Kevin Hulin (kjh061000@utdallas.edu) and Patrick Mealey (plm051000@utdallas.edu)
Dr. Sha
Information Security
10/15/2010

Programming Assignment 2

1. Compiling and Running
   a. Our project can be compiled by running make int the project directory. This will result in three files being compiled: transform, enc, and dec.
   b. transform is the forging program for part 1. It can be run in the following manner: ./ transform <MaxValue> <DesiredResult> <ForgedFile>
   c. enc is the encryption program for part 2. It can be run in the following manner: ./enc <Key> <Plaintext file>
   d. dec is the decryption program for part 2. It can be run in the following manner: ./dec <Key> <Plaintext file>
2. Forging Message Digest
   For this problem, we took a simple approach to forging contracts. By iteratively applying a padding of <space><backspace>, we were able to change the file without changing its printed appearance. Below are our results when run on the given signed files. All forgeries were created in less than 1 second run time.

	File size	MaxValue	Signature
Original forged	802	N/A	N/A
Forged1	824	15	4
Forged2	824	255	124
Forged3	1484	1023	289
Forged4	4964	2047	1336

   As apparent in the above table, as the signature parameter increases, the size of the forged file increases dramatically. While this difference will not be visible in the printed files, it is not desirable. A possible solution would be to perform the algorithm, varying the padding to be <char><backspace> where <char> could be any character. This would likely find a more optimal padding choice and yield shorter forged files.

3. Design your own encryption and decryption algorithms
   In designing our encryption algorithm, our goal was to ensure maximum difference between the encrypted files given in MSG1, MSG2, and MSG3. This proved to be a challenge, though in our final implementation, we believe the resulting encrypted files varied quite significantly.
   Our initial idea was to design a multi pass block cipher in which we arranged the plain text into a single column of width keylen and XOR'd each row with the previous row (using the key for the first row). This was much like Vigenere's Autokey cipher except that instead of shifting, we would be applying XOR. Furthermore, instead of aligning each row with the previous row, we constructed a function of two characters in the previous row so that it would be scrambled. By not applying the same key in each instance, we significantly reduce susceptibility to attacks by statistical analysis. However, this alone does not significantly change the file, so in a second pass, this time horizontally, we apply a similar method. In our horizontal pass, we apply a shift of the previous column's sum to each character, again using the original key for the first column. This horizontal pass is performed keylen rows at a time and repeated until the end of the file is reached. In order to ensure our algorithms do not fail, we must also pad the end of our file. Padding is done by appending X null bytes until the last row is filled. Finally, we append an additional row which contains the number of padding bytes and newline fills so that after we have decrypted it, we can remove this padding.
   Finally, because we chose to designate two bytes in the file to store the number of padding bytes,

the key length must be between 2 and 2^16 bytes long. The machine used for encrypting or decrypting data must have 2*keylen^2 bytes of memory available. Also, the file length should not be 0.

Algorithm pseudocode:

```
Encrypt(P,k): //P = plaintext; k = key
    P2 = Pad(P,keylen); // Append padding as described above
    C1 = VerticalPass(P,k);
    C2 = HorizontalPass(C1,k);
    return C2
VerticalPass(P,k): //P = plaintext; k = key
    C1 is a characterArray
    for i = 0 to numRows -1:
        for j = 0 to keylen - 1:
            int f = F(i,j,P[i-1])
            int g = G(i,j,P[i-1])
            int x1 = (j-f) mod keylen
            int x2 = (j-g) mod keylen
            x1 = (x1 < 0)?-x1:x1
            x2 = (x2 < 0)?-x2:x2
            int x = x1 XOR x2 XOR P[i+j*keylen]
            C1[i+j*keylen]=x mod 256
    return C1
HorizaontalPass(P,k): //P = plaintext; k = key
    C2 is a character Array
    for i = 0 to keylen -1:
        int v = sum(P[i+k*j],k=0 to keylen)
         for j = 0 to numRows - 1:
                C2[i + j*keylen] = P[i+j*keylen]+v mod 256
    return C2


F(k,i,j): //k = keylen string, i,j are indexes
    return k[i] XOR k[j]
G(k,i,j): //k = keylen string, i,j are indexes
    return k[i] * k[j]
```

Analysis:

Our algorithm is far from perfect. It's time complexity is O(n) because it requires a constant number (two) of passes through the text. In a known plain text attack, it is very easy to determine the key length simply by comparing the plaintext and cipher text file sizes. Furthermore, given an input that is N*keylen bytes long, an attacker will know that the last keylen bytes of the file are {0x00, 0x00, 0x0A, …….}. A chosen plaintext attack would likely be able to discover the key, however, despite our efforts, we were not able to break it within a reasonable amount of time.

Encrypting the provided messages yielded the following results:

Using key: AmzH81aL

	Diff MSG2	Diff MSG3
MSG1	43	17