Solutions to Homework 1

Problem 1

Perfect secrecy is the notion that observing the ciphertext should have no effect on the adversary's knowledge regarding the actual message that was sent. This means that the ciphertext reveals nothing about the underlying plaintext.

One-time pad satisfies perfect secrecy because it requires that we generate a random k (n-bits) to xor with our message every time. This means that the same message wouldn't have the same ciphertext because each has a different k. To prove more formally that one time pad satisfies perfect secrecy we have for every plaintext message m:

$$Pr[Enc_k(m) = c] = Pr[k \oplus m = c] = Pr[k = m \oplus c] = (\frac{1}{2})^n$$

Therefore if we pick message m_1, m_2 and a ciphertext c, we have:

$$Pr[Enc_k(m_1) = c] = Pr[Enc_k(m_2) = c] = (\frac{1}{2})^n$$

, which indicates that one time pad is perfectly secret.

The given one-time pad does not satisfy perfect secrecy, because if we choose for example Let $m_1 = 01, m_2 = 11$ and $c = 11$, we have $Pr[Enc_k(m_1) = c] = Pr[Enc_k(01) = 11] = 0$and $Pr[Enc_k(m_2) = c] = Pr[Enc_k(11) = 11] = 1/3$.We see that the two probabilities are not equal and hence this one time pad does not satisfy perfect secrecy.

Problem 2

Replication: We have n blocks $b_1, \ldots, b_n$. If we use simple replication which is storing replicas of the blocks. It is enough to delete all copies of a certain block to lose it permanently.

Error Correcting Codes: In this method $n + 1$ files need to be deleted to guarantee that some block b_i is irretrievable. This is because polynomial interpellation allows us to recover the initial data even if we more n or more blocks intact.

Problem 3

1. Find example online
2. Suppose we have a key k, and we use it to encrypt two messages m_1 and m_2 via the one-time pad technique. We thus get two ciphertexts $c_1 = m_1 \oplus k$ and $c_2 = m_2 \oplus k$. If we know c_1 and c_2, we can easily calculate $c_1 \oplus c_2 = m_1 \oplus k \oplus m_2 \oplus k = m_1 \oplus m_2$. At this point, if we know certain words in m_1 for example, we can figure out parts of m_2 as well. We don't even need to know that a word appears specifically in m_1, actually. Essentially a demonstration of the idea is: We can XOR the ciphertexts of the two expressions, then XOR the result with the words "SOLDIER" and "WITHDRAW" at various positions to recover more of the original plaintext. This is called a Crib-Dragging attack.
3. The following code is an implementation of the Crib-Dragging attack. Knowing that the words CRYPTOGRAPHIC, PASSWORDS, DECIPHER, and MATHEMATICS must appear somewhere in the three ciphertexts, we can take the steps outlined as follows:
   (a) Pairwise XOR the three ciphertext messages.
   (b) XOR one of the key words with the results from (a) at each position.
   (c) Repeat (b) for all key words.

(d) Not included in the code: once you uncover more words from the plaintext, you can expand your key word search.

The original plaintext is:
SUBSTITUTION CIPHERS ARE NOT REALLY HARD TO DECIPHER
NEVER SEND PASSWORDS OR CRYPTOGRAPHIC KEYS IN EMAILS
CRYPTOGRAPHIC TECHNIQUES RELY ON MATHEMATICS MOSTLY.

