**Tyler Noble**

**04/15/2023**

**Dr. Calvert**

**Spring 2023**

# CCM Implementation In C – CS 395 Independent Study

## I. Introduction

Counter Mode with CBC-MAC (CCM) is an advanced cryptographic technique that provides both encryption and authentication for sensitive data. CCM combines the best of two well-established cryptographic primitives: the Counter mode (CTR) for encryption and the Cipher Block Chaining Message Authentication Code (CBC-MAC) for authentication. CCM is widely utilized in a variety of applications, including wireless communications, due to its efficiency and security. As a symmetric key encryption algorithm, it is specifically designed for use with block ciphers, such as the Advanced Encryption Standard (AES). CCM is highly valued for its ability to ensure data confidentiality and integrity with a single pass through the data. This dual functionality provides robust security, making it an ideal choice for applications that require efficient and reliable encryption and authentication. Examples of such applications include secure communication protocols like the Internet Protocol Security (IPsec) and the Wi-Fi Protected Access (WPA) standard, as well as military communications and IoT devices. With CCM, users can trust that their data remains confidential and unaltered in transit, thus reducing the risk of data tampering, information leakage, or unauthorized access.

A CCM message consists of three distinct components: the nonce, the associated data, and the plaintext or payload. The nonce, also known as the initialization vector (IV), is a unique value generated for each message to ensure that the encryption process remains unpredictable. It is critical to use a different nonce for every message encrypted under the same key, as reusing a nonce could compromise the security of the system. The associated data (AD) is an optional component that contains information that needs to be authenticated but not encrypted. This could include TCP/UDP packet header information or metadata that helps identify the context of the message. The plaintext is the actual data to be encrypted and authenticated. When encrypting a message with CCM, the block cipher (e.g., AES) operates in counter mode (CTR) using the nonce as the initial counter. This process generates a key stream, which is then XORed with the plaintext to produce the ciphertext. Concurrently, the message authentication code (MAC) is generated by processing the entire message using the cipher block chaining message authentication code (CBC-MAC) mode. The final CCM message is formed by concatenating the ciphertext and the MAC. During decryption, the recipient uses the same process in reverse, first authenticating the received message using the associated data and MAC, and then decrypting the ciphertext using the key stream generated by the counter mode. This structure ensures that CCM messages maintain confidentiality through payload encryption while also ensuring data integrity and message authentication through the MAC.

CCM offers several advantages over other encryption and authentication methods, making it an attractive choice for many applications. One key advantage is its efficiency, as it can perform both encryption and authentication in a single pass through the data, reducing the time and resources required for secure communication. This is especially beneficial for resource-constrained devices and environments, such as IoT networks. Additionally, CCM's construction using widely accepted cryptographic primitives, like CTR and CBC-MAC, lends credibility to its security, as it combines proven techniques to achieve its dual functionality. However, CCM is not without its drawbacks. One of the primary disadvantages is its reliance on a single key for both encryption and authentication, which can introduce vulnerabilities if not managed carefully. In particular, it is essential to ensure that the key is never reused for different sets of data. Moreover, compared to some other authenticated encryption modes, like GCM (Galois/Counter Mode), CCM may exhibit lower performance and throughput due to its more complex computation requirements. Lastly, CCM's encryption and authentication processes are closely tied, meaning that if an issue arises with one component, it can potentially compromise the security of the entire scheme. Despite these drawbacks, CCM remains a widely used and trusted method for providing secure data communication.

## II. Project

The project discussed in this paper is the implementation of the CCM algorithm in C. Particularly replicating the steps of sending a message modified by CCM and receiving it via two mock applications: *consumer.c* and *producer.c*. This is possible by including the module *ccm.c* during the compilation of both applications. The CCM module is responsible for providing methods to construct and send the CCM messages as well as to receive and deconstruct those messages. Each of the applications designed for this project are executed on a Linux platform and send packets using the TCP protocol. It should be noted that there are a few key differences in this implementation compared to the exact CCM algorithm. These differences are:

- Block cipher plugins: The block cipher encryption algorithm used by the CCM module is provided by a function pointer which can be any valid 128-bit block cipher.
- 128-bit MAC length: CCM specifies that this length can vary, but for simplifying the implementation, a fixed length of an entire block was used.
- Nonce requirement: The Nonce is always included in the CCM structure at a fixed length of 7 bytes.
- Associated Data requirement: The AD is always included, and the length cannot exceed 62580 bytes
- 8-byte payload length: This limits the size of the payload to $2^{64}$

Another note should recognize that the only block cipher tested was a simple flip cipher which inverts the bits for a provided block. In theory, a more standardized and secure method could be used such as AES, but this was not tested due to time limitations.

### II a. Session Control Block (SCB)

During the creation of this implementation, it was evident that the applications should have a simple relationship with the CCM module. That is, the innerworkings of sending and receiving these messages should not be the responsibility of the applications themselves. Instead, the applications can fill a data structure with all of the necessary information to send or receive a message and pass that to the CCM module when needed. This solution was named the Session Control Block or SCB. The following code snippet from *ccm.c* represents the information required by the module to properly send a message.

```
typedef void (*blk_cipher)(char k[16], char* p, char* c);
typedef char blk[16];

struct SCB {

  int socket_fd;
  blk key;
  blk_cipher encrypt_block;

};

struct SCB* session_setup(int socket_fd, blk key, blk_cipher encrypt_block) {


  struct SCB* scb = malloc(sizeof(struct SCB));
  scb->socket_fd = socket_fd;
  scb->encrypt_block = encrypt_block;
  memcpy(scb->key, key, BLOCK_LEN);

  return scb;

}

void session_clear(struct SCB* scb) {

  scb->socket_fd = 0;
  scb->encrypt_block = 0;
  clear_bytes(scb->key, 0, BLOCK_LEN);

}

void session_destroy(struct SCB* scb) {

  session_clear(scb);
  free(scb);
}
```

The SCB structure contains a socket descriptor created by the application, a 128-bit encryption key, and a pointer to a block cipher algorithm that will be used in the MAC computation and payload encryption processes. Each application must create an SCB before calling the CCM module which can be done via the session_setup function which allocates and returns a pointer to a newly created SCB. This module also provides session_clear and session_destroy functions to aid the application in discarding or resetting the structure.

