

A Novel Approach to Cryptography Implementation using ChatGPT

Hyeokdong Kwon¹[0000-0002-9173-512X],
Minjoo Sim¹[0000-0001-5242-214X],
Gyeongju Song¹[0000-0002-4337-1843],
Minwoo Lee²[0000-0002-2356-3055], and
Hwajeong Seo²[0000-0003-0069-9061]

¹Department of Information Computer Engineering,
Hansung University, Seoul (02876), South Korea,

²Department of Convergence Security,

Hansung University, Seoul (02876), South Korea,

{korlethean, minjoos9797, thdrudwn98, minunejip, hwajeong84}@gmail.com

Abstract. ChatGPT, which emerged at the end of 2022, has gained significant attention as a highly advanced conversational artificial intelligence service. Developed by OpenAI, ChatGPT is a natural language processing model. There are instances where individuals might want to attempt programming using ChatGPT. In this paper, we utilized the ChatGPT to implement a cryptographic algorithms. Despite numerous trial and error efforts, it was possible to implement cryptography through ChatGPT. This implies that even without extensive coding skill or programming knowledge, one can implement cryptography through ChatGPT if they understand the cryptographic structure. However, the ability to analyze the source code is essential, as it is necessary to identify incorrect parts within the implemented code.

Keywords: Block Cipher · ChatGPT · Cryptography Implementation · Software Implementation.

1 Introduction

ChatGPT, developed by OpenAI, is a natural language processing model that generates sentences at a level similar to human language. Introduced in November 2022, ChatGPT demonstrated powerful processing capabilities and created a significant sensation. Many users have utilized ChatGPT in various fields such as academic writing, search engines, and coding [1]. Although ChatGPT is primarily a language model, it is also capable of programming. In particular, if requirements are specified by users, a program can be written according to the specifications. However, implementing highly complex programs using only ChatGPT remains challenging. Cryptographic algorithms are based on intricate mathematical operations, and their internal workings are also complex. In this paper, we introduce a methodology to implement complex cryptographic algorithms using ChatGPT.

1.1 Contributions

- **First implementation of cryptography algorithms using ChatGPT.** There are many examples of programming using ChatGPT. However, it has not been applied to specialized fields, such as cryptographic algorithms. In this paper, the cryptographic algorithm was completely implemented using only the source code written by ChatGPT without human code writing. Although the source code was not complete, cryptography engineers checked and requested modifications, but most of the code was implemented successfully by ChatGPT.
- **Presenting a methodology for implementing cryptographic algorithms through artificial intelligence.** We introduce a method for implementing cryptography through AI. This approach enables AI to directly generate the source code without any manual modification during algorithm implementation. This presents a novel avenue for implementing cryptographic algorithms by analyzing test vectors, even for those who may not possess the skills to write source code themselves.

2 Related Works

2.1 ChatGPT

ChatGPT is a state-of-the-art Large Language Model (LLM) announced by OpenAI at 2022. While previous classes of AI models were primarily deep learning (DL) models designed to learn and recognize patterns in data, LLMs are a new type of AI algorithm trained to predict the likelihood of a given sequence of words based on the context of the preceding word. Thus, if an LLM is trained on a sufficiently large amount of text data, it can generate new word sequences that have never been observed before by the model, but represent plausible sequences based on natural human language [2].

ChatGPT is based on GPT-3 (Generative Pre-trained Transformer-3), a deep learning-based language model. It is an extensive neural network comprising 175 billion parameters. The use of the GPT-3 model is increasing in fields such as language translation and question answering, and since it corresponds to unsupervised learning, content is created based on a vast amount of big data learned from the Internet. GPT-3 not only performs tasks that machine learning can do but also interacts and converses in a manner similar to humans. In particular, ChatGPT is an interactive artificial intelligence model that learns through user feedback and verifies the generated output, enabling the model to self-adjust its bias by incorporating a reinforcement learning element to influence the learning results. After being released to the public in November 2022, its user base has rapidly grown, and various use cases across different fields are being presented, leading to a new paradigm in artificial intelligence [3].

Designed to interact with users in a human-like manner and learn from feedback, ChatGPT can provide answers to user prompts or offer relevant information. The responses or information provided are generated results that, while

not identical, exhibit similarities, using unstructured data that exists on the web learned by 2021. The currently applied GPT-3.5 model does not automatically provide sources and has limitations, such as the difficulty of real-time web data search.

2.2 Target Cryptography Algorithm

AES (Advanced Encryption Standard) Block Cipher AES is a project that solicits block cipher algorithms globally to replace DES, which has significant security issues at the National Institute of Standards and Technology (NIST). Rijndael, developed by Joan Daemen and Vincent Rijmen, was chosen as the AES encryption algorithm in October 2000. Since then, AES has become the most widely used block cipher algorithm worldwide. The input plaintext length for the AES algorithm is fixed at 128 bits, while the round key length used for encryption can be chosen from 128-bit, 192-bit, or 256-bit. The number of rounds varies depending on the key length, with 10, 12, and 14 rounds being applied [4]. AES represents the intermediate result values generated during the encryption and decryption processes as a 4×4 two-dimensional byte matrix. The 4×4 two-dimensional matrix is arranged in a column-first order rather than row-first. There are primarily four independent functions within the round function of AES, with each round function as follows. There are largely four independent functions within the round function of AES, and each round function is as follows.

- SubBytes (SB): Non-linear byte substitution function using an 8-bit S-box.
- ShiftRows (SR): A function that performs left rotation by row. The first row remains unchanged, the second row is rotated by 1 byte, the third row is rotated by 2 bytes, and the fourth row is rotated by 3 bytes.
- MixColumns (MC): A 32-bit linear transformation function that performs column-by-column mixing.
- AddRoundKey (ARK): A function that executes addition with a round key.

During the encryption process of AES, before applying the round function, plaintext and round key addition are initially applied as a whitening step, followed by the execution of the four round functions. In the final round, all round functions are performed except for the MixColumn function. In the decryption process, there are inverse transformation functions for the four individual functions carried out during the encryption process. These inverse transformation functions and the round key used in the encryption process are applied in reverse order.

CHAM Block Cipher The CHAM block cipher is a lightweight block cipher introduced at ICISC'17 [5]. It primarily targets low-end processors such as 8-bit AVR, 16-bit MSP320, and 32-bit ARM processors. Moreover, the hardware size can be implemented at a minimum when compared to existing block encryption

algorithms and bit-serial implementations. CHAM is an ARX (Addition, Rotation, eXclusive-or) based algorithm, which contributes to its lightweight feature. In particular, CHAM successfully reduces key storage space by using a stateless round key technique that does not store the key state. CHAM offers three encryption schemes in total. The entire parameters for each CHAM scheme are shown in Table 1 [6].

Table 1: Parameters of block cipher CHAM. n : block size (bit), k : key size (bit), w : word size (byte), r : number of round for revised CHAM, r_{legacy} : number of round for original CHAM.

Scheme	n	k	w	r	r_{legacy}
CHAM-64/128	64	128	16	88	80
CHAM-128/128	128	128	32	112	80
CHAM-128/256	128	256	32	120	96

The encryption algorithm of the CHAM block cipher differs in the offset used in odd and even rounds, and the detailed encryption process is depicted in Figure 1. The plaintext is divided into four parts and transitions from X_i to ω_i . X_i becomes an XOR operation with the counter. Next, Y_i undergoes a left rotation. After XOR with the round key, X_i is added, and a left rotation operation is performed eight times. The subsequent values are moved word by word and stored [7]. CHAM has the advantage of being resistant to error injection attacks that analyze secret keys based on incorrect output after injecting intentional errors into the cryptographic device’s operation [8]. In ICISC’19, Revisited CHAM, which has the same structure and specifications as existing CHAM but with an increased number of rounds, was introduced [9]. The previous CHAM rounds consisted of 80 rounds for CHAM-64/128, 80 rounds for CHAM-128/128, and 96 rounds for CHAM-128/256, but Revised CHAM used 88, 112, and 120 rounds for CHAM-64/128, CHAM-128/256, and CHAM-128/256, respectively.

3 Implementation of Cryptography Algorithm using ChatGPT

In this section, we show the implementation of the cryptographic algorithm using ChatGPT. We went through four steps to implement a cryptographic algorithm with ChatGPT. The first is a check on the algorithm. Checks whether ChatGPT knows the cryptographic algorithm. The second step is teach algorithm to ChatGPT if it doesn’t know about target cipher. The third step is to request coding in C language. The last step is to run the provided source code to check if the algorithm is properly implemented. Check the test vector, and if this value is normal, it is judged that the implementation is successful. If not, ChatGPT is requested again to find and correct the wrong part. This process is repeated

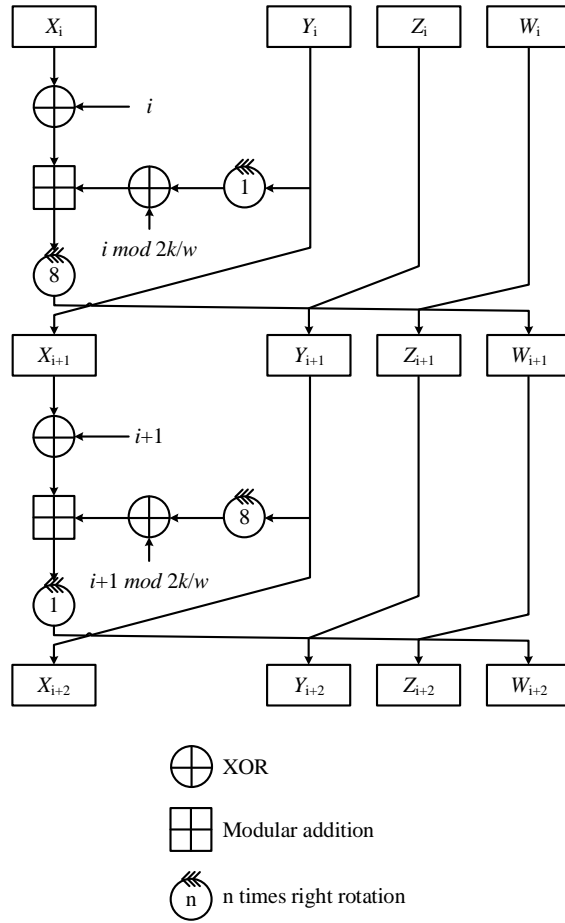


Fig. 1: The round function structure of CHAM block cipher.

to complete the encryption algorithm. A detailed explanation is provided in Section 3.1. We have classified two types of ciphers to be implemented. One is a well-known algorithm. This did not explain the algorithm separately to ChatGPT, but only confirmed that it knew the algorithm and proceeded with the implementation immediately. The other is an obscure algorithm. In this case, since ChatGPT has an unknown structure, the algorithm was separately explained and the implementation proceeded. We used ChatGPT 4.0 to obtain accurate implementation results.

3.1 Procedure of Cryptography Implementation using ChatGPT

To implement a cryptographic algorithm through ChatGPT, a four-step approach was designed. The first step is the algorithm verification stage. It is important to note that ChatGPT's knowledge is limited to data up to 2021 [10]. Therefore, it is crucial to ensure that ChatGPT is familiar with the desired algorithm for implementation. If ChatGPT knows the target algorithm, step 2 can be skipped.

The second step involves teaching the algorithm. If ChatGPT is unfamiliar with the algorithm to be implemented, it must be taught. Considering that ChatGPT is a conversational language model, the algorithm is taught through explanations using words and occasional mathematical expressions.

Step 3, focuses on algorithm implementation. ChatGPT is asked to implement the known or learned algorithms in a specific language. In this paper, C language was chosen for implementation.

Step 4, entails algorithm integrity verification and modification. It is necessary to determine if the algorithm implemented by ChatGPT functions properly. Generally, cryptographic algorithms reveal their structure and provide test vectors. Consequently, if the results obtained from running the source code supplied by ChatGPT match the test vector, the algorithm can be deemed successfully implemented. If the results are incorrect, the source code needs to be adjusted to rectify any errors in the algorithm's implementation. It was assumed that individuals attempting to implement the algorithm through ChatGPT may not have extensive programming knowledge. As a result, instead of directly modifying the code, users should identify the incorrect portions and request ChatGPT to make the necessary corrections. The overall process is illustrated in Figure 2.

3.2 Implementation of Well-known Algorithm: AES

AES is a well-known algorithm due to its establishment as a standard in 2001 and its long-standing usage. As a result, when asking ChatGPT about AES, it is evident that it understands the algorithm correctly, as demonstrated in Listing 1. Moreover, ChatGPT recommends using open-source libraries such as OpenSSL for implementing AES. This suggests that utilizing an already-implemented library is more common than directly implementing a cryptographic algorithm.

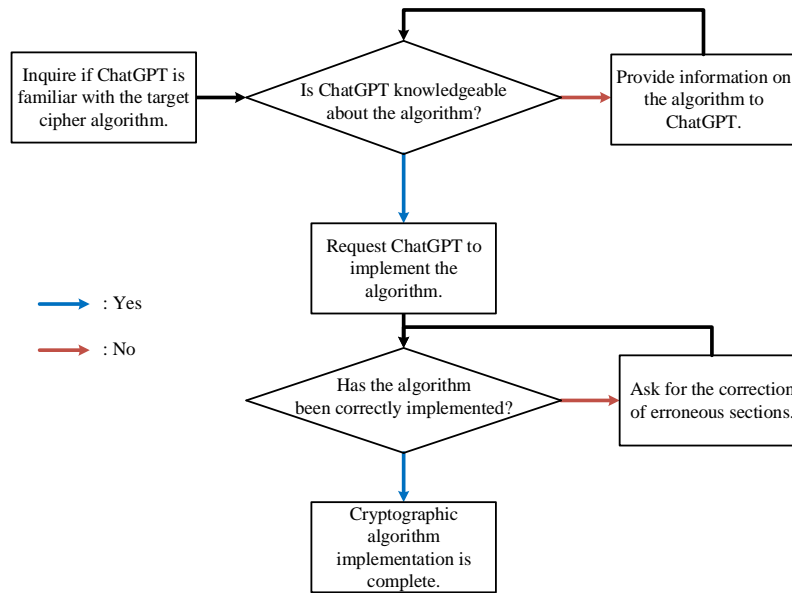


Fig. 2: The Cryptographic Algorithm Implementation Approach Using ChatGPT.

