CS 419 Project 1

Due Friday, February 16, 2022 11:59pm

Introduction

This assignment contains three parts:

1. In part 1, you will implement a block-oriented, pad-free version of a columnar transposition cipher that works on arbitrary binary data.

2. In part 2, you will implement a stream cipher that uses a linear congruential keystream generator and a hashed key as a seed.

3. In part 3, you will modify the cipher in part 2 to operate as a block cipher that will shuffle pairs of bytes within each block based on the keystream and will apply cipher block chaining (CBC) between blocks.

Part 1: Pad-free Columnar Transposition Cipher

We discussed transposition ciphers in class. The oldest known form was the scytale, which was used by the ancient Greeks and Spartans and involved winding a narrow strip of paper with a message written on it around a rod of a specific thickness. The characters of the message are then read horizontally, thus transposing the characters.

This technique can be recreated in the form of a table. The characters of a message are written horizontally across the rows of a table of a specific width and read vertically by columns. The secret is knowing the width of the table. For example, the text `BRAVEHEART` can be written in a table that's four characters wide as:

B	R	A	V
E	H	E	A
R	T		

We can read the characters vertically: `BER`, `RHT`, … Since we have empty spaces in the table, we filled them with padding – data that will be ignored by the reader but allow us to complete the table:

B	R	A	V
E	H	E	A
R	T	X	X

Making our ciphertext `BERRHTAEXVAX`.

Removing the pad

While adding padding ensures that we have a character for every position in the table, it is not strictly necessary. We can still read the characters vertically in the first version of the table and skip over empty cells to get: `BERRHTAEVA`. To reconstruct the data, the reader will know that there was only one way that the 10 characters could have been positioned in the table since they are always written out in a horizontal raster, starting from the first row, and filling each row before moving on to the next row. Hence, padding is not strictly necessary for this cipher and we can dispense with it, ensuring that the length of our ciphertext will be exactly the length of the plaintext with no extra data.

Supporting unlimited length

A feature of this type of cipher is that the table can become arbitrarily long. You need to have the entire message before deciphering any part of it, which does not work well for network streams. It also imposes a burden on storage. If you have 10 GB of content, you need to create a 10 GB buffer.

We can modify the cipher to encrypt and later decrypt data in blocks. Each block is a table of *r* rows and *c* columns. For example, suppose we use a 4x3 (4 columns, 3 rows) table. If we encrypting "`The pink cat sings on Jupiter`" results in three tables, the last of which is partially populated (I've replaced spaces with underscores just for clarity to differentiate the cells from true empty cells.

T	h	e	_
p	i	n	k
_	c	a	t

_	s	i	n
g	s	_	o
n	_	J	u

p	i	t	e
r			

The resulting ciphertext will be the concatenation of the cipehrtext from each of the tables:

```
    Tp hicena kt gnss I Jnouprite
```

Try it

The reference programs provided include a program called `table`. It enables you to specify a table size in rows and columns and one or more optional files. All output gets sent to the standard output (so redirect it if using non-printable contents):

```
    table [-u] [-r rows] [-c cols] [file ...]
```

The `-u` flag decrypts content. The default row and column values are 1.
For example:

```
    $ echo -n "The pink cat sings on Jupiter"|./table -r3 -c4;echo
    Tp hicena kt gnss i Jnouprite

    $ echo -n "Tp hicena kt gnss i Jnouprite"|./table -u -r3 -c4;echo
    The pink cat sings on Jupiter
```

Note that the first echo command has the -n flag so that the terminating newline doesn't become part of the message. The echo command at the end just produces a newline after the ciphertext, so your command prompt won't show up immediately after this. None of these are considerations if you're reading from a file or redirecting the output to a file.

Columnar transposition

Now we know we can encrypt arbitrary data without having to buffer it all in memory or wait for it all to arrive. The cipher is still incredibly weak because all the attacker needs to know is the table size. A columnar transposition cipher adds a *key* to the cipher. The key in this case is an alphanumeric string (although there's no reason it can't be arbitrary data) that defines with width of the table *and* defines the sequence in which columns are read from the table.

