FIT5124 – Assignment 1 – Xiaohui Ding – 31291252

3 Security Analysis of Searchable Symmetric Encryption

3.1 Security Definition

Ans:

Given the security games:

Real$_\Pi$A(λ):

1. Adversary A(1^k) choose M, **M**.

2. Game runs (K, C, **c**) ← Matrix.Gen(1^k), Matrix.Enc(K, M, **M**), and gives (C, **c**) to A.

3. Then A requests again and again using client input *in*.

4. The game then runs the scheme with client input (K, *in*) and server input (C, **c**) and gives the transcript to A.

5. Eventually, A returns a bit for game to use as its own output.

Ideal$_\Pi$A, S (λ):

1. Adversary A(1^k) choose M, **M**.

2. Game runs (K', C', **c**') ← Matrix.Gen($1^{L(k)}$), Matrix.Enc(K', L(C, **c**)) and gives (C', **c'**) to A.

3. Then A requests again and again to run the Lookup operation using client input *in*.

4. The game gives the output of L(*in*) to Simulator, which outputs a simulated transcript that is given to A.

5. Eventually, A returns a bit for game to use as its own output.

Definition: Π is Leakage-secure against non-adaptive attacks if for all adversaries A there exists a simulator S such that:

$$Pr[Real_{\Pi}(\lambda) = 1] - Pr[Ideal_{\Pi}(\lambda) = 1] \leq neg(\lambda)$$

3.2 Security Analysis

Ans:

1) Leakage function definition

- $L_{Setup:\, Gen(1^k),\ Enc(K,M,\boldsymbol{M})} = (C, \boldsymbol{c}, \lambda_1, \lambda_2, n, |j|, |v|, |c|)$,

  where:

  λ_1, λ_2 is the number of rows and columns of C, n is the number of elements in **c**,

$|j|$ is the length of label $\pi(i) \oplus F_{K_1}(\alpha, \beta)$,

$|v|$ is the length of $v_i \oplus F_{K_1}(\alpha, \beta)$,

$|c|$ is the length of the element in **c**.

- $L_{Search:\, Lkp_e(\gamma,\tau)} = \big(AP(\tau), SP(\tau)\big)$
  - $AP(\tau) = (s, \alpha', \beta'), (j, v)$

    where:

    (s, α', β') is the search token,

    j is the permuted pointer $\pi(i)$ to m_i,

    v is the semi-private data item related to m,
  - $SP(\tau) = (s, \alpha', \beta'), l$

    where:

    l is from 1 to x,

    x is the total number of queries.

2) How to simulate the security game

Given queries $\{\tau_1, \tau_2, \dots, \tau_x\}$, the simulator S will generate a simulated matrix and vector from $L_{Setup:\, Gen(1^k),\ Enc(K,M,\boldsymbol{M})}$ and $L_{Search:\, Lkp_e(\tau)}$.

S will create a $\lambda_1 \times \lambda_2$ matrix C'.

For each lookup token τ, S generates random keys $K_1{}'$, $K_2{}'$ with the same bit length of K_1, K_2, and a random integer pair (α'', β'') within the range $\lambda_1 \times \lambda_2$.

Then S computes the search token from $K_1{}'$, $K_2{}'$, (α'', β''):

$s' \coloneqq F_{K_1{}'}(\alpha'', \beta'')$,

$(\alpha^*, \beta^*) \coloneqq P_{K_2{}'}(\alpha'', \beta'')$,

S also computes $(j', v') \coloneqq (j, v) \oplus F_{K_1{}'}(\alpha'', \beta'')$ and inserts (j', v') into $C'[\alpha^*, \beta^*]$.

For the rest location of C', S generates random string pairs with the same bit length of $(j, v) \oplus s$.

We also want to simulate a **c'**. But unluckily, we don't know anything about **m**, which means we can't simulate a **c'** and use it with (j, v) to get a same search result **m**.

In fact, in the scheme, the encryption and decryption of **m** are both on the client side, and this is why we can't simulate the whole process.

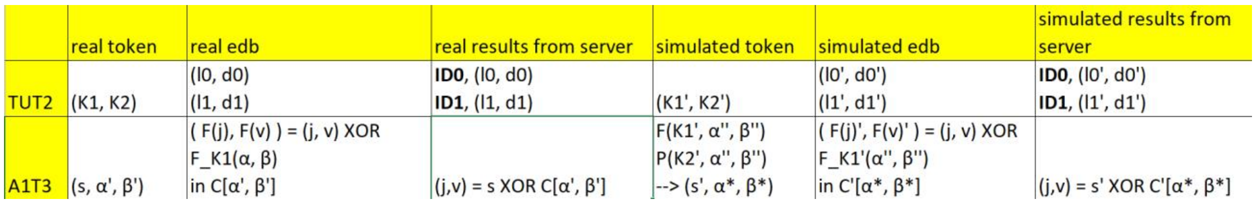
3) How the simulated view is indistinguishable from the real view

