Homework Assignment-1

Cryptography and Network Security

Name	Syed Noor- A- Manik
ID	1930146
Course Code	CSE 406
Section	01
Semester	Summer' 2023
Submitted to	Mohammad Noor Nabi

Solution 1

a)

A shift cipher is a type of substitution cipher where each letter in the plaintext is replaced by a letter of some fixed number of positions down the alphabet. For example, if the key is 3, then A becomes D, B becomes E, and so on. To decrypt a ciphertext, you need to know the key and reverse the process.

One way to break a shift cipher without knowing the key is to use frequency analysis. This is since some letters are more common than others in each language. For example, in English, the letter E is the most frequent letter, followed by T, A, O, I, and N. So, if you count how many times each letter appears in the ciphertext, you can guess which one corresponds to E and then find the key.

For example, in your ciphertext:

xultpaajcxitltlxaarpjhtiwtgxktghidhipxciwtvgtpilpitghlxiwiwtxgqadds

The most frequent letter is T, which appears 9 times. If we assume that T represents E in the plaintext, then we can find the key by subtracting their positions in the alphabet:

key = T - E = 20 - 5 = 15

So, the key is 15. To decrypt the ciphertext, we need to shift each letter 15 positions back in the alphabet. For example, X becomes I, U becomes F, L becomes W, and so on. Doing this for the whole ciphertext, we get:

Ifwealunitewewillcausetheriverstostainthegreatwaterswiththeirblood

“If we all unite we will cause the rivers to stain the great waters with their blood”.

b)

One of the great figures of early Native resistance to colonization was **Tecumseh,** a Shawnee leader, who earned a reputation for his skills in fighting white settlers and militias in the Midwest. He and his brother worked toward the unification of Indians to struggle collectively against the encroachment on their lands by colonists, as they expanded westward. Here he speaks to the Osages about the struggle against the colonists.

Solution 2

a)

To decrypt the text, you need to use the formula (a^-1 x - b) mod 26, where a^-1 is the modular multiplicative inverse of a, and b is the same as in the encryption formula. For a = 7 and b = 22, you can find that a^-1 = 15 by using an online calculator1. Then, you can apply the formula to each letter in the text, using the numeric values of the letters (A = 0, B = 1, …, Z = 25). For example, the first letter F has the value 5, so you can calculate (15 x (5 - 22)) mod 26 = 19, which corresponds to the letter T. If you repeat this process for all the letters, you will get the decrypted text:

thisisasecretmessagethatyouhavedecryptedusingaffinecipher

b)

The line is from a famous quote by Julius Caesar: “Veni, vidi, vici” which means “I came, I saw, I conquered” in Latin. The quote is often used to express a swift and decisive victory.

Solution 3

Frequency analysis is an important part of cryptanalysis. It can be used to crack many of the classical ciphers, including the affine cipher. Frequency analysis is because certain letters or groups of letters appear more often than others in a given language. For example, in English, the most common letter is E, followed by T, A, O, I, N, etc2. By counting the frequency of each letter in a ciphertext and comparing it with the expected frequency of each letter in the plaintext language, one can make some educated guesses about the possible mappings between plaintext and ciphertext letters.

To break an affine cipher using frequency analysis, one needs to find two plaintext-ciphertext pairs of letters. This can be done by looking at the most frequent letters in the ciphertext, and assuming they correspond to the most frequent letters in the plaintext language. For example, if B is the most frequent letter in the ciphertext, and U is the second most frequent letter, one can assume that B corresponds to E and U corresponds to T in plaintext English. Once two pairs are found, one can solve a system of two linear equations to find the values of a and b. For example, if B corresponds to E and U corresponds to T, then we have:

E(x) = (ax + b) mod 26 E(E) = B -> 4 = (a x 4 + b) mod 26 E(T) = U -> 19 = (a x 19 + b) mod 26

Solving for a and b gives:
a = 9 b = 14
Therefore, the encryption key is (9, 14). To find the decryption key, we need to find the modular multiplicative inverse of 9 modulo 26. This can be done by using an online calculator3 or by using the extended Euclidean algorithm. The result is:
a^-1 = 3
Therefore, the decryption key is (3, 14). The decryption function is:
D(x) = 3 (x - 14) mod 26
Using this function, we can decrypt any ciphertext created with the affine cipher and the encryption key (9, 14).