If we go to our first example of encoding BRAVEHEART, suppose we have the key FATE. That tells us that the width of the table is 4 characters and the sequence in which we read the characters will be { 3, 1, 4, 2 } since that reflects the sorting order of the characters in the key. Thus, we have:

3	1	4	2
B	R	A	V
E	H	E	A
R	T		

which produces the ciphertext:

```
RHTVABERAE
```

Putting it all together

Your program will create a binary version of a columnar transposition cipher that will read data in chunks of a specified block size and use a user-provided key to guide the transposition. The usage of the programs you write will be:

```
ctencrypt [-b blocksize] -k key [plaintextfile]

ctdecrypt [-b blocksize] -k key [ciphertextfile]
```

In both programs:

- Bracketed content is optional
- If an input file is not specified, the program will read from the standard input.
- Results are printed to the standard output.
- If you insist on printing any extraneous data, print it to the standard error stream so it will not mess up the resulting output.

- Assume that the content can be arbitrary binary data. Do not assume you're reading printable characters and do not write any extra characters (such as newlines or spaces) to your output.
- Do not make assumptions about the size of the content that will be encrypted. Your program should handle gigabytes of content and not be designed to buffer all contents in memory.
- The key is a string. Its length specifies the width of the table. Its content specifies the sequence in which columns are read from the table when encrypting. If multiple characters are repeated, they are sorted from left to right. For example, the key "`abba`" will result in the encryption reading the 1st column, 4th column, 2nd column, and then the 3rd column – a sort sequence of { 0, 2, 3, 1 }. Case matters and you can treat each character as its byte value. A key of "`abBA`" would have the column sequence { 2, 3, 1, 0 }.
- You need not impose limits on the key but may assume that it will be no longer than 512 bytes and will contain printable ASCII text.
- The key must be at least one character long (if that's the case, the ciphertext will be identical to the plaintext).
- The optional *blocksize* parameter specifies the maximum size of the data that you will read at one time. This enables us to identify the table size. Note that the block size does NOT have to be a multiple of the table size. For example:

  ```
  ctencrypt -b 12 -k mykey
  ```

  will result in a table of five columns (the length of `mykey`) and three rows. The last row will never have more than two characters in it due to the block size. You must be prepared to handle non-padded incomplete tables in every block.
- If the block size is not specified, use a default size of 16 bytes. The minimum acceptable block size is 1.

Testing

Test your programs thoroughly. Come up with different test cases and validate them. *No input should generate exceptions, stack dumps, or result in the program dying unexpectedly*.

Be sure to test and handle cases where:

- The key length is longer than the content.
- The block size is or is not a multiple of the key size.
- The key length is longer than the block size.
- The content is shorter than one block.
- The content is longer than one block.
- The content is arbitrary data (e.g., a jpeg file, a zip file).

Printing input & output of data (as hex #s, for example) can help you see what's going on in your program.

The Linux (and macOS) *od* command dumps binary data and may be useful for inspecting your output. The command

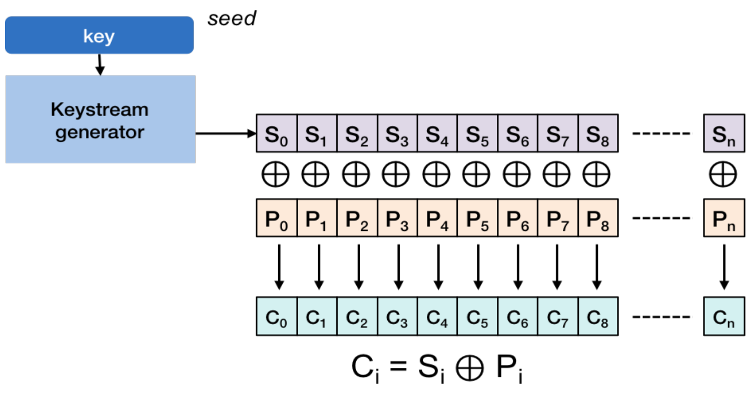
```
    od -t xC keyfile
```

shows the contents of `keyfile` as a series of hexadecimal bytes.

