# How to Make AES Algorithm Using Golang

Junsu Lim[1]

***Abstract*— In the AES, each round is encrypted and decrypted as a round function, and each round function is made up of SubBytes, ShiftRows, MixColumns and AddRoundKey. Understanding each sub-function of the round function and implementing the operating principle directly in code to confirm the results. CBC and CTR modes of operation were applied to the implemented AES to confirm that encryption and decryption were carried out on a block-by-block basis when an alphabet of 160 or less was entered in AES-128. Furthermore, I compared the performance of CBC and CTR using the benchmark provided by Go language, and I could see that the internally parallelized CTR performed better than CBC.**

## I. INTRODUCTION

AES is currently the most commonly used block cipher in real society. It is used in so many areas, such as encrypting, authenticating and generating random numbers through AES. These AES operating principles may seem simple, but they require a lot of understanding mathematically(about number theory). While most of the existing AES codes were written in general-purpose languages such as C, C++, Java and Python, this project developed using Go language(golang) developed by the Google. From now on, I will write Go language as golang.

## II. WHAT IS A GOLANG?

Go language is a programming language developed by the Google. It was first designed in 2007 by Kenneth Lane Thompson, Rob Pike, and Robert Griesemer and was developed in earnest in 2008. The Go language is a universal programming language developed with the goal of fast performance, safety, convenience and easy programming. The big features of Go language are as follows.

- Statically-typed language & Strongly-typed language.
- Compiled language.
- Garbage Collection.
- Concurrency & Parallelism.
- Multi-core environment support.
- Modularization and Packaging system.
- Fast compile speed.

The grammar of the Go language is based on C language, and instead of the complicated grammar of C++, that is pursuing simple and concise grammar. Go language is suitable for developing large and complex applications such as Web browsers, servers, and databases. Typical examples include Docker and Kubernetes. In summary, Go language can be used in large, complex, and frequently maintenance, even if memory management is somewhat loose(because of garbage collection).

*This work is about computer and network security project

In my project, Go language was used to implement AES algorithm to use garbage collection, concurrency, multi-core environment, and package.

## III. AES STRUCTURE

AES is a symmetric key cryptographic algorithm published by NIST(National Institute of Standards and Technology) in December 2001. The selection criteria of the AES presented by the NIST and Standards are three criteria: safety, cost and implementation efficiency. Finally, it was decided that Rijndael best matched the three conditions.

AES is a non-feistel algorithm that outputs 128-bit plaintext as 128-bit cipher-text. Non-feistal means it has an inverse function. With rounds 10, 12, 14 the corresponding key sizes for each round are 128, 192, 256 bits. However, even if the master key is different in size, the round key is all 128 bits. Figure 1 shows the structure of AES-128.

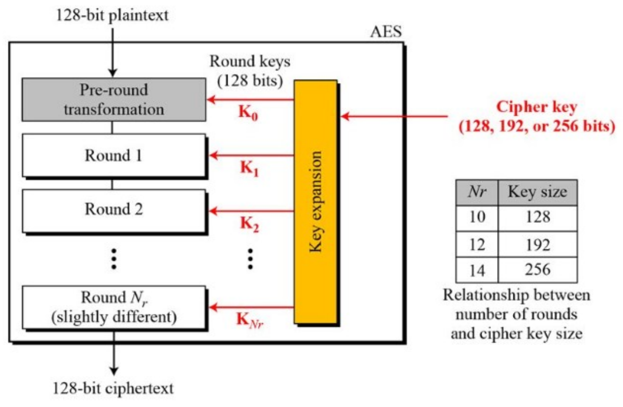


Fig. 1. AES structure

AES is calculated using five units: bit, byte, word, block and state. In AES, the block consists of 128bits and is represented by a state consisting of 16 bytes. When the AES is entered with a plain-text and a secret key, the AES converts the internally entered data into a state. Figure 2 state shows the process.

