COMP 2401 B/D

Assignment #2: The Walkie-Talkie Conundrum

Bits and Bytes

Goals:

In this assignment you will be working with encryption and decryption using a secret key. The algorithm that you implement will use an incrementing counter so that each character is encrypted in a slightly different way. This is a simplified version of the Counter (CTR) mode of the Advanced Encryption Standard (AES) symmetric encryption algorithm using a chaining approach similar to the Cipher block chaining (CBC) module. While this is handling encryption using user input, the long-term application would be to have this happen automatically between a server and client (specifically, communication between devices).

Learning Outcomes:

If you are keeping up with the lectures, completing tutorials, and read the course notes as needed, then by the end of this assignment you will have demonstrated that you can:

- Write a program in C which is designed with modular functions and correctly documented
- Organize a C program, utilizing existing functions as much as possible
- Use bit masks and bitwise operators to manipulate values at the bit level

Assignment Context (Optional Read):

Great work on your last job for the Carleton University Ghost Hunters Society! Those old devices are hard to use because they have no real way of communicating externally. With all the money we've saved not having to write everything by hand, we were able to pick up some new communications equipment! The problem is, it uh, doesn't work with our current setup.

These new Walkie-Talkies are perfect for both quiet communication *and* picking up ghost noises. They have two great features: It converts voices (ghostly or otherwise) into text, and it can transmit that text over a radio frequency. We want to use them to detect noises in ghost-filled rooms and get a transcript sent to our van so that it can flag any words like "Get out", "Run", or "Ahhhhhhhhh!".

The problem? It's using some weird encryption - a variant of AES, and our van just receives a bunch of garbage data. We want ghost voices, not nonsense! Can you write a program that lets us encrypt/decrypt messages from the walkie-talkies? I… *think*… we have a data specification somewhere… We'll send you some sample data from a walkie left in a room overnight to test it.

Instructions:

While we want to eventually be able to encrypt/decrypt data over a network, for this assignment, we will simply work with user input - no files or networking needed. Your program will prompt the user to indicate whether they wish to encrypt a readable message (the plaintext), or decrypt an already encrypted sequence of numbers (the ciphertext) back to its readable form. You will implement the encryption algorithm described below, and you will need to deduce and implement the corresponding decryption algorithm.

Your program must follow the programming conventions that have been discussed throughout the lectures, for example: Separate your code into small, modular functions; Perform some basic error checking, and avoid making assumptions about the state of parameters (error check and validate inputs), document each function with a function block and comment any complex or confusing code, make use of definitions to avoid 'magic numbers', and choose the correct data types to best match the data being used (if not specified). Functions should be really good at *one thing* and avoid side effects.

Note: This assignment requires more problem solving and some extra research on certain functions than the previous assignment. As such, please do not provide assistance to other students with the design of code or sharing problem solving steps for completing the assignment.

Approach for Completing:

This specification will cover each section more carefully, but here is the overall plan for approaching this assignment:

1. Take time to understand the encryption algorithm; potentially, try working with it on paper for a few values to ensure you understand how the values change
2. Take some time to consider the main function control flow, as it might take some research and experimentation, and more time than you expect:
    a. It has to accept space-separated plaintext
    b. It has to accept an arbitrary number of numerical decimal values, terminated by -1
    c. More information is available later in the specification
3. Implement the encryption functionality.
4. Implement some of the main control flow to allow for encryption, so that you can test some values out
5. Implement the decryption functionality. Try encrypting/decrypting a few times to validate it.
6. Finish implementing the control flow to accept input for the encryption/decryption functions.
7. Clean up the code and verify that it matches the assignment requirements.
8. Write your README file, package everything in a tar or zip file, submit, download, and compile & run on the course VM *one more time* to make sure everything is working.

Provided Code:
You have been provided with some code, *a2-posted.c* which contains:

- Definitions for the maximum text buffer size, an initial value for the encryption (IV), key, and counter
- Forward definitions for necessary functions
    - You may need to create some additional helper functions to have clean code
- Some of the main control flow
    - It utilizes a new function, 'fgets()' which is used to get input from files; however, we can pass it "stdin" and treat out user input as a "file". This allows us to get text with spaces, but it also tends to include the newline character…
    - It uses 'sscanf()', which takes a string and breaks it up into separate variables (just like scanf, but with a string as input)

Encryption Overview:

The encryption algorithm relies on keys and counter values to properly encrypt and decrypt. In order to correctly encrypt/decrypt, the processes should have identical keys/counters which are initialized at the beginning of the program - that is, if we have a different key, or a different counter, while encrypting than decrypting, we'll get garbage data back.

The key value of this particular algorithm is 0b11001011 and the initial counter value is set to 0b00110101, but this initial counter value is only that - the initial value.

The encryption algorithm processes each byte of the plaintext input, one byte at a time, and with each byte it will update the counter value. Each step of the encryption process requires the encrypted value of the *previous* byte; since the first byte to encrypt has no previous byte, we will use the initial value (IV) set to 0b10110001 in the provided definitions.

