CS 419 Project 1

Due Monday, February 17, 2025 6:00pm

Introduction

This assignment comprises three parts:

1. In part 1, you will implement a stream cipher that uses a linear congruential keystream generator and a hashed key.

2. In part 2, you will modify the cipher in part 1 to use an initialization vector so that encrypting the same plaintext with the same key produces different results.

3. In part 3, you will implement a simple Feistel cipher using 64-bit derived keys on 128-bit blocks and use padding.

A note about 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 content is shorter than one block, longer than one block, and exactly one block.
- The content is empty.
- The content is arbitrary data (e.g., a jpeg file, a zip file).
- The input file does not exist or is not readable.
- The output file cannot be created.
- The command usage is incorrect.

Compare your decrypted results to the original file using the `cmp` command, not visually.

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 myfile
```

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

Part 1: Stream cipher

Stream ciphers simulate a 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.

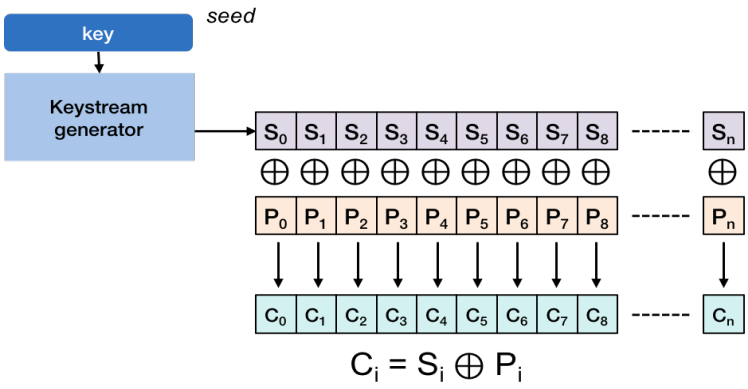
To implement this cipher, you will:

(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. Instead of asking users to use a number as a key for this program, 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

```
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 easily translate this to whatever language you're programming in. As you can see, this implementation is also three lines of code (ignoring declarations).

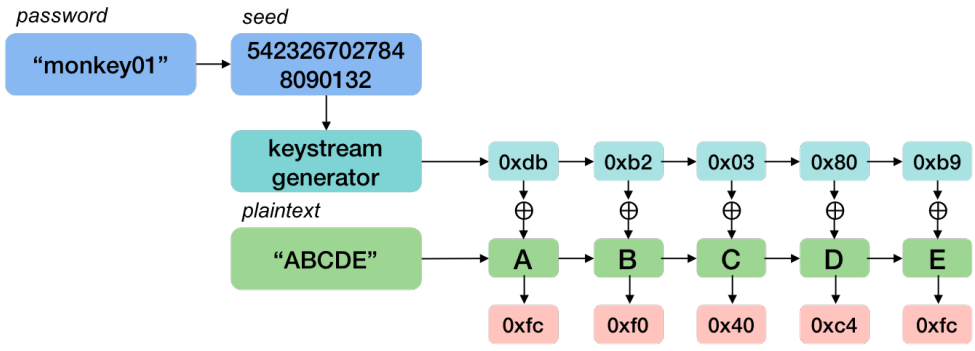
As with the previous step, this implementation should ensure that your output will be the same regardless of the programmer, programming language, or operating system.

(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 generated 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 2: Stream cipher with an initialization vector

This is an enhancement of Part 1 to add an initialization vector.

Initialization vector

If we use the same key for multiple encryptions, the same keystream will be used for each encryption session. That means that:

- Identical byte sequences in the same position in different files will encrypt to the same ciphertext.
- An attacker can exclusive-or two ciphertexts that were encrypted with the same key and produce a result that is equivalent to the exclusive-or of both plaintexts. In some cases, this can leak data (e.g., the class example with image files).
- An attacker can easily identify when identical messages are sent because they produce the same ciphertext.
- If an attacker happens to get the plaintext and ciphertext for one message, they can recover the keystream and use it to decrypt other messages that were encrypted with the same key.

This is the reason why it is critical never to reuse the same key in a one-time pad.

To avoid this problem in a stream cipher, we permute the key with an initialization vector (IV). Using a random initialization vector ensures that even the same data encrypted with the same key will generate a different ciphertext each time.

You will create a random 8-byte (64-bit) initialization vector. Reading bytes from `/dev/random` on Linux and macOS should give you a good source of randomization.