Listing 1 is a code that converts to a state using golang. In my project, a double for instruction was used to convert the entered data block to state. Figure 2 shows the process of converting block to state.

```go
//Convert text to state(data units used in AES).
  for i := 0; i < 4; i++ {
    for j := 0; j < 4; j++ {
      state[j][i] = paramPlain[i*4+j]
    }
  }
```

Listing 1. Convert block to state

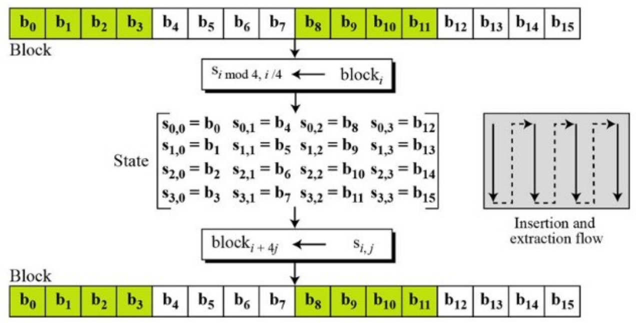

Block
$b_0$ $b_1$ $b_2$ $b_3$ $b_4$ $b_5$ $b_6$ $b_7$ $b_8$ $b_9$ $b_{10}$ $b_{11}$ $b_{12}$ $b_{13}$ $b_{14}$ $b_{15}$
$s_{i \bmod 4,\, i/4} \leftarrow \text{block}_i$
State
$s_{0,0}=b_0$ $s_{0,1}=b_4$ $s_{0,2}=b_8$ $s_{0,3}=b_{12}$
$s_{1,0}=b_1$ $s_{1,1}=b_5$ $s_{1,2}=b_9$ $s_{1,3}=b_{13}$
$s_{2,0}=b_2$ $s_{2,1}=b_6$ $s_{2,2}=b_{10}$ $s_{2,3}=b_{14}$
$s_{3,0}=b_3$ $s_{3,1}=b_7$ $s_{3,2}=b_{11}$ $s_{3,3}=b_{15}$
Insertion and extraction flow
$\text{block}_{i+4j} \leftarrow s_{i,j}$
Block

Fig. 2. AES state

## A. Round function structure

Each transformation takes one state as input, creating the next transformation or another state to be used in the next round. Before the round function is applied, only the AddRoundKey is used, and for the last round, three transformations are used except MixColumns. In decryption, the inverse function of each corresponding function is used for decryption. Figure 3 shows the structure of the round function. To provide safety, the AES algorithm uses four

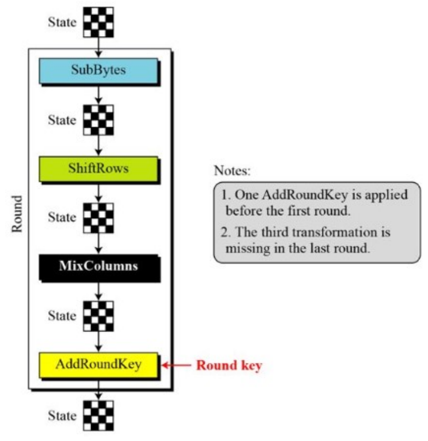
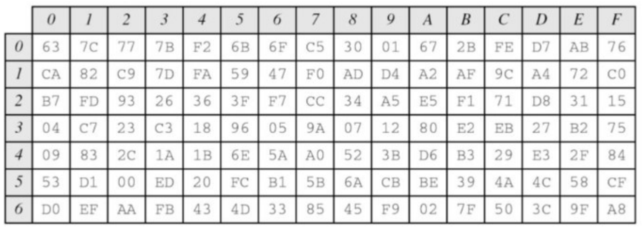

State
SubBytes
State
ShiftRows
State
MixColumns
State
AddRoundKey
Round key
State
Round
Notes:
1. One AddRoundKey is applied before the first round.
2. The third transformation is missing in the last round.

Fig. 3. State code

types of transformations: substituion, permutation, mixing and key-adding. Each of the four types of functions is described below.

## B. SubBytes

SubBytes is a substitution function used in the encryption process of AES. Transformation is performed by reading the left 4 bits as the row of the S-box and the right 4 bits as columns. This has a confusion effect. Confusion means that each binary bit of the cipher-text should depend on several parts of the key, obscuring the connections between the two. The property of confusion hides the relationship between the cipher-text and the key. Figure 4 shows S-BOX. For example, $31_{16}$ and $32_{16}$ are converted to $C7_{16}$, $23_{16}$ by S-BOX, although only one bit is different.



