# How to implement DES and Triple DES from scratch: A simple Ruby implementation with lots of comments

By Chris Hulbert - [chris.hulbert@gmail.com](mailto:chris.hulbert@gmail.com)

Full code can be found here: <http://github.com/chrishulbert/crypto/blob/master/ruby/ruby_des.rb>

As a learning exercise recently, I set about implementing a bunch of crypto algorithms from scratch, in such a way as to learn how they work, rather than simply cut and pasting some open source code. Here’s my attempt to document DES in particular, in as simple a way as possible to make it easy to follow and learn from.

This document references the following, which is a very good one-stop-shop for learning DES too: <http://orlingrabbe.com/des.htm>

Keep in mind that this implementation is more aimed at learning how the algorithm works, rather than implementing it in an optimised way, because I often find that these optimisations make it harder to understand what’s really going on. So, if you keep in mind that this code isn’t production-ready, we should get along just fine.

# DES quick summary

DES is a block cipher, eg it can encrypt/decrypt a block at a time. It uses 64 bit blocks, and 64 bit keys – in other words, it works on 8 bytes at a time. It’s getting a bit old and insecure now, but Triple DES is still alive and well, which is a simple addition to the algorithm that I’ll detail at the end. Basically, there are two steps to DES:

* Key expansion
* Encryption / Decryption

# Some utility functions

Ok here’s some utility functions we’ll need to get started:

class String

# Convert a "1010..." string into an array of bits

def to\_bits

bitarr=[]

self.each\_char{|c| bitarr << c.to\_i if c=='0' || c=='1'}

bitarr

end

end

class Array

# Join this array into a nicely grouped string

def pretty(n=8)

s=""

self.each\_with\_index{|bit,i| s+=bit.to\_s; s+=' ' if (i+1)%n==0}

s

end

end

# Expanding the key

Expanding the DES key is performed by the following steps:

* Permute the 64-bit key to produce the 56 bit K+, using the PC1 permutation
* Split K+ in half to produce C0 and D0
* Perform a series of 1 and 2-bit shifts to produce C1..16 and D1..16
* Permute each of C1D1..C16D16 to produce the subkeys K1..K16

In code, this’ll look like:

# Take a 64 bit key, and return all the subkeys K0..K16

def expand(k)

kplus = k.pc1 # Run the key through PC1 to give us "K+"

c0, d0 = kplus.split # Split K+ into C0D0

cdn = shifts(c0, d0) # Do the shifts to give us CnDn

cdn.map{|cd| cd.pc2} # For each CnDn, run it through PC2 to give us "Kn"

end

So lets cover each step in detail:

## PC1 Permutation

The PC1 permutation takes the 64 bit key and returns a 56 bit permutation. So yes, DES in fact throws away 8 bits. Generally they are only used for parity. The permutation code basically re-orders the bits like this:

class Array

# Perform a bitwise permutation on the current array, using the passed permutation table

def perm(table)

table.split(' ').map{ |bit| self[bit.to\_i-1] }

end

# Perform the PC1 permutation on the current array

# This is used to take the original 64 bit key "K" and return 56 bits "K+"

def pc1

perm "

57 49 41 33 25 17 9

1 58 50 42 34 26 18

10 2 59 51 43 35 27

19 11 3 60 52 44 36

63 55 47 39 31 23 15

7 62 54 46 38 30 22

14 6 61 53 45 37 29

21 13 5 28 20 12 4 "

end

end

So, the 57th bit is now the first bit, and so on.

## Split

This function is simple, we are just halving a set of bits into two equal left and right halves:

class Array

# split this array into two halves

def split

[self[0,self.length/2], self[self.length/2,self.length/2]]

end

end

## Shifts

At this stage of creating the subkeys, we get our left and right values from above called C0 and D0, and create C1..C16 and D1..D16. Each time we create the next C/D pair, we do a left-shift either one or two times. The list of shifts is this: 1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1. To make it easier for the next stage, at the end of this function, rather than returning a list of C’s and a list of D’s, we concatenate them to form CD0,CD1.

