# An Implementation of the Advanced Encryption Standard

*Will Puzella & Peter Kelly*

*Computer Science, Carleton College, Fall 2024*

## Introduction

The Advanced Encryption Standard (AES), is a symmetric block cipher algorithm that is the current US standard for encryption. The block cipher is in many protocols used worldwide daily, such as SSH and TLS. The algorithm is a version of the Rijndael family of ciphers which was developed by Joan Daemen and Vincent Rijmen. They submitted their ciphers to NIST during a competition that was being used to decide the next standard for encryption. Out of the collection of Rijndael ciphers that were submitted for varying block and key sizes, three were chosen all with a block length of 128 bits, and key lengths of 128, 192, and 256 bits. These three block ciphers make up what is now known as the Advanced Encryption standard.

AES is a quick and efficient algorithm that allows for easily parallelizable encryption and decryption of files. In the previous paragraph, the terms block and key lengths were discussed. These two terms will be used throughout this document so we will first define them as all further explanations will be based on both of these ideas.

## The BLOCK

First, what is a block? A block is a sequence of bits from the unencrypted text. In this case, the unencrypted text is the file you plan to encrypt, and as we will discuss later in our implementation it could be anything. In AES a block is specifically 128 bits long or 16 bytes.

Second, what is a key? A key is a sequence of bits that is also 128 bits long that then will be used to decrypt and encrypt each block of data used in an instance of the algorithm.

With those terms out of the way, we can begin to break down the algorithm. The AES algorithm has two scenarios of encryption: a file is 128 bits long and a file is larger than 128 bits. We will discuss the first scenario as the second one builds off the ideas of the first.

## AES Block Cipher Encryption

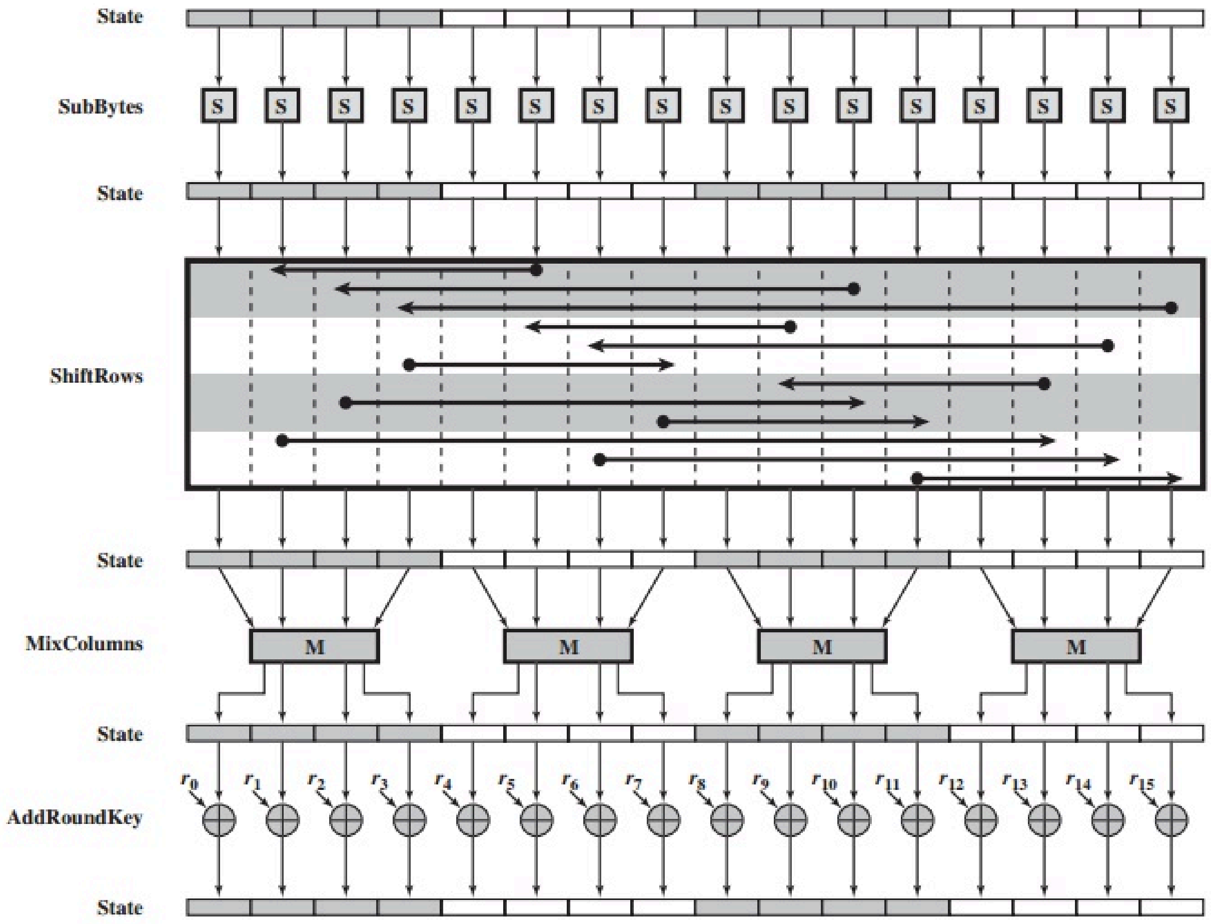
State
SubBytes
S
State
ShiftRows
State
MixColumns
M
State
$r_0$ $r_1$ $r_2$ $r_3$ $r_4$ $r_5$ $r_6$ $r_7$ $r_8$ $r_9$ $r_{10}$ $r_{11}$ $r_{12}$ $r_{13}$ $r_{14}$ $r_{15}$
AddRoundKey
State