Definitions:

Plaintext: The human readable text that we would like to encrypt.
Ciphertext: Text that has been encrypted.
Mirror Bit: Let's define two "mirror bit" positions inside a byte as two bit positions that are the same as each other, relative to each end of the byte. For example, bit positions 0 and 7 are mirror bits, because they are both in the first position at either end of the byte. Bits 1 and 6 are also mirror positions (second position from either end of the byte), as are bits 2 and 5, and bits 3 and 4. This definition will help us with the algorithm description.

Encryption Algorithm:

To encode the entire plaintext, then for each byte within the plaintext:

1. Process the counter value, using the key, as follows:
    a. Make a copy of the counter value into a temporary counter
    b. Compute the starting bit for the loop below: if the counter is an even number, the loop begins at bit position 0; if the counter is odd, it begins at bit position 1
    c. Loop over "every other" (i.e. alternating, skip 1, **a**b**c**d**e**f**g**) bit positions, starting at the bit position computed in (1b); For example: If the loop begins at bit position 0, it will iterate over all the even bit positions. For each bit processed:
        i. Perform an xor operation between the current bits of the counter and of the key
        ii. Set the current bit of the temp counter to the result of this xor operation
    d. Return the temporary counter value as the *updated* counter
2. We can now encrypt the plaintext byte using this updated counter and the previous byte of the encrypted ciphertext (or, if it is the first byte of our plaintext, we use the IV). We will encrypt the byte as follows:
    a. Initialize a temp byte to zero
    b. Loop over every bit position, starting at bit 0; For each bit:
        i. If the current bit of the counter is 1, perform an xor operation between the current bit of the plaintext and the *current bit* of the previous ciphertext byte
        ii. If the current bit of the counter is 0, perform an xor operation between the current bit of the plaintext and the *mirror bit* of the previous ciphertext byte
        iii. Set the current bit of the temp byte to the result of the xor operation above
    c. Return the temp byte value as the ciphertext byte
3. Increment the updated counter value by 1

Instructions:

Once you have tried to experiment with the encryption algorithm on paper for a few values so that you understand the process, you can start to implement the program. Begin with the starting code, *a2-posted.c* which is available on Brightspace. Remember that you *can not modify* the provided code, except where necessary for the program.

1. Encryption Functionality

1.1. Implement the `getBit()`, `setBit()`, and `clearBit()` functions as shown in class; you *must* use these functions anywhere that bitwise operations are being performed.

1.2. Implement the `unsigned char processCtr(unsigned char ctr, unsigned char key)` function that processes the given counter value using the given key, as described in Step 1 of the encryption algorithm above.

1.3. Implement the `unsigned char encryptByte(unsigned char pt, unsigned char ctr, unsigned char prev)` function that encrypts the given plaintext byte `pt`, using the counter value `ctr` and the previous byte of the ciphertext `prev`, as described in step (2) of the encryption algorithm found above. This function returns the corresponding encrypted ciphertext byte as the return value.

1.4. Implement the `void encode(unsigned char *pt, unsigned char* ct, int numBytes)` function that takes an array of plaintext characters stored in parameter `pt`, which contains `numBytes` bytes. The function encrypts each plaintext character into its corresponding ciphertext byte, as described in the encryption algorithm, and stores the encrypted byte into the ciphertext array `ct`.

2. Decryption Functionality

Write/design the functions required for the decryption algorithm. The decryption **processes the counter identically to the encryption**, but the decryption of each byte using the updated counter must perform the xor operation between different values than the encryption did.

Note: The xor operator (^) has a useful property. Assume that `a` and `b` are any numeric value, and that `a^b` results in `c`… then it's *also* true that `b^c` results in `a`, and `a^c` results in `b`! So your decryption algorithm will not need to "reverse" the xor operation performed during encryption, but instead performs the xor *between different values* than the encryption did.

2.1. Implement the `unsigned char decryptByte(unsigned char ct, unsigned char ctr, unsigned char prev)` function that decrypts the given ciphertext byte `ct`, using the counter value `ctr`, and the previous byte of ciphertext `prev`. This function returns the corresponding decrypted plaintext byte as the return value.

2.2. Implement the `void decode (unsigned char *ct, unsigned char *pt, int numBytes)` function that takes an array of ciphertext bytes stored in parameter `ct`, which contains `numBytes` bytes. The function decrypts each ciphertext byte into its corresponding plaintext byte, and stores the decrypted byte into the plaintext array `pt`.

3. Main Control Flow

3.1. Implement the main function to first prompt the user for whether they want to encrypt a string, or decrypt a sequence of numbers

3.2. If the user chooses to encrypt a string, your program does the following:

3.2.1. Read a plaintext string from the command line.

3.2.1.1. Because this string will contain multiple space-delimited words, you cannot use the `scanf()` library function for this; instead, use `fgets()` as shown in the provided code - you may need to do some research for how to use this function for string input, but you can also experiment with the provided code.