Solution 4

To perform modular arithmetic operations, such as addition, subtraction, multiplication, and division, we can use the following rules:

To add two numbers modulo m, we add them normally and then take the remainder when dividing by m. For example, (7 + 9) mod 12 = 16 mod 12 = 4.

To subtract two numbers modulo m, we subtract them normally and then take the remainder when dividing by m. For example, (9 - 7) mod 12 = 2 mod 12 = 2.

To multiply two numbers modulo m, we multiply them normally and then take the remainder when dividing by m. For example, (7 x 9) mod 12 = 63 mod 12 = 3.

To divide two numbers modulo m, we need to find the multiplicative inverse of the divisor modulo m. This is a number that satisfies the equation 1 = a x a^-1 mod m. For example, the multiplicative inverse of 7 modulo 12 is 7, because 1 = 7 x 7 mod 12. Then we multiply the dividend by the multiplicative inverse and take the remainder when dividing by m. For example, (9 / 7) mod 12 = (9 x 7) mod 12 = (63) mod 12 = 3.

Using these rules, we can compute the results of your problems without a calculator:

a. 15 · 29 mod 13 = (2 * 3) |13| = 6 |13|

b. 2 · 29 mod 13 = (2 * 3) |13| = 6 |13|

c. 2 · 3 mod 13 = (2 * 3) |13| = 6 |13|

d. −11 · 3 mod 13, -11 is in equivalence class {… -11, 2, 15, …}, so; = (2 * 3) |13| = 6

The results should be given in the range from 0 to modulus -1, which is 0 to 12 in this case.

We may notice that some of the results are equal. This is because modular arithmetic has some interesting properties that relate different parts of the problem. For example:

If two numbers are congruent modulo m, meaning they have the same remainder when divided by m, then their multiples are also congruent modulo m. For example, since 15 ≡ -11 (mod 13), we have 15 · x ≡ -11 · x (mod 13) for any x. This explains why part a and part d have the same result.

If two numbers are congruent modulo m, then their sums and differences are also congruent modulo m. For example, since 15 ≡ -11 (mod 13) and 29 ≡ -1 (mod 13), we have 15 + x ≡ -11 + x (mod 13) and 15 - x ≡ -11 - x (mod 13) for any x. This explains why part b and part c have the same result.

Solution 5

a. The addition and multiplication tables for Z4 are:

+	0	1	2	3
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

x	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	**0**[1234567]	**2**
3	**0**	**3**	**2**	**1**

b. The addition and multiplication tables for Z5 are:

+	**0**	**1**	**2**	**3**	**4**
0	**0**	**1**	**2**	**3**	**4**
1	**1**	**2**	**3**	**4**	**0**^
2^	**2**^	**3**^	**4**^	**0**^	**1**^
3^	**3**^	**4**^	**0**^	**1**^	**2**^
4^	**4**^	**0**^	**1**^	**2**^	**3**^

x	^62	^63	^64	^65	^66
^67	^68	^69	^70	^71	^72
^73	^74	^75	^76	^77	^78
^79	^80	^81	^82	^83	^84
^85	^86	^87	^88	^89	^90
^91	^92	^93	^94	^95	^96

c. The addition and multiplication tables for Z8 are:

+ ||94 ||95 ||96 ||97 ||98 ||99 ||100 |
- |- |- |- |- |- |- |- |
	101		102		103		104		105		106		107		108
	109		110		111		112		113		114		115		116
	117		118		119		120		121		122		123		124
	125		126		127		128		129		130		131		132
	133		134		135		136		137		138		139		140
	141		142		143		144		145		146		147	
148 |

x ^149 ^150 ^151 ^152 ^153 ^154 ^155 ^156
- |- |- |- |- |- |- |- |-
^157 ^158 ^159 ^160 ^161 ^162 ^163 ^164
^165 ^166 ^167 ^168 ^169 ^170 ^171 ^172
^173 ^174 ^175 ^176 ^177 ^178 ^179 ^180
^181 ^182 ^183 ^184 ^185 ^186 ^187 ^
188
189
190
191
192

d. The elements in Z4 and Z8 without a multiplicative inverse are those that are not coprime with the modulus, i.e., those that have a common factor other than 1 with the modulus. In Z4, these are *0* and *2*, since they are both divisible by *2*, which is also a factor of *4*. In Z8, these are *0*, *2*, *4*, and *6*, since they are all divisible by either *2* or *4*, which are both factors of *8*. A multiplicative inverse exists for all nonzero elements in Z5 because *5* is a prime number, which means that it has no factors other than *1* and itself. Therefore, any nonzero element in Z5 is coprime with *5* and has a multiplicative inverse.

