Cryptography - Breaking Single Byte XOR Encryption

Posted Dec 6, 2021

By [Alexander Wells](https://twitter.com/todo)

6 min read

In this post we’ll cover how to decrypt messages that have been XOR encrypted using a single byte key, such as b7. While this might not sound that useful, it’s a pre-cursor to breaking XOR encryption that uses a repeating key, such as 84 d2 7a 09 4c.

This post is also a solution to [challenge 3](https://cryptopals.com/sets/1/challenges/3) on the [cryptopals](https://cryptopals.com/) website. This site is a great resource for hands on learning about cryptography. The technique used to complete this challenge is outlined on that page. This post will explain the same technique with a bit more detail and examples.

1. [How to Break it](https://node-security.com/posts/cryptography-breaking-single-byte-xor-encryption/#how-to-break-it)
2. [English Scoring](https://node-security.com/posts/cryptography-breaking-single-byte-xor-encryption/#english-scoring)
3. [Brute Forcing the Key](https://node-security.com/posts/cryptography-breaking-single-byte-xor-encryption/#brute-forcing-the-key)
4. [What Next?](https://node-security.com/posts/cryptography-breaking-single-byte-xor-encryption/#what-next)

How to Break it

Since the key is only a single byte, the keyspace is very small, only 256 possible keys. As such, brute force is practical and reasonable solution.

We still need to determine which key is actually correct though. We’re assuming that the plain-text we are recovering is English text. After attempting decryption with a key, we’ll need a method to “score” the decrypted data based on how English it is.

English Scoring

Let’s start off by writing a function that can score a String based on how English it is.

We’ll give a String points for the following:

* +2 Points for each occurrence of the 6 most common English characters (and space)
* +1 Point for each occurrence of the next 6 most common English characters
* +1 Point for each occurrence of the 20 most common English Bigrams
* +1 Point for each occurrence of the 20 most common English Trigrams
* +1 Point for each occurrence of the 298 most common English Words

Brute Forcing the Key

Now that we can score a given String on how English it is, we can brute force the key.

As the key is only one byte, there is 256 possible keys to check. For each key we will decrypt the cipher-text with it, and score the decrypted string. The key that produces the best score is most likely to be the correct key.

What Next?

While this function might not seem that useful at first, it will be vital when we want to break repeating key XOR encryption

Cryptography - Breaking Repeating Key XOR Encryption

Posted Dec 11, 2021

By [Alexander Wells](https://twitter.com/todo)

10 min read

In this post we’ll cover how to decrypt messages that have been XOR encrypted using a repeated key, such as 84 d2 7a 09. The method we’ll be using to break the encryption uses statistics (letter frequencies and use of common words, bigrams, and trigrams), so the cipher-text needs to be a decent size otherwise it won’t work. The key used can be any arbitrary number of bytes long, but in general, the shorter the easy it is to break.

This technique requires you to know how to break single byte XOR encryption. If you’re unsure how to do that, please see my post about it [here](https://node-security.com/posts/cryptography-breaking-single-byte-xor-encryption/).

This post is also a solution to [challenge 6](https://cryptopals.com/sets/1/challenges/6) on the [cryptopals](https://cryptopals.com/) website. This site is a great resource for hands on learning about cryptography. The technique used to complete this challenge is outlined on that page. This post will explain the same technique with a bit more detail and visual examples.

1. [Repeating Key XOR Encryption Example](https://node-security.com/posts/cryptography-breaking-repeating-xor-key-encryption/#repeating-key-xor-encryption-example)
2. [Summary](https://node-security.com/posts/cryptography-breaking-repeating-xor-key-encryption/#summary)
3. [How to Break it - Step 1](https://node-security.com/posts/cryptography-breaking-repeating-xor-key-encryption/#how-to-break-it---step-1)
4. [How to Break it - Step 2](https://node-security.com/posts/cryptography-breaking-repeating-xor-key-encryption/#how-to-break-it---step-2)
5. [How to Break it - Step 3](https://node-security.com/posts/cryptography-breaking-repeating-xor-key-encryption/#how-to-break-it---step-3)
6. [Results](https://node-security.com/posts/cryptography-breaking-repeating-xor-key-encryption/#results)

