Understanding Cryptography Chapter 1 Solutions

Jakob Graves

December 4, 2021

1.1. The following was encrypted with a substitution cypher. Decrypt the cyphertext:

> lrvmnir bpr sumvbwvr jx bpr lmiwv yjeryrkbi jx qmbm wi bpr xjvni mkd ymibrut jx irhx wi bpr riirkvr jx ymbinlmtmipw utn qmumbr dj w ipmhh but bj rhnvwdmbr bpr yjeryrkbi jx bpr qmbm mvvjudwko bj yt wkbrusurbmbwjk lmird jk xjubt trmui jx ibndt wb wi kjb mk rmit bmiq bj rashmwk rmvp yjeryrkb mkd wbi iwokwxwvmkvr mkd ijyr ynib urymwk nkrashmwkrd bj ower m vjyshrbr rashmkmbwjk jkr cjnhd pmer bj lr fnmhwxwrd mkd wkiswurd bj invp mk rabrkb bpmb pr vjnhd urmvp bpr ibmbr jx rkhwopbrkrd ywkd vmsmlhr jx urvjokwgwko ijnkdhrii ijnkd mkd ipmsrhrii ipmsr w dj kjb drry ytirhx bpr xwkmh mnbpjuwbt lnb yt rasruwrkvr cwbp qmbm pmi hrxb kj djnlb bpmb bpr xjhhjcwko wi bpr sujsru msshwvmbwjk mkd wkbrusurbmbwjk w jxxru yt bprjuwri wk bpr pjsr bpmb bpr riirkvr jx jqwkmcmk qmumbr cwhh urymwk wkbmvb

Solution: Let A represent the ciphertext string and use python to do a letter frequency analysis. The following code will comlpete the task very quickly:

```
from collections import Counter

x = list('A')

# create a dictionary counting character frequency
c = Counter(x)

# remove spaces and new line characters etc.
del c[' '], c['\n'], c['.'], c[',']

# count all characters
number_letters = len(x) - x.count(' ') - x.count('\n')
print('Total number of letters:', number_letters)

# normalize the frequencies into a distribution
x = {k: v / number_letters for k, v in c.items()}
```

```
# sort by value. note that this is now a list of tuples
d = sorted(x.items(), key=lambda xd: xd[1], reverse=True)

# make a table for easy reading
print("{:<8} {:<15}".format('Letter', 'Freq. Dist.'))
for k, v in d:
    print("{:<8} {:<15}".format(k, v))

# now replace characters according to frequency analysis
# define ordered strings of letter frequency
eng_letter_freq = 'etaoinshrdlcumwfgypbvkjxqz'
ciph_freq = ''
for i, j in d:
    ciph_freq = ciph_freq + i
if len(ciph_freq) < 26:
    ciph_freq = ciph_freq + '.' * (26 - len(ciph_freq))

# use the strings to replace characters
trans = str.maketrans(ciph_freq, eng_letter_freq)
semi_decrypted = string.translate(trans)
print(semi_decrypted)
```

Here is the first part of the output arranged in a nice table:

Character Frequency From Sample			
Letter	Relative Frequency	Letter	Relative Frequency
r	0.130030	y	0.029411
b	0.105263	n	0.026315
m	0.095975	s	0.026315
k	0.0758513	t	0.020123
j	0.0743034	l	0.012383
w	0.072755	q	0.010835
i	0.0634674	o	0.010835
p	0.046439	e	0.007739
u	0.037151	a	0.007739
d	0.035603	c	0.007739
h	0.035603	f	0.001547
v	0.034055	g	0.001547
x	0.030959		

And here is the semi-decrypted text the program outputs with the characters replaced by a character frequency mapping:

> yecawse the fractnce iu the yasnc mivemeots iu pata ns the uicws aod masterg iu selu ns the esseoce iu matswyagashn rgw parate di n shall trg ti elwcndate the mivemeots iu the pata accirdnob ti

> mg noterfretatnio yased io uirtg gears iu stwdg nt ns oit ao easg tasp ti ekflano each mivemeot aod nts snbonuncaoce aod sime mwst remano woekflanoed ti bnve a cimflete ekflaoatnio ioe jiwld have ti ye xwalnuned aod nosfnred ti swch ao ekteot that he ciwld reach the state iu eolnbhteoed mnod cafayle iu recibonqnob siwodless siwod aod shafeless shafe n di oit deem mgselu the unoal awthirntg ywt mg ekferneoce jnth pata has leut oi diwyt that the uillijnob ns the frifer afflncatnio aod noterfretatnio n iuuer mg theirnes no the hife that the esseoce iu ipnoajao parate jnll remano notact

The program does not completely decode the cyphertext; however, it does decode enough to get some of the most frequent characters, enough to go through and manually make the cipher key. After a little bit of analysis, the following table gives us the complete key. Note that if there were proper punctuation or upper case letters or other symbols, it would be slighlty more complicated and the program would need to be adjusted. This program works for blocks of lowercase letter text with periods, commas, and spaces.

