# Exercise 3

## Chosen-plaintext attack(CPA)

Enigma was broken using both Chosen-plaintext attacks and known-plaintext attacks. We will describe the usage of Chosen-plaintext attacks.

## Known-plaintext attack(KPA)

The A5/1 algorithm was broken using known-plaintext attacks.

# Exercise 5

```
public class Hackz {
static String Key = null;

public static void main(String[] args) {

long startTime1 = System.currentTimeMillis();
int cntr = 0;
String key = "AAAAAAAAAAAADDAd", plaintext = "Secretfoemotherd";
StringBuilder cipher = new StringBuilder(), brutus = new StringBuilder();

byte[] ciphertext = encrypt(key.getBytes(), plaintext.getBytes());
for (int i = 0; i < ciphertext.length; i++) {
cipher.append(ciphertext[i]);
}

char[] nkey = new char[16];
for (int i = 0; i < 16; ++i) {
nkey[i] = 65;
}

while (true) {

cntr++;
byte[] brutusCipher = encrypt(byteC(nkey), plaintext.getBytes());
for (int k = 0; k < brutusCipher.length; k++) {
brutus.append(brutusCipher[k]);
}

if (brutus.toString().equals(cipher.toString())) {
System.out.println("Key: " + Key);
System.out.println("i ran: " + cntr + " times");
```

```
long endTime1 = System.currentTimeMillis();
System.out.println("Took " + (endTime1 - startTime1) + " ms");
return;
}
brutus.setLength(0);
int index = 15;
nkey[index]++;
while (index >= 0 && nkey[index] >= 122) {
nkey[index] = 65;
index--;
if (index < 0) {
break;
}
nkey[index]++;
}
}

}

public static byte[] byteC(char[] s) {
StringBuilder temp = new StringBuilder();
for (int i = 0; i < s.length; i++) {
temp.append(s[i]);
}
Key = temp.toString();
return temp.toString().getBytes();
}

public static byte[] encrypt(byte[] key, byte[] plaintext) {
byte[] d = new byte[key.length];
for (int i = 0; i < key.length; i++) {
d[i] = (byte) (key[i] ^ plaintext[i]);
}
```

```
return d;
}

}
```

# Exercise 6

The unicity distance is calculated based on a simple substitution cipher, using english letters only. We see captial and lowercase as the same. The unicity distance is calculated by using following equation:

$$|UD| = \frac{log_2(|K|)}{R_L \cdot log_2(|P|)} \tag{1}$$

We use the following:
$|K| = 26!$
$|P| = 26$
$R_L = 0.7$

$$UD = \frac{log_2(26!)}{0.7 \cdot log_2(26)} = \frac{88.39}{0.7 \cdot 4.7} = 26.8 \tag{2}$$

So the average number of ciphertext characters required to eliminate alle spurious keys is around 27.

# Exercise 9

The $\chi^2$ test is performed on the given data. The four pairs of bit should occour an equal ammount of times, if the random number generator is perfect. But as it's a quite small sample size, there will be fluctations. The following data is given:

$$k_1(''11'') = 228 \quad e_1(''11'') = 250 \quad k_2(''10'') = 270 \quad e_2(''10'') = 250$$
$$k_3(''01'') = 271 \quad e_3(''01'') = 250 \quad k_4(''00'') = 231 \quad e_4(''00'') = 250$$

The following equation is used:

$$\chi^2 = \frac{(o_1 - e_1)^2}{e_1} + \frac{(o_2 - e_2)^2}{e_2} + \dots \frac{(o_k - e_k)^2}{e_k} \tag{3}$$

$$\chi^2 = \frac{(228 - 250)^2}{250} + \frac{(270 - 250)^2}{250} + \frac{(271 - 250)^2}{250} + \frac{(231 - 250)^2}{250} = 6.744$$

Since the degree of freedom is 3 in our example, the probability value is:

$$P = \frac{1}{\Gamma\left(\frac{3}{2}\right)} \cdot \Gamma\left(\frac{3}{2}, \frac{6.744}{2}\right) = 0.08 = 8\% \tag{4}$$

A P-value of 0.05 or less is regarded as statistically significant. Therefore our example is regarded as non significant.

# Exercise 11

I've made a encryption tool:

```
void encrypt(int *plainText, int initVector){

    cipher[0] = key[initVector ^ plainText[0]];

    for (int i = 1; i < bitSize; i++) {
        cipher[i] = key[cipher[i - 1] ^ plainText[i]];
    }
}

int main(int argc, const char * argv[])
{

    for (int i = 0; i < bitSize; i++) {
        initVector = i;

        encrypt(plaintext, initVector);
        printf("InitVector: %d Cipher: ", initVector);
        for (int j = 0; j < bitSize; j++) {
            printf("%d ", cipher[j]);
        }
        printf("\n");
    }

}
```

The tool encrypts the plaintext, using every 4-bit initialisation vector. The following console output is given:

```
InitVector: 0 Cipher: 12 7 1 3 13 5 8 15 11 14 2 4 10 6 0 12
InitVector: 1 Cipher: 3 13 5 8 15 11 14 2 4 10 6 0 12 7 1 3
InitVector: 2 Cipher: 4 10 6 0 12 7 1 3 13 5 8 15 11 14 2 4
```

```
InitVector: 3 Cipher: 13 5 8 15 11 14 2 4 10 6 0 12 7 1 3 13
InitVector: 4 Cipher: 10 6 0 12 7 1 3 13 5 8 15  11 14 2 4 10
InitVector: 5 Cipher: 8 15 11 14 2 4 10 6 0 12 7 1 3 13 5 8
InitVector: 6 Cipher: 0 12 7 1 3 13 5 8 15 11 14 2 4 10 6 0
InitVector: 7 Cipher: 1 3 13 5 8 15 11 14 2 4 10 6 0 12 7 1
InitVector: 8 Cipher: 15 11 14 2 4 10 6 0 12 7 1 3 13 5 8 15
InitVector: 9 Cipher: 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9
InitVector: 10 Cipher: 6 0 12 7 1 3 13 5 8 15 11 14 2 4 10 6
InitVector: 11 Cipher: 14 2 4 10 6 0 12 7 1 3 13 5 8 15 11 14
InitVector: 12 Cipher: 7 1 3 13 5 8 15 11 14 2 4 10 6 0 12 7
InitVector: 13 Cipher: 5 8 15 11 14 2 4 10 6 0 12 7 1 3 13 5
InitVector: 14 Cipher: 2 4 10 6 0 12 7 1 3 13 5 8 15 11 14 2
InitVector: 15 Cipher: 11 14 2 4 10 6 0 12 7 1 3 13 5 8 15 11
```

We see that every unique initialisation vector, produces a different output of the cipher. We also see that, when the initialisation vector is 9 every number in the cipher text is also 9. This is expected, as 3 XOR 9 = 10 and a p = 10 produces a c = 9 creating a loop.