3.2.2. Encrypt the plaintext entered by the user, as described by the encryption algorithm

3.2.3. Print out, to the screen, the space-separated ciphertext bytes, as decimal numeric values.

3.3. If the user chooses to *decrypt* an encrypted sequence, your program does the following:

3.3.1. Read a space-separated sequence of decimal, numeric values from the command line, until reaching a sentinel value of -1.

3.3.1.1. The `scanf()` function is the best choice here, as a single call to `scanf()` reads a single value up to a delimiter, which can be either a space or a newline character.

3.3.1.2. Recall when we tried putting space-separated inputs to `scanf("%d")`, then when we called `scanf()` a second time it pulled the values out of the buffer and didn't let us type? I wonder if this is behaviour we can use here…

3.3.1.3. Be careful with variable types!

3.3.2. Decrypt the sequence entered by the user

3.3.3. Print out, to the screen, the resulting plaintext as a string of ASCII characters.

Testing your Program

First, note that a user should be able to decrypt any ciphertext that was produced by the algorithm, and produce the original string as plaintext. This is one thing that should help with testing.
Importantly: You must test your program with different keys and initial values; as long as they are the same while encoding and decoding, it should work correctly.

Your "Boss" has provided you with some sample data that you can try decrypting. Don't spoil it for anyone!

The Boss: Hey there! So we had a great ghost hunt at the abandoned corner store last night, and we left a walkie-talkie on logging mode there to catch any ghost sounds that might have popped up. There was a LOT of paranormal activity there! We can't decode the messages with our current system though, so it's just a bunch of random numbers.

Can you try decoding this message for us? I want to know what ghosts say when we aren't around!

153 212 157 219 189 209 164 192 224 213 137 228 129 226 174 197 203 243 182 137 240 206 250 137 246 147 254 222 176 209 163 133 246 78 5 192 114 35 231 143 154 106 35 161 237 142 25 237 141 255 150 107 155 237 157 250 31 237 209 178 20 119 215 164 71 130 112 111 147 225 167 156 28 74 35 175 195 172 65 163 237 216 49 218 63 142 52 4 80 114 27 160 0 40 97 214 48 106 58 125 89 123 12 76 89 62 29 85 20 30 90 29 85 36 87 59 8 56 83 49 1 114 17 116 89 47 7 108 66 37 11 113 30 29 25 106 50 51 106 46 66 98 43 44 95 40 20 123 18 52 76 62 74 42 67 46 10 104 8 103 71 154 53 216 56 111 139 240 108 10 98 42 35 243 178 48 100 14 51 221 189 219 180 67 227 138 35 194 42 69 161 137 196 87 141 211 167 128 100 6 117 207 134 1 226 46 7 203 193 -1

Requirements:

Make sure that you follow these requirements for the whole program to get full marks.

1. Document your program as demonstrated in class. A formatting guide *may* be posted to Brightspace to assist with this, and you will be expected to follow this if it is. Minimally but not all encompassing, your documentation should include:
    a. A description of the purpose and process of the function
    b. The role for each parameter (in, out, in/out)
    c. Possible return values
2. Include a README file; a plaintext file named README or README.txt or README.md which contains the following information:
    a. A preamble: The author, student ID, description of the program, and a list of files
    b. Instructions for compilation and launching, including command line arguments
3. Submit all files in a single .tar or .zip file
4. Ensure the code compiles and executes on the course VM following the directions that you provide in the README
5. Functions will only receive full marks if they are implemented, called in the program, and not commented out (this will be assumed to be rough, discarded work)
6. All data should be printed to the screen as required to demonstrate functioning code
7. The code should follow good coding practices:
    a. Functions are modular, reusable, documented, and reused where possible
    b. There are no global variables
    c. Constants (defines) are used where applicable
    d. Return values indicate error state, unless otherwise specified
    e. Helper functions are used when it's consistent with the program design
8. Only use techniques and tools used during the course (tutorials, notes, lectures)

Grading:

While specifics of the grading scheme may change, the following is an approximation of the weights of each section:

- 19 marks: correct design and implementation of main functionality
- 39 marks: correct design and implementation of encryption functionality
- 29 marks: correct design and implementation of decryption functionality
- 4 marks: correct packaging
- 4 marks: correct documentation
- 10 marks: correct program execution

Policy Reminders:

Please recall that all course materials are copyright and you may not share or distribute the materials to anyone outside of the class for any purpose.

Collaboration of any kind outside of general material discussion is strictly prohibited.

Late submissions will not be accepted, unless an announcement is made which overrides this policy.

Submit early, submit often. Technical issues at the deadline do not warrant an extension.
You are expected to submit your work periodically to ensure at least part marks for work done prior to the deadlines.

For support or clarifications, attend TA office hours or post to the course Brightspace discussion forums.

*You have **one week** after you receive your grades back to contact the TA that graded your assignment to dispute your grades, after which it will not be considered. Disputes are only valid if the TA clearly graded incorrectly; disagreeing with the grading scheme is not a valid dispute.*