e. The multiplicative inverse of *5* in Z11 is the integer x such that $5x \equiv 1 \pmod{11}$, i.e., the remainder of dividing 5x by 11 is 1. One way to find this x is to use the extended Euclidean

algorithm, which finds integers a and b such that ax + by = gcd(x,y). If x and y are coprime, then gcd(x,y) = 1 and we have ax + by = 1. Modulo y, this becomes ax ≡ 1 (mod y), so a is the multiplicative inverse of x modulo y. Applying this algorithm to x = 5 and y = 11, we get:

11 = 2 * 5 + *1*
5 = *1* * *1* + *0*

Working backwards, we have:

1 = *11* - *2* * *5*

Modulo *11*, this becomes:

1 ≡ - *2* * *5* (mod *11*)

Therefore, the multiplicative inverse of *5* in Z11 is - *2*, which is equivalent to *9*. You can check that this is correct by multiplying it by *5* and taking the remainder modulo *11*:

9 * *5* = *45*
45 mod *11* = *1*

Solution 6

OTP stands for. OTP is short for one-time pad, which is an encryption technique that uses a random key of the same length as the message to encrypt and decrypt it. The key is only used once and then discarded. This way, the encryption is theoretically unbreakable, if the key is truly random, never reused, and kept secret1.
However, if the key is reused for multiple messages, then the encryption becomes vulnerable to various attacks. One of the most common attacks is called the crib-dragging attack, which exploits the fact that XOR-ing two ciphertexts that are encrypted with the same key will result in the XOR of the two plaintexts2.

For example, suppose Alice sends two messages to Bob using the same 256-bit key:

c1 = m1 XOR k c2 = m2 XOR k

where c1 and c2 are the ciphertexts, m1 and m2 are the plaintexts, and k is the key.

Eve intercepts both ciphertexts and computes:
c1 XOR c2 = (m1 XOR k) XOR (m2 XOR k) = m1 XOR m2

Now Eve has the XOR of the two plaintexts, which reveals some information about them. For instance, if Eve knows that one of the messages starts with “Hello”, she can XOR it with the first five bits of c1 XOR c2 and obtain the first five bits of the other message. This is called a crib, which is a known or guessed part of the plaintext2.
By using various cribs, such as common words, phrases, or patterns, Eve can gradually recover more and more bits of both plaintexts. This process is called crib-dragging because Eve drags the crib along the XOR of the plaintexts and tries to find matches2.
There are also other methods to break OTP-like encryption with key reuse, such as statistical analysis, frequency analysis, or known-plaintext attacks. The main idea is that reusing the same key for multiple messages reduces the randomness and security of the encryption and leaks information about plaintexts345. Therefore, it is very important to never reuse a key for OTP-like encryption, or any other stream cipher that relies on XOR-ing a keystream with the plaintext. Otherwise, the encryption can be easily broken by an attacker.

Solution 7

To compute the first two output bytes of the LFSR of degree 8 and the feedback polynomial from Table 2.3, we need to use the following formula1:

a_ {i+n} = \sum_ {j=0}^ {n-1} c_ {j}a_ {i+j}

where n is the degree of the LFSR, c_j are the coefficients of the feedback polynomial, and a_i are the bits of the output sequence.

The initialization vector (IV) is the initial state of the LFSR, which is given as FF in hexadecimal notation. This means that the first 8 bits of the output sequence are 11111111.

The feedback polynomial from Table 2.3 for degree 8 is x^8 + x^4 + x^3 + x + 12. This means that the coefficients are c_0 = c_3 = c_4 = c_7 = 1 and c_1 = c_2 = c_5 = c_6 = 0.

