Practical Assignment 2

Name	Studienummer
Jacob Kjærager	201611661
Morten Sahlertz	201410062

*libraries used: wordfreq, pycryptodome, pandas, numpy,

Part 1: Cryptanalysis tool on OTP (One-Time Pad) cipher used twice

This exercise seeks to decode 2 files and subtract the key that is used to decode them.

For the task, 2 files are giving which has been encoded with the same key.
To solve the task, we'll use the following approach for decoding:

1) Xor the files, which gives the "mapping" value between File1 and File2.
2) Make an ASCII lookup table which equal to the values from the previous step.
3) Extract top 1000 words as vector using the python library "wordfreq"
4) Guess on words and validate this against the top 10000 vector. Between the two files.
5) Wait till the program finishes.

The attached program is fully automatic, but will fail when it reaches words which are not present in the top 10000 vector. This could be solved by making a larger vector with combinations like "the, " or "the. ". But the execution time for this is estimated to be too long.

When the program has broken, which can be seen at Figure(1). Words will not be extended

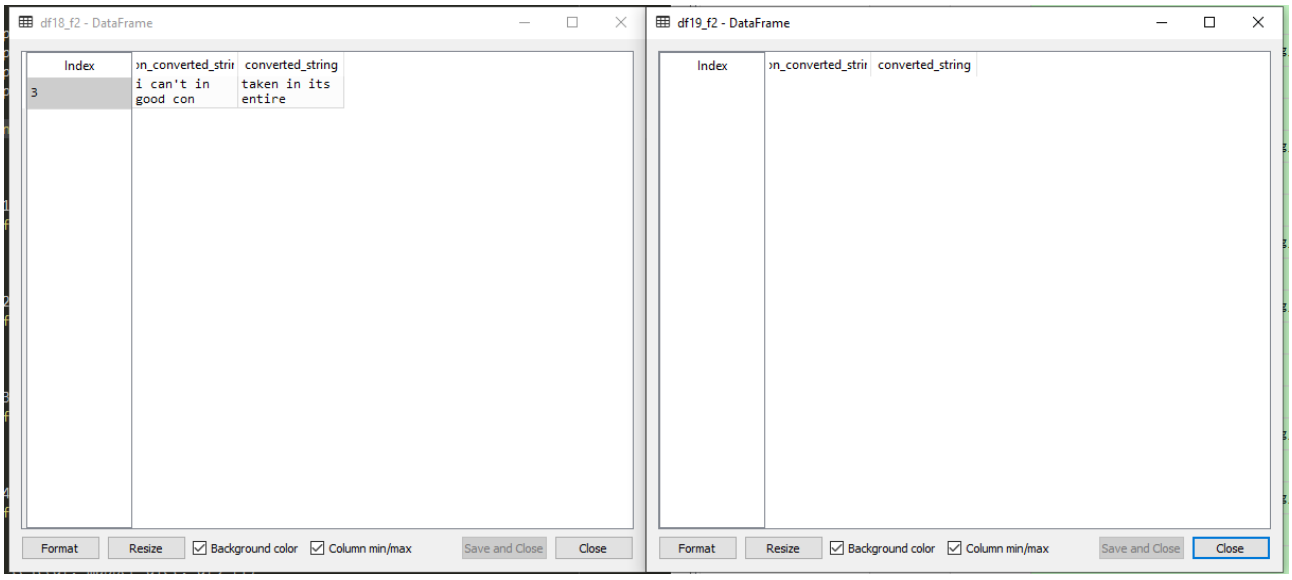


Figure 1: failed attempt using fully automatical software.

This requires manual intervention. When the word which is missing has been found, this can be added to the top 10000 vector.

This will make the program know the word for next time it has runned.

