Part (a)

To decode the ciphertexts, we XORed them together, which gives $C_1 \oplus C_2 = (M_1 \oplus P) \oplus (M_2 \oplus P) = M_1 \oplus M_2$. Thus the pad $P$ is irrelevant. Then we found a set of 8-character words from an English dictionary. For each $M_1$ in this set, we constructed $M_2$ by XORing $M_1$ with $C_1 \oplus C_2$, since $M_1 \oplus (C_1 \oplus C_2) = M_1 \oplus (M_1 \oplus M_2) = M_2$. Lastly, we checked if $M_2$ was also in the set. If both $M_1$ and $M_2$ were in the set (as valid English words), then we outputted them.

```python
C1 = [0xe9, 0x3a, 0xe9, 0xc5, 0xfc, 0x73, 0x55, 0xd5]
C2 = [0xf4, 0x3a, 0xfe, 0xc7, 0xe1, 0x68, 0x4a, 0xdf]
M1_XOR_M2 = [c ^ d for c, d in zip(C1, C2)]
with open('/usr/share/dict/words', 'r') as words_file:
    words = words_file.read().split()
    words = set([word for word in words if len(word) == len(M1_XOR_M2)])
for word1 in words:
    M1 = [ord(c) for c in word1]
    M2 = [c ^ d for c, d in zip(M1, M1_XOR_M2)]
    word2 = ''.join([chr(c) for c in M2])
    if word2 in words:
        pad = [c ^ d for c, d in zip(M1, C1)]
        print 'word1 = %s, word2 = %s, pad = %s' % (word1, word2, pad)
```

Output of running the code:

```plaintext
word1 = networks, word2 = security, pad = [135, 95, 157, 178, 147, 1, 62, 166]
word1 = security, word2 = networks, pad = [154, 95, 138, 176, 142, 26, 33, 172]
```

Though their ordering is indiscernible, the two words are security and networks.

Part (b)

Messages and Pad

We stand today on the brink of a revolution in cryptography. Probabilistic encryption is the use of randomness in an encryption protocol. Secure Sockets Layer (SSL), are cryptographic protocols that MIT developed Kerberos to protect network services provided by NIST. Kerberos. Crypto. MIT developed Kerberos to protect network services provided by NIST. Announced a competition to develop a new cryptographic protocol. Diffie-Hellman establishes a shared secret that can be used for public-key cryptography. The keys used to sign the certificates had been stolen from MIT. We hope this inspires others to work in this fascinating field.
\[
\]

**Process**

This part’s code was more interactive because, due to Ben’s addition of feedback, we couldn’t simply XOR the 10 ciphertexts together and look up possible messages in the dictionary. Our plan of attack was to first calculate all possible pad bytes \(p_i\)’s that would result in valid and likely English characters (letters and common punctuation) from all 10 ciphertexts.

\[
\text{valid_chars} = \text{set}((\text{range}(65, 65 + 26) + \text{range}(97, 97 + 26) + \text{range}(32, 44, 46, 63, 33, 45, 40, 41)) \# A-Z, a-z \\
\text{space, .,?!-})
\]

Let’s consider a particular index \(i\) in the 60-character messages \((0 \leq i < 60)\). For our purposes, \(p_i\) is independent from the pad bytes surrounding it, because it only depends on \(m_i, c_i, \text{and } c_i - 1\) (as the \text{calculate_pad} function below shows). We could have tried all \(2^6\) possible \(p_i\)’s, but \(|\text{valid_chars}|\) is only 60. Therefore, we took each valid character, calculated which \(p_i\) would result in that character in the first ciphertext, and checked if it resulted in valid characters for the other 9 ciphertexts.

```python
def calculate_pad(ctext, msg, prev_c=0):
    assert len(ctext) == len(msg)
    pad = []
    for i in xrange(len(ctext)):
        p = (ctext[i] - msg[i] - prev_c) % 256
        pad.append(p)
        prev_c = ctext[i]
    return pad

def prev_c_at(ciph, index):
    return 0 if index == 0 else ciph[index - 1]
def ctext_at(ciph, index):
    return ciph[index:index + 1]

msglen = 60
possible_pad_bytes = [0] for _ in range(msglen)
for index in range(msglen):
    for c in valid_chars:
        possible_pad_byte = calculate_pad(ctext_at(tenciphs[0], index), [c], prev_c=prev_c_at(tenciphs[0], index))
        is_valid = True
        for ciph in tenciphs:
            msg = ben_decrypt(ctext_at(ciph, index), possible_pad_byte, prev_c=prev_c_at(ciph, index))
            if not set(msg).issubset(valid_chars):
                is_valid = False
                break
        if is_valid:
            possible_pad_bytes[index].append(possible_pad_byte)
```