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 63 | 7C | 77 | 7B | F2 | 6B | 6F | C5 | 30 | 01 | 67 | 2B | FE | D7 | AB | 76 |
| 1 | CA | 82 | C9 | 7D | FA | 59 | 47 | F0 | AD | D4 | A2 | AF | 9C | A4 | 72 | C0 |
| 2 | B7 | FD | 93 | 26 | 36 | 3F | F7 | CC | 34 | A5 | E5 | F1 | 71 | D8 | 31 | 15 |
| 3 | 04 | C7 | 23 | C3 | 18 | 96 | 05 | 9A | 07 | 12 | 80 | E2 | EB | 27 | B2 | 75 |
| 4 | 09 | 83 | 2C | 1A | 1B | 6E | 5A | A0 | 52 | 3B | D6 | B3 | 29 | E3 | 2F | 84 |
| 5 | 53 | D1 | 00 | ED | 20 | FC | B1 | 5B | 6A | CB | BE | 39 | 4A | 4C | 58 | CF |
| 6 | D0 | EF | AA | FB | 43 | 4D | 33 | 85 | 45 | F9 | 02 | 7F | 50 | 3C | 9F | A8 |

Fig. 4. S-BOX part

The full S-BOX and full inverse S-BOX will be available on the Internet, so I won't attach them to the report. S-BOX and inverse S-BOX don't change into constant tables. Inverse S-BOX is used in the decryption process.

## C. ShiftRows

Permutation in AES is achieved by ShiftRows. Unlike DES shifting in bits, AES shifts in bytes. Therefore, the order of bits within a byte doesn't change. ShiftRows is used during encryption and moves to the left. The number of shifting at once time depends on the row number. For example, you can see that the first matrix row doesn't move, but the last matrix row moves as much as 3 bytes. InverseShiftRows used in the decryption process don't shift to the left, but shift to the right. Figure 5 below shows the operation process of ShiftRows.

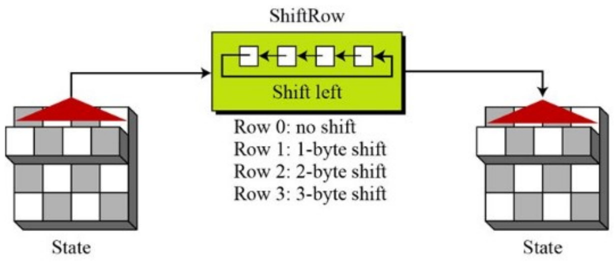


Fig. 5. ShiftRows

Listing 2 shows the implementation of ShiftRows & InverseShiftRows in golang.

```go
//ShiftRows is used during encryption and moves to the left.
//Row 0 : no shift
//Row 1 : 1 byte shift
//Row 2 : 2 byte shift
//Row 3 : 3 byte shift
func shiftRows(state [][]byte) {
  for i := 0; i < 4; i++ {
    for j := 0; j < i; j++ {
      cirCleShiftRows(state[i])
    }
  }
}

//InverseShiftRows is used during decryption and moves to the right.
//Row 0 : no shift
//Row 1 : 1 byte shift
//Row 2 : 2 byte shift
//Row 3 : 3 byte shift
func inverseShiftRows(state [][]byte) {
  for i := 0; i < 4; i++ {
    for j := 0; j < i; j++ {
      inverseCircleShiftRows(state[i])
    }
  }
}
```

Listing 2. ShiftRows code

## D. MixColumns

MixColumns provides a mixing function to provide safety of AES. SubBytes as previously described can be said to be intrabyte transformation. ShiftRows can be said to be byte-exchange transformation. In-byte and byte exchange, but internal transformation of bit units is required. MixColumns provides the requirement. In other words, byte mixing is provided, which gives the diffuse of bit units. Mixing

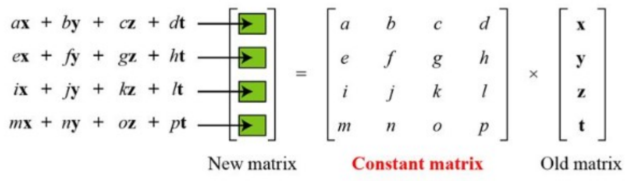


Fig. 6. Mixing the bytes using matrix multiplication.

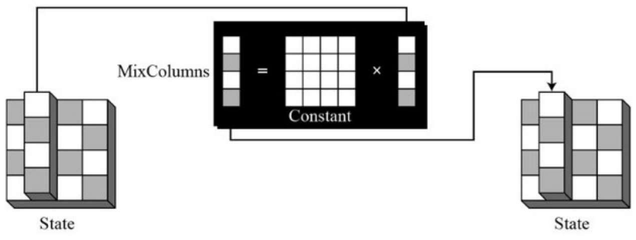


Fig. 7. Mixing the bytes using matrix multiplication (2).

transformation produces four new bytes by mixing bits of each byte with four bytes at a time. Figure 6 and 7 shows the creation of new matrix by multiplying the constant matrix with the old matrix. Constant matrix is given in the standards of the AES.

