REAL-WORLD IMPLEMENTATION AND SECURITY ANALYSIS OF STRUCTURED ENCRYPTION ON AMAZON WEB SERVICES

Lim Li Xin Jed, Song Yiyang, Ruth Ng Ii-Yung, and Seah Meng Yong Ryan

[1] NUS High School of Mathematics and Science
[2] Raffles Instituition
[3] DSO National Laboratories and Agency for Science, Technology and Research (A*STAR)
[4] McGill University and DSO National Laboratories

Abstract. In this project, we build a structured encryption system for SQL databases which supports SELECT and JOIN queries. This system is implemented on Amazon Web Services. Using our implementation, we investigate the leakage produced by our system and attempt to attack it using leakage abuse attacks both on real-world data and synthetic data. Using the full spectrum of attacks (with full or partial information, and single or double-column setups), we rigorously show that our system is more resilient and secure than past schemes.

1 Introduction

Storage of sensitive data such as medical records is challenging and will often require in-house servers to store the data as it may not be possible to trust 3rd-party cloud servers not to look at the data. However, such servers are expensive and can be difficult to maintain.

Encryption of the whole database is one way to ensure confidentiality by hiding all useful information about the data. However, this means that when a query needs to be made, the whole database must be decrypted which is time-consuming.

Structured encryption schemes provide a solution to these problems. Structured encryption enables secure encryption of data in any data structure while still allowing queries to be made on the encrypted data. Such schemes have been written for a wide variety of data structures such as multi-maps [2, 9], matrices [4], graphs [4] and SQL databases [2, 3, 5–7, 10].

Nevertheless, as a trade-off, some information about the data such as the number of rows in a database returned by a query will be leaked, known as the leakage profile. This leakage profile is what leakage abuse attacks (LAAs) seek to exploit to gain information about the data. Common and usually unavoidable information that is leaked is the number of rows in the response. This can be exploited by using auxiliary data (see Section 3) to identify some values in the column. Other information such as the order of rows has been exploited too [8].

This paper reports the real-world implementation and security analysis of a structured encryption system for SQL databases. In Section 2, we construct this system for SQL databases supporting SELECT and JOIN queries and explain our implementation on Amazon Web Services. In Section 3, we examine the leakage profile of our system, present some LAAs and explain their limitations. In Section 4, we run experiments on real-world and synthetic data to demonstrate the attacks.

2 Structured Encryption

Now, we outline the structured encryption scheme that we have built for SQL databases. Refer to Appendix A for a formal and more detailed definition. In order to answer queries instantaneously, we first precompute every SELECT and JOIN query on a given set of table(s) in $O(N)$ and $O(N^2)$ time respectively. In this report, in all complexity analysis, n is the number of unique ciphertexts and N is the number of rows in the table. Future work could implement better JOIN techniques [3], or come up with better ones, in order to reduce the time complexity and leakage (Section 3). From this, we construct a reverse-index multimap, **N**, which maps each query to the row indexes. Then, we construct another multimap, **M**, that maps each row index to the contents of that row. Finally, we encrypt **M** with response-hiding encryption and **N** with response-revealing encryption. The cryptographic primitives used were the Cryptographic Hash Function SHA-256 for encrypting labels and the Symmetric Encryption Scheme AES-CTR-256 for encrypting values.

2.1 Scheme Implementation and Benchmarking

We implemented the above-mentioned scheme, using Python and Amazon Web Services (AWS). We initialised an AWS ec2 Ubuntu instance to act as the server, while the client was our own computers. In order to communicate between client and server, we used TCP via Python's socket module. To send the multimaps over, we used Python's pickle module, converted them into raw bytes and sent it via socket. pycryptodome was used for all cryptographic functions. We benchmarked our system against 2 tables of 1000 rows each, and the results show that our system is rather efficient. The high initialisation time of 28s is mainly due to precomputing JOINs, which scales proportional to $O(N^2)$. In our implementation, we used naive row by row comparisons, so one area of future work could be to implement a "sort and compare" scheme with time complexity $O(N \log N + \text{number of rows in JOIN output})$, which is normally better than $O(N^2)$ for real-world data. Processing one SELECT and JOIN query took 0.0384s and 1.15s respectively, which is acceptably fast for a real-world application.

3 Leakage Abuse Attacks

Now, we will discuss the leakage profile of our system and how it can be abused to uncover information about the data. The motivation is to measure the security of our system by observing how well it stands up to these attacks. In this section, we will also refer to "ciphertexts". These are **not** the encrypted rows but rather refer to the encrypted values that we know certain rows contain. In LAAs, we seek to obtain the plaintexts associated with these encrypted values.

LEAKAGE PROFILE The following information about the data is leaked by the system.