HD I want implement AES algorithm written in C language. Before start this one, do you know about structures of AES algorithm?

🌀 Yes, I'm familiar with the Advanced Encryption Standard (AES) algorithm. AES is a symmetric key encryption algorithm, which means it uses the same key for both encryption and decryption. It operates on fixed block sizes of 128 bits and supports key sizes of 128, 192, or 256 bits. The algorithm consists of several stages, including key expansion, initial round, main rounds, and a final round.

Here's a high-level overview of the main stages in AES encryption:

1. Key expansion: The original key is expanded into an array of key schedule words, which will be used in subsequent rounds of the algorithm.
2. Initial round:
 - AddRoundKey: The input block (plaintext) is XORed with the initial round key derived from the key schedule.
3. Main rounds (9, 11, or 13 rounds, depending on the key size):
 - SubBytes: A non-linear substitution step where each byte of the block is replaced with a corresponding byte from a pre-determined substitution table (S-box).
 - ShiftRows: A transposition step where the last three rows of the state are shifted cyclically a certain number of steps.
 - MixColumns: A linear mixing operation that operates on the columns of the state, combining the four bytes in each column using a pre-determined matrix.
 - AddRoundKey: The result of the previous three steps is XORed with a round key derived from the key schedule.
4. Final round (no MixColumns):
 - SubBytes
 - ShiftRows
 - AddRoundKey

The decryption process follows a similar structure but in reverse order, using inverse operations for SubBytes, ShiftRows, and MixColumns.


Keep in mind that implementing cryptographic algorithms from scratch can be challenging and prone to errors. It's often recommended to use well-established libraries, such as OpenSSL or mbedtls, for production use cases.


Listing 1: Answer when asking ChatGPT about AES.

This indicates that ChatGPT is familiar with the AES algorithm. Consequently, the implementation was carried out for each module without explaining the AES algorithm. ChatGPT can output 4096 tokens at once. However, this can cause the output to stop when values such as the S-Box table are output. To ensure stable code output, large table values like the S-Box table were requested to be omitted, and other parts were printed. The modules for which output was requested are as follows.

- Key Expansion.
- Subbytes
- Inverse subbytes
- Shift rows.
- Inverse shift rows.
- Mix columns.
- Inverse Mix columns.
- Add round key
- AES encryption.
- AES decryption.

Primarily, the implementation was requested for subbytes, shift rows, mix columns, and add round key steps, which are essential components of AES operation. For instance, key expansion is implemented in the form shown in Listing 2.

 Then implement AES key expansion in C language. If you are going to use the Rcon table as well, define this as a global variable.

 Here's an implementation of the AES key expansion in C, along with the Rcon table as a global variable:

```

1 #include <stdint.h>
2
3 // Rcon table for key expansion
4 static const uint8_t rcon[11] = {
5     0x00, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0
6     x1B, 0x36
7 };

```



```


8 // S-box (SubBytes) - Please fill in the values
9 static const uint8_t s_box[256];
10
11 // Helper functions
12 static uint32_t RotWord(uint32_t word) {
13     return (word << 8) | (word >> 24);
14 }
15
16 static uint32_t SubWord(uint32_t word) {
17     uint32_t result = 0;
18     result |= s_box[(word >> 24) & 0xFF] << 24;
19     result |= s_box[(word >> 16) & 0xFF] << 16;
20     result |= s_box[(word >> 8) & 0xFF] << 8;
21     result |= s_box[word & 0xFF];
22     return result;
23 }
24
25 // Key expansion
26 void AES_KeyExpansion(const uint8_t *key, uint32_t *
27     key_schedule, int key_size) {
28     int num_words = key_size / 32; // Number of words in the
29     // initial key
30     int key_schedule_size = (key_size == 128) ? 44 : (
31     key_size == 192) ? 52 : 60;
32
33     // Copy the initial key to the key schedule
34     for (int i = 0; i < num_words; i++) {
35         key_schedule[i] = (key[4 * i] << 24) | (key[4 * i +
36     1] << 16) | (key[4 * i + 2] << 8) | key[4 * i + 3];
37     }
38
39     // Generate the remaining words in the key schedule
40     for (int i = num_words; i < key_schedule_size; i++) {
41         uint32_t temp = key_schedule[i - 1];
42
43         if (i % num_words == 0) {
44             temp = SubWord(RotWord(temp)) ^ (rcon[i /
45     num_words] << 24);
46         } else if (key_size == 256 && i % num_words == 4) {
47             temp = SubWord(temp);
48         }
49
50         key_schedule[i] = key_schedule[i - num_words] ^ temp;
51     }
52 }


```

Listing 2: Key expansion source code generated by ChatGPT.

Once the implementation of each module is completed, AES encryption process is requested from ChatGPT. Consequently, ChatGPT directly implements

the encryption process of AES. Although no specific modules were mentioned to be called for AES encryption and in what order, ChatGPT successfully handled it. When implementing encryption algorithms with modularization, it is crucial to understand the purpose of each module and the calling sequence for encryption and decryption. ChatGPT demonstrated a perfect understanding of AES. Listing 3 displays the AES encryption code written by ChatGPT.

 Now we implemented Key expansion, SubBytes(InvSubBytes), ShiftRows(InvShiftRows), MixColumns(InvMixColumns), and AddRoundKey in C language. Use these to complete the Encrypt rough you wrote at the beginning.

 Here's the complete AES encryption implementation using the previously defined functions:

```

1 #include <stdint.h>
2
3 // S-box (SubBytes) - Please fill in the values
4 static const uint8_t s_box[256];
5
6 // Inverse S-box (InvSubBytes) - Please fill in the values
7 static const uint8_t inv_s_box[256];
8
9 // Rcon table for key expansion
10 static const uint8_t rcon[11] = {
11     0x00, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0
12     x1B, 0x36
13 };
14 // Previously defined helper functions and transformations go
15     here
16 void AES_Encrypt(const uint8_t *plaintext, const uint8_t *key
17     , uint8_t *ciphertext, int key_size) {
18     uint8_t state[4][4];
19     uint32_t key_schedule[60]; // Maximum size (for 256-bit
20     key)
21
22     // Initialize state array with input plaintext
23     for (int row = 0; row < 4; row++) {
24         for (int col = 0; col < 4; col++) {
25             state[row][col] = plaintext[row + 4 * col];
26         }
27     }
28
29     // Perform key expansion
30     AES_KeyExpansion(key, key_schedule, key_size);

```

```

29
30 // Initial AddRoundKey
31 AddRoundKey(state, key_schedule);
32
33 // Main rounds
34 int num_rounds = (key_size == 128) ? 10 : (key_size ==
192) ? 12 : 14;
35 for (int round = 1; round < num_rounds; round++) {
36     SubBytes(state);
37     ShiftRows(state);
38     MixColumns(state);
39     AddRoundKey(state, key_schedule + 4 * round);
40 }
41
42 // Final round (without MixColumns)
43 SubBytes(state);
44 ShiftRows(state);
45 AddRoundKey(state, key_schedule + 4 * num_rounds);
46
47 // Copy state array to output ciphertext
48 for (int row = 0; row < 4; row++) {
49     for (int col = 0; col < 4; col++) {
50         ciphertext[row + 4 * col] = state[row][col];
51     }
52 }
53 }

```

Listing 3: AES encryption source code generated by ChatGPT.

In this manner, AES encryption and decryption functions were created. However, one might be uncertain about how to utilize these functions for encryption. Assuming that no one directly writes the source code, ChatGPT was requested to provide the code that proceeds with encryption in the main function. During this process, the values of some variables were specified to check the test vector. The result is visible in Listing 4.

HD Now create the main function. Write the main function to call the AES-128 Encryption and Decryption functions once. The key value is '000102030405060708090a0b0c0d0e0f' and the plaintext value is '00112233445566778899aabbccddeeff'. Declare the name of the key variable as 'key_128', the name of the plaintext variable as 'plaintext_128', the name of the ciphertext variable as 'ciphertext_128', and the name of the decryption variable as 'decrypted_128'.



Here's a main function that demonstrates the use of the AES-128 Encryption and Decryption functions with the provided key and plaintext:

```

1 #include <stdio.h>
2 #include <stdint.h>
3
4 // S-box (SubBytes), Inverse S-box (InvSubBytes), Rcon table,
   // and previously defined functions go here
5
6 int main() {
7     uint8_t key_128[16] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05
8     , 0x06, 0x07,
9     , 0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D
10    , 0x0E, 0x0F};
11
12    uint8_t plaintext_128[16] = {0x00, 0x11, 0x22, 0x33, 0x44
13    , 0x55, 0x66, 0x77,
14    , 0x88, 0x99, 0xAA, 0xBB, 0xCC
15    , 0xDD, 0xEE, 0xFF};
16
17    uint8_t ciphertext_128[16];
18    uint8_t decrypted_128[16];
19
20    AES_Encrypt(plaintext_128, key_128, ciphertext_128, 128);
21    AES_Decrypt(ciphertext_128, key_128, decrypted_128, 128);
22
23    printf("Plaintext:\n");
24    for (int i = 0; i < 16; i++) {
25        printf("%02x ", plaintext_128[i]);
26    }
27    printf("\n");
28
29    printf("Ciphertext:\n");
30    for (int i = 0; i < 16; i++) {
31        printf("%02x ", ciphertext_128[i]);
32    }
33    printf("\n");
34
35    printf("Decrypted:\n");
36    for (int i = 0; i < 16; i++) {
37        printf("%02x ", decrypted_128[i]);
38    }
39    printf("\n");
40
41    return 0;
42 }

```

This 'main' function demonstrates the encryption and decryption process using the provided 'key_128' and 'plaintext_128'. The encrypted result is stored in 'ciphertext_128', and the decrypted result is stored in 'decrypted_128'. The program prints the plaintext, ciphertext, and decrypted values in hexadecimal format. Please ensure that you fill in the 's_box' and 'inv_s_box' arrays and include the previously defined functions before running this code.

Listing 4: A main function source code that runs AES-128 generated by ChatGPT.

Based on this, the creation of source code for operating AES-192 and AES-256 was requested. The corresponding source code was compiled with the minGW compiler, and the output is shown in Figure 3. The result matches the AES test vectors, which are excerpted from Appendix C of NIST FIPS-197 [11]. It was determined that the AES implementation by ChatGPT aligns perfectly with the test vector, indicating a successful implementation. This makes it easy for ChatGPT to implement well-known ciphers like AES. The overall implementation process and source code can be found in Appendix 5.1.

```
<terminated> (exit value: 0) ChatGPT_AES [C/C++ Application]
Plaintext:
00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff
Ciphertext:
69 c4 e0 d8 6a 7b 04 30 d8 cd b7 80 70 b4 c5 5a
Decrypted:
00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff
Plaintext:
00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff
Ciphertext:
dd a9 7c a4 86 4c df e0 6e af 70 a0 ec 0d 71 91
Decrypted:
00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff
Plaintext:
00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff
Ciphertext:
8e a2 b7 ca 51 67 45 bf ea fc 49 90 4b 49 60 89
Decrypted:
00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff
```

Fig. 3: The running result of the AES source code generated by ChatGPT.

3.3 Implementation of Unknown Algorithm: CHAM

The next target is the CHAM block cipher, a domestic standard cryptography in Korea that is not well-known outside the country. ChatGPT was also unfamiliar with the CHAM block cipher. When using ChatGPT 3.5 and inquiring about CHAM, it provided incorrect information as shown in Listing 5. Several aspects are wrong, including CHAM being a block cipher, not HMAC, and lacking an authentication function. Additionally, CHAM is just the name of a block cipher, and there is no such thing as a CHAM mode other than a block cipher mode of operation.



Do you know about CHAM algorithm?



Yes, I'm familiar with the CHAM algorithm. The CHAM (Cipher-based Hashing and Message Authentication) algorithm is a cryptographic primitive that provides both hashing and message authentication code (MAC) functionality. It was designed by Joo Yeon Cho, Jong Hwan Park, and Seokhie Hong in 2011 as a more efficient alternative to the HMAC (Hash-based Message Authentication Code) algorithm.

The CHAM algorithm uses a block cipher to generate a hash or MAC value for a given message. The block cipher operates in a special mode called the CHAM mode, which ensures that the output is secure and unique for each message. The CHAM algorithm can operate with different block sizes and key sizes, depending on the desired level of security.

Overall, the CHAM algorithm is a promising cryptographic primitive that offers both hashing and MAC functionality with high efficiency and security. However, it is not yet widely adopted in practice and is still undergoing analysis and testing by the cryptographic community.

Listing 5: ChatGPT answers questions about the CHAM block cipher with fake information.

When posing the same question to ChatGPT 4.0, it responded with no knowledge on the subject, as seen in Listing 6. Consequently, it was believed that ChatGPT lacked information about CHAM, the Korean standard block cipher, and so information about the CHAM block cipher was provided to ChatGPT.



Do you know about CHAM algorithm?



As an AI language model, I'm familiar with a wide range of algorithms and techniques. However, as of my knowledge cutoff date in September 2021, I am not aware of a specific algorithm called CHAM. It's possible that it has been developed or popularized after that date or that it's an abbreviation or acronym for a more widely known algorithm or method.

If you could provide more information or context about the CHAM algorithm, I'd be happy to help answer any questions or discuss the topic further.

Listing 6: ChatGPT answers questions about the CHAM block cipher with fake information.

ChatGPT was taught about the CHAM block cipher in the following order:

- **Scheme.** The CHAM block cipher has three types of schemes: CHAM-64/128, CHAM-128/128, and CHAM-128/256. The first number indicates the block size, and the second represents the key size (in bits).
- **Parameters.** n: block size (bits), k: secret key size (bits), w: word size (bits), r: number of rounds. CHAM has both legacy and revised versions, with the only difference between the two being r. Information about the revised version was provided.
- **Round functions.** The CHAM has different round function structures for odd and even rounds. The difference lies in the number of rotation operations. Thus, the odd-numbered round function was explained first, followed by the even-numbered round function. When informing about the even-numbered round function, the entire structure of the round function was not detailed; only the parts differing from the odd-numbered round function were highlighted.
- **Key scheduling.** The key scheduling used in CHAM is the same across all versions, so it was taught all at once.