The MixColumns transformation performs a column unit operation. That is, convert each column of state into a new column. Multiplication of bytes is done in GF($2^8$) with law(10001101) or $x^8$+$x^4$+$x^3$+x+1, and addition is the same as the XOR operation of 8-bit words. The InverseMix-Columns transformation is the same as the MixColumns transformation. Because the constant matrix of MixColumns and constant matrix of InverseMixColumns are inverse relationship to each other.

```
//Mixing transformation takes four bytes at a time
    and combines them to generate four new bytes
//by changing each byte's bits.
//MixColumns gives diffusion in bits.
func mixColumns(state [][]byte) {
  //Constant matrix.
  a := [][]byte{
    {0x02, 0x03, 0x01, 0x01},
    {0x01, 0x02, 0x03, 0x01},
    {0x01, 0x01, 0x02, 0x03},
    {0x03, 0x01, 0x01, 0x02},
  }

  //state column matrix multiplied by 4x4 constant
    matrix.
  //Create new matrix by constant matrix * old
    matrix.
  for i := 0; i < 4; i++ {
    temp := make([]byte, 4)
    for j := 0; j < 4; j++ {
      for k := 0; k < 4; k++ {
        //Give the diffusion in bits.
        temp[j] ^= xTime(state[k][i], a[j][k])
      }

    }
    state[0][i] = temp[0]
    state[1][i] = temp[1]
    state[2][i] = temp[2]
    state[3][i] = temp[3]
  }
}
```

Listing 3. MixColumns code

Listing 3 is a code that implements MixColumns using golang. Multiply the column of the state with the constant matrix as described above to give a diffuse effect in bits. In MixColumns and InverseMixColumns, matrix multiplication is done through xTime(Listing 4). xTime used the algorithm presented by the AES standard.

```
//Multiplication operation in GF(2^8)
func xTime(b, n byte) byte {
  var temp, mask byte = 0, 0x01

  for i := 0; i < 8; i++ {
    if (n & mask) != 0 {
      temp ^= b
    }

    //If x7=0, y=(x<<1)
    //If x7=1, y=(x<<1) ^ 0x1b
    if (b & 0x80) == 0x80 {
      b = (b << 1) ^ 0x1B
    } else {
      b <<= 1
    }
    mask <<= 1
  }
  return temp
}
```

Listing 4. xTime of MixColumns code

Since InverseMixColumns are only different from constant matrix, I will not add the code of InverseMixColmuns here.

## E. Key Expansion

AES generates a round key for each round from the entered secret key. If Nr is called the number of rounds, Nr+1 round Keys are generated from key expansion. The first generated roundKey[0] is XOR by the AddRoundKey function in the pre-round and the remaining roundKey[1] roundKey[Nr] is XOR in the state due to the call of the AddRoundKey function in the final stage of each round function. That is, for AES-128, a round key of 44 words is required. Each word represents 32bits and 4 words represent 128bits. Therefore, it can be seen that 44/4=11 round keys. 52 words for AES-192, 60 words for AES-256. Figure 8 shows the key expansion

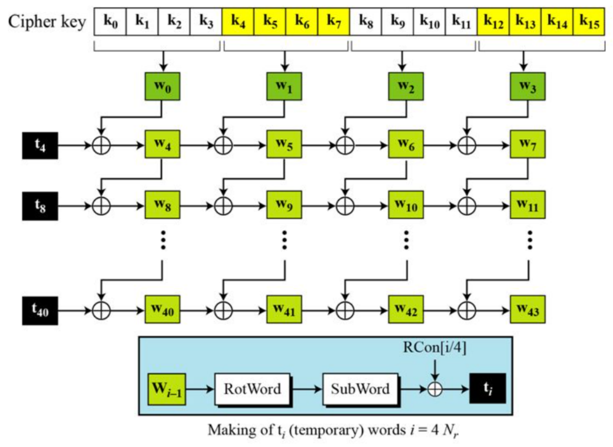


Fig. 8. Key expansion

process in the AES. In Figure 8, you can see that the first 4 words were generated from the entered cryptographic key. In AES-128, the remaining words $(W_i, 4 \leq i \leq 43)$ are generated by classifying them into two cases. First, if $i \bmod 4 \neq 0$, calculate $W_i = W_{i-1} \oplus X_{i-4}$. Second, if $i \bmod 4 = 0$, calculate $W_i = t \oplus W_{i-4}$. t is calculated by the expression $t = SubWord(RotWord(W_{i-1}) \oplus RCon_{i/4})$. If you describe each function in the expression t, RotWord is similar to ShiftRows. Move one byte to the left with just one word input. Listing 5 shows RotWord.

```
//Move 32 bits word by one byte to the left.
func rotWord(W DWORD) DWORD {
  W = ((W & 0xFF000000) >> 24) | (W << 8)
  return W
}
```