– **Total Number of Rows** can be inferred from size of client's message during initialisation.
– **Length of Longest Row** can be inferred from the longest entry in **M** during initialisation.

– **Query Uniqueness Pattern** can be inferred from the unique search token during query.
– **Row Access Pattern** can be inferred from which rows of **M** were accessed during query.
– **Number of Rows in Query** can be inferred from size of server's response during query.

We note that this leakage profile is in fact smaller than past schemes such as CryptDB by Popa et al. [11]. The number of rows in a query is only available when the query is made unlike in past schemes where the number of rows in a query is available even before any queries are made. Hence, our scheme is much more secure and resilient.

QUERY INFORMATION Using this data, we will first attempt to extract information about the query that was asked such as the query type (SELECT or JOIN) and which column and table the query was performed on. This information is necessary to know which attacks should be applied on the data. However, there is no sure-fire way to determine the type of query every time, these methods are just heuristics. We provide a brief summary of methods, the details are found in Appendix B.

– **Parity of number of rows returned.** JOINs always return an even number of rows.
– **Uniqueness of rows returned.** JOINs almost always return multiple copies of a row.
– **Number of rows returned for** JOIN**.** Auxiliary data can be used to guess the columns the JOIN was performed on.

3.1 Overview of LAAs

The end goal of an LAA is: given some ciphertexts and some auxiliary plaintext data, output a mapping from each plaintext to one ciphertext. Specifically, we work with ciphertext frequencies and the auxiliary data is a probability distribution of plaintexts on a column of the same type, obtained through looking at other similar but open-source tables (for example, auxiliary data of the Ohio Voter Registry could be the Florida Voter Registry). We call an LAA "optimal" when the mapping it outputs has the highest probability of being the correct mapping (elaborated later in each LAA). There are four scenarios we work with: Single/Double Column, Complete/Incomplete. Single Column refers to SELECT queries, while Double Column refers to JOIN queries. Complete means all relevant ciphertext equality patterns in the column(s) are known to an adversary, while Incomplete means there are some remaining ciphertexts that an adversary does not know whether they are equivalent to one another. Complete/Incomplete scenarios arise because we are using non-deterministic encryption, so the same plaintext will be mapped to different ciphertexts each time it is encrypted. Table 1 summarises the relevant information on the LAAs. c_0, d_0 refer to the number of ciphertexts not returned in the JOIN query in each column, while each a_i, c_i (respectively b_i, d_i) refer to a plaintext probability, ciphertext frequency from column 1 (respectively 2) respectively. We will assume in all our LAAs that the adversary knows what c_0 and d_0 are (one way of obtaining this knowledge is via observing past Complete queries made on different columns, but in the same table).

	Single Complete	Single Incomplete	Double Complete	Double Incomplete
Input (auxiliary)	$\{a_1..a_n\}$	$\{a_1..a_n\}$	$\{a_1..a_n\}, \{b_1..b_n\}$	$\{a_1..a_n\}, \{b_1..b_n\}$
Input (ciphertext)	$\{c_1..c_n\}$	$\{c_0..c_m\}$	$\{c_1..c_n\}, \{d_1..d_n\}$	$\{c_0..c_m\}, \{d_0..d_m\}$
Output	$f : [1, n] \mapsto [1, n]$	$f : [1, n] \mapsto [0, m]$	$f : [1, n] \mapsto [1, n]$	$f : [1, n] \mapsto [1, m + 2]$
Optimality	Optimal	Optimal	Optimal	Heuristic, NP-Hard
Condition	All attributes SELECTed	Some attributes not SELECTed	All rows returned in JOIN	Some rows not returned in JOIN

	Single Column	Double Column
Complete	Frequency Analysis	Graph Matching
Incomplete	Dynamic Programming	Probability Estimation Gradient Descent (Single- and Multi-step) Genetic Algorithm

Table 1. LAA scenarios (top) and attacks for each scenario (bottom). Note that for Incomplete scenarios, $m < n$.

3.2 LAA Implementation and Setup

In Figure 1, our adversary as previously mentioned is a man-in-the-middle adversary and sniffs packets exchanged by the AWS server and client, which we implemented with Wireshark and pyshark. The adversary is able to see all communications between the server and client, including the database initialisation, making it effectively equivalent to a server adversary.