### II b. Producer Application

The producer application diagram follows:

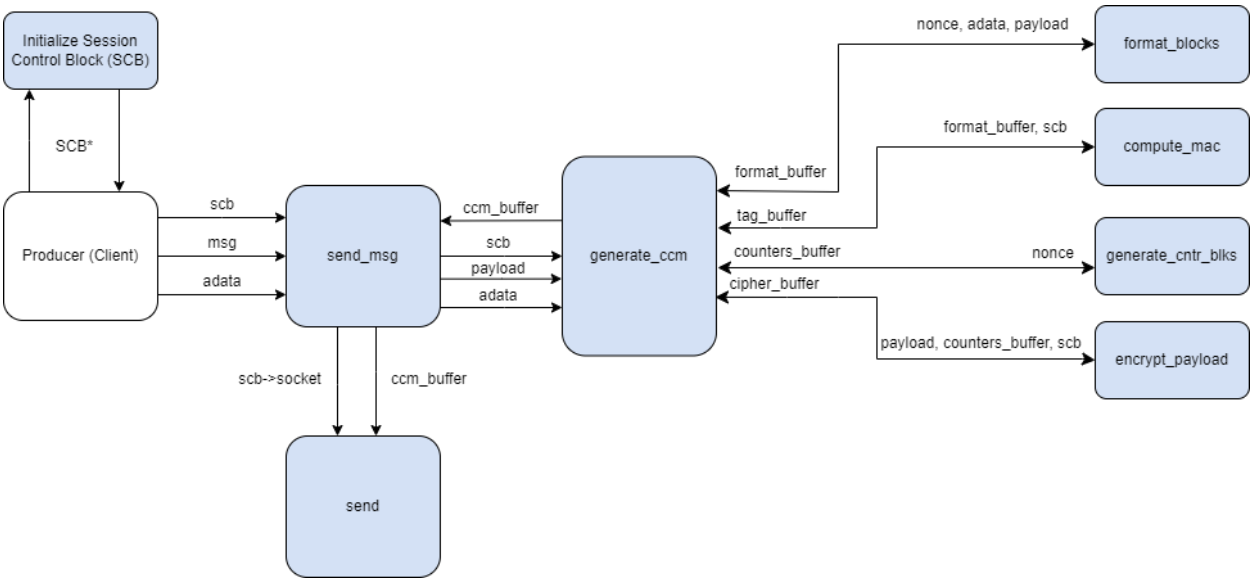



*Figure 1: producer.c mock application diagram*

Each blue block represents a function in the CCM module created for this project. The producer first creates a socket descriptor using the socket() function. Then a sockaddr_in structure is created to store the consumer's address information which is as follows:

- sin_family: set to AF_INET to use IPv4
- sin_port: set to the given port number, converted to network byte order using the htons() function
- sin_addr.s_addr: set to the IPv4 loopback address (127.0.0.1) using the inet_addr() function, which converts the string representation of an IPv4 address to a 32-bit integer in network byte order.

With the TCP socket set up, the producer creates an SCB to pass to the CCM module. Directly after the SCB creation, a connection request is sent out to the consumer using the send() function. Once a connection is established, now all that is required is a payload and associated data to send to the consumer. The associated data is kept constant for each compilation usually

set to the string “associated data stuff!” with a length of 23 bytes. The payload is specified by user input which waits until the user has typed a string of characters ending with a newline character. The following code represents the main functionality of the producer mock application:

```
for (;;) {

    // reset length for each msg
    msg_len = 0;
    clear_bytes(msg, 0, msg_len);

    // user input
    printf("Consumer Input: ");
    fgets(msg, MAX_MSG_LEN, stdin);

    // quit program
    if (!strcmp(msg, "quit\n"))
        break;

    // count char length for user input
    while (msg[msg_len] != '\0')
        msg_len++;


    int ret = send_msg(msg, msg_len, adata, adata_len, scb);

  }
```

The message length is measured after each user input, and the send_msg() function is called from the CCM module. This function will send the message using the information provided in the SCB structure passed. After a message is sent, the buffer is cleared, and the user input is requested again until the string “quit” is recognized which will close the socket and exit the program.

### II c. CCM Module - Producer

Just as designed, the producer application only calls two functions total during a normal execution. That is the SCB setup function and the send message function. This creates a simple interaction for the application and leaves the complexities of CCM up to the module itself. In Figure 1, the flow of the CCM module’s inner workings can be seen. As the send message function is invoked, the specifics of the CCM algorithm can be completed. First, the send message function pads the payload to fit into 128-bit blocks and then prepends the associated

data buffer with its length before padding it as well. It then invokes the generate_ccm function passing it the padded buffers, their pre-padded lengths, and the scb like so:

```
int send_msg(char* payload, unsigned long payload_len, char* adata, unsigned long adata_len,
struct SCB* scb) {

  // add 0 padding to payload
  unsigned long pad_payload_len = find_pad_len(payload_len);
  char* tmp_payload = malloc(pad_payload_len);
  memcpy(tmp_payload, payload, payload_len);
  clear_bytes(tmp_payload, payload_len, pad_payload_len);

  // add length concatenation and 0 padding to associated data
  unsigned long len_pad = 0;
  if (adata_len < 65536) {
    len_pad = 2;
  }
  else if (adata_len < 4294967296) {
    len_pad = 6;
  }
  else {
    len_pad = 10;
  }

  unsigned long pad_adata_len = find_pad_len(adata_len + len_pad);
  char* tmp_adata = malloc(pad_adata_len);
  clear_bytes(tmp_adata, 0, pad_adata_len);
  memcpy(&tmp_adata[len_pad], adata, adata_len);

  unsigned int adata_len_hl = ntohl(adata_len);
  char* adata_len_char = (char*)&adata_len_hl;
  memcpy(tmp_adata, &adata_len_char[2], len_pad); // only works for len < 4 bytes

  if (payload == NULL || adata == NULL) {
    printf("There was an issue while formatting the message... :(");
    return -1;
  }

  // ctrl-info block + associated data + payload + + TAG
  unsigned long data_len = HEADER_LEN + pad_adata_len + pad_payload_len + TAG_LEN;
  char* data_buffer = malloc(data_len);

  generate_ccm(tmp_adata, adata_len + len_pad, tmp_payload, payload_len, data_buffer,
data_len, scb);
```