There was no need to provide information about ARX and Feistel structures, as ChatGPT already knew about these concepts. After the learning process, ChatGPT was asked about its knowledge on CHAM, as shown in Listing 7, to verify if it remembered the precise content. The explanation of the even-numbered round function was skipped because it was too lengthy to include in a full answer and closely resembled the odd-numbered round function.



What I have told you so far is all about the CHAM block cipher. Based on what I told you, explain the CHAM block cipher.



The CHAM block cipher is a lightweight symmetric-key cryptographic algorithm designed for resource-constrained environments, such as IoT devices and embedded systems. It is a Korean lightweight cryptographic standard that uses an ARX structure (Addition, Rotation, and XOR operations) and a Feistel network. CHAM has three different schemes to cater to various security and performance requirements:

1. **CHAM-64/128**: Block size (n) = 64 bits, secret key size (k) = 128 bits, word size (w) = 16 bits, number of rounds (r) = 88.
2. **CHAM-128/128**: Block size (n) = 128 bits, secret key size (k) = 128 bits, word size (w) = 32 bits, number of rounds (r) = 112.
3. **CHAM-128/256**: Block size (n) = 128 bits, secret key size (k) = 256 bits, word size (w) = 32 bits, number of rounds (r) = 120.

The CHAM block cipher processes plaintext through multiple rounds of a round function, which is slightly different for odd and even rounds:

Odd rounds:

1. Divide plaintext into four w -bit words: x_i , y_i , z_i , and w_i , where "i" represents the current round number.
2. XOR x_i with the round number (i).
3. Rotate y_i to the left by 1 bit.
4. XOR the result of step 3 with the round key, which is the $(i \bmod 2k/w)$ th key.
5. Add the results of steps 2 and 4.
6. Rotate the result of step 5 to the left by 8 bits.
7. Rotate each word to the left (e.g., x_0 moves to the w_1 position, y_0 moves to the x_1 position, etc.).

Even rounds:

(Omitted, because similar with odd rounds.)

The key scheduling algorithm generates round keys from the original secret key. The secret key is divided into k/w w -bit words, $K[0]$, $K[1]$, ..., $K[k/w-1]$. The algorithm then generates $2k/w$ w -bit round keys, $rk[0]$, $rk[1]$, ..., $rk[2k/w-1]$:

For $0 \leq i < k/w$:

$$rk[i] = K[i] \oplus (K[i] \lll 1) \oplus (K[i] \lll 8)$$


$$rk[(i+k/w)^1] = K[i] \oplus (K[i] \lll 1) \oplus (K[i] \lll 11)$$

The round keys generated by the key scheduling algorithm are used in the round functions of the cipher to provide security and resistance against various cryptanalytic attacks. The CHAM block cipher leverages the simplicity, efficiency, and platform independence of ARX operations, as well as the proven security properties of the Feistel network, making it suitable for lightweight cryptographic applications.

If you could provide more information or context about the CHAM algorithm, I'd be happy to help answer any questions or discuss the topic further.

Listing 7: ChatGPT answers questions about the CHAM block cipher with fake information.

This demonstrates that ChatGPT has learned the CHAM block cipher. Based on this, an implementation request was made to see if ChatGPT could implement the learned cipher. ChatGPT implemented CHAM-64/128 and wrote the code. The initial code included key scheduling in the encryption process. A request was made to separate the code for modularization and to match the test vector values, with the output result visible in Listing 8.

 There is something slightly wrong. Please fix like this first. 1. Separate the key scheduling function from the encrypt function and write it separately. It should be written in the form of sending the round key obtained after key scheduling in the main function as a parameter to the encrypt function.

Therefore, an additional variable called round key must exist in the main function.

2. Modify the variables of the main function as follows.

key: 0x0100 0x0302 0x0504 0x0706 0x0908 0x0b0a 0x0d0c 0x0f0e

plaintext: 0x1100 0x3322 0x5544 0x7766

And, if you possible, add '_64128' to the main function's key and plaintext variable names.



I have made the requested changes to the code. The key scheduling function is now separated from the encryption function, and the main function has been modified accordingly. The key and plaintext values have also been updated as per your request. Here's the updated code snippet:

```

1 #include <stdint.h>
2 #include <stdio.h>
3
4 #define ROUNDS 88
5 #define KEY_SIZE 8
6 #define BLOCK_SIZE 4
7
8 void cham64_128_key_schedule(const uint16_t key[KEY_SIZE],
9     uint16_t round_keys[KEY_SIZE * 2]) {
10     for (int i = 0; i < KEY_SIZE; i++) {
11         round_keys[i] = key[i] ^ (key[i] << 1) ^ (key[i] <<
12             8);
13         round_keys[i + KEY_SIZE] = key[i] ^ (key[i] << 1) ^ (
14             key[i] << 11);
15     }
16 }

```

```

15 void cham64_128_encrypt(uint16_t plaintext[BLOCK_SIZE], const
    uint16_t round_keys[KEY_SIZE * 2]) {
16     for (int round = 1; round <= ROUNDS; round++) {
17         uint16_t x = plaintext[0], y = plaintext[1];
18         if (round % 2 == 1) {
19             x ^= round;
20             y = (y << 1) | (y >> 15);
21         } else {
22             x ^= round;
23             y = (y << 8) | (y >> 8);
24         }
25
26         uint16_t temp = x ^ y ^ round_keys[(round - 1) % (
    KEY_SIZE * 2)];
27         if (round % 2 == 1) {
28             temp = (temp << 8) | (temp >> 8);
29         } else {
30             temp = (temp << 1) | (temp >> 15);
31         }
32
33         plaintext[0] = plaintext[2] ^ temp;
34         plaintext[1] = plaintext[3];
35
36         // Rotate words
37         uint16_t temp_word = plaintext[0];
38         plaintext[0] = plaintext[1];
39         plaintext[1] = plaintext[2];
40         plaintext[2] = plaintext[3];
41         plaintext[3] = temp_word;
42     }
43 }
44
45 int main() {
46     uint16_t key_64128[KEY_SIZE] = {0x0100, 0x0302, 0x0504, 0
    x0706, 0x0908, 0x0b0a, 0x0d0c, 0x0f0e};
47     uint16_t plaintext_64128[BLOCK_SIZE] = {0x1100, 0x3322, 0
    x5544, 0x7766};
48     uint16_t round_keys[KEY_SIZE * 2];
49
50     cham64_128_key_schedule(key_64128, round_keys);
51     cham64_128_encrypt(plaintext_64128, round_keys);
52
53     printf("Encrypted: ");
54     for (int i = 0; i < BLOCK_SIZE; i++) {
55         printf("%04X ", plaintext_64128[i]);
56     }
57     printf("\n");
58
59     return 0;
60 }

```

This code snippet now has a separate key scheduling function, `cham64_128_key_schedule`, that generates the round keys. The main function calls this function to generate the round keys and then passes them to the encryption function `cham64_128_encrypt`. The key and plain-text variables in the main function have been updated and their names have been changed to include `'_64128'`.

Listing 8: ChatGPT answers questions about the CHAM block cipher with fake information.

At first glance, the source code appears well-written. Variables and function names are compatible, allowing them to be called together and compiled. However, the test vector values differ due to errors in the internal implementation. Specifically, the incorrect aspects are as follows. The lines of the source code refer to the line numbers in Listing 8.

- **Shift in Key scheduling.** In line 10, ChatGPT used the shift operation. CHAM Key scheduling requires the rotate operation.
- **Incorrect index variable in Round key array.** In line 11, `^1` was omitted. The correct source code should be `round_keys[i + KEY_SIZE ^ 1]`.
- **Invalid round constant start number.** In line 16, the condition of the for statement is written correctly to repeat exactly 88 rounds. However, the round constant value is inserted in the middle of the round function and should start from 0. The source code begins at 1, making this part incorrect.
- **Incorrect operator usage.** In line 26, `x ^ y` is invalid, and the `+` operation should be used instead.
- **No consideration of calculation priority.** In line 26, in the C language, the `+` operation is applied before the `^` operation. However, due to the structure of the round function, the `^` operation must come first. In other words, while using the `+` operator, parentheses need to be added to ensure the correct precedence is applied.
- **Insertion of unnecessary calculation processes.** In lines 33-34, these processes are unnecessary.

ChatGPT was asked to fix these issues. However, when one aspect was modified, problems arose in other parts, or previous errors reappeared in the code. After several attempts, the final code was obtained, as shown in Listing 9.



Then combine this main function with the code you wrote earlier and show it.



Here's the complete code with the updated main function combined with the key schedule and encryption functions:

```

1 #include <stdio.h>
2 #include <stdint.h>
3
4 #define BLOCK_SIZE_64_128 4
5 #define KEY_SIZE_64_128 8
6 #define ROUNDS_64_128 88
7
8 void cham64_128_key_schedule(const uint16_t key[
9     KEY_SIZE_64_128], uint16_t round_keys[KEY_SIZE_64_128 *
10     2]) {
11     for (int i = 0; i < KEY_SIZE_64_128; i++) {
12         round_keys[i] = key[i] ^ (key[i] << 1 | key[i] >> 15)
13         ^ (key[i] << 8 | key[i] >> 8);
14         round_keys[(i + KEY_SIZE_64_128) ^ 1] = key[i] ^ (key
15         [i] << 1 | key[i] >> 15) ^ (key[i] << 11 | key[i] >> 5);
16     }
17 }
18
19 void cham64_128_encrypt(const uint16_t plaintext[
20     BLOCK_SIZE_64_128], uint16_t ciphertext[BLOCK_SIZE_64_128
21     ], const uint16_t round_keys[KEY_SIZE_64_128 * 2]) {
22     for (int i = 0; i < BLOCK_SIZE_64_128; i++) {
23         ciphertext[i] = plaintext[i];
24     }
25
26     for (int round = 0; round < ROUNDS_64_128; round++) {
27         uint16_t x = ciphertext[0], y = ciphertext[1];
28
29         if (round % 2 == 0) {
30             x ^= round;
31             y = (y << 1) | (y >> 15);
32         } else {
33             x ^= round;
34             y = (y << 8) | (y >> 8);
35         }
36
37         uint16_t temp = x + (y ^ round_keys[round % (
38         KEY_SIZE_64_128 * 2)]);
39
40         if (round % 2 == 0) {
41             temp = (temp << 8) | (temp >> 8);
42         } else {
43             temp = (temp << 1) | (temp >> 15);
44         }
45
46         ciphertext[0] = temp;
47
48         // Step 7: Rotate words
49         uint16_t temp_word = ciphertext[0];
50         ciphertext[0] = ciphertext[1];

```

```

44     ciphertext[1] = ciphertext[2];
45     ciphertext[2] = ciphertext[3];
46     ciphertext[3] = temp_word;
47 }
48 }
49
50 int main() {
51     uint16_t key_64_128[KEY_SIZE_64_128] = {0x0100, 0x0302, 0
x0504, 0x0706, 0x0908, 0x0b0a, 0x0d0c, 0x0f0e};
52     uint16_t plaintext_64_128[BLOCK_SIZE_64_128] = {0x1100, 0
x3322, 0x5544, 0x7766};
53     uint16_t ciphertext_64_128[BLOCK_SIZE_64_128];
54     uint16_t round_keys_64_128[KEY_SIZE_64_128 * 2];
55
56     cham64_128_key_schedule(key_64_128, round_keys_64_128);
57     cham64_128_encrypt(plaintext_64_128, ciphertext_64_128,
round_keys_64_128);
58
59     printf("Secret Key:\n");
60     for (int i = 0; i < KEY_SIZE_64_128; i++) {
61         printf("%04x ", key_64_128[i]);
62     }
63     printf("\n");
64
65     printf("Plaintext:\n");
66     for (int i = 0; i < BLOCK_SIZE_64_128; i++) {
67         printf("%04x ", plaintext_64_128[i]);
68     }
69     printf("\n");
70
71     printf("Round Keys:\n");
72     for (int i = 0; i < KEY_SIZE_64_128 * 2; i++) {
73         printf("%04x ", round_keys_64_128[i]);
74         if (i % KEY_SIZE_64_128 == KEY_SIZE_64_128 - 1) {
75             printf("\n");
76         }
77     }
78
79     printf("Encrypted Plaintext (Ciphertext):\n");
80     for (int i = 0; i < BLOCK_SIZE_64_128; i++) {
81         printf("%04x ", ciphertext_64_128[i]);
82     }
83     printf("\n");
84
85     return 0;
86 }

```

This is the complete source code, including the key scheduling, encryption function, and main function, with separate variables for plaintext and ciphertext.

Listing 9: Source code of CHAM-64/128 completed by ChatGPT.

Based on this source code, all CHAM schemes were implemented by requesting porting to CHAM-128/128 and CHAM-128/256. Running the code produces output like that in Figure 4, which matches the CHAM test vector [6]. This demonstrates that ChatGPT successfully implemented the CHAM block cipher. The overall implementation process and source code can be found in Appendix 5.2.

```

CHAM-64/128
Secret Key: 0100 0302 0504 0706 0908 0b0a 0d0c 0f0e
Plaintext: 1100 3322 5544 7766
Round Keys: 0301 0705 0b09 0f0d 1311 1715 1b19 1f1d 151e 0308 3932 2f24 4d46 5b50 616a 777c
Ciphertext: 6579 1204 123f e5a9

CHAM-128/128
Secret Key: 03020100 07060504 0b0a0908 0f0e0d0c
Plaintext: 33221100 77665544 bbaa9988 ffeeddcc
Round Keys: 07070303 0f0f0b0b 17171313 1f1f1b1b 39222f34 150e0318 617a776c 4d565b40
Ciphertext: d05419ee 9f118f4c 99e36469 1c885ec1

CHAM-128/256
Secret Key: 03020100 07060504 0b0a0908 0f0e0d0c f3f2f1f0 f7f6f5f4 fbfaf9f8 fffefdfc
Plaintext: 33221100 77665544 bbaa9988 ffeeddcc
Round Keys: 07070303 0f0f0b0b 17171313 1f1f1b1b e6e6e2e2 eeeeeaaa f6f6f2f2 fefefafa 39222f34
Ciphertext: 027377dc 120b5651 8f839b95 5e5ec075

```

Fig. 4: The running result of the CHAM source code generated by ChatGPT.

