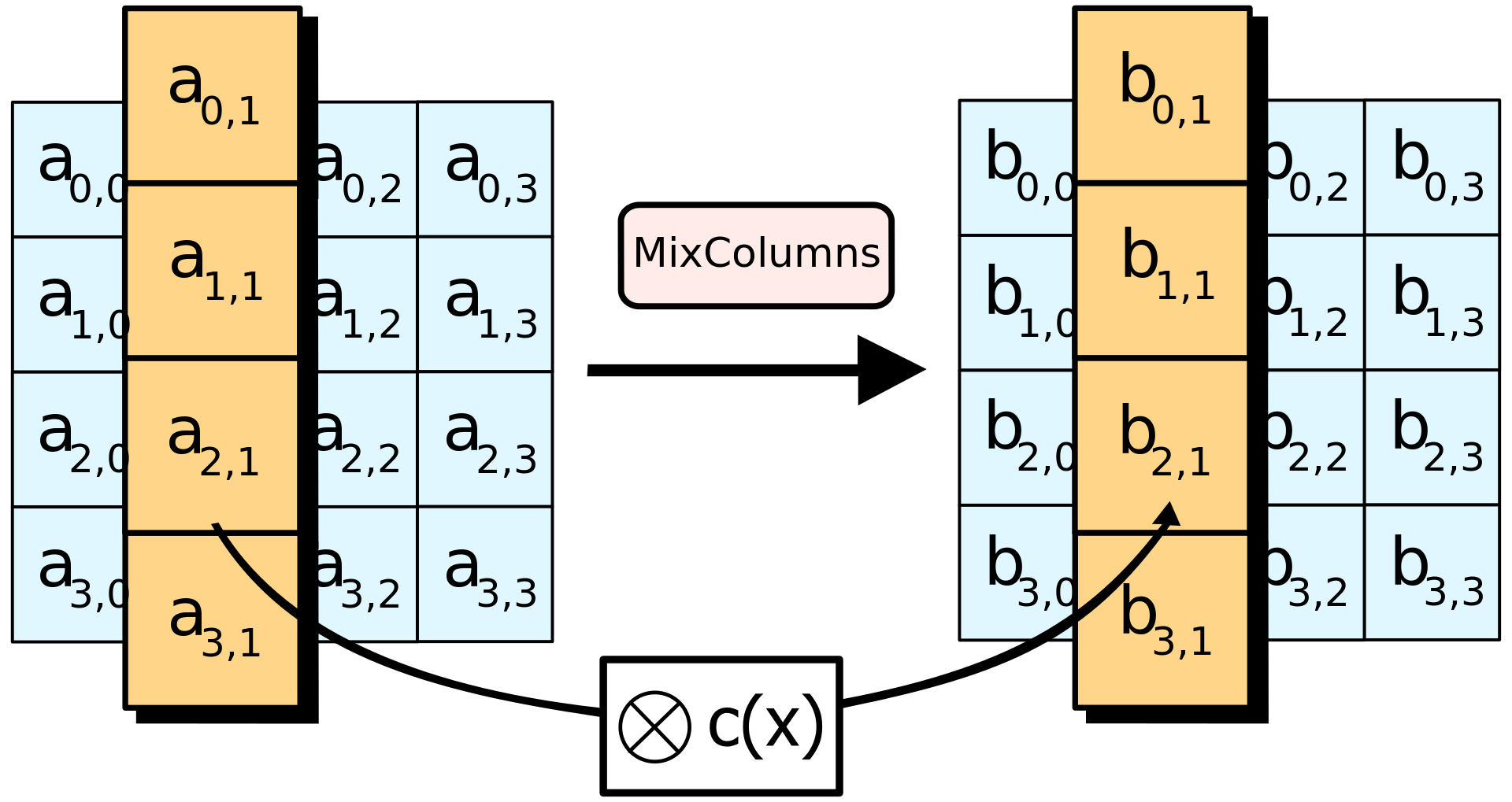
Moving on in the encryption process, the next function is the Shift\_Rows() function. In this step, the state array is shifted according to the same idea as logical shift left, only it is the array values themselves that are being shifted. The first row has no shift, the second is shifted once, the third row is shifted twice, and the fourth row is shifted by three. This shifting concept is outlined in Figure 4 below.

