# ADVERSARIAL LEAKAGE ANALYSIS USING DYNAMIC PROGRAMMING

Jemma Lee Miin Yee[1], Cadence Wern Sea Loh[1], Ruth Ng Ii-Yung[2,3]

[1]Raffles Girls' School, 2 Braddell Rise, Singapore 318871
[2]DSO National Laboratories, 20 Science Park Drive, Singapore 118230
[3]A*STAR, 1 Fusionopolis Way, #20-10 Connexis, Singapore 138632

---

## ABSTRACT

In this work, we take on the role of an eavesdropper adversary who captures leakage from client-server interactions during SELECT and JOIN queries on an encrypted SQL database. We present a novel Leakage Abuse Attack (LAA) that uses dynamic programming to solve the SI-LAA (Single-Column Incomplete Information LAA) problem optimally. We evaluate how the accuracy of the SI-LAA is influenced by the type and percentage of querying by the clients, as well as the number of plaintexts and ciphertexts. This approach is then extended to the DI-LAA (Double-Column Incomplete Information LAA) problem, providing a heuristic solution when used with the Partitioning Optimization Algorithm. We show that this approach is optimal for special cases where datasets are totally ordered and implement various heuristic sorting methods to make the heuristic attack more optimal for general cases. Experimentation on real-world and artificially generated datasets has shown that both the SI-LAA and DI-LAA can be solved with some success using Dynamic Programming.

## INTRODUCTION

In a real-world scenario, encryption is a common method used by clients to ensure the security of their databases so that adversaries will not gain access to the data, especially if the data is confidential. Clients use encryption schemes such as CryptDB to encrypt their datasets deterministically before storing the encrypted dataset on a cloud server, to retain the ability to query the data. In our project, we investigate how secure this scheme is by taking on the role of "man-in-the-middle" eavesdroppers standing between the client and the server. When the client requests data from the server, the data exchanged between the client and the server results in the eavesdropper gaining useful information about the database (otherwise known as a leakage profile). Based on the leakage profile, leakage abuse attacks (LAAs) can be conducted in order to deduce the encrypted values in these datasets. The input of a LAA is a full or partial leakage profile of a particular crypto scheme and auxiliary information. The output of this attack is the guessed mapping f of plaintexts to ciphertexts, and the probability of this mapping, Pr[f]. This mapping allows us to make inferences about the encrypted data. In this paper, we describe a novel LAA that investigates how to make use of partial information from a single column (SI-LAA) and two columns (DI-LAA) using Dynamic Programming. Our algorithm solves the Single Column Incomplete Information LAA (SI-LAA) optimally and the Double Column Incomplete Information LAA (DI-LAA) heuristically, and optimally in some cases. We have also experimentally demonstrated that both LAAs can be solved with relatively high accuracy on real-world datasets.

## BRIEF OVERVIEW OF SI-LAA AND DI-LAA

In the **SI-LAA**, there is one column of data of both auxiliary and ciphertext datasets. Some of the encrypted values in the ciphertext column are unknown. This LAA models a scenario where eavesdroppers gain access to some encrypted rows in a confidential database when clients make a series of SELECT queries on it. As seen in Table 1, when one query is made, all rows corresponding to that query are returned. The number of rows returned is leaked to

eavesdroppers, allowing them to learn the frequency information of that particular value. Subsequently, the eavesdropper derives new frequency information when a different value is queried by the client. As the client may not make all possible queries, the frequency information obtained by the eavesdropper may not be complete (partial leakage profile).

| **Original dataset (T_1)** | | The stakeholder has made 2 queries thus far:<br>1. SELECT (Encrypted value) FROM T_1 WHERE Encrypted value = A<br>2. SELECT (Encrypted value) FROM T_1 WHERE Encrypted value = C | **Leaked dataset (T_2)** | |
|---|---|---|---|---|
| Encrypted value | Frequency | | Encrypted value | Frequency |
| A | 100 | | A | 100 |
| B | 78 | | C | 59 |
| C | 59 | | "Unknown" | 78 |

Fig 1: Example of Input Tables (left) and Output Table (right) in a SQL SELECT Query

We looked at 3 different scenarios of select queries: (1) **Top-x% queries**: Any queries that are made are in the top-x% of frequencies; (2) **Unweighted queries**: The adversary is equally likely to ask for any value in the columns' range; (3) **Weighted queries**: The more often a value appears in the column, the more likely that an adversary queries it.

In the **DI-LAA**, each auxiliary and ciphertext dataset has two columns of data and both ciphertext columns have partial data. The client sends a JOIN request to the server, so values found in the intersection of the two columns are combined to form a new column (that is leaked to the eavesdropper), while values that are not found in this intersection are "unknown". An example of a JOIN query is shown in Tables 2 and 3.

| Encrypted value (T_1) | Frequency |
|---|---|
| Dog | 4 |
| Cat | 9 |
| Meerkat | 23 |
| Fly | 10 |
| Giraffe | 3 |

| Encrypted value (T_2) | Frequency |
|---|---|
| Crocodile | 5 |
| Meerkat | 30 |
| Dog | 15 |
| Rabbit | 13 |
| Anteater | 7 |

| JOIN(Encrypted value) FROM T_1 and T_2 | | | |
|---|---|---|---|
| Meerkat | 23 | Meerkat | 30 |
| Dog | 4 | Dog | 15 |
| "Unknown" | 32 | "Unknown" | 25 |

Fig 2: Example of Input Tables (left) and Output Table (right) in a SQL JOIN Query

## SI-LAA: FRAMEWORK

We use a Dynamic Programming approach to solve the SI-LAA problem. Dynamic Programming consists of recursively defined algorithms that store previously calculated values to save on computational costs (memoisation). Stored data can be easily accessed for use in subsequent steps. This is an improvement on the Brute Force approach - Dynamic Programming algorithms reduce time and space complexity as they only calculate a portion of all permutations based on key observations that govern the algorithm's control flow. Our key observation is that at each step of our algorithm, the largest unassigned auxiliary probability should only be mapped to the highest possible ciphertext frequency or the "unknown" row.

