Project 3: Side Channel Analysis of an Embedded Crypto Device

Jason Gramse, David Ware

***Abstract*—This paper demonstrates a successful Correlation Power Analysis (CPA) attack on an embedded crypto-processor running a DES encryption algorithm. The goal of the attack was to extract the 64-bit encryption key by analyzing power traces of the device that were taken during the encryption. To minimize the number of traces used, we implemented two improvements over the standard CPA attack.**

I. Introduction

Encryption algorithms are used to obscure information using a secret key in such a way that it can't be deciphered without the key. Very little information can be inferred about the plain-text from the cypher-text. So instead of attacking the encryption algorithm directly it is often faster and easier to attack it indirectly using side channel attacks. One side channel attack is correlation power analysis (CPA). CPA uses the power consumed by a device during encryption to infer information about the key that is being used to do the encryption. To do this we assume that we first have physical access to the device being attacked. Second that we can encrypt many plain texts using the device and collect information about the power consumption of the device while doing this. We are using the DPA contest traces to demonstrate our attack. In this paper we will, briefly review the DES encryption algorithm, explain our CPA attack and our improvements, and present the results of our implementation.

II. DES Encryption

DES encrypts 64 bits of a plain text at a time with a 64 bit key using a feistel structure. At the beginning of the encryption the plain-text goes through a series of permutations that mix up the bits and then divide them into a left and right half. These are then encrypted using the feistel function. A graphical representation of the feistel function used in DES can be seen in FIGURE 1. The left half of the output is the right half of the input. However, the right half output is more complicated. First, the right half of the input is expanded to 48 bits. This is done by splitting the 32-bit word into 4-bit sections then appending the last bit of the previous word to the beginning and the first bit of the next to the end. This 48-bit number is then combined with the round key using a bit-wise exclusive or. The source of the round key will be explained latter. Then this output is broken up into eight 6-bit blocks which are put into 8 different substitution tables (s-boxes). These s-boxes have four rows and 8 columns each corresponding with a 4-bit number. The first and last bit of the input indicate the row and the rest of the bits indicate the column. The outputs from the s-boxes are then combined with the left half of the message using a bit-wise exclusive or. This produces the right half of the output for each round. This is done 16 times to produce the completed DES encryption.

J. Gramse and D. Ware are with the Electrical and Computer Engineering College at Utah State University.

Manuscript received April 4, 2016;

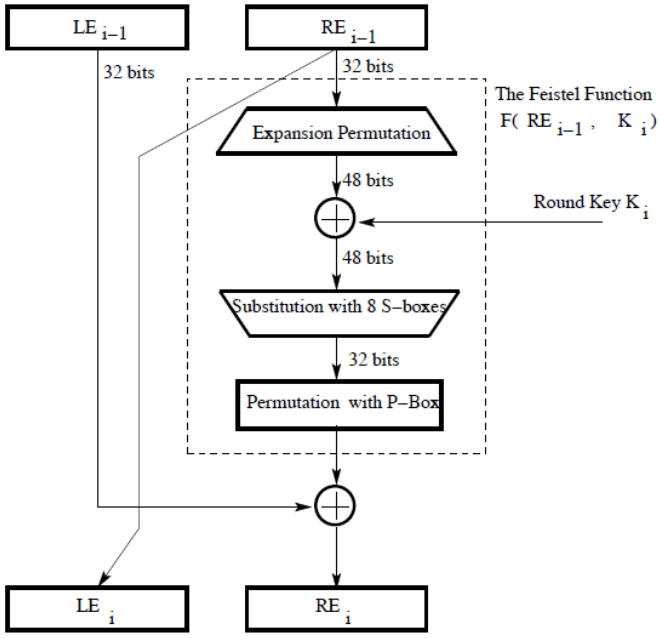

Fig. 1. DES Round

Each round key is generated from the 64 bit key. This key contains 8 bits of parity information which are removed from the key when it goes through the first permutation which also mixes up the bits. This 56-bit output is then divided up into two 28-bit sets which are circularly shifted left once or twice depending on the round. The two parts of the 56-bit key are then combined together again and 48-bits are selected using another permutation which again mixes up the bits to generate a round key. This process is repeated on the 56-bit output to generate each 48-bit round key.

III. CPA Method

Our baseline CPA implementation uses the Hamming Distance model to correlate actual power used with the estimated power used one s-box at a time. The estimated power is based on the hamming distance of the output of each s-box with the original message. For each trace, the plain-text is extracted as well as the max power for round 1 of DES. The plain-text

is sent through the initial permutation and broken into right and left halves. We then extract the bits used for the hamming distance from the right and left halves. The right half is then sent through the expansion permutation to increase from 32-bits to 48-bits and the 6 bits corresponding to our s-box are extracted from the expansion.

Once we have the correct bits of the right half, left half, and expanded right half, we iterate through all 64 combinations of keys for the s-box we are are attacking and xor them with the expanded right half bits. The result is sent through the s-box routine and the output is then xored with the bits of the left half to achieve the final 4 bits of the first round. Our hamming distance used for correlation is the hamming distance between the extracted bits of the right half, and the 4 bits of the last round.