Part 2: Stream cipher

We also covered stream ciphers in class. These simulate the one-time pad by using a *keystream generator* to create a keystream that is the same length as the message.

The keystream generator is simply a pseudorandom number generator, and the seed is derived from the password. You will always see the same sequence of numbers for the same seed.


key
seed
Keystream generator
S_0 S_1 S_2 S_3 S_4 S_5 S_6 S_7 S_8 ------ S_n
P_0 P_1 P_2 P_3 P_4 P_5 P_6 P_7 P_8 ------ P_n
C_0 C_1 C_2 C_3 C_4 C_5 C_6 C_7 C_8 ------ C_n
$C_i = S_i \oplus P_i$

To implement this cipher, you will:

(a) Implement a linear congruential generator

It is a trivial formula that is described here:
https://en.wikipedia.org/wiki/Linear_congruential_generator

This is one of the best-known and widely used pseudorandom number generators. Each pseudorandom number is a function of the previous one and is defined as:

$X_{n+1} = aX_n + c \bmod m$

where:

X_{n+1} is the next pseudorandom number in the sequence
X_n is the number before that in the sequence
m is a modulus. We will be working only on bytes in this assignment, so you will use 256 for the modulus (since that is 2^8 and will produce a range of values that fit within a byte).

The values a and c are magic parameters. Certain values were found to produce better sequences of data. You will use the same parameters that are used in ANSI C, C99, and many other places:

Modulus, $m = 256$ (1 byte)
Multiplier, $a = 1103515245$
Increment, $c = 12345$

Implementing this is only three lines of code. By using a well-known formula, your output should be the same regardless of the programming language or operating system you use.

(b) Convert the password to a seed

The seed for a pseudorandom number generator is just a number. For this program, instead of asking users to use a number as a key, you will let them use a textual password. You will then apply a hash function to this password to create a seed for the keystream generator.

To create the seed, we will use a hash function that works well and is easy to implement. This is the *sbdm* hash that is used in gawk, the *sbdm* database, Berkeley DB, and many other places. You can find the C code for it here: http://www.cse.yorku.ca/~oz/hash.html

```
static unsigned long
sdbm(unsigned char *str) {
      unsigned long hash = 0;
      int c;
      while (c = *str++)
            hash = c + (hash << 6) + (hash << 16) - hash;
      return hash;
}
```

You should be able to translate it to whatever language you're programming in easily. As you can see, this implementation is also three lines of code (logic, not declarations).

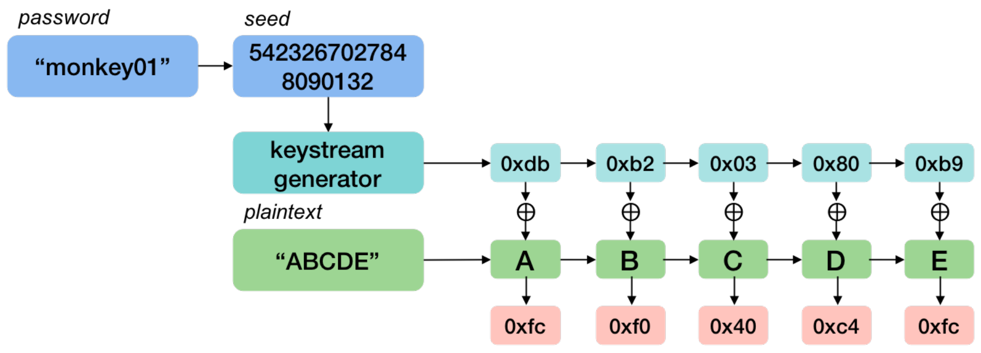
As with the previous step, this implementation should ensure that your output will be the same regardless of the programmer, programming language, or operating system.

(c) Apply the stream cipher

The ciphertext is generated byte by byte and is simply:

$$ciphertext_i = plaintext_i \oplus keytext_i$$

Each byte of plaintext is XORed with the next byte from the keystream generator to produce a byte of ciphertext. Because applying an exclusive-or of the same key a second time undoes the first exclusive-or, you only need to implement one command.