**Figure 1**: Diagram of a single round of the AES block cipher algorithm. Adapted from Stallings.

The AES block cipher takes a 128-bit block of data and a correctly sized key and returns a block that has been uniquely encrypted according to that key. A correctly sized key is 128, 192, or 265 bits long, and the number of rounds AES does is determined by the length of the key. For a 128-bit key there are 10 rounds, or for a 192-bit key there are 12, and for a 265-bit key there are 14. Aside from the first and last rounds, each round is made up of a SubBytes, ShiftRows, MixColumns, and AddRound key step. The first round only adds a round key and the last round does not mix columns. The goal of the block cipher is to encrypt the block in such a way that any change to the input or key is widely propagated through the state in a pseudo-random way. The encryption of each round is handled by the key expansion and AddRoundKey while propagating any change through the state is done by SubBytes, MixColums, and ShiftRows.

## KeyExpansion

The goal of KeyExpansion is to turn a key into several round keys. These round keys are then used every round to encrypt the result of the other steps. We create these round keys by letting the first four round keys be the four 4-byte sections of the original key, while the last 6-10 are formed by XORing a previous round key,  whose location depends on the key length, with the previous round key after it has been substituted and permutated.

## SubBytes

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} \tilde{b}_0 \\ \tilde{b}_1 \\ \tilde{b}_2 \\ \tilde{b}_3 \\ \tilde{b}_4 \\ \tilde{b}_5 \\ \tilde{b}_6 \\ \tilde{b}_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}.$$

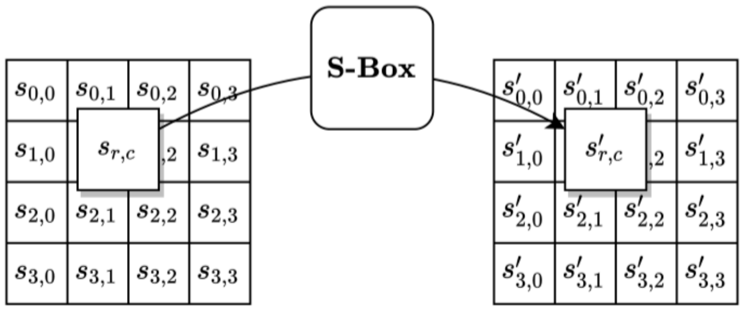

**Figure 2:** Diagrams of the SubBytes function. Adapted from FIPS 197.

The goal of SubBytes is to substitute the bytes of a given input for a different set of bytes in a non-linear manner. This can be performed by an algorithm that calculates each value every time, but since it is a deterministic function we can simplify this step using a table lookup.