The inputs of the algorithm are:
(1) **Vector a**: a tuple of $n$ auxiliary probabilities sorted in ascending order, [$a_1$, $a_2$, $a_3$, ..., $a_n$ ].
(2) **Vector c**: a tuple of ($m$+1) ciphertext frequencies, [$c_0$, $c_1$,$c_2$, $c_3$, ... $c_m$ ] where $c_0$ is the sum of ciphertext frequencies mapped to "Unknown". $c_1$ to $c_m$ are known ciphertext frequencies sorted in ascending order.
The output is the mapping of ciphertext frequencies to auxiliary probabilities with the highest probability. The probability of the mapping, Pr[f], is calculated as $\left(\prod_{f(i)\neq 0} a_i^{c_{f(i)}}\right)\left(\sum_{f(i)=0} a_i\right)^{c_0}$ In our algorithm, we define DP[n,m,**δ**] where $n$ and $m$ refer to the number of unassigned plaintexts (in **a**) and ciphertexts (in **c**), respectively, and **δ** refers to the sum of probabilities of all plaintexts mapped to $c_0$. DP[n,m,**δ**] stores previously calculated mappings and their corresponding probabilities and can be retrieved for future calculations, reducing the number of computations required. The flow of the algorithm is shown in Fig 1.

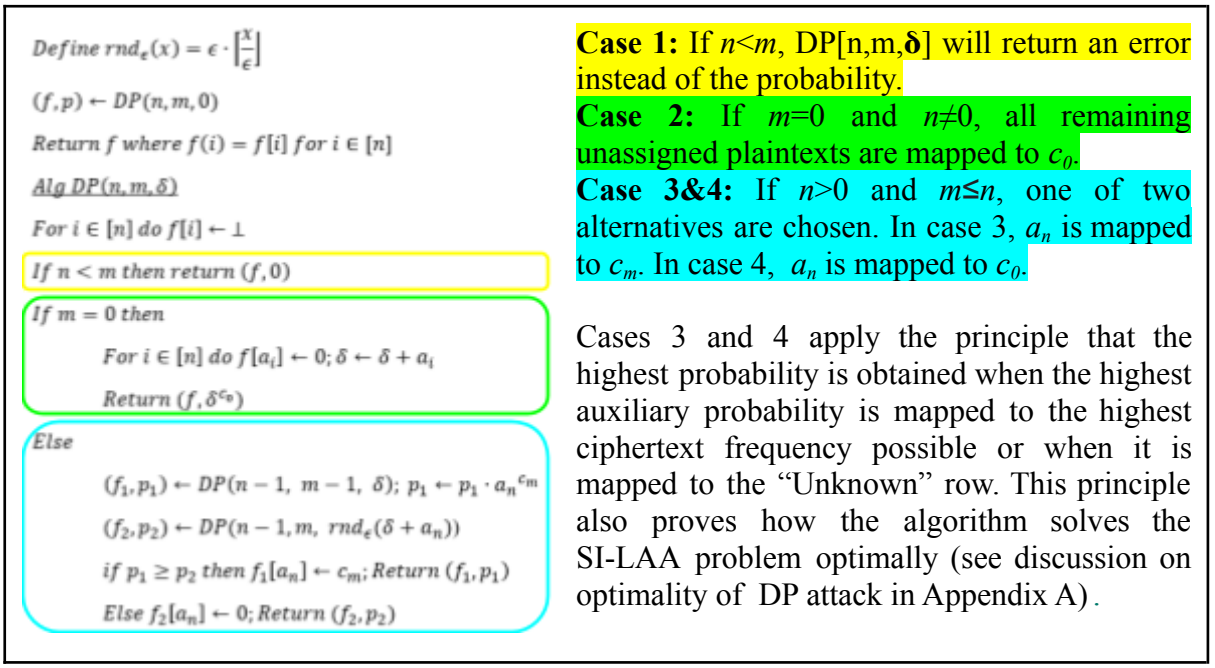

Fig 3: Pseudocode (left) and Attack Control Flow (right) for DP attack on SI-LAA

## SI-LAA: RESULTS AND DISCUSSION

We carried out various experiments using real-life datasets such as HCUP and voter databases (for more information, see appendix B) and tested the accuracy of the attack for different querying types and percentages. The accuracy of the attack is measured using: **(1) "V-score"** - Percentage of known ciphertext values mapped correctly; **(2) "R-score"** - Percentage of known ciphertext rows mapped correctly. Both these scores concern only the ciphertext dataset and do not consider rows mapped to "Unknown".

Our full results can be found in Appendix E. In this section, we summarise our results and trends observed via various graphs.


Fig. 4: Graph of V-score against percentage queried for large and small datasets

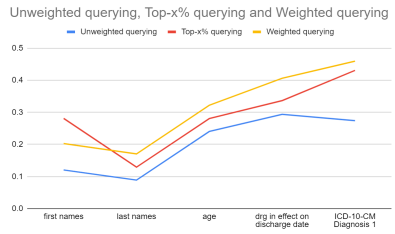


Fig. 5: Graph of Unweighted, Top-x%, and Weighted querying against various datasets

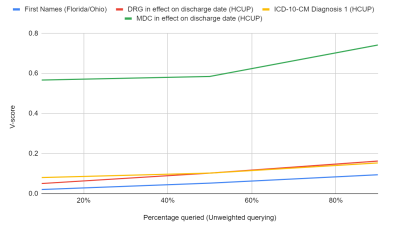


Fig. 6: Graph of V-score against percentage queried using unweighted querying

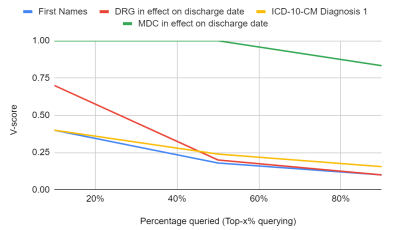


Fig. 7: Graph of V-score against percentage queried using Top-x% querying

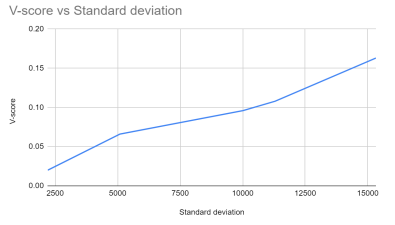


Fig. 8: Graph of V-score against standard deviation of ciphertext frequencies

As seen in Fig. 4, which compares the V-score for large datasets (>90 rows) against that of small datasets (<10 rows), an increase in the number of rows leads to a decrease in the accuracy of the attack, due to decreased chances of mapping correctly when there are more values. Datasets with fewer rows generally have V-scores and R-scores close to 1.

As seen in Fig. 5, for the same percentage queried, the average accuracy of the attack usually follows the order: weighted (0.31) > top-x% (0.29) > unweighted (0.20), as weighted and top-x% querying have a higher chance of querying ciphertexts with high frequencies, and weighted querying provides more variance to the data (explained below).

As seen in Fig. 6, for **unweighted querying**, accuracy of the attack increases with percentage queried. On average, the V-score increases by 10%, from 0.179 to 0.288, as the percentage queried increases from 10% to 90%. Given that more values are "known" with a higher percentage queried, the chances of mapping correctly should be higher as well.

As seen in Fig. 7, for the **top-x% querying**, as the V-score and R-score are inversely correlated to the percentage queried. This is because the smaller ciphertext frequencies have less variance and the number of values matched correctly remains approximately constant. Since the proportion represented by these correctly matched values decreases, the V-score and R-score decrease. On average, as the percentage queried increases from 10% to 90%, the V-score decreases by over 30%, from 0.625 to 0.297, and the R-score decreases by 21% from 0.679 to 0.465.

As seen in Fig. 8, the higher the standard deviation of the ciphertext frequencies, the higher the accuracy of the attack. We compared the standard deviation of the sampled ciphertext frequencies (using unweighted sampling) derived from 5 similar datasets containing 100 plaintexts each. As standard deviation of the ciphertext frequencies increases from 2200 to 15320, V-score increases from 0.02 to 0.163. When the standard deviation of the ciphertext frequencies is high, they are further from one another and more distinct, so chances of mapping correctly increase. This conclusion is valid only when the datasets have a Zipfian distribution, since a high standard deviation of ciphertext frequencies is usually representative of a Zipfian (exponential) distribution. If the ciphertext frequencies follow an oscillating/sinusoidal pattern, while standard

deviation may be high, the highest and lowest ciphertext frequencies are close to each other, so the chances of mapping correctly is relatively low.

## DI-LAA: FRAMEWORK

In this section, we introduce and compare variations of solutions to the DI-LAA problem, which looks at two columns of partial data, instead of one column of partial data in the SI-LAA. The DI-LAA problem is solved using the Dynamic Programming algorithm (introduced above) and the Partitioning Optimization Approximation Algorithm (POAA; see Appendix C). In each variation of the DI-LAA, the algorithms used for the attack differ slightly and have varying degrees of success in solving the DI-LAA. However, for all the cases, the inputs and outputs are the same. As compared to the SI-LAA which has 2 inputs, the DI-LAA has 4 inputs:

(1) **Vector a**: a tuple of auxiliary data with $n$ elements, [$a_1$, $a_2$, $a_3$, ..., $a$ ].
(2) **Vector b**: a tuple of auxiliary data with $n$ elements, [$b_1$, $b_2$, $b_3$, ..., $b$ ].
(3) **Vector c**: a tuple of ciphertext data with ($m$+1) elements, [$c_0$, $c_1$, $c_2$, $c_3$, ... $c_m$].
(4) **Vector d**: a tuple of ciphertext data with with ($m$+1) elements, [$d_0$, $d_1$, $d_2$, $d_3$, ... $d$ ].
**a** (resp. **b**) is the auxiliary data corresponding to ciphertext frequency **c** (resp. **d**), both of which refer to column 1 (resp. 2). $c_1$-$c$ in **c** [$d_1$-$d$ in **d**] represents all the frequencies of values found in the intersection of the two columns of plaintext. $c_0$ and $d_0$, similarly, are defined as the values not found in the intersection of the two columns of plaintext.

Similar to the SI-LAA, the output of the attack for the DI-LAA is a single mapping f of plaintexts (**a**, **b**) to ciphertexts (**c**, **d**) that maximises the probability of the mapping,

$$Pr[f_k] = argmax_f\left[\left[\prod_{j=1}^{m}\left(a_{f_{k^{-1}(j)}}\right)^{c_j}\left(b_{f_{k^{-1}(j)}}\right)^{d_j}\right]\left[\sum_{j\in f^{-1}(m+1)} a_j\right]^{c_0}\left[\sum_{j\in f^{-1}(m+2)} b_j\right]^{d_0}\right] \text{ where } k = 1 \text{ or } 2$$

The goal is to return a high-probability mapping, f, of plaintexts to ciphertexts, which should apply to both **a-c** and **b-d** pairs. This may not necessarily be the mapping of highest probability as most of these attacks are only heuristic and this problem is hard. This is because our key observation from the DP attack on the SI-LAA case does not apply here. The two columns of auxiliary data may not follow similar distributions - for example, in the lin-invlin case, the values in one column are strictly increasing, while the values in the other are strictly decreasing. When $a_i$ (largest plaintext in **a**) is mapped to $c_m$ and this mapping is applied to $b_i$ and $d_m$, $b_i$ may not be the largest plaintext in vector b, and vice versa. Given the set of $\{(a_i,b_i)\}$ where $i \in n$, we define a partial ordering where $(a_i,b_i) \leq (a_j,b_j) \Leftrightarrow a_i \leq a_j$ and $b_i \leq b_j$. However, this does not constitute a total ordering - there is no ordering of vectors a and b such that $a_1 \leq a_2 \leq a_3 \ldots a_n$ and $b_1 \leq b_2 \leq b_3 \ldots b_n$. Hence, it is not guaranteed that mapping $a_n$ to $c_m$ and $b_n$ to $d_m$ always produces the mapping of the highest probability. We implemented several different attacks to overcome this problem. The description of each algorithm, as well as the results, can be found below.

## DP1: FRAMEWORK

The key intuition of DP1 is to run the DP algorithm on both sets of auxiliary-ciphertext pairs, as there are now two columns instead of one. The algorithm is described in Fig 9.

| | |
|---|---|
| $\underline{Alg\ DP1(a,b,c,d)}$<br>$(f,p_1) = DP(SORT(a),c)$ | $(g,p_2) = DP(SORT(b),d)$<br>$g^{-1}(0) = (x_1,\ldots x_p)$ |

| | |
|---|---|
| $f^{-1}(0) = (x_1,\ldots x_p)$<br>$a_1 = (a_{x_1},\ldots, a_{x_p})$<br>$b_1 = (b_{x_1},\ldots, b_{x_p})$<br>$P_1 \leftarrow POAA(a_1, b_1, c_0, d_0)$<br>$For\ i \in [n]$:<br>$\quad If\ i \in P_1\ then\ f[i] = M + 1$<br>$\quad Else\ f[i] \leftarrow M + 2$ | $a_2 = (a_{x_1},\ldots, a_{x_p})$<br>$b_2 = (b_{x_1},\ldots, b_{x_p})$<br>$P_2 \leftarrow POAA(a_2, b_2, c_0, d_0)$<br>$For\ i \in [n]$:<br>$\quad If\ i \in P_2\ then\ g[i] = M + 1$<br>$\quad Else\ g[i] \leftarrow M + 2$<br>$If\ Pr[f] \geq Pr[g]\ Return\ f$<br>$Else\ Return\ g$ |

Fig 9: DP1 algorithm pseudocode

No heuristic ordering is used - the DP algorithm is run on both columns and POAA is used to divide the unknown values into $c_0$ and $d_0$. The mapping with the higher probability ($f_1$ or $f_2$) is taken as the final mapping.

## DP2: FRAMEWORK

The key intuition of DP2 makes use of the fact that some distributions are identical. DP2 uses a modified version of the Dynamic Programming algorithm, as shown in Fig. 10.

$Alg\ DP2(n, m, \delta_1, \delta_2)$<br>
$For\ i \in [n]\ do\ f[i] = \perp$<br>
$If\ n < m\ then\ return\ (f, 0)$<br>
$If\ m = 0\ then$<br>
$\quad a' \leftarrow (a_1,\ldots, a_n, \delta_1, 0)$<br>
$\quad b' \leftarrow (b_1,\ldots, b_n, 0, \delta_2)$<br>
$\quad P = POAA(a', b', c_0, d_0)$<br>
$\quad For\ i \in [n]$:<br>
$\quad\quad If\ i \in P\ then\ f[i] = M + 1\ ;\ \delta_1 \leftarrow \delta_1 + a_i$<br>
$\quad\quad Else\ f[i] \leftarrow M + 2\ ;\ \delta_2 \leftarrow \delta_2 + b_i$<br>
$\quad Return\ \left(f, \delta_1^{c_0} \cdot \delta_2^{d_0}\right)$<br>
$Else$<br>
$\quad (f_1, p_1) \leftarrow DP2(n - 1, m - 1, \delta_1, \delta_2)$<br>
$\quad p_1 \leftarrow p_1 \cdot a_i^{c_m} \cdot b_i^{d_m}$<br>
$\quad (f_2, p_2) \leftarrow DP2(n - 1, m, \delta_1 + a_i, \delta_2)$<br>
$\quad (f_3, p_3) \leftarrow DP2(n - 1, m, \delta_1, \delta_2 + b_i)$<br>
$\quad If\ p_1 \geq p_3\ and\ p_1 \geq p_2\ then\ f_1[n] \leftarrow m\ ;\ Return\ (f_1, p_1)$<br>
$\quad If\ p_2 \geq p_3\ then\ f_2[n] \leftarrow M + 1\ ;\ Return\ (f_2, p_2)$<br>
$\quad Else\ f_3[n] \leftarrow M + 2\ ;\ Return\ (f_3, p_3)$

$\boldsymbol{\varepsilon}$ = parameter defining how accurately we round the auxiliary values; M = number of ciphertexts

Fig 10: Modified DP algorithm pseudocode for DP2 and DP3

As previously mentioned, the DI-LAA is suboptimal when the distributions are not totally ordered. However, when the datasets are totally ordered (i.e. when **a** and **b** follow the same distribution), the DP algorithm can be optimal - when $a_i$ is mapped to $c_m$ and $b_i$ is mapped to $d_m$, it is guaranteed that both $a_i$ and $b_i$ are the largest plaintexts in **a** and **b** respectively. Since $a_i$=$b_i$ for all $n$, the probability of the mapping can be simplified to $Pr[f] = \left[\prod_{f(i)\neq 0} a_i^{c_{f(i)}+d_{f(i)}}\right]\left[\sum_{f(i)=m+1} a_i\right]^{c_0}\left[\sum_{f(i)=m+2} a_i\right]^{d_0}$. In this special DI-LAA case, it is efficiently solvable by slightly modifying the SI-LAA DP algorithm mentioned in Fig. 1, in order to consider both the "unknown" buckets $c_0$ and $d_0$.

## DP3: FRAMEWORK

The key intuition of DP3 is to extend DP2 to general cases where datasets are not totally ordered by using heuristic sorting to order ($a_i$, $b_i$) pairs.

$Alg\ DP3A(a, b, c, d)$
$Define\ x = \{\}$
$For\ i \in n\ do\ x \leftarrow \{(a_i, b_i)\}$
$x \leftarrow SORT(x)$
$For\ i \in n\ do\ a[i] \leftarrow x[i][0]\ ;\ b[i] \leftarrow x[i][1]$
$(f_1, p_1) \leftarrow DP2(n, m, 0, 0)$
$Define\ y = \{\}$
$For\ i \in n\ do\ y \leftarrow \{(b_i, a_i)\}$
$y \leftarrow SORT(y)$
$For\ i \in n\ do\ a[i] \leftarrow y[i][0]\ ;\ b[i] \leftarrow y[i][1]$
$(f_2, p_2) \leftarrow DP2(n, m, 0, 0)$
$Return\ f = argmax_f(p_1, p_2)$

$Alg\ DP3B(a, b, c, d)$
$Define\ x = \{\}$
$For\ i \in n\ do\ x \leftarrow \{(a_i, b_i)\}$
$x \leftarrow SORT(x, key = a_i + b_i)$
$(f, p) \leftarrow DP2(n, m, 0, 0)$
$Return\ f$

Fig 11 (left): DP3A heuristic sorting
Fig 12 (right): DP3B heuristic sorting

The first heuristic sorting method **(DP3A - see Fig. 11)** sorts one vector according to the ascending order of the other vector, before repeating this in the opposite manner. DP2 is then carried out on the two different sets of **a** and **b** (derived from the different sortings), and the mapping of higher probability is chosen. The second method **(DP3B - see Fig. 12)** uses the summation of terms to sort ($a_i$, $b_i$) pairs. ($a_i$, $b_i$) pairs are sorted by $a_i$+$b_i$ for $i \in n$. For example, if **a** = [0.5,0.6,0.1] and **b** = [0.5,0.3,0.6], DP3A will produce two sortings: (1) **a** = [0.1,0.5,0.6] and **b** = [0.6,0.5,0.3]; (2) **a** = [0.6,0.5,0.1] and **b** = [0.3,0.5,0.6]. DP3B will result in the following sorting: **a** = [0.1,0.6,0.5] and **b** = [0.6,0.3,0.5].

## DI-LAA: FINDINGS

To test the accuracy of the different DI-LAA, we used randomly-generated double-column datasets (i.e. random **a**, **b**, **c**, and **d**) based on different distributions (for more information, see Appendix D) - and varying lengths of the vectors, before randomly removing some ciphertexts in **c** and **d** (similar to an unweighted query) to mimic a partial leakage profile. We introduced different amounts of noise (5%, 10%, 20%) to **a** and **b** to better simulate real-life datasets. We then measured the V-score and R-score (see results table in Appendix E).

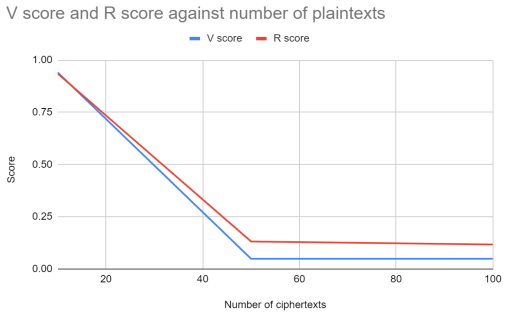


Fig. 13: V score and R score against number of plaintexts

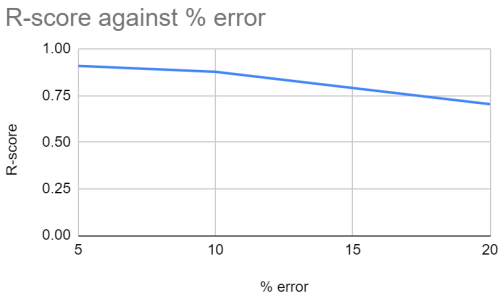


Fig. 14: R-score against % error introduced to auxiliary probability

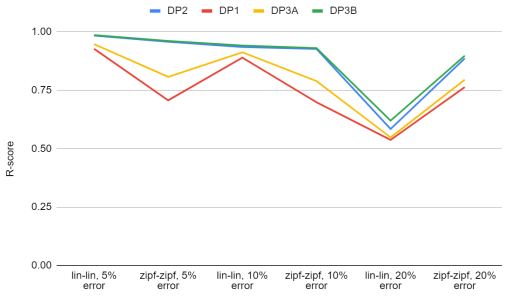


Fig 15: Graph of R-score for different solutions for correlated distributions

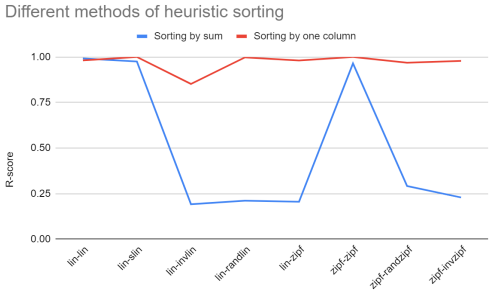


Fig 16: Graph of R-score for DP3A and DP3B for different distributions

Based on our results, we can conclude that our attacks have some accuracy in matching the plaintexts to the ciphertexts (all V-scores and R-scores are greater than 0). As seen in Fig. 13, similar to the SI-LAA, as the number of ciphertexts increases, accuracy of the DI-LAA decreases. For small datasets with fewer ciphertexts, the DI-LAA has almost 100% accuracy with V and R scores close to 1.

As seen in Fig. 14, accuracy is inversely correlated with the error percentage introduced to vectors a and b. This is because there is more variance in the auxiliary dataset and it is more likely that the auxiliary values do not truly represent the distribution of ciphertexts.

As seen in Fig. 15, DP3B performs the best for correlated distributions, followed by DP2, DP3A, then DP1. Due to randomness introduced to the auxiliary dataset, DP3B outperforms DP2 as the mapping with the highest probability may no longer be the solution with the highest accuracy. This can be seen from how the difference in R-score between DP2 and DP3B is the highest when 20% error is introduced into the auxiliary data.

It is also noted that DP1 performs the most poorly as heuristic sorting is not used. Heuristic sorting would increase the chances of the highest possible plaintext being mapped to the highest possible ciphertext for both a-c and b-d pairs, thus increasing the chance that the mapping of the highest probability is chosen

As seen in Fig. 16, for cases where the two distributions are decorrelated (such as in lin-invlin), the sorting by sum heuristic has a low accuracy, since the sum of all ($a_i$, $b_i$) pairs become approximately equal and difficult to sort correctly. For other combinations of distributions, the accuracy of the DI-LAA is similar for both methods of heuristic sorting.

## LIMITATIONS OF SI-LAA AND DI-LAA

While both the SI-LAA and DI-LAA have relatively high success rates, they have a few limitations. Firstly, the speed of computation is slow when $n$ and $m$ are large. For the SI-LAA, we truncated all experimental databases to the top 100 unique ciphertexts, so the results were not fully representative of the actual distribution. However, the results are still

largely representative as the values with highest frequency are still chosen. Secondly, all **δ** values were rounded to 3 decimal places (i.e. discretization) to reduce the number of unique **δ** values and the number of computations required. When the number of unique **δ** values is increased, the number of unique DP[n,m,**δ**] is increased and the total time taken for computation increases. Thirdly, as the memory of the computer is limited and dynamic programming depends on the storage of previously calculated values, our program is unable to run to completion for extremely large datasets.

## CONCLUSION

Our research has demonstrated that Leakage Abuse Attacks on partial leakage profiles can be quite effective. In particular, the Dynamic Programming algorithm can be used to solve the SI-LAA optimally. When the DP algorithm is used together with the Partitioning Optimization Approximation Algorithm (POAA), it is able to solve the DI-LAA heuristically. If both columns follow the same distribution (i.e. totally ordered datasets), the attack becomes optimal when modified slightly. The accuracy of the DI-LAA can be further improved by using heuristic sorting methods. When the two columns are decorrelated or random, heuristic sorting based on only one column should be used. Otherwise, either heuristic sorting by one column or by sum of $a_i$ and $b_i$ can be used to achieve results of similar accuracy. A hybrid of the two different types of sorting can also be implemented so that the sorting that produces a mapping of higher probability can be chosen. Our attacks show that deterministically encrypting data is dangerous, even if only partial information is leaked to an eavesdropper. As a large number of confidential databases (such as healthcare records) are still encrypted deterministically, we believe that more secure methods for encryption should be implemented to better protect such data. In the future, we also hope to improve our attacks by finding a more optimal solution for the DI-LAA.

## ACKNOWLEDGEMENTS

We would like to thank our mentor, Ruth Ng Ii-Yung, for her guidance and support throughout our research process and for providing us with datasets. We would also like to thank Li Yao'An for guiding us with our DP algorithm and Khu Boon Tat Daren for helping us write up the POAA. Finally, we would like to thank Alexander Hoover for providing us with the various diagrams explaining our algorithms.

## REFERENCES

[1] Kamara, S., & Moataz, T. (2019). Computationally Volume-Hiding Structured Encryption. Retrieved December 13, 2022, from https://www.iacr.org/archive/eurocrypt2019/114760319/114760319.pdf

[2] HCUP-US Home Page. (2016). Ahrq.gov. https://www.hcup-us.ahrq.gov/

[3] Popa, Raluca Ada et al (2011). "CryptDB: Protecting confidentiality with encrypted query processing." ACM Press. https://dl.acm.org/doi/10.1145/2043556.2043566

# APPENDIX

## Appendix A: Discussion on Optimality of DP attack

In this discussion, we assume that discretization is not used in the Dynamic Programming (DP) algorithm. SI-LAA can be solved optimally using the Dynamic Programming Algorithm as the mapping of the highest probability is always produced. This is because the Dynamic Programming Algorithm is an improvement on the Brute Force method, which can be proven to be optimal. This is because the Brute Force method can simply be reduced to the problem of choosing n-m plaintexts to be mapped to the "Unknown" bucket, while mapping the remaining m plaintexts to the m known ciphertexts using frequency analysis. Frequency analysis always maps the largest plaintext to the largest ciphertext, the 2nd largest plaintext to the 2nd largest ciphertext, and so on and so forth, thus maximising the probability of the mapping of the m plaintexts to the known ciphertexts. Given that the mapping to known ciphertexts is always optimal, we now consider the unknown bucket - by taking all possible subsets of n-m plaintexts (i.e. all possible combinations of plaintexts mapped to the unknown bucket), we can then choose the subset that produces the mapping of highest probability. Hence, the Brute Force method can solve the SI-LAA optimally.

The above explanation also supports the key observation mentioned for our Dynamic Programming Algorithm - the largest plaintext will always be mapped to either the unknown bucket $c_0$ or the largest ciphertext, because frequency analysis ensures that if the largest plaintext is not mapped to the unknown bucket, it will be mapped to the largest plaintext.

Although the Dynamic Programming algorithm does fewer computations ($n$C$m$ subsets) than the Brute Force Method ($m$! subsets), it is still guaranteed that the mapping of highest probability is among the reduced number of subsets calculated. Suppose the Brute Force Method has a search space S (shown in Fig. 16) with $m$! subsets, and S can be partitioned into subspaces for each $S \subseteq [n]$ where $|S| = m$. For each case, the mappings are arranged in ascending order of their probabilities. The Dynamic Programming algorithm considers only the right-most mapping with the highest probability of each subspace. It finds the mapping with the highest probability as it searches all possible subspaces - for every plaintext $a_n$, it considers both options (mapping to $c_0$ or $c_m$) that might produce an optimal mapping. Since the optimal mapping is one of the mappings found in the last column, the Dynamic Programming Algorithm will always output the mapping of the highest probability and it is thus optimal.

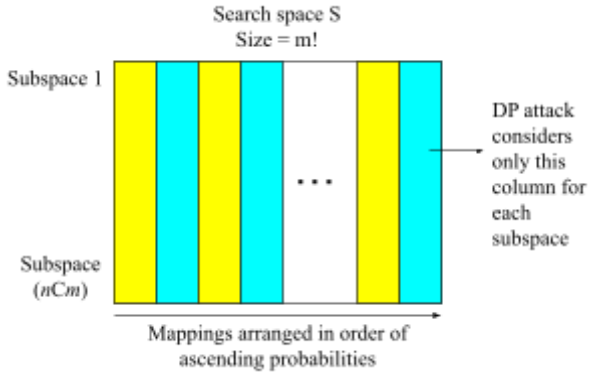


Fig. 16: Diagram comparing Search Space of DP algorithm and Brute Force Method

## Appendix B: Experimentation for SI-LAA

To prove the experimental accuracy of our attack, we ran the DP attack on various real-life datasets: 1) Florida state data (auxiliary) and Ohio state data (ciphertext): These two datasets contain similar sets of information about their respective citizens. We used the "First Name" and "Last Name" columns; 2) HCUP 2018 (auxiliary) and 2019 (ciphertext) datasets: HCUP is America's most comprehensive source of hospital care data. We used the "Age", "DRG in effect on discharge date", "MDC in effect on discharge date", "ICD-10-CM Diagnosis 1", and "Race" columns.

Before conducting the DP attack, if the number of plaintexts and ciphertexts in the dataset exceeded 100, we extracted only the top 100 values with the highest frequency. If there is a value in the auxiliary dataset that is not found in the ciphertext dataset or vice versa, we run the DP attack as per normal. This mimics a real-life situation, as the adversary does not have this knowledge. For each sampling type, we varied the number of rows queried: 10%, 50% and 90%. However, when calculating the "R-score" and the "V-score", these values should not be incorporated into either numerator because there is no chance of "matching" them correctly. The success of our attack can be measured without encrypting the data as we assume that the database is encrypted deterministically, where the encryption system always produces the same ciphertext for a given plaintext and key.

## Appendix C: Partitioning Optimization Algorithm (POAA)

We worked with external mentors (Yao'An and Daren) and implemented the POAA using C++. The algorithm is shown in Fig. 17.

$Alg\ POAA^{\epsilon}(a, b, e_1, e_2)$
$Define\ rnd_{\epsilon}(x, y) = \left(\epsilon A \cdot \lceil \frac{x}{\epsilon A} \rfloor, \epsilon B \cdot \lceil \frac{x}{\epsilon B} \rfloor\right)$
$L[rnd_{\epsilon}(a(1), 0)] \leftarrow 1; L[rnd_{\epsilon}(0, b(1)] \leftarrow 0$
$For\ i = 2,\ldots, n:$
  $For\ j = 1,\ldots, \lfloor \frac{1}{\epsilon} \rfloor:$
    $For\ k = 1,\ldots, \lfloor \frac{1}{\epsilon} \rfloor:$
      $x \leftarrow \epsilon A \cdot j;\ y \leftarrow \epsilon B \cdot k$
      $If\ |L[(x, y)]| = i - 1:$
        $L[rnd_{\epsilon}(x + a(i), y)] = 1 || L[(x, y)]$
        $L\left[rnd_{\epsilon}(x, y + b(i))\right] = 0 || L[(x, y)]$
$m \leftarrow 0; s \leftarrow 0^n$
$For\ j = 0,\ldots, \lfloor \frac{1}{\epsilon} \rfloor:$
  $For\ k = 0,\ldots, \lfloor \frac{1}{\epsilon} \rfloor:$
    $x \leftarrow \epsilon A \cdot j;\ y \leftarrow \epsilon B \cdot k$
    $If\ |L[(x, y)]| = n\ and\ x^{e_1} y^{e_2} > m:$
      $s \leftarrow L[(x, y)]; m \leftarrow x^{e_1} y^{e_2}$
$Parse\ s = b_1 || b_2 || \ldots || b_n;\ P \leftarrow \emptyset$
$For\ i \in [i]\ do\ if\ b_i = 1\ then\ P \leftarrow \{i\}$

*Return P*
where $\boldsymbol{\varepsilon}$ = constant defining precision of x

Fig 17: POAA

## Appendix D: Types of distributions used for DI-LAA

We used the following distributions to generate **a**, **b**, c and **d** for experimentation: (1) lin-lin, (2) lin-slin, (3) lin-invlin, (4) lin-randlin, (5) lin-zipf, (6) zipf-zipf, (7) zipf-randzipf and (8) zipf-invzipf. Distributions (3) and (8) are decorrelated distributions, while (1), (2) and (6) are correlated distributions. The definition of the terms used can be found in the table below.

| **Distribution** | **Definition** |
|---|---|
| Zipfian (zipf) | **Zipf-distributed multi-maps.** To get a concrete bound on the number of truncations, we have to make an assumption on how the response lengths of the multi-map are distributed. Here, we will assume that they are distributed according to the Zipf distribution which is a standard assumption in information retrieval [14,43]. We note that our analysis can be extended to any power-law distribution. More precisely, we say that a multi-map MM is $\mathcal{Z}_{a,b}$-distributed if its $r^{th}$ response has length $\frac{r^{-b}}{H_{a,b}} \cdot N$ where $N \stackrel{def}{=} \sum_{\ell \in \mathbb{L}} \#\mathsf{MM}[\ell]$ is the volume of MM and $H_{a,b}$ is the harmonic number $\sum_{i=1}^{a} i^{-b}$. Throughout, we will consider multi-maps that are $\mathcal{Z}_{m,1}$-distributed where $m = \#\mathbb{L}_{\mathsf{MM}}$. From this assumption, it follows that the set of all response lengths is $L = (L_1, \ldots, L_m) = \left(\frac{N}{1 \cdot H_{m,1}}, \ldots, \frac{N}{m \cdot H_{m,1}}\right)$, **Image taken from https://www.iacr.org/archive/eurocrypt2019/114760319/114760319.pdf** |
| Random-zipfian (randzipf) | Auxiliary values follow a Zipfian distribution but are ordered randomly in the set. |
| Inverse-zipfian (invzipf) | Auxiliary values are opposite to that of a Zipfian distribution. |
| Linear | Distribution with constant gradient of 1 |
| Slow-linear (slin) | Distribution with constant gradient of 0.5 |
| Inverse-linear (invlin) | Auxiliary values are opposite to that of a linear distribution |
| Random-linear (randlin) | Auxiliary values follow a linear distribution but are ordered randomly in the set. |

Table 1: Definitions of Distributions

## Appendix E: Results table for SI-LAA and DI-LAA

**Table 2: DP Results Table**

| | 10% | | 50% | | 90% | |
|---|---|---|---|---|---|---|
| | V-score | R-score | V-score | R-score | V-score | R-score |
| **Top-x% querying** | | | | | | |
| First Names (Florida/Ohio) | 0.400 | 0.474 | 0.180 | 0.281 | 0.100 | 0.202 |
| Last names (Florida/Ohio) | 0.200 | 0.301 | 0.040 | 0.129 | 0.033 | 0.106 |
| Age (HCUP) | 0.200 | 0.496 | 0.149 | 0.281 | 0.286 | 0.312 |
| DRG in effect on discharge date (HCUP) | 0.700 | 0.729 | 0.200 | 0.432 | 0.100 | 0.347 |
| ICD-10-CM Diagnosis 1 (HCUP) | 0.400 | 0.682 | 0.240 | 0.524 | 0.156 | 0.448 |
| MDC in effect on discharge date (HCUP) | 1 | 1 | 1 | 1 | 0.833 | 0.981 |
| **Weighted querying** | | | | | | |
| First Names (Florida/Ohio) | 0.080 | 0.157 | 0.090 | 0.203 | 0.091 | 0.200 |
| Last names (Florida/Ohio) | 0.100 | 0.266 | 0.058 | 0.171 | 0.042 | 0.112 |
| Age (HCUP) | 0.160 | 0.423 | 0.185 | 0.322 | 0.245 | 0.302 |
| DRG in effect on discharge date (HCUP) | 0.300 | 0.588 | 0.186 | 0.457 | 0.154 | 0.382 |
| ICD-10-CM Diagnosis 1 (HCUP) | 0.310 | 0.732 | 0.202 | 0.531 | 0.184 | 0.465 |
| MDC in effect on discharge date (HCUP) | 0.633 | 0.573 | 0.838 | 0.921 | 0.833 | 0.981 |
| **Unweighted querying** | | | | | | |
| First Names (Florida/Ohio) | 0.020 | 0.021 | 0.052 | 0.121 | 0.094 | 0.187 |
| Last names (Florida/Ohio) | 0.030 | 0.024 | 0.026 | 0.089 | 0.039 | 0.109 |

| | | | | | | |
|---|---|---|---|---|---|---|
| Age (HCUP) | 0.060 | 0.056 | 0.170 | 0.240 | 0.279 | 0.298 |
| DRG in effect on discharge date (HCUP) | 0.050 | 0.125 | 0.102 | 0.297 | 0.162 | 0.371 |
| ICD-10-CM Diagnosis 1 (HCUP) | 0.080 | 0.128 | 0.102 | 0.304 | 0.153 | 0.422 |
| MDC in effect on discharge date (HCUP) | 0.567 | 0.785 | 0.585 | 0.714 | 0.742 | 0.954 |

**Table 3: DP1 Results Table**

| Set-up | V-score | R-score |
|---|---|---|
| **10 ciphertexts** | | |
| lin-lin | 0.928 | 0.884 |
| lin-randlin | 0.913 | 0.890 |
| lin-slin | 0.978 | 0.966 |
| lin-invlin | 0.900 | 0.897 |
| lin-zipf | 0.925 | 0.920 |
| zipf-zipf | 0.925 | 0.969 |
| zipf-invzipf | 0.950 | 0.935 |
| zipf-randzipf | 0.963 | 0.982 |
| **50 ciphertexts** | | |
| lin-lin | 0.046 | 0.006 |
| lin-randlin | 0.033 | 0.027 |
| lin-slin | 0.25 | 0.013 |
| lin-invlin | 0.036 | 0.046 |
| lin-zipf | 0.052 | 0.234 |
| zipf-zipf | 0.047 | 0.328 |
| zipf-invzipf | 0.037 | 0.170 |
| zipf-randzipf | 0.043 | 0.125 |
| **100 ciphertexts** | | |
| lin-lin | 0.057 | 0.067 |

| | | |
|---|---|---|
| lin-randlin | 0.050 | 0.050 |
| lin-slin | 0.049 | 0.056 |
| lin-invlin | 0.045 | 0.051 |
| lin-zipf | 0.049 | 0.086 |
| zipf-zipf | 0.061 | 0.286 |
| zipf-invzipf | 0.096 | 0.301 |
| zipf-randzipf | 0.056 | 0.178 |

**Table 4: DP2 Results Table**

| Set-up | V-score | R-score |
|---|---|---|
| **10 ciphertexts, 10000 rows** | | |
| lin-lin, 5% error | 0.938 | 0.984 |
| lin-lin, 10% error | 0.903 | 0.935 |
| lin-lin, 20% error | 0.646 | 0.584 |
| zipf-zipf, 5% error | 0.875 | 0.957 |
| zipf-zipf, 10% error | 0.813 | 0.927 |
| zipf-zipf, 20% error | 0.736 | 0.886 |
| **10 ciphertexts, 1000000 rows** | | |
| lin-lin, 5% error | 0.0925 | 0.961 |
| lin-lin, 10% error | 0.950 | 0.974 |
| lin-lin, 20% error | 0.875 | 0.948 |
| zipf-zipf, 5% error | 0.889 | 0.952 |
| zipf-zipf, 10% error | 0.878 | 0.869 |
| zipf-zipf, 20% error | 0.925 | 0.942 |

**Table 5: DP3A Results Table**

| Set-up | V-score | R-score |
|---|---|---|
| **10 ciphertexts, 10000 rows** | | |
| lin-lin, 5% error | 0.963 | 0.992 |

| | | |
|---|---|---|
| lin-lin, 10% error | 0.878 | 0.912 |
| lin-lin, 20% error | 0.608 | 0.548 |
| lin-slin, 5% error | 1 | 0.975 |
| lin-invlin, 5% error | 0.863 | 0.192 |
| lin-randlin, 5% error | 0.988 | 0.212 |
| lin-zipf, 5% error | 0.950 | 0.206 |
| zipf-zipf, 5% error | 1 | 0.965 |
| zipf-zipf, 10% error | 0.675 | 0.789 |
| zipf-zipf, 20% error | 0.608 | 0.548 |
| zipf-randzipf, 5% error | 0.913 | 0.292 |
| zipf-invzipf, 5% error | 0.940 | 0.230 |
| **10 ciphertexts, 1000000 rows** | | |
| lin-lin, 5% error | 0.988 | 0.204 |
| lin-lin, 10% error | 0.838 | 0.891 |
| lin-lin, 20% error | 0.650 | 0.799 |
| lin-slin, 5% error | 1 | 0.989 |
| lin-invlin, 5% error | 1 | 0.103 |
| lin-randlin, 5% error | 0.988 | 0.204 |
| lin-zipf, 5% error | 1 | 0.206 |
| zipf-zipf, 5% error | 1 | 0.980 |
| zipf-zipf, 10% error | 0.715 | 0.800 |
| zipf-zipf, 20% error | 0.800 | 0.847 |
| zipf-randzipf, 5% error | 0.913 | 0.367 |
| zipf-invzipf, 5% error | 0.940 | 0.237 |

**Table 6: DP3B Results Table**

| Set-up | V-score | R-score |
|---|---|---|
| **10 ciphertexts, 10000 rows** | | |

| | | |
|---|---|---|
| lin-lin, 5% error | 0.975 | 0.992 |
| lin-slin, 5% error | 0.963 | 0.975 |
| lin-invlin, 5% error | 0.179 | 0.192 |
| lin-randlin, 5% error | 0.169 | 0.212 |
| lin-zipf, 5% error | 0.107 | 0.206 |
| zipf-zipf, 5% error | 0.900 | 0.965 |
| zipf-randzipf, 5% error | 0.178 | 0.292 |
| zipf-invzipf, 5% error | 0.098 | 0.230 |
| **10 ciphertexts, 1000000 rows** | | |
| lin-lin, 5% error | 0.988 | 0.996 |
| lin-slin, 5% error | 0.975 | 0.989 |
| lin-invlin, 5% error | 0.113 | 0.103 |
| lin-randlin, 5% error | 0.206 | 0.204 |
| lin-zipf, 5% error | 0.125 | 0.206 |
| zipf-zipf, 5% error | 0.938 | 0.980 |
| zipf-randzipf, 5% error | 0.275 | 0.367 |
| zipf-invzipf, 5% error | 0.113 | 0.237 |