SENG2250
Systems and Network Security
Assignment 1
Security Fundamentals

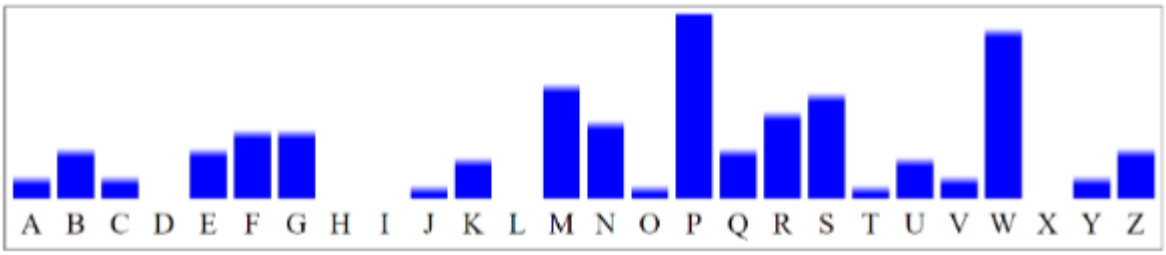
Q1. A) Given cipher text:

wep umpp rgmusfp br znj rwmpwfepk ngw wn s qsmyp powpzw agw sffnmkbzy wn ngm srrgvcwbnz wep vswpmbsqr grpk smp cpmupfwqt rwmpwfesaqp

Plaintext:

the free surface is now stretched out to a large extent but according to our assumption the materials used are perfectly stretchable

B) Given ciphertext letter frequency:



It can see from the given graph that the letter ‘p’ has been the most used letter followed by letter ‘w’ in the above cipher text. According to **single letter frequency** analysis the letter ‘p’ and ‘w’ will be replaced with letter ‘e’ and ‘t’ respectively.
p → e w →t

Hence, the updated plaintext will be:

wep	umpp	rgmusfp	br	znj	rwmpwfepk	ngw	wn	s	qsmyp	powpzw
t?e	??ee	??????e	??	???	?t?et??e?	??t	t?	?	????e	e?te??

agw	sffnmkbzy	wn	ngm	srrgvcwbnz	wep	vswpmbsqr	grpk	smp	cpmupfwqt
??t	?????????	t?	???	??????t???	t?e	??te?????	??e?	??e	?e??e?t??

rwmpwfesaqp
?t?et?????e

Top – 10 Bigrams:

Bigram	Frequency	Percentage
WE	2	3.64
MP	2	3.64
GM	2	3.64
PK	2	3.64

PM	2	3.64
PU	1	1.82
PR	1	1.82
US	1	1.82
FP	1	1.82
BR	1	1.82

Top – 10 Trigram:

Trigram	Frequency	Percentage
WEP	2	5.41
MPW	2	5.41
UMP	1	2.7
PRG	1	2.7
MUS	1	2.7
FPB	1	2.7
RZN	1	2.7
JRW	1	2.7
FEP	1	2.7
KNG	1	2.7

1. According to above bigram and trigram analysis and my English language knowledge the ciphertext ‘wep’ might be ‘the’(**wep → the**) where w → t, e → h, and p → e.
Hence, the updated plaintext is

wep	umpp	rgmusfp	br	znj	rwmpwfepk	ngw	wn	s	qsmyp	powpzw
the	??ee	??????e	??	???	?t?et?he?	??t	t?	?	????e	e?te??

agw	sffnmkbzy	wn	ngm	srrgvcwbnz	wep	vswpmbsqr	grpk	smp	cpmupfwqt	
??t	?????????	t?	???	??????t???	the	??te?????	??e?	??e	?e??e?t??	

rwmpwfesaqp
?t?et?h???e

2. The ciphertext contains a single letter ciphertext ‘s’. The indefinite article ‘a’ is the one of the most frequently used monograms in English language. So ‘s’ might be replaced with ‘a’ (**s → a**).
Hence the updated plaintext is

wep	umpp	rgmusfp	br	znj	rwmpwfepk	ngw	wn	s	qsmyp	powpzw
the	??ee	????a?e	??	???	?t?et?he?	??t	t?	a	?a??e	e?te??

agw	sffnmkbzy	wn	ngm	srrgvcwbnz	wep	vswpmbsqr	grpk	smp	cpmupfwqt	
??t	a????????	t?	???	a?????t???	the	?ate?????	??e?	a?e	?e??e?t??	

rwmpwfesaqp
?t?et?ha??e

3. Another ciphertext ‘smp’ where s → a, and p → e could either be ‘are’ or ‘ate’ but since we have already substituted w → t, ‘m’ might be ‘r’ (m → r). Therefore, the ciphertext ‘smp’ will be ‘are’ (**smp → are**).