Issues

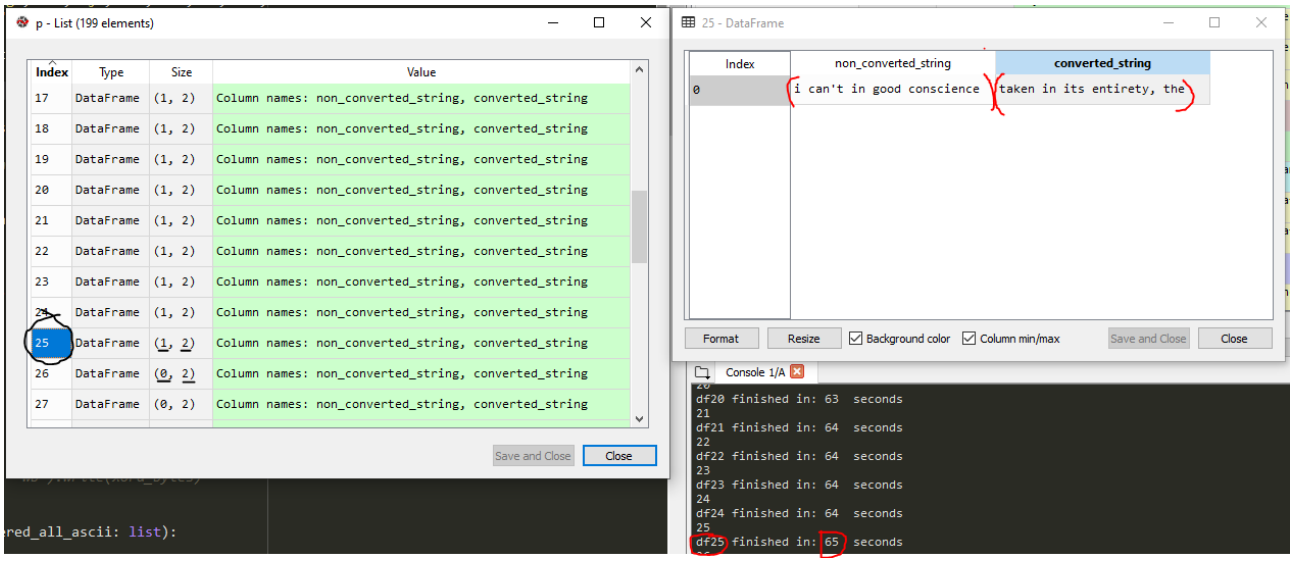
At the time of project delivery, the program fails at byte 25 out of 200. This assumably caused by the program's logic uses sample K-1 for sample K, and at byte 25 the mapping on both sides' maps to a space

(“ “). This has not been solved an if the program is executed it will stop at byte 25.

Conclusion on failing software

The program runs well but misses just the last bit. This could be done using more than 1 sample for validation.

The program correctly returns the mapping as follows:



Here we see it fails after 65 seconds at byte 25. It correctly decodes the first 25 characters between the files but can’t after here. (not because Snowden is after, this has been tried to manually add to the top 10000 vector)

File used for text decoding: **xor_files.py**

Workarounds

After the 25-byte problem was not fixable, a more manual approach was taken.
This was by manually looking at the mappings and choose the top words when letters were given.

This gave the solution to the task as:

Key finding

The key is obtained by Xoring the found plaintext with the ciphertext. The result of this must be the key. This key is tested on both ciphertexts for obtaining the plaintext. This is given below

```
morten@morten-VirtualBox:~/Desktop/Network Security/Assignment2$ python3 xor2.py
 result.txt challenge1.txt result3.txt
i can't in good conscience allow the U.S. government to destroy privacy, interne
t freedom and basic liberties for people around the world with this massive surv
eillance machine they're secretly buildM
morten@morten-VirtualBox:~/Desktop/Network Security/Assignment2$ python3 xor2.py
 result.txt challenge2.txt result3.txt
taken in its entirety, the Snowden archive led to an ultimately simple conclusio
n: the U.S. government had built a system that has as its goal the complete elim
ination of electronic privacy worldwide
```

File used for key obtaining: **xor2.py**

Final results

The files were decoded to:

File1:

I can't in good conscience allow the U.S. government to destroy privacy, internet freedom and basic liberties for people around the world with this massive surveillance machine they're secretly build

File2:

Taken in its entirety, the Snowden archive led to an ultimately simple conclusion: the U.S. government had built a system that has as its goal the complete elimination of electronic privacy worldwide

Part 2: Rabin cryptosystem

Usage:

The implementation of the Rabin cryptosystem is implemented in Python3. To run the program have the rabin.txt and rabin.py files in the same folder and simply run the python program. It is necessary to download the Cryptodome library for python to run the program.

