STATISTICAL ANALYSIS OF LEAKAGE IN DUAL-COLUMN CRYPTOSYSTEM USING GRAPH MATCHING

Kang Ming Kee[1], Lee Tse Yee Megan[1], Lim Sze Min Joelle[2,3], Ruth Ng Ii-Yung[2]
[1]Raffles Girls' School (Secondary), 2 Braddell Rise, Singapore 318871
[2]DSO National Laboratories, 20 Science Park Dr, Singapore 118230
[3]A*STAR, 1 Fusionopolis Way, Singapore 138632

Abstract
We construct a new attack which addresses the Double-Column Full-Information Leakage Abuse Attack problem through the use of graph matching. We worked with synthetic SQL data, containing different types of data distributions and column correlations. We then implemented the attack in Python and analysed its effectiveness on the abovementioned synthetic data. We also provide a mathematical proof that the attack is optimal and corroborate this with our experimental results.

Introduction

First, we will be introducing the motivations for this project.

Big data is often stored online and there are many cloud data service providers which owners of big data engage in to store their data, such as Microsoft SQL Server, Amazon RDS etc. Sensitive data such as credit cards and personal information could be stored in such databases, and could be wanted by hackers. While random encryption is optimal to prevent hackers from easily accessing the data, it also makes it impossible to run queries such as joins on the data. Random encryption produces unique ciphertexts each time it is used. This means that the same plaintext, if repeated in the same database, can have multiple different ciphertexts. This makes it challenging to search on the encrypted data. In contrast, deterministic encryption produces stable ciphertext, meaning that encrypting a given plaintext will always return the same ciphertext. With deterministic encryption, rows can be compared to other rows to check if their values are equal, even without knowing the plaintext. This allows for searches, selections and joins to be performed more easily, which is really useful especially in large datasets. Any encryption mentioned after this point would be referring to deterministic encryption.

The condition that we are working with in this experiment is that somewhere in the server, there is an encrypted database, with information that is worth encrypting. People may choose to encrypt their data to facilitate easy access of large databases. For example, people's date of birth or salaries. An attack is performed on a leakage, which occurs when a client sends a query to the server with said encrypted data. In return, the server will provide the client with what they have requested for. This leakage can be accessed by two different adversaries - the eavesdropper adversary and the server adversary

In our project, we model the perspective of the server adversary, who has access to all frequency information from client queries and the encrypted database.

We run the attack in the scenario that the client sends a query for a join between two tables in the database. The client will then receive a joined table as a response. Our goal now as the server adversary is to guess the possible plaintexts in the database, by making use of the frequency vectors ($\vec{c}$ and $\vec{d}$) of the encrypted response, and the probability distributions ($\vec{a}$ and $\vec{b}$) of auxiliary data, which is on the internet and not encrypted.

Auxiliary data is data from an external source that should be representative of the encrypted data we have on our hands. Auxiliary data is sometimes used to supplement collected data. For example, it would not make sense for someone to compare the hourly wage of a fast-food joint (e.g. McDonalds) to the hourly wage of a bookshop (e.g.Times). It would be more reasonable to instead compare the hourly wages of two fast-food joints to each other, and two bookshops to each other, as shown in Figure 1 on the right.

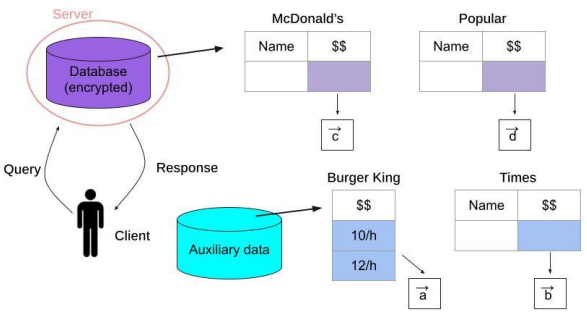


Figure 1: Diagram depicting how ciphertext and auxiliary vectors could be generated

For the server adversary to be able to guess the possible plaintexts of the encrypted database, it must make use of a Leakage Abuse Attack (LAA), inputting the four different vectors and receiving the possible plaintexts as an output.