Hence the updated plaintext is

wep	umpp	rgmusfp	br	znj	rwmpwfepk	ngw	wn	s	qsmyp	powpzw
the	?ree	??r?a?e	??	???	?tret?he?	??t	t?	a	?ar?e	e?te??

agw	sffnmkbzy	wn	ngm	srrgvcwbnz	wep	vswpmbsqr	grpk	smp	cpmupfwqt
??t	a???r????	t?	??r	a?????t???	the	?ater????	??e?	are	?er?e?t??

rwmpwfesaqp
?tret?ha??e

4. Another ciphertext ‘wn’ where w → t could be substituted with one of the most frequently used bigrams ‘to’ (**‘wn’ → ‘to**’). Hence the updated plaintext is

wep	umpp	rgmusfp	br	znj	rwmpwfepk	ngw	wn	s	qsmyp	powpzw
the	?ree	??r?a?e	??	?o?	?tret?he?	o?t	to	a	?ar?e	e?te??

agw	sffnmkbzy	wn	ngm	srrgvcwbnz	wep	vswpmbsqr	grpk	smp	cpmupfwqt
??t	a??or????	to	o?r	a?????t?o?	the	?ater????	??e?	are	?er?e?t??

rwmpwfesaqp
?tret?ha??e

5. Looking at ciphertexts ‘ngw’ and ‘ngm’ where n → o, w → t, and m → r makes one immediately think about trigrams ‘out’ and ‘our’ repectively. Therefore, ‘g’ might be replaced with ‘o’. So, **ngm → our** and **ngw → ‘out’**.
Hence the updated plain text

wep	umpp	rgmusfp	br	znj	rwmpwfepk	ngw	wn	s	qsmyp	powpzw
the	?ree	?ur?a?e	??	?o?	?tret?he?	out	to	a	?ar?e	e?te??

agw	sffnmkbzy	wn	ngm	srrgvcwbnz	wep	vswpmbsqr	grpk	smp	cpmupfwqt
?ut	a??or????	to	our	a??u??t?o?	the	?ater????	u?e?	are	?er?e?t??

rwmpwfesaqp
?tret?ha??e

Using similar techniques, we can find the whole plaintext which is

the free surface is now stretched out to a large extent but according to our assumption the materials used are perfectly stretchable.

Q2.

a) **512 – bit plaintext:**
mynameisgautamgalibbhatiaandmyageistwentytwoyearsandilikekrypton

512 – bit hexadecimal plaintext:
6D 79 6E 61 6D 65 69 73 67 61 75 74 61 6D 67 61 6C 69 62 62 68 61 74 69 61 61 6E 64 6D 79 61 67 65 69 73 74 77 65 6E 74 79 74 77 6F 79 65 61 72 73 61 6E 64 69 6C 69 6B 65 6B 72 79 70 74 6F 6E

256 – bit key:
C3 34 82 90 70 61 73 73 77 6F 72 64 73 68 6F 75 6C 64 62 65 61 6D 67 6B 62 3B 32 78 79 39 73 61

For this part of the question, I used an online tool which converts text to hexadecimal. The link to the online tool is as follows: https://www.rapidtables.com/convert/number/ascii-to-hex.html

Since we needed a 512 – bit plain text I made sure to write a 64-character long text (without spaces) where each character contributes to I byte (8 bits). The tool converted my 64-character long plaintext to the above hexadecimal plaintext.