Listing 5. RotWord code

SubWord is similar to SubBytes transformation. This function substates each byte of the word using S-Box. Listing 6 shows SubWord.

```
//Extract 8 bits each from MSB of 32 bits word to
    apply S-BOX, and save the applied values from
    MSB.
func subWord(W DWORD) DWORD {
  var out, mask DWORD = 0, 0xFF000000
  var shift byte = 24

  for i := 0; i < 4; i++ {
    //The top four bits of the byte are replaced by
     rows in S-BOX and
    //bottom four bits by columns in S-BOX.
    out += DWORD(_SBox[extractHighHEX(byte((W&mask)
    >>shift))][extractLowHEX(byte((W&mask)>>shift))
    ]) << shift
    mask >>= 8
    shift -= 8
  }
  return out
}
```

Listing 6. SubWord code

RCon is used as a round constant. Listing 7 shows RCon.

```
//Round constants
  rcon = [11]DWORD{
    0x01000000, 0x02000000, 0x04000000, 0x08000000,
    0x10000000, 0x20000000, 0x40000000, 0x80000000,
    0x1b000000, 0x36000000,
  }
```

Listing 7. RCon

```
// Key expansion of AES-128.
// In case of AES-128, which consists of 10 rounds,
     it has 44 words(4 x (Nr+1)).
func keyExpansion(paramKey []byte, W []DWORD) {
  var temp DWORD
  var i int
  // The first four words are made from
     cryptographic keys.
  for i = 0; i < keySize; i++ {
    W[i] = byteToDword(paramKey[4*i], paramKey[4*i
    +1], paramKey[4*i+2], paramKey[4*i+3])
  }
  //Calculate words from 4 to 43.
  i = keySize
  for i < (blockSize * (numberOfRound + 1)) {
    temp = W[i-1]
    // In case of (i mod 4)=0.
    // Wi-1 -> RotWord() -> SubWord() -> ^ RCON[i
    /4] -> ti
    if i%keySize == 0 {
      temp = subWord(rotWord(temp)) ^ rcon[i/
    keySize-1]
      W[i] = W[i-keySize] ^ temp
      i += 1
      // In case of (i mod 4) !=0.
      // Wi = Wi-1 ^ Wi-4
    } else {
      W[i] = W[i-1] ^ W[i-4]
      i += 1
    }
  }
}
```

Listing 8. Key expansion

The entire key expansion code can be seen in Listing 8. The description above will give you an understanding of the key expansion process in AES-128. The generated round key through key expansion will be used in the AddRoundKey function.

## F. AddRoundKey

The substitution, permutation and mixing described above are algorithms that have inverse functions. If the cryptographic key is not added to the state of each round, the attacker can get a plain-text very easily. So AES generates a round key for the encryption key through key expansion and adds each round key inside the round function. This process increases the confusion effect. The AddRoundKey function is a matrix addition that adds round key words to each state's column matrix. The AddRoundKey function is a inverse function for itself because the addition and subtraction operations are the same in GF(2). Figure 9 shows the process of AddRoundKey. Listing 9 is the real implementation code for the AddRoundKey described above.

```
//AddRoundKey adds round key words to the column
    matrix for each state.
//AddRoundKey' operation is matrix addition.
//The AddRoundKey transformation is own inverse
    function on GF(2).
func addRoundKey(state [][]byte, roundKey []DWORD)
    {
  var mask, shift DWORD
```

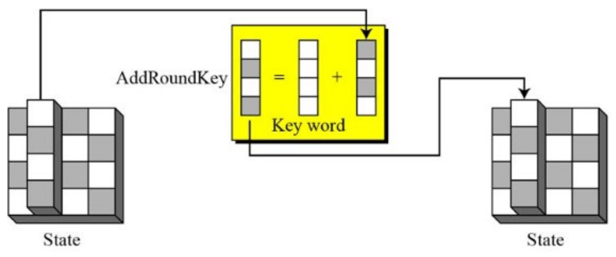


Fig. 9. AddRoundKey

```go

  for i := 0; i < 4; i++ {
    shift = 24
    mask = 0xFF000000
    //From the first column of the state, to
    perform round keys and XOR operations in turn.
    for j := 0; j < 4; j++ {
      state[j][i] = byte((roundKey[i]&mask)>>shift)
     ^ state[j][i]
      mask >>= 8
      shift -= 8
    }
  }
}
```