It should be noted that the associated data length prepending was not completely implemented and currently only works with lengths represented by 4 bytes or less. The generate_ccm function then takes the padded data and recalculates the padded lengths to be used later. The first step in generating this CCM module is taken which is the creation of the nonce. This module uses the getrandom() system call to generate the buffer of random bits. As specified before, this length is fixed to 7 bytes which is allocated before the nonce generation. With all of the appropriate data for a CCM message, the format_blocks() function is invoked to format the message into blocks, add control info, and create the remainder of the header to the CCM message. A buffer is returned which holds the newly formatted data in 128-bit blocks. The following code demonstrates the beginning functionality of generate_ccm():

```
void generate_ccm(char* adata, unsigned long adata_len, char* payload, unsigned long
payload_len, char* final_buffer, unsigned long final_buffer_len, struct SCB* scb) {

  // initializing buffer lengths
  unsigned long format_buffer_len = final_buffer_len - TAG_LEN;
  unsigned long format_buffer_blocks = format_buffer_len / BLOCK_LEN;

  unsigned long pad_payload_len = find_pad_len(payload_len);
  unsigned long pad_adata_len = find_pad_len(adata_len);

  // number of assocated_data/payload 128-bit blocks
  unsigned long num_adata_blocks = pad_adata_len / BLOCK_LEN;
  unsigned long num_payload_blocks = pad_payload_len / BLOCK_LEN;

  // generate nonce
  char* nonce = malloc(NONCE_LEN);
  int ret = getrandom(nonce, NONCE_LEN, GRND_RANDOM);
  if (ret != NONCE_LEN)
     perror("getrandom failed!");

  // allocating buffers
  char* format_buffer = malloc(format_buffer_len);
  char* tag_buffer = malloc(TAG_LEN);
  char* counters_buffer = malloc(pad_payload_len);
  char* cipher_buffer = malloc(pad_payload_len);

  // clear format buffer before using
  clear_bytes(format_buffer, 0, format_buffer_len);

  format_blocks(nonce, adata, pad_adata_len, payload, payload_len, format_buffer); // format
into 128-bit blocks
```

The buffer format_buffer now holds the data in the required block format for CBC-MAC. The next step is to calculate the MAC by invoking the compute_mac() function. This function uses the specifications of CBC-MAC to generate a 128-bit tag to append to the final CCM message like so:

```
void compute_mac(char* data, unsigned long data_len, struct SCB* scb, char* tag_buffer) {

  // encrypt the first block b0
  (*(scb->encrypt_block))(scb->key, data, tag_buffer);

  // iterate through each block of data
  for (unsigned long i = BLOCK_LEN; i < data_len; i += BLOCK_LEN) {

    xor_blocks(tag_buffer, &data[i], tag_buffer); // XOR with previous block
    (*(scb->encrypt_block))(scb->key, tag_buffer, tag_buffer); // encrypt XORed block
  }
}
```

This function utilizes the block cipher specified in the SCB and the xor_blocks function provided by the internal module. With those tools, the MAC can be computed fairly easily and the tag buffer is filled with the 128-bit computed MAC. The next step is to encrypt the payload before sending the data. This is done using the counter mode blocks. These blocks are generated using the generate_cntr_blks() function. Since the blocks are initialized from the nonce and the fixed payload length, there are only two parameters required for this function. The blocks are generated in a simple algorithm which creates one block for each payload block. The blocks consist of a byte of flags followed by the nonce. The remainder of the block is reserved for a counter initialized at 0. Each block generated after the initial block increments that counter by 1 until the necessary number of blocks have been made. The following code demonstrates this behavior:

```
void generate_cntr_blks(unsigned long counter_buffer_len, char* nonce, char* counter_buffer) {

  char flags = 7; // fixed payload length
  unsigned char* counter = malloc(15-NONCE_LEN);
  clear_bytes(counter, 0, 15-NONCE_LEN);

  // fill buffer with generated blocks
  for (int i = 0; i < counter_buffer_len; i += BLOCK_LEN) {

    memcpy(&counter_buffer[i], &flags, 1); // add flags
    memcpy(&counter_buffer[i+1], nonce, NONCE_LEN); // add nonce
    memcpy(&counter_buffer[i+1+NONCE_LEN], counter, 15-NONCE_LEN); // add ctr
    increment_cntr(counter, 15-NONCE_LEN); // increment ctr

  }
```

```
  free(counter);

}
```

With the help of an allocated buffer, the counter can be added to the end of each counter block and a final counter buffer can be filled. Equipped with the counter blocks, the block cipher algorithm, and the unencrypted payload, the final CCM algorithm step is ready for execution. That is, to encrypt the payload via the encrypt_payload() function. This function first encrypts each counter block that is then XORed with the corresponding payload block it was generated for. The result is the encrypted payload block or ciphertext that will be sent along with the final message. The code below demonstrates this behavior:

```
void encrypt_payload(char* payload_buffer, unsigned long payload_buffer_len, char*
counters_buffer, char* cipher_buffer, struct SCB* scb) {

  // for each payload block
  for (int i = 0; i < payload_buffer_len; i += BLOCK_LEN) {
    // encrypt the counter block
    (*(scb->encrypt_block))(scb->key, &counters_buffer[i], &counters_buffer[i]);
    // xor with unencrypted payload
    xor_blocks(&payload_buffer[i], &counters_buffer[i], &cipher_buffer[i]);
  }

}
```

All the steps to the CCM algorithm have been complete, so the message now needs to be placed into a final buffer before being sent out as TCP packets. The generate_ccm() function takes care of rearranging each piece of the CCM message into a final buffer before freeing each of the allocated buffers that were used in generation of the message. The buffer is handed back to send_msg() where the socket descriptor is used to send the final CCM buffer to the consumer before also freeing any memory that was allocated during its execution.

## II d. Consumer Application

The consumer application diagram follows:

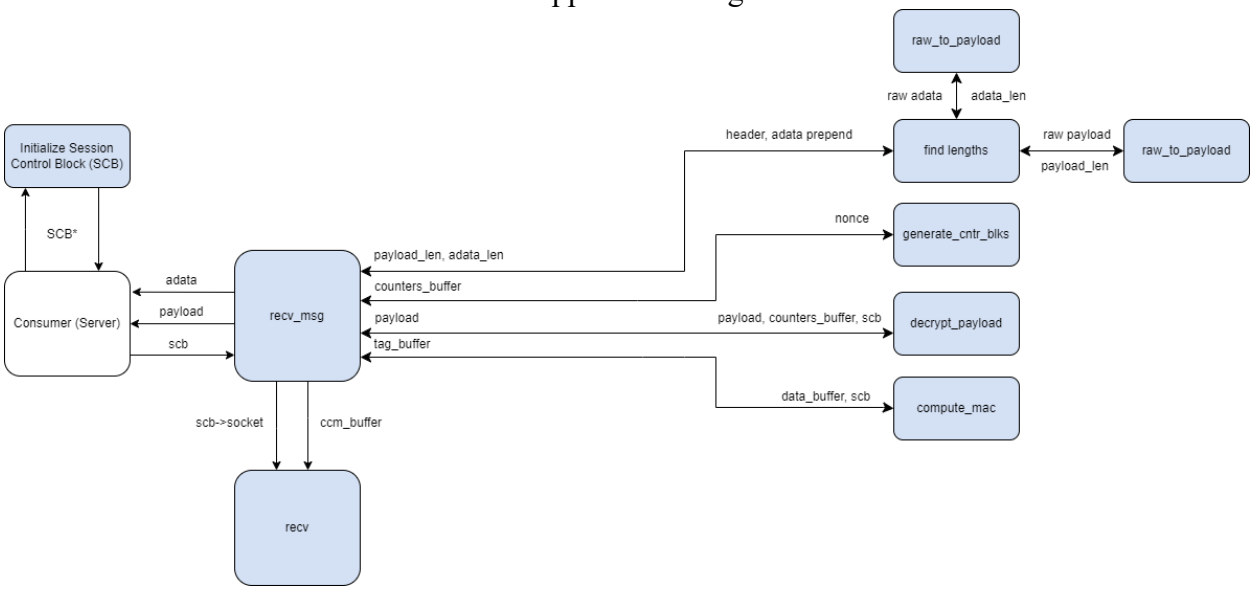



*Figure 2: consumer.c mock application diagram*

Similar to the producer, the consumer must also setup an SCB before using the CCM module to send messages. To start, the consumer starts by creating a socket and receiving it's socket descriptor. It then initializes the server address structure with the IPv4 family, a provided port number, and the loopback address. This program then attempts to bind to the socket described earlier and exits if it fails to do so. After binding successfully, the listen() function is invoked to await a connection request from the producer. Once a request has been recognized, the accept() function completes the TCP connection to the producer and a message is displayed with the IP address and port number of the connection. The following code snippet is what the consumer uses to do so:

```
// create socket
int socket_dsc = socket(AF_INET, SOCK_STREAM, 0);
struct sockaddr_in client_addr;

// initialize server address structure
struct sockaddr_in server_addr;
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(port);
server_addr.sin_addr.s_addr = inet_addr("127.0.0.1");

// bind to port
if(bind(socket_dsc, (struct sockaddr*)&server_addr, sizeof(server_addr))<0){
    printf("Couldn't bind to the port\n");
    return -1;
}
```

```
    printf("Done with binding\n");

    // listen for incoming connections
    if(listen(socket_dsc, 1) < 0) {
        printf("Error while listening\n");
        return -1;
    }

    printf("\nListening for incoming connections.....\n");

    // accept connection req and save socket descriptor
    int client_size = sizeof(client_addr);
    int client_sock = accept(socket_dsc, (struct sockaddr*)&client_addr, &client_size);

    // error if invalid socket
    if (client_sock < 0){
        printf("Can't accept\n");
        return -1;
    }

    // setup SCB for receiving transmmissions
    struct SCB* scb = session_setup(client_sock, "12345678", &flip_cipher);


    printf("Producer connected at IP: %s and port: %i\n", inet_ntoa(client_addr.sin_addr),
ntohs(client_addr.sin_port));
```

This setup for the consumer program is most of the work, as the following code snippet represents the remainder of the program:

```
    struct vbuf* adata = vbuf_create();
    struct vbuf* msg = vbuf_create();

    for (;;) {
        recv_msg(adata, msg, scb);
        vbuf_free(adata);
        vbuf_free(msg);
    }

    close(client_sock);
    close(socket_dsc);
```

Here, two structures are created to hold the associated data and payload received from the CCM after deconstruction. Because the lengths are unknown at compile time, vbuf structures were created that can keep track of the lengths of each character array like so:

```
struct vbuf {

  char* data;
  unsigned long len;

};
```

The general structure of the consumer logic is to call recv_msg() from the CCM module which inherently loops until a full message is received, and then reset each vbuf before repeating the process.

### II e. CCM Module – Consumer

The technical involvement of the consumer is like the producer by design in that only one function is called to fully receive a message. The right tools are provided through the SCB structure initialized at the beginning of the consumer program, and from there the CCM does all the heavy lifting returning only the associated data and unencrypted payload from the producer. The recv_msg() function starts by allocating a buffer to hold the CCM message being transmitted. The SCB socket descriptor is then used when looping the recv() function to await packets from the producer. Due to the nature of this implementation, all lengths specified in the message can be observed in the first 24 bytes. This is because the first block of header information provides nonce and payload lengths while the associated data is prepended to the first 8 bytes of the second block. The recv() function returns 8 bytes of information at a time, so the data is received until a total of at least 24 bytes is received like so:

```
int recv_msg(struct vbuf* adata, struct vbuf* payload, struct SCB* scb) {

  char* data_buffer = malloc(MAX_CCM_LEN);
  clear_bytes(data_buffer, 0, MAX_CCM_LEN);
  unsigned long recv_len = 0;
  unsigned long data_len = 0;

  // read ctrl-info, payload len, and adata len
  while (recv_len < 23) {
    recv_len += recv(scb->socket_fd, &data_buffer[recv_len], sizeof(data_buffer), 0);
  }
```

The data_buffer now holds all the necessary information for calculating the total message length. The recv_msg() function uses this partial message to do exactly that:

```
// calculate total message length from header information
unsigned long payload_len = raw_to_payload(&data_buffer[1+NONCE_LEN]);
unsigned long pad_payload_len = find_pad_len(payload_len);
unsigned long adata_len = raw_to_adata(&data_buffer[16]);
unsigned long pad_adata_len = find_pad_len(adata_len);

data_len = HEADER_LEN + pad_adata_len + pad_payload_len + TAG_LEN;

while (recv_len < data_len) {
 recv_len += recv(scb->socket_fd, &data_buffer[recv_len], sizeof(data_buffer), 0);
}
```