Usage:

```
    scrypt password plaintext ciphertext
    scrypt password ciphertext plaintext
```

The *password* is a text string. The parameters *plaintext* and *ciphertext* are files. The same program can be used to encrypt or decrypt.

Your program should handle data of any size (e.g., gigabytes). As such, it's not a good idea to read the entire file into memory a priori.

Hints & testing

Before implementing your cipher, you should test that your seed and keystream generation works as expected. It's important that different cipher implementations are all compatible – even if they are written by different people in different languages and on different platforms.

You are provided with a program called `prand-test` that lets you enter a password and see the seed and enter either a password or seed to get a pseudorandom stream of bytes.

The usage is:

```
  prand-test [-p password | -s seed] [-n num]
```

You can supply a password with the `-p` parameter and see the seed:

```
  $ ./prand-test -p monkey01
  using seed=5423267027848090132 from password="monkey01"
```

It's possible that the seed may be different if you're using python, which implements arbitrary-precision arithmetic instead of 64-bit integers, but your sequence should be the same since we're taking the modulus of the results.

To see a sequence of pseudorandom bytes, use the `-n` parameter. Here are the first five bytes from the password `monkey01`:

```
./prand-test -p monkey01 -n 5
using seed=5423267027848090132 from password="monkey01"
189
178
3
128
185
```

You can also test the sequence from a seed number with the `-s` parameter. Here are the first four bytes from the seed 85:

```
./prand-test -s 85 -n 5 using seed=85
106
91
248
209
54
```

Part 3: Block encryption with cipher block chaining and padding

This is an enhancement of Part 2 to use the concepts of:

- Processing data in 16-byte blocks
- Padding – adding it and removing it correctly
- Shuffling bits within a block
- Cipher block chaining.

We modify the stream cipher above to have it operate on 16-byte blocks instead of bytes. This turns it into a form of block cipher. A block cipher generally uses multiple iterations (rounds) of substitutions & permutations to add confusion & diffusion. Confusion refers to changing bit values as a function of the key so that each bit of the ciphertext is determined by several parts of the key. Diffusion refers to the property that a change in one bit of plaintext will result in many bits of the ciphertext changing within the block (about half).

In this implementation, we will not use multiple rounds of an SP network. Instead, we will keep the mechanisms of the stream cipher in place but enhance it in two ways: cipher block chaining and shuffling bytes within the block.

Padding

Block ciphers work on a block (a group of bytes) of data at a time. In our case, we will be processing 16-byte blocks.

Not every file is an exact multiple of 16 bytes so we may encounter a partial block at the end. To support this, every block cipher needs to support padding, which is the mechanism for adding extra bytes to fill the block. Padding must be added in such a way that we can detect and remove the padding when decrypting a message.

The way you will implement padding is by adding between 1 and 16 extra bytes at the end of the file. Each byte of the padding data is a number that tells you how many bytes were added. This is a technique that allows you to know how much padding needs to be removed when decrypting and writing the plaintext output.

In the case that the file was an exact multiple of 16 bytes, we added an entire extra block of padding. Otherwise, we would never know if we had padding. Simply looking at the last byte of the last block of the file will tell us how much padding to ignore.

Here are a few examples. In the first, the text "`I am done.`" takes up 10 bytes so we have 6 bytes left over. Each of those bytes will contain a pad byte with a value of 6. Note that this is not the ASCII character 6 but the number 6.

I		a	m		d	o	n	e	.	06	06	06	06	06	06

In this example, we have the text "`This is the end`". It takes up 15 bytes, so we need to add one byte of padding. This padding byte contains the value 1.

T	h	i	s		i	s		t	h	e		e	n	d	01

In the final example, we have the text "`This is the end.`" with a period at the end. This message takes up exactly 16 bytes. Because of this, we need to add an extra block filled with bytes containing the number 16.

T	h	i	s		i	s		t	h	e		e	n	d	.