Repeating Key XOR Encryption Example

**Key (4 Byte Length)**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 84 | d2 | 7a | 09 |  |  |  |  |

**Plain-Text**

This is the HEX representation of “ABCDEFGH”:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 |

**Encryption**

XOR the Plain-Text with key

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 41 ⊕ 84 | 42 ⊕ d2 | 43 ⊕ 7a | 44 ⊕ 09 | 45 ⊕ 84 | 46 ⊕ d2 | 47 ⊕ 7a | 48 ⊕ 09 |

**Cipher-Text**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| c5 | 90 | 39 | 4d | c1 | 94 | 3d | 41 |

Summary

As previously mentioned, we’ll use statistics to break the encryption. Because of this, very short cipher-texts such as the example above will not be able to be broken using this method.

The steps we’ll use to break the encryption are as follows:

1. Find the key length, we’ll call this length KEYSIZE.
2. Split the cipher-text into KEYSIZE length sections, and create KEYSIZE different blocks where block 1 contains the 1st byte of every section, block 2 contains the 2nd byte of every section etc.
3. Find the key for each block by performing single byte key XOR decryption.

Let’s get into some more detail on each step.

How to Break it - Step 1

To find the key length, we’ll use some brute force and statistics. We don’t know the KEYSIZE, so we’ll guess that it’s from between 2 and 40 bytes long.

For each KEYSIZE we’ll take the first KEYSIZE worth of bytes, and the second KEYSIZE worth of bytes and find the [hamming distance](https://en.wikipedia.org/wiki/Hamming_distance) between them. We’ll then normalise this value by dividing by KEYSIZE. This should be repeated multiple times and the average taken. That is, do the same process with the second KEYSIZE worth of bytes, and the third KEYSIZE worth of bytes. Then the 3rd and 4th, 4th and 5th etc.

The KEYSIZE with the lowest normalised hamming distance **should** be the KEYSIZE.

Let’s show an example for a KEYSIZE of 2. Below is the first 8 bytes of cipher-text:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1d | 42 | 1f | 4d | 0b | 0f | 02 | 1f |

We need to find the hamming distance between block 1 0x1d42 and block 2 0x1f4d, which is 5. We then need to divide this by KEYSIZE to normalise it. So we are left with 5/2 = 2.5.

We repeat this with block 2 0x1f4d and block 3 0x0b0f, which is 4. 4/2 = 2.

Then again with block 3 0x0b0f and block 4 0x021f, which is 3. 3/2 = 1.5.

We can repeat this as many times as we like if we have more cipher-text. Since we did it 3 times above, we need to add the 3 results together and divide by 3 to get the average hamming distance:

(2.5 + 2 + 1.5) / 3 = 2

We have completed the step for a KEYSIZE of 2. We need to repeat all of this for each KEYSIZE (2-40).

The KEYSIZE with the lowest normalised hamming distance **should** be correct, but it’s not guaranteed. We’ll take the top performing KEYSIZEs and continue with the next step.

How to Break it - Step 2

We’ll now convert the problem into a series of single byte XOR encryption problems. Here’s how to do that.

Let’s say that after Step 1 we think the KEYSIZE is 5. We’ll split the cipher-text into KEYSIZE length sections, and then we’ll create KEYSIZE number of blocks.

The first byte of each section goes into block 1. The second byte of each section goes into block 2. The third byte of each section goes into block 3 etc.

