# COMP 7402 Assignment 5 Design

Peyman / Dimitry

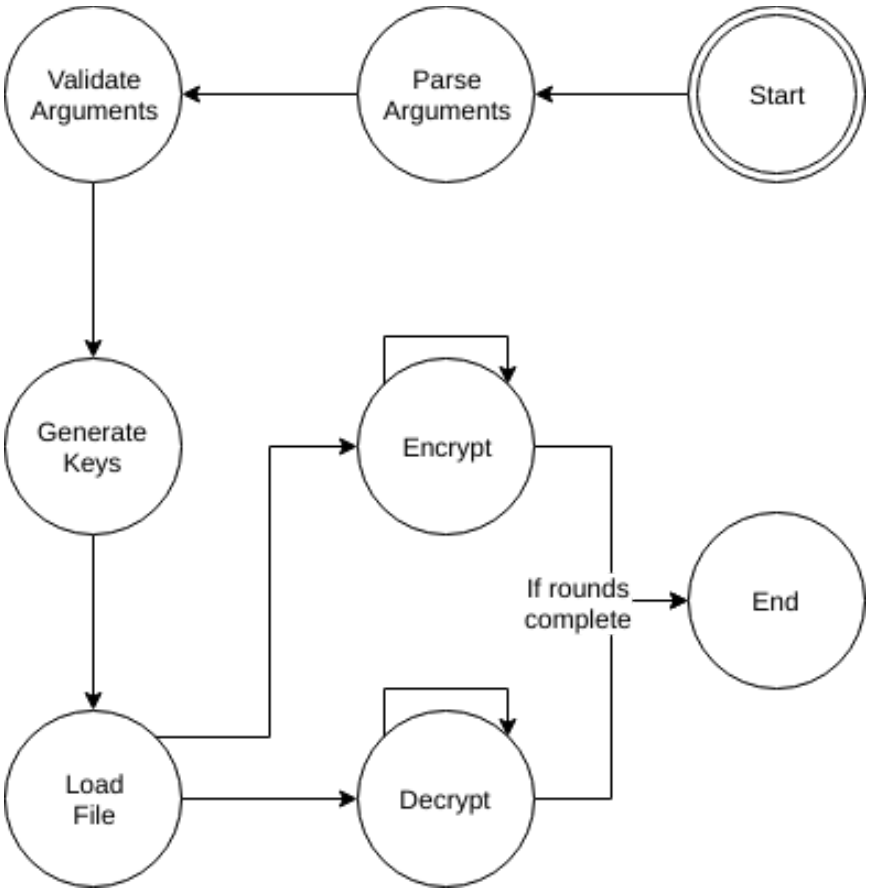
# Design

This application was written in C and is compiled with cmake. For ease of use two scripts “compile.sh” and “run.sh” which are very self explanatory. The compilation script uses cmake to compile the code correctly while the runner script executes the program on a project file and tests for differences. The runner encrypts and decrypts the input file in both the ECB and CBC modes. The code uses bitwise rotation to permute the initial key passed in through the command line. The key “abcd” which is 32 bits becomes “0x64636261” initially. Here are the permutations of this key:

1. 0x64636261
2. 0xc8c6c4c2
3. 0x918d8985
4. 0x231b130b
5. 0x46362616
6. 0x8c6c4c2c
7. 0x18d89859
8. 0x31b130b2

As you can see, with this method a rather great degree of randomness is added to each round. The function f(x,k) from the feistel documentation was implemented by using xor on the key and the right section. The blocks of data were originally 8 byte long character arrays but upon further research they were changed to the type uint64_t for ease of computation.

# State Diagram



# Pseudocode

```
main(Arguments) {
        Parse Arguments
        Validate_inputs()
        Open Input file
        Open Output file
        Generate_keys()
        If (encryption selected) {
                If (ECB selected)
                        encrypt_ecb(input, output, keys)
                Else if (CBC selected)
                        encrypt_cbc(input, output, keys)
                Else if (CTR selected)
                        encrypt_ctr(input, output, keys)
        }else if (decryption selected) {
                If (ECB selected)
                        decrypt_ecb(input, output, keys)
                Else if (CBC selected)
                        decrypt_cbc(input, output, keys)
                Else if (CTR selected)
                        encrypt_ctr(input, output, keys)
        }
}

encrypt_ctr(input, output, keys){
        Instantiate seed for random
        Instantiate 64bit Block
        Instantiate 32 bit sections Left, Right
        Generate nonce
        While input not done
                Left and right from nonce
                Block = block xor encrypt()
                write(block)
                Increment nonce
}

decrypt_ctr(input, output, keys) {
        Instantiate seed for random
        Instantiate 64bit Block
        Instantiate 32 bit sections Left, Right
        Generate nonce
```

```
        While input not done
                Left and right from nonce
                Block = block xor encrypt()
                write(block)
                Increment nonce
}

encrypt_ecb(input, output, keys) {
        Instantiate 64bit Block
        Instantiate 32 bit sections Left, Right
        While (input not done) {
                Read ( block, 64 bits, from input)
                If ( read error) {
                        exit()
                }
                Left = first 32 bits of block
                Right = last 32 bits of block
                Left = xor (Left ,  xor ( Right , keys) )
                Block = Right + Left // swapped
                Write( Block, 64 bits, into output)
        }
}

decrypt_ecb(input, output, keys) {
Instantiate 64bit Block
        Instantiate 32 bit sections Left, Right
        While (input not done) {
                Read ( block, 64 bits, from input)
                If ( read error) {
                        exit()
                }
                Left = first 32 bits of block
                Right = last 32 bits of block
                Left = xor (Left ,  xor ( Right , keys) ) //keys read in inverse
                Block = Right + Left // swapped
                Write( Block, 64 bits, into output)
        }

}

encrypt_cbc(input, output, keys) {
        Instantiate 64bit Block, Previous_Block
        Instantiate 32 bit sections Left, Right
```

```
        Set Previous_Block = 0xFFFFFFFF // for initial value
        While (input not done) {
                Read ( block, 64 bits, from input)
                If ( read error) {
                        exit()
                }
                Block = xor(Block, Previous_Block)
                Left = first 32 bits of block
                Right = last 32 bits of block
                Left = xor (Left ,  xor ( Right , keys) )
                Block = Right + Left // swapped
                Previous_Block = Block
                Write( Block, 64 bits, into output)
        }

}

decrypt_cbc(input, output, keys) {
        Instantiate 64bit Block, Previous_Block, Saved_Block
        Instantiate 32 bit sections Left, Right
        Set Frirst = 1
        While (input not done) {
                Read ( block, 64 bits, from input)
                If ( read error) {
                        exit()
                }
                Saved_Block = Block
                Left = first 32 bits of block
                Right = last 32 bits of block
                Left = xor (Left ,  xor ( Right , keys) ) // Keys in reverse
                Block = Right + Left // swapped
                If (First is 1) {
                        Block = xor( Block, 0xFFFFFFFF) // because of initial value of prev block
                        First = 0
                } else {
                        Block = xor(Block, Previous_Block)
                }
                Previous_Block = Saved_Block
                Write( Block, 64 bits, into output)
        }

}
```

# Usage

## Compiling

1. chmod +x compile.sh
2. ./compile.sh

## Testing

1. chmod +x run.sh
2. ./run.sh

## Encrypting

- ./bin/fiestel -e -i <infile> -o <outfile> -m [ecb|cbc|ctr] -k <32bitkey (4chars)>

## Decrypting

- ./bin/fiestel -d -i <infile> -o <outfile> -m [ecb|cbc|ctr] -k <32bitkey (4chars)>