*Figure 4: Shift\_Rows() Function*



The final function that makes up the AES encryption algorithm is the Mix\_Columns() function, which is one of the more involved steps. Just as the previous function, Shift\_Rows(), mixed up the state array via its rows, the Mix\_Columns() function mixes up the state array via its columns. However, it is not as simple as just shifting each item. Each column essentially undergoes a form of matrix multiplication. However, since we are operating under a finite field and our total bit count is limited to 128, regular multiplication cannot be done. Instead, the multiplication step is done with a fixed polynomial value, c(x), where c(x) consists of a combination of modulo operations with 0x1b and XOR operations within each value of a column. The math behind this operation can be explained more by finite fields, or Galois Field logic, however the group’s goal was not to understand finite field logic, but rather implement a successful variant of AES encryption. Figure 5 below demonstrates the main idea of the Mix\_Columns() function.

*Figure 5: Mix\_Columns() Function*



Each of the five previously described functions ultimately each combine in a series of steps in order to produce a final encrypted result. The first step occurs at step 0, where the designated encryption key must first be expanded into round keys according to the Expand\_Keys() function. Next, step 1 adds the first round key to the state array with Add\_Round\_Key().The state array initially begins as plaintext characters converted to hexadecimal values. This sets up the state array for step 2, where a majority of the encryption takes place. Step 2 is a process that is repeated depending on how large of an encryption key is used. For the group's implementation of a 256 bit key, this step is repeated a total of 13 times. If the key were 128 bits in length, it would be repeated 9 times, and if the key were 192 bits in length, it would be repeated 11 times. Step 2 runs through the four AES functions of Sub\_Bytes(), Shift\_Rows(), Mix\_Columns(), and Add\_Round\_Key(), in that order. After concluding step 2, the process is completed in step 3 with a single round of the functions Sub\_Bytes(), Shift\_Rows(), Add\_Round\_Key(). The previously described encryption process can be summarized in Table 2 below.

*Table 2: Encryption Process*

|  |  |
| --- | --- |
| **Step 0** | 1. Exapnd\_Keys() |
| **Step 1** | 1. Add\_Round\_Key() |
| **Step 2**  (Repeated 13 times for 256 bit key) | 1. Sub\_Bytes() 2. Shift\_Rows() 3. Mix\_Columns() 4. Add\_Round\_Key() |
| **Step 3** | 1. Sub\_Bytes() 2. Shift\_Rows() 3. Add\_Round\_Key() |

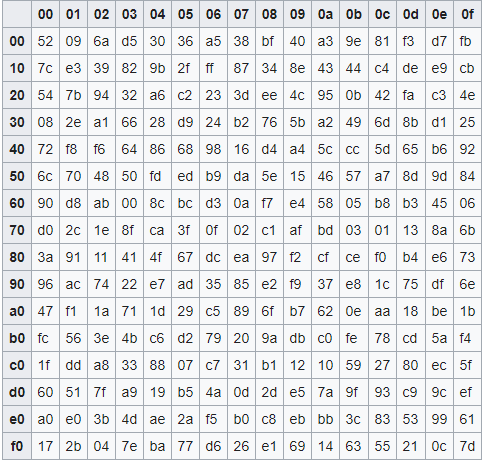
## Decryption:

Now that plaintext has been converted into ciphertext, a method is required in order to reverse the effects of the encryption algorithm. Since the AES algorithm is a symmetric-key algorithm, the same key used for encryption will be required for decryption. Therefore, the result of the Expand\_Keys() function will be the same, and therefore will not be necessary to incorporate into the decryption process. The other four functions of Add\_Round\_Key(), Sub\_Bytes(), Shift\_Rows() and Mix\_Columns() will need to be applied, however their operations will need to be done inversely. The following will explain how the inverse can be applied to each of these functions.

Starting with Add\_Round\_Key(), its only purpose is to assign round keys based on the calculated key scheduling algorithm. If the encryption process applied these round keys in chronological order, then the inverse of this would be applying them in the reverse order. That’s exactly what is done in the decryption function. No edits are needed to manipulate the function, the only change is the index of which round key is taken, which will start from 13 and decrease down to 0.

The next decryption function is Sub\_Bytes(), where we need to substitute values from the standard AES sBox array. The inverse of these values can be calculated by the inverse affine transformation followed by the multiplicative inverse. However, the calculation is complicated, and the resulting inverse sBox will always be the same constant. Instead of calculating the inverse value, it can simply be stored as a separate array and substituted just like the encryption process. We can call this function Inv\_Sub\_Bytes() and the corresponding inverse sBox grid can be seen in Figure 6 below.