Thus, performing a LAA is necessary to compare the data groups to find the most 'correct' answer (although there is no way to confirm it is 100% correct without actually having access to the site).

Materials and Methods - Two-column LAA

Two-column LAAs are attacks that are carried out on two deterministically encrypted columns. In our attacks, leakage is derived from ciphertext frequencies revealed by deterministic encryption.

The server adversary gets all the encrypted leakage from the encrypted database and the frequency information when queries are made.

In our project, we work with two columns and have four inputs. They consist of two auxiliary tuples called **a** and **b**, where **a** = $(a_1,....,a_n)$ and **b** = $(b_1,....,b_n)$ and two ciphertext tuples called **c** and **d**, where **c** = $(c_1,....,c_n)$ and **d** = $(d_1,...,d_n)$. Tuples a and b are probability distributions of the plaintext in the 1st and 2nd columns respectively. Tuples c and d contain positive integers, indicating the frequency distribution of the ciphertexts in the 1st and 2nd columns respectively.

The output function is the function f* where $f^* = \underset{f}{argmax} Pr[f]$ where

$$\Pr[f] = \left(a_1^{cf(1)} \cdot \ldots \cdot a_N^{cf(N)}\right) \cdot \left(b_1^{df(1)} \cdot \ldots \cdot b_N^{df(N)}\right) = \prod_{i=1}^{N} a_i^{cf(i)} b_i^{df(i)}$$

For the LAA to be "correct", it should return the matching (between the auxiliary vectors and ciphertext vectors) with the maximum probability.

A possible two-column LAA would be the trivial exhaustive search, or brute force attack, where we go through each and every possibility of the plaintext in the two encrypted columns. However, the process would take too long to complete, especially since there are two columns to compare. We use a more efficient method using Graph Matching and by reduction, the Hungarian Algorithm then experimentally verify that it works on simulated data.

Materials and Methods - Graph Matching

We work with balanced bipartite graphs, which are a set of graph vertices, decomposed into two disjoint sets of the same size, and vertices in the same set cannot be mapped to each other.

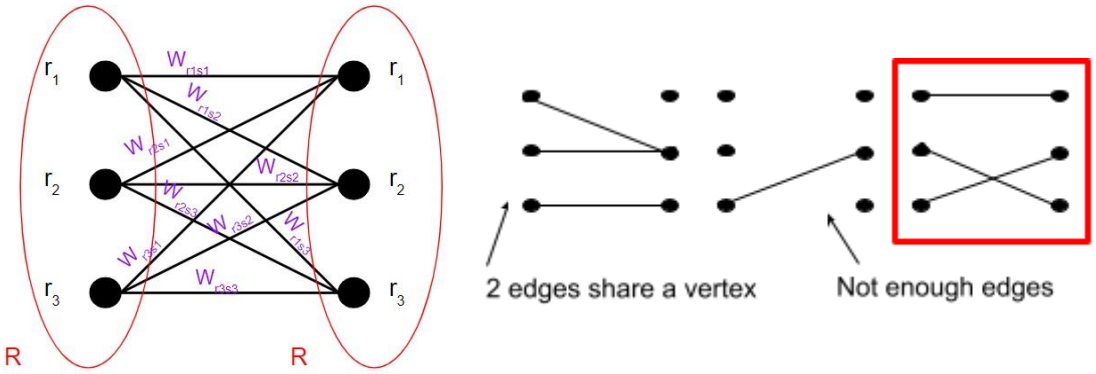


Figure 2a (Left): Diagram of complete bipartite graph with 3 vertices in both sides
Figure 2b (Right): Diagram depicting three possible subgraphs of the complete graph in Figure 2a

A complete graph is when all vertices in one set, R, are joined to all the vertices in the other set, R, where R = $\{r_1, r_2, \ldots, r_r\}$

We can also assign each edge a weight, which we can define as W_{ij}, where i and j are the (left and right respectively) vertices the edge joins.