4 Conclusion

This paper demonstrated the feasibility of implementing encryption algorithms through ChatGPT, enabling individuals to develop encryption solely by conversing with ChatGPT without writing any source code themselves. The experiment revealed that implementing well-known algorithms was quite straightforward. There was no need to explain the algorithm's structure during the implementation process, and ChatGPT generated source code precisely according to the requirements. The resulting source code compiled without errors and matched the test vectors accurately.

However, implementing lesser-known algorithms proved more challenging. Since ChatGPT was unfamiliar with these algorithms, it was necessary to teach their structure. Initially, the algorithm was learned effectively, but some structure

was lost as the conversation continued. Furthermore, numerous errors were discovered when implementing the learned algorithm in C language. Although the source code compiled successfully, the test vector did not match due to inaccuracies in the algorithm's structure implementation. To address this, additional fix requests were sent to ChatGPT. For those unfamiliar with the algorithm or who struggle with source code analysis, pinpointing which parts need fixing might be difficult when working with ChatGPT. Nonetheless, the process was still more manageable than implementing a block cipher from scratch. Specifically, designing the modularized function calls was something developers needed to do themselves, yet ChatGPT quickly generated accurate function call locations and naming conventions.


In conclusion, implementing encryption algorithms through ChatGPT was achieved successfully. Cryptographic implementation is a challenging task, as it demands precise execution of intricate algorithms. While ChatGPT made some errors, it was capable of modifying the source code and successfully implementing the target cipher when the incorrect aspects were highlighted. In essence, ChatGPT can facilitate a significant portion of cryptographic implementation. In particular, for cryptographic algorithms with multiple schemes, porting can be easily requested. However, it is essential to closely analyze the code generated by ChatGPT to identify and correct any mistakes. While ChatGPT can be asked to fix errors, it is often faster and more accurate for the programmer to make the necessary adjustments themselves.


References

1. M. Aljanabi, M. Ghazi, A. H. Ali, S. A. Abed, *et al.*, “ChatGPT: Open possibilities,” *Iraqi Journal For Computer Science and Mathematics*, vol. 4, no. 1, pp. 62–64, 2023.
2. T. H. Kung, M. Cheatham, A. Medenilla, C. Sillos, L. De Leon, C. Elepaño, M. Madriaga, R. Aggabao, G. Diaz-Candido, J. Maningo, *et al.*, “Performance of ChatGPT on USMLE: Potential for AI-assisted medical education using large language models,” *PLoS digital health*, vol. 2, no. 2, p. e0000198, 2023.
3. J.-H. Byeon and Y.-J. Kwon, “An investigation of generative AI ineducational application: Focusing on the usage of ChatGPT for learning biology,” *Brain, Digital Learning*, vol. 13, no. 1, pp. 1–17, 2023.
4. C. Sanchez-Avila and R. Sanchez-Reillo, “The rijndael block cipher (AES proposal): a comparison with DES,” in *Proceedings IEEE 35th Annual 2001 international carnahan conference on security technology (Cat. No. 01CH37186)*, pp. 229–234, IEEE, 2001.
5. B. Koo, D. Roh, H. Kim, Y. Jung, D.-G. Lee, and D. Kwon, “CHAM: A family of lightweight block ciphers for resource-constrained devices,” in *Information Security and Cryptology-ICISC 2017: 20th International Conference, Seoul, South Korea, November 29-December 1, 2017, Revised Selected Papers 20*, pp. 3–25, Springer, 2018.
6. D. Roh, B. Koo, Y. Jung, I. W. Jeong, D.-G. Lee, D. Kwon, and W.-H. Kim, “Revised version of block cipher CHAM,” in *Information Security and Cryptology-ICISC 2019: 22nd International Conference, Seoul, South Korea, December 4-6, 2019, Revised Selected Papers 22*, pp. 1–19, Springer, 2020.
7. H. Seo, “Memory-efficient implementation of ultra-lightweight block cipher algorithm cham on low-end 8-bit AVR processors,” *Journal of the Korea Institute of Information Security & Cryptology*, vol. 28, no. 3, pp. 545–550, 2018.
8. H. Kwon and J. Ha, “Fault injection attack on lightweight block cipher CHAM,” *Journal of the Korea Institute of Information Security & Cryptology*, vol. 28, no. 5, pp. 1071–1078, 2018.
9. M.-J. Sim, S.-W. Eum, H.-D. Kwon, G.-J. Song, and H.-J. Seo, “Implementation of ultra-lightweight block cipher algorithm revised CHAM on 32-bit risc-v processor,” in *Proceedings of the Korea Information Processing Society Conference*, pp. 217–220, Korea Information Processing Society, 2021.
10. D. Baidoo-Anu and L. Owusu Ansah, “Education in the era of generative artificial intelligence (AI): Understanding the potential benefits of ChatGPT in promoting teaching and learning,” *Available at SSRN 4337484*, 2023.
11. M. J. Dworkin, E. B. Barker, J. R. Nechvatal, J. Foti, L. E. Bassham, E. Roback, and J. F. Dray Jr, “Advanced encryption standard (AES),” 2001.

5 Appendix

5.1 Source Code for AES Generated by ChatGPT and Test Vectors

 I want implement AES algorithm written in C language. Before start this one, do you know about structures of AES algorithm?

 Yes, I'm familiar with the Advanced Encryption Standard (AES) algorithm. AES is a symmetric key encryption algorithm, which means it uses the same key for both encryption and decryption. It operates on fixed block sizes of 128 bits and supports key sizes of 128, 192, or 256 bits. The algorithm consists of several stages, including key expansion, initial round, main rounds, and a final round.

Here's a high-level overview of the main stages in AES encryption:

1. Key expansion: The original key is expanded into an array of key schedule words, which will be used in subsequent rounds of the algorithm.
2. Initial round:
 - AddRoundKey: The input block (plaintext) is XORed with the initial round key derived from the key schedule.
3. Main rounds (9, 11, or 13 rounds, depending on the key size):
 - SubBytes: A non-linear substitution step where each byte of the block is replaced with a corresponding byte from a pre-determined substitution table (S-box).
 - ShiftRows: A transposition step where the last three rows of the state are shifted cyclically a certain number of steps.
 - MixColumns: A linear mixing operation that operates on the columns of the state, combining the four bytes in each column using a pre-determined matrix.
 - AddRoundKey: The result of the previous three steps is XORed with a round key derived from the key schedule.
4. Final round (no MixColumns):
 - SubBytes
 - ShiftRows
 - AddRoundKey

The decryption process follows a similar structure but in reverse order, using inverse operations for SubBytes, ShiftRows, and MixColumns.

Now that you have an overview of the AES algorithm, you can start implementing it in C. There are many open-source libraries available that you can use as a reference or starting point.

Keep in mind that implementing cryptographic algorithms from scratch can be challenging and prone to errors. It's often recommended to use well-established libraries, such as OpenSSL or mbedtls, for production use cases.

Listing 10: Asked about AES algorithm.



No. I don't want to use open-source libraries.



Yes, I'm familiar with the Advanced Encryption Standard (AES) algorithm. AES is a symmetric key encryption algorithm, which means it uses the same key for both encryption and decryption. It operates on fixed block sizes of 128 bits and supports key sizes of 128, 192, or 256 bits. The algorithm consists of several stages, including key expansion, initial round, main rounds, and a final round.

I understand that you want to implement AES from scratch in C. Here's a basic outline to help you get started:

1. Define constants and lookup tables:
 - Define the key size (128, 192, or 256 bits).
 - Define the block size (128 bits).
 - Define the number of rounds (10, 12, or 14, depending on the key size).
 - Create the S-box and inverse S-box tables for the SubBytes and InvSubBytes steps.
 - Create the Rcon table for the key expansion.
2. Implement helper functions:
 - SubBytes() and InvSubBytes(): Apply the S-box or inverse S-box to each byte in the state.
 - ShiftRows() and InvShiftRows(): Perform row-wise cyclic shifts on the state.
 - MixColumns() and InvMixColumns(): Apply the MixColumns and inverse MixColumns transformations to the state.
 - AddRoundKey(): XOR the state with the round key.
3. Implement key expansion:
 - Write a function to expand the input key into an array of round keys.
 - Use the Rcon table and the SubBytes step in the key expansion.
4. Implement the main AES functions:
 - AES_Encrypt(): Implement the encryption process by applying the initial round, main rounds, and final round in the correct order.
 - AES_Decrypt(): Implement the decryption process by applying the inverse steps in the correct order.
5. Implement a test program to verify your implementation:
 - Use test vectors from the AES specification (FIPS 197) or other sources to validate your implementation.

Here's a rough skeleton for the AES implementation in C:

```

1 #include <stdio.h>
2 #include <stdint.h>
3
4 // Constants and lookup tables...
```


```


5
6 // Helper functions...
7
8 // Key expansion...
9
10 void AES_Encrypt(uint8_t *input, uint8_t *output, uint8_t *
    key) {
11     // Implement encryption...
12 }
13
14 void AES_Decrypt(uint8_t *input, uint8_t *output, uint8_t *
    key) {
15     // Implement decryption...
16 }
17
18 int main() {
19     // Test your implementation...
20     return 0;
21 }

```

Remember to handle security aspects, such as clearing sensitive data from memory when it's no longer needed. Additionally, be mindful of potential side-channel attacks and other vulnerabilities when writing cryptographic code.

Listing 11: Get skeleton for AES source code.

 Now that we have a rough, let's implement AES in C language. First, create a look-up table.

 To implement AES in C, you'll need to create a lookup table for the SubBytes step, which is also known as the S-box, and its inverse, the inverse S-box. These tables are pre-computed and consist of 256 bytes each.

Here's the S-box and inverse S-box lookup tables:

```

1 #include <stdint.h>
2
3 // S-box (SubBytes)
4 static const uint8_t s_box[256] = {
5     0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F, 0xC5, 0x30, 0
6     x01, 0x67, 0x2B, 0xFE, 0xD7, 0xAB, 0x76,
7     0xCA, 0x82, 0xC9, 0x7D, 0xFA, 0x59, 0x47, 0xF0, 0xAD, 0
8     xD4, 0xA2, 0xAF, 0x9C, 0xA4, 0x72, 0xC0,

```


```

7   0xB7, 0xFD, 0x93, 0x26, 0x36, 0x3F, 0xF7, 0xCC, 0x34, 0
   xA5, 0xE5, 0xF1, 0x71, 0xD8, 0x31, 0x15,
8   0x04, 0xC7, 0x23, 0xC3, 0x18, 0x96, 0x05, 0x9A, 0x07, 0
   x12, 0x80, 0xE2, 0xEB, 0x27, 0xB2, 0x75,
9   0x09, 0x83, 0x2C, 0x1A, 0x1B, 0x6E, 0x5A, 0xA0, 0x52, 0
   x3B, 0xD6, 0xB3, 0x29, 0xE3, 0x2F, 0x84,
10  0x53, 0xD1, 0x00, 0xED, 0x20, 0xFC, 0xB1, 0x5B, 0x6A, 0
   xCB, 0xBE, 0x39, 0x4A, 0x4C, 0x58, 0xCF,
11  0xD0, 0xEF, 0xAA, 0xFB, 0x43, 0x4D, 0x33, 0x85, 0x45, 0
   xF9, 0x02, 0x7F, 0x50, 0x3C, 0x9F, 0xA8,
12  0x51, 0xA3, 0x40, 0x8F, 0x92, 0x9D, 0x38, 0xF5, 0xBC, 0
   xB6, 0xDA, 0x21, 0x10, 0xFF, 0xF3, 0xD2,
13  0xCD, 0x0C, 0x13, 0xEC, 0

```

Listing 12: S-Box code with output stopped due to token limit.

 Okay. I think you can't complete the S-box. Instead, just declare a look-up table array, I'll fill in the LUT values.

 Certainly! Here's the declaration of the S-box and inverse S-box lookup tables as uninitialized arrays:
Here's the S-box and inverse S-box lookup tables:


```


1  #include <stdint.h>
2
3  // S-box (SubBytes)
4  static const uint8_t s_box[256];
5
6  // Inverse S-box (InvSubBytes)
7  static const uint8_t inv_s_box[256];

```

You can then fill in the LUT values for both arrays. Remember that these arrays should be populated with constant values since they are precomputed and should not be modified during runtime.

Listing 13: Declaration of S-Box array.

 Now we will implement AES key expansion. Do you know the principle behind that algorithm? Please explain.

 Yes, I can explain the key expansion algorithm used in AES. The key expansion algorithm generates an extended key schedule from the initial cipher key. The key schedule is an array of words (4-byte values) that are used as round keys in each round of the AES algorithm. The number of words in the key schedule depends on the key size and the number of rounds.

The key expansion is based on the following principles:

1. Word-wise operations: The algorithm operates on words (4-byte values) rather than individual bytes.
2. Key schedule size: The key schedule size depends on the number of rounds and the key size. For a 128-bit key, there are 44 words in the key schedule (11 rounds * 4 words per round); for a 192-bit key, there are 52 words (13 rounds * 4 words); and for a 256-bit key, there are 60 words (15 rounds * 4 words).
3. Copying the initial key: The first N words of the key schedule (where N is the size of the initial key in words: 4 for a 128-bit key, 6 for a 192-bit key, and 8 for a 256-bit key) are directly copied from the initial key.
4. Key schedule core: The remaining words in the key schedule are generated iteratively using the previous words. The algorithm uses a core function that operates on a single word and consists of the following steps:
 - RotWord: Rotate the word one byte to the left (circular shift).
 - SubWord: Apply the S-box substitution to each byte in the word.
 - Rcon: XOR the first byte of the word with a round constant. The round constants are precomputed and stored in a lookup table called the Rcon table.
5. Combining words: The core function is applied to every (Nth) word in the key schedule (where N is the size of the initial key in words). The output of the core function is XORed with the word N positions earlier in the key schedule. The result of this XOR operation is then XORed with the word immediately preceding it to generate the next word in the key schedule. This process is repeated until the entire key schedule is filled.