- C & C' are indistinguishable. They used the same PRF, PRP and symmetric encryption Π. They have same sizes, and each data item stored in both of the pairs have the same bit lengths.
- For each lookup query q_m, K_1', K_2' have the same length of K_1, K_2, and (α'', β'') is within $\lambda_1 \times \lambda_2$ which means it is also indistinguishable from (α, β).
- Thus, for each lookup query q_m, the simulated query (s', α^*, β^*) is identical to (s, α', β').
- The same (j, v) with the help of the same PRF & PRP will be inserted into C'. Thus, the recovered (j, v) from $C'[\alpha^*, \beta^*]$ are also same with the recovered (j, v) from $C[\alpha', \beta']$. The search result is indistinguishable.
- For each lookup query q_m, after (j, v) is recovered and returned to the client side, the client can use the same j with his/her own K_3 to recover the same m_j. The final result is indistinguishable.

In conclusion, the simulator and the real-world case are indistinguishable.

Note: S firstly use $K_1', K_2', (\alpha'', \beta'')$ to calculate (s', α^*, β^*) rather than directly use the leakage (s, α', β') to simulate a (s', α^*, β^*) because we want to simulate the whole process as much as possible. If we don't, there might be some residual leakage, for example, simulating time, from the simulating process.

I also want to add this comparison with our Tut 2 solution to demonstrate more clearly how the whole simulation process works:



	real token	real edb	real results from server	simulated token	simulated edb	simulated results from server
TUT2	(K1, K2)	(I0, d0) (I1, d1)	**ID0**, (I0, d0) **ID1**, (I1, d1)	(K1', K2')	(I0', d0') (I1', d1')	**ID0**, (I0', d0') **ID1**, (I1', d1')

A1T3	(s, α', β')	(F(j), F(v)) = (j, v) XOR F_K1(α, β) in C[α', β']	(j,v) = s XOR C[α', β']	F(K1', α'', β'')
				P(K2', α'', β'')
				--> (s', α*, β*)
					(F(j)', F(v)') = (j, v) XOR F_K1'(α'', β'') in C'[α*, β*]	(j,v) = s' XOR C'[α*, β*]

4 Count Attacks against Searchable Symmetric Encryption

4.1 Case study I

d1 = {w1, w2, w3, w4, w6}	d2 = {w1, w2, w3, w6}
d3 = {w1, w2, w4, w6}	d4 = {w1, w4, w6}
d5 = {w4, w5, w6}	d6 = {w4, w6}

We can convert this into a table:

	d1	d2	d3	d4	d5	d6

w1	1	1	1	1		
w2	1	1	1			
w3	1	1				
w4	1		1	1	1	1
w5					1	
w6	1	1	1	1	1	1

We then generate the **keyword co-occurrence matrix** (presented as the table below):

	w1	w2	w3	w4	w5	w6
w1	4	3	2	3	0	4
w2	3	3	2	2	0	3
w3	2	2	2	1	0	2
w4	3	2	1	5	1	5
w5	0	0	0	1	1	1
w6	4	3	2	5	1	6

In general, we have to fill the rest of the table. But in this case, since the numbers in diagonal are all unique, we can simply deduct the match between keywords and queries without the grey numbers. (I put them here for better understanding how key-word co-occurrence works.)

q	**eids**						**Total number**
q1	e1	e2	e3	e5	e6		5
q2	e3	e4					2
q3	e1	e2	e3	e4			4
q4	e2	e3	e4				3
q5	e1	e2	e3	e4	e5	e6	6
q6	e6						1

1) q1 matches 5 permuted document IDs, and w4 matches 5 document IDs too. Thus, q1 → w4.

2) q2 matches 2 permuted document IDs, and w3 matches 2 document IDs too. Thus, q2 → w3.

3) q3 matches 4 permuted document IDs, and w1 matches 4 document IDs too. Thus, q3 → w1.

4) q4 matches 3 permuted document IDs, and w2 matches 3 document IDs too. Thus, q4 → w2.

5) q5 matches 6 permuted document IDs, and w6 matches 6 document IDs too. Thus, q5 → w6.

6) q6 matches 1 permuted document IDs, and w5 matches 1 document IDs too. Thus, q6 → w5.