## ShiftRows

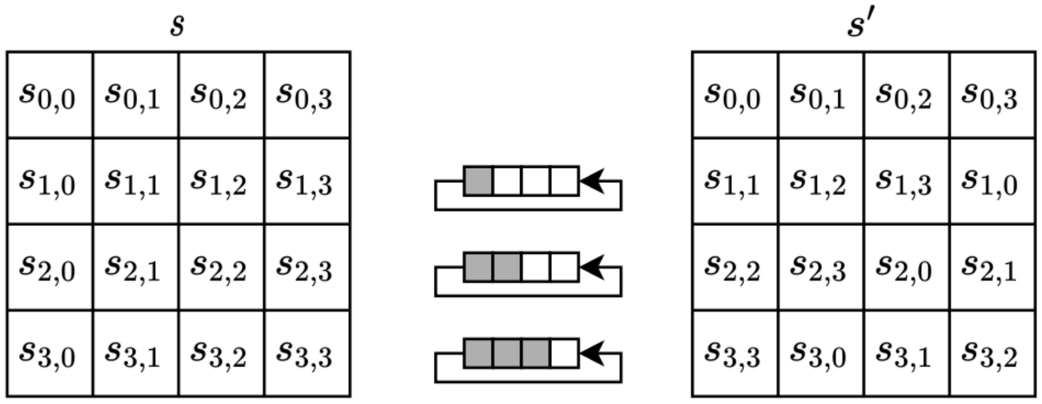

**Figure 3:** Diagram of ShiftRows procedure. Adapted from FIPS 197.

The goal of ShiftRows is to take the state and permutate it so that one byte from each input column ends up in every resulting column. This is done by not changing the first row, shifting the second row one byte to the left, the third row two bytes, and the fourth row three bytes.

## Mix Columns

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad \text{for } 0 \le c < 4,$$

**Figure 4:** Diagram of Mix Columns procedure. Adapted from FIPS 197.

The goal of MixColumns is to perform a fixed matrix multiplication on a column. This part of the permutation of the state, and ensures that the value of every resulting byte is determined by all of the other bytes in the column.

## AddRoundKey

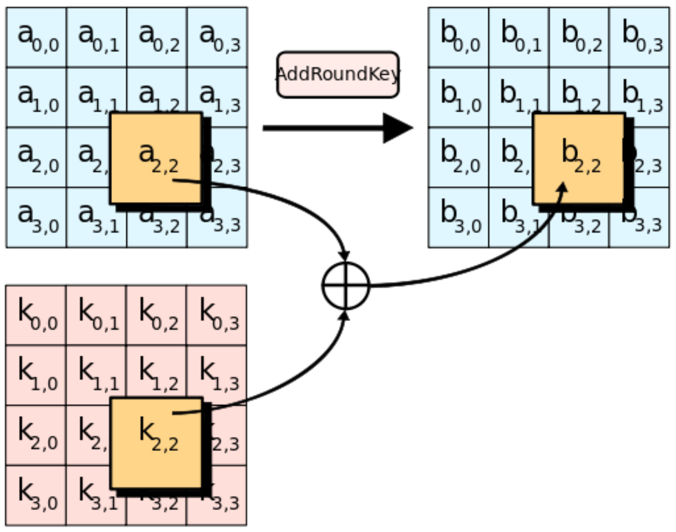

**Figure 5:** Diagram showing the Key AddRoundKey process. Adapted from FIPS 197.

The goal of the add round key function is to add a round key to the state. It does so by XORing every byte of the round key with its corresponding byte in the state.

# AES Block Cipher Decryption

Decrypting with AES is simple because it uses the same key and each function has a simple way to reverse it. However, this still requires a 128-bit block of data and a correctly sized key. Computing the inverse cipher follows the same steps as the forward cipher using each function's inverse.

### InvKeySchedule

The inverse key schedule calculates the key schedule then reverses it and applies InverseMixColumns to all but the first and last keys.

### InvSubBytes

Inverse subbytes reverses subbytes, and while this can also be implemented with an algorithm, we chose to use another table lookup.

### InvShiftRows

Inverse shift rows is the reverse of shift rows, so instead of shifting to the left, you shift to the right.

### InvMixColumns

Inverse mix columns is just the inverse of the fixed matrix multiplication in mix columns.