Cipher Key			
Plaintext	Cyphertext	Plaintext	Cyphertext
a	m	n	k
b	l	o	j
c	v	p	s
d	d	q	f
e	r	r	u
f	x	s	i
g	o	t	b
h	p	u	n
i	w	v	e
j	z	w	c
k	q	x	a
l	h	y	t
m	y	z	g

Here is the hidden message:

> because the practice of the basic movements of kata is the focus and mastery of self is the essence of matsubayashi ryu karate do i shall try to elucidate the movements of the kata according to my interpretation based on forty years of study
>
> it is not an easy task to explain each movement and its significance and some must remain unexplained to give a complete explanation one would have to be qualified and inspired to such an extent that he could reach the state of enlightened mind capable of recognizing soundless sound and shapeless shape
>
> i do not deem myself the final authority but my experience with kata has left no doubt that the following is the proper application

> and interpretation i offer my theories in the hope that the essence of okinawan karate will remain intact

This was said by Shōshin Nagamine in his book

1.2. We recieved the following cyphertext which was encoded wuth a shift cypher:

> xultpaajcxitltlxaarpjhtiwtgxktghidhipxciwtvgtpilpitghlxiwiwtxgqadds

Who wrote the message?

Solution: The first part of the python program will give us a frequency distribution, so run that on the new cyphertext. It shows that t and i in the cyphertext are the most common by a wide margin. One of them is almost certainly mapped to e, and the other to t. Since t is not mapped to t, it should be the case that $t \to i$. Recall

> **Definition 1.4.3**: Let $x, y, k \in \mathbb{Z}_{26}$
>
> **Encryption:** $e_k(x) \equiv x + k \mod 26$
>
> **Decryption:** $d_k(y) \equiv y - k \mod 26$

i is the ninth letter of the alphabet and t is the twentieth, so in this case $k = -11$. The following python function cycles letters in a string modulo 26. This is the simplest form of Caesar cypher.

```
import string


def caesar(plaintext, shift):
    alphabet = string.ascii_lowercase
    shifted_alphabet = alphabet[shift:] + alphabet[:shift]
    table = str.maketrans(alphabet, shifted_alphabet)
    return plaintext.translate(table)
```

Adding 11 to each charcter modulo 26 and manually inputing spaces gives:

> if we all unite we will cause the rivers to stain the great waters with their blood

This is a line from Tecumseh's Speech to the Osages (1812)

1.3. We consider the long-term security of the Advanced Encryption Standard (AES) with a key length of 128-bit with respect to exhaustive key-search attacks. AES is perhaps the most widely used symmetric cipher at this time.

(a) Assume that an attacker has a special purpose application specific integrated circuit (ASIC) which checks 5×10^8 keys per second, and she has a budget of \$1

million. One ASIC costs \$50, and we assume 100% overhead for integrating the ASIC. How many ASICs can we run in parallel with the given budget? How long does an average key search take? Relate this time to the age of the universe, which is about 10^{10}.

(b) We try now to take advances in computer technology into account. Predicting the future tends to be tricky but the estimate usually applied is Moore's Law, which states that the computer power doubles every 18 months while the costs of integrated circuits stay constant. How many years do we have to wait until a key-search machine can be built for breaking AES with 128 bit with an average search time of 24 hours? Again, assume a budget of \$1 million (do not take inflation into account).

Solution:

> **Fact:** It is known that a cipher whose key length is n **bits** has a keyspace with 2^n elements.

(a) With \$1 million, we can buy 10,000 ASICs at \$100 each and check 5×10^{12} keys per second. There are 2^{128} possible keys, and on average we need to only check half the keys before breaking the code, so we want the time to make 2^{127} checks. It would take $(2^{127})(5 \times 10^{-12}) = 3.4 \times 10^{25}$ seconds to do those checks. This works out to about 1.08×10^{18} years, so it would take approximately until the end of time to go through all the keys. This is longer than the universe has existed.

> **Moore's Law:** Computing power doubles every 18 months while the costs stay constant

(b)

1.4. We now consider the relation between passwords and key size. For this purpose we consider a cryptosystem where the user enters a key in the form of a password.

