

Question 1

Finding ϵ

A good approximation for the entire SPN is $P_{16} \oplus U_{4,1} \oplus U_{4,9}$. This approximation uses 3 S-Boxes, S_{14} , S_{21} , and S_{31} . The following equations were used to approximate each S-Box.

$$\begin{array}{ll} S_{14} : X_4 = Y_1 & \text{bias: } -1/4 \\ S_{21} : X_4 = Y_1 & \text{bias: } -1/4 \\ S_{31} : X_1 = Y_1 \oplus Y_3 & \text{bias: } -1/4 \end{array}$$

These equations were obtained from the Linear Approximation Table for this S-Box (Table 1).

Table 1: Linear Approximation Table for the S-Box

Input Sum	Output Sum															
	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	+8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	+2	-2	0	+2	0	0	-2	-4	-2	-2	0	+2	0	-4	+2
2	0	+2	0	+2	-2	0	-2	0	-2	0	+2	-4	-4	-2	0	+2
3	0	0	+2	+2	0	+4	+2	-2	-2	-2	0	0	+2	-2	+4	0
4	0	+2	0	-2	0	+2	0	-2	+2	-4	+2	0	-2	0	-2	-4
5	0	+4	-2	-2	+2	+2	0	+4	-2	+2	0	0	0	0	+2	-2
6	0	0	+4	0	+2	+2	-2	+2	0	0	0	+4	-2	-2	-2	+2
7	0	-2	-2	0	+4	-2	+2	0	0	-2	-2	0	-4	-2	+2	0
8	0	+2	+2	0	0	-2	-2	0	+2	0	-4	-2	+2	-4	0	-2
9	0	0	0	+4	+2	+2	-2	+2	+2	-2	-2	-2	0	+4	0	0
a	0	+4	+2	+2	-2	-2	+4	0	0	0	-2	+2	-2	+2	0	0
b	0	-2	+4	-2	0	-2	0	+2	-4	-2	0	-2	0	+2	0	-2
c	0	0	+2	+2	+4	0	+2	-2	0	+4	+2	-2	0	0	-2	-2
d	0	-2	0	-2	-2	+4	+2	0	0	+2	-4	-2	-2	0	-2	0
e	0	-2	-2	+4	-2	0	0	+2	-2	0	0	+2	0	-2	-2	-4
f	0	0	0	0	0	0	+4	+4	+2	-2	+2	-2	+2	-2	-2	+2

Using these three approximations we get the following equations:

$$\begin{array}{l} U_{1,16} = V_{1,13} \\ U_{2,4} = V_{2,1} \\ U_{3,1} = V_{3,1} \oplus V_{3,3} \end{array}$$

Substituting based on the SPN we get:

$$\begin{aligned}P_{16} \oplus K_{1,16} &= V_{1,13} \\V_{1,13} \oplus K_{2,4} &= V_{2,1} \\V_{2,1} \oplus K_{3,1} &= V_{3,1} \oplus V_{3,3} \\V_{3,1} \oplus K_{4,1} &= U_{4,1} \\V_{3,3} \oplus K_{4,9} &= U_{4,9}\end{aligned}$$

Finally, collapsing all these equations down to one results in:

$$P_{16} \oplus K_{1,16} \oplus K_{2,4} \oplus K_{3,1} \oplus K_{4,1} \oplus U_{4,1} \oplus K_{4,9} \oplus U_{4,9} = 0$$

With the key bits removed, this results in $P_{16} \oplus U_{4,1} \oplus U_{4,9} = 0$.

Using the Piling-Up Lemma we can calculate the bias of this equation. It is:

$$\begin{aligned}2^{3-1} \epsilon_1 \epsilon_2 \epsilon_3 \\4 \times -1/4 \times -1/4 \times -1/4 \\-1/16\end{aligned}$$

Generating Random Plaintext/Ciphertext Pairs

Now that we know ϵ we know how many plaintext/ciphertext pairs we need to attack the cipher. Approximately $8\epsilon^{-2}$, or 2048 pairs need to be generated.