Here is how that would look visually with the first 16 bytes of cipher-text:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1d | 42 | 1f | 4d | 0b | 0f | 02 | 1f |
| 4f | 13 | 4e | 3c | 1a | 69 | 65 | 1f |

**Block 1 (1st byte of each section)**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1d | 0f | 4e | 1f |  |  |  |  |

**Block 2 (2nd byte of each section)**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 42 | 02 | 3c |  |  |  |  |  |

**Block 3 (3rd byte of each section)**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1f | 1f | 1a |  |  |  |  |  |

**Block 4 (4th byte of each section)**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 4d | 4f | 69 |  |  |  |  |  |

**Block 5 (5th byte of each section)**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0b | 13 | 65 |  |  |  |  |  |

We’re now ready to find the key.

How to Break it - Step 3

For each of the blocks in Step 2, we’ll break them as if they are single byte key XOR encrypted, which if we have the right KEYSIZE, they are. After doing so, let’s say we get the following results:

* Block 1 Key = 69
* Block 2 Key = 20
* Block 3 Key = 69
* Block 4 Key = 6e
* Block 5 Key = 69

Our final decryption key would be the concatenation of all these, so it would be 69 20 69 6e 69.

Back at the end of Step 1 we said we didn’t know for sure what the KEYSIZE was, only that we had a good idea. For each of the best performing KEYSIZEs, we do Step 2 & 3. This will give a decryption key for each of the KEYSIZEs.

To find out which one is most likely to be correct, we decrypt the cipher-text using each decryption key, then score the results based on how English it is. Scoring English was covered in the breaking single byte XOR key encryption, and the exact same method is used here.

And that’s all there is to it! You now know how to break repeated key XOR encryption. As a side note, Step 1 isn’t technically necessary as it only serves to reduce the amount of computation done in the subsequent steps.

Results

Using this exact technique on the challenge cipher-text yielded the following results.

The best performing KEYSIZEs were:

* KEYSIZE of 2 (Hamming Distance: 2)
* KEYSIZE of 3 (Hamming Distance: 2.66)
* KEYSIZE of 29 (Hamming Distance: 2.79)
* KEYSIZE of 5 (Hamming Distance: 2.79)
* KEYSIZE of 18 (Hamming Distance: 2.94)

English scores after doing Steps 2 & 3 with each of the above KEYSIZEs:

* KEYSIZE of 29 (English Score: 4782)
* KEYSIZE of 18 (English Score: 1768)
* KEYSIZE of 2 (English Score: 1727)
* KEYSIZE of 5 (English Score: 1697)
* KEYSIZE of 3 (English Score: 1670)

# Crypto - Part 1. Breaking XOR Encryption.

13 Apr 2017

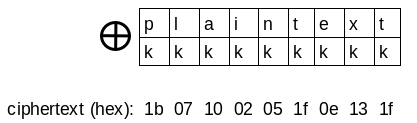
# Introduction

In the Crypto series of posts I’ll try to explain different encryption algorithms, implement them in code and then try to break them. They’re also writeups for the [cryptopals](http://cryptopals.com/) crypto challenges and I recommend trying to solve them youtself before reading this and other crypto posts.

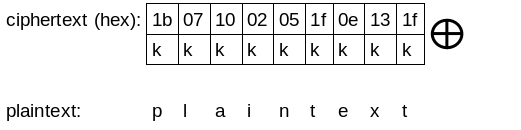
I’m not a cryptographer, nor am I an expert in programming! The purpose of these posts (and the blog in general) is for me to write down what I’ve learned so it can be useful to others (and for others to point out my mistakes!).

# Single-byte XOR cipher

This cipher applies the XOR operation on every byte of the plaintext with the same one-byte key. For example:  
key = ‘k’ ; plaintext = ‘plaintext’ ; ciphertext = kkkkkkkkk XOR plaintext



And to decrypt the message XOR every byte of the ciphertext with the key:  
key = ‘k’ ; plaintext = kkkkkkkkk XOR ciphertext



Below is a function that does XOR of two strings of equal length:

def xor(str1, str2):

if len(str1) != len(str2):

raise "XOR EXCEPTION: Strings are not of equal length!"

s1 = bytearray(str1)

s2 = bytearray(str2)

result = bytearray()

for i in range( len(s1) ):

result.append( s1[i] ^ s2[i] )

return str( result )

