Exercise Report 01

Karamov, Nikita
Enrolment Nr.: 5035512
Informatik, BPO 2017

27.11.2019

Symmetric Cryptography Fundamentals

1 Security goals

Alice returns home and finds the door to her apartment open. Mallory broke in while she was away.

a) **Confidentiality** could have been violated: Alice could've been hiding her key under the floor mat, which isn't a good hiding spot. **Integrity** could have been violated: Mallory could've tampered with the door so that it isn't able to be closed all the way. **Availability** could have been violated: Alice's door could've been too flimsy to stop Mallory from just breaking it.

b) **Prevention** could be deployed: Alice could install a secondary door or a better lock. **Detection** could be deployed: Alice could install an alarm system or a smart CCTV, that would notify her and/or the police in case of breaking in. **Analysis** could be deployed: Alice could call the police or hire a detective to find the perpetrator and return the stolen stuff (if things were stolen)

2 Simple combinatorics

a) If only the English alphabet (26 letters) is considered, the size of the key space for ROT13 is 1, as there is only one key. The key space could be bigger for bigger alphabets (if the shift remains at 13 characters), but then the cipher would not be reversible.

b) For an alphabet with c characters and a key length of n the size of the key space for Vigenère would be c^n.

c) For 256-bit AES the size of the key space is $2^{256} \approx 1.158 \times 10^{77}$. It is basically the same formula as in **b)**, except that we know the size of the alphabet (2 characters – either zero or one) and the length of the key.

d) For an alphabet (and thus the key size) of k letters the size of the key space is the amount of k-permutations – $k!$.

3 XOR cipher

The messages were encrypted with the same key, which leads to the possibility of doing the following operation:

$$C_1 \oplus C_2 = (M_1 \oplus K) \oplus (M_2 \oplus K) = M_1 \oplus M_2$$

The result of this can then be XOR'ed with M_2 to get the value of M_1:

$$(M_1 \oplus M_2) \oplus M_2 = M_1$$

This way Eve can recover the first message as well.

Eve then can calculate the possible keys by applying XOR to M_1 and either of the encrypted messages. Since

$$C_i = M_i \oplus K$$

the result of the aforementioned operation would be

$$M_1 \oplus C_i = M_1 \oplus M_i \oplus K$$

If M_i happens to be M_1, the summands will cancel each other out, and Eve will get the correct key, which she'll be able to check using the deciphered messages. If the key is incorrect, Even will have to do the operation again with M_2.

Mono- and Polyalphabetic Ciphers

4 Monoalphabetic (en/de)cryptor

```
mono/mono.py (--encrypt KEY | --decrypt KEY) [--out FILE] FILE
```

Usage

The positional argument `FILE` accepts the file, containing the message to be encrypted. Apart from this parameter, an argument with one of two flags `encrypt` or `decrypt` needs to be supplied, holding the substitution key, that will be used to either encrypt or decrypt the message respectively.

The optional `out` argument can be used to specify the output file. If this parameter is omitted, the result of the encryption/decryption process will be output to Python's `stdout`.

Way it works

Before processing the input message the program strips any non-alphabetic symbols from the input message and makes all characters lowercase. This gets rid of spaces, numbers and other symbols, which cannot be restored when decrypting the message. However, the builtin function `isalpha()` traits other alphabets (e.g. Cyrillic) as alphabetic symbols as well, which is important to know.

For the encryption process the program first defines a plaintext alphabet; in this case it's the English alphabet string (`abcde...`). Then in a loop, each character of the message is being replaced with the character from the substitution key, that has the same index, as the input character has in the plaintext alphabet.

Example. For alphabet `"abcde"` and key `"edbac"`, all the letters `a` in the message will be replaced with the letter `e`.

The cipher is symmetric, meaning that decrypting an encrypted message is basically encrypting the message again, except the alphabet and the key are swapped. Basically, the ciphered message needs to be ciphered again, but with an inverted alphabet-key pair, in order to get decrypted.

Example. Continuing the example above, in order to decrypt a message all letters `e` need to be replaced with the letter `a`.

Since the decryption and encryption processes are similar, the program utilises one common method instead of two separate.

Sample Run

Message (`in.txt`) `"Hello world!"`

Key `qwertyuiopasdfghjklzxcvbnm`

Encryption command

```
mono.py --encrypt qwertyuiopasdfghjklzxcvbnm --out out.txt in.txt
```

Ciphered message (`out.txt`) `"itssgvgksr"`

Decryption command

```
mono.py --decrypt qwertyuiopasdfghjklzxcvbnm out.txt
```

Deciphered message `"helloworld"`

5 Monoalphabetic code breaker

`mono/break_mono.py MESSAGE`

Usage

The only positional argument `MESSAGE` is the encrypted message.

Way it works

The program tries to analyse the text using the most popular English words as well as the frequencies of letters.

At first the program analyses the probable presence of popular English words, such as "the", "and" and "have". For each word, the program shingles the whole message with the shingle size equal to the word length, finds the most popular shingle and matches it with the word, recording each of the letter in a map.

Then, the program analyses the rest of the letters. The letters of the message are sorted by how often they appear and are being matched with the statistical data. The found letter pairs are also recorded to the map, omitting duplicate records.

The map, the keys of which represent the letters of the alphabet and the values of which represent the substitute, is then sorted by keys. The values are being transformed into a string, which is the final key.

I have been experimenting with different words, but I coulnd't develop an algorithm good enough to decipher the example text. The result is close enough; if the spaces are preserved, one can understand the words:

habe yiu veen ti the desert
habe yiu walged woth the dead...

Sample Run

Ciphered message (`out.txt`) `"gryticdettpjcjgtmtntaj..."`

Breaking command

```
break_mono.py gryticdettpjcjgtmtntaj...
```

Hypothetical key `"ryhmtzsgcxfqlpowuanjdekbiv"`

Decryption command

```
mono.py --decrypt ryhmtzsgcxfqlpowuanjdekbiv out.txt
```

Deciphered message `"habeyiuveentithedesert..."`

6 Vigenère (en/de)cryptor

```
vigenere/vig.py (--encrypt KEY | --decrypt KEY) [--out FILE] FILE
```

Usage

The usage is identical to that of the monoalphabetic substitution encryptor (*task 4*).

Way it works

The message is being transformed identically to how it is done in *task 4*.

The first step for both encrypting and decrypting is to "multiply" the key by repeating it until the length of the repeated string reaches the length of the message.

Example. For the message `"helloworld"` the key `"cat"` has to become `"catcatcatc"`.

Then in a loop, the alphabet index of each message character is added to the alphabet index of the corresponding key character. If the resulting sum is larger than the alphabet length, the sum is reduced by the alphabet length. The letter of the alphabet, the index of which is equal to the resulting number is the ciphered letter.

Example. Continuing the example above, in the English alphabet the letter `w` has the index 22 and the letter`t` has the index 19. So, the sum will be 41. Since its bigger than the length of the English alphabet, it is being reduced by 26 down to 15. The letter with the index 15 is the letter `p`.

The decryption process is similar to the encryption process. The differences are that the index of the key character is being subtracted from the index of the message character, rather than being added, and that instead of going over the length of the alphabet the resulting index may go lower than 0.

Since the encryption and decryption processes are similar, the program uses one common method for both, just like in `mono.py`.

Sample Run

Message (`in.txt`) `"Hello world!"`

Key `iamcool`

Encryption command

```
vig.py --encrypt iamcool --out out.txt in.txt
```

Ciphered message (`out.txt`) `"pexnckzzlp"`

Decryption command

```
vig.py --decrypt iamcool out.txt
```

Deciphered message `"helloworld"`