## Block Cipher Modes

Now that we have gone through how each 128-bit block is encrypted we can discuss how files larger than 128 bytes can be encrypted with AES. This is done by concatenating 128-bit blocks together in the output file on top of possibly adding additional transformations to the data which we discuss further in our CBC and GCM sections. If a file is not exactly divisible by 128 then the last block will be padded with zeros which can be removed on decryption. The three block cipher modes that we implemented, electronic codebook, cipher block chaining, and

Galois counter mode, are common schemes for combining blocks in order of least to most secure.

## Electronic Codebook (ECB)

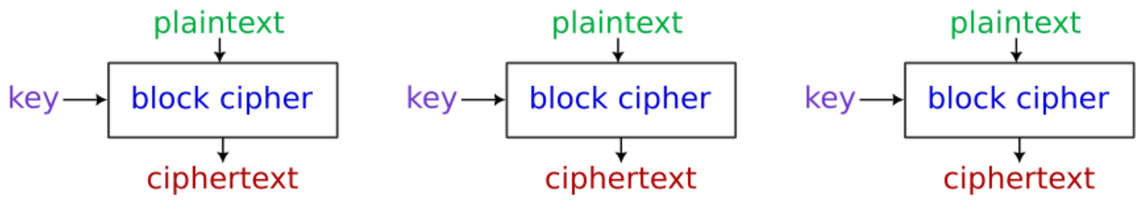

**Figure 6:** Electronic Codebook block diagram. Adapted from A.M. Rowsell and Epachamo, CC BY-SA 4.0, via Wikimedia Commons.

The Electronic Codebook block cipher mode is the simplest of block cipher modes. ECB does not add any additional block transformations to the data. Instead, it executes the block cipher on each block independently. This makes ECB the fastest mode as it can be easily parallelizable since each block only needs the information contained in its block. Unfortunately, this block cipher mode is the least secure as repeated data will all be encrypted the same. This can be seen easily in a situation where you would want to encrypt a picture. Each color of the image is encrypted by its RGB values which are usually stored as 3-bytes. Since each block is 16 bytes you will be able to encrypt multiple pixels within each block. Making the output block the result of multiple pixels. Yet what if the picture is an object with a single-colored background? Now you will be able to at least see the difference between the foreground and background in the photo as each pixel in the background is encrypted the same. Being able to discern any elements from the encrypted document does not constitute a secure algorithm. These downsides to ECB have been recognized and the block cipher mode and more secure modes are now being used in its stead.

Even with the downsides of the ECB in mind, we implemented this block cipher mode in our implementation of AES. Our implementation of ECB reads in a file one block at a time, calling the block cipher encryption function each time a full block is read. The resulting encrypted block is then written to the output file. The final block is then zero-padded up to 128 bits to allow for proper encryption. Similarly, the decryption function reads in a file one block at a time, calling the block cipher decryption function each time a full block is read. The final block is then decrypted removing any zero padding that was added during the encryption process.

To run our encryption algorithm in ECB mode:

```
./PK_AES -E -I [input file] -O [output file]
-E: This Argument defines a block cipher mode as ECB
```

To decrypt in ECB Mode:

```
./PK_AES -D -E -I [input file] -O [output file]
-E: This Argument defines a block cipher mode as ECB
-D: This Argument defines decrypt mode
```

## Cipher Block Chaining (CBC)

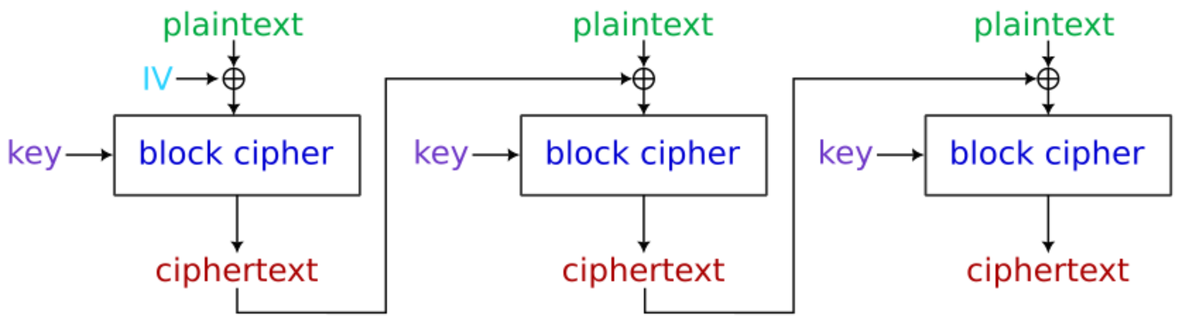

**Figure 7:** Cipher Block Chaining diagram. Adapted from A.M. Rowsell and Epachamo, CC BY-SA 4.0, via Wikimedia Commons.