After the hamming distance has been found for all 64 possible keys, we obtain the Pearson's Correlation Coefficient between the hamming distance, and the maximum power of round 1. Theoretically the best key guess is the one with the greatest correlation coefficient; however, multiple traces are necessary to ensure the correct key has been found. The process is repeated with another trace, new hamming distance values calculated for all 64 key guess and appended on to the previous traces hamming distance data, and the correlation coefficient calculated again until the same key has the highest correlation coefficient for 100 consecutive rounds. The whole process then starts over for each s-box until we have all 8 6-bit keys used for each s-box. We reuse all traces that have been used previously and only add more if they are necessary to find the correct key. This allows us to use as few traces as possible.

The eight s-box keys are concatenated together to complete our 48-bit round key. This key, however, is only good for the first round. To complete the key, we need to send it through a de-permutation of Permutation Choice 2 to reorder the bits to the order they were in before the permutation. The problem with this is that we are missing 6 bits of the 56 bit key. In our implementation, these bits are found by brute force. They are initially set to zero. The plain-text of the last trace and the assumed key is then sent through a complete DES cycle and the output is compared with the cypher-text of the last trace. If they are the same, we have our 56 bit key, otherwise, we brute force every combination of those 6 bits until the correct key is found.

Our initial key is 64 bits. Similar to the previous step, the 56 bit key is sent through a de-permutation of Permutation Choice 1. The missing bits of this permutation are simple to compute using a basic even parity computation.

IV. Improvements

The improvements implemented were designed to decrease the total number of traces used to recover the key. Both of the improvements focus on increasing the correlation of our guess to the actual power traces and are explained below.

A. Including Left half of the Register

Since CPA is based on statistics it is vulnerable to noise. In this case we are correlating the power consumption of 4 bits to the power consumption to write 64 bits. There is a correlation and with enough information we can find the correct guess despite this noise. In order to reduce the noise we took into account the Left half of the message. By doing a bit wise exclusive or of the left half of the message with the right half we were able to obtain the hamming distance of this register write. We did this with every plain text that we had and then added this to the predicted hamming distance of the s-box key guess. The goal was to reduce the amount of noise and increase the correlation of the correct key guess making it easier to find with less information.

B. Including already found S-box Information

Similar to the first improvement, the goal here was to increase the correlation for each guess to bring down the total number of traces necessary to recover the key. The idea is that once we have obtained the correct guess for our previous s-box key, we add the hamming distance of that key to the guess for our next s-box, thereby increasing the correlation.

V. Results

Our base implementation of CPA (see Appendix A) recovered the key with an average number of traces equal to 335.25. And there were some sequences of trace files that the base implementation failed to produce the correct key. With the improvements made to increase correlation, we were able to bring the average number of traces necessary down to 283.25. We also did not see any of the failures to produce a key that we had with the base CPA implementation in the improved results.

The base implementation correlation was quite low as you can see from the output of our program. The correlation is printed after the correct key had the max correlation for 100 iterations.

```
Base
Correlation: 0.0888176207806
S-box1 guess found: 56, Number of traces: 132
Correlation: -0.0625413974538
S-box2 guess found: 11, Number of traces: 323
Correlation: -0.0609469215379
S-box3 guess found: 59, Number of traces: 264
Correlation: 0.0308503484245
S-box4 guess found: 38, Number of traces: 253
Correlation: 0.00828225130913
S-box5 guess found: 0, Number of traces: 236
Correlation: -0.010902259489
S-box6 guess found: 13, Number of traces: 109
Correlation: -0.038710129434
S-box7 guess found: 25, Number of traces: 231
Correlation: -0.15976295133
S-box8 guess found: 55, Number of traces: 146
Subkey: 0xbfdd248926d4L
Success!
Final Key: 0x6a65786a65786a65L
```

Our improved implementation increased the correlation significantly. The first S-box starts out around 0.3. This is with the hamming distance of the left half added to the correlation. The correlation increases as each S-box is computed, showing that both our improvements did in fact increase the correlation coefficient, reduce the number of traces necessary, and increase the accuracy of results.

Improved:
Correlation: 0.313258216046
S-box1 guess found: 56, Number of traces: 128
Correlation: 0.377213149391
S-box2 guess found: 11, Number of traces: 179
Correlation: 0.431963142778
S-box3 guess found: 59, Number of traces: 268
Correlation: .496138019702
S-box4 guess found: 38, Number of traces: 254
Correlation: 0.549043921899
S-box5 guess found: 0, Number of traces: 254
Correlation: 0.468529881696
S-box6 guess found: 13, Number of traces: 108
Correlation: 0.579568890826
S-box7 guess found: 25, Number of traces: 165
Correlation: 0.592664727543
S-box8 guess found: 55, Number of traces: 143
Subkey = 0xbfdd248926d4L
Success!
Final Key: 0x6a65786a65786a65L

VI. Conclusion

Implementing CPA on the DPA contest traces is straightforward. This attack on the implementation significantly decreases the amount of brute forcing that was required to obtain the key. Basic CPA by a single s-box certainly decreases the number of possibilities a brute force method would require. Unfortunately it is not always accurate with its guesses because of the low correlation in the data and its statistical nature. Increasing the correlation of the data using the left half of the register write and previous guessed s-boxes increases the accuracy and speed in most cases. Optimizing the order that the s-boxes are found would help this method as well.