The first step to generating these pairs is to actually implement the cipher.

```
type Word4 = Word16
data SBox = SBox (Array Word4 Word4)

sbox :: [Word4] -> SBox
sbox = SBox . array (0,15) . zip [0..]

reverseSBox :: SBox -> SBox
reverseSBox (SBox a) = SBox $ array (0,15) $ zip (elems a) [0..]

splitBits :: Word16 -> [Word4]
splitBits x = map ((.&.0xf).(x`shiftR`)) [12,8,4,0]

concatBits :: [Word4] -> Word16
concatBits = foldr (.|.) 0 . flip (zipWith shiftL) [12,8,4,0]

sboxStage :: SBox -> Word16 -> Word16
sboxStage (SBox a) = concatBits . map (a!) . splitBits
```

The SBox stage splits the 16-bit input into 4 4-bit words, runs each through the sbox, then combines the 4 4-bit words back to one 16-bit word.

```
xorStage :: Word16 → Word16 → Word16
xorStage = xor
```

The xor stage simply xors the 16-bit input with 16-bits of the key.

```
permuteStage :: [Int] → Word16 → Word16
permuteStage mapping x = foldr (|.|) 0 $ map f $ zip [1..] $ mapping
  where f (s,d) | testBit x (16-s) = bit (16-d)
              | otherwise = 0
```

The permute stage moves the bits of the 16-bit input around based on a predefined mapping.

```
encrypt :: [Int] → SBox → [Word16] → Word16 → Word16
encrypt permuteMap sbox key x
  = xorStage finalKey
  $ foldl (\x' (k,p) → p permuteMap $ sboxStage sbox $ xorStage k x') x
  $ zip roundKeys ((take (rounds-1) $ repeat permuteStage) ++ [const id])
  where
    rounds = length key - 1
    (roundKeys,[finalKey]) = splitAt rounds key
```

The entire cipher consists of n “normal” rounds, a final round, and a final xoring. “Normal” rounds consist of an xor with the key, a sbox substitution, and a permutation. The last round doesn’t do the permutation.

```
stdEncrypt :: SBox → [Word16] → Word16 → Word16
stdEncrypt = encrypt
  (concatMap (\x → map (\y→y*4+x+1) [0,1,2,3]) [0,1,2,3])
```

stdEncrypt implements encryption with the permutations described in Heys’ paper.

```
hw61SBox :: SBox
hw61SBox = sbox $ map (fromIntegral.digitToInt) "8421C63DA5E7FB90"
```

hw61SBox is the S-Box for this homework.

```
key :: [Word16]
key = [0xdead,0xbeef,0xcafe,0xbabe,0xc0de]
```

key is the key to use for encryption.

```
numPairs :: Int
numPairs = 2048
```

numPairs is the number of know plaintext/ciphertext pairs to generate.

```

knownPairs :: [(Word16,Word16)]
knownPairs = map (\e → (e,stdEncrypt hw61SBox key e))
              $ take numPairs
              $ map fromIntegral
              $ (randoms $ mkStdGen 42 :: [Int])

```

knownPairs is a list of numPairs plaintext/ciphertext pairs.

The Attack

To implement the attack we need an equation to approximate the first 3 rounds and the xor stage of the fourth round. This equation is $P_{16} \oplus U_{4,1} \oplus U_{4,9} = 0$ (which was calculated above).

```

maybeCorrect :: Word16 → Word16 → Bool
maybeCorrect pw uw = xorList [p 16, u 1, u 9] == False
  where p i = testBit pw (16-i)
        u i = testBit uw (16-i)

```

maybeCorrect returns true if the given P and U pair agrees with the equation used to approximate the SPN.

With all the information above we can finally create a table containing the bias for every possible partial subkey that effects the output of S_{31} ($[K_{5,1} \dots K_{5,4}, K_{5,9} \dots K_{5,12}]$).

```

table :: [(Word4,Word4),Double]
table = map (\sk → (sk,runSubkey sk)) partialSubkeys
  where

```

table is a list of all possible partial subkeys and the bias for the approximation assuming that subkey is correct.

```

rsbox = reverseSBox hw61SBox
partialSubkeys = [(x,y) | x←[0..0xf],y←[0..0xf]]