The function for encryption and decryption:

def single\_byte\_xor(plaintext, key):

if len(key) != 1:

raise "KEY LENGTH EXCEPTION: In single\_byte\_xor key must be 1 byte long!"

return xor(plaintext, key\*len(plaintext))

# Break the single-byte XOR cipher

This cipher is essentialy a substitution cipher, so it’s vulnerable to frequency analysis and because the key is only one byte it’s also easy to bruteforce (there are only 256 possible keys…).  
The frequency analysis is suitable for longer messages, so I’ll implement only the bruteforce method which always works regardless of the message length.

But when we try every one of the 256 possible keys, how do we know that the produced output is the actual plaintext? If the plaintext is written in english, we need a way to test if a given string is an english text.

To decide wether a string is an english text I’ll use some the following rules:

1. The string contains only ascii printable characters
2. Letter ‘E’ and space are the most frequent characters (for sufficiently long messages)
3. The letters E,T,A,O,I,N make up around 40% of the text (those are the most frequent letters in the english language)
4. The digraphs cj, fq, gx, hx, jf, jq, jx, jz, qb, qc, qj, qk, qx, qz, sx, vf, vj, vq, vx, wx, xj, zx never occur in english words
5. Punctuation makes up to 2%-3% of the text (for short messages up to 10%).
6. Has at least one vowel (every word should have at least one vowel)
7. Around 80%-90% or more of the text should be made up of letters

I found the digraphs using the following script and [this dictionary](https://github.com/dwyl/english-words/blob/master/words.txt) which contains 355k english words.

import string, itertools

f = open('words.txt','r')

file = f.read().lower()

f.close()

digraphs = []

for digraph in itertools.product(string.lowercase, repeat=2):

d = ''.join(digraph)

if file.count(d) == 0:

digraphs.append(d)

print "Digraphs: ", digraphs

For the punctuation statistic similar script was used and a 1000 page ebook.

import string

f = open('book.txt','r')

file = f.read().lower()

f.close()

cnt = 0

for char in string.punctuation:

cnt += file.count(char)

print "Punctuation makes up %f %% of the text!" % ( float(cnt)\*100/len(file) )

And below is the code I wrote that checks if a given string is an english text, by using the rules mentioned above. It’s not perfect but most of the time works well enough.

import string

def has\_nonprintable\_characters( text ):

for char in text:

if char not in string.printable:

return True

return False

def has\_vowels( text ):

vowels = list("eyuioa")

for char in vowels:

if char in text:

return True

return False

def has\_forbidden\_digraphs( text ):

forbidden\_digraphs = ['cj','fq','gx','hx','jf','jq','jx','jz','qb','qc','qj','qk','qx','qz','sx','vf','vj','vq','vx','wx','xj','zx']

for digraph in forbidden\_digraphs:

if digraph in text:

return True

return False

def has\_necessary\_percentage\_frequent\_characters( text, p=38 ):

most\_frequent\_characters = list("etaoin")

cnt = 0

for char in most\_frequent\_characters:

cnt += text.count(char)

percent\_characters = float(cnt)\*100/len(text)

# The most\_frequent\_characters shoud be more than 38% of the text.

# For short messages this value may need to be lowered.

if (percent\_characters < p):

return False

return True

def has\_necessary\_percentage\_punctuation( text, p=10 ):

cnt = 0

for char in string.punctuation:

cnt += text.count(char)