APPENDIX A
BASIC CPA CODE

```
import math
import operator
import os

import numpy as np
from itertools import imap

from DES_enc_bit import *

def get_trace_files(start, end):
    #traces = []
    for csv_file in os.listdir(r'c:\users\vip\desktop\csv'): #[start:end]:
        m = csv_file.find('m=')+2
        message = csv_file[m:m+16]
        c = csv_file.find('c=')+2
        cyphertext = csv_file[c:c+16]
        with open(os.path.join(r'c:\users\vip\desktop\csv', csv_file)) as tracefile:
            content = tracefile.read()
        content = content.split('\n')
        try:
            content.remove('')
        except Exception:
            pass
        tuplepairs = []
        tuplepairs.append((message, cyphertext))
        for entry in content:
            a, b = entry.split(', ')
            tuplepairs.append((float(a), float(b)))
        yield tuplepairs
    #return traces

def HD(num1, num2):
    """input is two integers. output is hamming distance as an integer"""
    num1 = bin(num1).replace('0b','').zfill(64) # 64 bit representation of num1
    num2 = bin(num2).replace('0b','').zfill(64) # 64 bit representation of num2
    return sum(imap(operator.ne, num1, num2))

def break_encryption():
    start = 0
    end = 500
    correlation = {}
    best_guess = 0 # arbitrary value
    prev_guess = 65 # arbitrary value
    consecutive_guesses = 0
    keys = {}
    for box in range(1, 9):
        # for each S-box, find greatest power correlation
        count = 0 # keeps track of how many traces we've used for each S-box
        max_power = []
        hamming_distances = [[] for i in range(64)] # list of 64 empty lists
        for trace in get_trace_files(start, end):
            count += 1
            # initial plaintext of trace file
            plaintxt = trace[0][0]
            max_power.append(max([t[1] for t in trace[5700:5800]]))
            # send plain text through initial permutation
            mixtxt = Initial_permutation(int(plaintxt,16))
            r_0 = mixtxt & 0xffffffff
            # expand right half of plaintxt
            r_0exp = expansion_permuatation(r_0)
            exp_bits = (r_0exp & (0xfc0000000000 >> ((box - 1) * 6))) >> ((8-box) * 6)
            r_0 = bin(r_0).replace('0b', '').zfill(32)
            l_0 = bin((mixtxt >> 32) & 0xffffffff).replace('0b', '').zfill(32)
            num_r0, num_l0 = get_bits(r_0, l_0, box)
            for key_guess in range(2**6):
                xored = key_guess ^ exp_bits
                S-box_output = S-box(box, xored)
                # compute the bits of r_1 we care about
                num_r1 = S-box_output ^ num_l0
                # store the hamming distance between r_0 and r_1
                hamming_distances[key_guess].append(HD(num_r1, num_r0))
            if count > 4: # just to get enough data to do any meaningful correlation
                best_guess = correlate(hamming_distances, max_power)
                if best_guess == prev_guess:
                    consecutive_guesses += 1
                else:
                    consecutive_guesses = 0
                if consecutive_guesses >= 100:
                    print "S-box%s guess found: %s, Number of traces: %s" % (
                        box, best_guess, count)
                    keys['S-box%s' % box] = best_guess
                    break
                else:
                    prev_guess = best_guess
                #correlation['S-box%s' % box].append(correlate(hamming_distances, trace[1:]
    key = 0
    for i in range(1, 9):
        key |= keys["S-box%s" % i]
        key = key << 6
    key = de_permute_c2(key)
    print "Subkey: %s" % hex(int(key, 2))
    missing_bit_index = [9, 18, 22, 25, 35, 38, 43, 54]
    crypt_txt = int(trace[0][1], 16)
    encrypted_msg = DES_enc_56(int(plaintxt, 16), int(key, 2))
    if crypt_txt == encrypted_msg:
        print "Success!"
    else:
        for a in range(2):
            for b in range(2):
                for c in range(2):
                    for d in range(2):
                        for e in range(2):
                            for f in range(2):
                                for g in range(2):
                                    for h in range(2):
                                        key = key[:9] + str(a) \
                                            + key[10:18] + str(b) \
                                            + key[19:22] + str(c) \
                                            + key[23:25] + str(d) \
                                            + key[26:35] + str(e) \
                                            + key[36:38] + str(f) \