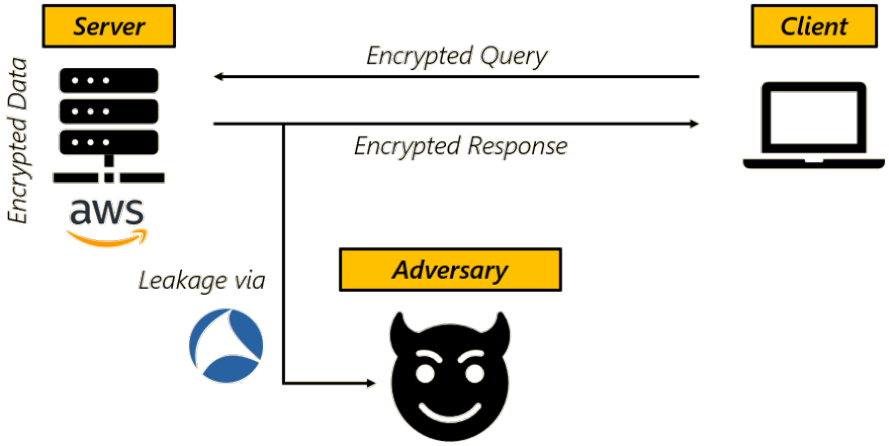

Fig. 1. A diagram of our experimental setup

For every attack, we ran it on 1000 rows on the system we implemented, together with Wireshark as a proof-of-concept. Then we simulated larger scale attacks on 1 million rows on leakage computed directly from the raw data. For Single Column Complete, we used the Ohio (ciphertext) and Florida (auxiliary) Voter Registry and the 2018 (ciphertext) and 2019 (auxiliary) HCUP NIS as our datasets. For the other scenarios, we used synthetic data modelled

after realistic data distributions (the Zipfian distribution). Details on data generation are found in Appendix C. For Incomplete scenarios, 10% of the types of ciphertexts were removed from the column(s) by unweighted random sampling. Future work could explore different percentages of ciphertexts removed and different methods for sampling removed ciphertexts, such as random sampling weighted by frequency.

Next, we will provide a brief summary of the various LAAs we came up with and implemented. We mention only the key intuition; for more detailed explanations, sub-algorithms used, proofs and pseudocode, refer to Appendix D.

3.3 Single Column, Complete

FREQUENCY ANALYSIS This attack is run when all possible SELECT queries have been run on a column. Checking that this condition is satisfied requires the total length of the table that the query was run on. This can be obtained if there is only 1 table by looking at the length of $\mathbf{M}$. Such conditions are likely to be satisfied by columns with a smaller number of possible values such as gender or race. The optimal mapping simply maps the highest a_i to the highest c_i, the second highest a_i to the second highest c_i and so on. The time complexity is $O(n \log n)$.

3.4 Single Column, Incomplete

DYNAMIC PROGRAMMING The intuition remains the same as frequency analysis. Let "largest" refer to the highest probability or the highest frequency. We consider the largest unmapped plaintext. We either map it to the largest unmapped ciphertext, or to c_0. Then, we consider the next largest unmapped plaintext and ciphertext. This constitutes a smaller but identical subproblem, thus we use dynamic programming to solve this. When all ciphertexts are mapped, all remaining plaintexts are mapped to c_0. The time complexity is $O(nm10^{\epsilon})$, where $\epsilon =$ number of decimal places the probabilities are rounded to. In our experiments, $\epsilon = 3$.

3.5 Double Column, Complete

GRAPH MATCHING Now, we consider the scenario of a JOIN query where all the rows in each table are returned at least once in the JOIN query. To check that this condition is satisfied, there can be at most 2 tables in the database. The first step is to convert the pairs of rows in the leakage into ciphertext frequencies. Since $\mathsf{Pr}[f] = \prod_i a_i^{c_{f(i)}} b_i^{d_{f(i)}} = \exp\left(\sum_i c_{f(i)} \log a_i + d_{f(i)} \log b_i\right)$, we represent each summand as an edge in a complete bipartite graph, then run Hungarian Algorithm to compute an optimal matching in $O(n^3)$. By the Hungarian Algorithm [1] and the monotonicity of $\log$, this LAA is optimal too.

3.6 Double Column, Incomplete

Now, we will discuss the scenario of JOIN queries where $c_0, d_0 > 0$ (i.e. there are rows in either table not included in the JOIN). We define n as the number of plaintexts and m as the

number of revealed ciphertexts. In addition, for a plaintext i that maps to c_0 or d_0, $f(i) = m+1$ or $f(i) = m+2$ respectively. In this scenario, the probability function for a mapping f is

$$\mathsf{Pr}[f] = \left(\prod_{f(i) \leq m} a_i^{c_{f(i)}} b_i^{d_{f(i)}} \right) \left(\sum_{f(i)=m+1} a_i \right)^{c_0} \left(\sum_{f(i)=m+2} b_i \right)^{d_0}$$

This probability can be broken into 2 parts. The first multiplicand (representing plaintexts in the JOIN, call it set X) can be maximised using the Hungarian Algorithm. The second and third multiplicands (representing those not in the JOIN, call it set Y) can be maximised using a new algorithm, the Partition Optimisation Algorithm. Thus, if we can determine which plaintexts are in the JOIN and which are not, we can solve each portion optimally. However, we do not believe that we can do so optimally, because the Double Column, Incomplete scenario is NP-hard [1]. Next, we present 3 LAAs to tackle this problem.