# Punctuation characters should be no more than 10% of the text.

punctuation = float(cnt)\*100/len(text)

if punctuation > 10:

return False

return True

def has\_english\_words( text ):

most\_frequent\_words = ['the', 'and', 'have', 'that', 'for',

'you', 'with', 'say', 'this', 'they', 'but', 'his', 'from',

'that', 'not', "n't", 'she', 'what', 'their', 'can', 'who',

'get', 'would', 'her', 'make', 'about', 'know', 'will',

'one', 'time', 'there', 'year', 'think', 'when', 'which',

'them', 'some', 'people', 'take', 'out', 'into','just', 'see',

'him', 'your', 'come', 'could', 'now', 'than', 'like', 'other',

'how', 'then', 'its', 'out', 'two', 'more ,these', 'want',

'way', 'look', 'first', 'also', 'new', 'because', 'day',

'more', 'use', 'man', 'find', 'here', 'thing', 'give', 'many']

for word in most\_frequent\_words:

if word in text:

return True

return False

def is\_english( input\_text ):

text = input\_text.lower()

if has\_nonprintable\_characters( text ):

return False

# If the text contains one of the most frequent english words

# it is very likely that it's an english text

if has\_english\_words( text ):

return True

if not has\_vowels( text ):

return False

if has\_forbidden\_digraphs( text ):

return False

if not has\_necessary\_percentage\_frequent\_characters( text ):

return False

if not has\_necessary\_percentage\_punctuation( text ):

return False

return True

Now we are ready to construct the bruteforce function.

def break\_single\_byte\_xor( ciphertext ):

keys = []

plaintext = []

for key in range(256):

text = single\_byte\_xor( ciphertext , chr(key))

if is\_english( text ):

keys.append( chr(key) )

plaintext.append( text )

# There might be more than one string that match the rules of the is\_english function.

# Return all those strings and their corresponding keys and inspect visually to

# determine which is the correct plaintext.

return keys, plaintext

Lets test it!

msg = 'This is a very secret message!'

key = '\x0f'

ciphertext = single\_byte\_xor(msg, key)

k, pt = break\_single\_byte\_xor( ciphertext )

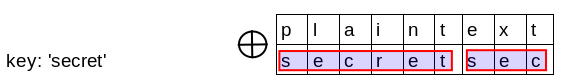
print "Keys: ", k

print "Plaintexts: ", pt

The output is:  
Keys: [‘\x0f’]  
Plaintexts: [‘This is a very secret message!’]

# Repeating-key XOR cipher

This cipher uses a key that is more than one byte long. The key is repeated until it matches the length of the message.  
For example: key=’secret’ ; plaintext = ‘plaintext’ ; ciphertext = secretsec XOR plaintext



Here is the implementation:

def repeating\_key\_xor(plaintext, key):

if len(key) == 0 or len(key) > len(plaintext):

raise "KEY LENGTH EXCEPTION!"

ciphertext\_bytes = bytearray()

plaintext\_bytes = bytearray(plaintext)

key\_bytes = bytearray(key)

# XOR every byte of the plaintext with the corresponding byte from the key

for i in range( len(plaintext) ):

k = key\_bytes[i % len(key)]

c = plaintext\_bytes[i] ^ k

ciphertext\_bytes.append(c)

return str(ciphertext\_bytes)

# Breaking the repeating-key XOR cipher

This one is trickier. There are mainly two steps here:

1. Find the key size
2. Crack the key

Finding the key size is done by the following algorithm:

1. Make a guess about the key length
2. Divide the ciphertext by blocks, each with length equal to the one chosen at step 1
3. Calculate the hamming distance between the first few blocks (I had best results with 4-5 blocks) and then take the average
4. Normalize the hamming distance by dividing it by the chosen key length
5. The key length that gives the smallest normalized hamming distance is PROBABLY the actual key length (if it’s not, it is usually one of the three with the smallest normalized hamming distance)

Hamming distance is equal to the number of bits by which two strings of equal length differ. Take this two bytes:

00101010  
01000010

The hamming distance between them is 3, because they differ by three bits - the 4th, the 6th and the 7th (counting from the least significant). Calculating the hamming distance is easy - just XOR the two strings/bytes and count the number of ones in the resuting string.

def hamming\_distance(str1, str2):