Approach:

An implementation using the Rabin Miller method for finding a big prime has been implemented, this was done through the Cryptodome library. Then the message was changed from ASCII to an integer value and then encrypted. After the decryption the whole message is changed back to ASCII. The encryption is done through the use of the Rabin cryptography, with the help of the extended Euclidean algorithm and the Chinese remainder theorem. The encryption is done by the two generated prime numbers p and q, summing these together equals the public key n. The decryption uses the extended Euclidean algorithm and the Chinese remainder theorem, which allows the system to find four square roots of c modulo n, where c is the cipher text. The implementations for these can also be found in the functions in rabin.py. Since the plain text is $\sqrt{c}$ mod n one of the square roots is the plain text message. By adding some padding to the message it is possible to introduce some redundancy, that makes it possible to find which root is the original text message.

Example:

Below is an example of the program running in Linux terminal:

```
Message: i can't in good conscience allow the U.S. government to destroy privacy, internet fr
eedom and basic liberties for people around the world with this massive surveillance machine
they're secretly building. i can't in good conscience allow the U.S. government to destroy pr
ivacy, internet freedom and basic liberties for people around the world with this massive sur
veillance machine they're secretly building.

Private keys (1536 bits generated prime numbers):
p: 150448526252160340342235870715282255628725799818426427924480145833093595204078151798308793
906082242923005318794809667293457361463453617244146332968274778643466225427396956409168251445
971820830094558482602659193972218738131054357826984224542222399288520683556063081448157598579
544005048523844734364295797340310340486019720109570460154847509304832486389451320914601005236
137985649832903449728270142689489671999841138425170342069298455957262372121451409095686484913
1

q: 223665786082784272876836180935497273083479253247082138914291794239205450050862302015099585
083988479051099088649686834542575384773016548147431163008479323455316940022994602583654479317
543424820931348240535495043254274727649278506127264037924412323723295437488151246717946049151
232066554922092592650067620276040817397082135481955824878667030677941791447338069138088984521
119636482075160021406513627643161989259912180626122744864084123384959486312783857561109557357
1
```

This shows the original message and the generated prime numbers used as private keys.