The lengths are calculated and the remainder of the message can be received just as above. The deconstruction process of the message can now begin. The last step of the producer was to encrypt the payload, so the consumer must decrypt the received ciphertext to start. For the counter blocks to be generated, the nonce must be extracted first which can be found in the first block of the message. After extracting this, the same generate_cntr_blks() and encrypt_payload() functions are used just as in the producer to unencrypt the ciphertext. This is a nice feature of the CCM algorithm in which the same counter blocks are generated and encrypted using the block cipher. The encrypted counter block XOR with the ciphertext will return the plaintext without the hassle of dealing with a separate decryption process. Now that the CCM has returned to its completely unencrypted form, the MAC can be computed to verify the integrity and authenticity of the message. This MAC is computed in the same process as the producer in which a tag buffer is filled with the 128-bit MAC of the message. Now, the computed MAC can be compared to the received MAC:

```
compute_mac(data_buffer, data_len - TAG_LEN, scb, tag_buffer);

 if (!strncmp(&data_buffer[HEADER_LEN+pad_adata_len+pad_payload_len], tag_buffer,
TAG_LEN)) {
  printf("MAC Calculation Match: authenticity and integrity verified! :D\n\n");
 }
 else {
  printf("MAC Calculation Mismatch: This message has been altered! :O\n\n");
  return -1;
 }
```

If the computed MAC and received MAC match, a message is printed and the function continues execution. In the case that they do not match, the message is discarded and an error value is returned. For a validated message, the two provided vbufs are filled with the necessary data:

```
// filling adata buffer
adata->data = malloc(adata_len);
memcpy(adata->data, &data_buffer[HEADER_LEN+2], adata_len);
adata->len = adata_len;

// filling payload buffer
payload->data = malloc(payload_len);
memcpy(payload->data, &data_buffer[HEADER_LEN+pad_adata_len], payload_len);
payload->len = payload_len;
```

From here, the consumer can use the sent information as it wishes and can sleep well at night knowing it was verified using the CCM algorithm.

## III. Results

This program is executed on two Linux machines where the consumer must first be compiled and awaiting connections before the producer sends a connection request. Once a connection is established, the user will observe this message on the consumer's output:


From here the client is prompted to input a string of characters which will be sent as the payload. In this first test, I will be sending the string "Dr. Calvert". This is the output of the client for this input:

```
Consumer Input: Dr. Calvert

------------------------------------------------------- Starting CBC-MAC w/ Counter Impl
Nonce:  c6 26 1f 8b ed 5c 01
B0:     7f c6 26 1f 8b ed 5c 01 00 00 00 00 00 00 00 0c
B1:     00 17 61 73 73 6f 63 69 61 74 65 64 20 64 61 74


B0E         =  80 39 d9 e0 74 12 a3 fe ff ff ff ff ff ff ff f3
XOR1        =  80 2e b8 93 07 7d c0 97 9e 8b 9a 9b df 9b 9e 87
E1          =  7f d1 47 6c f8 82 3f 68 61 74 65 64 20 64 61 78
XOR2        =  1e f1 34 18 8d e4 59 49 61 74 65 64 20 64 61 78
E2          =  e1 0e cb e7 72 1b a6 b6 9e 8b 9a 9b df 9b 9e 87
XOR3        =  a5 7c e5 c7 31 7a ca c0 fb f9 ee 91 df 9b 9e 87
E3          =  5a 83 1a 38 ce 85 35 3f 04 06 11 6e 20 64 61 78

MAC       =  5a 83 1a 38 ce 85 35 3f 04 06 11 6e 20 64 61 78

CTR Start =  07 c6 26 1f 8b ed 5c 01 00 00 00 00 00 00 00 00
CTR E1      =  f8 39 d9 e0 74 12 a3 fe ff ff ff ff ff ff ff ff
CTR XOR1    =  bc 4b f7 c0 37 73 cf 88 9a 8d 8b f5 ff ff ff ff

                                                          Final CCM Message
B0   =  7f c6 26 1f 8b ed 5c 01 00 00 00 00 00 00 00 0c
B1   =  00 17 61 73 73 6f 63 69 61 74 65 64 20 64 61 74
B2   =  61 20 73 74 75 66 66 21 00 00 00 00 00 00 00 00
B3   =  bc 4b f7 c0 37 73 cf 88 9a 8d 8b f5 ff ff ff ff
B4   =  5a 83 1a 38 ce 85 35 3f 04 06 11 6e 20 64 61 78


Total Length: 80 bytes: [Header-Info, Assoc-Data, Payload, MAC] <--> [16 + 32 + 16 + 16]
-------------------------------------------------------------------------------------------

Consumer Input:
```

And the consumer receives this sent message and outputs the following:

```
lily:~/CS395-CCM> ./server 31333
Done with binding

Listening for incoming connections.....
Producer connected at IP: 127.0.0.1 and port: 35890

------------------------------------------------------ Starting CBC-

                                                                Receivec
B0   =  7f c6 26 1f 8b ed 5c 01 00 00 00 00 00 00 00 0c
B1   =  00 17 61 73 73 6f 63 69 61 74 65 64 20 64 61 74
B2   =  61 20 73 74 75 66 66 21 00 00 00 00 00 00 00 00
B3   =  bc 4b f7 c0 37 73 cf 88 9a 8d 8b f5 ff ff ff ff
B4   =  5a 83 1a 38 ce 85 35 3f 04 06 11 6e 20 64 61 78


CTR Start =  07 c6 26 1f 8b ed 5c 01 00 00 00 00 00 00 00 00
CTR E1      =  f8 39 d9 e0 74 12 a3 fe ff ff ff ff ff ff ff ff
CTR XOR1    =  44 72 2e 20 43 61 6c 76 65 72 74 0a 00 00 00 00


B0E         =  80 39 d9 e0 74 12 a3 fe ff ff ff ff ff ff ff f3
XOR1        =  80 2e b8 93 07 7d c0 97 9e 8b 9a 9b df 9b 9e 87
E1          =  7f d1 47 6c f8 82 3f 68 61 74 65 64 20 64 61 78
XOR2        =  1e f1 34 18 8d e4 59 49 61 74 65 64 20 64 61 78
E2          =  e1 0e cb e7 72 1b a6 b6 9e 8b 9a 9b df 9b 9e 87
XOR3        =  a5 7c e5 c7 31 7a ca c0 fb f9 ee 91 df 9b 9e 87
E3          =  5a 83 1a 38 ce 85 35 3f 04 06 11 6e 20 64 61 78

MAC       =  5a 83 1a 38 ce 85 35 3f 04 06 11 6e 20 64 61 78
RECV MAC  =  5a 83 1a 38 ce 85 35 3f 04 06 11 6e 20 64 61 78
MAC Calculation Match: authenticity and integrity verified! :D

ADATA: associated data stuff!

PAYLOAD: Dr. Calvert

------------------------------------------------------------------
```