(a) Assume a password consisting of 8 letters, where each letter is encoded by the ASCII scheme (7 bits per character, i.e., 128 possible characters). What is the size of the key space which can be constructed by such passwords?

(b) What is the corresponding key length in bits?

(c) Assume that most users use only the 26 lowercase letters from the alphabet instead of the full 7 bits of the ASCII-encoding. What is the corresponding key length in bits in this case?

(d) At least how many characters are required for a password in order to generate a key length of 128 bits in case of letters consisting of (i) 7-bit characters? (ii) 26 lowercase letters from the alphabet?

Solution:

(a) There are $128^8 = 2^{56}$ passwords in the keyspace

(b) The key length is $7 \times 8 = 56$ bits since the password is 8 characters and each is 7 bits.

(c) We need x such that $2^x = 26 \Rightarrow x = \log_2 26 = 4.7$. So each character is 4.7 bits and the key length in bits is $4.7 \times 8 = 37.6$ bits

(d) (i) $128/7 = 18.29$ so we need at least 19 characters to have a 128-bit keylength. (ii) Similarly, $128/4.7 = 27.23$ so we need at least 28 characters.

1.11. This problem deals with the affine cipher with the key parameters a = 7, b = 22.

(a) Decrypt the text below:

falszztysyjzyjkywjrztyjztyynaryjkyswarztyegyyj

(b) Who wrote the line?

Solution:

(a) The decoded plaintext is

first the sentence and then the evidence said the queen

I will write some python code to solve this later

(b) This is a line from *Alice's Adventures Underground* by Lewis Carol, which is apparantly the original version of the story before it was later rewritten to become *Alice's Adventures in Wonderland*

1.12.

Solution:

(a) The encryption and decryption equations are the same as the affine cypher, except now we need $\mod 30$

(b) The key space is the product between the number of elements with inverses mod 30 and the number of characters. $|K| = m\varphi(m)$ where m is the number of charactres in the cypher.

1.13. In an attack scenario, we assume that the attacker Oscar manages somehow to provide Alice with a few pieces of plaintext that she encrypts. Show how Oscar can break the affine cipher by using two pairs of plaintext–ciphertext, (x_1, y_1) and (x_2, y_2). What is the condition for choosing x_1 and x_2?

In practice, such an assumption turns out to be valid for certain settings, e.g., encryption by Web servers, etc. This attack scenario is, thus, very important and is denoted as a chosen plaintext attack

Solution:

Oscar has the two following linear congruences that he needs to solve for the key parameters $[a, b]$

$$y_1 \equiv ax_1 + b \mod m$$

$$y_2 \equiv ax_2 + b \mod m$$

Solving one for b and inserting it into the other yields

$$y_2 - y_1 \equiv a(x_2 - x_1) \mod m$$

which is just a linear congruence that is solvable for a if and only if $gcd(x_2 - x_1, m) = 1$. Once we have a, then b follows easily.

1.14. An obvious approach to increase the security of a symmetric algorithm is to apply the same cipher twice, i.e.:

$$y = e_{k2}(e_{k1}(x))$$

As is often the case in cryptography, things are very tricky and results are often different from the expected and/ or desired ones. In this problem we show that a double encryption with the affine cipher is only as secure as single encryption! Assume two affine ciphers $e_{k1} = a_1x + b_1$ and $e_{k2} = a_2x + b_3$.

(a) Show that there is a single affine cipher $e_{k3} = a_3x + b_3$ which performs exactly the same encryption (and decryption) as the combination $e_{k2}(e_{k1}(x))$.

(b) Find the values for a_3, b_3, when $a_1 = 3, b_1 = 5$ and $a_2 = 11, b_2 = 7$.

(c) For verification: (i) encrypt the letter κ first with e_{k1} and the result with e_{k2}, and (ii) encrypt the letter κ with e_{k3}.

(d) Briefly describe what happens if an exhaustive key-search attack is applied to a double-encrypted affine ciphertext. Is the effective key space increased?

Solution:

(a) Encrypting twice gives

$$e_{k2}(e_{k1}(x)) = a_2(a_1x + b_1) + b_2$$

$$= a_1a_2x + a_2b_1 + b_2$$

Let $a_1a_2 = a_3$ and let $a_2b_1 + b_2 = b_3$, then

$$e_{k2}(e_{k1}(x)) = a_3x + b_3$$

(b)

$$a_3 = 3 \cdot 11 = 33 \equiv 6 \mod 36$$

$$b_3 = 11 \cdot 5 + 7 = 62 \equiv 10 \mod 26$$

(d) The keyspace has not changed. It is still $m\varphi(m)$ where m is the number of distinct characters and φ is Euler's Totient Function.