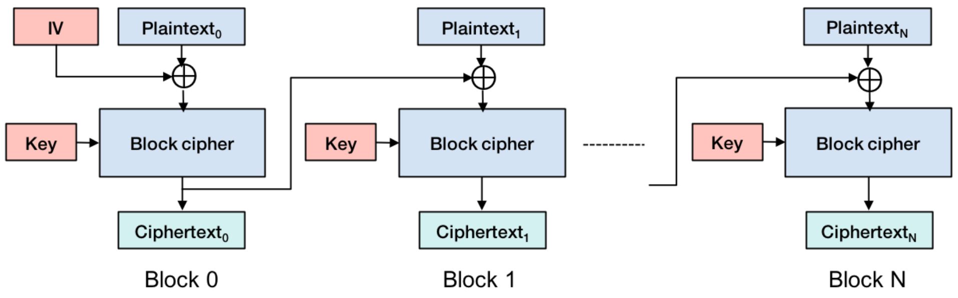
16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16

When you decode the message, you will need to remove the padding.

Cipher block chaining

Stream ciphers do not provide diffusion. The change of a bit in plaintext will generally affect only that bit in ciphertext. We will add diffusion across the output by adding cipher block chaining (CBC). With cipher block chaining, we exclusive-or the previous cipher block with the current block.

$$c_i = E_K(m_i) \oplus c_{i-1}$$



Permutations

Confusion is determined by the seed and the pseudorandom output of the keystream generator in this implementation, but we will enhance the degree of confusion by permuting data in the form of transposition: shuffling bytes of the block.

We will use the 16-byte key from the keystream to determine which sets of bytes in the block to exchange (swap). Each byte of the key is treated as two 4-bit values (a 4-bit value represents values between 0 and 15). These values will identify which set of bytes to exchange. We do this 16 times, going through all 16 bytes of the key.

For each 16-byte block, do the following:

```
for (i=0; i < blocksize; i=i+1)
      first = key[i] & 0xf  (lower 4 bits of the keystream)
      second = (key[i] >> 4) & 0xf  (top 4 bits of the keystream)
      swap(block[first], block[second])  (exchange the bytes)
```

Cipher operation

The flow of the cipher is the following:

Start by creating an initialization vector (IV) for applying CBC to the first block. In our case, this will be the first 16 bytes read from the keystream generator. These bytes will not be used for anything else.

Then, for each 16-byte plaintext_block *i*:

1. If it is the last block, add padding. This will be an amount from 1 through 16 bytes (e.g., finish up a block or add a new block). The padding is added before any encryption or shuffling takes place.
2. Apply CBC: $temp\_block_N = plaintext\_block_N \oplus ciphertext\_block_{N-1}$
   Use the initialization vector if this is the first block.
3. Read 16 bytes from the keystream.
4. Shuffle the bytes based on the keystream data.
5. $ciphertext\_block_N = temp\_block_N \oplus keystream_N$.
6. Write $ciphertext\_block_N$.

You need to use the same password hashing and keystream generator as in Part 2.

You will need to write two programs for this part, one to encrypt and another to decrypt:

```
sbencrypt password plaintext ciphertext
sbdecrypt password ciphertext plaintext
```

As with Part 2, the program will take a `password` string, which will be hashed and used as a seed for the keystream generator. The parameters `plaintext` and `ciphertext` are both file names.

Reference programs

It's important that encryption software works consistently across multiple systems regardless of author, programming language, or operating system. I should be able to encrypt a message on my Mac and expect you to be able to decrypt it with your program on your Raspberry Pi running Linux.

You are provided with reference versions of the programs that you can use to compare with yours and, perhaps, help debug your code. The `linux` directory contains intel architecture linux versions of the executables and the `macos` directory contains macOS versions (arm64 and i386 universal binary). The `samples` directory contains some sample files you can use for testing, but you should also create your own files and keys to test your program thoroughly. Be sure to test edge cases, such as empty files, one-byte files, and files that require different amount of padding (for the block cipher).

Here are the programs provided:

Columnar transposition cipher

```
table [-u] [-r rows] [-c cols] [file ...]
```

is a test program for a basic table-based transposition cipher that does not use padding described at the beginning of this document. The -u flag handles decryption.

```
      ctencrypt [-d] [-b blocksize] -k key [plaintextfile]
```

Encrypt a *plaintext* file to create *ciphertext* output using a row width and transpositions defined by the key. Perform encryption in chunks of the block size.

```
      ctdecrypt [-d] [-b blocksize] -k key [ciphertextfile]
```

Decrypts content encrypted with `ctencrypt`.