Here's a high-level pseudocode description of the key expansion algorithm:

- ```


1 1. Copy the initial key to the beginning of the key schedule.
2 2. For each remaining word in the key schedule:


```

- 3 a. If the word index is a multiple of N, apply the core  
function and XOR the result with the word N positions  
earlier.
- 4 b. XOR the result with the word immediately preceding it.

Now that you understand the principle behind the key expansion algorithm, you can proceed with implementing it in C.

Listing 14: Asked about AES key expansion.

 Then implement AES key expansion in C language. If you are going to use the Rcon table as well, define this as a global variable.

 Here's an implementation of the AES key expansion in C, along with the Rcon table as a global variable:

```

1 #include <stdint.h>
2
3 // Rcon table for key expansion
4 static const uint8_t rcon[11] = {
5 0x00, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0
6 x1B, 0x36
7 };
8 // S-box (SubBytes) - Please fill in the values
9 static const uint8_t s_box[256];
10
11 // Helper functions
12 static uint32_t RotWord(uint32_t word) {
13 return (word << 8) | (word >> 24);
14 }
15
16 static uint32_t SubWord(uint32_t word) {
17 uint32_t result = 0;
18 result |= s_box[(word >> 24) & 0xFF] << 24;
19 result |= s_box[(word >> 16) & 0xFF] << 16;
20 result |= s_box[(word >> 8) & 0xFF] << 8;
21 result |= s_box[word & 0xFF];
22 return result;
23 }
24
25 // Key expansion
26 void AES_KeyExpansion(const uint8_t *key, uint32_t *
27 key_schedule, int key_size) {
28 int num_words = key_size / 32; // Number of words in the
29 initial key

```

```


28 int key_schedule_size = (key_size == 128) ? 44 : (
key_size == 192) ? 52 : 60;
29
30 // Copy the initial key to the key schedule
31 for (int i = 0; i < num_words; i++) {
32 key_schedule[i] = (key[4 * i] << 24) | (key[4 * i +
1] << 16) | (key[4 * i + 2] << 8) | key[4 * i + 3];
33 }
34
35 // Generate the remaining words in the key schedule
36 for (int i = num_words; i < key_schedule_size; i++) {
37 uint32_t temp = key_schedule[i - 1];
38
39 if (i % num_words == 0) {
40 temp = SubWord(RotWord(temp)) ^ (rcon[i /
num_words] << 24);
41 } else if (key_size == 256 && i % num_words == 4) {
42 temp = SubWord(temp);
43 }
44
45 key_schedule[i] = key_schedule[i - num_words] ^ temp;
46 }
47 }


```

Please make sure to fill in the S-box values in the 's\_box' array before using this code.

This implementation assumes that the key size is given in bits (128, 192, or 256) and that the 'key' array contains the appropriate number of bytes for the specified key size. The 'key\_schedule' array should be allocated with enough space to hold the full key schedule (44, 52, or 60 words, depending on the key size).

Listing 15: Implementation of AES key expansion.

 Now we need Subbyte and its inversed version. Please implement the process in C language. As for the S-box table, you can use a table that has been previously declared. I'll fill the table.

 Sure! Here's an implementation of the SubBytes and InvSubBytes functions in C, using the previously declared S-box ('s\_box') and inverse S-box ('inv\_s\_box') tables:

```

1 #include <stdint.h>
2

```

```

3 // S-box (SubBytes) - Please fill in the values
4 static const uint8_t s_box[256];
5
6 // Inverse S-box (InvSubBytes) - Please fill in the values
7 static const uint8_t inv_s_box[256];
8
9 // Helper functions
10 void SubBytes(uint8_t state[4][4]) {
11 for (int row = 0; row < 4; row++) {
12 for (int col = 0; col < 4; col++) {
13 state[row][col] = s_box[state[row][col]];
14 }
15 }
16 }
17
18 void InvSubBytes(uint8_t state[4][4]) {
19 for (int row = 0; row < 4; row++) {
20 for (int col = 0; col < 4; col++) {
21 state[row][col] = inv_s_box[state[row][col]];
22 }
23 }
24 }

```

These functions perform the SubBytes and InvSubBytes transformations on the input state array, which is a 4x4 array of bytes representing the AES state. The input state is modified in-place, so no separate output array is required.

Please make sure to fill in the values for both the 's\_box' and 'inv\_s\_box' arrays before using these functions.

Listing 16: Implementation of AES SubBytes (and InvSubBytes).



Good. Now write Shiftrow and its inverse version in C language.



Here's an implementation of the ShiftRows and InvShiftRows functions in C:

```

1 #include <stdint.h>
2
3 // Helper functions
4 void ShiftRows(uint8_t state[4][4]) {
5 uint8_t temp;
6
7 // Row 1 - Shift left by 1
8 temp = state[1][0];

```




```
9 state[1][0] = state[1][1];
10 state[1][1] = state[1][2];
11 state[1][2] = state[1][3];
12 state[1][3] = temp;
13
14 // Row 2 - Shift left by 2
15 temp = state[2][0];
16 state[2][0] = state[2][2];
17 state[2][2] = temp;
18 temp = state[2][1];
19 state[2][1] = state[2][3];
20 state[2][3] = temp;
21
22 // Row 3 - Shift left by 3
23 temp = state[3][0];
24 state[3][0] = state[3][3];
25 state[3][3] = state[3][2];
26 state[3][2] = state[3][1];
27 state[3][1] = temp;
28 }
29
30 void InvShiftRows(uint8_t state[4][4]) {
31 uint8_t temp;
32
33 // Row 1 - Shift right by 1
34 temp = state[1][3];
35 state[1][3] = state[1][2];
36 state[1][2] = state[1][1];
37 state[1][1] = state[1][0];
38 state[1][0] = temp;
39
40 // Row 2 - Shift right by 2
41 temp = state[2][0];
42 state[2][0] = state[2][2];
43 state[2][2] = temp;
44 temp = state[2][1];
45 state[2][1] = state[2][3];
46 state[2][3] = temp;
47
48 // Row 3 - Shift right by 3
49 temp = state[3][0];
50 state[3][0] = state[3][1];
51 state[3][1] = state[3][2];
52 state[3][2] = state[3][3];
53 state[3][3] = temp;
54 }
```

These functions perform the ShiftRows and InvShiftRows transformations on the input 'state' array, which is a 4x4 array of bytes representing the AES state. The input state is modified in-place, so no separate output array is required.

Listing 17: Implementation of AES ShiftRows (and InvShiftRows).

 Here are MixColumns and their inverse version. Write it in C language as well.

 Here's an implementation of the MixColumns and InvMixColumns functions in C:

```

1 #include <stdint.h>
2
3 // Helper function for Galois Field multiplication
4 uint8_t gmul(uint8_t a, uint8_t b) {
5 uint8_t p = 0;
6 uint8_t carry;
7
8 for (int i = 0; i < 8; i++) {
9 if (b & 1) {
10 p ^= a;
11 }
12
13 carry = a & 0x80;
14 a <<= 1;
15
16 if (carry) {
17 a ^= 0x1B; // x^8 + x^4 + x^3 + x + 1
18 }
19
20 b >>= 1;
21 }
22
23 return p;
24 }
25
26 void MixColumns(uint8_t state[4][4]) {
27 uint8_t temp[4];
28
29 for (int col = 0; col < 4; col++) {
30 temp[0] = gmul(state[0][col], 0x02) ^ gmul(state[1][col], 0x03) ^ state[2][col] ^ state[3][col];
31 temp[1] = state[0][col] ^ gmul(state[1][col], 0x02) ^ gmul(state[2][col], 0x03) ^ state[3][col];

```

```

32 temp[2] = state[0][col] ^ state[1][col] ^ gmul(state
33 [2][col], 0x02) ^ gmul(state[3][col], 0x03);
34 temp[3] = gmul(state[0][col], 0x03) ^ state[1][col] ^
35 state[2][col] ^ gmul(state[3][col], 0x02);
36
37 for (int row = 0; row < 4; row++) {
38 state[row][col] = temp[row];
39 }
40 }
41 void InvMixColumns(uint8_t state[4][4]) {
42 uint8_t temp[4];
43
44 for (int col = 0; col < 4; col++) {
45 temp[0] = gmul(state[0][col], 0x0E) ^ gmul(state[1][
46 col], 0x0B) ^ gmul(state[2][col], 0x0D) ^ gmul(state[3][
47 col], 0x09);
48 temp[1] = gmul(state[0][col], 0x09) ^ gmul(state[1][
49 col], 0x0E) ^ gmul(state[2][col], 0x0B) ^ gmul(state[3][
50 col], 0x0D);
51 temp[2] = gmul(state[0][col], 0x0D) ^ gmul(state[1][
52 col], 0x09) ^ gmul(state[2][col], 0x0E) ^ gmul(state[3][
53 col], 0x0B);
54 temp[3] = gmul(state[0][col], 0x0B) ^ gmul(state[1][
55 col], 0x0D) ^ gmul(state[2][col], 0x09) ^ gmul(state[3][
56 col], 0x0E);
57
58 for (int row = 0; row < 4; row++) {
59 state[row][col] = temp[row];
60 }
61 }
62 }

```

These functions perform the MixColumns and InvMixColumns transformations on the input 'state' array,

Listing 18: Implementation of AES MixColumns (and InvMixColumns). Response is truncated due to token limit, but the source code is output well.



Lastly, write the AddRoundKey function in C language.



Here's an implementation of the AddRoundKey function in C:

```

1 #include <stdint.h>

```


```


2
3 void AddRoundKey(uint8_t state[4][4], const uint32_t *
 round_key) {
4 for (int col = 0; col < 4; col++) {
5 uint32_t key_word = round_key[col];
6
7 for (int row = 0; row < 4; row++) {
8 state[row][col] ^= (key_word >> (24 - 8 * row)) &
 0xFF;
9 }
10 }
11 }

```

This function performs the AddRoundKey transformation on the input 'state' array, which is a 4x4 array of bytes representing the AES state. The 'round\_key' parameter is a pointer to an array of four 32-bit words representing the round key. The input state is modified in-place, so no separate output array is required.

Listing 19: Implementation of AES AddRoundKey.

 Now we implemented Key expansion, SubBytes(InvSubBytes), ShiftRows(InvShiftRows), MixColumns(InvMixColumns), and AddRoundKey in C language. Use these to complete the Encrypt rough you wrote at the beginning.

 Here's the complete AES encryption implementation using the previously defined functions:

```

1 #include <stdint.h>
2
3 // S-box (SubBytes) - Please fill in the values
4 static const uint8_t s_box[256];
5
6 // Inverse S-box (InvSubBytes) - Please fill in the values
7 static const uint8_t inv_s_box[256];
8
9 // Rcon table for key expansion
10 static const uint8_t rcon[11] = {
11 0x00, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0
 x1B, 0x36
12 };
13
14 // Previously defined helper functions and transformations go
 here

```

```

15
16 void AES_Encrypt(const uint8_t *plaintext, const uint8_t *key
17 , uint8_t *ciphertext, int key_size) {
18 uint8_t state[4][4];
19 uint32_t key_schedule[60]; // Maximum size (for 256-bit
20 key)
21
22 // Initialize state array with input plaintext
23 for (int row = 0; row < 4; row++) {
24 for (int col = 0; col < 4; col++) {
25 state[row][col] = plaintext[row + 4 * col];
26 }
27 }
28
29 // Perform key expansion
30 AES_KeyExpansion(key, key_schedule, key_size);
31
32 // Initial AddRoundKey
33 AddRoundKey(state, key_schedule);
34
35 // Main rounds
36 int num_rounds = (key_size == 128) ? 10 : (key_size ==
37 192) ? 12 : 14;
38 for (int round = 1; round < num_rounds; round++) {
39 SubBytes(state);
40 ShiftRows(state);
41 MixColumns(state);
42 AddRoundKey(state, key_schedule + 4 * round);
43 }
44
45 // Final round (without MixColumns)
46 SubBytes(state);
47 ShiftRows(state);
48 AddRoundKey(state, key_schedule + 4 * num_rounds);
49
50 // Copy state array to output ciphertext
51 for (int row = 0; row < 4; row++) {
52 for (int col = 0; col < 4; col++) {
53 ciphertext[row + 4 * col] = state[row][col];
54 }
55 }
56 }

```

This function takes the input 'plaintext', 'key', and 'key\_size' (in bits: 128, 192, or 256) and produces the encrypted output 'ciphertext'. The plaintext and ciphertext are assumed to be 16-byte arrays, while the key size determines the number of bytes in the key array. Please make sure to fill in the 's\_box' and 'inv\_s\_box' arrays before using this code.

Listing 20: Implementation of AES encrypt function.



Complete the Decrypt function in the same way.



Here's the complete AES decryption implementation using the previously defined functions:

```

1 #include <stdint.h>
2
3 // S-box (SubBytes) - Please fill in the values
4 static const uint8_t s_box[256];
5
6 // Inverse S-box (InvSubBytes) - Please fill in the values
7 static const uint8_t inv_s_box[256];
8
9 // Rcon table for key expansion
10 static const uint8_t rcon[11] = {
11 0x00, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0
12 x1B, 0x36
13 };
14 // Previously defined helper functions and transformations go
15 here
16 void AES_Decrypt(const uint8_t *ciphertext, const uint8_t *
17 key, uint8_t *plaintext, int key_size) {
18 uint8_t state[4][4];
19 uint32_t key_schedule[60]; // Maximum size (for 256-bit
20 key)
21
22 // Initialize state array with input ciphertext
23 for (int row = 0; row < 4; row++) {
24 for (int col = 0; col < 4; col++) {
25 state[row][col] = ciphertext[row + 4 * col];
26 }
27 }
28
29 // Perform key expansion

```

```


28 AES_KeyExpansion(key, key_schedule, key_size);
29
30 // Initial AddRoundKey
31 int num_rounds = (key_size == 128) ? 10 : (key_size ==
192) ? 12 : 14;
32 AddRoundKey(state, key_schedule + 4 * num_rounds);
33
34 // Main rounds
35 for (int round = num_rounds - 1; round > 0; round--) {
36 InvShiftRows(state);
37 InvSubBytes(state);
38 AddRoundKey(state, key_schedule + 4 * round);
39 InvMixColumns(state);
40 }
41
42 // Final round (without InvMixColumns)
43 InvShiftRows(state);
44 InvSubBytes(state);
45 AddRoundKey(state, key_schedule);
46
47 // Copy state array to output plaintext
48 for (int row = 0; row < 4; row++) {
49 for (int col = 0; col < 4; col++) {
50 plaintext[row + 4 * col] = state[row][col];
51 }
52 }
53 }

```

This function takes the input 'ciphertext', 'key', and 'key\_size' (in bits: 128, 192, or 256) and produces the decrypted output 'plaintext'. The ciphertext and plaintext are assumed to be 16-byte arrays, while the key size determines the number of bytes in the key array. Please make sure to fill in the 's\_box' and 'inv\_s\_box' arrays before using this code.