                                            + key[39:43] + str(g) \
                                            + key[44:54] + str(h) \
                                            + key[55:]
                                        encrypted_msg = DES_enc_56(int(plaintxt, 16),
                                                                   int(key, 2))
                                        if crypt_txt == encrypted_msg:
                                            print "Success!"
                                            final_key = de_permute_c1(key)
                                            print "Final Key: %s" % hex(int(final_key, 2))

def de_permute_c1(key):
    permutation = [8, 16, 24, 56, 52, 44, 36, 0,
                   7, 15, 23, 55, 51, 43, 35, 0,
                   6, 14, 22, 54, 50, 42, 34, 0,
                   5, 13, 21, 53, 49, 41, 33, 0,
                   4, 12, 20, 28, 48, 40, 32, 0,
                   3, 11, 19, 27, 47, 39, 31, 0,
                   2, 10, 18, 26, 46, 38, 30, 0,
                   1, 9,  17, 25, 45, 37, 29, 0]
    depermuted_key = ''
    for bit in permutation:
        if bit == 0:
            depermuted_key += str(depermuted_key[len(depermuted_key) - 7:].count('1') % 2)
        else:
            depermuted_key += key[bit-1]
    return depermuted_key

def de_permute_c2(key):
    permutation = [ 5, 24,  7, 16,  6, 10, 20, 18,  0,
                   12,  3, 15, 23,  1,  9, 19,  2,  0,
                   14, 22, 11,  0, 13,  4,  0, 17, 21,
                    8, 47, 31, 27, 48, 35, 41,  0, 46,
                   28,  0, 39, 32, 25, 44,  0, 37, 34,
                   43, 29, 36, 38, 45, 33, 26, 42,  0,
                   30, 40]
    key = bin(key).replace('0b', '').zfill(48)
    depermuted_key = ''
    for bit in permutation:
        if bit == 0:
            depermuted_key += '0'
        else:
            depermuted_key += key[bit-1]
    depermuted_key = int(depermuted_key, 2)
    key_r = depermuted_key & 0xffffffff
    key_l = (depermuted_key >> 28) & 0xffffffff
    upper_r = (0x1 & key_r) << 27
    upper_l = (0x1 & key_l) << 27
    key_r = (key_r >> 1) & 0xffffffff
    key_l = (key_l >> 1) & 0xffffffff
    key_r = key_r | upper_r
    key_l = key_l | upper_l
    depermuted_key = (key_l << 28) | key_r
    return bin(depermuted_key).replace('0b','').zfill(56)

def get_bits(right, left, box):
    """input is string representation of 32 bit binary numbers. the bits are
    rearranged to correspond to the output of the S-box after its sent through
    the pbox. numbers are then converted to integers and returned"""
    if box == 1:
        r_bits = right[8] + right[16] + right[22] + right[30]
        r_bits = int(r_bits, 2)
        l_bits = left[8] + left[16] + left[22] + left[30]
        l_bits = int(l_bits, 2)
    elif box == 2:
        r_bits = right[12] + right[27] + right[1] + right[17]
        r_bits = int(r_bits, 2)
        l_bits = left[12] + left[27] + left[1] + left[17]
        l_bits = int(l_bits, 2)
    elif box == 3:
        r_bits = right[23] + right[15] + right[29] + right[5]
        r_bits = int(r_bits, 2)
        l_bits = left[23] + left[15] + left[29] + left[5]
        l_bits = int(l_bits, 2)
    elif box == 4:
        r_bits = right[25] + right[19] + right[9] + right[0]
        r_bits = int(r_bits, 2)
        l_bits = left[25] + left[19] + left[9] + left[0]
        l_bits = int(l_bits, 2)
    elif box == 5:
        r_bits = right[7] + right[13] + right[24] + right[2]
        r_bits = int(r_bits, 2)
        l_bits = left[7] + left[13] + left[24] + left[2]
        l_bits = int(l_bits, 2)
    elif box == 6:
        r_bits = right[3] + right[28] + right[10] + right[18]
        r_bits = int(r_bits, 2)
        l_bits = left[3] + left[28] + left[10] + left[18]
        l_bits = int(l_bits, 2)
    elif box == 7:
        r_bits = right[31] + right[11] + right[21] + right[6]
        r_bits = int(r_bits, 2)
        l_bits = left[31] + left[11] + left[21] + left[6]
        l_bits = int(l_bits, 2)
    else:
        r_bits = right[4] + right[26] + right[14] + right[20]
        r_bits = int(r_bits, 2)
        l_bits = left[4] + left[26] + left[14] + left[20]
        l_bits = int(l_bits, 2)
    return r_bits, l_bits

def correlate(hamming_distances, max_power):
    correlations = {}
    for key_guess in range(64):
        correlations[key_guess] = np.corrcoef(hamming_distances[key_guess], max_power)[0, 1]
    best_guess = max(correlations, key=correlations.get)
    #print best_guess
    return best_guess

break_encryption()
```