# Performs the shifts to produce CnDn

def shifts(c0,d0)

cn, dn = [c0], [d0]

# This is the schedule of shifts. Each CnDn is produced by shifting the previous by 1 or 2 bits

[1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1].each{|n|

cn << cn.last.left(n)

dn << dn.last.left(n)

}

cdn=[]

cn.zip(dn) {|c,d| cdn << (c+d)} # Concatenate the c's and d's to produce CDn

cdn

end

class Array

# shift this array one or two bits left

def left(n)

self[n,self.length] + self[0,n]

end

end

## PC2 permutation

Once we’ve got our list of CD0..CD16, we need to run each of them through the PC2 permutation to give us the subkeys K0..K16. This permutation takes the 56-bit CDn and produces the 48 bit Kn:

class Array

# Perform the PC2 permutation on the current array

# This is used on each of the 56 bit "CnDn" concatenated pairs to produce

# each of the 48 bit "Kn" keys

def pc2

perm "

14 17 11 24 1 5

3 28 15 6 21 10

23 19 12 4 26 8

16 7 27 20 13 2

41 52 31 37 47 55

30 40 51 45 33 48

44 49 39 56 34 53

46 42 50 36 29 32"

end

end

## Testing the key expansion:

Let’s test that the subkeys are made correctly:

# Step 1, make the subkeys

k = '00010011 00110100 01010111 01111001 10011011 10111100 11011111 11110001'.to\_bits # This is the key

subkeys = expand(k)

puts "Key: " + k.pretty(8)

subkeys.each\_with\_index { |sk,i|

puts "Subkey %2d: %s" % [i,sk.pretty(6)]

}

Your subkey #16 should be:

110010 110011 110110 001011 000011 100001 011111 110101

# Encrypting

Now we’ve got the subkeys, we can perform the encryption steps:

* Perform the IP permutation on the message
* Split the results into left and right, giving you L0 and R0
* Do 16 encryption rounds with the keys
  + Ln => Rn-1
  + Rn => Ln-1 + f(Rn-1,Kn)
    - Note that ‘+’ in the mathematical formula above really means Xor in implementation
* Swap and concatenate the two sides into R16L16
* Perform the IP-1 permutation to give the result

In code, this looks like:

# Take a 8 byte message and the expanded keys, and des encrypt it

def des\_encrypt(m,keys)

ip = m.ip # Run it through the IP permutation

l, r = ip.split # Split it to give us L0R0

(1..16).each { |i| # Run the encryption rounds

l, r = r, l.xor(f(r,keys[i])) # L => R, R => L + f(Rn-1,Kn)

}

rl = r + l # Swap and concatenate the two sides into R16L16

c = rl.ip\_inverse # Run IP-1(R16L16) to give the final "c" cryptotext

end

Now we’ll need to implement the new functions introduced in the encrypt function above.

## IP permutation

The IP permutation takes 64 bits, reorders them, and outputs 64 bits. The 58th bit becomes the 1st bit, etc:

class Array

# This performs the initial permutation aka "IP"

# This is the first thing applied to the 64 bit message "M" to give us "IP"

# Inputs 64 bits, outputs 64 bits

def ip

perm "

58 50 42 34 26 18 10 2

60 52 44 36 28 20 12 4

62 54 46 38 30 22 14 6

64 56 48 40 32 24 16 8

57 49 41 33 25 17 9 1

59 51 43 35 27 19 11 3

61 53 45 37 29 21 13 5

63 55 47 39 31 23 15 7"

end

end

## Xor

We need a way to XOR the previous left bits with the results of the f(…) function:

class Array

# xor's this and the other array

def xor(b)

i=0

self.map{|a| i+=1; a^b[i-1]}

end

end

## The f(…) function

This function is probably the most complicated part of the operation. It takes the right side bits, and one of the round keys, and performs the following:

* Performs the E-bit permutation on R
* Xors the above result with the subkey
* Splits the above result into 8 groups of 6 bits
* Puts each 6-bit value through substitution -boxes to get 4 bit results
* Concatenate the 8 \* 4 bit results
* Perform the P permutation on the above result

In code, it looks like this:

# The 'f' function as used in the encryption rounds

# For each round, we want: Rn = Ln-1 + f(Rn-1,Kn)

# f(Rn-1,Kn) is to be calculated like this:

# Kn + E(Rn-1) => B1..B8

# f = P( S1(B1)..S8(B8) )

def f(r,k)

e = r.e\_bits # Calculate E(Rn-1)

x = e.xor(k) # Calculate Kn + E(Rn-1)

bs = x.split6 # Split into B1..B8

s = [] # Concatenate S1(B1)..S8(B8)

bs.each\_with\_index{|b,i| s += b.s\_box(i+1)}

s.perm\_p # Calculate P(S1..S8)

end

## E Bit permutation

The E-Bit permutation expands 32bits input to 48 bits output by repeating some bits. The 32nd bit becomes the first bit, and so on:

class Array

# This is the E-Bit selection table

# It inputs 32 bits and outputs 48 bits

# This is used in the 'f' function to calculate "E(Rn-1)" on each of the rounds

def e\_bits

perm "

32 1 2 3 4 5

4 5 6 7 8 9

8 9 10 11 12 13

12 13 14 15 16 17

16 17 18 19 20 21

20 21 22 23 24 25

24 25 26 27 28 29

28 29 30 31 32 1"

end

end

## Split6

This function takes the 48-bit output of Kn + E(Rn-1) and splits it into 8 groups of 6 bits, ready to be fed to the s-boxes:

class Array

# splits into arrays of 6 bits

def split6

arr=[]

subarr=[]

self.each{|a|

subarr<<a

if subarr.length==6

arr<<subarr

subarr=[]

end

}

arr

end

end

## Substitution Boxes

Each of the 8 groups of 6 bits, in turn, gets fed into an S-Box. There are 8 different S-boxes, one for each group. The result is looked up, and converted to 4 bits then returned. An S-Box looks like this:

S7

4 11 2 14 15 0 8 13 3 12 9 7 5 10 6 1

13 0 11 7 4 9 1 10 14 3 5 12 2 15 8 6

1 4 11 13 12 3 7 14 10 15 6 8 0 5 9 2

6 11 13 8 1 4 10 7 9 5 0 15 14 2 3 12

The first and last bits of the 6-bit group represent the row, and the middle 4 bits represent the column in the S-box. So if the bits were 100001, the row is 11 = 3+1 = 4th row (+1 because it is 0-based), and the middle bits represent column 0, and so the S-box output is 6 as you can see in the table above. The S-box function looks like this:

class Array

# The S-Box lookup

# This takes the 6 bits input and produces 4 bits output

# The 'b' variable is which s-box table to use

# This is used in the 'f' function. "Kn+E(Rn-1)" is calculated then split

# into 6-bit blocks B1..B8, each of which is passed through the s-box S1..S8

def s\_box(b)

s\_tables = "

S1

14 4 13 1 2 15 11 8 3 10 6 12 5 9 0 7

0 15 7 4 14 2 13 1 10 6 12 11 9 5 3 8

4 1 14 8 13 6 2 11 15 12 9 7 3 10 5 0

15 12 8 2 4 9 1 7 5 11 3 14 10 0 6 13

S2

15 1 8 14 6 11 3 4 9 7 2 13 12 0 5 10

3 13 4 7 15 2 8 14 12 0 1 10 6 9 11 5

0 14 7 11 10 4 13 1 5 8 12 6 9 3 2 15

13 8 10 1 3 15 4 2 11 6 7 12 0 5 14 9

S3

10 0 9 14 6 3 15 5 1 13 12 7 11 4 2 8

13 7 0 9 3 4 6 10 2 8 5 14 12 11 15 1

13 6 4 9 8 15 3 0 11 1 2 12 5 10 14 7

1 10 13 0 6 9 8 7 4 15 14 3 11 5 2 12

S4

7 13 14 3 0 6 9 10 1 2 8 5 11 12 4 15

13 8 11 5 6 15 0 3 4 7 2 12 1 10 14 9

10 6 9 0 12 11 7 13 15 1 3 14 5 2 8 4

3 15 0 6 10 1 13 8 9 4 5 11 12 7 2 14

S5

2 12 4 1 7 10 11 6 8 5 3 15 13 0 14 9

14 11 2 12 4 7 13 1 5 0 15 10 3 9 8 6

4 2 1 11 10 13 7 8 15 9 12 5 6 3 0 14

11 8 12 7 1 14 2 13 6 15 0 9 10 4 5 3

S6

12 1 10 15 9 2 6 8 0 13 3 4 14 7 5 11

10 15 4 2 7 12 9 5 6 1 13 14 0 11 3 8

9 14 15 5 2 8 12 3 7 0 4 10 1 13 11 6

4 3 2 12 9 5 15 10 11 14 1 7 6 0 8 13

S7

4 11 2 14 15 0 8 13 3 12 9 7 5 10 6 1

13 0 11 7 4 9 1 10 14 3 5 12 2 15 8 6

1 4 11 13 12 3 7 14 10 15 6 8 0 5 9 2

6 11 13 8 1 4 10 7 9 5 0 15 14 2 3 12

S8

13 2 8 4 6 15 11 1 10 9 3 14 5 0 12 7

1 15 13 8 10 3 7 4 12 5 6 11 0 14 9 2

7 11 4 1 9 12 14 2 0 6 10 13 15 3 5 8

2 1 14 7 4 10 8 13 15 12 9 0 3 5 6 11

"

# Find only the table they want

s\_table = s\_tables[s\_tables.index('S%d'%b)+3,999]

s\_table = s\_table[0,s\_table.index('S')] if s\_table.index('S')

s\_table = s\_table.split(' ') # Convert from text to usable array

row = self.first\*2 + self.last # The row is found from the first and last bits

col = self[1]\*8 + self[2]\*4 + self[3]\*2 + self[4] # The column is from the middle 4 bits

s\_table[row\*16+col].to\_i.to\_bits # The correct value is looked up, then converted to 4 bits output

end

end

class Integer

# Converts an integer into a 4-bit array, as used by the s-boxes

def to\_bits

[self>>3, (self>>2)&1, (self>>1)&1, self&1]

end

end

## P Permutation

Once all the groups of 6 bits have been passed through the s-boxes to yield 8 groups of 4 bits, they are concatenated to form 32 bits. These 32 bits are passed through the P permutation, and the 32 bits output is returned as the result of the f(…) function. Here is the code, where you can see the 16th bit becomes the 1st bit, and so on:

class Array

# The P permutation

# Inputs 32 bits, outputs 32 bits

# At the end of the 'f' function, this is run on the concatenated results of the s-boxes

def perm\_p

perm "

16 7 20 21

29 12 28 17

1 15 23 26

5 18 31 10

2 8 24 14

32 27 3 9

19 13 30 6

22 11 4 25"

end

end

## Inverse IP Permutation

After using all the subkeys, R and L are concatenated and passed through the IP-1 permutation and returned as the final encrypted output. The 40th bit becomes the 1st bit, etc:

class Array

# The IP^-1 final permutation

# Inputs 64 bits, outputs 64 bits

# At the end of the rounds, this is run over "R16L16" to produce the final result

def ip\_inverse

perm "

40 8 48 16 56 24 64 32

39 7 47 15 55 23 63 31

38 6 46 14 54 22 62 30

37 5 45 13 53 21 61 29

36 4 44 12 52 20 60 28

35 3 43 11 51 19 59 27

34 2 42 10 50 18 58 26

33 1 41 9 49 17 57 25"

end

end

## Testing the encryption

OK now let’s test the encryption:

# Step 2, encode it

m = '0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111'.to\_bits # The message to encode

c = des\_encrypt(m,subkeys)

puts "Encrypt: " + c.pretty(8) # The output value

You should get the following result:

10000101 11101000 00010011 01010100 00001111 00001010 10110100 00000101

# Decryption

Decryption is performed the same as encryption, except the subkeys are applied in reverse order:

# Take a 8 byte message and the expanded keys, and des decrypt it

def des\_decrypt(m,keys)

ip = m.ip # Run it through the IP permutation

l, r = ip.split # Split it to give us L0R0

(1..16).to\_a.reverse.each { |i| # Run the encryption rounds

l, r = r, l.xor(f(r,keys[i])) # L => R, R => L + f(Rn-1,Kn)

}

rl = r + l # Swap and concatenate the two sides into R16L16

c = rl.ip\_inverse # Run IP-1(R16L16) to give the final "c" cryptotext

end

Let’s test this:

# Step 3, decode it

d = des\_decrypt(c,subkeys)

puts "Decrypt: " + d.pretty(8) # The output value

You should get the following result:

00000001 00100011 01000101 01100111 10001001 10101011 11001101 11101111

# Triple DES

Triple DES is a variant of DES that uses two DES keys that are applied three times (hence the triple), on a standard 64 bit DES message block. When encrypting, it does a (single) DES encrypt with the first key, DES decrypt with the second key, then encrypt again with the first key. Decrypting is simple the reverse:

# Takes a 128-bit TripleDES key, and encrypts a 64-bit message with it

def tripledes\_encrypt(m, key)

key\_a, key\_b = key.split # Split the 128-bit TripleDES key into two DES keys

keys\_a = expand(key\_a) # Expand the two DES keys

keys\_b = expand(key\_b)

c = des\_encrypt(m, keys\_a) # Encrypt by the first key

c = des\_decrypt(c, keys\_b) # Decrypt by the second key

c = des\_encrypt(c, keys\_a) # Encrypt by the first key again

end

# Takes a 128-bit TripleDES key, and decrypts a 64-bit message with it

def tripledes\_decrypt(c, key)

key\_a, key\_b = key.split # Split the 128-bit TripleDES key into two DES keys

keys\_a = expand(key\_a) # Expand the two DES keys

keys\_b = expand(key\_b)

c = des\_decrypt(c, keys\_a) # Encrypt by the first key

c = des\_encrypt(c, keys\_b) # Decrypt by the second key

c = des\_decrypt(c, keys\_a) # Encrypt by the first key again

end

So lets test this:

k3d = '00010001 00100010 00110011 01000100 01010101 01100110 01110111 10001001

10000111 10011000 01111001 01000101 00110101 00100001 00110101 01000100'.to\_bits

m3d = '00010010 00110100 01010110 01111000 10010000 10101011 11001101 11101111'.to\_bits

r3d = '00111010 00111010 11001110 01100101 00001101 10110011 10111011 11011100'.to\_bits

e = tripledes\_encrypt(m3d,k3d)

d = tripledes\_decrypt(e,k3d)

puts "Triple Des Message: " + m3d.pretty(8)

puts "Triple Des Encrypt: " + e.pretty(8)

puts "Triple Des Decrypt: " + d.pretty(8)

Your results should be:

Triple Des Message: 00010010 00110100 01010110 01111000 10010000 10101011 11001101 11101111

Triple Des Encrypt: 00111010 00111010 11001110 01100101 00001101 10110011 10111011 11011100

Triple Des Decrypt: 00010010 00110100 01010110 01111000 10010000 10101011 11001101 11101111

That’s it, thanks for reading! The full code can be found here: <http://github.com/chrishulbert/crypto/blob/master/ruby/ruby_des.rb>