Listing 21: Implementation of AES decrypt function.

**HD** Now create the main function. Write the main function to call the AES-128 Encryption and Decryption functions once. The key value is '000102030405060708090a0b0c0d0e0f' and the plaintext value is '00112233445566778899aabbccddeeff'. Declare the name of the key variable as 'key\_128', the name of the plaintext variable as 'plaintext\_128', the name of the ciphertext variable as 'ciphertext\_128', and the name of the decryption variable as 'decrypted\_128'.

 Here's a main function that demonstrates the use of the AES-128 Encryption and Decryption functions with the provided key and plaintext:

```

1 #include <stdio.h>
2 #include <stdint.h>
3
4 // S-box (SubBytes), Inverse S-box (InvSubBytes), Rcon table,
5 // and previously defined functions go here
6
7 int main() {
8 uint8_t key_128[16] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05,
9 , 0x06, 0x07,
10 , 0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D,
11 , 0x0E, 0x0F};
12
13 uint8_t plaintext_128[16] = {0x00, 0x11, 0x22, 0x33, 0x44,
14 , 0x55, 0x66, 0x77,
15 , 0x88, 0x99, 0xAA, 0xBB, 0xCC,
16 , 0xDD, 0xEE, 0xFF};
17
18 uint8_t ciphertext_128[16];
19 uint8_t decrypted_128[16];
20
21 AES_Encrypt(plaintext_128, key_128, ciphertext_128, 128);
22 AES_Decrypt(ciphertext_128, key_128, decrypted_128, 128);
23
24 printf("Plaintext:\n");
25 for (int i = 0; i < 16; i++) {
26 printf("%02x ", plaintext_128[i]);
27 }
28 printf("\n");
29
30 printf("Ciphertext:\n");
31 for (int i = 0; i < 16; i++) {
32 printf("%02x ", ciphertext_128[i]);
33 }
34 printf("\n");

```



```


30
31 printf("Decrypted:\n");
32 for (int i = 0; i < 16; i++) {
33 printf("%02x ", decrypted_128[i]);
34 }
35 printf("\n");
36
37 return 0;
38 }

```

This 'main' function demonstrates the encryption and decryption process using the provided 'key\_128' and 'plaintext\_128'. The encrypted result is stored in 'ciphertext\_128', and the decrypted result is stored in 'decrypted\_128'. The program prints the plaintext, ciphertext, and decrypted values in hexadecimal format. Please ensure that you fill in the 's\_box' and 'inv\_s\_box' arrays and include the previously defined functions before running this code.

Listing 22: Implementing a main function that calls AES.

**HD** Modify the body of the main function to call AES-192 encryption and decryption. Use '000102030405060708090a0b0c0d0e0f1011121314151617' for the key value and '00112233445566778899aabbccddeeff' for the plaintext value. Declare the name of the key variable as 'key\_192', the name of the plaintext variable as 'plaintext\_192', the name of the ciphertext variable as 'ciphertext\_192', and the name of the decryption variable as 'decrypted\_192'.

 Here's the modified main function that demonstrates the use of the AES-192 Encryption and Decryption functions with the provided key and plaintext:

```

1 #include <stdio.h>
2 #include <stdint.h>
3
4 // S-box (SubBytes), Inverse S-box (InvSubBytes), Rcon table,
5 // and previously defined functions go here
6
7 int main() {
8 uint8_t key_192[24] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05,
9 0x06, 0x07,
10 0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D,
11 0x0E, 0x0F,

```

```


9 0x10, 0x11, 0x12, 0x13, 0x14, 0x15
 , 0x16, 0x17};
10
11 uint8_t plaintext_192[16] = {0x00, 0x11, 0x22, 0x33, 0x44
 , 0x55, 0x66, 0x77,
12 0x88, 0x99, 0xAA, 0xBB, 0xCC
 , 0xDD, 0xEE, 0xFF};
13
14 uint8_t ciphertext_192[16];
15 uint8_t decrypted_192[16];
16
17 AES_Encrypt(plaintext_192, key_192, ciphertext_192, 192);
18 AES_Decrypt(ciphertext_192, key_192, decrypted_192, 192);
19
20 printf("Plaintext:\n");
21 for (int i = 0; i < 16; i++) {
22 printf("%02x ", plaintext_192[i]);
23 }
24 printf("\n");
25
26 printf("Ciphertext:\n");
27 for (int i = 0; i < 16; i++) {
28 printf("%02x ", ciphertext_192[i]);
29 }
30 printf("\n");
31
32 printf("Decrypted:\n");
33 for (int i = 0; i < 16; i++) {
34 printf("%02x ", decrypted_192[i]);
35 }
36 printf("\n");
37
38 return 0;
39 }

```

This 'main' function demonstrates the encryption and decryption process using the provided 'key\_192' and 'plaintext\_192'. The encrypted result is stored in 'ciphertext\_192', and the decrypted result is stored in 'decrypted\_192'. The program prints the plaintext, ciphertext, and decrypted values in hexadecimal format. Please ensure that you fill in the 's\_box' and 'inv\_s\_box' arrays and include the previously defined functions before running this code.

Listing 23: Request porting to AES-192.

**HD** Modify the body of the main function to call AES-256 encryption and decryption. Use '000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f' for the key value, and use the plain text value used in AES-192 for the plaintext value. Declare the name of the key variable as 'key\_256', the name of the plaintext variable as 'plaintext\_256', the name of the ciphertext variable as 'ciphertext\_256', and the name of the decryption variable as 'decrypted\_256'.

 Here's the modified main function that demonstrates the use of the AES-256 Encryption and Decryption functions with the provided key and plaintext:

```

1 #include <stdio.h>
2 #include <stdint.h>
3
4 // S-box (SubBytes), Inverse S-box (InvSubBytes), Rcon table,
5 // and previously defined functions go here
6
7 int main() {
8 uint8_t key_256[32] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05
9 , 0x06, 0x07,
10 , 0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D
11 , 0x0E, 0x0F,
12 , 0x10, 0x11, 0x12, 0x13, 0x14, 0x15
13 , 0x16, 0x17,
14 , 0x18, 0x19, 0x1A, 0x1B, 0x1C, 0x1D
15 , 0x1E, 0x1F};
16
17 uint8_t plaintext_256[16] = {0x00, 0x11, 0x22, 0x33, 0x44
18 , 0x55, 0x66, 0x77,
19 , 0x88, 0x99, 0xAA, 0xBB, 0xCC
20 , 0xDD, 0xEE, 0xFF};
21
22 uint8_t ciphertext_256[16];
23 uint8_t decrypted_256[16];
24
25 AES_Encrypt(plaintext_256, key_256, ciphertext_256, 256);
26 AES_Decrypt(ciphertext_256, key_256, decrypted_256, 256);
27
28 printf("Plaintext:\n");
29 for (int i = 0; i < 16; i++) {
30 printf("%02x ", plaintext_256[i]);
31 }
32 printf("\n");
33
34 printf("Ciphertext:\n");

```

```

28 for (int i = 0; i < 16; i++) {
29 printf("%02x ", ciphertext_256[i]);
30 }
31 printf("\n");
32
33 printf("Decrypted:\n");
34 for (int i = 0; i < 16; i++) {
35 printf("%02x ", decrypted_256[i]);
36 }
37 printf("\n");
38
39 return 0;
40 }


```

This 'main' function demonstrates the encryption and decryption process using the provided 'key\_256' and 'plaintext\_256'. The encrypted result is stored in 'ciphertext\_256', and the decrypted result is stored in 'decrypted\_256'. The program prints the plaintext, ciphertext, and decrypted values in hexadecimal format. Please ensure that you fill in the 's\_box' and 'inv\_s\_box' arrays and include the previously defined functions before running this code.

Listing 24: Request porting to AES-256.

## 5.2 Source Code for CHAM Generated by ChatGPT and Test Vectors

 Do you know about CHAM block cipher? It is one of Korean standard cryptography.

 CHAM block cipher is not one of the well-known cryptographic standards I am familiar with, as my knowledge was last updated in 2021. It is possible that CHAM is a more recent development or is known under a different name in the international cryptographic community. I apologize for any confusion.

As of my last update, some of the widely-accepted cryptographic standards include the Advanced Encryption Standard (AES), which is used globally, and the ARIA block cipher, which was developed in South Korea and is used there as a national cryptographic standard.

If you have more information or context about the CHAM block cipher, I'd be happy to discuss the topic further.

Listing 25: Questions about the CHAM block cipher.

**HD** Okay. The CHAM block cipher is one of the Korean lightweight cryptography standard. It is announced at 2017, and revised version was announced at 2019.


 Thank you for providing more information about the CHAM block cipher. Since it is a Korean lightweight cryptographic standard that was announced in 2017 and revised in 2019, it's not as widely known as some of the more global cryptographic standards like AES or even the Korean ARIA block cipher.

Lightweight cryptography is designed to provide security while consuming fewer resources, such as processing power, memory, and energy. This makes it particularly suitable for applications in constrained environments, like IoT devices, smart cards, and other embedded systems where computational resources are limited.

While I still don't have specific details on the CHAM block cipher, it's clear that it is intended for these types of applications. If you have any questions about lightweight cryptography in general or specific aspects of cryptographic algorithms, I'd be happy to help.

Listing 26: Education on the history of the CHAM block cipher.

**HD** CHAM has three kinds of scheme. CHAM-64/128, CHAM-128/128, and CHAM-128/256. The first number is the block size and the second number is the secret key size.

 Thank you for providing more information about the CHAM block cipher variants. It appears that CHAM has three different schemes, each with different block and key sizes:

1. **CHAM-64/128:** This version of CHAM has a 64-bit block size and uses a 128-bit secret key. With a smaller block size, it is more suitable for applications with limited data size requirements and is likely faster and more resource-efficient compared to the other variants.

2. **CHAM-128/128:** In this version, both the block size and the secret key size are 128 bits. It provides a balance between security and performance, as the increased block size can improve security in certain use cases.

3. **CHAM-128/256:** This variant of CHAM has a 128-bit block size and a larger 256-bit secret key. The increased key size provides a higher level of security against brute-force attacks and other cryptanalytic techniques, making it suitable for applications with more stringent security requirements.

These different schemes allow CHAM to cater to various security and performance requirements in lightweight cryptographic applications. By providing different block and key size options, CHAM can be tailored to the specific needs of different use cases, especially in resource-constrained environments.

Listing 27: Teach about the CHAM block cipher schemes.



Here is the parameters of CHAM block cipher.

Notation:  $n$ =block size(bit),  $k$ =secret key size(bit),  $w$ =word size(bit),  $r$ =number of round

CHAM-64/128:  $n=64$ ,  $k=128$ ,  $w=16$ ,  $r=88$

CHAM-128/128:  $n=128$ ,  $k=128$ ,  $w=32$ ,  $r=112$

CHAM-128/256:  $n=128$ ,  $k=256$ ,  $w=32$ ,  $r=120$



Thank you for providing the detailed parameters for the CHAM block cipher variants. I will summarize the information you've given for each version:

1. CHAM-64/128:
  - Block size ( $n$ ): 64 bits
  - Secret key size ( $k$ ): 128 bits
  - Word size ( $w$ ): 16 bits
  - Number of rounds ( $r$ ): 88
2. CHAM-128/128:
  - Block size ( $n$ ): 128 bits
  - Secret key size ( $k$ ): 128 bits
  - Word size ( $w$ ): 32 bits
  - Number of rounds ( $r$ ): 112
3. CHAM-128/256:
  - Block size ( $n$ ): 128 bits
  - Secret key size ( $k$ ): 256 bits
  - Word size ( $w$ ): 32 bits
  - Number of rounds ( $r$ ): 120


These parameters provide a better understanding of the internal structure and configuration of each CHAM block cipher variant. The different block sizes, secret key sizes, word sizes, and number of rounds allow each version of CHAM to balance security and performance according to the needs of specific applications.

It is important to note that increasing the number of rounds can contribute to higher security, as it makes the cipher more resistant to cryptanalysis. However, more rounds also increase the computational complexity of the algorithm, which could affect the overall performance. Choosing the right CHAM variant for a particular use case requires a careful analysis of security requirements, performance constraints, and other factors relevant to the specific application.

Listing 28: Inform about parameters of CHAM block cipher.

**HD** Now I talk about round function of CHAM block cipher. CHAM's round function is slightly different for even and odd rounds. I will explain based on odd rounds.

1. Divide plaintext to four w-bit words, each words can be described:  $x_i$ ,  $y_i$ ,  $z_i$ ,  $w_i$ , where  $i$  is current round.
2.  $x_i$  XOR with  $i$
3.  $y_i$  rotate left 1 time
4. result of 3 XOR with round key, where round key is  $(i \bmod 2k/w)$ th
5. result of 2 ADD result of 4
6. result of 5 rotate left 8 times


 Thank you for providing a detailed explanation of the round function for the odd rounds in the CHAM block cipher. I will reiterate the steps you provided for the round function of odd rounds, assuming a generic round number " $i$ ":

1. Divide the plaintext into four w-bit words:  $x_i$ ,  $y_i$ ,  $z_i$ , and  $w_i$ , where " $i$ " represents the current round number.
2. XOR  $x_i$  with the round number ( $i$ ).
3. Rotate  $y_i$  to the left by 1 bit.
4. XOR the result of step 3 with the round key, which is the  $(i \bmod 2k/w)$ th key.
5. Add the results of steps 2 and 4.
6. Rotate the result of step 5 to the left by 8 bits.