It can be observed that the message was successfully transmitted and received. The computed MAC matches the received MAC and the final message is correctly displayed at the bottom of the server output. This CCM module can also be tested with larger inputs such as the Fitness Gram Pacer Test prompt like so:

```
Consumer Input: The FitnessGram™ Pacer Test is a multistage aerobic capacity test that progressively gets more difficult as it continues. The 20 meter pacer tes
t will begin in 30 seconds. Line up at the start. The running speed starts slowly, but gets faster each minute after you hear this signal. [beep] A single lap s
hould be completed each time you hear this sound. [ding] Remember to run in a straight line, and run as long as possible. The second time you fail to complete a
 lap before the sound, your test is over. The test will begin on the word start. On your mark, get ready, start.

------------------------------------------------------------ Starting CBC-MAC w/ Counter Implementation ------------------------------------------------------------
Nonce:  65 92 44 b0 90 e7 81
B0:     7f 65 92 44 b0 90 e7 81 00 00 00 00 00 00 02 43
B1:     00 17 61 73 73 6f 63 69 61 74 65 64 20 64 61 74


B0E         =  80 9a 6d bb 4f 6f 18 7e ff ff ff ff ff ff fd bc
XOR1        =  80 8d 0c c8 3c 00 7b 17 9e 8b 9a 9b df 9b 9c c8
E1          =  7f 72 f3 37 c3 ff 84 e8 61 74 65 64 20 64 63 37
XOR2        =  1e 52 80 43 b6 99 e2 c9 61 74 65 64 20 64 63 37
E2          =  e1 ad 7f bc 49 66 1d 36 9e 8b 9a 9b df 9b 9c c8
XOR3        =  b5 c5 1a 9c 0f 0f 69 58 fb f8 e9 dc ad fa f1 2a
E3          =  4a 3a e5 63 f0 f0 96 a7 04 07 16 23 52 05 0e d5
```

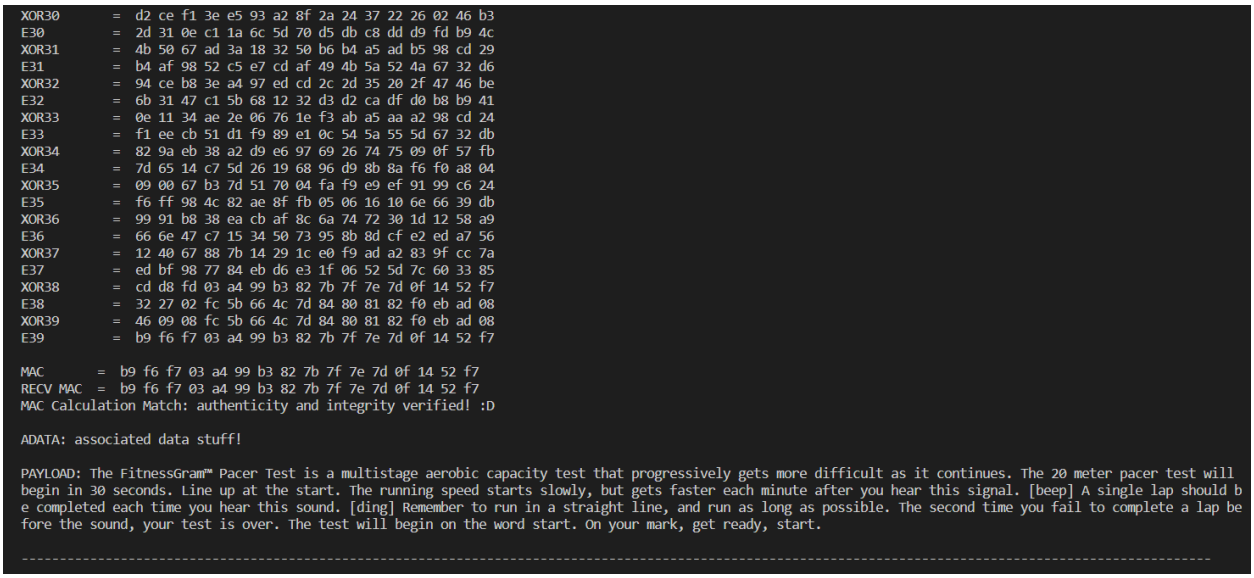
...

```
 B19  =  d8 fb 0b cf 2a 1d 38 07 90 8a df 97 9a 9e 8d cf
 B20  =  8c f2 04 c8 6f 1c 71 19 91 9e 93 d1 df a4 9d 8b
 B21  =  9d ea 30 9b 0e 4f 6b 17 91 98 93 9a df 93 9e 9d
 B22  =  d8 e9 05 d4 3a 03 7c 5e 9d 9a df 9c 90 92 8f 80
 B23  =  9d ee 08 df 6f 0a 79 1d 97 df 8b 96 92 9a df 92
 B24  =  97 ef 4d d3 2a 0e 6a 5e 8b 97 96 8c df 8c 90 9f
 B25  =  96 fe 43 9b 14 0b 71 10 98 a2 df ad 9a 92 9a 84
 B26  =  9a ff 1f 9b 3b 00 38 0c 8a 91 df 96 91 df 9e c8
 B27  =  8b ee 1f da 26 08 70 0a df 93 96 91 9a d3 df 86
 B28  =  96 fe 4d c9 3a 01 38 1f 8c df 93 90 91 98 df 87
 B29  =  8b ba 1d d4 3c 1c 71 1c 93 9a d1 df ab 97 9a c5
 B30  =  8b ff 0e d4 21 0b 38 0a 96 92 9a df 86 90 8a c4
 B31  =  9e fb 04 d7 6f 1b 77 5e 9c 90 92 8f 93 9a 8b 86
 B32  =  d8 fb 4d d7 2e 1f 38 1c 9a 99 90 8d 9a df 8b 8a
 B33  =  9d ba 1e d4 3a 01 7c 52 df 86 90 8a 8d df 8b 84
 B34  =  8b ee 4d d2 3c 4f 77 08 9a 8d d1 df ab 97 9a c0
 B35  =  8c ff 1e cf 6f 18 71 12 93 df 9d 9a 98 96 91 ff
 B36  =  97 f4 4d cf 27 0a 38 09 90 8d 9b df 8c 8b 9e ac
 B37  =  8c b4 4d f4 21 4f 61 11 8a 8d df 92 9e 8d 94 f1
 B38  =  d8 fd 08 cf 6f 1d 7d 1f 9b 86 d3 df 8c 8b 9e ae
 B39  =  8c b4 67 bb 4f 6f 18 7e ff ff ff ff ff ff ff db
 B40  =  b9 f6 f7 03 a4 99 b3 82 7b 7f 7e 7d 0f 14 52 f7


 Total Length: 656 bytes: [Header-Info, Assoc-Data, Payload, MAC] <--> [16 + 32 + 592 + 16]
 ------------------------------------------------------------------------------------------------------------------------------------------------------------------
```