Final result:

q1 → w4	q2 → w3	q3 → w1
q4 → w2	q5 → w6	q6 → w5

4.2 Case study II

d1 = {w1, w2, w3, w4, w6}

d2 = {w2, w3, w6}

d3 = {w1, w3, w4, w6}

d4 = {w1, w4, w6}

d5 = {w4, w5, w6}

d6 = {w5}

	d1	d2	d3	d4	d5	d6
w1	1		1	1		
w2	1	1				
w3	1	1	1			
w4	1		1	1	1	
w5					1	1
w6	1	1	1	1	1	

	w1	w2	w3	w4	w5	w6
w1	3	1	2	3	0	3
w2	1	2	2	1	0	2
w3	2	2	3	2	0	3
w4	3	1	2	**4**	1	4
w5	0	0	0	1	2	1
w6	3	2	3	4	1	**5**

q	**eids**						**Total number**
q1	e2	e3	e4	e5	e6		5
q2	e2	e3	e5	e6			4
q3	e1	e3					2

q4	e4	e5					2
q5	e2	e5	e6				3
q6	e4	e5	e6				3

Analysis:

1) Unique pair:

q1 → w6, for unique occurrence 5.

q2 → w4, for unique occurrence 4.

2) Possible pairs:

{q3, q4} × {w2, w5} for occurrence 2.

{q5, q6} × {w1, w3} for occurrence 3.

3) We will need **query co-occurrence matrix**.

	q1	q2	q3	q4	q5	q6
q1	5	4	1	2	3	3
q2	4	4	1	1	3	2
q3	1	1	2	0	0	0
q4	2	1	0	2	1	2
q5	3	3	0	1	3	2
q6	3	4	0	2	2	3

4)

Usually, we will observe the unique correlation of q3-q6 with q1 & q2, but here we have a special pattern: **correlation 0 only occurs in q3 & w5**. Thus, we immediately know that **q3 → w5**, which leads to the other pair **q4 → w2**.

Check:

Correlation(w2, w4) = 1 = Correlation(q4, q2)

Correlation(w2, w6) = 2 = Correlation(q4, q1)

Thus, this is correct.

5)

Finally, we will look at {q5, q6} × {w1, w3}.

By observing, we should use q6 as our break point.

Correlation (q6, q2) = 2 = Correlation (w3, w4), and this is a unique correlation. Thus, **q6 → w3**, which also leads to **q5 → w1**.

Check:

Correlation (q5, q1) = 3 = Correlation (w1, w6)

Correlation (q5, q2) = 3 = Correlation (w1, w4)

Thus, this is correct.

6) The counterattack does recover all queries, and the final results are below:

q1 → w6	q2 → w4	q3 → w5
q4 → w2	q5 → w1	q6 → w3

4.3 Case study III

d1 = {w1, w2, w3, w6}   d2 = {w2, w3, **w4,** w6} # update!

d3 = {w1, w2, w3, w4, w6}   d4 = {w1, w3, w4, w6}

d5 = {w5, w6}   d6 = {w5, w6}

	d1	d2	d3	d4	d5	d6
w1	1		1	1		
w2	1	1	1			
w3	1	1	1	1		
w4		1	1	1		
w5					1	1
w6	1	1	1	1	1	1

	w1	w2	w3	w4	w5	w6
w1	3	2	3	2	0	3
w2	2	3	3	1	0	3
w3	3	3	4	2	0	4
w4	2	1	2	3	0	2
w5	0	0	0	0	2	2
w6	3	3	4	2	2	6

q	**eids**						**Total number**
q1	e1	e5	e6				3
q2	e1	e2	e6				3
q3	e3	e4					2
q4	e1	e2	e3	e4	e5	e6	6
q5	e2	e5	e6				3
q6	e1	e2	e5	e6			4

	q1	q2	q3	q4	q5	q6
q1	**3**	2	0	3	2	3
q2	2	**3**	0	3	2	3
q3	0	0	2	2	0	0
q4	3	3	2	6	3	4
q5	2	2	0	3	**3**	3
q6	3	3	0	4	3	4

Analysis:

1) ~~We notice that the occurrence in query does not totally match occurrence in keyword. This is an obvious evidence that the encryption has some padding.~~(Since the assignment has been updated, this is not the situation.)

So, I'd like to start from Correlation 0, since the padding can only add files rather than removing files. Thus, we can see the only one who has Correlation = 0 more than once is q3 & w5.

q3 → w5

2) Unique pair

But we are lucky. Since the unique occurrence 4 and 6 in keyword matrix are larger than other occurrences, and in query matrix they are also unique and larger than others, we can conclude that they are matched.

Thus, **q4 → w6**, and **q6 → w3**.

3) Possible pair

{q1, q2, q5} × {w1, w2, w4}

We will analyze the Correlation between them and the revealed pairs.

For w3:

Correlation (w3, w1) = 3		Correlation (q6, q1) = 3
Correlation (w3, w2) = 3	&	Correlation (q6, q2) = 3
Correlation (w3, w4) = ~~2~~ 3		Correlation (q6, q5) = 3

This tells nothing.

For w5:

Correlation (w5, w1) = 0		Correlation (q3, q1) = 0
Correlation (w5, w2) = 0	&	Correlation (q3, q2) = 0
Correlation (w5, w4) = 0		Correlation (q3, q5) = 0

This gives no information.

For w6, situations are similar.

It seems that we have no way to go. But hang on, can we break this by matching eid & id?

q3 → w5: {e3, e4} × {d5, d6}, thus {e1, e2, e5, e6} × {d1, d2, d3, d4}

We can observe this:

q1	e1	e5	e6
q2	e1	e2	e6
q5	e2	e5	e6

	d1	d2	d3	d4
w1	1		1	1
w2	1	1	1	
w4		1	1	1

Thus, **e6 → d3**. And you can finally conclude that no more information can be retrieved further. Sorry, adversary!

4) Information that leaks:

q3 → w5 q4 → w6 q6 → w3

e6 → d3

possible pairs: {q1, q2, q5} × {w1, w2, w4}

possible eid & id matches: {e3, e4} × {d5, d6}, {e1, e2, e5} × {d1, d2, d4}

5 Padding Countermeasures in Searchable Symmetric Encryption (20 Marks)

I wrote 3 .py files to not only calculate the result, but also finish the padding and output as .txt files. I will only put the main part of my code here, so you won't see something like writer.write repeatedly.

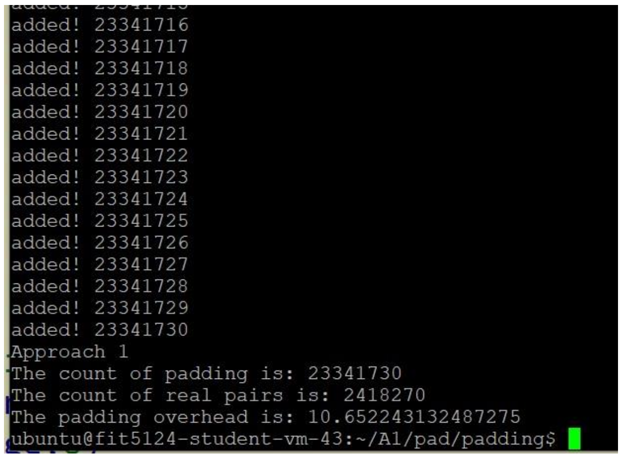
5.1 Approach #1

Source code:

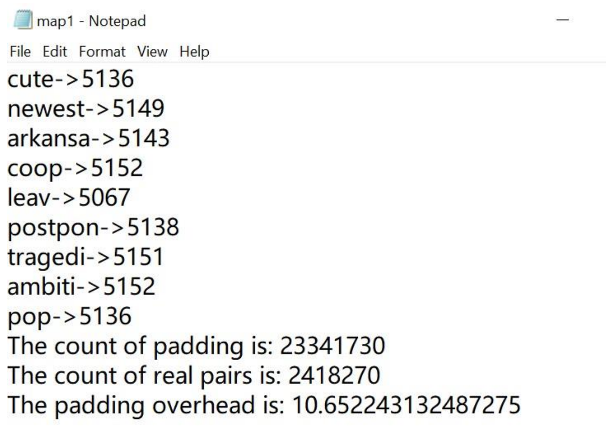
```
longest = 0
for id_list in inverted_dict.values():
    if len(id_list) > longest:
        longest = len(id_list)

# append pid & count real & padding
count_real = 0
count_padding = 0
for id_list in inverted_dict.values():
    count_real += len(id_list)
    if len(id_list) < longest:
        diff = longest - len(id_list)
        for i in range(diff):
            # id_list.add(utilities.generate_random_id())
            # because the program runs so slow, we replace the generator with
i. But in real case, we will use generate_random_id().
            id_list.add(i)
            count_padding += 1
            print("added!",count_padding)

# calculate overhead
pad_overhead = (count_real + count_padding) / count_real

print("Approach 1")
print("The count of padding is: " + str(count_padding))
print("The count of real pairs is: " + str(count_real))
print("The padding overhead is: " + str(pad_overhead))
```

The padding solution is to pad the id list of each keyword whose length(i.e., frequency; i.e., number of file id) is not the longest length to the longest length.

Results:



```
added! 23341716
added! 23341717
added! 23341718
added! 23341719
added! 23341720
added! 23341721
added! 23341722
added! 23341723
added! 23341724
added! 23341725
added! 23341726
added! 23341727
added! 23341728
added! 23341729
added! 23341730
Approach 1
The count of padding is: 23341730
The count of real pairs is: 2418270
The padding overhead is: 10.652243132487275
ubuntu@fit5124-student-vm-43:~/A1/pad/padding$
```



```
cute->5136
newest->5149
arkansa->5143
coop->5152
leav->5067
postpon->5138
tragedi->5151
ambiti->5152
pop->5136
The count of padding is: 23341730
The count of real pairs is: 2418270
The padding overhead is: 10.652243132487275
```

Fig 5.1.1 Fig 5.1.2

The overhead is ~10.65.

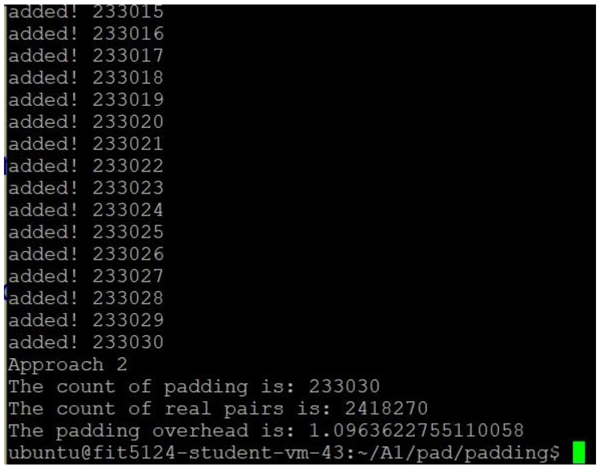
5.2 Approach #2

Source code:

```
# select multiplication
n = 100
# append pid to the nearest multiplication of 5 & count padding
count_real = 0
count_padding = 0
for id_list in inverted_dict.values():
    count_real += len(id_list)
    i = 1
    while (len(id_list) > i * n):
        i += 1 # find the nearest 5i larger than len(id_list)
    if len(id_list) < i*n:
        diff = i*n - len(id_list)
        for i in range(diff):
            if i in id_list:
                id_list.add(utilities.generate_random_id)
            else:
                id_list.add(i)
            count_padding += 1
            print("added!",count_padding)
# calculate overhead
pad_overhead = (count_real + count_padding) / count_real
```

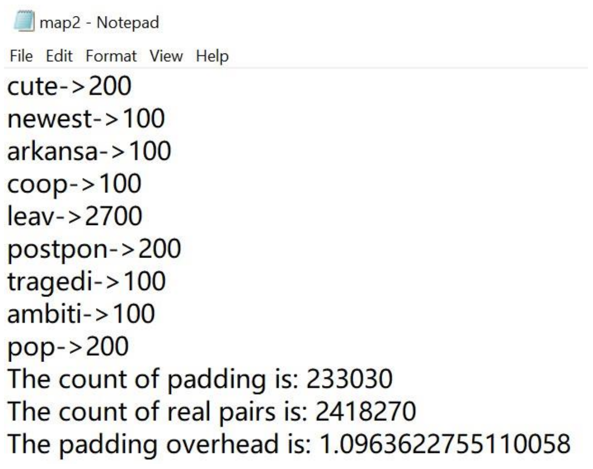
The padding solution is to pad the id list of each keyword whose length is not a multiplication of 100 to the nearest multiplication of 100 larger than the original length.

Results:



```
added! 233015
added! 233016
added! 233017
added! 233018
added! 233019
added! 233020
added! 233021
added! 233022
added! 233023
added! 233024
added! 233025
added! 233026
added! 233027
added! 233028
added! 233029
added! 233030
Approach 2
The count of padding is: 233030
The count of real pairs is: 2418270
The padding overhead is: 1.0963622755110058
ubuntu@fit5124-student-vm-43:~/A1/pad/padding$
```

Fig 5.2.1



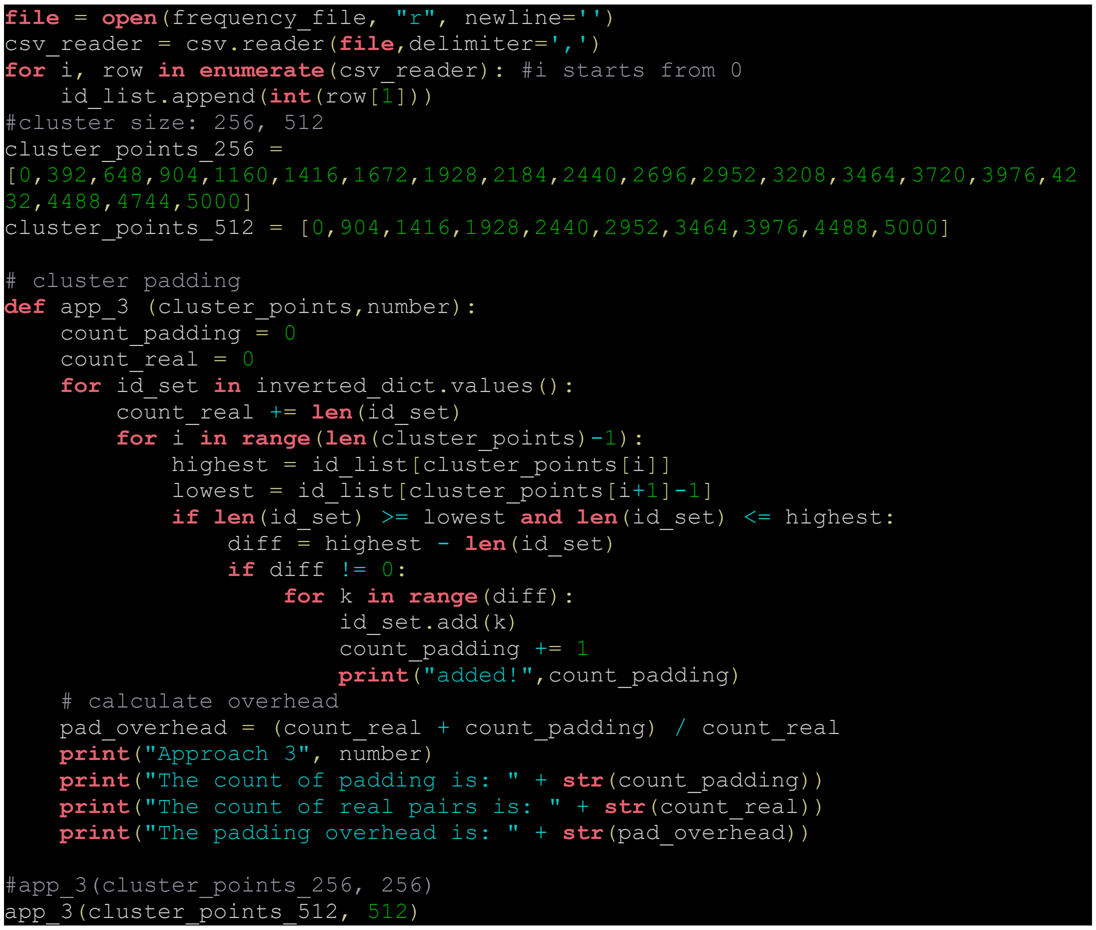
```
cute->200
newest->100
arkansa->100
coop->100
leav->2700
postpon->200
tragedi->100
ambiti->100
pop->200
The count of padding is: 233030
The count of real pairs is: 2418270
The padding overhead is: 1.0963622755110058
```

Fig 5.2.2

The overhead is ~1.10.

5.3 Approach #3

Source code:

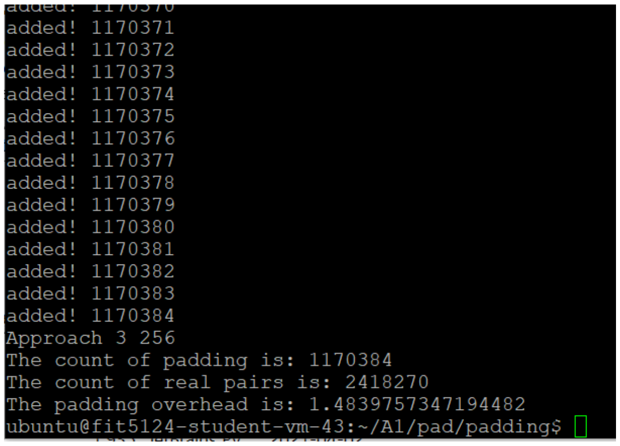


```
file = open(frequency_file, "r", newline='')
csv_reader = csv.reader(file,delimiter=',')
for i, row in enumerate(csv_reader): #i starts from 0
    id_list.append(int(row[1]))
#cluster size: 256, 512
cluster_points_256 =
[0,392,648,904,1160,1416,1672,1928,2184,2440,2696,2952,3208,3464,3720,3976,42
32,4488,4744,5000]
cluster_points_512 = [0,904,1416,1928,2440,2952,3464,3976,4488,5000]

# cluster padding
def app_3 (cluster_points,number):
    count_padding = 0
    count_real = 0
    for id_set in inverted_dict.values():
        count_real += len(id_set)
        for i in range(len(cluster_points)-1):
            highest = id_list[cluster_points[i]]
            lowest = id_list[cluster_points[i+1]-1]
            if len(id_set) >= lowest and len(id_set) <= highest:
                diff = highest - len(id_set)
                if diff != 0:
                    for k in range(diff):
                        id_set.add(k)
                        count_padding += 1
                        print("added!",count_padding)
    # calculate overhead
    pad_overhead = (count_real + count_padding) / count_real
    print("Approach 3", number)
    print("The count of padding is: " + str(count_padding))
    print("The count of real pairs is: " + str(count_real))
    print("The padding overhead is: " + str(pad_overhead))

#app_3(cluster_points_256, 256)
app_3(cluster_points_512, 512)
```

This padding solution is to use a cluster to put the keywords having similar frequencies into one small group and inside the group, pad all keywords to the largest frequency.

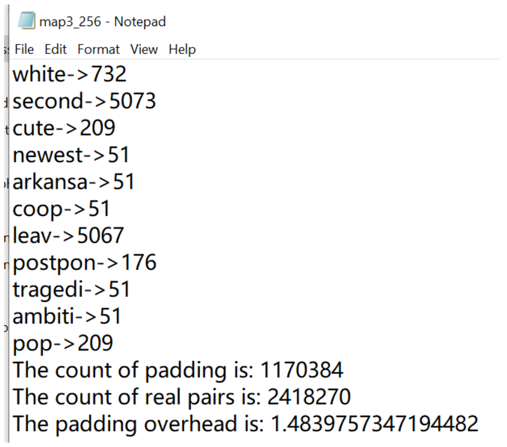
Results:

App #3 – 256



```
added! 1170371
added! 1170372
added! 1170373
added! 1170374
added! 1170375
added! 1170376
added! 1170377
added! 1170378
added! 1170379
added! 1170380
added! 1170381
added! 1170382
added! 1170383
added! 1170384
Approach 3 256
The count of padding is: 1170384
The count of real pairs is: 2418270
The padding overhead is: 1.4839757347194482
ubuntu@fit5124-student-vm-43:~/A1/pad/padding$
```

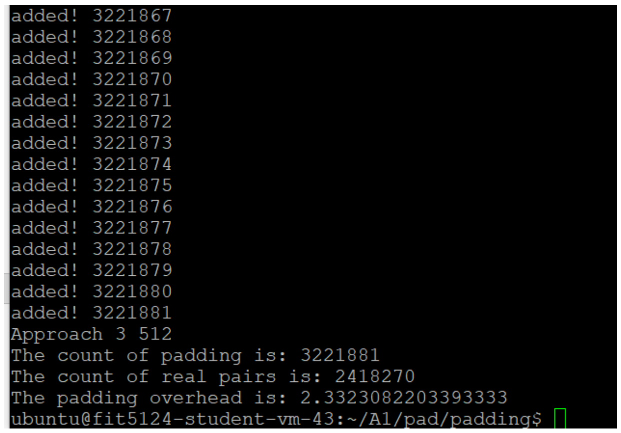
Fig 5.3.1



```
white->732
second->5073
cute->209
newest->51
arkansa->51
coop->51
leav->5067
postpon->176
tragedi->51
ambiti->51
pop->209
The count of padding is: 1170384
The count of real pairs is: 2418270
The padding overhead is: 1.4839757347194482
```

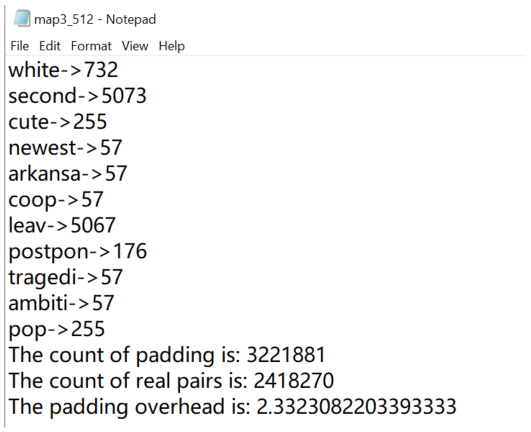
Fig 5.3.2

App #3 – 512



```
added! 3221867
added! 3221868
added! 3221869
added! 3221870
added! 3221871
added! 3221872
added! 3221873
added! 3221874
added! 3221875
added! 3221876
added! 3221877
added! 3221878
added! 3221879
added! 3221880
added! 3221881
Approach 3 512
The count of padding is: 3221881
The count of real pairs is: 2418270
The padding overhead is: 2.3323082203393333
ubuntu@fit5124-student-vm-43:~/A1/pad/padding$
```

Fig 5.3.3



```
white->732
second->5073
cute->255
newest->57
arkansa->57
coop->57
leav->5067
postpon->176
tragedi->57
ambiti->57
pop->255
The count of padding is: 3221881
The count of real pairs is: 2418270
The padding overhead is: 2.3323082203393333
```

Fig 5.3.4

The overhead for cluster 256 is ~1.48, for cluster 512 is ~ 2.33.