Listing 9. AddRoundKey

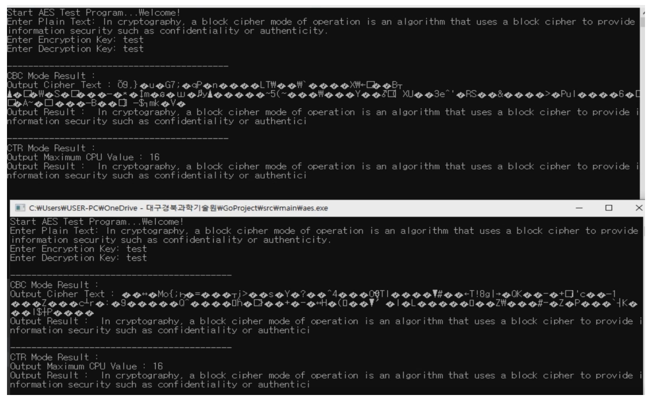
In the decryption process, round keys are applied in reverse order of the encryption process and that used in the AddRoundKey function. The explanations we have followed have shown how AES is implemented and how it is key expansion. Next, we will look at the modes of operation that apply AES to use in the real world and we will look at the AES-CBC and AES-CTR that I have implemented in the AES.

## IV. BLOCK CIPHER MODE OF OPERATION

In cryptography, a block cipher mode of operation is an algorithm that uses a block cipher to provide information security such as confidentiality or authenticity. A mode of operation describes how to repeatedly apply a cipher's single-block operation to securely transform amounts of data larger than a block. As such, a block cipher mode of operation plays a very important role in block ciphers and has a very important impact on security and performance depending on which a block cipher mode of operation is applied. Next, I will explain the CBC and CTR used in this project.

### A. *AES-CBC*

Each block in the CBC performs an XOR operation with the encryption results of the previous block before it is encrypted, and the first block uses IV(Initialization Vector). It is necessary to generate an IV value randomly for each encryption because if the IV value is not random, the cipher may be broken by an attacker. Because for the first cipher-text, IV is used instead of cipher-text, so IV can be the second key. CBC mode is most widely used as the most secure encryption mode. CBC mode has a chain structure inside, so parallel processing is not possible(decryption can be parallelized), and in the event of an error, it can be seen that the error affects the corresponding block and the next block. Also, CBC mode is a probabilistic encryption algorithm that generates statistically different cipher texts each time when the same plain texts is encrypted. The Figure 10 shows the probabilistic encryption. you look at



```
Start AES Test Program...Welcome!
Enter Plain Text: In cryptography, a block cipher mode of operation is an algorithm that uses a block cipher to provide
information security such as confidentiality or authenticity.
Enter Encryption Key: test
Enter Decryption Key: test

-------------------------------------------
CBC Mode Result :
Output Cipher Text : Õ9,}�u�G7;�qP�n����LT₩��₩`����X₩←□��B┬
♟�□�W�S�□���-�*�↨m�ಠ�ш�♫y♟������~5(~���₩����Y���♂□ XU��3e^'�RS��&�����>�Pul�����6�□
□�A~�□����-B��□ -$┐mk�V�
Output Result :  In cryptography, a block cipher mode of operation is an algorithm that uses a block cipher to provide i
nformation security such as confidentiality or authentici

-------------------------------------------
CTR Mode Result :
Output Maximum CPU Value : 16
Output Result :  In cryptography, a block cipher mode of operation is an algorithm that uses a block cipher to provide i
nformation security such as confidentiality or authentici
```