The output was shortened due to the length of the data structure sent. In this case, there were a total of 40 blocks sent. The consumer received the following:

```
------------------------------------------------------------ Starting CBC-MAC w/ Counter Implementation ------------------------------------------------------------

                                                                Received CCM Message
B0   =  7f 65 92 44 b0 90 e7 81 00 00 00 00 00 00 02 43
B1   =  00 17 61 73 73 6f 63 69 61 74 65 64 20 64 61 74
B2   =  61 20 73 74 75 66 66 21 00 00 00 00 00 00 00 00
B3   =  ac f2 08 9b 09 06 6c 10 9a 8c 8c b8 8d 9e 92 1d
B4   =  7c 38 4d eb 2e 0c 7d 0c df ab 9a 8c 8b df 96 8d
B5   =  d8 fb 4d d6 3a 03 6c 17 8c 8b 9e 98 9a df 9e 98
B6   =  8a f5 0f d2 2c 4f 7b 1f 8f 9e 9c 96 8b 86 df 88
B7   =  9d e9 19 9b 3b 07 79 0a df 8f 8d 90 98 8d 9a 88
B8   =  8b f3 1b de 23 16 38 19 9a 8b 8c df 92 90 8d 9f
B9   =  d8 fe 04 dd 29 06 7b 0b 93 8b df 9e 8c df 96 8d
B10  =  d8 f9 02 d5 3b 06 76 0b 9a 8c d1 df ab 97 9a d8
B11  =  ca aa 4d d6 2a 1b 7d 0c df 8f 9e 9c 9a 8d df 83
B12  =  9d e9 19 9b 38 06 74 12 df 9d 9a 98 96 91 df 9f
B13  =  96 ba 5e 8b 6f 1c 7d 1d 90 91 9b 8c d1 df b3 9c
B14  =  96 ff 4d ce 3f 4f 79 0a df 8b 97 9a df 8c 8b 95
B15  =  8a ee 43 9b 1b 07 7d 5e 8d 8a 91 91 96 91 98 d3
B16  =  8b ea 08 de 2b 4f 6b 0a 9e 8d 8b 8c df 8c 93 9d
B17  =  8f f6 14 97 6f 0d 6d 0a df 98 9a 8b 8c df 99 90
B18  =  8b ee 08 c9 6f 0a 79 1d 97 df 92 96 91 8a 8b 95
B19  =  d8 fb 0b cf 2a 1d 38 07 90 8a df 97 9a 9e 8d cf
```

...




Again, the CCM MAC is verified, and the message is returned. This time with a much larger message. The final test will involve a significantly larger message including a description of the first year engineering program found on the University of Kentucky Computer Science department website. This message was repeated four times to exaggerate a lengthy payload. Here is the end of the producer output:

```
B228 =  d8 8c 1b 6d b8 d5 e6 3a df 9e 91 9b df 9a 87 6e
B229 =  97 99 0f 7c a8 99 e6 2c df 9a 91 98 96 91 9a 78
B230 =  8a 83 14 69 2f 39 0b 30 df 98 8d 9a 9e 8b 9a 6f
B231 =  8c ca 19 66 ac d5 fe 26 91 98 9a 8c d3 df 8b 73
B232 =  9d ca 3c 57 88 99 e2 31 90 98 8d 9e 92 df 98 7f
B233 =  8c 99 5a 77 a2 cc b2 2a 91 8b 90 df 8b 97 9a 39
B234 =  9f 8b 17 6b ed df e0 2c 92 df 9b 9e 86 df 90 76
B235 =  9d c4 2e 66 a8 99 e1 2e 9e 8d 8b 9a 8c 8b d3 37
B236 =  95 85 09 7a ed cd f3 2f 9a 91 8b 9a 9b df 9a 78
B237 =  9f 83 14 6b a8 cb e1 63 9e 8d 90 8a 91 9b df 61
B238 =  90 8f 5a 79 a2 cb fe 27 df 9e 8d 9a df 9b 9a 62
B239 =  97 9e 13 60 aa 99 e6 2b 9a 92 8c 9a 93 89 9a 60
B240 =  d8 9e 15 2e b9 d8 f1 28 93 96 91 98 df 96 92 7f
B241 =  9d 84 09 6b ed de fe 2c 9d 9e 93 df 9c 97 9e 7d
B242 =  94 8f 14 69 a8 ca bc 63 be 8c df 9e df b9 96 62
B243 =  8b 9e 57 57 a8 d8 e0 63 ba 91 98 96 91 9a 9a 7d
B244 =  91 84 1d 2e e5 ff cb 06 d6 df 8c 8b 8a 9b 9a 60
B245 =  8c c6 5a 77 a2 cc b2 24 9a 8b df 8b 90 df 95 62
B246 =  91 84 5a 7a a5 dc ff 62 df b6 91 df cd cf cf 34
B247 =  d4 ca 0e 66 a8 99 dc 22 8b 96 90 91 9e 93 df 4a
B248 =  9b 8b 1e 6b a0 c0 b2 2c 99 df ba 91 98 96 91 6f
B249 =  9d 98 13 60 aa 99 fb 27 9a 91 8b 96 99 96 9a 6d
B250 =  d8 db 4e 2e 2f 39 0e 04 8d 9e 91 9b df bc 97 69
B251 =  94 86 1f 60 aa dc e1 63 99 90 8d df ba 91 98 6e
B252 =  96 8f 1f 7c a4 d7 f5 63 96 91 df 8b 97 9a df 34
B253 =  c9 99 0e 2e 8e dc fc 37 8a 8d 86 1d 7f 62 1d 85
B254 =  6c 85 0a 7e a2 cb e6 36 91 96 8b 96 9a 8c df 70
B255 =  97 ca 1d 7c a8 d8 e6 2f 86 df 96 91 9c 8d 9a 62
B256 =  8b 8f 5a 66 b8 d4 f3 2d 96 8b 86 1d 7f 66 8c 22
B257 =  8b 9f 09 7a ac d0 fc 22 9d 96 93 96 8b 86 d3 21
B258 =  90 8f 1b 62 b9 d1 be 63 8c 9a 9c 8a 8d 96 8b 0a
B259 =  f8 ea 7a 0e cd b9 92 43 ff ff ff ff ff ff fe ff
B260 =  75 94 e0 5f df da 3d 98 ab 46 18 af f1 9d 5f 4f


Total Length: 4176 bytes: [Header-Info, Assoc-Data, Payload, MAC] <--> [16 + 32 + 4112 + 16]
--------------------------------------------------------------------------------------------
```