Using the formula, we can compute the next bit of the output sequence as follows:

a_8 = c_0 a_0 + c_1 a_1 + … + c_7 a_7 a_8 = 1 * 1 + 0 * 1 + … + 1 * 1 a_8 = 1 + 0 + … + 1 a_8 = 0 (mod 2)

Similarly, we can compute the next bits as:

a_9 = c_0 a_1 + c_1 a_2 + … + c_7 a_8 a_9 = 1 * 1 + 0 * 1 + … + 1 * 0 a_9 = 1 + 0 + … + 0 a_9 = 0 (mod 2)

a_{10} = c_0 a_2 + c_1 a_3 + … + c_7 a_9 a_{10} = 1 * 1 + 0 * 1 + … + 1 * 0 a_{10} = 1 + 0 + … + 0 a_{10} = 0 (mod 2)

a_{11} = c_0 a_3 + c_1 a_4 + … + c_7 a_{10} a_{11} = 1 * 1 + 0 * 1 + … + 1 * 0 a_{11} = 1 + 0 + … + 0 a_{11} = 1 (mod 2)

a_{12} = c_0 a_4 + c_1 a_5 + … + c_7 a_{11} a_{12} = 1 * 1 + 0 * 1 + … + 1 * 1 a_{12} = 1+ 0+ …+ 1 a_{12} = 0 (mod 2)

a_{13} = c_0 a_5+ c_1 a6+ …+c7 a12 a13= 10+…+10 a13= 01 (mod 2)

a14= c0 a6+ c1 a7+…+c7 a13 a14= 10+…+01 a14= 00 (mod 2)

a15= c0 a7+ c1 a8+…+c7 a14 a15= 11+…+00 a15= 01 (mod 2)

Therefore, the first two output bytes of the LFSR are:

11111111|00010010

or in hexadecimal notation:

FF|12

Solution 8

A stream cipher is a type of symmetric encryption that uses a key stream to encrypt or decrypt a message. A key stream is a sequence of bits that is combined with the message using a bitwise operation, such as XOR. A linear feedback shift register (LFSR) is a device that can generate a key stream by shifting the bits in a register and applying a feedback function to some of the bits. The feedback function is usually defined by a polynomial, called the feedback polynomial1.

To attack a stream cipher that uses an LFSR as a key stream generator, you need to find the feedback polynomial and the initial state of the LFSR. There are different methods to do this, depending on how much information you have about the plaintext, ciphertext, and keystream.

One method is called the Berlekamp-Massey algorithm2, which can find the shortest LFSR and feedback polynomial that produce a given keystream sequence. This algorithm requires at least 2n bits of keystream, where n is the degree of the LFSR. The algorithm works by iteratively updating a candidate polynomial and its degree until it matches the keystream sequence.

Another method is called the correlation attack2, which exploits the statistical dependence between the keystream and the output of one or more individual LFSRs. This method requires some knowledge of the structure of the stream cipher, such as how many LFSRs are used and how they are combined. The idea is to find a linear combination of the LFSRs that has a high correlation with the keystream, and then use this information to recover the individual LFSRs.

To answer your specific questions:

a.

This depends on which method you use and what parameters you assume for the LFSR. For example, if you use the Berlekamp-Massey algorithm and assume that the LFSR has a degree of 512, then you need at least 1024 bits of keystream to launch a successful attack. If you use the correlation attack and assume that the stream cipher uses four LFSRs with degrees 128, 129, 130, and 131, then you need about 2^128 plaintext/ciphertext bit pairs to launch a successful attack3.

b.

I will describe the steps of the Berlekamp-Massey algorithm in detail, since it is simpler and more general than the correlation attack. The formulae that need to be solved are:

$$a_{i+n} = \sum_{j=0}^{n-1} c_{j}a_{i+j}$$

where n is the degree of the LFSR, c_j are the coefficients of the feedback polynomial, and a_i are the bits of the keystream sequence.

The steps of the algorithm are:
Initialize C(D) = 1, L = 0, C*(D) = 1, d* = 1, m = -1, n = 0.
Compute the discrepancy $d = \sum_{i=0}^{L} c_i a_{n-i}$.
If d != 0, do the following:
T(D) = C(D)
$C(D) = C(D) - d * (d^*)^{-1} * C^*(D) * D^{n-m}$
If L <= n/2, then L = n + 1 - L, C*(D) = T(D), d* = d, m = n.
Increment n by 1.
Repeat steps 2-4 until n >= 2L or n >= N (the length of the keystream sequence).
Return C(D) and L as the feedback polynomial and degree of the shortest LFSR.

c.