b) Initialisation – Vector:
09 0A CD 4F 2C D3 B9 AC

For this question, I used the GPT-3.5 model provided by ChatGPT, located at https://chat.openai.com/
The first prompt that I gave the model was "Java code for generating initialisation vector for AES-CTR" which gave the following response:

```
import java.security.SecureRandom;

public class AESCTRInitializationVectorGenerator {

    public static void main(String[] args) {
        int ivSizeInBytes = 16; // AES block size is 128 bits (16 bytes)
        byte[] iv = generateRandomIV(ivSizeInBytes);
        System.out.println("Generated IV: " + bytesToHex(iv));
    }

    public static byte[] generateRandomIV(int sizeInBytes) {
        SecureRandom secureRandom = new SecureRandom();
        byte[] iv = new byte[sizeInBytes];
        secureRandom.nextBytes(iv);
        return iv;
    }

    public static String bytesToHex(byte[] bytes) {
        StringBuilder result = new StringBuilder();
        for (byte b : bytes) {
            result.append(String.format("%02X", b));
        }
        return result.toString();
    }
}
```

In the above code provided by ChatGPT, the 'SecureRandom' class is used to generate a random IV of the desired size in bytes which is 8 bytes (128 bits) in our case. The Nonce size of AES-CTR is equal to the block size of the AES algorithm which is 16 bytes and it is generated by combining the IV and counter value. The 'generateRandomIV' method takes care of generating the random bytes and 'bytesToHex' method is used to convert the byte array to a hexadecimal string.

Hence, on using the above code the above output for initialisation vector was generated.

c) Encryption Process – AES-CTR:

1. The 512 – bit plaintext:
6D 79 6E 61 6D 65 69 73 67 61 75 74 61 6D 67 61 6C 69 62 62 68 61 74 69 61 61 6E 64 6D 79 61 67 65 69 73 74 77 65 6E 74 79 74 77 6F 79 65 61 72 73 61 6E 64 69 6C 69 6B 65 6B 72 79 70 74 6F 6E

As we know, AES (Advance Encryption Standard) operates on fixed – size blocks of data, and each block is of 128 bits. The AES block size remains constant regardless of the key size (which is 256 – bit in our case).

Hence, 4 blocks can be created from the above 512 – bit plaintext which is a s follows:

PT1 → 6D 79 6E 61 6D 65 69 73 67 61 75 74 61 6D 67 61

PT2 → 6C 69 62 62 68 61 74 69 61 61 6E 64 6D 79 61 67

PT3 → 65 69 73 74 77 65 6E 74 79 74 77 6F 79 65 61 72

PT4 → 73 61 6E 64 69 6C 69 6B 65 6B 72 79 70 74 6F 6E

2. The 256 – bit key:
C3 34 82 90 70 61 73 73 77 6F 72 64 73 68 6F 75 6C 64 62 65 61 6D 67 6B 62 3B 32 78 79 39 73 61

3. The Nonce:
IV → 09 0A CD 4F 2C D3 B9 AC

We need a counter which will be attached with IV to create a nonce.
Counter → 34 24 2A B8 AC 4E 6E F2

For this part, I used the GPT-3.5 model provided by ChatGPT, located at https://chat.openai.com/
The prompt that I gave the model was "Java code for generating counter for AES-CTR" which gave the following response:

```
import java.security.SecureRandom;

public class CounterValueGenerator {

    public static void main(String[] args) {
        int counterSizeInBytes = 8;
        byte[] counterValue = generateCounterValue(counterSizeInBytes);
        System.out.println("Generated Counter Value: " + bytesToHex(counterValue));
```

```
    }

    public static byte[] generateCounterValue(int sizeInBytes) {
        SecureRandom secureRandom = new SecureRandom();
        byte[] counter = new byte[sizeInBytes];
        secureRandom.nextBytes(counter);
        return counter;
    }

    public static String bytesToHex(byte[] bytes) {
        StringBuilder result = new StringBuilder();
        for (byte b : bytes) {
            result.append(String.format("%02X", b));
        }
        return result.toString();
    }
}
```