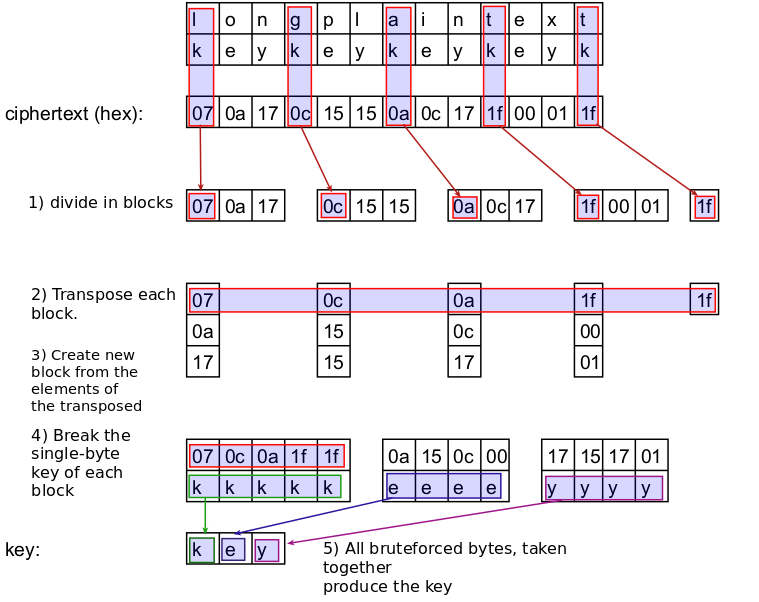
result = xor(str1, str2)

return bin( int( result.encode('hex'), 16) ).count('1')

To crack the key there are several steps:

1. Divide the ciphertext by blocks with equal length, same as the length of the key
2. Transpose the blocks. That is, make a new block from the first bytes of the blocks, then a second block containing the second bytes of the blocks and so on…
3. Each of the transposed blocks contains bytes that are encrypted with the same byte. That is the single-byte XOR cipher! And we already know how to break it.
4. Crack the single-byte key for each of the transposed blocks.
5. All bytes taken together produce the key

Lets illustrate those steps:



By now I hope you see how this method works :)

Here is the function I wrote for finding the probable key length:

def find\_xor\_keysize( ciphertext, hamming\_blocks, minsize=2, maxsize=10 ):

hamming\_dict = {} # <keysize> : <hamming distance>

if (hamming\_blocks\*maxsize) > len(ciphertext):

raise "OUT OF BOUND EXCEPTION! Lower the hamming\_blocks or the key maxsize!"

for key\_length in range(minsize, maxsize):

# Take the first 'hamming\_blocks' blocks

# with size key\_length bytes

blocks = []

for i in range(hamming\_blocks):

blocks.append( ciphertext[i\*key\_length : (i+1)\*key\_length] )

# Calculate the hamming distance between the blocks

# (first,second) ; (first,third) ; (first,fourth)

# (second, third) ; (second, fourth)

# (third, fourth) ; There are sum(1,hamming\_blocks-1) combinations

hd = [] # hamming distance

for i in range( hamming\_blocks - 1 ):

for j in range( i+1, hamming\_blocks ):

hd.append( hamming\_distance(blocks[i], blocks[j] ))

hd\_average = float(sum(hd))/len(hd)

hd\_normalized = hd\_average/key\_length

hamming\_dict[key\_length] = hd\_normalized

# Get sorted (ascending order) list of tuples. Sorted by dictionary value (i.e. hamming distance)

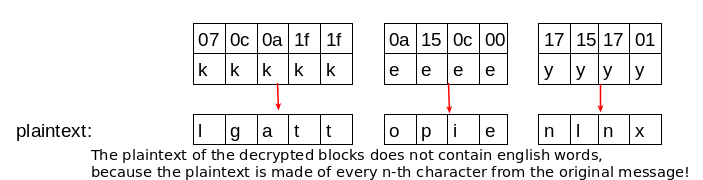
sorted\_list\_tuples = sorted(hamming\_dict.items(), key=lambda x: x[1])

# One of the three keys that produced the lowest hamming distance

# is likely the actual size

return [ sorted\_list\_tuples[0][0], sorted\_list\_tuples[1][0], sorted\_list\_tuples[2][0] ]

The cracking step turns out to be a little harder. The transposed blocks are every n-th character of the ciphertext and so their corresponding plaintext isn’t composed of english words. This makes it harder to distinguish which one-byte key produces the correct plaintext. That’s why it’s necessary to have a long message (longer message -> longer blocks) to be able to use statistical methods on the transposed blocks.



