Project 2 Simplified DES

Adam Eisert

The data Encryption Standard is a symmetric key algorithm used to encrypt digital data. It normally operates with a block size of 64 bits, but for this project a simplified version of DES was implemented, Scaled Down (SD_DES). Instead of 64 bits, the block size is 8 bits, the key size is 12 bits, and the number of rounds is 3. Rounds are modeled Feistel rounds. The program was created using Python, for the ease of converting between data types.

The block size of 8 bits is just enough to encode a single character in ascii, from 0 to 255. The block size is also enough to cover values 0-4095, and can be represented as 3 hexadecimal values. The program was created with these assumptions. The users supply either a string of ascii characters, binary, hex values, or decimal values to encrypt or decrypt, as well as 3 hex characters to act as the key.

The first step of SD_DES is to verify that the key is an appropriate size. If not, it returns, asking for a new key. If the key is within the proper range 0-4095, the actual encryption begins. Since the input is a string of ascii/binary/hex, SD_DES splits the input into an array where each cell holds a character representing an 8 bit binary string. The key is then sent to the key scheduler key_sched. Key_sched converts the valid hex representation of the key into the corresponding binary. The first 4 binary bits are assigned to key 1 (k1), the second set of 4 binary bits is assigned to k2, and the last 4 are assigned to k3. The pairs k1 +k2, k2 + k3, and k3 + k1 are then XOR'd together, and represented as 3 new keys. The first pair is assigned as k1, second pair as k2, and third pair as k3.

After this pre-processing phase, the actual encryption/decryption can begin. The list of 8 bit blocks is iterated through. Each block is sent through the Feistel cipher 3 times, each time sending the round number and key set. The Feistel cipher then determines, depending on the round number, and whether it is encryption or decryption, which key to XOR the results with. No matter the round, the block is split in two, a right half and left half. Depending on the operation, either the left half or right half is held static. The non static half is then XOR'd with the appropriate key, and the other static half. The two halfs are then reformatted as a binary string, and returned. The output of the first round of the Feistel cipher is then sent back in with an incremented round number. The output of the third round of encryption/decryption is then stored, and the next block to be encrypted/decrypted is sent through the cipher. The final output is then returned with both hex and ascii representation, for the user's pleasure.

An example output is shown below, using the input "Hello Friend, its good to see you again.", and the key 0xf8c.

c = pre_e('Hello Friend, its good to see you again.',0xf8c)

print(c[1])

- R[[XqE^RYS^CDPXXSCXDRRNXBVPV^Y

p = pre_d(c[1],0xf8c)

p[1]

Hello Friend, its good to see you again.

As can be seen, the output is complete nonsense. This is not unique, many ciphers can create absolute nonsense. However the difference is that instead of simply shifting the characters around or mapping alphabetical characters to other alphabetical characters, the values are XOR'd with values up to 4095. This creates a massive search space, but the statistics of this cipher will be explored next.

In order to analyze DES, the first chapter of The Hobbit was used. To simplify, the baseband information was all that was extracted and operated on. The text was then encrypted, and the summary statistic information of the cipher and plaintext was calculated. This was repeated for two keys,

K1 = 0xf8c

K2 = 0x1fb.

The effects of cascading ciphers were also explored.

To start, the summary statistics for plaintext can be seen. The story begins;

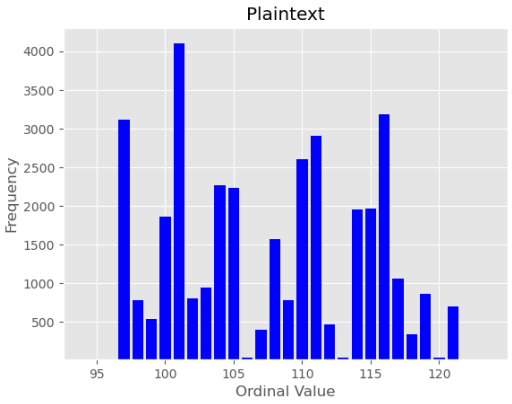
Plaintext: Inaholeinthegroundtherelivedahobbit