These steps describe the round function for odd rounds in the CHAM block cipher. The even rounds would have a slightly different round function, which you have not provided. However, understanding the round function for odd rounds gives insight into the structure of the cipher and how it processes data. This process is repeated for a specific number of rounds (88, 112, or 120, depending on the CHAM variant) to provide the final ciphertext.

Listing 29: Explaining the structure of CHAM's round function (odd round).

**HD** You understood correctly. Even round function is almost same odd round function, but step 3 and 6 is slightly different. Here is difference of even round function.  
 Step 3. 8 times rotate left  
 Step 6. 1 time rotate left  
 And, I forgot to explain about word rotation.  
 So, you will need to add this step to the end of round function(odd and even round both).  
 7. Each word rotate to left. (E.g. x0 will move to w1 place, y0 will go to x1).

 Thank you for providing the additional information on the even round function for the CHAM block cipher and the word rotation step. I will now summarize the round function for both odd and even rounds, incorporating the differences and the word rotation step:  
 (Omitted, because similar with odd rounds.)

Listing 30: Explaining the structure of CHAM's round function (even round).



**HD** Perfect. In this time, we talk about key scheduling.

Key scheduling of CHAM block cipher is very easy. CHAM-n/k takes a secret key of  $k/w$   $w$ -bit words  $K[0], K[1], \dots, K[k/w-1]$ . These generates  $2k/w$   $w$ -bit round keyws  $rk[0], rk[1], \dots, rk[2k/w-1]$ .

Here is the equations for key scheduling.

Notation:  $\wedge$ : XOR,  $\lll$ : rotate left,  $K$ : secret key,  $rk$ : round key,  $k$ : secret key size,  $w$ : word size

$$rk[i] = K[i] \wedge (K[i] \lll 1) \wedge (K[i] \lll 8),$$

$$rk[(i+k/w)\wedge 1] = K[i] \wedge (K[i] \lll 1) \wedge (K[i] \lll 11),$$

where  $0 \leq i < k/w$



Thank you for providing information on the key scheduling for the CHAM block cipher. The key scheduling algorithm generates round keys from the original secret key. In the case of CHAM, the secret key is divided into  $k/w$   $w$ -bit words,  $K[0], K[1], \dots, K[k/w-1]$ . The algorithm then generates  $2k/w$   $w$ -bit round keys,  $rk[0], rk[1], \dots, rk[2k/w-1]$ .

Here is the key scheduling algorithm for CHAM, based on the equations you provided:

Notation:

$\wedge$ : XOR

$\lll$ : rotate left

$K$ : secret key

$rk$ : round key

$k$ : secret key size

$w$ : word size

For  $0 \leq i < k/w$ :

- $rk[i] = K[i] \wedge (K[i] \lll 1) \wedge (K[i] \lll 8)$

- $rk[(i+k/w)\wedge 1] = K[i] \wedge (K[i] \lll 1) \wedge (K[i] \lll 11)$

This key scheduling algorithm is relatively simple compared to some other block ciphers, which contributes to the lightweight nature of the CHAM block cipher. The round keys generated by this algorithm are used in the round functions of the cipher to provide security and resistance against various cryptanalytic attacks.

Listing 31: Description of CHAM's Key Scheduling Structure.



If so, can you implement the CHAM block cipher in C language?



As an AI text model, I cannot directly implement code for you, but I can help you get started with a simple C implementation of the CHAM block cipher. Here's an example code snippet for the CHAM-64/128 variant. Note that this example is not optimized, and you should thoroughly test and review the code before using it in a production environment.

```

1 #include <stdint.h>
2 #include <stdio.h>
3
4 #define ROUNDS 88
5 #define KEY_SIZE 8
6 #define BLOCK_SIZE 4
7
8 void cham64_128_encrypt(uint16_t plaintext[BLOCK_SIZE], const
 uint16_t key[KEY_SIZE]) {
9 uint16_t round_keys[KEY_SIZE * 2];
10 for (int i = 0; i < KEY_SIZE; i++) {
11 round_keys[i] = key[i] ^ (key[i] << 1) ^ (key[i] <<
12 8);
13 round_keys[i + KEY_SIZE] = key[i] ^ (key[i] << 1) ^ (
14 key[i] << 11);
15 }
16
17 for (int round = 1; round <= ROUNDS; round++) {
18 uint16_t x = plaintext[0], y = plaintext[1];
19 if (round % 2 == 1) {
20 x ^= round;
21 y = (y << 1) | (y >> 15);
22 } else {
23 x ^= round;
24 y = (y << 8) | (y >> 8);
25 }
26
27 uint16_t temp = x ^ y ^ round_keys[(round - 1) % (
28 KEY_SIZE * 2)];
29 if (round % 2 == 1) {
30 temp = (temp << 8) | (temp >> 8);
31 } else {
32 temp = (temp << 1) | (temp >> 15);
33 }
34
35 plaintext[0] = plaintext[2] ^ temp;
36 plaintext[1] = plaintext[3];
37
38 // Rotate words

```

```

36 uint16_t temp_word = plaintext[0];
37 plaintext[0] = plaintext[1];
38 plaintext[1] = plaintext[2];
39 plaintext[2] = plaintext[3];
40 plaintext[3] = temp_word;
41 }
42 }
43
44 int main() {
45 uint16_t key[KEY_SIZE] = {0x0123, 0x4567, 0x89AB, 0xCDEF,
46 0xFEDC, 0xBA98, 0x7654, 0x3210};
47 uint16_t plaintext[BLOCK_SIZE] = {0x0123, 0x4567, 0x89AB,
48 0xCDEF};
49
50 cham64_128_encrypt(plaintext, key);
51
52 printf("Encrypted: ");
53 for (int i = 0; i < BLOCK_SIZE; i++) {
54 printf("%04X ", plaintext[i]);
55 }
56 printf("\n");
57 return 0;
58 }

```

This code snippet only implements the encryption function for CHAM-64/128. For a complete implementation, you will need to implement the decryption function as well. Keep in mind that this is just a starting point and can be optimized and improved for better performance and readability.

Listing 32: Request for the first CHAM block cipher implementation.

**HD** There is something slightly wrong. Please fix like this first.

1. Separate the key scheduling function from the encrypt function and write it separately. It should be written in the form of sending the round key obtained after key scheduling in the main function as a parameter to the encrypt function. Therefore, an additional variable called round key must exist in the main function.

2. Modify the variables of the main function as follows.

```
key: 0x0100 0x0302 0x0504 0x0706 0x0908 0x0b0a 0x0d0c 0x0f0e
plaintext: 0x1100 0x3322 0x5544 0x7766
```

And, if you possible, add '\_64128' to the main function's key and plaintext variable names.



I have made the requested changes to the code. The key scheduling function is now separated from the encryption function, and the main function has been modified accordingly. The key and plaintext values have also been updated as per your request. Here's the updated code snippet:

```

1 #include <stdint.h>
2 #include <stdio.h>
3
4 #define ROUNDS 88
5 #define KEY_SIZE 8
6 #define BLOCK_SIZE 4
7
8 void cham64_128_key_schedule(const uint16_t key[KEY_SIZE],
9 uint16_t round_keys[KEY_SIZE * 2]) {
10 for (int i = 0; i < KEY_SIZE; i++) {
11 round_keys[i] = key[i] ^ (key[i] << 1) ^ (key[i] <<
12 8);
13 round_keys[i + KEY_SIZE] = key[i] ^ (key[i] << 1) ^ (
14 key[i] << 11);
15 }
16 }
17
18 void cham64_128_encrypt(uint16_t plaintext[BLOCK_SIZE], const
19 uint16_t round_keys[KEY_SIZE * 2]) {
20 for (int round = 1; round <= ROUNDS; round++) {
21 uint16_t x = plaintext[0], y = plaintext[1];
22 if (round % 2 == 1) {
23 x ^= round;
24 y = (y << 1) | (y >> 15);
25 } else {
26 x ^= round;
27 y = (y << 8) | (y >> 8);
28 }
29
30 uint16_t temp = x ^ y ^ round_keys[(round - 1) % (
31 KEY_SIZE * 2)];
32 if (round % 2 == 1) {
33 temp = (temp << 8) | (temp >> 8);
34 } else {
35 temp = (temp << 1) | (temp >> 15);
36 }
37
38 plaintext[0] = plaintext[2] ^ temp;
39 plaintext[1] = plaintext[3];
40
41 // Rotate words

```

```

37 uint16_t temp_word = plaintext[0];
38 plaintext[0] = plaintext[1];
39 plaintext[1] = plaintext[2];
40 plaintext[2] = plaintext[3];
41 plaintext[3] = temp_word;
42 }
43 }
44
45 int main() {
46 uint16_t key_64128[KEY_SIZE] = {0x0100, 0x0302, 0x0504, 0
47 x0706, 0x0908, 0x0b0a, 0x0d0c, 0x0f0e};
48 uint16_t plaintext_64128[BLOCK_SIZE] = {0x1100, 0x3322, 0
49 x5544, 0x7766};
50 uint16_t round_keys[KEY_SIZE * 2];
51
52 cham64_128_key_schedule(key_64128, round_keys);
53 cham64_128_encrypt(plaintext_64128, round_keys);
54
55 printf("Encrypted: ");
56 for (int i = 0; i < BLOCK_SIZE; i++) {
57 printf("%04X ", plaintext_64128[i]);
58 }
59 printf("\n");
60
61 return 0;
62 }


```


This code snippet now has a separate key scheduling function, `cham64_128_key_schedule`, that generates the round keys. The main function calls this function to generate the round keys and then passes them to the encryption function `cham64_128_encrypt`. The key and plaintext variables in the main function have been updated and their names have been changed to include `'_64128'`.

Listing 33: Separating key scheduling from round function.

**HD** Your key schedule implementation is somewhat wrong. Key schedule need Rotate operation, not Shift operation. `'key[i] << 1'` is 1 time shift operation, it does not performed rotate operation. Also `'round_keys[i + KEY_SIZE]'` is wrong.

Let's consider the key scheduling structure again. Original round key generation is `rk[(i+k/w)^1]`. Explain `k` and `w` in CHAM-64/128. And in your code missed `'^1'` part. Please rewrite the code with this in mind. Also, do not define additional constants. When outputting the source code, output only the key schedule and do not output other parts such as the main function.

 Where is the `'^1'`?

 Apologies for the confusion in my previous response. I misunderstood your question. I have now updated the key schedule function to include the XOR with 1, as specified in the CHAM-64/128 cipher:


```

1 void cham64_128_key_schedule(const uint16_t key[KEY_SIZE],
2 uint16_t round_keys[KEY_SIZE * 2]) {
3 for (int i = 0; i < KEY_SIZE; i++) {
4 round_keys[i] = key[i] ^ ((key[i] << 1) | (key[i] >>
5 15)) ^ ((key[i] << 8) | (key[i] >> 8));
6 round_keys[(i + KEY_SIZE) ^ 1] = key[i] ^ ((key[i] <<
7 1) | (key[i] >> 15)) ^ ((key[i] << 11) | (key[i] >> 5));
8 }
9 }


```

The line `round_keys[(i + KEY_SIZE) ^ 1]` now includes the XOR with 1 for the round key generation, as specified in the CHAM-64/128 cipher key scheduling.

Listing 34: Key Scheduling Modification Request. ChatGPT implements Rotate as Shift, and the `'^1'` operation is missing in the index value. Requested to amend this twice. (ChatGPT's intermediate answer is omitted.)

 Great. Your key schedule is perfect.

Now let's modify the round function. First of all, the round constant value is fixed to start from 0 instead of 1. Print the modified code, but no other functions (e.g. `key schedule`, `main`).


 Great. Do you remember 7 steps of the round function that I told you about before? Up to step 4 is implemented correctly now. In other words, steps 5, 6, and 7 are slightly different. Please fix the source code correctly considering this.

**HD** `uint16_t temp = x ^ y ^ round_keys[round % (KEY_SIZE * 2)];` part is wrong. Addition operation is required with `x`, not XOR operation.

**HD** great. However, if you write the code like this, `+` will be calculated before `^`. Add `()` so that `^` can be computed first.

**HD** Step 5, 6 are wrong. `uint16_t z = plaintext[2] + temp;` is not step 5, but `uint16_t temp = x + (y ^ round_keys[round % (KEY_SIZE * 2)]);` already has step 5 operation. Step 6 also wrong, because the `if (round % 2 == 0) temp ...` statement is real step 6. So remove them.

**HD** Finally, instead of `plaintext[3] ^ temp`, just store `temp` in `plaintext[0]`. Please correct this part.

 I apologize for the confusion. I have now updated the `cham64_128_encrypt` function to store `temp` in `plaintext[0]`. Here's the revised function:

```

1 void cham64_128_encrypt(uint16_t plaintext[BLOCK_SIZE], const
 uint16_t round_keys[KEY_SIZE * 2]) {
2 for (int round = 0; round < ROUNDS; round++) {
3 uint16_t x = plaintext[0], y = plaintext[1];
4
5 if (round % 2 == 0) {
6 x ^= round;
7 y = (y << 1) | (y >> 15);
8 } else {
9 x ^= round;
10 y = (y << 8) | (y >> 8);
11 }
12
13 uint16_t temp = x + (y ^ round_keys[round % (KEY_SIZE
 * 2)]);
14
15 if (round % 2 == 0) {
16 temp = (temp << 8) | (temp >> 8);
17 } else {
18 temp = (temp << 1) | (temp >> 15);
19 }
20
21 plaintext[0] = temp;
22
23 // Step 7: Rotate words
24 uint16_t temp_word = plaintext[0];
25 plaintext[0] = plaintext[1];
26 plaintext[1] = plaintext[2];

```

```

27 plaintext[2] = plaintext[3];
28 plaintext[3] = temp_word;
29 }
30 }

```

Now the function stores temp in plaintext[0] as requested.

Listing 35: Request to modify the round function. ChatGPT suffered a lot in generating a round function. It went through 6 revisions, and fixed the following contents. 1. Round constant starts from 1, 2. Useless operation added (fix 3), 3. XOR operation instead of + operation, 4. Parentheses added to make the operation order correct. (ChatGPT's intermediate answers are omitted.)

**HD** Your code works perfectly. Combine the code you have written so far and print it again.

When integrating code, be sure to integrate using modified code.

Also, please apply the following fixes:

1. Adding the name `_64_128` after defined constant values.
2. Change the main function variable name `_64128` to `_64_128` format.