*Figure 6: Standard Inverse sBox Array*

****

The next inverse function needed is the inverse function for Shift\_Rows(). Developing an inverse function for shifting is very easy. We just need to shift the state array in the same manner as before, however, instead of shifting left, the array must be shifted to the right. So, in the same process as encryption, just in the opposite direction, the first row is not shifted, the second row is shifted right once, the third row is shifted right twice, and the fourth row is shifted right three times. This inverse shift function is named as Inv\_Shift\_Rows(). Refer back to Figure 4 moving in the opposite direction.

For the final required function, an inverse to the encryption function of Mix\_Columns() is required. This function will be referred to as Inv\_Mix\_Columns(). Just as in the function for encryption, a form of matrix multiplication is required in order to reverse the results of Mix\_Columns(). Again, this calculation takes place within the finite field, or Galois field, so that the multiplication process does not cause an overflow of bits. The mathematical process is complex, and thorough understanding was not required in order to complete the project. The important thing to grasp, however, is that the Mix\_Columns() function is a method of altering only the columns of the state array and the inverse function is required to set it back to its original state. Refer back to Figure 5 for a visual of the Mix\_Columns() function.

The final decryption function is similar in structure to that of the encryption function in that step 2 of the process must be repeated 13 times for a key size of 256 bits. The order in which the AES functions are called are the exact opposite of how they were called in the encryption function. The entire decryption process can be seen below in Table 3.

*Table 3: Decryption Process*

|  |  |
| --- | --- |
| **Step 0** | 1. Exapnd\_Keys() |
| **Step 1** | 1. Add\_Round\_Key(reverse) |
| **Step 2**  (Repeated 13 times for 256 bit key) | 1. Inv\_Shift\_Rows() 2. Inv\_Sub\_Bytes() 3. Add\_Round\_Key(reverse) 4. Inv\_Mix\_Columns() |
| **Step 3** | 1. Inv\_Shift\_Rows() 2. Inv\_Sub\_Bytes() 3. Add\_Round\_Key(0) |

# Discussion of Results:

The first step in developing a fully functional AES algorithm is to create an encryption/decryption key. To accomplish this, the group utilized the previously described Diffie-Hellman key exchange. Since the required key size is 256 bits, the group had to somehow generate a key of this size, even though the maximum size of a C++ long long integer is 64 bits. To accomplish this, a key array was created, where each element of the array would store a value of 8 bits. With the Diffie-Hellman algorithm, the resulting secret message must be broken down as modulo 256 in order to confirm only 8 bits are generated. This random result of the Diffie-Hellman algorithm is then added as the first element of the key array and the process is repeated a total of 32 times. At the end of this loop, the resulting key array contains a 256 bit key that will be used for both the encryption and decryption processes.

The next step required is to read in an input text file. As previously mentioned, the AES algorithm works in blocks of 128 bits. Obviously, there may be cases where a file larger than 128 bits is used as an input. In these cases, the algorithm will read in the first 128 bits, perform the full encryption process on that block, and then append the resulting ciphertext to an “encrypted\_data” string. This process of reading in a block of plaintext, encrypting it and appending the ciphertext to the string is repeated for the remainder of the input file. Once this process is completed, the resulting string is then written to a file titled “encryptedData.txt” and the encryption process is completed. Overall, the encryption process of an entire file has a time complexity of O(n/128), where n is the number of characters within the file. This can just be simplified down to O(n).

In order to decrypt the message, the same process of reading input in 128 bit blocks is used. Now, instead of a plaintext file being used as input, the resulting “encrytpedData.txt” file must be inputted. It is also important to note that as a symmetric-key algorithm, the same encryption key must be used for the decryption process to be successful. Again, as 128 bit blocks are decrypted, the plaintext result is appended to a “decrypted\_data” string. At the end of the loop of reading and decrypting input, the final string is then written to a file named “decryptedData.txt”. This file should be a mirror image of the original plaintext input file. Again, like encryption, the full decryption process of a file has a time complexity of O(n).

The overall main implementation of the algorithm was ultimately condensed into a switch statement, where the user may choose to encrypt, decrypt, or generate a new key. For both the encrypt and decrypt options, the only input required is the name of the plaintext file or ciphertext file, depending on the chosen operation. In addition, the option to generate a new key is available if multiple files are desired to be tested. However, it is extremely important that the decryption process immediately follows the encryption process. If a new key is generated in between these processes, there is currently no method of retrieving the prior used key. New keys should only be generated after the decryption process is completed.