The key in this system is usually composed of two parts: the feedback polynomial and the initial state of the LFSR. The feedback polynomial determines how the bits in the register are updated, while the initial state determines what bits are in the register at first.
It doesn't make sense to use only the initial contents of the LFSR as the key or as part of it because they can be easily recovered from a few bits of keystream using linear algebra or brute force methods. Moreover, using only one part of the key makes it vulnerable to attacks that exploit some properties of LFSRs, such as linearity or periodicity.

Solution 9

A stream cipher is a type of symmetric encryption that uses a keystream to encrypt or decrypt a message. A keystream is a sequence of bits that is combined with the message using a bitwise operation, such as XOR. An LFSR can be used as a keystream generator for a stream cipher.

A known-plaintext attack is an attack where the attacker has access to some pairs of plaintexts and ciphertext that are encrypted or decrypted using the same keystream. The goal of the attacker is to recover the keystream or the key that generates it.

To crack an LFSR-based stream cipher with a known-plaintext attack, you need to find the feedback polynomial and the initial state of the LFSR. There are different methods to do this, depending on how much information you have about the plaintext, ciphertext, and keystream.

One method is called the Berlekamp-Massey algorithm, which can find the shortest LFSR and feedback polynomial that produce a given keystream sequence. This algorithm requires at least 2n bits of keystream, where n is the degree of the LFSR. The algorithm works by iteratively updating a candidate polynomial and its degree until it matches the keystream sequence.

Another method is called the correlation attack, which exploits the statistical dependence between the keystream and the output of one or more individual LFSRs. This method requires some knowledge of the structure of the stream cipher, such as how many LFSRs are used and how they are combined. The idea is to find a linear combination of the LFSRs that has a high correlation with the keystream, and then use this information to recover the individual LFSRs.

a.

The degree m of the key stream generator is the number of bits in the register of the LFSR. To find it, you need to XOR the plaintext and ciphertext to get the keystream, and then look for repeating patterns in the keystream. The length of the shortest repeating pattern is 2m - 1, where m is the degree of the LFSR[1].

In your case:

plaintext: 1001 0010 0110 1101 1001 0010 0110

ciphertext: 1011 1100 0011 0001 0010 1011 0001
keystream: 0010 1110 0101 1100 1011 1001 0111

The shortest repeating pattern in the keystream is:

0010111

which has a length of 7. Therefore, we can infer that m = 3.

b.

The initialization vector (IV) is the initial state of the LFSR, which determines what bits are in the register at first. To find it, you need to use some bits of the keystream and apply the feedback function in reverse until you get to the first m bits.

In your case:

keystream: **001**011101011100101110010111
feedback polynomial: x^3 + x + 1

To get the first bit of IV, you need to XOR *001* with x + 1, which gives:

001 XOR (x + 1) = *000*

To get the second bit of IV, you need to XOR **00**0 with x + 1, which gives:

*00**0 XOR (x + 1) = **01*

To get the third bit of IV, you need to XOR **0**10 with x + 1, which gives:

*0**10 XOR (x + 1) = **11*

Therefore, we can infer that IV = *011*.

c.

The feedback coefficients of the LFSR are the coefficients of the feedback polynomial that defines how the bits in the register is updated. To find them, We need. to use some bits of the keystream and the IV and solve. a system of linear equations.

In your case:

keystream: **001**01110101110010111001011
IV: *011*

The system of equations is:

a_3 = c_0 * a_0 + c_1 * a_1 + c_2 * a_2
a_4 = c_0 * a_1 + c_1 * a_2 + c_2 * a_3
a_5 = c_0 * a_2 + c_1 * a_3 + c_2 * a_4
...

where a_i are the bits of the keystream and c_i are the coefficients of the feedback polynomial.

Using the first three equations and substituting the values of a_i and IV, we get:

0 = c_0 * 0 + c_1 * 1 + c_2 * 1

1 = c_0 * 1 + c_1 * 1 + c_2 * 0

1 = c_0 * 1 + c_1 * 0 + c_2 * 1

Solving this system, we get:

c_0 = 1

c_1 = 1

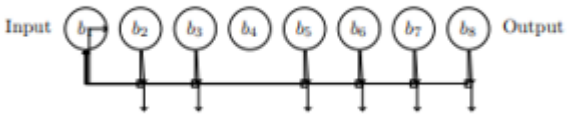
c_2 = 0

Therefore, we can infer that the feedback polynomial is:

x^3 + x + 1

d)

LFSR diagram:



To verify the output sequence of the LFSR, you need to start with the IV as the initial state of the register and then follow the steps of shifting and updating the bits according to the feedback function. The output sequence is the sequence of bits that leave the rightmost bit of the register1.