```

rsbox is the inverse of the S-Box. partialSubkeys contains a list of all 256 possible values for $[K_{5,1} \dots K_{5,4}, K_{5,9} \dots K_{5,12}]$.

```

toBias n = abs
          $ fromIntegral (n-numPairs `div` 2) / fromIntegral numPairs
runSubkey (sk1,sk2) = toBias $ length $ filter id
                    $ map runPair knownPairs
  where
    runPair (pt,ct) = maybeCorrect pt
                    $ sboxStage rsbox
                    $ xorStage key ct
    key = concatBits [sk1,0,sk2,0]

```

runSubKey finds the bias for the approximation assuming that subkey is correct. To do this it reverses each ciphertext up to the U_4 step by xoring with the subkey and reversing the sbox. It

Table 2: Experimental Results for Linear Attack

Partial Subkey/ bias															
00	.0029	20	.0054	40	.0146	60	.0010	80	.0146	a0	.0024	c0	.0107	e0	.0117
01	.0029	21	.0015	41	.0205	61	.0049	81	.0029	a1	.0034	c1	.0117	e1	.0010
02	.0078	22	.0054	42	.0127	62	.0107	82	.0068	a2	.0103	c2	.0049	e2	.0010
03	.0127	23	.0112	43	.0166	63	.0088	83	.0039	a3	.0229	c3	.0049	e3	.0078
04	.0146	24	.0259	44	.0127	64	.0156	84	.0029	a4	.0034	c4	.0039	e4	.0068
05	.0063	25	.0039	45	.0122	65	.0015	85	.0024	a5	.0088	c5	.0161	e5	.0093
06	.0083	26	.0254	46	.0044	66	.0122	86	.0073	a6	.0283	c6	.0210	e6	.0024
07	.0068	27	.0161	47	.0273	67	.0000	87	.0127	a7	.0054	c7	.0098	e7	.0068
08	.0068	28	.0083	48	.0127	68	.0195	88	.0029	a8	.0015	c8	.0186	e8	.0068
09	.0010	29	.0054	49	.0020	69	.0029	89	.0010	a9	.0220	c9	.0059	e9	.0107
0a	.0059	2a	.0093	4a	.0098	6a	.0127	8a	.0049	aa	.0083	ca	.0010	ea	.0127
0b	.0088	2b	.0024	4b	.0146	6b	.0098	8b	.0078	ab	.0190	cb	.0127	eb	.0127
0c	.0020	2c	.0054	4c	.0166	6c	.0078	8c	.0049	ac	.0171	cc	.0205	ec	.0049
0d	.0024	2d	.0225	4d	.0249	6d	.0015	8d	.0005	ad	.0068	cd	.0444	ed	.0015
0e	.0005	2e	.0010	4e	.0171	6e	.0093	8e	.0093	ae	.0127	ce	.0073	ee	.0083
0f	.0059	2f	.0151	4f	.0020	6f	.0078	8f	.0049	af	.0083	cf	.0068	ef	.0049
10	.0190	30	.0059	50	.0029	70	.0059	90	.0220	b0	.0059	d0	.0098	f0	.0020
11	.0132	31	.0010	51	.0029	71	.0215	91	.0083	b1	.0068	d1	.0010	f1	.0107
12	.0122	32	.0098	52	.0020	72	.0273	92	.0171	b2	.0049	d2	.0078	f2	.0098
13	.0132	33	.0000	53	.0117	73	.0146	93	.0015	b3	.0166	d3	.0049	f3	.0264
14	.0239	34	.0195	54	.0098	74	.0244	94	.0054	b4	.0020	d4	.0010	f4	.0078
15	.0186	35	.0034	55	.0103	75	.0063	95	.0137	b5	.0073	d5	.0024	f5	.0024
16	.0010	36	.0063	56	.0034	76	.0063	96	.0020	b6	.0093	d6	.0132	f6	.0317
17	.0288	37	.0176	57	.0020	77	.0244	97	.0181	b7	.0020	d7	.0049	f7	.0068
18	.0171	38	.0059	58	.0020	78	.0000	98	.0103	b8	.0020	d8	.0283	f8	.0059
19	.0142	39	.0010	59	.0107	79	.0020	99	.0132	b9	.0010	d9	.0029	f9	.0059
1a	.0132	3a	.0117	5a	.0059	7a	.0039	9a	.0044	ba	.0010	da	.0059	fa	.0146
1b	.0112	3b	.0117	5b	.0166	7b	.0205	9b	.0103	bb	.0127	db	.0137	fb	.0225
1c	.0024	3c	.0068	5c	.0068	7c	.0010	9c	.0142	bc	.0039	dc	.0098	fc	.0029
1d	.0293	3d	.0122	5d	.0103	7d	.0083	9d	.0137	bd	.0093	dd	.0103	fd	.0112
1e	.0117	3e	.0024	5e	.0034	7e	.0083	9e	.0020	be	.0112	de	.0210	fe	.0181
1f	.0024	3f	.0088	5f	.0186	7f	.0010	9f	.0093	bf	.0078	df	.0059	ff	.0039