The `-d` flag turns on debugging information and shows you what data is being read and output.

Stream Cipher – keystream test

Before you test your cipher, make sure that your password hash and pseudorandom number generator are producing the proper results. You can test this with the `prand-test` program:

```
      prand-test [-p password | -s seed] [-n num]
```

If the program is supplied a password with the `-p` parameter, the password will be hashed and the result shown.

The `-n` parameter lets you specify the number of pseudorandom numbers to be printed. The default is 0.

If you just want to see the list of pseudorandom numbers generated from a specific seed, you can specify a seed number instead of a password with the `-s` parameter.

Stream Cipher

```
      scrypt [-d] password plaintext ciphertext
```

Encrypt a *plaintext* file into a *ciphertext* file using a keystream derived from the *password* string. The same command decrypts a *ciphertext* file into a *plaintext* file.

```
      scrypt [-d] password ciphertext plaintext
```

The `-d` flag turns on debugging mode and shows the series of xor operations from the source file to the output file.

Block Cipher

```
      sbencrypt [-d] password plaintextfile ciphertextfile
```

Encrypts a *plaintext* file into a *ciphertext* file using a keystream derived from the *password* string. The command:

```
      sbdecrypt [-d] password ciphertextfile plaintextfile
```

Decrypts a *ciphertext* file into a *plaintext* file using a keystream derived from the *password* string.

For both commands, the `-d` flag enables debugging, showing the sequence of shuffling per block.

Samples

You should test your programs on arbitrary keys and content. The samples directory contains some content you can use to test your programs:

`0123.bin`	The bytes 0, 1, 2, 3
`01234.bin`	The bytes 0, 1, 2, 3, 4
`alice.txt`	The text to Alice in Wonderland
`clown.jpg`	A jpeg file
`oz-letters.txt`	The Wizard of Oz – with all lowercase letters converted to uppercase and any streams of non-printable characters converted to a single space.
`oz.txt`	The Wizard of Oz
`poem.txt`	The Walrus and the Carpenter
`test-15.txt`	15 bytes of text.
`test-16.txt`	16 bytes of text.
`test-41.txt`	41 bytes of text.
`test-a.txt`	Just the letter '`A`'
`test-abc.txt`	The string `ABC`
`test-null.txt`	An empty file.

What to submit

Place your source code into a single *zip* file. If code needs to be compiled (i.e., Java, C, or Go), please include a `Makefile` that will create the necessary executables.

We don't want to figure out how to compile or run your program. We expect to:

1. unzip your submission
2. run make if there's a Makefile to build everything that needs to be built

3. Set the mode of the programs to executable:
   `chmod u+x ctencrypt ctdecrypt scrypt sbencrypt sbdecrypt`
4. Run the commands as:

```
./ctencrypt -b blocksize  -k key plaintext >ciphertext
./ctdecrypt -b blocksize  -k key ciphertext >plaintext
./scrypt password plaintext ciphertext
./sbencrypt password plaintext ciphertext
./sbencrypt password ciphertext plaintext
```

If you are using python, you can submit either:

A. A shell script with the name of each command that runs each program. For example, a file named `vencrypt` that contains the text:

```
#!/bin/bash
/usr/bin/python3 ./ctencrypt "$@"
```

or

B. (*preferably*) a program named (e.g., named `ctencrypt`) that contains your source code and starts with the line:

```
#!/usr/bin/python3
```

If you are using Java, you will have a simple `makefile` that compiles the Java code to produce class files. The `ctencrypt` program will be a script that runs the *java* command with the necessary arguments, containing contents such as:

```
#!/bin/bash
CLASSPATH=. java CTencrypt "$@"
```

If you're using C, Go, or any language available on the iLab machines, provide a `makefile` for generating the required executables.

Test your scripts on an iLab machine to make sure they work prior to submitting.