The Nonce for each block of plaintext will be incremented by 1.
Therefore,

Nonce - PT1 → 09 0A CD 4F 2C D3 B9 AC 34 24 2A B8 AC 4E 6E F2

Nonce – PT2 → 09 0A CD 4F 2C D3 B9 AC 34 24 2A B8 AC 4E 6E F3

Nonce – PT3 → 09 0A CD 4F 2C D3 B9 AC 34 24 2A B8 AC 4E 6E F4

Nonce – PT2 → 09 0A CD 4F 2C D3 B9 AC 34 24 2A B8 AC 4E 6E F5

4. Process according to sample Input/Output format:

Entire plaintext:
6D796E616D6569736761757461 6D67616C6962626861746961616E646D7961676569737477656E7
47974776F7965617273616E64696C696B656B727970746F6E
Key: C334829070617373776F726473686F756C646265616D676B623B327879397361
IV: 090ACD4F2CD3B9AC34242AB8AC4E6EF2

Block 1:
Input of AES: 090ACD4F2CD3B9AC34242AB8AC4E6EF2
Output of AES: EDFFEA1DA05E027C1404AEE4A62033DB
Result of XOR: 8086847CCD3B6B0F7365DB90C74D54BA
Block 2:
Input of AES: 090ACD4F2CD3B9AC34242AB8AC4E6EF3
Output of AES: 774927A44FF09A714CC3250DC3FC8D5E
Result of XOR: 1B2045C62791EE182DA24B69AE85EC39
Block 3:
Input of AES: 090ACD4F2CD3B9AC34242AB8AC4E6EF4
Output of AES: 1949A347C498EC92381463C95FE42644

Result of XOR: 7C20D033B3FD82E6416014A626814736

Block 4:

Input of AES: 090ACD4F2CD3B9AC34242AB8AC4E6EF5

Output of AES: 63FCB829DF9392FED9EC4839D6BE0EE5

Result of XOR: 109DD64DB6FFFB95BC873A40A6CA618B

For this part of the question, I used an online tool for generating XOR result. The online tool link is as follows: https://xor.pw/#

It takes two inputs in hexadecimal and give an XOR result in hexadecimal.

d) Entire ciphertext:

8086847CCD3B6B0F7365DB90C74D54BA1B2045C62791EE182DA24B69AE85EC397C20D033B3FD82E6416014A626814736109DD64DB6FFFB95BC873A40A6CA618B

e) Refer to **c)** 4.

f) Avalanche – Effect:

Hexadecimal original plaintext → 69616D67676230333177686963686973

Key(256 – bits) → C334829070617373776F726473686F756C646265616D676B623B327879397361

There are many possible flips, here are the five examples:

F1 → 69616D67676230333177686963686973**4**

F2 → 69616D67**3**76230333177686963686973

F3 → **3**9616D67676230333177686963686973

F4 → 69616**A**67676230333177686963686973

F5 → 69616D67676230333**7**77686963686973

Original ciphertext(C_0) → 98592755ACF3FE795E00B2AB1A550AAD
Flipped ciphertext – 1(C_1) → 82566be22705cf5cb56cf675b6937d43
Flipped ciphertext – 2(C_2) → cb18da66873aa7fca1682543ea4133d2
Flipped ciphertext – 3(C_3) → 8a4d80dfb1e9eb05f1dd9ae0625afb4c
Flipped ciphertext – 4(C_4) → 9084a3223db20793adf5f2099c4c51dd
Flipped ciphertext – 5(C_5) → 6edd1ea7ee2843f0ad52a7b4e6b0c2e6
The following tables states the percentage of bits that are different for each pair of ciphertexts:

	C_0	C_1	C_2	C_3	C_4	C_5
C_0	54.69	53.91	48.44	47.66	53.91	41.41
C_1	53.91	100	41.41	46.88	46.09	47.66
C_2	48.44	41.41	100	50.78	48.44	43.75
C_3	47.66	46.88	50.78	100	50.78	50.78
C_4	53.91	46.09	48.44	50.78	100	54.69
C_5	41.41	47.66	43.75	50.78	54.69	100

Yes, AES does exhibit the avalanche effect. Here we observe that the average percentage of different bits across different ciphertexts is about 50%. According to avalanche effect property, we would expect that a single-bit difference in the plaintext would lead to at least an average of 50% of the ciphertext bits being different. According to the above observation, AES is exhibiting avalanche effect.