1) I take every possible one-byte key for a single block and test if it produces ascii printable output. If it does, I store it in a list (that way I filter out may invalid keys). There is one such list for every block, which contains the keys that produce printable output.

block1: keys[a,b,c,d]  
block2: keys[1,2,3]  
block3: keys[w,x,y,z]

2) Then I store all those lists in another list. This list now contains all possible one-byte keys for every block.

list: [ [a,b,c,d], [1,2,3], [w,x,y,z] ]

3) After that I generate all possible combinations of the collected single-byte keys (with key length as returned from find\_xor\_keysize) using that list.

a1w  
a1x  
a1y  
a1z  
a2w  
a2x  
and so on…

4) Try every one of the produced multi-byte keys against the whole ciphertext, and test if the output is an english text.

ciphertext  
xor  
a1wa1wa1wa  
=  
output

test if output is english text

import itertools

def divide\_text\_by\_blocks(text, block\_size):

blocks = []

num\_blocks = len(text)/block\_size

for i in range(num\_blocks):

blocks.append( text[i\*block\_size : (i+1)\*block\_size] )

return blocks

def transpose( blocks ):

transposed = []

block\_size = len(blocks[0])

num\_blocks = len(blocks)

for i in range(block\_size):

tmp = []

for j in range(num\_blocks):

# tmp is composed of the i-th character of every block

tmp.append( blocks[j][i] )

transposed.append( ''.join(tmp) )

return transposed

def has\_necessary\_percentage\_letters( text,p=80 ):

characters = string.letters + ' '

cnt = 0

for char in characters:

cnt += text.count(char)

percent\_characters = float(cnt)\*100/len(text)

# The characters shoud be more than 38% of the text.

if (percent\_characters < p):

return False

return True

def is\_printable\_text( text ):

text = text.lower()

if has\_nonprintable\_characters(text):

return False

if not has\_necessary\_percentage\_punctuation( text ):

return False

if not has\_necessary\_percentage\_letters( text ):

return False

if not has\_vowels( text ):

return False

return True

def break\_repeat\_key\_xor( ciphertext ):

# Tweaking this is useful. Lower value (0.03-0.05) helps find longer keys

# Higher value (0.1 - 0.15) helps find shorter keys

hamming\_blocks = int(len(ciphertext)\*0.06)

key\_sizes = find\_xor\_keysize(ciphertext, hamming\_blocks , 2)

print "Key sizes: ", key\_sizes

for ks in key\_sizes:

print "Current key size: ", ks

blocks = divide\_text\_by\_blocks(ciphertext, ks)

transposed = transpose(blocks)

all\_keys = [] # list of lists. One list for every block. The list has all possible one-byte keys for the block.

for block in transposed:

block\_keys = [] # store all possible one-byte keys for a single block

for key in range(256):

text = single\_byte\_xor( block , chr(key))

if is\_printable\_text(text):

block\_keys.append(chr(key))

print block\_keys

all\_keys.append(block\_keys)

real\_keys = [] # Stores keys with size ks. Generated from all possible combinations of one-byte keys contained in all\_keys

for key in itertools.product(\*all\_keys):

real\_keys.append( ''.join(key) )

print "Keys to try: ", len(real\_keys)

# Try every possible multy-byte key.

for key in real\_keys:

text = repeating\_key\_xor(ciphertext,key)

if is\_english(text):

print "Plaintext: " ,text

print "Key: ", key

raw\_input()

print "=================="

Lets test it!

msg = '''In today's electronic communication forums, encryption can be very

mportant! Do you know for a fact that when you send a message to someone else,

that someone hasn't read it along the way? Have you ever really sent something

you didn't want anyone reading except the person you sent it to? As more and

more things become online, and "paperless" communication predictions start

coming true, it's all the more reason for encryption. Unlike the normal U.S.