The Cipher Block Chaining block cipher mode was our second implemented block cipher mode. It allows for parallelization during the decryption only but the loss of threading in the forward direction is made up with the additional level of encryption that the mode provides. The CBC mode takes one additional parameter called an initialization vector which is used to initialize the first block in the chain. The CBC mode works by taking the previous block's cipher text and XOR it with the unencrypted text of your current block. In the case of the first block of a file, the initialization vector is XORed rather than a previous block. This results in additional pseudorandomness being added since each block does not just contain its data; instead, it takes into account all the previous bits of the file as well. This relieves the encryption of a similar block problem that I mentioned in the ECB section as each block even if the data they encode is the same will be located in different parts of the file making the resulting XORed text different.

As mentioned above, the beauty of the AES algorithm is that if any bit of block text or key is changed then the entire encrypted block will be different. This idea can be expanded with the CBC block cipher mode as changing the initialization vector will also change the entire encrypted text. Additionally, if you change a singular bit in the encrypted text, on top of the fact that the resulting block will be entirely different, the remainder of the blocks will all be different since they are based on the encrypted text of the current block.

Our implementation of CBC is similar to ECB since it reads in the blocks one byte at a time until a full block is read. The block is then XORed with the previous block before it is encrypted using the block cipher. As mentioned earlier in the case of the first block the initialization vector is used instead of the previous block. Since the blocks have to be read consecutively to be encrypted the encryption is not easily parallelizable. Yet the decryption is! The decrypt function is the same process in reverse order. The blocks are decrypted and then the previous encrypted block is XORed with the decrypted current block resulting in the original plain text being returned. Since all of the encrypted forms of the blocks are contained in the input file during the decryption process the process can be parallelized allowing for even faster decryption.

<u>To run our encryption algorithm in ECB mode:</u>

```
./PK_AES -C -I [input file] -O [output file] -V [IV]
-C: This Argument defines a block cipher mode as CBC
-V [IV]: (Required) This argument provides input for the
initialization vector. The input can either be a string of 16
characters or a string of 32 hex digits in the format of
0x123456789ABCDE… (The Ox is not included) The IV must be 128 bits.
```

<u>To decrypt in ECB Mode:</u>

```
./PK_AES -D -C -I [input file] -O [output file] -V [IV]
-E: This Argument defines a block cipher mode as CBC
-D: This Argument defines decrypt mode
```

-V [IV]: (Required) This argument provides input for the initialization vector. The input can either be a string of 16 characters or a string of 32 hex digits in the format of 0x123456789ABCDE… (The Ox is not included) The IV must be 128 bits.