APPENDIX B
IMPROVED CPA

Changes to the basic CPA code are shown below in between lines of # symbols for your benefit.

```
import math
import operator
import os

import numpy as np
from itertools import imap

from DES_enc_bit import *

def get_trace_files(start, end):
    #traces = []
    for csv_file in os.listdir(r'c:\users\vip\desktop\csv'): #[start:end]:
        m = csv_file.find('m=')+2
        message = csv_file[m:m+16]
        c = csv_file.find('c=')+2
        cyphertext = csv_file[c:c+16]
        with open(os.path.join(r'c:\users\vip\desktop\csv', csv_file)) as tracefile:
            content = tracefile.read()
        content = content.split('\n')
        try:
            content.remove('')
        except Exception:
            pass
        tuplepairs = []
        tuplepairs.append((message, cyphertext))
        for entry in content:
            a, b = entry.split(', ')
            tuplepairs.append((float(a), float(b)))
        yield tuplepairs
    #return traces

def HD(num1, num2):
    """input is two integers. output is hamming distance as an integer"""
    num1 = bin(num1).replace('0b','').zfill(64) # 64 bit representation of num1
    num2 = bin(num2).replace('0b','').zfill(64) # 64 bit representation of num2
    return sum(imap(operator.ne, num1, num2))

def break_encryption():
    start = 0
    end = 500
    correlation = {}
    best_guess = 0
    prev_guess = 65
    consecutive_guesses = 0
    keys = {}
    for box in range(1, 9):
        # for each S-box, find greatest power correlation from 500 traces
        #traces = get_trace_files(start, end)
        count = 0
        max_power = []
        hamming_distances = [[] for i in range(64)]
        for trace in get_trace_files(start, end):
            count += 1
            # initial plaintext of trace file
            plaintxt = trace[0][0]
            max_power.append(max([t[1] for t in trace[5700:5800]]))
            # send plain text through initial permutation
            mixtxt = Initial_permutation(int(plaintxt,16))
            r_0 = mixtxt & 0xffffffff
            # expand right half of plaintxt
            r_0exp = expansion_permuatation(r_0)
            exp_bits = (r_0exp & (0xfc0000000000 >> ((box - 1) * 6))) >> ((8-box) * 6)
            r_0 = bin(r_0).replace('0b', '').zfill(32)
            l_0 = bin((mixtxt >> 32) & 0xffffffff).replace('0b', '').zfill(32)
            ########################################################
            left_half_HD = HD(int(r_0, 2), int(l_0, 2))
            ########################################################
            num_r0, num_l0 = get_bits(r_0, l_0, box)
            for key_guess in range(2**6):
                xored = key_guess ^ exp_bits
                S-box_output = S-box(box, xored)
                ######################################################################
                additional_hd = calculate_additional(keys, box, r_0exp, r_0, l_0)
                ######################################################################
                # compute the bits of r_1 we care about
                num_r1 = S-box_output ^ num_l0
                # store the hamming distance between r_0 and r_1
                ######################################################################
                hamming_distances[key_guess].append(HD(num_r1, num_r0) + \
                    left_half_HD + additional_hd)
                ######################################################################
            if count > 4:
                best_guess = correlate(hamming_distances, max_power, count)
                if best_guess == prev_guess:
                    consecutive_guesses += 1
                else:
                    consecutive_guesses = 0
                if consecutive_guesses >= 100:
                    print "S-box%s guess found: %s, Number of traces: %s" % (
                        box, best_guess, count)
                    keys['S-box%s' % box] = best_guess
                    break
                else:
                    prev_guess = best_guess
    key = 0
    for i in range(1, 9):
        key |= keys["S-box%s" % i]
        key = key << 6
    key = de_permute_c2(key)
    print "Subkey = %s" % hex(int(key, 2))
    missing_bit_index = [9, 18, 22, 25, 35, 38, 43, 54]
    crypt_txt = int(trace[0][1], 16)
    encrypted_msg = DES_enc_56(int(plaintxt, 16), int(key, 2))
    if crypt_txt == encrypted_msg:
        print "Success!"
    else: # brute force missing bits
        for a in range(2):
            for b in range(2):
                for c in range(2):
                    for d in range(2):
                        for e in range(2):
                            for f in range(2):
                                for g in range(2):
                                    for h in range(2):
                                        key = key[:9] + str(a) \
                                            + key[10:18] + str(b) \
                                            + key[19:22] + str(c) \
                                            + key[23:25] + str(d) \
                                            + key[26:35] + str(e) \
                                            + key[36:38] + str(f) \