A matching, M, is defined as a subset of the r^2 edges where no two edges in M share a vertex (the circle). If there are r edges in a matching, this is known as a complete matching. A maximal matching is a matching of maximum size (which is the maximum number of edges). If any other edge is added to a maximal matching, it is no longer a matching (as no two edges should share a vertex).

In Figure 2b, there are three subgraphs of the complete (3,3) bipartite graph as shown in Fig.2a The only complete matching is matching 3, as it matches the criteria:

1. no two edges share a vertex (mapping 1 shows how this is not applied), and
2. There are r edges in the set (mapping 2 shows how this is not applied)

Taking the last matching in Figure 3 for example, the total weight of this matching would be the sum of all the weights of all edges, i.e. ($W_{r1r1} + W_{r2r3} + W_{r3r2}$).

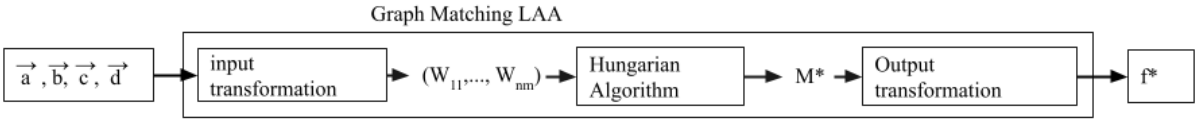
In our attack, we build our balanced bipartite graphs such that the vertices are the ciphertext frequency vectors and auxiliary probability distributions. As previously defined, both columns of ciphertexts would be matched to the auxiliary frequencies. Since both columns are deterministically encrypted with the same key, a matching between vector a and vector c would imply a matching between vector b and vector d. Thus, only one single mapping should be produced. This links to graph matching, where the vectors are the vertices and the matchings are the edges joining the vertices.

However, there is a limit to the number of possible matchings there can be. We can find this limit by taking the factorial **n! = n x (n-1) x (n-2) … x 1, where n is the number of edges in a complete matching**

Materials and Methods - Reduction to the Hungarian Algorithm

To solve the graph matching problem optimally, we make use of the Hungarian Algorithm, which is a known algorithm for graph matching. This method of reducing the original problem to a simpler form, such that we can use known algorithms to solve the problem is known as reduction. Making use of transformations, we transform the original inputs to ones that the algorithm we choose to use recognises. After the algorithm solves the problem, we then transform the output returned by the algorithm into a form that we want.

Here is the goal of input and output transformations:



The first transformation is the input transformation. We previously defined a weight as just W_{ij}. We will now define it as $W_{ij} = c_j \ln(a_i) + d_j \ln(b_i)$ to better allow the Hungarian Algorithm recognise this input, where c_j is the value of vector c, d_j the value of vector d, a_i the value of vector a, and b_i the value of vector b, within that edge.

However, given the nature of the logarithmic function when the value in the logarithmic function is less than one, using the original weight definition, $W_{ij} = c_j \ln(a_i) + d_j \ln(b_i)$, would give rise to a negative answer. In our graph matching problem, we would have to find the complete matching with maximum weight. This means that we choose the matching whose weight has the lowest absolute value, since -|-21| > -|-25|. Noting that the Hungarian Algorithm which finds the matching with the minimal weight, we negate the values in the weight to ensure that the new values are now positive, or the absolute value of the weight. Hence, the new weight would be $W_{ij} = -c_j \ln(a_i) - d_j \ln(b_i)$. The Hungarian Algorithm after receiving the negated inputs, will return M*, which is the minimum weight matching. The total weight of this matching can be calculated by taking the sum of all the weights of the edges in it.

Just as how we had an input transformation, we will also have an output transformation, which is the argmax f* of the sum of weights of each matching. Below is a proof of why our reduction works, and the resultant LAA is correct as defined.

Even if we take ln or log of a maximum term, it will still be a maximum term as ln or log is a monotonically increasing function. Thus, if x<y, ln(x)<ln (y).

Figure 3: Diagram of how the original f* equation changes after taking the natural logarithm of it

$$
\begin{aligned}
f^* &= \underset{f}{\arg max} \left(\prod_{i=1}^{N} a_i^{c_{f(i)}} b_i^{d_{f(i)}} \right) \\
&= \underset{f}{\arg max} \ln \left(\prod_{i=1}^{N} a_i^{c_{f(i)}} b_i^{d_{f(i)}} \right) \\
&= \underset{f}{\arg max} \left(\ln \left(a_i^{c_{f(i)}} \right) + \ln \left(b_i^{d_{f(i)}} \right) \ldots \ln \left(a_N^{c_{f(N)}} \right) + \ln \left(b_N^{d_{f(N)}} \right) \right) \\
&= \underset{f}{\arg \max} \sum_{i=1}^{N} \left(c_{f(i)} \ln (a_i) + d_{f(i)} \ln (b_i) \right)
\end{aligned}
$$

There is a function f where f(i) = Ki (output transformation → applied to M*). Then, the maximum over all M can be expressed as a maximum over all f. This thus proves that M* corresponds to f* and thus the best M* corresponds to the best f*

While using the Hungarian Algorithm, we also decided to learn more about the steps in the algorithm, in which Wij = cj ln(ai) + dj ln(bi) is an input to get the output of matching M*.

Given an n by n cost matrix:
1) Row reduction. Subtract the minimum value of each row from each value in the row.
2) Column reduction. Subtract the minimum value (can be = 0) of each column from each value in the column.
3) Test for optimal assignment. Find the minimum number of straight lines to cover all the zeros in our matrix. This is done by the following steps:
i) Construct graph: From the cost matrix that we are left with after row and column reduction, we identify the zeroes. We first build a bipartite graph from the cost matrix, where the vertices in the left set are representative of the rows, and those in the right set of the columns. We then sketch out the edges, where one edge represents a zero. Figure 5 below demonstrates this.

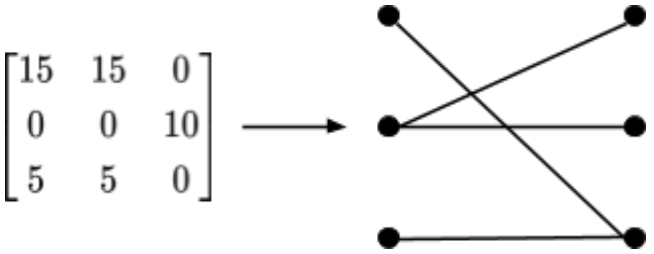


Figure 4: Diagram of how we can obtain a bipartite graph from a cost matrix

ii) Find any matching: To start, we have to find the maximal matching of the graph we built. A maximal matching is defined as a matching M of a graph G that is not a subset of any other matching. We used augmenting paths to find the maximal matching from the bipartite graph. An augmenting path is defined as an alternating path that starts from and ends on free (unmatched) vertices. An alternating path can be defined as a path that begins with an unmatched vertex and whose edges belong alternately to the matching and not to the matching. Below is a simple overview of how it works:
Given a graph, choose any matching and find an alternating path that starts with an unmatched vertex in the left set. If an alternating path ends with an unmatched vertex, current matching is not maximal matching. Add starting edge that isn't in matching to new matching and repeat step 1. If an alternating path ends with a vertex with an edge in the matching, maximal matching has been found.

iii) Get maximal matching using augmenting paths. If maximal matching is a complete matching, skip to step v. Else continue

iv) Get minimum vertex cover (using Konig theorem constructive proof). Below are parts to prove König's Theorem. After getting the maximal matching, we find the minimum vertex cover. A minimum vertex number is the minimum number of vertices selected to cover all the edges in the matching. It is defined as $K = (A \setminus Z) \cup (B \cap Z)$, where: A = set of vertices on the left, B = set of vertices on the right U = set of vertices in A without a matching edge, Z = set of vertices connected to U by an alternating path. Given that there exists a maximal matching M, where there are arbitrary values of vertices in both sets A and B, named ai and bj respectively.

1. Given an edge with vertices ai and bj, ai, bj are not in K and ai is in Z while bj not in Z. There is an alternating path to ai, ending with a matching edge. So, bj should be in Z. This is a contradiction to our earlier assumption of bj. Thus, we can conclude that every edge has a vertex in K, and K is a vertex cover.
2. The next step would be to prove that for every matching edge, only either the right or the left vertices will be in K. Given a matching edge with vertices ai and bj, using ai

and bj, we can draw an alternating path linking these two vectors to an unmatched vertex, U. See the diagram for this below. … This means that both ai and bj will be in Z. Which, according to the formula for K, proves that ai cannot be in K, while bj is in K. This proves that only one of the vertices connected by a matching edge can be a vertex cover.

3. Next, we prove that every vertex in K is touching a matching edge in M. U is the set of unmatched vertices in A. U is also in Z. A\Z has no unmatched vertices precisely because U is in Z or U is a subset of Z. As A\Z is a condition for vertex cover K, we can conclude that every vertex in K is touching a matching edge in M. Thus, #K=#M （2&3）
4. Here is proof that there is no vertex covering with fewer than #M vertices. This is because for each edge, there are two vertices. As #M = the number of edges, there will never be vertex covers with fewer than #M vertices.
5. We will notice that #K = #M. This is true because: Number of vertices in the minimum vertex covering ≥ number if edges in a maximal matching (proven by Step 4) Number of vertices in the minimum vertex covering ≤ number if edges in a maximal matching (proven by Step 2 and 3) Step 2 and 3 give you a vertex cover with at most #M number of vertices, minimum vertex cover cannot have >#M vertices since #M is the upper bound Thus, Konig's Theorem is true.

After step 3: Draw lines. If number of lines = n, skip to step 5. If number of lines < n, continue onto step 4

4) Shifting of zeroes. To do this, we must find the smallest uncovered value. We then subtract this value from all the uncovered values and add this value to the value in the intersection of the lines. Below is an example of shifting zeros in the matrix:

5) Making the final assignment. Each row and column should only have 1 value assigned to it. We can star the zeros in each row and column that meet this criteria. Afterward, we can look at the original matrix and the values in the positions with the starred zeros are the final assignment values. Do note that there could be more than 1 final assignment, but they would ultimately give us the same minimal cost assignment (found by taking the sum of all the assigned values).

Materials and Methods - Example

Here is a simple example of how our attack works. Given the following inputs:

auxiliary information Order of tuples: (1, 2, 3)	$\vec{a} = (0.4, 0.35, 0.25), \vec{b} = (0.8,\ 0.1,\ 0.1)$
ciphertext information Order of tuples: (aaa, bbb, ccc)	$\vec{c} = (3, 3, 2), \vec{d} = (4, 3, 2)$

Figure 5: Diagram of all possible matchings and their respective total weights

After constructing our graphs, we notice that we have 3 edges in each subset. Hence, the number of possible matchings between the vectors is 3x2x1=6. Note how in the figure, there are only **six** matchings as calculated

Here, the mapping with the minimal weight is the first one (top left). Thus, 1 → aaa, 2 —> bbb, 3 → ccc is the most "correct" mapping we can derive from our attack. In this scenario, we are able to check if our answer is correct, as an answer key is given, in which the correct matching is the one where 1 → aaa, 2 → ccc, 3 → bbb. This is different from the matching that we guessed, which had the highest probability of being correct. However in real-life

scenarios, similar conclusions cannot be made since we have no answer key to refer to, unless you actually decrypt the entire database.