This calculated all possible \(p_i\)’s at each \(i\), which is a good start since some choice of \(P = p_1, \ldots, p_{60}\) within these \(p_i\)’s would result in Ben’s messages. However, here’s the issue:

```python
>>> [len(p) for p in possible_pad_bytes]
[20, 6, 2, 1, 2, 1, 1, 1, 1, 5, 5, 4, 1, 1, 1, 8, 3, 2, 2, 2, 1, 1, 1, 3, 3, 1, 1, 1, 1, 5, 4, 1, 1, 2, 1, 2, 2, 9, 1, 4, 2, 1, 1, 2, 1, 1, 1, 1, 1, 2, 2, 1, 2, 1, 1, 1, 3, 2, 1]
```
There is a combinatorial explosion of $20 \times 6 \times \cdots \times 1$, over $2^{47}$, choices for $P$. However, there are a tractable 480 choices for the first 9 bytes. We figured that we could restrict the problem to first choosing those: listing all 480 $p_1, \ldots, p_{9}$ pads, decrypting the first 9 bytes of the 10 ciphertexts with each pad, and checking which pads gave intelligible English plaintext.

While we could scan the plaintext manually, we preferred to have the computer do it and score its intelligibility. So, we loaded an English dictionary (the Ubuntu dictionary is excellent; it even contains Hellman). For each set of 10 plaintext messages, we checked how many valid English words from the dictionary appeared in it and gave it $|word|^2$ points for each word that did. This scoring function strongly favors longer words like cryptography. Lastly, we outputted the best-scoring messages and pad.

```python
def recursively_expand_pad(cur_pad, cur_index, words):
    if cur_index == msglen:
        # To start, msglen is 9 instead of 60.
        # Reached the leaves of our search, so decrypt the 10 ciphertexts and score the
        # resulting text.
        texts = [bytes_to_text(ben_decrypt(ciph[:msglen], cur_pad))
                 for ciph in tenciphs]
        text_to_score = '\n'.join(texts).lower()
        score = sum(len(word)**2 for word in words if word in text_to_score)
        return (score, texts, cur_pad)
    else:
        best_score = 0; best_texts = None; best_pad = None
        for p in possible_pad_bytes[cur_index]:
            score, texts, pad = recursively_expand_pad(cur_index + 1, cur_pad + [p], words)
            if best_score < score:
                best_score = score; best_texts = texts; best_pad = pad
        return (best_score, best_texts, best_pad)

with open('/usr/share/dict/words', 'r') as words_file:
    words = set([word.lower() for word in words_file.read().split()])
    texts, pad = recursively_expand_pad(0, [], words)
    print 'messages = %s, pad = %s' % (texts, pad)
```

Output of running the code:

messages = ['We stand ', 'Probabilistic', 'Secure So', 'This docu', 'MIT devel',
            'NIST anno', 'Diffie-He', 'Public-ke', 'The keys ', 'We hope t'],
pad = [119, 75, 116, 51, 85, 113, 72, 105, 76]

We could have searched these plaintext prefixes on Google, but we decided to continue running our code to choose $p_9, \ldots, p_{18}$ (often the messages we want to decrypt won’t be available online). First we wrote the pad above to possible_pad_bytes[0:9] (so there is only 1 choice for $p_i$, $0 \leq i < 9$), and then we increased msglen to 18. Output of rerunning the code:

messages = ['We stand today on ', 'Probabilistic encr', 'Secure Sockets Lay',
            'This document will', 'MIT developed Kerb', 'NIST announced a c',
            'Diffie-Hellman est', 'Public-key cryptog', 'The keys used to s',
            'We hope this inspi'],
pad = [119, 75, 116, 51, 85, 113, 72, 105, 76, 78, 114, 79, 84, 49, 71,
            101, 71, 88]

Repeating this process $4 \times$, we eventually got the desired output (pasted at the beginning) for all 60 characters. At this point, we manually checked it with online articles to be confident in our decryption. For instance, *We stand today… appears in a well-cited 1976 paper by Whitfield Diffie* and Martin Hellman.