6.857 Homework	**Problem Set 3**	# 3-3 - Kalns
Anonymous		April 15, 2014

a) Our suspicion is well founded, which we can see from a very simple analysis. Consider an original key combination (A, b, S), where we call $Ax + b = v$. Let us then consider a possible key combination (n*A, n*b, S'), where n is some number in GF16. The output of this new key pair will be $S'(nAx + nb) = S'(nv)$. Thanks to the properties of GF16, we know that $n * m$, where n is a GF16 value held constant and m is a GF16 value varied from 0 to 16, return all different values ranging from 0 to 16. (Of course, the case where m = 0 maps to 0, but the other cases map more interestingly.) Thus, multiplying v by n can be considered as just an extra permutation, P(v). (This is true if n is nonzero, but we are simply choosing this to be the case.) This means we can just adjust S' to look like S after canceling out this original mapping, i.e $S'(x) = S(P^{-1}(x))$, and we have an equivalent key pair. Thus we have generated several equivalent key pairs. It turns out that, since we have such freedom with S, we can actually generate even more equivalent key pairs by basically generating new v's and adjusting our S accordingly, as shown above. Thus, we know that there will be many equivalent key pairs.

b) Let's consider a series of possible inputs and outputs that we could try encrypting/decrypting, and consider the information gleaned from each.

If we encrypt the vector $x_0 = [0, 0, 0, 0, ..., 0]$, we will get $y_0 = S(b)$. Additionally, if we encrypt the vectors x_u^i, which are the 16 unit vectors (i.e. the vectors with all 0's except a 1 at position i), we will get back $y_u^i = S(A^i + b)$, where A^i is the i'th column of A.

We now have information about A and b, but they are embedded in S, the permutation. In order to get information about this, we can decrypt the values $y_v^i = [i, 0, 0, 0, ...0]$. Reversing our algorithm, we can see that $x = A^{-1}(S^{-1}(y) - b)$, which means we will get back $x_v^i = (A^{-1})^0(S^{-1}(y_v^i) - b)$. If we take the first element of these x_v^i, we will get $(A^{-1})^{0,0}(S^{-1}(i) - b^0)$, and if we take the differences $d^i = x_v^0 - x_v^i$, we will get $(A^{-1})^{0,0}(S^{-1}(0) - S^{-1}(i))$. Here, we take advantage of the properties of GF16 to define a new, valid permutation, S' such that $S'^{-1} = (A^{-1})^{0,0}S^{-1}$, such that the values d^i are actually differences between the inverse permutation values for 0 and i. Defining S'(0) = 0, we thus have a valid permutation that we have fully defined, and which we can construct an equivalent key pair using. (Recall from part a that we can do this because our permutation is correct up to a multiplicative constant!).

Using this new S', we can decrypt S(b) such that $S'^{-1}(S(b)) = b'$, and similarly extract each column of A as a column of a new matrix A'. This new key combination (A', b', S') is equivalent to the original combination (A, b, S). Since we now have an equivalent key set to the original combination, we can freely encrypt and decrypt data as we please, completely compromising the Kalns encryption scheme.

c) We have successfully implemented this algorithm. See MCRBFinalFast on `http://6857.scripts.mit.edu/kalns/`. Our code follows the exact process detailed in part b. The MITx submission site only allows one uploaded file, so we did not submit our actual code file, but the code we used is reproduced below using the verbatim tag.

```
from kalns import *
```

```
tokenString = remote_query('keygen?team=MCRBFinalFast')

theToken = tokenString[80:112]

rk = RemoteKalns(theToken)

b = int64_to_GF16_vec(rk.enc(0))

r_unit = []
for i in range(16):
    r_unit.append(int64_to_GF16_vec(rk.enc(2**(4*(15-i)))))

A = []
for row in range(16):
    A.append([])
    for col in range(16):
        A[row].append(r_unit[col][row])

iTimesAInv = []
for i in range(16):
    #print int64_to_GF16_vec(i*(2**(4)))
    iTimesAInv.append(int64_to_GF16_vec(rk.dec(i*(2**(4)))))

topVals = []
for i in range(16):
    topVals.append(iTimesAInv[i][0])

#These for loops are actually irrelevant, they were
#originally to ensure that we didn't need to try cyclic permutations of
#the S that we derived. It turns out we don't, so we just set the ranges
#to 1.
for i in range(1):
    for ii in range(1):
        AA = []
        BB = []
        s = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
        sinv = []
        for iii in range(16):
            x = topVals[iii]
            x0 = topVals[0]
            s[int((GF16(i) - ((x0-x)/GF16(ii+1))).val)] = int(iii)

        for ij in range(16):
            sinv.append(s.index(ij))
        for k in range(16):
            AA.append([])
            for l in range(16):
                AA[k].append(GF16(sinv[A[k][l].val]))
            BB.append(GF16(sinv[b[k].val]))
```

```
for kk in range(16):
    for ll in range(16):
        newVar = AA[kk][ll] - BB[kk]
        AA[kk][ll] = newVar

print "s = " + str(i) + ", Ainv[0][0] = " + str(ii)
print rk.answer(AA, BB, s)
```