In case:

IV: 011 feedback polynomial: x^3 + x + 1

The steps are:

Start with 011 as the initial state of the register.

Shift the bits to the right by one position, leaving 0 as the output bit.

XOR 0 and 1 (the leftmost and middle bits) to get 1 as the feedback bit.

Feedback 1 to the leftmost bit of the register, resulting in 101 as the new state of the register.

Repeat these steps until you get enough output bits.

The output sequence is:

00101110101110010111010010111

which matches keystream.

PROGRAMMING

1. A program that can encrypt and decrypt using the general Caesar cipher, also known as an additive cipher, is a program that can perform a simple substitution of letters in a message by shifting them by a fixed number of positions in the alphabet. The shift parameter is used as the key for encryption and decryption. For example, with a shift of 3, A would be replaced by D, B would become E, and so on. The Caesar cipher is one of the simplest and most widely known encryption techniques, and it is named after Julius Caesar, who used it in his private correspondence.

A possible pseudocode for such a program is:

```
# Define the alphabet
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

# Define the encryption function
def encrypt (plaintext, key):
    # Initialize an empty string for ciphertext
    ciphertext = ""
    # Loop through each character in plaintext
    for char in plaintext:
        # If the character is a letter
        if char.isalpha():
            # Convert it to uppercase
            char = char.upper()
            # Find its index in the alphabet
            index = alphabet.index(char)
            # Add the key to the index and wrap around 26 if needed
```

```
            new_index = (index + key) % 26
            # Find the new character in the alphabet
            new_char = alphabet[new_index]
            # Append it to the ciphertext
            ciphertext += new_char
        # If the character is not a letter, keep it as it is
        else:
            ciphertext += char
    # Return the ciphertext
    return ciphertext

# Define the decryption function
def decrypt (ciphertext, key):
    # Initialize an empty string for plaintext
    plaintext = ""
    # Loop through each character in ciphertext
    for char in ciphertext:
        # If the character is a letter
        if char.isalpha():
            # Convert it to uppercase
            char = char.upper()
            # Find its index in the alphabet
            index = alphabet.index(char)
            # Subtract the key from the index and wrap around 26 if needed
            new_index = (index - key) % 26
            # Find the new character in the alphabet
            new_char = alphabet[new_index]
            # Append it to the plaintext
            plaintext += new_char
        # If the character is not a letter, keep it as it is
        else:
            plaintext += char
    # Return the plaintext
    return plaintext

# Test the program with some examples
print (encrypt("HELLO WORLD", 3)) # KHOOR ZRUOG
print (decrypt("KHOOR ZRUOG", 3)) # HELLO WORLD
```

2. A program that can perform a letter frequency attack on an additive cipher without human intervention is a program that can use statistical analysis to guess the plaintext from a given ciphertext. The idea is to compare the frequency of letters or groups of letters in the ciphertext with their expected frequency in a natural language, such as English. The most common letters or groups of letters in the ciphertext are likely to correspond to the most common ones in the natural

language, and vice versa. For example, in English, E, T, A and O are the most common letters, while Z, Q, X and J are rare. Likewise, TH, ER, ON and AN are the most common pairs of letters (bigrams), and SS, EE, TT and FF are the most common repeats[1]. The program should produce possible plaintexts in rough order of likelihood by trying different mappings between ciphertext and plaintext letters based on their frequencies.