An initialization vector is not considered a secret, but both the encrypting and decrypting sides need to know what it is. The initialization vector is sometimes sent during an initial handshake when setting up communications. In this case, we will write the initialization vector as the first 8 bytes of the ciphertext file so the decrypting program can simply read those bytes to get the vector.

The initialization vector will be written out in little-endian byte ordering, meaning the least significant byte will be written first. That is, if your initialization vector is `0x0102030405060708` then you will write the bytes to the file in the sequence { `0x08`, `0x07`, `0x06`, … , `0x01` }. This is the standard data storage format on Intel architectures and is commonly used on ARM architectures. It's important to specify since a program reading decrypted data would need to get the right value regardless of the machine architecture.

Note that in block ciphers, initialization vectors are often the same length as the block since, in modes such as cipher block chaining (CBC), they are exclusive-ORed with the block of ciphertext. In a stream cipher, the initialization vector is treated as a number that will provide a random seed to a keystream generator even if the provided key is the same.

The program

Because you are writing the initialization vector to the file when encrypting and reading the initialization vector when decrypting, encryption and decryption become separate functions.

Usage:

```
vcrypt -e password plaintext ciphertext
vcrypt -d password ciphertext plaintext
```

The *password* is a text string. The parameters *plaintext* and *ciphertext* are files.

The *vcrypt* program with the `-e` flag encrypts the plaintext file to produce ciphertext. It hashes the password to generate a key that is then exclusive-ORed with a random 8-byte initialization vector. The result is used to seed the keystream generator. The initialization vector is written in little-endian format as the first eight bytes of the ciphertext file.

The *vcrypt* program with the `-d` flag decrypts the ciphertext file, hashing the password in the same manner and reading the first eight bytes of the file to extract the initialization vector. As with *sencrypt*, the key derived from the password is exclusive-ORed with the initialization vector to generate the seed for the keystream generator and successive bytes are decrypted as in Part 2.

Part 3: Block cipher

The last part is implementing a 10-round Feistel cipher with 64-bit keys and 128-bit blocks.

A Feistel cipher is a symmetric encryption structure used in many modern block ciphers, including DES (Data Encryption Standard), Camellia, and Blowfish. It divides the plaintext into two halves and processes them through multiple rounds of encryption:

1. **Splitting**: The input block is divided into **left (L) and right (R) halves**.
2. **Rounds**:
   - In each round, the **right half (R)** is input into a **round function** (which can involve key mixing, substitutions, and permutations). These are guided by the **round key**, a key that is derived from the primary key.
   - The result is **XORed** with the left half (L).
   - The halves are then **swapped**:

     $$R_{i+1} = L_i \oplus F(R_i, K_i)$$
     $$L_{i+1} = R_i$$

     Where

     L_i and R_i are the left and right halves of the block.
     K_i is the round key.
     F is the round function.
     i is the round iteration.
3. **Final Swap**: After all rounds, the halves are swapped one last time.

Some properties of the Feistel structure that make it attractive are:

- Decryption is identical to encryption but uses the round keys in reverse order.
- Round functions do not need to be reversible (unlike other ciphers like AES).
- The structure can be used with various round functions and key schedules.
- The cipher can be highly secure when with properly designed round functions.
- The symmetric nature makes it easy to implement and design round functions.

Your assignment is to implement a Feistel cipher. It will use a simple *round* function, which will not make it cryptographically secure. Here are the requirements:

Program usage

Usage:

```
feistel -e password plaintext ciphertext
```

Will encrypt the given plaintext file using a key that is derived from the provided textual password as described in the *Key Generation* section below.

```
feistel -d password ciphertext plaintext
```

Will decrypt the given ciphertext file using a key that is derived from the provided textual password.

Key generation

Main key
The program will accept a textual password that will be converted into a 64-bit key using the *sbdm* hash function you used in the first two parts. Be sure that your function returns a 64-bit value.

Round keys:
The cipher iterates through each block 10 times. You will generate ten (10) 64-bit subkeys (round keys), one for each iteration (round) using the linear congruential generator from the stream cipher.

```
a = 1103515245
c = 12345
round_key[0] = key;
round_key[i] = (round_key[i-1]*a + c) & 0xffffffffffffffff
```

Encryption

Read a block (128 bits) of data at a time. For each of 10 iterations:

1. Split the block in half: a left half and a right half.
2. New `right_half = left_half ^ round(right_half, key[i])`, where `i` is the round number 0 .. 9.
3. New `left_half = original right_half`.

After the 10 iterations, swap the right and left halves of the block to produce the final ciphertext block.