5.4 Comparison

	Approach #1	Approach #2	Approach #3 – 256	Approach #3 – 512
Padding overhead	~10.65224	~1.09636	~1.48398	~2.33231

The biggest overhead is approach #1, which is also called native padding.

App #2 has the smallest padding overhead, but we can see from the Fig 5.2.2: it’s very easy to guess that the padding pattern is multiplication of 100. Although we are not required to calculate the recovery ratio, we can observe the security strength of App #2 should be weaker than App #3.

Thus, under the same comparable security strength, the minimal padding will be App #3. We can also see that cluster 256 is smaller than cluster 512. This is because the smaller the cluster group is, the less padding will the approach need.

6 Inference Attacks against Order-preserving Encryption

Source code:

```
import random
import os
import operator
import csv
from itertools import cycle
import numpy as np
from scipy.optimize import linear_sum_assignment

def cumulative_attack(raw_data,ope_data):

    #Step 1: compute the histograms of list_raw_data and ope_data
    raw_data_hist = {}
    ope_data_hist = {}
    histo(raw_data, raw_data_hist)
    histo(ope_data, ope_data_hist)

    #Step 2: compute emperical CDFs of raw and ope data
    raw_data_keys = sorted(list(raw_data_hist.keys()))
    ope_data_keys = sorted(list(ope_data_hist.keys()))

    raw_data_cdf = cdf_histo(raw_data_keys, raw_data_hist)
    ope_data_cdf = cdf_histo(ope_data_keys, ope_data_hist)

    #Step 3_1: Compute the cost matrix C and then use LSAP
    C =   [[] for _ in range(len(ope_data_keys))]

    for i in range(len(ope_data_keys)):
        for j in range(len(raw_data_keys)):
            C[i].append((abs(ope_data_hist[ope_data_keys[i]] -
raw_data_hist[raw_data_keys[j]]))**2 + (abs(ope_data_cdf[ope_data_keys[i]] -
raw_data_cdf[raw_data_keys[j]]))**2)

    #Step 3_2: apply linear_sum_assignment to identify the weight matrix X
    row_ind, col_ind = linear_sum_assignment(C)
```

```
        #Step 4: export the mapping
    with open("map.txt", "w", newline='') as writer:
        writer.write("enc_age,raw_age\n")
        for index in range(len(raw_data_keys)):
            writer.write(str(ope_data_keys[index]) + "->" +
str(raw_data_keys[col_ind[index]])+ "\n")
    writer.close()
    return 0

def file_reader (reader, lst,index):

    for i, row in enumerate(reader): #i starts from 0
        if i!=0:
            lst.append(int(row[index]))

def histo (list_data, dic):

    for item in list_data: #i starts from 0
        if item in dic:
            dic[item] += 1
        else:
            dic[item] =1

def cdf_histo (sorted_list_data, dic):

    new_dic = dic.copy()

    for index in range(len(sorted_list_data)):
        if index != 0:
            new_dic[sorted_list_data[index]] +=
new_dic[sorted_list_data[index-1]]
    return new_dic

if __name__ == '__main__':
    #raw dict and enc dict
    raw_age= list()
    enc_age = list()
    #read the plaintext  file
    plaintext_file = open("auxiliary.csv", "r", newline='') #   covid_true
dataset.csv
    p_reader = csv.reader(plaintext_file,delimiter=',')
    file_reader(p_reader,raw_age,0)
    #read the cipher  file
    cipher_file = open("ope_enc_covid_age.csv", "r", newline='')
    c_reader = csv.reader(cipher_file,delimiter=',')
    file_reader(c_reader,enc_age,0)
    cumulative_attack(raw_age,enc_age)
```

Results:

enc_age,raw_age

0->0

1->1

11->2

18->3

22->4

39->5

46->6

52->7

56->8

62->9

63->10

69->11

73->12

77->13

79->14

93->15

95->16

100->17

102->18

104->19

107->20

117->21

120->22

141->23

149->24

150->25

153->26

159->27

162->28

163->29

168->30

172->31

186->32

191->33

193->34

206->35

207->36

208->37

209->38

211->39

212->40

217->41

226->42

230->43

232->44

238->45

255->46

259->47

263->48

264->49

270->50

271->51

275->52

276->53

277->54

```
278->55

284->56

286->57

293->58

295->59

297->60

300->61

310->62

313->63

315->64

317->65

327->66

332->67

337->68

339->69

345->70

353->71

356->72

357->73

359->74

365->75

369->76

378->77

383->78

387->79

394->80

402->81
```

407->82

409->83

410->84

414->85

424->86

428->87

430->88

431->89

432->90

434->91

438->92

447->93

459->94

466->95

468->96

To check this result, I sorted the 2 .csv files in ascending order in Excel and observe the matches. It’s evident that this result is correct.