                                            + key[39:43] + str(g) \
                                            + key[44:54] + str(h) \
                                            + key[55:]
                                        encrypted_msg = DES_enc_56(int(plaintxt, 16),
                                                                   int(key, 2))
                                        if crypt_txt == encrypted_msg:
                                            print "Success!"
                                            final_key = de_permute_c1(key)
                                            print "Final Key: %s" % hex(int(final_key, 2))

##############################################################################
def calculate_additional(key, box, r_0exp, r_0, l_0):
    additional_hd = 0
    for b in range(1, box):
        num_r0, num_l0 = get_bits(r_0, l_0, b)
        exp_bits = (r_0exp & (0xfc0000000000 >> ((b - 1) * 6))) >> ((8-b) * 6)
        xored = key['S-box%s' % b] ^ exp_bits
        S-box_output = S-box(b, xored)
        num_r1 = S-box_output ^ num_l0
        additional_hd += HD(num_r0, num_r1)
    return additional_hd
##############################################################################

def de_permute_c1(key):
    permutation = [8, 16, 24, 56, 52, 44, 36, 0,
                   7, 15, 23, 55, 51, 43, 35, 0,
                   6, 14, 22, 54, 50, 42, 34, 0,
                   5, 13, 21, 53, 49, 41, 33, 0,
                   4, 12, 20, 28, 48, 40, 32, 0,
                   3, 11, 19, 27, 47, 39, 31, 0,
                   2, 10, 18, 26, 46, 38, 30, 0,
                   1, 9,  17, 25, 45, 37, 29, 0]
    depermuted_key = ''
    for bit in permutation:
        if bit == 0:
            depermuted_key += str(depermuted_key[len(depermuted_key) - 7:].count('1') % 2)
        else:
            depermuted_key += key[bit-1]
    return depermuted_key

def de_permute_c2(key):
    permutation = [ 5, 24,  7, 16,  6, 10, 20, 18,  0,
                   12,  3, 15, 23,  1,  9, 19,  2,  0,
                   14, 22, 11,  0, 13,  4,  0, 17, 21,
                    8, 47, 31, 27, 48, 35, 41,  0, 46,
                   28,  0, 39, 32, 25, 44,  0, 37, 34,
                   43, 29, 36, 38, 45, 33, 26, 42,  0,
                   30, 40]
    key = bin(key).replace('0b', '').zfill(48)
    depermuted_key = ''
    for bit in permutation:
        if bit == 0:
            depermuted_key += '0'
        else:
            depermuted_key += key[bit-1]
    depermuted_key = int(depermuted_key, 2)
    key_r = depermuted_key & 0xfffffff
    key_l = (depermuted_key >> 28) & 0xfffffff
    upper_r = (0x1 & key_r) << 27
    upper_l = (0x1 & key_l) << 27
    key_r = (key_r >> 1) & 0xfffffff
    key_l = (key_l >> 1) & 0xfffffff
    key_r = key_r | upper_r
    key_l = key_l | upper_l
    depermuted_key = (key_l << 28) | key_r
    return bin(depermuted_key).replace('0b','').zfill(56)

def get_bits(right, left, box):
    """input is string representation of 32 bit binary numbers. the bits are
    rearranged to correspond to the output of the S-box after its sent through
    the pbox. numbers are then converted to integers and returned"""
    if box == 1:
        r_bits = right[8] + right[16] + right[22] + right[30]
        r_bits = int(r_bits, 2)
        l_bits = left[8] + left[16] + left[22] + left[30]
        l_bits = int(l_bits, 2)
    elif box == 2:
        r_bits = right[12] + right[27] + right[1] + right[17]
        r_bits = int(r_bits, 2)
        l_bits = left[12] + left[27] + left[1] + left[17]
        l_bits = int(l_bits, 2)
    elif box == 3:
        r_bits = right[23] + right[15] + right[29] + right[5]
        r_bits = int(r_bits, 2)
        l_bits = left[23] + left[15] + left[29] + left[5]
        l_bits = int(l_bits, 2)
    elif box == 4:
        r_bits = right[25] + right[19] + right[9] + right[0]
        r_bits = int(r_bits, 2)
        l_bits = left[25] + left[19] + left[9] + left[0]
        l_bits = int(l_bits, 2)
    elif box == 5:
        r_bits = right[7] + right[13] + right[24] + right[2]
        r_bits = int(r_bits, 2)
        l_bits = left[7] + left[13] + left[24] + left[2]
        l_bits = int(l_bits, 2)
    elif box == 6:
        r_bits = right[3] + right[28] + right[10] + right[18]
        r_bits = int(r_bits, 2)
        l_bits = left[3] + left[28] + left[10] + left[18]
        l_bits = int(l_bits, 2)
    elif box == 7:
        r_bits = right[31] + right[11] + right[21] + right[6]
```

```
            r_bits = int(r_bits, 2)
            l_bits = left[31] + left[11] + left[21] + left[6]
            l_bits = int(l_bits, 2)
        else:
            r_bits = right[4] + right[26] + right[14] + right[20]
            r_bits = int(r_bits, 2)
            l_bits = left[4] + left[26] + left[14] + left[20]
            l_bits = int(l_bits, 2)
        return r_bits, l_bits

def correlate(hamming_distances, max_power, count):
    correlations = {}
    for key_guess in range(64):
        correlations[key_guess] = np.corrcoef(hamming_distances[key_guess], max_power)[0, 1]
    best_guess = max(correlations, key=correlations.get)
    #print best_guess
    if count == 104:
        print correlations[key_guess]
    return best_guess

break_encryption()
```