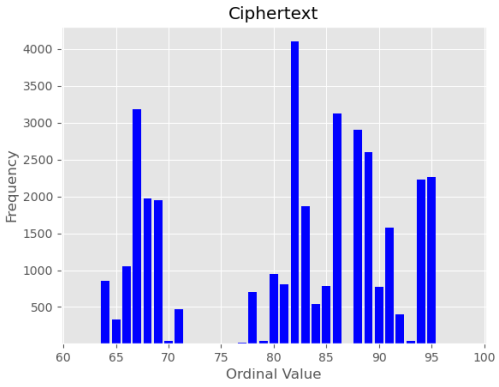
Key 1: ^YV_X[R^YC_RPEXBYSC_RER[^ARSV

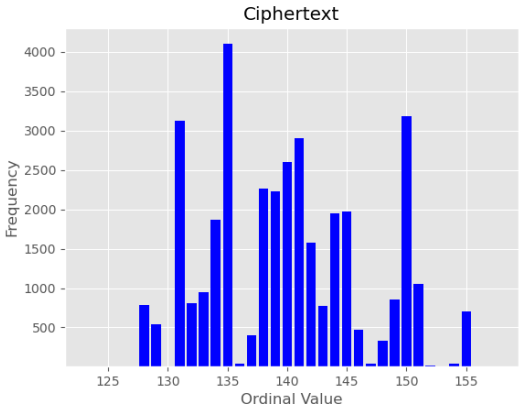
Key 2:
\x8b\x8c\x83\x8a\x8d\x8e\x87\x8b\x8c\x96\x8a\x87\x85\x90\x8d\x97\x8c\x86\x96\x8a\x87\

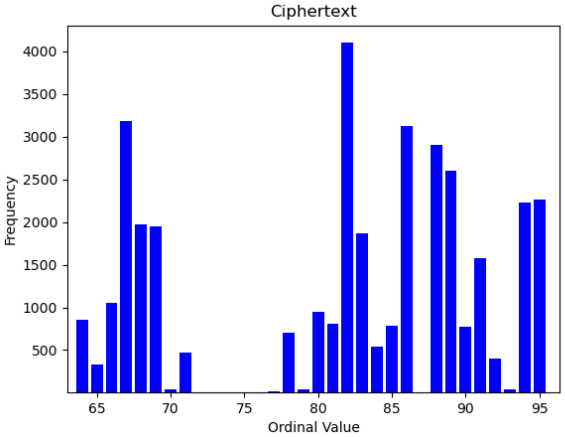
	Average	Median	Mode	Variance	Entropy
Plaintext	107.54	119	66	46.46	15.1
K1	81.56	87	82	93.7	15.1
K2	139.53	149	135	42.14	15.1

The averages vary wildly from key to key, as do the mean, median, mode, and variance. However, entropy stays constant. No information is lost in the encryption process. It's difficult to piece a full picture with this table, however, histograms of the frequency of each key are shown below. Plaintext is s in the top left, Key 1 at top right, and Key 2 at the bottom left, and cascading cipher at the bottom right.









The histograms are all both focus around different values, shifted from the plaintexts position like a shift cipher. They also have patterns similar to a permutation of standard character frequency, like a substitution cipher. Since this cipher only encrypts one character at a time, it shares similarities with a substitution cipher, including its weaknesses, statistical analysis. The frequencies are shifted and permuted, but they are still vulnerable to analysis. For instance, the most frequent value from K1 is R, which maps to e in plaintext. The same attacks against a substitution cipher would work on this baseband.

However, this is only due to the substitution ciphers small block size. If block size was 16, it would be more difficult to crack with individual frequency analysis, but it would still be vulnerable to digraph frequency analysis. If block size was 16, it could still be cracked with trigraph analysis. The only way to prevent this frequency is to increase the block size, since it both shifts values, permutes them, and would approach a standard distribution as block and key sizes increase.