PROBABILITY ESTIMATION One possible method is to estimate the probability that a given plaintext i is in Y. This probability $\mathsf{Pr}[i]$ is given by

$$\mathsf{Pr}[i] = (1 - a_i)^{N_1} + (1 - b_i)^{N_2} - (1 - a_i)^{N_1}(1 - b_i)^{N_2}$$

where N_1, N_2 are the total number of rows in column 1 and 2 respectively. Since we know there are $n - m$ plaintexts in Y, we can take the top $n - m$ plaintexts with the highest $\mathsf{Pr}[i]$ and place them in Y. However, this is suboptimal since this assumes the probability of a given plaintext i being in Y is independent of any other plaintext. Of course, this is not true, but this assumption is necessary to ensure the algorithm is efficient. We also ran this attack without assumption on knowledge of c_0 and d_0, i.e. assuming $c_0 = d_0$.

GRADIENT DESCENT Since we know that the Probability Estimation LAA is suboptimal, we will now try and improve it by making small changes to the solutions and looking at what leads to the greatest improvement in the probability. We start with a random mapping, then consider all possible single-swaps of 2 individual plaintext mappings. We take the best swap and apply it. This process is repeated until all swaps decrease $\mathsf{Pr}[f]$. We also implemented a multi-step version, where at each step, the algorithm can perform any number of non-overlapping swaps. While gradient descent is intuitive, one pitfall is that it easily gets stuck in a local optimum.

GENETIC ALGORITHM To resolve this issue, we now take inspiration from natural selection and attempt to use a genetic algorithm to solve this problem. We maintain a population of several potential mappings. At each iteration, we randomly mutate them (via performing some swaps) to spawn more mappings. Then, we pick some of the mappings with highest $\mathsf{Pr}[f]$, but also keeping some poorer ones to maintain population diversity. The diversity is what increases the likelihood of some mappings escaping a local optimum. Its potential to jump over local optima is its key difference and improvement from Gradient Descent.

[1] A proof of this is currently under submission which reduces the problem from Set Partition, known to be NP-complete.

4 Results

4.1 Overview

In order to characterise the performance of our attacks, we make use of v-score and r-score. The v-score measures the proportion of ciphertexts that the attack correctly matched while the r-score measures the proportion of rows that the attack correctly matched. These two measures are different as each value has a different number of rows associated with it. For the Incomplete scenarios, we only consider known ciphertext frequencies, i.e. we ignore c_0 and d_0. Our raw results data and detailed formula for v- and r-scores are found in Appendix E.

In general, we noticed that our attacks performed better the lower the number of distinct ciphertexts. Intuitively, this makes sense because a smaller number of ciphertexts means their frequencies are more distinct, leading to the algorithms being able to distinguish them more easily. The characteristic frequencies are also more visible after introducing random noise during auxiliary data generation, due to their large inherent differences in value.

Another overall trend we observed is that performance worsens as the percentage error introduced during generation of auxiliary data increases, which is expected because the auxiliary data becomes a less and less accurate reflection of the actual data.

Overall, the runtime of the attacks were satisfactory, with the longest attack (genetic algorithm) taking an hour to terminate. We consider this reasonable given that an adversary would likely have sufficient time and that breaching databases should intuitively take some effort.

4.2 Overall Statistics

At a high level, all of our attacks worked. Refer to Table 2, where we used synthetic data with 1 million rows and 50 ciphertexts, with $e = 5\%$ for auxiliary data generation. Table 2 shows the performances of each attack across all distribution pairs.

4.3 Single Column, Complete

The first attack that we ran is the frequency analysis attack. In accordance with the HCUP Data Use Agreement, we ran the the HCUP datasets locally and the Florida and Ohio datasets online between our computers and AWS. We now summarise our results.

For columns with a small number of possible values, the difference in frequencies between ciphertexts is more prominent and thus easily picked up by frequency analysis, thus the attack performs well. In fact, it obtained 100% accuracy for gender and race (HCUP) columns.

For columns with more varied values, the performance decreases significantly. It was only able to correctly match the few most frequent values, along with some lucky guesses of less frequent ones. Specifically, for the first name column, it correctly matched the most common name "Michael" and a few other less common ones, thus explaining its low v-score of 0.239% but a comparatively higher r-score of 8.46%.

Scenario	LAA	Minimum		Maximum		Average	
		v-score	r-score	v-score	r-score	v-score	r-score
SC	Frequency Analysis	0.340	0.285	0.760	0.913	0.654	0.677
SI	Dynamic Programming	0.338	0.333	0.693	0.776	0.471	0.561
DC	Graph Matching	0.676	0.508	1.0	1.0	0.852	0.837
DI	Probability Estimation	0.399	0.400	0.695	0.988	0.538	0.659
DI	Probability Estimation, $c_0 = d_0$	0.211	0.203	0.646	0.979	0.472	0.613
DI	Single-step Gradient Descent	0.439	0.447	0.991	0.998	0.782	0.772
DI	Multi-step Gradient Descent	0.650	0.459	0.998	1.0	0.823	0.804
DI	Genetic Algorithm	0.621	0.459	0.963	0.999	0.826	0.849

Table 2. Scenario, worst, best and average performance of each attack, taken over 10 runs on each distribution, when all attacks are run on comparable synthetic data. S = Single Column, D = Double Column, C = Complete, I = Incomplete

However, the power of this attack cannot be underestimated: 8.46% of the 1048575 rows of the 2018 HCUP dataset corresponds to more than 88700 entries, a scary amount of leaked information if this attack was used for malicious intent.

4.4 Single Column, Incomplete

There was no significant trend to analyse, other than the expected decreasing performance as error and number of ciphertext increases. Similar to the Single Column, Complete scenario, we observed that compared to other distributions, Zipfian distributions have a lower v-score, but a much higher r-score. Due to its nature, the most common ciphertexts are very frequent, thus by only matching those correctly, a small proportion of values are correct but they make up a large percentage of the rows. Overall, the performance is very commendable, with the worst r-score being 1.1%, and several r-scores are even more than 50%. With a million rows in the datasets, the prowess of this attack cannot be underestimated.

4.5 Double Column, Complete

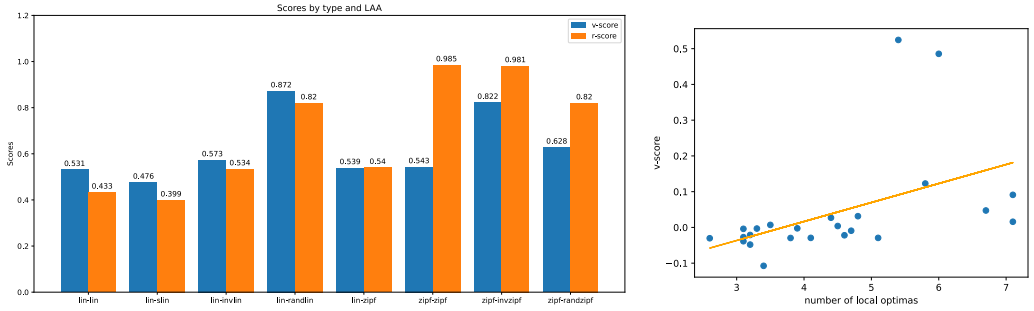

Fig. 2. Bar graph showing the performance of the graph matching LAA for varying distributions with a 5% error in the auxiliary data (left), and a graph of how much genetic algorithms outperform single-step gradient descent in v-score against number of local optima (blue points are experimental results, orange line is the best-fit line) (right).

After running the graph matching LAA on the synthetic data, we can plot the performance of the attack for various distributions in Figure 2 (left). We notice that inversely correlated distributions such as linear - inverse linear and uncorrelated distributions such as linear - random linear perform better than correlated distributions such as linear - linear or Zipfian - Zipfian. This is likely because for such distributions, ciphertexts that do not occur frequently in one column are most likely paired with ciphertexts that occur frequently in the other column. This reduces the impact of the error in the auxiliary data since there will be a large number of occurrences of the plaintexts in at least one column. However, this is not the case for correlated distributions where ciphertexts that occur infrequently in one column are paired with other ciphertexts that also occur infrequently in the other column.

4.6 Double Column, Incomplete

First, we compare the performances of each algorithm for varying amounts of error in the auxiliary data in Figure 3. For probability estimation, there was no significant difference between assuming $c_0 = d_0$ and not ($c_d = d_0$ only performed 6% worse on average), thus we analyse them as one attack.

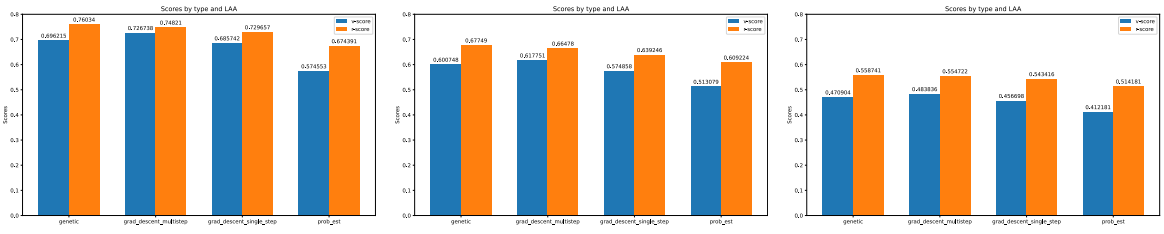

Fig. 3. A bar graph showing the average v-score and r-score of each attack for varying errors (5%, 10%, 20%) in the auxiliary data. From left to right: genetic algorithm, multi-step gradient descent, single-step gradient descent, probability estimation. Blue = v-score, orange = r-score.

We see that both genetic algorithms and gradient descent outperform the probability estimation LAA, since probability estimation can only make one sub-optimal guess while the other attacks can refine their guesses over multiple iterations.

In addition, multi-step gradient descent outperforms single-step gradient descent, most likely because the multi-step algorithm is able to make more overlapping swaps between iterations, making it more likely to find the global maxima. We can also see this in how much genetic algorithms outperform single-step gradient descent. To find the number of local optima, we start the gradient descent algorithm from a large number of random mappings and count the number of unique mappings that result from this. Plotting how much the genetic algorithms outperforms single-step gradient descent in v-score against the number of local optima in Figure 2 (right), we see an increasing trend. This validates our hypothesis that the poor performance of gradient descent is caused by the algorithm getting stuck in local optima.

Furthermore, we notice that genetic algorithms provide comparable performance to multi-step gradient descent. We hypothesise that this is because genetic algorithms are more likely than multi-step gradient descent to find the global optima. However, gradient descent provide more directed optimisation than genetic algorithms which optimise the guesses stochastically.

Thus, for attackers, the recommended approach is to run both the genetic algorithm LAA and the multi-step gradient descent LAA and pick the mapping with the highest probability.

5 Conclusion

In conclusion, we have implemented on Amazon Web Services and benchmarked a structured encryption scheme for SQL databases using multimap encryption that supports both SELECT and JOIN and is easily extendable to other types of precomputed SQL queries. Through analysis of our leakage profile, we have shown that our scheme is more secure than past schemes such as CryptDB [11]. We have also explored a wide variety of LAAs on our scheme by exploring all 4 posible scenarios on information availability, including genetic algorithms and gradient descent, and analysed their performance. Such improvements in structured encryption schemes will help us to better guard against increasingly common cyberattacks on databases and safeguard sensitive data and our online privacy. Our attacks also teach attackers how to better attack such encryption schemes to maximise the information they can gain from simply looking at the encrypted data. In the future, we could extend our work to support even more SQL queries, such as recursive queries, and look into potential improvements of our system to better safeguard against the LAAs mentioned in this report.

6 Acknowledgements

We would like to thank our mentors, Dr Ruth Ng and Mr Ryan Seah for their invaluable guidance and support in the course of this project. We would also like to thank Mr Alexander Hoover, Mr Daren Khu, Mr Ng Wei Cheng and Mr Quek Yuxuan for their support.

References

1. Hungarian algorithm, Oct 2022.
2. David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit S Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Dynamic searchable encryption in very-large databases: data structures and implementation. 14:23–26, 2014.
3. David Cash, Ruth Ng, and Adam Rivkin. Improved structured encryption for sql databases via hybrid indexing. pages 480–510, 2021.
4. Melissa Chase and Seny Kamara. Structured encryption and controlled disclosure. *Advances in Cryptology-ASIACRYPT*, pages 577–594, 12 2010.
5. Valentina Ciriani, Sabrina De Capitani Di Vimercati, Sara Foresti, Sushil Jajodia, Stefano Paraboschi, and Pierangela Samarati. Keep a few: Outsourcing data while maintaining confidentiality. pages 440–455, 2009.
6. Ernesto Damiani, S De Capitani Vimercati, Sushil Jajodia, Stefano Paraboschi, and Pierangela Samarati. Balancing confidentiality and efficiency in untrusted relational dbmss. pages 93–102, 2003.
7. Sergei Evdokimov and Oliver Günther. Encryption techniques for secure database outsourcing. pages 327–342, 2007.
8. Paul Grubbs, Kevin Sekniqi, Vincent Bindschaedler, Muhammad Naveed, and Thomas Ristenpart. Leakage-abuse attacks against order-revealing encryption. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 655–672. IEEE, 2017.
9. Seny Kamara and Tarik Moataz. Encrypted multi-maps with computationally-secure leakage. *IACR Cryptol. ePrint Arch.*, 2018:978, 2018.
10. Seny Kamara and Tarik Moataz. Sql on structurally-encrypted databases. pages 149–180, 2018.
11. Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: A practical encrypted relational dbms. October 2011.

Appendix

A Structured Encryption

A.1 Basic Definitions

DATA STRUCTURES. A data type DT is a type of data structure. It defines three things:

- A set of data elements $\mathsf{DT.DS}$
- A query set $\mathsf{DT.QS}$
- A deterministic query evaluation algorithm defining how data is queried of the form $\mathsf{DT.Qry} : \mathsf{DT.DS} \times \mathsf{DT.QS} \rightarrow \{0,1\}^*$.

STRUCTURED ENCRYPTION. A structured encryption (StE) scheme is a cryptographic primitive defined for a particular data type. Given any StE scheme StE for data type DT, it should define:

- A randomized encryption algorithm $\mathsf{StE.Enc}$ that takes as input a data element $DS \in \mathsf{DT.DS}$ and returns a client state, $St \in \{0,1\}^*$, and an encrypted data structure $EDS \in \{0,1\}^*$.
- A search protocol $\mathsf{StE.Srch}$ where the client input is its state and a query $q \in \mathsf{DT.QS}$, and the server input is the encrypted data structure. The client output is an updated state. The server output is an updated encrypted data structure and a ciphertext.
- A deterministic decryption algorithm $\mathsf{StE.Dec}$ that takes as input the client state and ciphertext, and returns the query output.

We say that a structured encryption scheme is correct if the query output is $\mathsf{DT.Qry}(DS, q)$. We say that an StE scheme is "response revealing" if the ciphertext is also equal to the query output. Note that in this case, $\mathsf{StE.Dec}(K, C) = C$ for all K and C. We refer to StE schemes where this is not the case as "response hiding".

A.2 Multimap Encryption Scheme

In order to build our scheme for structured encryption of SQL tables, we first need to build a scheme for multimaps to store the precomputed queries. We will adopt the multimap scheme used by Cash et al. [2].

There are 2 different schemes for multimap encryption that we need - response hiding and response revealing. Response hiding prevents the server from being able to access the decrypted values while response revealing allows the server to access the decrypted values when they are queried. In these schemes, the client has access to randomly generated keys K_f, K_s.

For response hiding, the each label l of the multimap is hashed using K_f to obtain a new key K. For each value v in the multimap associated with l, the index i of value v is hashed with key K. This becomes the key associated with v in the encrypted multimap. In doing so, we

can prevent the server from being able to know the number of values associated with label l. Finally, we encrypt v with K_s and put it into the multimap. For decryption, the hashed label K is sent to the server which increments i starting from 0 and generates the keys of the encrypted multimap. The encrypted values are sent back to the client, who can decrypt them with key K_s.

For response revealing, the hashed label l with key K_f is now known as K_1. A second hashed label K_2 is obtained by hashing l with key K_2. Instead of encrypting the values with K_s, they are now encrypted with key K_2 which we will send to the server at decryption time. This allows decryption to be run on the server while ensuring security of the values that were not queried since they were encrypted with a different K_2.

MULTIMAP DATA STRUCTURE The multimap data type MmDt is defined as follows:

- MmDt defines a label length $\mathsf{MmDt.lLen}$ and value length $\mathsf{MmDt.vLen}$. These in turn define the label set (i.e. $\{0,1\}^{\mathsf{MmDt.lLen}}$) and value set (i.e. $\{0,1\}^{\mathsf{MmDt.vLen}}$).
- Each data element is a multimap. A multimap $\mathbf{M}$ is a mapping from labels $\ell \in \{0,1\}^{\mathsf{MmDt.lLen}}$ to sets of values $\mathbf{M}[\ell] \subseteq \{0,1\}^{\mathsf{MmDt.vLen}}$. In other words,

$$\mathsf{MmDt.DS} = \{\mathbf{M} : \forall \ell \in \{0,1\}^{\mathsf{MmDt.lLen}}, \mathbf{M}[\ell] \subseteq \{0,1\}^{\mathsf{MmDt.vLen}}\}.$$

  Note that when multimaps are initialised, all ℓ map to $\perp$ so $\mathbf{M}[\ell] = \perp$.
- The query set is $\mathsf{MmDt.QS} = \{0,1\}^{\mathsf{MmDt.lLen}}$.
- The query evaluation algorithm is defined as $\mathsf{MmDt.Qry}(\mathbf{M}, \ell) = \mathbf{M}[\ell]$.

MULTIMAP ENCRYPTION SCHEME We define two multimap encryption schemes, RH and RR that are response hiding and response revealing respectively. They both make use of a function family F and symmetric encryption SE. We will assume that RH and RR return the items in the same order that they were inserted.