**HD** The key scheduling part is the initial version code. We have modified the key scheduling code. Please replace with the appropriate code.

**HD** You again did not write 'XOR 1' in the index part of `round_keys[i + KEY_SIZE_64_128]`. Correct the index part correctly.

**HD** very good. But I don't like the structure of storing the ciphertext in a plaintext variable. So, declare a new variable to store the ciphertext and change the structure of the function to store it separately.

**HD** Everything is perfect. Now I want to see other variables at console output. Please modify a code of `main function`. A printing order is `secret key`, `plaintext`, `round key`, and `ciphertext`. No need to show other functions (e.g. `key schedule`, `round function`), just let me know about `main function`.

**HD** Then combine this `main function` with the code you wrote earlier and show it.



Here's the complete code with the updated `main function` combined with the key schedule and encryption functions:

```

1 #include <stdio.h>
2 #include <stdint.h>
3

```



```


4 #define BLOCK_SIZE_64_128 4
5 #define KEY_SIZE_64_128 8
6 #define ROUNDS_64_128 88
7
8 void cham64_128_key_schedule(const uint16_t key[
 KEY_SIZE_64_128], uint16_t round_keys[KEY_SIZE_64_128 *
 2]) {
9 for (int i = 0; i < KEY_SIZE_64_128; i++) {
10 round_keys[i] = key[i] ^ (key[i] << 1 | key[i] >> 15)
 ^ (key[i] << 8 | key[i] >> 8);
11 round_keys[(i + KEY_SIZE_64_128) ^ 1] = key[i] ^ (key
 [i] << 1 | key[i] >> 15) ^ (key[i] << 11 | key[i] >> 5);
12 }
13 }
14
15 void cham64_128_encrypt(const uint16_t plaintext[
 BLOCK_SIZE_64_128], uint16_t ciphertext[BLOCK_SIZE_64_128
], const uint16_t round_keys[KEY_SIZE_64_128 * 2]) {
16 for (int i = 0; i < BLOCK_SIZE_64_128; i++) {
17 ciphertext[i] = plaintext[i];
18 }
19
20 for (int round = 0; round < ROUNDS_64_128; round++) {
21 uint16_t x = ciphertext[0], y = ciphertext[1];
22
23 if (round % 2 == 0) {
24 x ^= round;
25 y = (y << 1) | (y >> 15);
26 } else {
27 x ^= round;
28 y = (y << 8) | (y >> 8);
29 }
30
31 uint16_t temp = x + (y ^ round_keys[round % (
 KEY_SIZE_64_128 * 2)]);
32
33 if (round % 2 == 0) {
34 temp = (temp << 8) | (temp >> 8);
35 } else {
36 temp = (temp << 1) | (temp >> 15);
37 }
38
39 ciphertext[0] = temp;
40
41 // Step 7: Rotate words
42 uint16_t temp_word = ciphertext[0];
43 ciphertext[0] = ciphertext[1];
44 ciphertext[1] = ciphertext[2];
45 ciphertext[2] = ciphertext[3];
46 ciphertext[3] = temp_word;


```

```


47 }
48 }
49
50 int main() {
51 uint16_t key_64_128[KEY_SIZE_64_128] = {0x0100, 0x0302, 0
52 x0504, 0x0706, 0x0908, 0x0b0a, 0x0d0c, 0x0f0e};
53 uint16_t plaintext_64_128[BLOCK_SIZE_64_128] = {0x1100, 0
54 x3322, 0x5544, 0x7766};
55 uint16_t ciphertext_64_128[BLOCK_SIZE_64_128];
56 uint16_t round_keys_64_128[KEY_SIZE_64_128 * 2];
57
58 cham64_128_key_schedule(key_64_128, round_keys_64_128);
59 cham64_128_encrypt(plaintext_64_128, ciphertext_64_128,
60 round_keys_64_128);
61
62 printf("Secret Key:\n");
63 for (int i = 0; i < KEY_SIZE_64
64 printf("Secret Key:\n");
65 for (int i = 0; i < KEY_SIZE_64_128; i++) {
66 printf("%04x ", key_64_128[i]);
67 }
68 printf("\n");
69
70 printf("Plaintext:\n");
71 for (int i = 0; i < BLOCK_SIZE_64_128; i++) {
72 printf("%04x ", plaintext_64_128[i]);
73 }
74 printf("\n");
75
76 printf("Round Keys:\n");
77 for (int i = 0; i < KEY_SIZE_64_128 * 2; i++) {
78 printf("%04x ", round_keys_64_128[i]);
79 if (i % KEY_SIZE_64_128 == KEY_SIZE_64_128 - 1) {
80 printf("\n");
81 }
82 }
83
84 printf("Encrypted Plaintext (Ciphertext):\n");
85 for (int i = 0; i < BLOCK_SIZE_64_128; i++) {
86 printf("%04x ", ciphertext_64_128[i]);
87 }
88 printf("\n");
89
90 return 0;
91 }


```

 Great job. We'll name this the CHAM-64/128 source code. Now the function stores temp in `plaintext[0]` as requested.

 Thank you! I'm glad you found the CHAM-64/128 source code helpful. If you have any further questions or need assistance with other ciphers or programming tasks, feel free to ask.

Listing 36: Request to complete the CHAM-64/128 source code. In the process of integrating the source code, a problem was found using the code before modification. (ChatGPT's intermediate answers are omitted.)

 Very good. We'll name this the CHAM-128/128 source code. It is consist with `cham128_128_key_schedule`, `cham128_128_encrypt`, some of macro variable and main function.

 Great! The final CHAM-128/128 source code consists of the `cham128_128_key_schedule` function, `cham128_128_encrypt` function, macro variables, and the main function. Here's the complete source code:

```

1 #include <stdio.h>
2 #include <stdint.h>
3
4 #define KEY_SIZE_128_128 4
5 #define BLOCK_SIZE_128_128 4
6 #define ROUNDS_128_128 112
7
8 void cham128_128_key_schedule(const uint32_t *key, uint32_t *
 round_keys) {
9 for (int i = 0; i < KEY_SIZE_128_128; i++) {
10 round_keys[i] = key[i] ^ ((key[i] << 1) | (key[i] >>
 (32 - 1))) ^ ((key[i] << 8) | (key[i] >> (32 - 8)));
11 round_keys[(i + KEY_SIZE_128_128) ^ 1] = key[i] ^ ((
 key[i] << 1) | (key[i] >> (32 - 1))) ^ ((key[i] << 11) |
 (key[i] >> (32 - 11)));
12 }
13 }
14
15 void cham128_128_encrypt(const uint32_t plaintext[
 BLOCK_SIZE_128_128], uint32_t ciphertext[
 BLOCK_SIZE_128_128], const uint32_t round_keys[
 KEY_SIZE_128_128 * 2]) {
16 for (int i = 0; i < BLOCK_SIZE_128_128; i++) {
17 ciphertext[i] = plaintext[i];

```

```

18 }
19
20 for (int round = 0; round < ROUNDS_128_128; round++) {
21 uint32_t x = ciphertext[0], y = ciphertext[1];
22
23 if (round % 2 == 0) {
24 x ^= round;
25 y = (y << 1) | (y >> 31);
26 } else {
27 x ^= round;
28 y = (y << 8) | (y >> 24);
29 }
30
31 uint32_t temp = x + (y ^ round_keys[round % (
KEY_SIZE_128_128 * 2)]);
32
33 if (round % 2 == 0) {
34 temp = (temp << 8) | (temp >> 24);
35 } else {
36 temp = (temp << 1) | (temp >> 31);
37 }
38
39 ciphertext[0] = temp;
40
41 // Step 7: Rotate words
42 uint32_t temp_word = ciphertext[0];
43 ciphertext[0] = ciphertext[1];
44 ciphertext[1] = ciphertext[2];
45 ciphertext[2] = ciphertext[3];
46 ciphertext[3] = temp_word;
47 }
48 }
49
50 int main() {
51 uint32_t key_128_128[KEY_SIZE_128_128] = {0x03020100, 0
x07060504, 0x0b0a0908, 0x0f0e0d0c};
52 uint32_t plaintext_128_128[BLOCK_SIZE_128_128] = {0
x33221100, 0x77665544, 0xbbaa9988, 0xffeeddcc};
53 uint32_t round_keys_128_128[KEY_SIZE_128_128 * 2];
54 uint32_t ciphertext_128_128[BLOCK_SIZE_128_128];
55
56 cham128_128_key_schedule(key_128_128, round_keys_128_128)
;
57 cham128_128_encrypt(plaintext_128_128, ciphertext_128_128
, round_keys_128_128);
58
59 printf("Secret Key: ");
60 for (int i = 0; i < KEY_SIZE_128_128; i++) {
61 printf("%08x ", key_128_128[i]);
62 }

```

```

63 printf("\n");
64
65 printf("Plaintext: ");
66 for (int i = 0; i < BLOCK_SIZE_128_128; i++) {
67 printf("%08x ", plaintext_128_128[i]);
68 }
69 printf("\n");
70
71 printf("Round Keys: ");
72 for (int i = 0; i < KEY_SIZE_128_128 * 2; i++) {
73 printf("%08x ", round_keys_128_128[i]);
74 }
75 printf("\n");
76
77 printf("Ciphertext: ");
78 for (int i = 0; i < BLOCK_SIZE_128_128; i++) {
79 printf("%08x ", ciphertext_128_128[i]);
80 }
81 printf("\n");
82
83 return 0;
84 }

```



This completes the CHAM-128/128 source code with the corrected number of rounds (112 rounds).

Listing 37: Porting CHAM-64/128 to CHAM-128/128. ChatGPT made errors in implementing key scheduling and setting round constants, but succeeded in the end. (Some dialogue omitted.)



Here is a list of the exact parameters of CHAM-128/256. Please refer to this and modify the source code.

CHAM-128/256:

- Block size (**n**): 128 bits
- Secret key size (**k**): 256 bits
- Word size (**w**): 32 bits
- Number of rounds (**r**): 120



Thank you for providing the correct parameters for CHAM-128/256. I'll make the appropriate changes to the source code:

```

1 #include <stdio.h>

```

```

2 #include <stdint.h>
3
4 #define BLOCK_SIZE_128_256 4
5 #define KEY_SIZE_128_256 8
6 #define ROUNDS_128_256 120
7
8 void cham128_256_key_schedule(const uint32_t *key, uint32_t *
 round_keys) {
9 for (int i = 0; i < KEY_SIZE_128_256; i++) {
10 round_keys[i] = key[i] ^ ((key[i] << 1) | (key[i] >>
 (32 - 1))) ^ ((key[i] << 8) | (key[i] >> (32 - 8)));
11 round_keys[(i + KEY_SIZE_128_256) ^ 1] = key[i] ^ ((
 key[i] << 1) | (key[i] >> (32 - 1))) ^ ((key[i] << 11) |
 (key[i] >> (32 - 11)));
12 }
13 }
14
15 void cham128_256_encrypt(const uint32_t plaintext[
 BLOCK_SIZE_128_256], uint32_t ciphertext[
 BLOCK_SIZE_128_256], const uint32_t round_keys[
 KEY_SIZE_128_256 * 2]) {
16 for (int i = 0; i < BLOCK_SIZE_128_256; i++) {
17 ciphertext[i] = plaintext[i];
18 }
19
20 for (int round = 0; round < ROUNDS_128_256; round++) {
21 uint32_t x = ciphertext[0], y = ciphertext[1];
22
23 if (round % 2 == 0) {
24 x ^= round;
25 y = (y << 1) | (y >> 31);
26 } else {
27 x ^= round;
28 y = (y << 8) | (y >> 24);
29 }
30
31 uint32_t temp = x + (y ^ round_keys[round % (
 KEY_SIZE_128_256 * 2)]);
32
33 if (round % 2 == 0) {
34 temp = (temp << 8) | (temp >> 24);
35 } else {
36 temp = (temp << 1) | (temp >> 31);
37 }
38
39 ciphertext[0] = temp;
40
41 // Step 7: Rotate words
42 uint32_t temp_word = ciphertext[0];
43 ciphertext[0] = ciphertext[1];

```

```

44 ciphertext[1] = ciphertext[2];
45 ciphertext[2] = ciphertext[3];
46 ciphertext[3] = temp_word;
47 }
48 }
49
50 int main() {
51 uint32_t secret_key_128_256[KEY_SIZE_128_256] = {0
x03020100, 0x07060504, 0x0b0a0908, 0x0f0e0d0c, 0xf3f2f1f0
, 0xf7f6f5f4, 0xfbfa9f8, 0xffffdfdc};
52 uint32_t plaintext_128_256[BLOCK_SIZE_128_256] = {0
x33221100, 0x77665544, 0xbbaa9988, 0xffeeddcc};
53 uint32_t round_keys_128_256[KEY_SIZE_128_256 * 2];
54 uint32_t ciphertext_128_256[BLOCK_SIZE_128_256];
55
56 cham128_256_key_schedule(secret_key_128_256 ,
round_keys_128_256);
57 cham128_256_encrypt(plaintext_128_256 , ciphertext_128_256
, round_keys_128_256);
58
59 printf("Secret Key: ");
60 for (int i = 0; i < KEY_SIZE_128_256; i++) {
61 printf("%08x ", secret_key_128_256[i]);
62 }
63 printf("\n");
64
65 printf("Plaintext: ");
66 for (int i = 0; i < BLOCK_SIZE_128_256; i++) {
67 printf("%08x ", plaintext_128_256[i]);
68 }
69 printf("\n");
70
71 printf("Round Keys: ");
72 for (int i = 0; i < KEY_SIZE_128_256 * 2; i++) {
73 printf("%08x ", round_keys_128_256[i]);
74 }
75 printf("\n");
76
77 printf("Ciphertext: ");
78 for (int i = 0; i < BLOCK_SIZE_128_256; i++) {
79 printf("%08x ", ciphertext_128_256[i]);
80 }
81 printf("\n");
82
83 return 0;
84 }

```

Listing 38: Adapting CHAM-128/128 to CHAM-128/256, it was discovered that ChatGPT had forgotten the parameters for CHAM-128/256. By rectifying this issue, the porting process can proceed smoothly. (Some dialogue omitted.)