C:₩Users₩USER-PC₩OneDrive - 대구경북과학기술원₩GoProject₩src₩main₩aes.exe

```
Start AES Test Program...Welcome!
Enter Plain Text: In cryptography, a block cipher mode of operation is an algorithm that uses a block cipher to provide
information security such as confidentiality or authenticity.
Enter Encryption Key: test
Enter Decryption Key: test

-------------------------------------------
CBC Mode Result :
Output Cipher Text : ���↔�Mo{;ㅑ�=����┬j>���s�Y�?��^4����0ȩTl�����▼#���←T!8gl→�QK���-�+□'c���-1
����Z����c┴r�:�9�������Q^�����▯ĥ�□���+�-�↔H�(▯���▼' �l�L�������▯���Z₩����#-�Z�P����`┤K�
��l$╂P����
Output Result :  In cryptography, a block cipher mode of operation is an algorithm that uses a block cipher to provide i
nformation security such as confidentiality or authentici

-------------------------------------------
CTR Mode Result :
Output Maximum CPU Value : 16
Output Result :  In cryptography, a block cipher mode of operation is an algorithm that uses a block cipher to provide i
nformation security such as confidentiality or authentici
```

Fig. 10. AES-CBC result

the cipher-text on the CBC Mode Result, you can see that the statistically different cipher text is generated even if the same sentence is encrypted with input. Listing 10 shows implemented CBC code using golang.

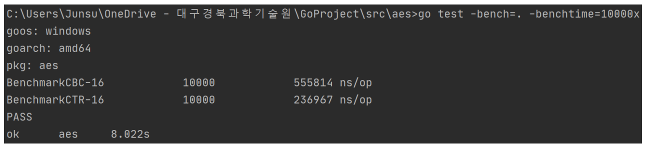
```go
//Implementing the most secure CBC in block
    encryption mode.
//Each block in the plain-text is computed through
    XOR operation with the previous cipher-text,
//and IV is used instead of cipher-text for the
    first cipher-text, IV can be the second key.
//Encryption must be performed sequentially, not in
     parallel.
//Thus the error propagates to the corresponding
    block of the broken cipher-text and to the
    plain-text of the next block.
func EncryptCbcMode(paramPlainText []byte, paramKey
     []byte) {
  plainText := make([]byte, 160)
  cipherText := make([]byte, 160)
  secretKey := make([]byte, 16)
  initializeVector = make([]byte, 16)
  var i = 0

  //Create initialize vector(IV).
  createInitializeVector(initializeVector)

  copy(plainText, paramPlainText)
  _, plainText = trimSpace(plainText)

  copy(secretKey, paramKey)
  _, secretKey = trimSpace(secretKey)

  //plain-text XOR IV.
  for j := 0; j < blockSize*4; j++ {
    plainText[j] = plainText[j] ^ initializeVector[
     j]
  }

  EncryptAES(plainText[i*blockSize*4:(i+1)*
    blockSize*4], cipherText, secretKey)

  for i = 1; i < 10; i++ {
    //previous cipher text XOR plain text.
    for j := 0; j < blockSize*4; j++ {
      //chain structure.
33          plainText[(blockSize*4*i)+j] = plainText[(blockSize*4*i)+j] ^ cipherText[(blockSize*4*(i-1))+j]
34        }
35        EncryptAES(plainText[i*blockSize*4:(i+1)*blockSize*4], cipherText[i*blockSize*4:(i+1)*blockSize*4], secretKey)
36      }
37    }
```

Listing 10. CBC code

The 33rd line of Listing 10 shows that the chain structure is constructed using slice(such as array). Since the CBC decryption process is the reverse of the encryption process, I will skip the detailed code here.

## B. AES-CTR

The counter mode has a structure that replaces the block cipher with the stream cipher. The counter mode takes the value of the current block number for each block and combines the number and nonce(cryptographically secure nonce) to use it as an input to the block cipher module. Counter mode allows parallelization because encryption and decryption of each block do not depend on the previous block. That is, in CTR mode, encryption and decryption can be run in parallel, and in decryption, encryption module is used instead of decryption module. This results in lower initial design costs than other block cipher mode of operation. The CTR can be run in parallel because there is no chain structure internally, and the error does not propagate to the next block. Figure 11 shows the performance analysis results in CBC and CTR. Benchmarks in golang are very important features that



```
C:\Users\Junsu\OneDrive - 대구경북과학기술원\GoProject\src\aes>go test -bench=. -benchtime=10000x
goos: windows
goarch: amd64
pkg: aes
BenchmarkCBC-16            10000            555814 ns/op
BenchmarkCTR-16            10000            236967 ns/op
PASS
ok      aes     8.022s
```

Fig. 11. AES-CTR result

allow us to empirical check the performance of code. Figure 11 also used benchmarks to analyze performance in CBC and CTR. Before the performance analysis, the environment in which the program ran was Inter i9-9980HK, 2.40GHz, and 16 core. The internal code of the CTR is programmed in parallel. Thus, as the CPU performance increases, so increases the AES-CTR performance.