APPENDIX C
DES ENCRYPTION

```
def DES_enc(plain_txt, key_64):
    #plain_txt is a 64 bit input
    #key is 64 bits 8 bytes and each with a parrity bit
    #byte is defined to be 8 bits
    mix_txt = Initial_permutation(plain_txt)

    mask_32bit = 0xffffffff
    RE = mix_txt & mask_32bit
    LE = (mix_txt >> 32) & mask_32bit
    key_56 = Permutation_Choice1(key_64)

    for round in range(0, 16):
        key_48, key_56 = calc_r_key(key_56, round)
        LE, RE = DES_round(LE, RE, key_48)

    RE = RE << 32
    LE_RE = LE | RE
    enc_mess = Final_Permutation(LE_RE)
    return enc_mess

def DES_enc_56(plain_txt, key_56):
    #plain_txt is a 64 bit input
    #key is 56 bits
    #byte is defined to be 8 bits
    mix_txt = Initial_permutation(plain_txt)

    mask_32bit = 0xffffffff
    RE = mix_txt & mask_32bit
    LE = (mix_txt >> 32) & mask_32bit

    for round in range(0, 16):
        key_48, key_56 = calc_r_key(key_56, round)
        LE, RE = DES_round(LE, RE, key_48)

    RE = RE << 32
    LE_RE = LE | RE
    enc_mess = Final_Permutation(LE_RE)
    return enc_mess

def calc_r_key(key_56, r):
    shift_t = [1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1]
    mask_28bit = 0xfffffff
    key_r = key_56 & mask_28bit
    key_l = key_56 >> 28
    #circular shift key halves left based on table
    mask_h1 = 0x8000000 #r shift 27
    mask_h2 = 0xc000000 #r shift 26
    if shift_t[r] == 1:
        upper_r = (mask_h1 & key_r) >> 27
        upper_l = (mask_h1 & key_l) >> 27
        key_r = (key_r << 1) & mask_28bit
        key_l = (key_l << 1) & mask_28bit
        key_r = key_r | upper_r
        key_l = key_l | upper_l
    elif shift_t[r] == 2:
        upper_r = (mask_h2 & key_r) >> 26
        upper_l = (mask_h2 & key_l) >> 26
        key_r = (key_r << 2) & mask_28bit
        key_l = (key_l << 2) & mask_28bit
        key_r = key_r | upper_r
        key_l = key_l | upper_l
    else:
        print "Error!!!!"
    key_56 = (key_l << 28) | key_r
    r_key = Permutation_Choice2(key_56)
    return (r_key, key_56)

def DES_round(LE, RE, r_key):
    #expansion step returns a string
    RE_exp = expansion_permuatation(RE)

    #key mixing
    k_mix = RE_exp ^ r_key

    #sbox substitution
    RE_32 = Sbox_sub(k_mix)

    #permutation
    RE_32 = Pbox_sub(RE_32)

    RE_32 = RE_32 ^ LE
    LE_32 = RE
    return (LE_32, RE_32)

def expansion_permuatation(RE):
    bit_32 = 0x80000000
    bit_1 = 0x1
    RE = RE
    first = RE & bit_32
    last = RE & bit_1

    first = first >> 31
    last = last << 47

    k1 = mid_expansion(RE, 0)
    RE = RE >> 4
    k2 = mid_expansion(RE, 6)
    RE = RE >> 4
    k3 = mid_expansion(RE, 12)
    RE = RE >> 4
    k4 = mid_expansion(RE, 18)
    RE = RE >> 4
    k5 = mid_expansion(RE, 24)
    RE = RE >> 4
    k6 = mid_expansion(RE, 30)
    RE = RE >> 4
    k7 = mid_expansion(RE, 36)
    RE = RE >> 4
    k8 = RE << 42

    RE_exp = k1 | k2 | k3 | k4 | k5 | k6 | k7 | k8
    RE_exp = RE_exp << 1
    RE_exp = first | RE_exp | last
    return (RE_exp)

def mid_expansion(RE, shift):
    bit_5_mask = 0x1f
    bit_1_mask = 0x8
    temp = RE & bit_5_mask
    bit_6 = RE & bit_1_mask
    bit_6 = bit_6 << 2
    temp = temp | bit_6
    temp = temp << shift
    temp = temp
    return temp

def sbox(box_num, xored):
    if box_num == 1:
        return sbox1(xored)
    elif box_num == 2:
        return sbox2(xored)
    elif box_num == 3:
        return sbox3(xored)
    elif box_num == 4:
        return sbox4(xored)
    elif box_num == 5:
        return sbox5(xored)
    elif box_num == 6:
        return sbox6(xored)
    elif box_num == 7:
        return sbox7(xored)
    else:
        return sbox8(xored)

def sbox1(xored):
    """input is the relevant 6 bits of input message xored with the corresponding
    6 bits of key. returns 4 bit output of sbox1 table"""
    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]]
    row, column = S_access_location(xored)
    return S1[row][column] & 0xf

def sbox2(xored):
    """input is the relevant 6 bits of input message xored with the corresponding
    6 bits of key. returns 4 bit output of sbox2 table"""
    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]]
    row, column = S_access_location(xored)
    return S2[row][column] & 0xf

def sbox3(xored):
    """input is the relevant 6 bits of input message xored with the corresponding
    6 bits of key. returns 4 bit output of sbox3 table"""
    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]]
    row, column = S_access_location(xored)
    return S3[row][column] & 0xf

def sbox4(xored):
    """input is the relevant 6 bits of input message xored with the corresponding
    6 bits of key. returns 4 bit output of sbox4 table"""
    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]]
    row, column = S_access_location(xored)
    return S4[row][column] & 0xf

def sbox5(xored):
    """input is the relevant 6 bits of input message xored with the corresponding
    6 bits of key. returns 4 bit output of sbox5 table"""
    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]]
    row, column = S_access_location(xored)
    return S5[row][column] & 0xf

def sbox6(xored):
    """input is the relevant 6 bits of input message xored with the corresponding
    6 bits of key. returns 4 bit output of sbox6 table"""
    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]]
    row, column = S_access_location(xored)
    return S6[row][column] & 0xf

def sbox7(xored):
    """input is the relevant 6 bits of input message xored with the corresponding
    6 bits of key. returns 4 bit output of sbox7 table"""
    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]]
    row, column = S_access_location(xored)
    return S7[row][column] & 0xf

def sbox8(xored):
    """input is the relevant 6 bits of input message xored with the corresponding
    6 bits of key. returns 4 bit output of sbox8 table"""
    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]]
    row, column = S_access_location(xored)
    return S8[row][column] & 0xf

def Sbox_sub(RE_key):
    access_mask = 0x3f

    loc = RE_key & access_mask
    pos_1 = sbox8(loc)
    RE_key = RE_key >> 6

    loc = RE_key & access_mask
    pos_2 = sbox7(loc)
    pos_2 = pos_2 << 4
    RE_key = RE_key >> 6

    loc = RE_key & access_mask
    pos_3 = sbox6(loc)
    pos_3 = pos_3 << 8
    RE_key = RE_key >> 6

    loc = RE_key & access_mask
    pos_4 = sbox5(loc)
    pos_4 = pos_4 << 12
    RE_key = RE_key >> 6

    loc = RE_key & access_mask
    pos_5 = sbox4(loc)
    pos_5 = pos_5 << 16
    RE_key = RE_key >> 6

    loc = RE_key & access_mask
    pos_6 = sbox3(loc)
    pos_6 = pos_6 << 20
    RE_key = RE_key >> 6

    loc = RE_key & access_mask
    pos_7 = sbox2(loc)
    pos_7 = pos_7 << 24
    RE_key = RE_key >> 6

    loc = RE_key & access_mask
    pos_8 = sbox1(loc)
    pos_8 = pos_8 << 28

    R_sub = pos_1 | pos_2 | pos_3 | pos_4 | pos_5 | pos_6 | pos_7 | pos_8
    return R_sub

def S_access_location(access_loc):
    #high_order_bit = 0x80
    #access_loc = access_loc | high_order_bit
    c_mask = 0xf
    column = c_mask & (access_loc >> 1)
    r_mask1 = 0x20
    r_mask2 = 0x1
    row = (access_loc & r_mask1) >> 4
    row = row | (access_loc & r_mask2)
    return (row, column)

def Pbox_sub(RE):
    Pbox = [
        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]
    out = r_num_from_table(RE, Pbox, 32)
    return out

def Permutation_Choice1(k_64):
    PC1 = [
        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]
    out = r_num_from_table(k_64, PC1, 64)
    return out

def Permutation_Choice2(k_56):
    PC2 = [
        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]
    out = r_num_from_table(k_56, PC2, 56)
    return out

def Initial_permutation(data):
    """data is 64 bit intial message"""
    IP = [
        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]
    out = num_from_table(data, IP)
    return out

def Final_Permutation(data):
    FP = [
        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]
    out = num_from_table(data, FP)
    return out

def bit_select(num, bit_pos):
    if bit_pos > 0:
        bit_pos = bit_pos - 1
    mask = 2**bit_pos
    bit = int(num) & int(mask)
    bit = bit >> bit_pos
    return bit

def r_bit_select(num, bit_pos, num_len):
    num_len = num_len - 1
    bit_pos = bit_pos - 1
    pow = num_len - bit_pos
    mask = 2**pow
    bit = int(num) & int(mask)
    bit = bit >> pow
    return bit

def num_from_table(num, table):
    out = 0
    length = len(table)
    for x in range(0, length):
        bit = bit_select(num, table[x])
        bit = bit << x
        out = out | bit
    return out

def r_num_from_table(num, table, bits):
    out = 0
    length = len(table)
    for x in range(0, length):
        bit = r_bit_select(num, table[x], bits)
        out = out << 1
        out = out | bit
    return out

#Uncomment to test encryption
#k_64 = 0x3B3898371520F75E
#data = 0x44455321617A6277
#enc = DES_enc(data, k_64)
```