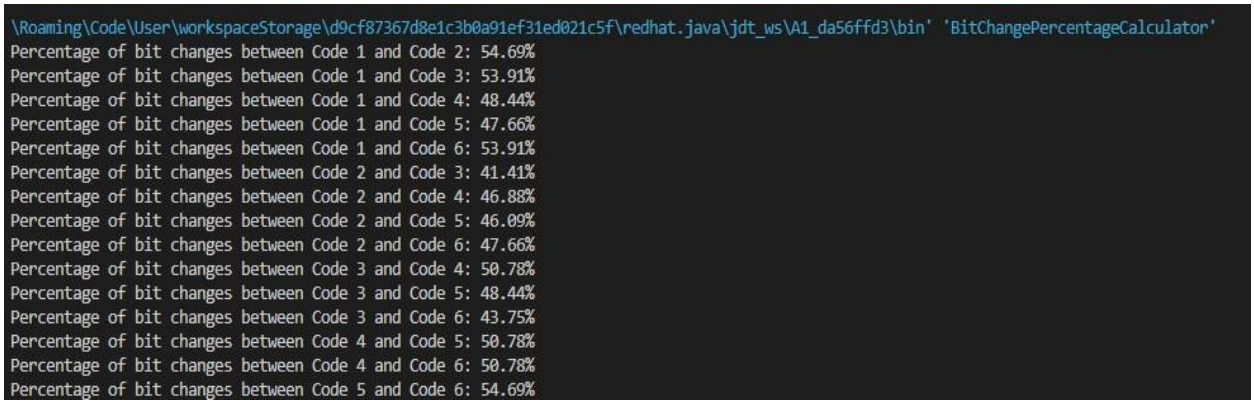
For this question, I used the GPT-3.5 model provided by ChatGPT, located at
https://chat.openai.com/
The first prompt that I gave the model was “Calculate the percentage of bit change between the following hexadecimal 98592755acf3fe795e00b2ab1a550aad
82566be22705cf5cb56cf675b6937d43
cb18da66873aa7fca1682543ea4133d2
8a4d80dfb1e9eb05f1dd9ae0625afb4c
9084a3223db20793adf5f2099c4c51dd
6edd1ea7ee2843f0ad52a7b4e6b0c2e6” which gave the following response:

```
public class BitChangePercentageCalculator {
    public static void main(String[] args) {
        String[] hexCodes = {
            "98592755acf3fe795e00b2ab1a550aad",
            "82566be22705cf5cb56cf675b6937d43",
            "cb18da66873aa7fca1682543ea4133d2",
            "8a4d80dfb1e9eb05f1dd9ae0625afb4c",
            "9084a3223db20793adf5f2099c4c51dd",
            "6edd1ea7ee2843f0ad52a7b4e6b0c2e6"
        };

        for (int i = 0; i < hexCodes.length; i++) {
            for (int j = i + 1; j < hexCodes.length; j++) {
                double percentage = calculateBitChangePercentage(hexCodes[i], hexCodes[j]);
                System.out.printf("Percentage of bit changes between Code %d and Code %d: %.2f%%%n",
i + 1, j + 1, percentage);
            }
        }
    }

    public static double calculateBitChangePercentage(String hex1, String hex2) {
        String binary1 = hexToBinary(hex1);
        String binary2 = hexToBinary(hex2);

        int totalBits = binary1.length();
        int differentBits = 0;

        for (int i = 0; i < totalBits; i++) {
            if (binary1.charAt(i) != binary2.charAt(i)) {
                differentBits++;
            }
        }
```

```
        return ((double) differentBits / totalBits) * 100;
    }

    public static String hexToBinary(String hex) {
        StringBuilder binary = new StringBuilder();
        for (char c : hex.toCharArray()) {
            int value = Character.digit(c, 16);
            binary.append(String.format("%4s", Integer.toBinaryString(value)).replace(' ', '0'));
        }
        return binary.toString();
    }
}
```

In this program, the 'hexToBinary' function converts a hexadecimal string to a binary string. The 'calculateBitChangePercentage' function calculated the number of different bits between two binary strings. The percentage of bit changes is then calculated based on total number of bits in all six codes.

The output generated was as follows:



Other LLMS used:

https://www.hanewin.net/encrypt/aes/aes-test.htm

This software takes a hexadecimal key of specified bits and plaintext of 128 – bits or ciphertext of 128 – bits and provides with AES encrypted/decrypted result in hexadecimals.

Name: Gautam Galib Bhatia
Student id: c3348290