Here is the output of the consumer after receiving and deconstructing the message:

```
E256        =  91 4a 49 af 60 f2 03 9c 45 b5 e8 4c 08 72 f8 9a
XOR257      =  e2 3f 3a db 01 9b 6d fd 27 dc 84 25 7c 0b d4 ba
E257        =  1d c0 c5 24 fe 64 92 02 d8 23 7b da 83 f4 2b 45
XOR258      =  75 a5 a4 48 8a 0c be 22 ab 46 18 af f1 9d 5f 4f
E258        =  8a 5a 5b b7 75 f3 41 dd 54 b9 e7 50 0e 62 a0 b0
XOR259      =  8a 5a 5b b7 75 f3 41 dd 54 b9 e7 50 0e 62 a0 b0
E259        =  75 a5 a4 48 8a 0c be 22 ab 46 18 af f1 9d 5f 4f

MAC       =  75 a5 a4 48 8a 0c be 22 ab 46 18 af f1 9d 5f 4f
RECV MAC  =  75 a5 a4 48 8a 0c be 22 ab 46 18 af f1 9d 5f 4f
MAC Calculation Match: authenticity and integrity verified! :D

ADATA: associated data stuff!

PAYLOAD: The smartest, most talented engineers around the world are devoting themselves to tackling immense global challenges. As a First-Year Engineering (FYE)
 student, you get to join them! In 2008, the National Academy of Engineering identified 14 “Grand Challenges for Engineering in the 21st Century”–opportunities
to greatly increase humanity’s sustainability, health, security and joy of living. Themes include making solar energy economical, enhancing virtual reality, rev
erse-engineering the brain, securing cyberspace, providing access to clean water and more. These ambitious goals demand engineers roll up their sleeves and get
to work, which is why we put them front and center during your first year as an engineering student. We have designed the FYE program to inspire you. We want yo
u to discover your passion. We want you to explore where you might make your unique contribution. We want you to get your hands dirty and make stuff that might,
 one day, lead to a breakthrough. Why wait until you’re taking upper-level classes to figure out what interests you? Through real engineering classes taught by
top faculty and exposure to engineering’s greatest challenges, the FYE program gets you into the game from day one.The smartest, most talented engineers around
the world are devoting themselves to tackling immense global challenges. As a First-Year Engineering (FYE) student, you get to join them! In 2008, the National
Academy of Engineering identified 14 “Grand Challenges for Engineering in the 21st Century”–opportunities to greatly increase humanity’s sustainability, health,
 security and joy of living. Themes include making solar energy economical, enhancing virtual reality, reverse-engineering the brain, securing cyberspace, provi
ding access to clean water and more. These ambitious goals demand engineers roll up their sleeves and get to work, which is why we put them front and center dur
ing your first year as an engineering student. We have designed the FYE program to inspire you. We want you to discover your passion. We want you to explore whe
re you might make your unique contribution. We want you to get your hands dirty and make stuff that might, one day, lead to a breakthrough. Why wait until you’r
e taking upper-level classes to figure out what interests you? Through real engineering classes taught by top faculty and exposure to engineering’s greatest cha
llenges, the FYE program gets you into the game from day one.The smartest, most talented engineers around the world are devoting themselves to tackling immense
global challenges. As a First-Year Engineering (FYE) student, you get to join them! In 2008, the National Academy of Engineering identified 14 “Grand Challenges
 for Engineering in the 21st Century”–opportunities to greatly increase humanity’s sustainability, health, security and joy of living. Themes include making sol
ar energy economical, enhancing virtual reality, reverse-engineering the brain, securing cyberspace, providing access to clean water and more. These ambitious g
oals demand engineers roll up their sleeves and get to work, which is why we put them front and center during your first year as an engineering student. We have
 designed the FYE program to inspire you. We want you to discover your passion. We want you to explore where you might make your unique contribution. We want yo
u to get your hands dirty and make stuff that might, one day, lead to a breakthrough. Why wait until you’re taking upper-level classes to figure out what intere
sts you? Through real engineering classes taught by top faculty and exposure to engineering’s greatest challenges, the FYE program gets you into the game from d
ay one.The smartest, most talented engineers around the world are devoting themselves to tackling immense global challenges. As a First-Year Engineering (FYE) s
tudent, you get to join them! In 2008, the National Academy of Engineering identified 14 “Grand Challenges for Engineering in the 21st Century”–opportunities to
 greatly increase humanity’s sustainability, health, securit

--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
```

## IV. Conclusion

In conclusion, this project successfully implemented the Counter Mode with CBC-MAC (CCM) algorithm in C, providing a robust and efficient solution for securing sensitive data. The developed CCM implementation leverages the strengths of both the counter mode and the Cipher Block Chaining Message Authentication Code (CBC-MAC) to achieve a high level of confidentiality and integrity for transmitted data. Although this project only includes a basic inverting cipher to test, the use of a more robust block cipher such as AES would be much more practical. With that update, this implementation can be readily integrated into a variety of applications, such as secure communication protocols, IoT devices, and wireless networks, where the protection of data is paramount.

Throughout the development of this project, careful attention was given to the crucial aspects of the CCM algorithm, such as nonce generation and key management. Ensuring that the unique nonce is used for each message and that keys are properly managed is vital for maintaining the security of the CCM system. The successful implementation of the CCM algorithm in C demonstrates the power and versatility of this cryptographic technique, providing a foundation for further exploration, optimization, and adaptation to various use cases and environments. In this project, I have learned a lot about socket programming and the delicacies of implementing such a specific algorithm in C. Overall, a lot of learning opportunities were presented, and I am glad I got to experience each one of them during the development of this project.

## V. References

*CCM v4.0 Implementation Guidelines*. CSA. (n.d.). Retrieved April 28, 2023, from https://cloudsecurityalliance.org/artifacts/ccm-v4-0-implementation-guidelines/

*Recommendation for block cipher modes of operation: The CCM mode ... - NIST*. (n.d.). Retrieved April 28, 2023, from https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38c.pdf

Whiting, D., Housley, R., & Ferguson, N. (1970, September 1). *Counter with CBC-MAC (CCM)*. RFC Editor. Retrieved April 28, 2023, from https://www.rfc-editor.org/rfc/rfc3610