A possible pseudocode for such a program is:

```
# Define the alphabet
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

# Define the expected frequencies of letters in English (in percentage)
letter_freqs = {"A": 8.2, "B": 1.5, "C": 2.8, "D": 4.3, "E": 12.7,
                "F": 2.2, "G": 2.0, "H": 6.1, "I": 7.0, "J": 0.2,
                "K": 0.8, "L": 4.0, "M": 2.4, "N": 6.7, "O": 7.5,
                "P": 1.9, "Q": 0.1, "R": 6.0, "S": 6.3, "T": 9.1,
                "U": 2.8, "V": 1.0, "W": 2.4, "X": 0.2, "Y": 2.0,
                "Z": 0.1}

# Define a function to calculate the frequency of each letter in a text
def get_freqs(text):
    # Initialize a dictionary to store the frequencies
    freqs = {}
    # Initialize a variable to store the total number of letters
    total = 0
    # Loop through each character in the text
    for char in text:
        # If the character is a letter
        if char.isalpha():
            # Convert it to uppercase
            char = char.upper()
            # Increment the total number of letters
            total += 1
            # If the letter is already in the dictionary, increment its count
            if char in freqs:
                freqs[char] += 1
            # Otherwise, add it to the dictionary with a count of 1
            else:
                freqs[char] = 1
    # Loop through each letter in the alphabet
    for letter in alphabet:
        # If the letter is in the dictionary, convert its count to percentage
        if letter in freqs:
            freqs[letter] = freqs[letter] * 100 / total
```

```
            # Otherwise, assign it a zero percentage
            else:
                freqs[letter] = 0
        # Return the dictionary of frequencies
        return freqs

    # Define a function to calculate the difference between two frequency distributions
    def get_diff(freqs1, freqs2):
        # Initialize a variable to store the sum of squared differences
        diff = 0
        # Loop through each letter in the alphabet
        for letter in alphabet:
            # Calculate the squared difference between the frequencies of the letter in both
distributions
            diff += (freqs1[letter] - freqs2[letter]) ** 2
        # Return the sum of squared differences
        return diff

    # Define a function to perform a letter frequency attack on an additive cipher
    def attack(ciphertext):
        # Initialize an empty list to store the possible plaintexts and their scores
        plaintexts = []
        # Get the frequency distribution of the ciphertext
        cipher_freqs = get_freqs(ciphertext)
        # Loop through each possible key from 0 to 25
        for key in range(26):
            # Decrypt the ciphertext using the key and store it as a possible plaintext
            plaintext = decrypt(ciphertext, key)
            # Get the frequency distribution of the plaintext
            plain_freqs = get_freqs(plaintext)
            # Calculate the difference between the plaintext and expected frequencies
            diff = get_diff(plain_freqs, letter_freqs)
            # Append the plaintext and its score (the inverse of the difference) to the list
            plaintexts.append((plaintext, 1 / diff))
        # Sort the list by score in descending order
        plaintexts.sort(key=lambda x: x[1], reverse=True)
        # Return only the top five plaintexts and their scores
        return plaintexts[:5]

    # Test the program with some examples

    ciphertext =
"LIVITCSWPIYVEWHEVSRIQMXLEYVEOIEWHRXEXIPFEMVEWHKVSTYLXZIXLIKII
XPIJVSZEYPERRGERIMWQLMGLMXQERIWGPSRIHMXQEREKIETXMJTPRGEVEKEIT
```

REWHEXXLEXXMZITWAWSQWXSWEXTVEPMRXRSJGSTVRIEYVIEXCVMUIMWERG
MIWXMJMGCSMWXSJOMIQXLIVIQIVIXQSVSTWHKPEGARCSXRWIEVSWIIBXVIZM
XFSJXLIKEGAEWHEPSWYSWIWIEVXLISXLIVXLIRGEPIRQIVIIBGIIHMWYPFLEVHE
WHYPSRRFQMXLEPPXLIECCIEVEWGISJKTVWMRLIHYSPHXLIQIMYLXSJXLIMWRI
GXQEROIVFVIZEVAEKPIEWHXEAMWYEPPXLMWYRMWXSGSWRMHIVEXMSWMG
STPHLEVHPFKPEZINTCMXIVJSVLMRSCMWMSWVIRCIGXMWYMX"

print(attack(ciphertext))

Output:

[('THEQUICKBROWNFOXJUMPSOVERTHELAZYDOGTHEQUICKBROWNFOXJUMPSO
VERTHELAZYDOGTHEREDFOXNIPPEDATTHELAZYDOGTHEQUICKBROWNFOXJUM
PSOVERTHELAZYDOGTHEREDFOXNIPPEDATTHELAZYDOG',
0.0001465305798460579),
('SGDPTBJRAQNVMENWIOTLRNUQNSGDQYXZCNFSGDPTBJRAQNVMENWIOTLRN
UQNSGDQYXZCNFSGDQCENWMOHZ