Along the way, the group encountered a few major challenges in the algorithm’s implementation. The first encountered challenge was the case where an input file was not an exact multiple of 128 bits. Say there were only 32 bits to read during the final iteration of the encryption process. The resulting output “encrypted\_data” string would contain 96 bits at the end of the string that were identical to the previous iteration. To get around this problem the group implemented a custom hash table for converting characters to hexadecimal value and vice versa for decryption. This hashtable was only used for the conversion from the plaintext input to hexadecimal value, for encryption, and the conversion from hexadecimal back to plaintext at the end of the decryption process. When ciphertext is outputted to the “encryptedData.txt” file, this conversion is automatically done through ASCII character conversion through the string library. In the custom hash table, the value of 0 was hard coded as a space character. In addition, each value of the encrypted data array, where the final encryption result is stored, is always initialized back to 0 after appending its results to the “encrypted\_data” string. This way, in the case previously described scenario, where the final iteration does not contain a full 128 bits, the remaining bits at the end of the array will simply write whitespace to the end of the string, which has no effect on interpretation of the final result.

The second, more difficult challenge, came with the ASCII conversion when outputting to the “encryptedData.txt” result and reading it back in. Since it will never be known how long a given input file will be, the algorithm is designed to read until it reaches an EOF character, which indicates the end of a file. However, the hexadecimal value of “0x1a” is mapped to the EOF character in ASCII conversion. So every time the value of “0x1a” was written as ciphertext, when that value was read back in for decryption, the algorithm would see it as the end of file and immediately stop reading input, often times leaving a large amount of ciphertext not decrypted. To solve this issue, the group needed to somehow manipulate values of “0x1a” so that they are never read by the decryption process. To accomplish this, a boolean valid array was created, with one value for every character of an input file. This also required an integer to act as a counter for every character being encrypted. Every time the encryption process resulted in the value “0x1a”, the corresponding indexed value in the valid array was set as true. In addition, “0x1a” was incremented to “0x1b” before being written to the output. Now, when the decryption process is reading in values of ciphertext, it first checks if the value is”0x1b”. If this is true, it then checks if the corresponding index in the valid array is true. If both are true, the value of “0x1b” was decremented back to “0x1a” and the corresponding valid bit was returned to false. This process ultimately avoids ever reading an EOF character unless it is truly the end of the file.

# Conclusion:

All in all, the group was able to create a working program, however, a few improvements could be made to the final implementation. Firstly, the current version has no method of saving keys for use in the future. Meaning, that once a key is used and the program stops, there’s no way of ever knowing what that key was. The code could be improved to potentially export the key used to a separate file. An additional feature could be reading an input key from a file or user input, rather than always generating a random key. An option could also be implemented for the user to specify the length of key they want to use: 128, 192 or 256 bits. If this were implemented, the encryption/decryption loops would have to be made dynamic in order to account for the varying repetitions that are required with different length keys.

Aside from these potential improvements, the group was able to produce a working algorithm of AES 256 bit encryption. The combined team effort along with the vast amount of references regarding AES encryption helped the team to succeed overall. The algorithms used within the class helped the project move forward with a lot of the early processes, such as using Miller-Rabin as a primality test or using powermod() as a medium to calculate encryption keys under the Diffie-Hellman algorithm. Additionally, the skills acquired within class were useful in achieving a working algorithm within the timeframe that was allotted. Working with libraries like <vector> or even working with arrays throughout the semester were essential to reaching the final implementation of the project.

The AES encryption method was an interesting topic to work on, mainly due to how applicable it is to modern times. Now, more than ever in history, people need a near flawless method to protect their data in all virtual environments. It is up to the people who are knowledgeable in this field to guarantee that near flawless system to the public. For those reasons, it can be said that the AES encryption method is essential for everyone in the field to be familiar with, and working on this project properly introduced the team as to how to make it work.

# References:

[1] “Advanced Encryption Standard”. Wikipedia. <https://en.wikipedia.org/wiki/Advanced_Encryption_Standard>

[2] Dr. M. Pound. “AES Explained (Advanced Encryption Standard)”. youtube/Computerphile. Nov 22, 2019. <https://youtu.be/O4xNJsjtN6E>

[3] M. Mkhitaryan. “AES-Rijndael-Encryption”. GitHub. April 23, 2017.

<https://github.com/manvelmk/AES-Rijndael-Encryption/blob/master/AES.cpp>

[4] S.Kallam. “Diffie-Hellman:Key Exchange and Public Key Cryptosystems,” Indiana State University, Sept 30, 2015. <http://cs.indstate.edu/~skallam/doc.pdf>