We can see Figure 11 the name of the test followed by the number of allocated processor number. Next is the number of iterations run. Then we see the ns/op column, this is the average time each function call takes to complete. Therefore, Figure 11 can confirm that both CBC and CTR are executed 10,000 times each, and that the CBC's average execution time is 555 microseconds per encryption/decryption and the CTR is 236 microseconds. This parallel programming can confirm the increased performance of the algorithm as an empirical result. The following is the implementation code of CTR.

```
1  //The encryption/decryption of CTR mode becomes a completely identical structure, making it simple to implement.
2  //In CTR mode, the order of blocks can be randomly encryption/decryption. That means parallel processing is possible,
3  //and I used goroutine of golang.
4  func EncryptCtrMode(paramPlainText []byte, paramKey []byte) {
5    plainText := make([]byte, 160)
6    cipherText = make([]byte, 160)
7    secretKey := make([]byte, 16)
8    encNonce = make([]byte, 16)
9    decNonce = make([]byte, 16)
10   _nonce := make([]byte, 16)
11   wg = new(sync.WaitGroup)
12   var temp uint64 = 0
13
14   //Create cryptographically secure nonce.
15   createNonce(encNonce)
16   //Maximum CPU usage.
17   runtime.GOMAXPROCS(runtime.NumCPU())
18   //fmt.Printf("%s : %d", "Output Maximum CPU Value", runtime.GOMAXPROCS(0))
19   //fmt.Printf("%s", "\n")
20
21   copy(plainText, paramPlainText)
22   _, plainText = trimSpace(plainText)
23
24   copy(secretKey, paramKey)
25   _, secretKey = trimSpace(secretKey)
26
27   for i := 0; i < 10; i++ {
28     //Add 1 as wg.Add for each iteration.
29     wg.Add(1)
30     //Concatenate 64 bits nonce and 64 bits counter. (nonce | counter)
31     temp |= uint64(i)
32     //Convert uint64 data to slice.
33     uint64ToByteSlice(temp, encNonce)
34     copy(_nonce, encNonce)
35     //Run encryption simultaneously with 10 goroutine.
36     go subEncryptCtrMode(plainText[i*blockSize*4:(i+1)*blockSize*4], cipherText[i*blockSize*4:(i+1)*blockSize*4], secretKey, _nonce)
37   }
38   //Waiting until all the goroutine is finished.
39   wg.Wait()
40 }
```

Listing 11. CTR code

If you look at the CTR code of the 36th line in Listing 11, you can see that the sub-function is called using the goroutine(thread) of the golang. You can see that a total of 10 goroutines are generated and that code distributed and executed in 16 cores. The decryption uses the encryption module as described above, also using the same structure, so I will skip decryption code in here.

## V. FUTURE WORK

In this project, the AES-128 was implemented using golang. The program should be modified in the future to expand to support AES-192, AES-256. Also, synchronization problem sometimes occur when parallelizing in CTR mode, which makes the wrong results. I tried to protect the area where the lace condition occurred with mutex, but mutex did not slove the problem. Assumed, there seems to be a problem when the goroutine is assigned to the core. Therefore, further research is needed on this part.

## VI. CONCLUSIONs

AES is the most commonly used block cipher in modern society. However, the AES implementation is very complex, and each one has a mathematical meaning. Just by learning over the internet, you cannot understand the powerful performance and service of AES. During this project, I implemented AES directly through AES standards and books(not using external libraries) from beginning to end, and I was able to understand the principles of AES. Furthermore, AES's performance was analyzed in typical CBC and CTR by applying a block cipher mode of operation to AES, and the implementation principles of each mode were understood. Through this project, I was able to understand the block ciphers, and what I learned this lecture in the field I work in will be very helpful.

## REFERENCES

[1] Behrouz A. Forouzan, 암호학과 네트워크 보안, 서울, 한국: McGraw-Hill Education Korea, 2008, pp.193-223, pp.227-240.

[2] William Stallings, Cryptography and Network Security, 7/E. New York, NA: Pearson Higher Education, 2016, pp.171-206.

[3] 이재홍, 가장 빨리 만나는 Go언어, 서울, 한국: 도서출판 길벗, 2016, pp.13-354.

[4] wikipedia, "Confusion and diffusion," April, 2020. Available: https://en.wikipedia.org/wiki/Confusion_and_diffusion.

[5] wikipedia, "Block cipher mode of operation", May, 2020. Available: https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation.