Cryptanalysis of SQL joins in Encrypted Databases under Varying Leakage Profiles

Allison Law Li Xuan[1], Naomi Wang Chencheng[2], Ruth Ng Ii-yung[3,4], Ng Wei Cheng[3]
[1]Hwa Chong Institution, 661 Bukit Timah Rd, Singapore 269734
[2]Raffles Institution (Junior College), 1 Raffles Institution Ln, Singapore 575954
[3]DSO National Laboratories, 12 Science Park Dr, Singapore 118225
[4]A*STAR, 1 Fusionopolis Way, Singapore 138632

Abstract

Our work presents new leakage abuse attacks (LAA) which address the Double-Column Full-Information (DF) problem using graph matching, and the Double-Column Incomplete-Information (DI) problem using graph matching and data partitioning. We implemented the attack using Python, and analysed its effectiveness on synthetic data. In doing so, we make use of different distributions (such as uniform, linear, and Zipfian) for partitioning in the DI case, and analyse the effectiveness of data recovery for both DF-LAA and DI-LAA.

Introduction

Modern cloud SQL (structured query language) database applications are encrypted using "leakage-based" encryption schemes, whose security depends on the amount of data leakage adversaries have access to. One such scheme is the deterministic encryption scheme, which is defined by a key, an encryption algorithm that encrypts a plaintext input with the key to produce a ciphertext, and a deterministic decryption algorithm that decrypts the ciphertext with the same key and returns the exact same plaintext [1]. A crucial reason why the deterministic encryption scheme is favoured in the encryption of databases as compared to standard encryption schemes, is that for all keys and plaintexts, the output plaintext from the decryption algorithm will match the input plaintext to the encryption algorithm. This makes it easier for users to perform searches and queries on encrypted databases. However, due to the nature of such encryption schemes, information such as the frequency and distribution of plaintexts in the database will be leaked to an adversary [2]. With access to leakage, adversaries can conduct leakage abuse attacks (LAAs) and statistically deduce the plaintext values of the dataset [3].

Leakage data can be derived from network traffic, where the client or user makes a query on columns in an SQL database, and the server returns a result (Fig. 1). An example of a query is the JOIN query, which can be performed on a pair of columns. The JOIN query compares the two columns and selects records that have matching ciphertext values in the two columns. The returned output would then be a table that reflects the distribution of common ciphertexts between the two columns.[4]

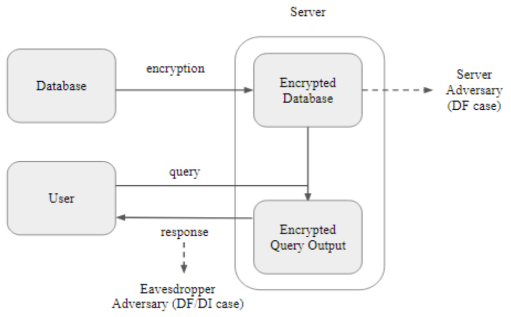


Fig. 1: A simplified diagram illustrating the network traffic between the user and server

Two types of adversaries are taken into consideration in this work: the server adversary and the eavesdropper adversary, which are caused by the Double-Column Full-Information (DF) case and the Double-Column Incomplete-Information (DI) case, respectively. In the DF case, the server adversary is able to view the full encrypted database on the server, resulting in full leakage. This is unlike the DI case, where the eavesdropper adversary intercepts the response by the server and thus only gains access to the returned output of the query made. Given the nature of a JOIN query, ciphertexts that do not appear in both columns of the encrypted database will not be returned in the output. This results in incomplete leakage, as the adversary is unable to gain access to the full distribution of ciphertexts.

Our project models LAAs carried out by the server adversary and eavesdropper adversary: DF-LAA and DI-LAA respectively, given the adversaries have complete or incomplete access to the ciphertext data. We aim to achieve the maximum success rate in attacking the double-column encrypted database for both the DF and DI cases, as well as expose the vulnerabilities of the existing encryption scheme by exploring the extent of which leakage data can be exploited by an adversary in the deduction of plaintext values.

Materials and Methods

In a real-world scenario, the LAA may be used in attacks on databases with two columns deterministically encrypted with the same key, for example, a school database containing the examination results of two classes or a database with medical records from two hospitals. There are two types of inputs to the LAA: the ciphertext frequency vector, which represents the frequency distribution of the ciphertext in a column, and the auxiliary frequency vector, which provides information about plaintext distribution in the same column and thus has a similar distribution as the ciphertext frequency vector [5].

To obtain ciphertext data, we used an existing synthetically generated double-column dataset of 1,000,000 rows with different possible numbers of distinct ciphertexts from the set {10, 50, 100, 200, 500} in each dataset (Refer to Appendix A). These datasets follow different distributions for the two columns, including same distributions (lin-lin, zipf-zipf), correlated distributions (lin-slin, lin-zipf), inversely correlated distributions (lin-invlin, zipf-invzipf) and independent distributions (lin-randlin, zipf-randzipf). Do note that lin refers to linear, zipf refers to Zipfian, slin refers to slow linear, inv refers to inverse, and rand refers to random (details of dataset distribution in Appendix B). In total, we conducted 10 runs for each dataset. As many naturally observable collections of data follow linear and Zipfian distributions, these distributions were chosen in order to model our synthetic data after real world data.[6] The datasets used for the DF-LAA and the DI-LAA are distinct as they are based on different scenarios, but the dataset distributions are identical.

In a realistic scenario, an adversary would have easy access to auxiliary data as they are usually sourced from open source databases available on the Internet. However, since our ciphertext data is synthetically generated, we would need to generate our own synthetic auxiliary data. To do so, we designed an algorithm as follows: The algorithm takes as input the ciphertext frequency vector of integers $\vec{x} = (x_1 ,..., x_n)$, where n is any integer. The resultant output of the algorithm is a vector of non-negative real numbers $\vec{y} = (y_1,..., y_n)$, where $y_1 + y_2 + ... + y_n = 1$, with an approximately identical distribution as the input vector, which we take to be the auxiliary frequency vector. To achieve this, an error percentage from the following set of errors {5%, 10%, 20%} will be introduced into $\vec{x}$, before it is scaled to obtain $\vec{y}$, which sums up to 1. (details in Appendix C). Our auxiliary data is a good representation of realistic auxiliary data as it is sufficiently correlated to the ciphertext data generated.

DF-LAA (Full leakage):
In the DF case, it is assumed that all the data in both columns of the encrypted database is returned in the JOIN output, and that the ciphertext frequencies are complete.

The inputs to the DF-LAA would then consist of 2 auxiliary frequency vectors, $\{\vec{a} = (a_1, a_2, a_3, \ldots, a_n), \vec{b} = (b_1, b_2, b_3, \ldots, b_n)\}$, and 2 ciphertext frequency vectors, $\{\vec{c} = (c_1, c_2, c_3, \ldots, c_n), \vec{d} = (d_1, d_2, d_3, \ldots, d_n)\}$, where $\vec{c}$ is obtained from Column 1 of the database, $\vec{d}$ is obtained from Column 2. The same ciphertexts in both columns are associated to the same plaintexts. The output of the LAA is a function $f^*: [n] \to [n]$, where $f^* = \underset{f:[n]\to[n]}{argmax}\, Pr[f]$, where $Pr[f] = \prod_{i=1}^{n} a_i^{c_{f(i)}} \prod_{i=1}^{n} b_i^{d_{f(i)}}$. This expression allows us to find the optimal mapping of ciphertext frequency vectors to auxiliary frequency vectors, i.e. the mapping which outputs the maximum probability $Pr[f]$.

We approach the DF case using a graph matching method, which consists of the following steps: input transformation, Hungarian Algorithm, and output transformation. In input transformation, the auxiliary frequency vectors $\vec{a}$ and $\vec{b}$ are converted into their respective probability distribution vectors. As for $\vec{c}$ and $\vec{d}$, they will be expressed in terms of the ciphertexts' frequencies, which are represented by whole numbers.

Next, we will use the Hungarian Algorithm [7], a technique that finds the maximum weight matching of a bipartite graph. To facilitate the use of this algorithm, we represent our problem in the form of a bipartite graph as follows: During mapping, $\vec{a}$, $\vec{b}$ form a (n,n) - bipartite graph with $\vec{c}$, $\vec{d}$, where n refers to the number of vertices on each group (Fig 2). The edge between (a_i, b_i) and (c_j, d_j) is denoted as e_{ij}.

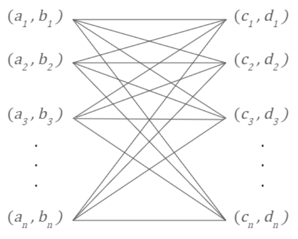


Fig 2: Overview of a (n,n) - bipartite graph

Each edge between the vertices (n,n) is associated with a weight w_{ij}, which is applied to all possible edges between (n,n) vertices. For ease of using the Hungarian Algorithm, the weights are expressed in linear form. This is done by taking the logarithm of $Pr[f]$ (Fig 3). As such, the weight is given by:

$$w_{ij} = c_j \ln(a_i) + d_j \ln(b_i).$$

Each matching is a subset of the edges, where no edges share a common vertex (example in Appendix D) [8]. A matching in the bipartite graph in Fig. 2 would involve n edges. The maximum weighted matching is then found using the Hungarian Algorithm as follows:

$$M^* = \underset{M}{argmax} \sum_{e_{ij} \in M} w_{ij} \text{ where each matching } M = \{e_{1j_1},\ e_{2j_2},\ e_{3j_3},\ \dots,\ e_{nj_n}\}$$

To achieve this in code, we import the linear sum assignment problem (LSAP) function from the Scipy library. In the input, we include all the different weights in the form of a *(n×n)* cost matrix. The LSAP function matches each row of the cost matrix to a different column in such a way that the sum of the corresponding entries is maximised. Each row is assigned to at most one column, and each column is assigned to at most one row. The function then returns the corresponding column and row indices, giving the optimal assignment (the matching that outputs the maximum sum of weights).

In output transformation, the output of the Hungarian Algorithm can be translated into the DF-LAA output, returning the optimal mapping. The DF-LAA output will be optimal (Fig 3) as the logarithmic function applied on *Pr*[*f*] during weight calculation is a monotone increasing function, so for any *x*,*y*, if *x* < *y*, then ln(*x*) < ln(*y*). Besides, each matching $M = \{e_{1j_1},\ e_{2j_2},\ e_{3j_3},\ \dots,\ e_{nj_n}\}$ corresponds to *f* where $f(k) = j_k$. Thus, the maximum over all *M* is also the maximum over all of *f*, i.e. *M** = *f**. This means that maximising the weighted matching using the Hungarian algorithm would also maximise the probability of the matching for the DF-LAA.

$$\begin{aligned}
f^* &= \underset{f:[n]\to[n]}{argmax} Pr[f] \\
&= \underset{f:[n]\to[n]}{argmax} ln(Pr[f]) \ (\because ln \text{ is a monotonic function}) \\
&= \underset{f:[n]\to[n]}{argmax} ln(\prod_{i=1}^{n} a_i^{c_{f(i)}} \prod_{i=1}^{n} b_i^{d_{f(i)}}) \\
&= \underset{f:[n]\to[n]}{argmax} \sum_{i=1}^{N} (ln(a_i^{c_{f(i)}}) + ln(b_i^{d_{f(i)}})) \\
&= \underset{f:[n]\to[n]}{argmax} \sum_{i=1}^{N} (c_{f(i)} ln(a_i) + d_{f(i)} ln(b_i))
\end{aligned}$$

Fig 3: proof that the DF-LAA output is optimal

DI-LAA(Incomplete leakage):

The input to DI-LAA is largely similar to that of DF-LAA, but with a few differences. For DI-LAA, the length of auxiliary frequency vectors $\vec{a}$ and $\vec{b}$ remains the same at *n*. However, ciphertext frequency vectors $\vec{c}$ and $\vec{d}$ would only contain *m* distinct ciphertexts that appear in both columns of the database and are hence returned in the JOIN output, where $m \le n$. Assuming that we know the total number of rows in both columns, all the other ciphertext frequencies that are not in the returned output are grouped as one frequency and represented by c_o and d_o for Columns 1 and 2 respectively. The input to DI-LAA would then be $\vec{a} = (a_1, a_2, \dots, a_n)$, $\vec{b} = (b_1, b_2, \dots, b_n)$, $\vec{c} = (c_0, c_1, \dots, c_m)$, $\vec{d} = (d_0, d_1, \dots, d_m)$, where the length of $\vec{c}$ and $\vec{d}$ is $(m + 1)$.

During mapping, auxiliary frequency vectors $(\vec{a}, \vec{b})$ form a (*n*,*m+2*) bipartite graph (Fig 4) with ciphertext frequency vectors $(\vec{c}, \vec{d})$. In the DI case, a valid output mapping would consist of the following (example in Appendix D):

- *m* edges between *m* (a_i, b_i) pairs and all the (c_j, d_j) pairs where $i \in [n], j \in [m]$, and no edges share a vertex
- $(n - m)$ edges between the remaining $(n - m)$ (a_i, b_i) pairs and the c_0 and d_0 vertices, where each (a_i, b_i) pair is matched to only either c_0 or d_0

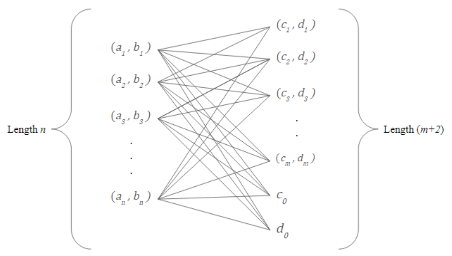


Fig 4: Overview of a (n,m+2) - bipartite graph

The probability of the mapping f can be computed by the following equation:

$$Pr[f] = \prod_{i=1}^{m} a_{f^{-1}(i)}^{\;c_i} \prod_{i=1}^{m} b_{f^{-1}(i)}^{\;d_i} \Big(\sum_{i \in f^{-1}(m+1)} a_i \Big)^{c_0} \Big(\sum_{i \in f^{-1}(m+2)} b_i \Big)^{d_0}$$

However, computing this function for every possible mapping is time consuming given its exponential time complexity, especially when the values of the exponents are large. Hence, we will be using the DF-LAA algorithm as a solution to the DI case.

Before we can run the DI-LAA, c_0 and d_0 need to be split up into individual nodes, $\{c_{m+1}, \ldots, c_n\}$ and $\{d_{m+1}, \ldots, d_n\}$ respectively (Fig 5). This can be done by applying a distribution function (e.g. uniform distribution) to c_0 and d_0 respectively. The splitting should be done in a way such that $\sum_{i=m+1}^{n} c_i = c_0$ and $\sum_{i=m+1}^{n} d_i = d_0$ respectively. Additionally, since the corresponding ciphertext is not returned in the JOIN output, one of the nodes in each (c_i, d_i) pair where $i \in \{m+1, \ldots, n\}$ needs to be zero while the other is non-zero.

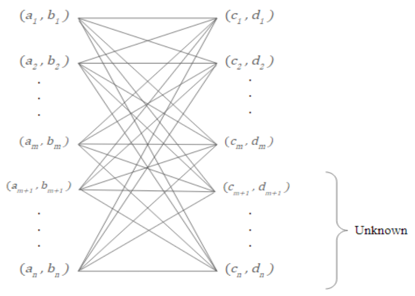


Fig 5: Overview of an (n,n) - bipartite graph after the splitting of c_0 and d_0

It is worth noting that while the graph matching method for the DF case is an optimal attack in theory, the DI case can only be solved by heuristic attacks as it is hard to split the c_0 and d_0 buckets in a way that is guaranteed to find the matching with maximum probability other than brute force. Nevertheless, our experimental results show that the DI-LAA results are comparable to those of DF-LAA, even though it is an approximation of the optimal attack.
The splitting of c_0 and d_0 is done via a 2-step process:

1. Generate a set of vectors of guessed ciphertext frequencies, $\{\vec{e}_1, \vec{e}_2, \ldots, \vec{e}_l\}$, each following a particular probability distribution.
2. Assign the corresponding ciphertext to either Column 1 or 2 of the double-column database for each $\vec{e}_k$ where $k \in \{1, 2, \ldots, l\}$

Now, we can define our DI-LAA as follows:

$$DI - LAA(\vec{a}, \vec{b}, \vec{c}, \vec{d}) = \underset{k \in \{1, \ldots, l\}}{argmax} Pr[f_k], \quad f_k = DI - LAA'(\vec{a}, \vec{b}, \vec{c}, \vec{d}, \vec{e_k})$$

where DI-LAA' is a version of DI-LAA with a new input $\vec{e_k}$ for all $\vec{e_k}$ in $\{\vec{e_1}, \vec{e_2}, \ldots, \vec{e_l}\}$.

For Step 1 of the splitting, we generate a probability distribution vector of unknown values, $\vec{e_k} = (e_{k_1}, e_{k_2}, e_{k_3}, \ldots, e_{k_{n-m}})$, where $\sum_{i=1}^{n-m} e_{k_i} = 1$. $\vec{e_k}$ represents the guessed ciphertext frequencies of $(n - m)$ ciphertexts not returned in the JOIN output, given that their frequencies are unknown to the adversary. In our project, we will be experimenting with three partitioning distributions: uniform, linear and Zipfian, in the generation of $\vec{e_k}$, as these partitionings largely match the underlying dataset distributions. Taking $l = 3$, we let $\vec{e_1}$, $\vec{e_2}$, and $\vec{e_3}$ follow a uniform, linear and Zipfian partitioning respectively (details in Appendix E). To generate $\vec{e_1}$, we let all the values in $\vec{e_1}$ be the same, given the condition that the values sum up to 1. $\vec{e_2}$ is obtained by first plotting a $y = x$ graph. We then obtain the y-coordinates of $(n - m)$ evenly spaced points from the graph of domain $[1, n - m]$, before scaling them accordingly. The Zipfian distribution is governed by the following relationship: For integers $x \geq 1$, $Z(x) = \frac{x^{-1}}{H_x}$, where H_x is the generalised harmonic number, $H_x = \sum_{n=1}^{x} \frac{1}{x}$. To obtain $\vec{e_3}$, we generate a series of values proportional to the first $(n - m)$ numbers of the Zipfian distribution, before scaling the values accordingly.

Having obtained $\vec{e_1}$, $\vec{e_2}$, and $\vec{e_3}$, the subsequent steps until the end of the attack would be repeated for each $\vec{e_k}$, $k \in \{1, 2, 3\}$:

In Step 2 of the splitting, we will assign the corresponding ciphertexts in $\vec{e_k}$ to either Column 1 or 2. This is achieved using a partition optimisation algorithm that splits $\vec{e_k}$ into 2 groups, *R* and *S*. *R* represents the ciphertexts that appear in Column 1 of the encrypted database, while *S* refers to the ciphertexts that only appear in Column 2. While the partition optimisation algorithm does not appear in literature, it is solvable via a dynamic programming approach (details in Appendix F). The algorithm takes in one vector, $\vec{e_k}$ as well as two integers, c_0 and d_0 as input. The output partitions *R*,*S* of $[n - m]$ where:

$$(R, S) = \underset{(R,S)\ partition\ of\ [n-m]}{argmax} \left(\sum_{i \in R} e_{k_i}\right)^{c_0} \left(\sum_{i \in S} e_{k_i}\right)^{d_0}$$

Having split c_0 and d_0 into their individual nodes, we can generate the updated ciphertext frequency vectors $\vec{c'_k}$ and $\vec{d'_k}$ of length *n* so that the ciphertext frequency vectors are of the same length as the auxiliary frequency vectors.

First, defining $\lceil x \rfloor$ as rounding *x* to the nearest integer, we let:

$$\{x_{k_1}, x_{k_2}, \ldots, x_{k_p}\} = \{\lceil (c_0 + d_0)\, e_{k_i} \rfloor\}_{i \in R}, \quad \{y_{k_1}, y_{k_2}, \ldots, y_{k_q}\} = \{\lceil (c_0 + d_0)\, e_{k_i} \rfloor\}_{i \in S}.$$

Note that $p + q = n - m$.

We next define $\vec{c'}_k = (c'_{k_1}, c'_{k_2}, \ldots, c'_{k_n})$ and $\vec{d'}_k = (d'_{k_1}, d'_{k_2}, \ldots, d'_{k_n})$, where:

$$c'_{k_i} = \begin{cases} c_i \text{ for } i \in [m] \\ x_{k_{i-m}} \text{ for } i \in [m+1, \ldots, m+p] \\ 0 \text{ otherwise} \end{cases} \qquad d'_{k_i} = \begin{cases} d_i \text{ for } i \in [m] \\ 0 \text{ for } i \in [m+1, \ldots, m+p] \\ y_{k_{i-m-p}} \text{ otherwise} \end{cases}$$

We are now ready to solve the DI case by running DF-LAA with $\vec{a}, \vec{b}, \vec{c'}_k$ and $\vec{d'}_k$ as inputs to the attack. Thus, let $g_k = DF - LAA(\vec{a}, \vec{b}, \vec{c'}_k, \vec{d'}_k)$, with f_k values as follows:

$$f_k(i) = \begin{cases} g_k(i) \text{ if } g_k(i) \in [m] \\ m+1 \text{ if } g_k(i) \in [m+1, \ldots, m+p] \\ m+2 \text{ otherwise} \end{cases}$$

After repeating the steps above for $\vec{e}_1$, $\vec{e}_2$, and $\vec{e}_3$ separately, we obtain f_1, f_2, and f_3, from which we find the mapping that returns maximum probability, $f^* = \underset{k \in \{1,2,3\}}{argmax}\, Pr[f_k]$.

<u>Results</u>

To evaluate the success of our attack, we compare our derived mapping to the actual mapping using two scores: r-score and v-score, where r-score calculates the percentage of rows that are guessed correctly by the adversary, while v-score refers to the percentage of distinct ciphertext values guessed accurately by the adversary.

We observed trends for the following relationships: 1. Effectiveness of DF-LAA and DI-LAA, 2. Score vs error, 3. Score vs number of distinct ciphertexts, 4. Score vs distribution type in dataset, 5. Score vs partitioning (individual dataset distributions), 6. Score vs partitioning (overall). We observed results for both DF-LAA and DI-LAA, but only included results for the DI case. One example for each trend is included in the body while raw data is included in Appendix G.

<u>Result 1:</u>

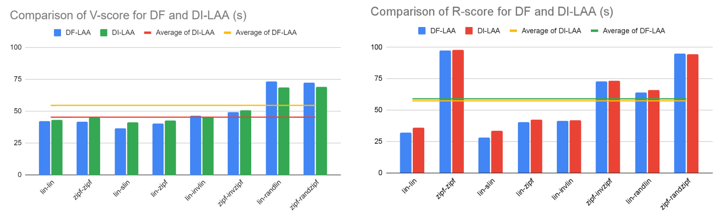


Fig 6: Comparison of v-score and r-score between the DF-LAA and DI-LAA attacks

By observing each column pair in Fig 6, we can see that both the DF-LAA and DI-LAA attacks share similar v-scores and r-scores separately across the different dataset distributions. The average v-score is 50.3 and 45.3, while the average r-score is 58.8 and 57.2, for DF-LAA and DI-LAA respectively.

<u>Result 2:</u>

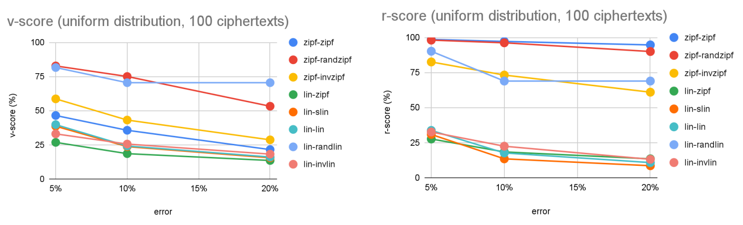


Fig 7: How magnitude of error affects accuracy of mappings

As seen from Fig 7, an increase in error percentage leads to a decrease in v-score and r-score respectively.

<u>Result 3:</u>

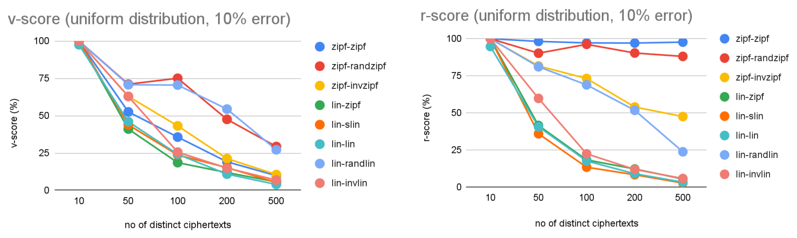


Fig 8: How the number of ciphertexts affects the accuracy of mappings

Fig 8 illustrates a general decrease in v-score and r-score with an increasing number of distinct ciphertexts.

<u>Result 4:</u>

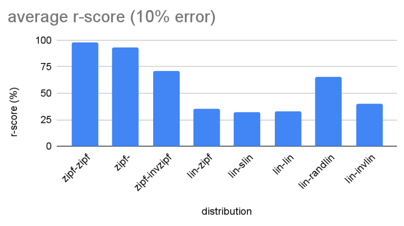


Fig 9: Average r-score across different underlying dataset distributions

From Fig 9, it can be observed that datasets following Zipfian distribution generally produce higher r-scores than those following linear distributions.

<u>Result 5:</u>

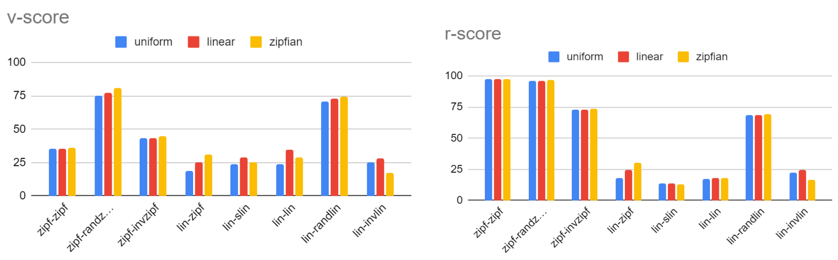


Fig 10: Average r- and v-score across different partitionings for each dataset distribution

From Fig 10, we notice that linear partitioning has the highest average r-score and v-score for datasets that are distributed in a linear fashion. This is similarly observed for Zipfian partitioning, which produces the highest average r-score and v-score for datasets following a Zipfian distribution.

Result 6:
Moving on, we analysed the results generated by using different partitionings. We recorded the partitioning that produces the highest r-score for each csv file and error, i.e. the "best partitioning", and tabulated the "best partitioning" count for each partitioning used.

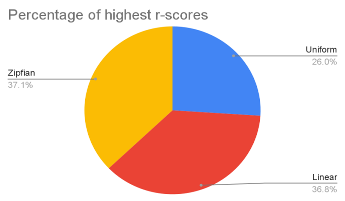


Fig 11: Percentage of times each partitioning achieved the highest r-score

Fig 11 shows that among the highest r-scores achieved for each csv file and error combination, zipfian achieved it the most frequently (37.1%), followed by linear (36.8%) and lastly by uniform (26.0%).

Discussion
We proceeded to analyse the trends from Results 1 to 6.

Result 1 shows that both the DI-LAA and DF-LAA attacks can accurately decipher >50% of rows in a dataset, proving that both attacks are successful. Consider an attack performed on a real life dataset containing millions of rows: a significant number of rows in the dataset would have been revealed to the adversary. Additionally, the results of the DI-LAA attack are comparable to those of the optimal approach of using graph matching to solve the DF case, with the difference in v-score and r-score falling within ± 5% of that of DF-LAA. We thus argue that our DI-LAA is as effective as the DF-LAA, and that the DI-LAA is considered an effective attack.

Result 2, on the other hand, shows that when the percentage of error introduced in the generation of auxiliary data increases, the v-score and r-score decrease. This can be explained as the auxiliary frequency vector experiences greater deviation from the original ciphertext distribution in the double-column. As such, it becomes less accurate in reflecting the plaintext distribution in the database, and thus the accuracy of mapping from the auxiliary data to the ciphertext data decreases.

Result 3 shows that an increase in the number of distinct ciphertexts leads to a decrease in both v-score and r-score. This is likely due to a smaller difference between the ciphertext frequencies as the total number of rows in the database remains constant. As a result, the introduction of error into the auxiliary frequency vector becomes more significant as there is a higher chance that $a_i > a_j$ but $c_{f(i)} < c_{f(j)}$ or vice versa for $i, j \in \{1, \ldots, n\}$. This results in a higher likelihood of incorrect mapping of rows and ciphertexts.

Result 4 illustrates the trend that datasets following Zipfian distributions tend to have a significantly higher r-score than other distributions. This is caused by a greater difference in value between ciphertext frequencies, for ciphertext frequencies with larger values in datasets following a Zipfian distribution (refer to Appendix B). This results in a greater chance of achieving a correct mapping of ciphertext to plaintext values. Since these ciphertexts with large ciphertext frequencies contribute to a large percentage of rows of ciphertexts accurately mapped to their corresponding plaintexts, this leads to a higher r-score.

Result 5 shows that the attack works best when the partitioning distribution matches the underlying distribution, for example, the highest score is produced when a linear partition is used on datasets following a linear distribution. This can be attributed to a few reasons. First, c_0 and d_0 are more likely to be linearly distributed across the runs, since we aggregate the scores over 10 runs for each dataset, and the values in c_0 and d_0 are selected at random across the linearly distributed points in $\vec{e}$. Additionally, given that the underlying distribution of the dataset matches how the data is partitioned, the partitioned data is more likely to resemble the original distribution of the dataset.

Lastly, with reference to Results 5 and 6, we propose a few recommendations to achieve a higher success rate in the running of our DI-LAA. For experiments run on datasets of an unknown distribution, it is recommended to experiment with different partitionings before eventually selecting the distribution which outputs the highest r-score. We concluded that different partitionings perform differently on datasets with different underlying distributions, and from the results, there is no particular partitioning which performs significantly better than the rest across the different dataset distributions. However, if the dataset distribution is known, it is recommended to use a partitioning that matches the dataset distribution, provided that the values from c_0 and d_0 are selected evenly across the entire continuum of the partitioning distribution. This may not work for all cases. For example, if the dataset distribution is Zipfian and the value of $(n - m)$ is small. Since the gradient of the Zipfian graph is close to zero at smaller values, the difference between values is small, hence, uniform partitioning is preferred instead of Zipfian partitioning.

Conclusion

Through our experiments, we found that both our DF-LAA and DI-LAA attack are effective in deciphering ciphertexts. However, one limitation to our experiment is that our experimental dataset is synthetically generated to reflect real-world data, but may not be entirely representative of real world data distribution. In the future, we hope to run our attack on a real world deterministically encrypted database, or explore other, more secure forms of database encryption, such as searchable encryption. We believe our work will have a potential impact on different stakeholders. Ethical security hackers carrying out similar attacks to ensure database security would be able to take our recommendations into consideration in achieving a higher attack success rate. Moreover, with a better understanding of the extent of which leakage can be exploited by adversaries, encryption scheme designers can improve on existing encryption schemes to further minimise leakage, while companies could evaluate other encryption schemes for the encryption of SQL databases, such as random encryption schemes, to enhance security.

Acknowledgements

We would like to thank our mentors, Dr Ruth Ng and Mr Ng Wei Cheng for taking time out of their busy schedules to meet with us and provide invaluable guidance and support. We would also like to thank our teachers from HCI and RI for facilitating the project. Lastly, we would like to thank Song Yiyang for providing the code for the partition optimisation algorithm.

References

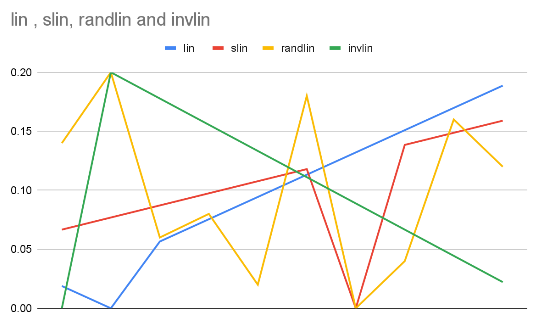
[1] Popa, R. A., Redfield, C. M. S., Zeldovich, N., & Balakrishnan, H. (2012). CryptDB. *Communications of the ACM*, *55*(9), 103–111. https://doi.org/10.1145/2330667.2330691
[2] Lambregts, S., Chen, H., Ning, J., & Liang, K. (2022). VAL: Volume and Access Pattern Leakage-Abuse Attack with Leaked Documents. *Computer Security – ESORICS 2022*, 653–676. https://doi.org/10.1007/978-3-031-17140-6_32
[3] Cash, D., Grubbs, P., Perry, J., & Ristenpart, T. (2015). Leakage-Abuse Attacks Against Searchable Encryption. *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. https://doi.org/10.1145/2810103.2813700
[4] *SQL Joins*. (n.d.). https://www.w3schools.com/sql/sql_join.asp
[5] J. Li, G. Wei, J. Liang, Y. Ren, P. P. C. Lee and X. Zhang, "Revisiting Frequency Analysis against Encrypted Deduplication via Statistical Distribution," *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications*, 2022, pp. 290-299, doi: 10.1109/INFOCOM48880.2022.9796897
[6] "Zipf Distribution - an Overview | ScienceDirect Topics." *Zipf Distribution - an Overview | ScienceDirect Topics*, www.sciencedirect.com/topics/computer-science/zipf-distribution.
[7] *Hungarian Maximum Matching Algorithm | Brilliant Math & Science Wiki*. (n.d.). https://brilliant.org/wiki/hungarian-matching/
[8] *Matching (Graph Theory) | Brilliant Math & Science Wiki*. (n.d.). https://brilliant.org/wiki/matching/

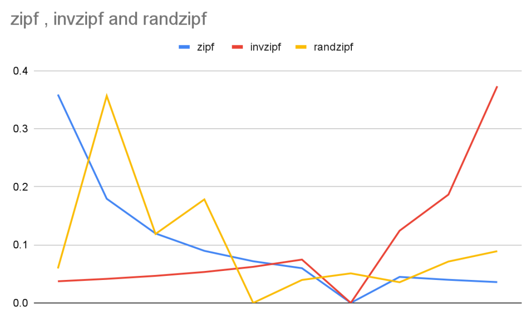
Appendix

Appendix A: Souce of synthetically-generated dataset

Github link to all synthetic data: Synthetic Data

Appendix B: Distribution of ciphertext in datasets used





a) Graph of linear distributions (linear, slow-linear, random linear and inverse linear)
b) Graph of zipfian distributions (zipfian, inverse zipfian and random zipfian)

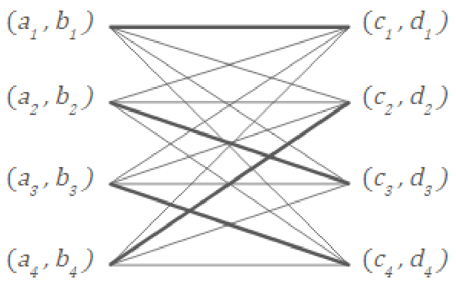
Do note that lin refers to linear, zipf refers to Zipfian, slin refers to slow linear, inv refers to inverse, and rand refers to random.

Appendix C: Generation of auxiliary frequency vector

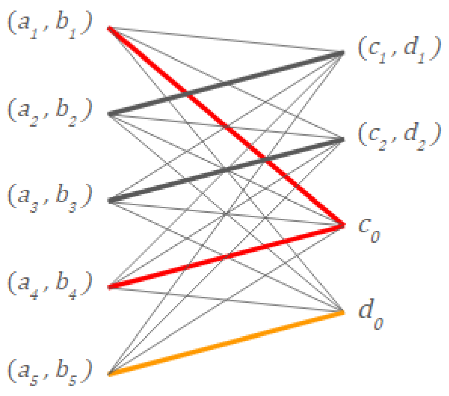
To achieve an approximately identical distribution, we apply a random percentage p, where range of p is $R_p = [-5\%,5\%]$ to each integer x_i, where i = 1, … , n, to obtain an intermediate vector $\vec{x}_{int}$. We then scale the intermediate vector by dividing each integer in the intermediate vector by $S = \sum_{i=1}^{n} \vec{x}_{int}$ to obtain the resultant output $\vec{y}$. We repeated the experiment with other values of p in the range of $R_p = [-10\%,10\%]$ and $R_p = [-20\%,20\%]$.

Appendix D: Sample matching for DF-LAA and DI-LAA

For the DF-LAA attack, given $n = 4$, a sample matching could be as follows:



For the DI-LAA attack, given $n = 5$, $m = 2$, a sample matching could be as follows:



Appendix E: Generation of $\vec{e}$

For $\vec{e}$ to follow a uniform distribution, we generate $\vec{e}$ where $e_i = \frac{1}{n-m}$, $i \in \{1, 2, \dots, n - m\}$.

As for linear and Zipfian distributions, we first generate an intermediate vector $\vec{e}_{int}$. We then scale $\vec{e}_{int}$ by dividing each value e_{int_i} in the vector by $S_{e_{int}} = \sum_{i=1}^{n-m} e_{int_i}$ to obtain the final probability distribution vector $\vec{e}$, such that $\sum_{i=1}^{n-m} \vec{e} = 1$.

For $\vec{e}_{int}$ to follow a linear distribution, we generate $\vec{e}_{int}$ such that $e_{int_i} = i$, $i \in \{1, 2, \dots, n - m\}$.

Given that the probability mass function for Zipfian distribution is:

$$f_s(x) = \frac{x^{-s}}{H_{x,s}} \text{ for integers } x \geq 1,$$

where $H_{x,s}$ is the generalised harmonic number, $H_{x,s} = \sum_{n=1}^{x} \frac{1}{x^s}$, s = 1

For $\vec{e}_{int}$ to follow a Zipfian distribution, we generate $\vec{e}_{int}$ such that $e_{int_i} = f_s(i)$, $i \in \{1, 2, \dots, n - m\}$.

Appendix F: Code for Partition Optimisation Algorithm (credits to Song Yiyang)

```
def partition(A, B, c, d):
    '''
    A, B are lists of integers of length n
    c, d are integers
     Returns a partition R, S of [0, n-1] that maximises sum(A_i, i in R)^c *
sum(B_i, i in S)^d
    '''
    big_no = 1000000
    A, B = [round(i * big_no) for i in A], [round(j * big_no) for j in B]

    n = len(A)
    L = max(sum(A), sum(B))

    best, path = [-inf * inf] * (L + 1), [0] * (L + 1)
    best[0] = 0

    for i in range(n):
        new_best, new_path = [-inf * inf] * (L + 1), [0] * (L + 1)
        for a_sum in range(L + 1):
            old_a, old_b = a_sum, best[a_sum]
            new_a, new_b = a_sum + A[i], old_b + B[i]
              if new_a <= L: new_best[new_a], new_path[new_a] = old_b, path[old_a] *
2
              if new_b > new_best[old_a]: new_best[old_a], new_path[old_a] = new_b,
path[old_a] * 2 + 1

        best, path = new_best, new_path

    max_val, max_path = log(0), -1
    for i in range(L + 1):
        val = c * log(i) + d * log(best[i])
        if val > max_val: max_val, max_path = val, path[i]

    R, S = [], []
    for i in range(n - 1, -1, -1):
        if max_path % 2 == 1:
            S.append(i)
        else:
            R.append(i)
        max_path //= 2

    return R, S
```

Appendix G: Raw table of results

Each dataset is labelled in the following format: {DF/DI}_{number of rows}_{number of distinct ciphertexts}_{dataset distribution}, with run number as well as the partitioning used for the DI case (indicated in brackets) at the back.

Link to the raw results: DF/DI Experimental Results