Lab A–Breaking traditional substitution cipher & Exploring the MD5 collisions

1. Generating monoalphabetic substitution ciphertext with linux tr

This paragraph describes the guidelines how to generate ciphertext for an ascii plaintext document (plain.txt). The original text is first converted to lower cases, and then removed all the punctuations and numbers. I do keep the spaces between words, so you can still see the boundaries of the words in the ciphertext. In real encryption using monoalphabetic cipher, spaces will be removed. I keep the spaces to simplify the task:

$tr [:upper:] [:lower:] < plain.txt > lcase.txt

Remove delimiters and numbers from source files while preserving spaces.

$tr –cd '[a-z][\n][:space:]' < lcase.txt > nodelim.txt

Generate the key (i.e. the substitution table)

$python

>>>import random

>>>skey = 'abcdefghijklmnopqrstuvwxyz'

>>>list = random.sample(skey,len(skey))

>>>''.join(list)

'ibejrzpwqtolmsnycxkfhuvdqa'

Encrypt nodelim.txt with tr:

$tr 'abcdefghijklmnopqrstuvwxyz' 'ibejrzpwqtolmsnycxkfhuvdqa' < nodelim.txt > cipher.txt

Decrypt cipher.txt with reverse key in tr:

$tr 'ibejrzpwqtolmsnycxkfhuvdqa' 'abcdefghijklmnopqrstuvwxyz' < cipher.txt > p.txt

You now should be able to read the original text from p.txt

2. Breaking the Substitution Cipher

a. The following text was encrypted with a substitution cipher. (Spaces and punctuation marks were preserved from the plaintext)

THVVWCZTQ!

CZ WECQ YIAQQ, LFN JCII IVAHZ WEV DCPPVHVZYV MVWJVVZ VZYHLOWCFZ AZD ANWEVZWCYAWCFZ. WECQ RVQQATV CQ VZYHLOWVD JCWE A BVHL CZQVYNHV VZYHLOWCFZ QYEVRV. CDVAIIL, AZ VZYHLOWCFZ QYEVRV QEFNID AIIFJ FZIL ANWEFHCGVD OAHWCVQ, JEF XZFJ WEV XVL, WF HVAD WEV RVQQATV. EFJVBVH, LFN SNQW HVAD WEV RVQQATV JCWEFNW XZFJCZT WEV XVL. EVZYV, WEV VZYHLOWCFZ QYEVRV CQ CZQVYNHV.

Find the plaintext. Feel free to do this by hand, or with a computer program. You may find tables of English letter frequencies on the web.

b. Try to decrypt to find the plaintext from ciphertext.txt using whichever techniques that you could. Describe in detail how did you do to achieve the final result

3. Breaking the Vigenere Cipher

The one-time pad is along sequence of random letters. These letters are combined with the plaintext message to produce the ciphertext. To decipher the message, one must have a copy of the one-time pad to reverse the process. A one-time pad should be used only once (hence the

name) and then destroyed. This is the first and only encryption algorithm that has been proven to be unbreakable.

To encipher a message, you take the first letter in the plaintext message and “add" it to the letter from the one-time pad. To decipher a message, you take the ciphertext and “subtract” the letter from the one-time pad. By “adding”, we mean the following:

Let A be the 0th letter in the alphabet. ... Let Z be the 25th letter in the alphabet. To “add” letter S (the 18th letter) and C (the 2nd letter), take: 18+2 mod 26 =20

which gives us U (the 20th letter).

To “subtract” letter U (the 20th letter) and C (the 2nd letter), take: 20−2 mod 26 =18

which gives us S (the 18th letter). Here’s a larger example:

Plaintext SECRETMESSAGE

One-time pad CIJTHUUHMLFRU

Ciphertext UMLKLNGLEDFXY

Mr. X was not paying attention in her security class, and decided to reuse the same 5-letter “one-time pad” over and over to encipher her plaintext. The result is shown below. (punctuation in the message is retained to make codebreaking easier)

TWT CMGWI ZJTWT AIOEAP XO QT DICJGP MN IWPMR ETCWOCH, SSUHTD, TAETCW, ACS PJFTREW, AVPTRSI JYVEPHZRAQAP WEPGNLEH PYH STXKYRTH, DLAAA YST QT GMOAPEID, PCO RO LPCVACID WHPAW MSHJP, FUI JASN EGZFAQAP GAJHP, WUEEZVTTS MC OPIS SR PUQMRBPEMOC, PYH PPGEMCJALVLN SPWCGXMMNV ISI PAPNI TD QP WEPGNLES, PYH TWT AIRHDYW OG ISMNVH ES BT HPMZTS.