then runs each P and U_4 pair through the approximation equation and counts the number of times the approximation equation hold true.

The results of this are shown in Table 2. The partial subkey with the highest bias was “C D” which corresponds to bits 1-4 and 9-12 of “0xc0de”. From this we can conclude that there is a very good chance that “C” and “D” are the partial subkey (and because we know the key in this case we can verify that this is indeed the case).

Question 2

```
hw62SBox :: SBox
hw62SBox = sbox $ map (fromIntegral.digitToInt) "E213D906F45A8C7B"
```

hw62SBox is the S-Box for this question.

Difference Distribution Table

```

differenceDistTable :: SBox -> [[Int]]
differenceDistTable (SBox sbox) = map f [0..0xf]
  where
    f i = elems $ accumArray (+) 0 (0,0xf)
          $ map (\x -> ((sbox!x) `xor` (sbox!(x`xor`i)),1)) [0..0xf]

```

differenceDistTable calculates the difference distribution table for an S-Box. This is calculated by iterating through all possible ΔX s, and all possible X' s, calculating X'' for each X' , running both through the S-Box, calculating ΔY , and recording how many times each ΔY comes up.

Table 3: Difference Distribution Table for the S-Box

Input Difference	Output Difference															
	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	2	0	4	0	2	0	0	0	0	2	4	0	0	2
2	0	2	0	0	0	0	0	2	0	0	2	0	0	2	2	6
3	0	2	0	4	0	2	0	0	0	2	0	4	0	2	0	0
4	0	4	2	2	0	2	0	2	2	0	0	2	0	0	0	0
5	0	0	0	4	0	0	0	4	0	0	0	0	2	2	2	2
6	0	0	0	0	2	0	2	0	2	0	2	0	2	2	2	2
7	0	0	4	2	2	0	0	0	4	2	0	0	0	0	2	0
8	0	2	0	0	2	4	2	2	0	2	0	0	0	2	0	0
9	0	6	0	0	0	0	2	0	0	0	2	4	0	2	0	0
a	0	0	2	0	0	0	0	2	4	0	4	2	0	0	2	0
b	0	0	0	0	2	2	2	2	0	0	0	0	4	0	4	0
c	0	0	2	0	0	2	4	0	2	0	0	0	2	2	2	0
d	0	0	2	2	2	0	2	0	0	2	6	0	0	0	0	0
e	0	0	2	2	0	0	0	0	2	6	0	0	0	0	0	4
f	0	0	0	0	2	4	0	2	0	2	0	2	2	2	0	0

Differential Characteristic for the Whole Cipher

Using Table 3 we can calculate the differential characteristic for the whole cipher. To calculate this we need to find a path between the 4 S-Boxes given.

If we start with 1001 as the difference to S_{11} and S_{14} there is a high probability that the output difference will be 0001 on each S-Box. After the permutation each of these bits end up in S_{24} . The difference in the input the S_{24} will be 1001. Just like the S_1 S-Boxes, there is a high probability the output difference here will be 0001. After the permutation this bit end up in S_{34} . The input to S_{34} will be 0001. There is a high probability that the output difference will be 0100 here. This creates a path from P to U_4 .

Looking at each S-Box individually we find that:

$$S_{11} : R_p(1001, 0001) = 6/16$$

$$S_{14} : R_p(1001, 0001) = 6/16$$

$$S_{24} : R_p(1001, 0001) = 6/16$$

$$S_{34} : R_p(0001, 0100) = 4/16$$

If we assume each propagation ratio is independent then we can combine them all together by multiplying. The propagation ration from P to U_4 is therefore $(6/6)^3 4/16$, or $27/2048$.