Mail where it is a crime to tamper with your mail, email-reading can commonly

go unnoticed on electronic pathways as your message hops from system to system

on its route towards its final destination. Just think, the average Internet

letter makes at least two hops before it reaches its recipient, usually more.

Even on public BBS's, your mail is usually stored in plaintext. '''

key = "r!ck\_@nd\_m0rty"

c = repeating\_key\_xor(msg, key)

break\_repeat\_key\_xor2(c)

And the output is:

Key sizes: [14, 7, 2]

Current key size: 14

['r']

['!']

['c']

['k']

['\_']

['@']

['n']

['d']

['\_']

['m']

['0', '7']

['r']

['t']

['y']

Keys to try: 2

Plaintext: In today's electronic communication forums, encryption can be very

mportant! Do you know for a fact that when you send a message to someone else,

that someone hasn't read it along the way? Have you ever really sent something

you didn't want anyone reading except the person you sent it to? As more and

more things become online, and "paperless" communication predictions start

coming true, it's all the more reason for encryption. Unlike the normal U.S.

Mail where it is a crime to tamper with your mail, email-reading can commonly

go unnoticed on electronic pathways as your message hops from system to system

on its route towards its final destination. Just think, the average Internet

letter makes at least two hops before it reaches its recipient, usually more.

Even on public BBS's, your mail is usually stored in plaintext.

Key: r!ck\_@nd\_m0rty

==================

mportant! Do'you know for f fact that whbn you send a jessage to sombone else,

thas someone hasn t read it aloig the way? Hfve you ever rbally sent sombthing

more things bbcome online, fnd "paperless% communicatioi predictions ttartand

coming tuue, it's all she more reasoi for encryptihn. Unlike thb normal U.S.

Jail where it ns a crime to samper with yorr mail, email\*reading can chmmonly

go unnhticed on elecsronic pathwayt as your messfge hops from tystem to systbm

on its routb towards its ainal destinatnon. Just thiik, the averagb Internet

letser makes at lbast two hops eefore it reacoes its recipibnt, usually mhre.

Even on prblic BBS's, yhur mail is usrally stored ii plaintext.

Key: r!ck\_@nd\_m7rty

==================

Current key size: 7

[]

[]

[]

[]

[]

[]

[]

Keys to try: 0

Current key size: 2

[]

[]

Keys to try: 0

And it worked! There were two possible keys for one of the blocks - [‘0’, ‘7’]. If the message was shorter there would’ve beem many possible keys with thousands of combinations (or none).

msg = '''In today's electronic communication forums, encryption can be very

mportant! Do you know for a fact that when you send a message to someone else,

that someone hasn't read it along the way? '''

And the output is:

Key sizes: [14, 4, 7]

Current key size: 14

['r']

['\x0c', '\r', '!', '#', '%', "'", ',', '-', '6', '7']

['c']

['k', '|', '}']

['B', 'Q', 'R', 'S', 'X', 'Y', '[', ']', '^', '\_', 'b', 'c', 'q', 'r', 's', 'x',

'y', '{', '}', '~', '\x7f']

[]

['n']

['d']

['I', 'J', 'X', 'Z', ']', '\_']

['F', 'G', 'm', 'u']

['0']

['^', 'e', 'r']

['t']

['N', 'O', 'X', 'Y', '[', ']', '^', '\_', 'n', 'o', 'x', 'y', '{', '}', '~']

Keys to try: 0

Current key size: 4

[]

[]

[]

[]

Keys to try: 0

Current key size: 7

[]

[]

[]

[]

['X', 'Y', '[', '^', '\_', 'r']

[]

['n']

Keys to try: 0

As you can see, for some blocks there are many possible keys, and for others none were found.