Submit your python program and the deciphered message. You can use whatever decipherment technique you want; feel free to read online to find ideas! (Just make sure to cite your sources!)

4. MD5 Collisions

MD5 was once the most widely used cryptographic hash function, but today it is considered dangerously insecure. This is because cryptanalysts have discovered efficient algorithms for finding collisions—pairs of messages with the same MD5 hash value.

The first known collisions were announced on August 17, 2004 by Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu. Here’s one pair of colliding messages they published:

Message 1:

d131dd02c5e6eec4693d9a0698aff95c
2fcab58712467eab4004583eb8fb7f8955ad340609f4b30283e488832571415a
085125e8f7cdc99fd91dbdf280373c5bd8823e3156348f5bae6dacd436c919c6
dd53e2b487da03fd02396306d248cda0e99f33420f577ee8ce54b67080a80d1e
c69821bcb6a8839396f9652b6ff72a70

Message 2:

d131dd02c5e6eec4693d9a0698aff95c
2fcab50712467eab4004583eb8fb7f8955ad340609f4b30283e4888325f1415a
085125e8f7cdc99fd91dbd7280373c5bd8823e3156348f5bae6dacd436c919c6
dd53e23487da03fd02396306d248cda0e99f33420f577ee8ce54b67080280d1e
c69821bcb6a8839396f965ab6ff72a70

Convert each group of hex strings into a binary file.
$ xxd -r -p file.hex > file

What are the MD5 hashes of the two binary files? Verify that they're the same.

$ openssl dgst -md5 file1 file2

What are their SHA-256 hashes? Verify that they're different.

$ openssl dgst -sha256 file1 file2

Lab B: Crypto-Lab: – Exploring Collision-Resistance, Pre-Image Resistance and MACs

(This lab is adapted from SEED labs of Dr. Wenliang Du, Syracuse University)

1. **Overview**

The learning objective of this lab is for students to get familiar with pre-image resistant hash functions and Message Authentication Code (MAC). After finishing the lab, in addition to gaining a deeper understanding of the concepts, students should be able to use tools and write programs to generate hash values and a MAC for a given message.

2. **Lab Environment**

   Secure Sockets Layer (SSL) is an application-level protocol which was developed by the Netscape Corporation for the purpose of transmitting sensitive information, such as Credit Card details, via the Internet

   OpenSSL is a robust, commercial-grade implementation of SSL tools, and related general-purpose library based upon SSL, developed by Eric A. Young and Tim J. Hudson

   OpenSSL is already installed on SEEDUbuntu.

   Bless Hex editor (install yourself)

3. **Lab Tasks**

Task 1: Generating Message Digest and MAC

In this task, we will play with various hash algorithms. You can use the openssl dgst command to generate the hash value for a file. To see the manuals, you can type man openssl and man. Replace the dgsttype with a specific hash algorithm, such as -md5, -sha1, -sha256, etc.

In this task, you are encouraged to try at least 3 different algorithms. You can find the supported hash algorithms by typing **man openssl**.

Task 2: The Randomness of a Hash

To understand the properties of hash functions, we would like to do the following exercise for MD5 and SHA256:

a. Create a text file of any length.
b. Generate the hash value H1 for this file using a specific hash algorithm.
c. Flip one bit of the input file. You can achieve this modification using Bless.
d. Generate the hash value H2 for the modified file.
e. Observe whether H1 and H2 are similar or not. Write a short program to count how many bits are different between H1 and H2.

Calculating the Hamming Distance is a mathematical way to determine the differences between two objects (or strings, boxes, files, hashes, etc.). You are required to write a program which calculates the Hamming distance between two different hashes for their bits. For example, take the hash ”ab123” and ab122”. These two hexadecimal values result in:

10101011000100100011

10101011000100100010

in binary respectively. The Hamming Distance between both of them is 1, because only one bit is different between the two hex values. For this program, input two hexadecimal values via command line arguments, and name your program yourname_hamming.py.

Important! Make sure that you can run your program in terminal like:

python3 yourname_hamming.py ab123 ab122

The output for this case will be: 1

Another sample command and subsequent output:

python3 yourname_hamming.py ff3a7 ff4b1

The output for this case will be: 6

Next, create two files of at least 10 readable ascii characters (not hex values) that differ by only 1 bit (not just a character!). Hash both of these strings using a cryptographic hash and then observe the hamming distance between both hashes of your two strings.

For this task you need to submit the following files:

Yourname_hamming.py

A file named string1.txt with a single string on a single line

A file named string2.txt with a single string on a single line with a one-bit difference from string1.txt