Materials and Methods - Our Experiment

We tried looking for real-life data to carry out the attack on, but we were unable to find any that could fit the scenario we are working with. Thus, we ran our experiment on datasets that were randomly generated by our mentor. These datasets are each derived from a probability distribution and are named as : "DF_1000000_Number of ciphertexts_distribution of col0-distribution of col1{run_number}_col{0/1}", where DF refers to Double-Column Full Information (i.e. having full frequency information of the ciphertexts), number of ciphertexts = [10, 50, 100, 200, 500], distribution-distribution = ["lin-lin", "lin-slin", "lin-invlin", "lin-randlin", "lin-zipf", "zipf-zipf", "zipf-randzipf", "zipf-invzipf"]

Linear (lin) distribution occurs when probabilities increase with a constant gradient. Slow linear (slin) occurs when probabilities increase with a smaller constant gradient. Inverse linear (invlin) and zipfian (invzipf) refer to probabilities that are opposite of linear and zipfian. Random linear (randlin) and zipfian (randzipf) just refer to probabilities that have the same values as their original counterparts, but are ordered randomly. For a clearer representation, you can refer to Figure 7a and 7b under Discussions.

We take column 0 and column 1 as our dual-column cryptosystem, generating from the probability distributions in the CSVs the auxiliary data by introducing noise. This is done by introducing percentage errors (p), such as p% less or p% more to the already existing probability distributions and choosing a random number which lies between the two new values of the original probability value. After this, we scale down the tuple of entries to give us a new tuple with the total sum being 1, since they are meant to be probabilities. This process can be seen in Fig 1 in the appendix.

Following that we generate a cost matrix, which will then be inputs placed into the variant Hungarian Algorithm, where the matching will then be returned, and placed into our functions that calculate our success rates.

There are two ways to do so, using the 'v-score' (values score) and the 'r-score' (rows score). The former refers to the percentage of ciphertext values mapped correctly. The latter refers to the percentage of ciphertext rows guessed correctly. Since we are working with a joined table, the number of ciphertext rows would be n•m rows if the two ciphertext tables have n and m rows respectively.

We ran the code for all datasets, introducing p = 5, 10, 20%. For every different percentage of error introduced, we ran each dataset 10 times as denoted by the run number in the name of the CSV file, and took the average of the ten scores generated to have more accurate and reliable results. Below are graphs plotted using our results. For full results, you may refer to our appendix.

Results

Figure 6a: Graph showing r-scores of different distributions and different number of ciphertexts when error introduced = 5%

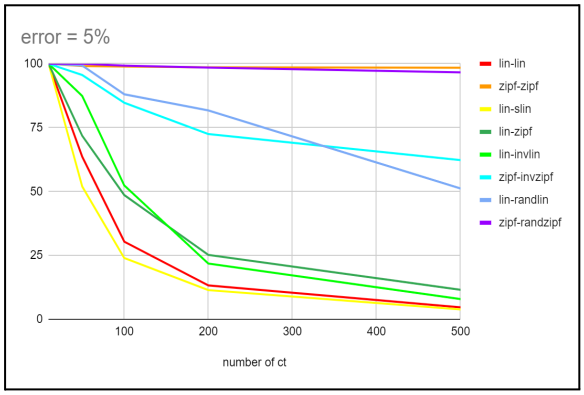


Figure 6b: Graph showing r-scores of different distributions and different number of ciphertexts when error introduced = 10%

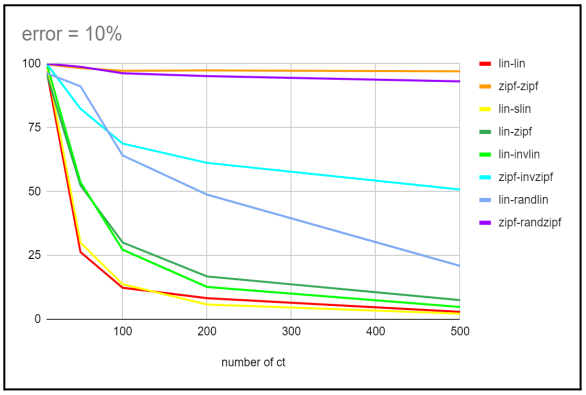


Figure 6c: Graph showing r-scores of different distributions and different number of ciphertexts when error introduced = 20%