## Galois Counter Mode (GCM)

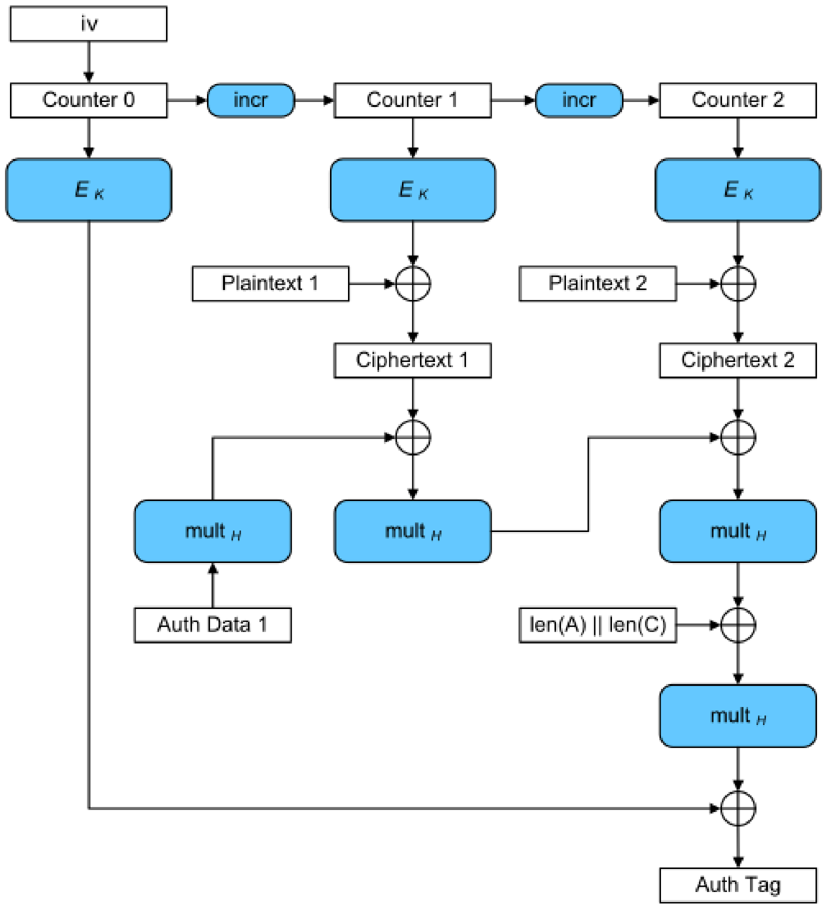

**Figure 8:** Galois Counter Mode block diagram with an initialization vector, adapted from a diagram by Dworkin / NIST.

Galois counter mode was the final block cipher mode that we implemented. GCM is the current standard for encryption and decryption acting as the encryption standard for both SSH and TLS protocols. The benefit of GCM over other modes such as CBC or ECB is the creation of an authentication tag. In other block cipher modes for AES, there is no verification that your decrypted text is the same as the text that was encrypted. The Authentication tag is used to verify if the decrypted file is the same as its original counterpart. This is done through Galois finite field multiplication similar to what is being used in the block cipher.

GCM takes an initialization vector like in CBC mode but it serves a different purpose. Instead of being used to initialize the first block it is used to seed the counter. If no IV is provided our implementation will calculate a random one for you. Similar to Counter Mode, another block cipher mode we did not implement, GCM uses a counter to encrypt the data rather than the data itself. The counter is then XORed with the plain text to create an encrypted text. After each block, the counter is incremented by one. The use of an encrypted counter allows the encryption to be very parallelizable since each encrypted counter block can be quickly calculated as it relies on only IV and the Key. The decryption of the function involves a recalculation of the encrypted counter blocks then each encrypted counter can be XORed with its respective encrypted block resulting in the original plaintext. This means that no separate decryption function needs to be written instead the encryption of the decrypted document results in the plaintext.

The second part of GCM is the calculation of the Authentication key. This is done after each cipher text is calculated by XORing the plaintext with the encrypted counter. The hash is first initialized by calculating the encrypted block of all zeros using the key and original counter value. This encrypted zero block is then multiplied using Galois Finite Field ($2^n$) against the first ciphertext block. The product of that multiplication is then multiplied against the second cipher block. This hashing procedure is repeated for each remaining block up to the last block. After the last block is hashed the length of the file and any unencrypted header information is multiplied into the hash. The 16-byte hash is then returned.

The decryption step of GCM takes an additional argument of Tag which is then used in a comparison at the end of decryption. The comparison is between the provided hash and a recalculated hash from the data. If the two hashes are the same then the data is Authenticated and returned otherwise the decryption will fail indicating that the data has been tampered with.