Even if the block size was increased, blocks could be tested against every possible key, so to protect against this, a larger key should be implemented, to make frequency analysis unviable, and exhaustive search too expensive.

A final consideration should be that of keeping keys hidden. Even if the keys are massive, with a keyspace of a million possible combinations, it would be broken if the key was stolen. Having all communications based off a large block and key cipher can be cracked with a stolen key. As such, communication with DES should use randomized keys, a new key each session, with no repeats to prevent data from being stolen. All in all, the weaknesses of this crypto system stem from key and block size, and potential key theft.

The SD_DES program is shown below. Decryption can be performed with either the pre_d or pre_e function.

```
#Block Size b = 8 bits#Key Size K = 12 bits##Rounds N = 3#8 bit, 2 hex

import pdb

import math


def feistel(message,key,n1):
    x = format(ord(message),"08b")
    L1,R1 = x[0:4],x[4:8]
    L2 = R1
    t = "0b"+R1
    t = int(t,base=2)
    L11 = int(L1,base=2)
    if(n1 == 0):
        m = t ^ key[0] ^ L11
    if(n1==1):
        m = t ^ key[1] ^ L11
    if(n1==2):
        m = t ^ key[2] ^ L11
    m = format(m,"04b")
    m = L2+m
```

```
    m = int(m,base=2)
    return(m)


def feistel_dec(message,key,n1):
    x = format(ord(message),"08b")
    L2,R2 = x[0:4],x[4:8]
    R1 = L2
    t="0b"+L2
    t=int(t,base=2)
    R11 = int(R2,base=2)
    if(n1==2):
        m=t^key[2]^R11
    if(n1==1):
        m=t^key[1]^R11
    if(n1==0):
        m=t^key[0]^R11
    m=format(m,"04b")
    m=m+R1
    m=int(m,base=2)
    return(m)

def key_sched(key):
    k = list(bin(key))
    k1,k2,k3 = "".join(k[2:6]),"".join(k[6:10]),"".join(k[10:14])
    k1,k2,k3 = '0b'+str(k1),'0b'+str(k2),'0b'+str(k3)
    k1,k2,k3 = int(k1,base=2),int(k2,base=2),int(k3,base=2)
    k1_2,k2_2,k3_2 = k1^k2,k2^k3,k3^k1
```

```
    return(k1_2,k2_2,k3_2)

def sd_des(message,key):
    m = list(message)
    k = key_sched(int(key))
    encTot,enc = [],[]
    for i in range(len(m)):
        if(i > 0):
            enc = []
        for ii in range(3):
            if(ii == 0):
                message = m[0]
                temp = feistel(m[i],k,ii)
                enc.append(hex(temp))
            else:
                message = enc[ii-1]
                temp = feistel(chr(int(message,base=16)),k,ii)
                enc.append(hex(temp))
            if(ii==2):
                encTot.append(enc[2])
    return encTot

def pre_e(message,key):
    if(key > 4095):
        print("Key size Too Large")
        return
    else:
        x = sd_des(message,key)
```

```
        s = ''.join(chr(int(e,base=16)) for e in x)
        return(x,s)

def sd_des_dec(message,key):
    i=0
    m=list(message)
    k=key_sched(int(key))
    decTot,dec=[],[]
    for i in range(len(m)):
        if(i>0):
            dec=[]
        for ii in range(3):
            if(ii==0):
                message=m[0]
                temp = feistel_dec(m[i],k,2)
                dec.append(hex(temp))
            else:
                message=dec[ii-1]
                temp=feistel_dec(chr(int(message,base=16)),k,2-ii)
                dec.append(hex(temp))
            if(ii==2):
                decTot.append(dec[2])
    return decTot

def pre_d(message,key):
    if(key > 4095):
        print("Key size too large")
        return
```

```
    else:
        x = sd_des_dec(message,key)
        s = ''.join(chr(int(e,base=16)) for e in x)
        return(x,s)

c = pre_e('Hello Friend, its good to see you again.',0xf8c)
p = pre_d(c[1],0xf8c)
```