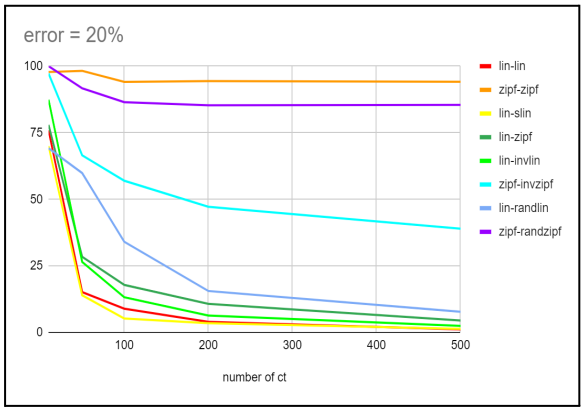


Figure 6d: Graph comparing v-score and r-score across percentage introduced and distribution of data

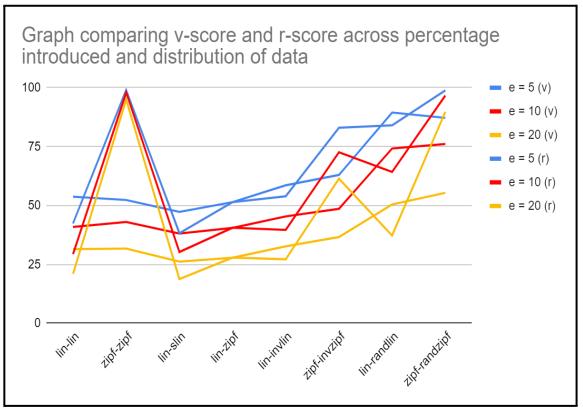


Figure 6e: Graph comparing the average r-scores of each type of error

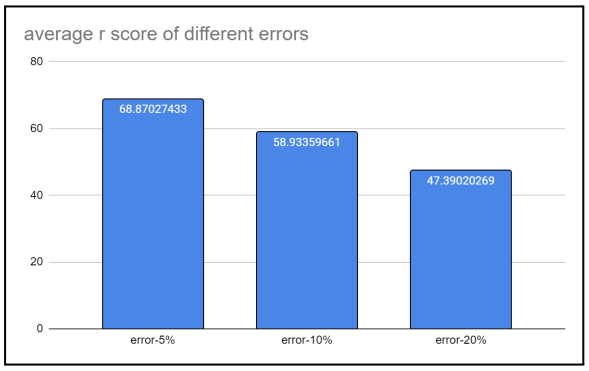
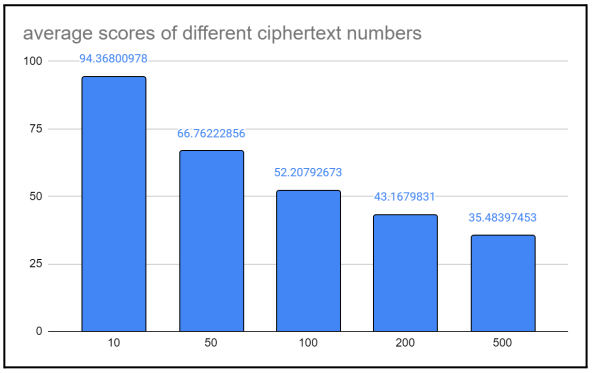


Figure 6f: Graph comparing the average r-scores of datasets with different ciphertext numbers



Here are some of the trends we observed from the graphs of our results:

1) In Figure 6a, 6b and 6c, for errors 5%, 10% and 20% respectively, it can be observed that the best performing dataset was the one where both columns had a zipfian distribution (zipf-zipf, orange). On the other hand, the worst performing distribution

was when column 0 had the linear distribution, and column 1 had the slow linear distribution (lin-slin, yellow).

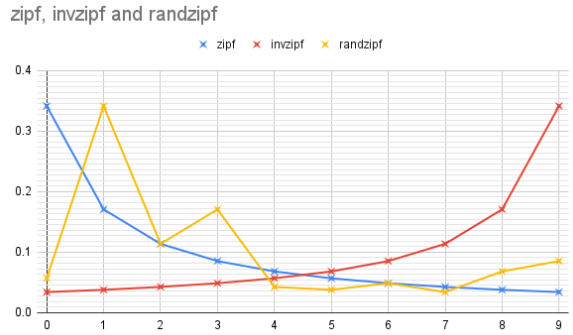
2) In Figure 6d, it is observed that as the percentage of error introduced increased, the scores decreased
3) In Figure 6f, it is observed that as the percentage of error introduced increased, the scores decreased

Discussion

We will make the following hypotheses on why each trend is true.

Addressing the first trend, we will start by discussing the possible reasons behind the better performance of the dataset where both columns have a Zipfian distribution, and the worst performance of lin-slin. Below are some graphs of the distribution.

Figure 7a: Graph showing Zipfian distributions Figure 7b: Graph showing linear distributions



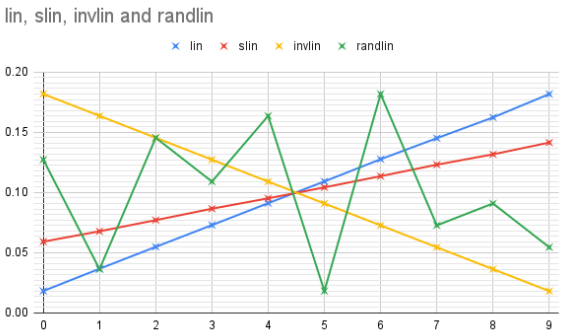


Figure 7a shows different Zipfian distributions. Looking at the blue graph (which shows a standard Zipfian distribution) It can be observed that the first few ciphertexts in a standard Zipfian distribution have a substantially larger number of rows compared to the rest (. Thus if the first few ciphertexts are already mapped correctly, this causes the number of correct rows to be very high.

The first few ciphertexts in Figure 7b (linear distribution) however, have a lower number of rows compared to the standard Zipfian distribution. Thus, if only the first few ciphertexts are mapped correctly the number of correctly matched rows is significantly lower.

Addressing the second trend, our reasoning behind why the r-score decreases as the percentage of error introduced increases. This is because the higher the percentage of error (which means more noise introduced), the greater the extent of deviation of the auxiliary frequency from the probability distribution. This gives rise to a higher probability that the ciphertexts are mapped incorrectly, thus decreasing the r-score.

Lastly, **addressing the third trend**, our explanation as to why the r-score decreases as the number of ciphertexts increases. As the number of ciphertexts increases, this causes the differences between the probabilities in the probability distribution and hence ciphertext frequencies to be smaller. This also means that there could be instances where the auxiliary probabilities could be larger or smaller than the original probability, to the extent that they become representative of a larger or smaller probability in the original probability distribution. For example, we have a1 and a2, and c1 and c2, both generated from p1 and p2. Because a1 and a2 are generated by introducing errors, due to the smaller difference between

p1 and p2 and hence c1 and c2, a1 might be representative of p2 rather than p1. Thus, assuming that those with the same numbers should be matched to each other, a1 will then be mapped to c2, which is wrong.

The implication of this is that it is more probable that the guessed matchings are wrong, and more ciphertexts are mapped incorrectly, causing the r-score to decrease.

Acknowledgements
We would like to express our deepest appreciation to our mentors, Dr Ruth and Ms Joelle who have given us much guidance over the course of the project.
We would also like to express our sincerest gratitude to the team/lab who came up with all the different cases and generated this project of ours.

References

[1] J. A. Bondy, U. S. R. Murty. 1976. Theorem 5.3, p74-75 In: Graph Theory With Applications. Elseyier Science Publishing Co., Inc. U.S.A.
https://www.zib.de/groetschel/teaching/WS1314/BondyMurtyGTWA.pdf Theorem 5.3