To run our encryption algorithm in GCM mode:

```
./PK_AES -G -I [input file] -O [output file] -V [IV]
-G: This Argument defines a block cipher mode as GCM
-V [IV]: (Optional) This argument provides input for the
initialization vector. The input can either be a string of 16
characters or a string of 32 hex digits in the format of
0x123456789ABCDE… (The Ox is not included) The IV must be 128 bits.
If you do not input an IV then a random one will be chosen for you
and displayed.
```

To decrypt in GCM Mode:

```
./PK_AES -D -G -I [input file] -O [output file] -V [IV] -T [tag]
-G: This Argument defines a block cipher mode as GCM
-D: This Argument defines decrypt mode
-V [IV]: (Required) This argument provides input for the
initialization vector. The input can either be a string of 16
characters or a string of 32 hex digits in the format of
0x123456789ABCDE… (The Ox is not included) The IV must be 128 bits.
```

`-T [tag] (Required) This argument provides input for the authentication tag. The input can either be a string of 16 characters or a string of 32 hex digits in the format of 0x123456789ABCDE… (The 0x is not included) The tag must be 128 bits.`

## Command Line Interface

We implemented a command line interface to receive inputs to our algorithm. We parse through the command line arguments to find those that begin with `-`, then change the flag that corresponds to that option. If we find a flag that requires additional input then we will save the next command line argument to be its input. After parsing we will run whichever version of our algorithm the flags indicate on the data from the input file and write the output to the specified output file.

## CLI Interface

### To run

To run the encryption/decryption software run:

```
./PK_AES -I [input file directory] -O [Output File directory] -K
[key] [additional arguments]
```

### Argument Style

The style of the arguments is such that they can be put in any order as long as the format is:

```
 <argument tag> | <argument input> | <argument tag>
```

This can be seen in the sample command above.

## All Arguments

`-I [input file]` Input file

`-O [output file]` Output file

`-K [key]` Key

`-D` Decrypt mode (by default it will encrypt a file)

`-E` Encrypts in Electronic Code Book (ECB) block cipher mode

`-C` Encrypts in Counter Block Cipher (CBC) mode

`-G` Encrypts in Galois Counter Mode (GCM)

`-T` Authentication Tag (Required for GCM Decryption)

`-V` Initialization Vector (Required for CBC and GCM decryption)

## Argument Requirements

Tags `-I`, `-O`, and `-K` are required for running `PK_AES`. Additionally `-E`, `-C`, or `-G` must be chosen to indicate which block cipher mode is being used. Each block cipher mode has the additional arguments below. **Unless marked otherwise all arguments below are required for their respective block cipher mode.**

### `-E` Arguments

`-K [Key String]` Input string can either be in hex 0xXXXXX format or a regular string. Key needs to be 128, 192, or 256 bits long.

### `-C` Arguments

`-K [Key String]` Input string can either be in hex 0xXXXXX... format or a regular string. Key needs to be 128, 192, or 256 bits long.
`-V [Initialization Vector]` Input string can either be in hex 0xXXXXX... format or a regular string. Initialization vector needs to be 128 bits long.

### `-G` Arguments

`-K [Key String]` Input string can either be in hex 0xXXXXX format or a regular string. Key needs to be 128, 192, or 256 bits long.
`-V [seed for n]` (Optional for Encryption) Input string can either be in hex 0xXXXXX... format or a regular string. n seed needs to be 128 bits long.
`-T [Authentication Tag]` (Only for Decryption) Input string can either be in hex 0xXXXXX... format or a regular string. Authentication Tag needs to be 128 bits long.

## Conclusion

Our implementation of the advanced encryption standard adheres to the standard set out by NIST, making it a valid AES implementation for 128, 192, and 256-bit keys. Our implementation allows for the full encryption and decryption of any file in ECB, CBC, and GCM block cipher modes. This implementation process has taught us many things about finite field arithmetic, byte manipulation in C, and the relationship between symmetric encryption ciphers and Block Cipher modes.

## References

Daemen, Joan, and Vincent Rijmen. A Specification for Rijndael, the AES Algorithm. 2003.

Dworkin, Morris. *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. NIST Special Publication (SP) 800-38D, National Institute of Standards and Technology, 28 Nov. 2007. *csrc.nist.gov*, https://doi.org/10.6028/NIST.SP.800-38D.

National Institute of Standards and Technology (US). Advanced Encryption Standard (AES). NIST FIPS 197-upd1, National Institute of Standards and Technology (U.S.), 9 May 2023, p. NIST FIPS 197-upd1. DOI.org (Crossref), https://doi.org/10.6028/NIST.FIPS.197-upd1.

Stallings, W. "Cryptography and Network Security: Principles and Practices." Third Edition, Pearson

Matt Crypto, Public domain, via Wikimedia Commons