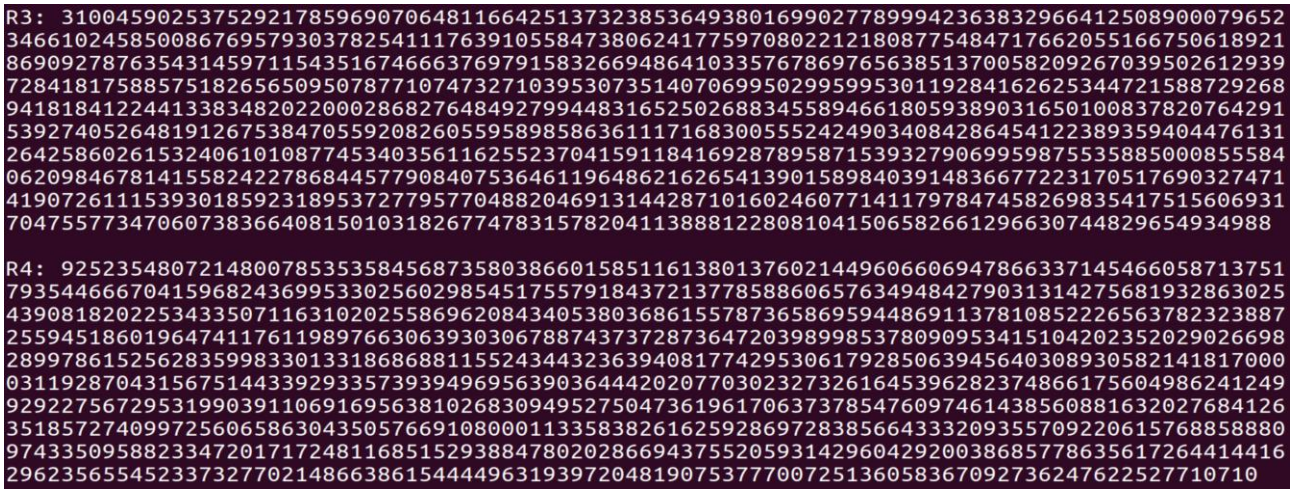
```
Public key:
n: 310045902537529217859690706481166425137323853649380169902778999423638329664125089000796523
466102458500867695793037825411176391055847380624177597080221218087754847176620551667506189218
690927876354314597115435167466637697915832669486410335767869765638513700582092670395026129397
284181758857518265661453997620458331015556284025073909814107105304402752737264957126397403699
328287332152149159314103240293174952488449439957105898656701276292004901226612019981419254018
602750637993524086496808803294016107708296273221571511066254100452253874652827103805454871422
653696929362951620111984343747122135426914969628227040201197664044293909809866762061218666951
831794781351269653390395467794052364056222860532737952200162195949659977175443315587414777925
100437623049813669335106884490988858930979942252582025934710925027688384993209364278367679727
159661646116177048150924921728379358643681413925598633618114661668570730664138513066643598 9

Encrypted message (without padding):
134192512302143224282637225557278616016010755316009510908156592996979323560551727063344264421
795662025819249090980248247663318294190415381339201824050989555234882000251586400068311013494
680314831654025391709025199964777179561530009438322372883220702686240295579375158305533318443
481566715921353225774649396351732503362149173387761126552816660818148107927055515465512200230
331856461105712363741629922956052787927411240310424185671218011497215469557300088854307457050
388902778235807047252355818517111161624917378755187006713063730307758349144721082638541132715
556104015924100295828644041260476223972359693191826960893044761807293457809356938790540152481
247920669873998008091055418283224714558957017668607036505720339729152022808812267263120856687
742419302353481315012500027882615734973889277462513287136707930594456980260017055961119488791
536790270686051937250849823051821712563380936709633002527850196216107260431186825790053 6
```

This shows the generated public key n, and the encrypted message.

```
R1: 105032099097110039116032105110032103111111100032099111110115099105101110099010320971081 0
811111903211610410103208504608304603210311111810111411010910111011603211611103210010111511611
411112103211211410511809709912104403210511011610111411010111603210211410110110011110903209711
010003209809711510509903210810509810111411610510111503210211111403211210111111210810103209711
411111711010003211610410103211911111410810003211910511610403211610410511503210909711511510511
810103211511711411810110510810809711009910103210909709910410511010103211610410112103911410103
211510109911410111610812103209811710510810010511010304601001064030101150100 1

R2: 217522354465381210006154860793808386477165342035578793881329392816690463326979622942082 77
167255779179670801335604249515514653653828946218696381849164641738270568145306275985573326 19
325184605612897124640380414721076807707242728911779475703128280618982256277124044383124380551
002823657283787085390025502098981902794766884665220026261011711992631179932216075477436837700
103830871689586555948108992160629379724500620356302412436139509700694095558630308939927743701
857155776756195672062879467554621138144392628801363808042980838806857592277960928200468630 17
272446936206763171620091517418331186711741969458086507849482387856819644837130588042919098282
547993750725154404680409111721736128405508927670657535933043381028532676781973409497164591904
412610252716747894916338207280583592008317791358314447387539662898476834630743072866110326531
086342599066383972044877625786683490901174169344369109584804214806273402139051375081387252 79
```



The above screenshots show the four found square roots (R1, R2, R3, R4).

Then the found message is converted, and the whole process is tried again, where a new set of primes p and q is found. The result of this indicates that the message cannot be found without the original p and q generated for the encryption.

Found problems:

A few found problems:

1. If the text starts with a specific character the found string is messed up, this is because if the number in the integer is a 0, the revert string function cannot find the proper ASCII characters.
2. If the text is very short, it is possible to replicate the text found with another generated prime number keys. The reason for this is unknown by the group, and if a reason is apparent an explanation would be appreciated.