Code:

```
#include <stdio.h>
#include <string.h>

void runMatch(int*, char*);

int main() {
    // The basic concept for this exercise is figuring out that XORing
    // the messages will leak information. Visual analysis of key words
    // matched with XORed pairs of ciphertext messages should reveal
    // meaningful English text buried in the middle of nonsense.

    int cipher1[52] = {0x73, 0x7f, 0x68, 0x79, 0x7e, 0x69, 0x1a, 0x10,
                       0x2, 0xc, 0x1d, 0x6e, 0x75, 0x10, 0xc, 0x70, 0x9,
                       0x65, 0x1d, 0x1d, 0x65, 0x61, 0x6, 0xc, 0x6d, 0xb,
                       0x6f, 0x4, 0x61, 0x16, 0x65, 0xc, 0x3, 0x1e, 0x1c,
                       0x0, 0x1c, 0x9, 0x13, 0xa, 0x0, 0x1b, 0x1, 0x63,
                       0x1, 0x65, 0x69, 0x63, 0x7a, 0x62, 0x6f, 0x72};
    int cipher2[52] = {0x6e, 0x6f, 0x7c, 0x6f, 0x78, 0x0, 0x1d, 0x0, 0x18,
                       0x1, 0x72, 0x70, 0x14, 0x0, 0x16, 0x77, 0xe, 0x72,
                       0xb, 0x1d, 0x65, 0x6f, 0x6, 0x69, 0xe, 0x17, 0x79,
                       0x0, 0x15, 0xb, 0x67, 0x1f, 0xe, 0x2, 0xd, 0x69,
                       0x17, 0x68, 0xa, 0xb, 0x79, 0x1c, 0x6e, 0xa, 0xb,
                       0x0, 0x6f, 0x67, 0x6b, 0x63, 0x66, 0x73};
    int cipher3[52] = {0x63, 0x78, 0x73, 0x7a, 0x7e, 0x6f, 0x9, 0x17, 0x17,
                       0x15, 0x1a, 0x69, 0x16, 0x73, 0x11, 0x65, 0x2, 0x68,
                       0x1, 0x7, 0x14, 0x75, 0x11, 0x1a, 0x6d, 0x17, 0x65,
                       0x1c, 0x18, 0x64, 0x6f, 0x3, 0x6f, 0x1f, 0x4, 0x74,
                       0x1c, 0xd, 0xc, 0xf, 0x74, 0x6, 0xd, 0x10, 0x65,
                       0x6d,0x65, 0x79, 0x7e, 0x66, 0x73, 0xe};

    char word1[] = "CRYPTOGRAPHIC";
    char word2[] = "PASSWORDS";
    char word3[] = "DECIPHER";
    char word4[] = "MATHEMATICS";

    int xor_12[52];
    int xor_23[52];
    int xor_13[52];

    int i;

    // Pairwise XOR the ciphertexts
    for (i = 0; i < 52; i++) {
        xor_12[i] = cipher1[i] ^ cipher2[i];
        xor_23[i] = cipher2[i] ^ cipher3[i];
        xor_13[i] = cipher1[i] ^ cipher3[i];
```

```
47      }
48
49      printf("\n\nExamining Word 1 versus the XORed messages\n");
50      printf("* CIPHERS 1 and 2 \n");
51      runMatch(xor_12, word1);
52      printf("* CIPHERS 1 and 3 \n");
53      runMatch(xor_13, word1);
54      printf("* CIPHERS  2 and 3 \n");
55      runMatch(xor_23, word1);
56
57      printf("\n\nExamining Word 2 versus the XORed messages\n");
58      printf("* CIPHERS 1 and 2 \n");
59      runMatch(xor_12, word2);
60      printf("* CIPHERS  1 and 3 \n");
61      runMatch(xor_13, word2);
62      printf("* CIPHERS 2 and 3 \n");
63      runMatch(xor_23, word2);
64
65      printf("\n\nExamining Word 3 versus the XORed messages\n");
66      printf("* CIPHERS  1 and 2 \n");
67      runMatch(xor_12, word3);
68      printf("* CIPHERS  1 and 3 \n");
69      runMatch(xor_13, word3);
70      printf("* CIPHERS  2 and 3 \n");
71      runMatch(xor_23, word3);
72
73      printf("\n\nExamining Word 4 versus the XORed messages\n");
74      printf("* CIPHERS 1 and 2 \n");
75      runMatch(xor_12, word4);
76      printf("* CIPHERS 1 and 3 \n");
77      runMatch(xor_13, word4);
78      printf("* CIPHERS 2 and 3 \n");
79      runMatch(xor_23, word4);
80
81      return 0;
82  }
83
84  void runMatch(int* xoredMessage, char* word) {
85      int i, j;
86      for (i = 0; i <= 52 - strlen(word); i++) {
87          printf("\tAt i = %d \t", i);
88          for (j = 0; j < strlen(word); j++) {
89              printf("%c", xoredMessage[i+j] ^ word[j]);
90          }
91          printf("\n");
92      }
93  }
```

Output: the output of this program is very long, thus we just show portions of it

```
Examining Word 1 versus the XORed messages
* CIPHERS 1 and 2
  ...
  At i = 23    &1EFP;ZPR]TX*
  At i = 24     NOT REALLY H
  At i = 25    _D]$IMT_]A!B"
```

```
   ...
* CIPHERS 1 and 3
   At i = 0   SUBSTITUTION
   At i = 1   DIZPR\@GXWO*
   At i = 2   XQYVGHRKFW+*^
   ...
* CIPHERS  2 and 3
   At i = 0   NEVER SEND PA
   At i = 1   T]LV;[P]U8QK0
   At i = 2   LG_?@XHF)IJ:D
   ...
Examining Word 2 versus the XORed messages
* CIPHERS 1 and 2
   ...
   At i = 10    ?_2CMHUSE
   At i = 11    N CIPHERS
   At i = 12    1QITPXDDS
   ...
* CIPHERS  1 and 3
   ...
   At i = 39    U5N_$+ZHI
   At i = 40    $\_ 3G^^W
   At i = 41    MM 7_CH@W
   At i = 42    \27[[UV@O
   At i = 43    #%[_MKVX/
* CIPHERS 2 and 3
   ...
   At i = 10    8XQ P]^^Y
   At i = 11    IC TECHNI
   At i = 12    R2TA[UX^"
   ...
```

Problem 4

$y = k_b \oplus x$, so $y \oplus x = k_b \oplus x \oplus x = k_b$, so Eve can compute $y \oplus x$ to find the key.

Since there is no message authentication between Alice and Bob's exchanges, Eve can easily act as a woman-in-the-middle attacker and change the content of Bob's response to Alice. Alice will thus receive a y' that looks like it comes from Bob, but $y' \neq r$