Decryption

Decryption is the same as encryption except that you'll go through the round keys in reverse order:

1. Split the block in half: a left half and a right half.
2. New `right_half = left_half ^ round(right_half, key[i])`, where `i` is the round number 9..0.
3. New `left_half = original right_half`.

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 (128-bit) blocks.

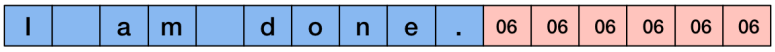
Since not every file is an exact multiple of 16 bytes, we may encounter a partial block when we reach the end of the file. To support this, every block cipher needs to support padding, which adds extra bytes to fill the block. Padding must be added in such a way that we can detect and remove the padding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

You will use the PKCS#7 scheme. PKCS stands for Public-Key Cryptography Standards and is a collection of standards to handle various aspects of encryption and information storage for public key cryptography. PKCS#7 defines a padding format and ensures that data blocks align with the required block size for block ciphers.

You will implement padding 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.

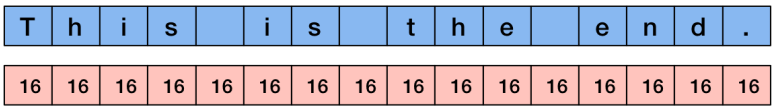
In the case that the file was an exact multiple of 16 bytes, we must add an entire extra block of padding. Otherwise, we would never know if we had padding since we cannot control the unpadded file's last byte. Simply looking at the last byte of the last block of the file will tell us how much padding 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.



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.

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.



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

For encrypting, you will add padding before encrypting the block (since you cannot encrypt a partial block). If you added an entire block of padding, you will encrypt that as well.

For decrypting, you will decrypt each block. When you reach the final block of the file, you will decrypt it and then strip off padding before writing the output.

Round function

A good Feistel cipher will use a round function that contains a series of substitutions and permutations. However, the nice thing about implementing a Feistel cipher is that any round function will work. Unlike an SP-network cipher like AES that goes through multiple rounds of substitutions and permutations, we don't have to worry about preserving data. The Feistel's XOR on left and right halves preserves it for us.

We will apply a simple transformation on the data. The round function will do the following:

```
temp = (input ^ round_key) * 0xa3b2c1
return (temp >> 23) | (temp << 41)
```

Where `^` is the xor operation, `>>23` performs a right shift by 23 bits, `<<41` performs a left shift by 41 bits, and `|` is the XOR operation. All operate on 64-bit unsigned data.

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 using 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.
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 amounts of padding (for the block cipher).

Here are the programs provided:

Part 1: 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.

Part 1: Stream Cipher

```
scrypt [-Ds] 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 [-Ds] 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.

The `-s` flag shows the results of hashing the password to generate the seed for the keystream generator.

Part 2: Stream Cipher with Initialization Vector

```
vcrypt -e [-Dzs] password plaintext ciphertext
```

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

```
vcrypt -d [-Dzs] password ciphertext plaintext
```

The command with a `-d` flag decrypts a *ciphertext* file into a *plaintext* file.

The `-D` flag turns on debugging, showing all the xor operations taking place.

The `-s` flag shows the initialization vector and the results of hashing the password to create the seed for the keystream generator.

The `-z` flag uses 0 for an initialization vector instead of a random number.

Part 3: Feistel cipher

```
feistel -e [-Dk] password plaintext ciphertext
```

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

```
feistel -d [-Dk] password ciphertext plaintext
```

The command with a `-d` flag decrypts a *ciphertext* file into a *plaintext* file.

The `-D` flag turns on debugging, showing all the block values in each round of encryption.

The `-k` flag shows the round keys that are generated from the supplied password.

Samples

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

`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 scrypt vcrypt feistel`
4. Run the commands as:
   `./scrypt` *`password plaintext ciphertext`*
   `./vcrypt -e` *`password plaintext ciphertext`*
   `./vcrypt -d` *`password ciphertext plaintext`*
   `./feistel -e` *`password plaintext ciphertext`*
   `./feistel -d` *`password ciphertext plaintext`*

If you are programming in python, submit a program named (e.g., named `cencrypt`) that contains your source code and starts with the line:

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

DO NOT submit programs named `cencrypt.py`, `cdecrypt.py`, etc. We expect to be able to run your programs within scripts, so the usage of your programs must align exactly with step 4 above. **Do not give your programs different names. If they don't run as instructed, you will lose many or all points.**

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

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

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.