Alg $\mathsf{RH.Enc}(\mathbf{M})$
Initialize empty dictionary $\mathbf{D}$
$K_f \leftarrow\!\!\$\ \mathsf{F.KS}$; $K_s \leftarrow\!\!\$\ \mathsf{SE.KS}$
For $\ell \in \{0,1\}^{\mathsf{MmDt.lLen}}$ do
  $K \leftarrow \mathsf{F.Ev}(K_f, \ell)$; $i \leftarrow 0$
  For $v \in \mathbf{M}[\ell]$ do
    $x \leftarrow \mathsf{F.Ev}(K, i)$; $y \leftarrow \mathsf{SE.Enc}(K_s, v)$
    $\mathbf{D}[x] \leftarrow y$; $i \leftarrow i + 1$
  End for
End for
Return $((K_f, K_s), \mathbf{D})$

Protocol $\mathsf{RH.Srch}\big(((K_f, K_s), \ell)\ ;\ \mathbf{D}\big)$

$K \leftarrow \mathsf{F.Ev}(K_f, \ell)$	$C \leftarrow \emptyset$; $i \leftarrow 0$
Send K	While $\mathbf{D}[\mathsf{F.Ev}(K, i)] \neq \perp$ do
	$C \leftarrow C \cup \{\mathbf{D}[\mathsf{F.Ev}(K, i)]\}$; $i \leftarrow i + 1$
	End while
Return (K_f, K_s)	Return $(\mathbf{D}, C)$

Alg $\mathsf{RH.Dec}((K_f, K_s), C)$
$S \leftarrow \emptyset$
For $c \in C$ do
  $v \leftarrow \mathsf{SE.Dec}(K_s, c)$
End for
Return S

Alg $\mathsf{RR.Enc}(\mathbf{M})$
Initialize empty dictionary $\mathbf{D}$
$K_f \leftarrow\!\!\$\ \mathsf{F.KS}$; $K_s \leftarrow\!\!\$\ \mathsf{SE.KS}$
For $\ell \in \{0,1\}^{\mathsf{MmDt.lLen}}$ do
  $K_1 \leftarrow \mathsf{F.Ev}(K_f, \ell)$; $K_2 \leftarrow \mathsf{F.Ev}(K_s, \ell)$; $i \leftarrow 0$
  For $v \in \mathbf{M}[\ell]$ do
    $x \leftarrow \mathsf{F.Ev}(K_1, i)$; $y \leftarrow \mathsf{SE.Enc}(K_2, v)$
    $\mathbf{D}[x] \leftarrow y$; $i \leftarrow i + 1$
  End for
End for
Return $((K_f, K_s), \mathbf{D})$

Protocol $\mathsf{RR.Srch}\big(((K_f, K_s), \ell)\ ;\ \mathbf{D}\big)$

$K_1 \leftarrow \mathsf{F.Ev}(K_f, \ell)$; $K_2 \leftarrow \mathsf{F.Ev}(K_s, \ell)$	$C \leftarrow \emptyset$; $i \leftarrow 0$
Send (K_1, K_2)	While $\mathbf{D}[\mathsf{F.Ev}(K, i)] \neq \perp$ do
	$z \leftarrow \mathbf{D}[\mathsf{F.Ev}(K_1, i)]$; $C \leftarrow C \cup \{\mathsf{SE.Dec}(K_2, z)\}$; $i \leftarrow i + 1$
	End while
Return (K_f, K_s)	Return $(\mathbf{D}, C)$

Fig. 4. Definitions of MME schemes RH and RR.

A.3 SQL Database Encryption Scheme

Now, let us outline our scheme for encryption of SQL databases, making use of structured encryption of multimaps.

As previously mentioned, to ensure queries can be executed efficiently, we will fully precompute all SELECT and JOIN queries on the SQL database. The precomputed queries can be represented in the form of the multimaps shown in Figure 5. One multimap $\mathbf{N}$ stores the

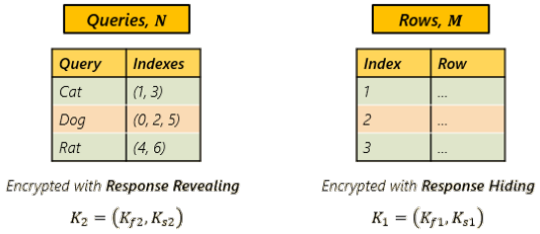

Fig. 5. Diagram showing our SQL scheme

mapping from the query to the indexes of the relevant rows while another multimap **M** maps each row index to the encrypted row contents in the SQL table. We store the queries and rows separately to avoid invoking an additional memory cost by storing some rows repeatedly since the rows will take up more space in memory than the indexes.

At the start, the client will randomly generate 4 keys $(K_{f1}, K_{s1}), (K_{f2}, K_{s2})$ and we define K_1 as (K_{f1}, K_{s1}) and K_2 as (K_{f2}, K_{s2}). They are assigned as shown in Figure 5. After the precomputed queries have been loaded into **M** and **N**, they are encrypted using the keys with response hiding and response revealing respectively. Before putting in the indexes into **N**, they are also hashed with K_1, allowing them to be used to query **M** and concealing the order of rows in the table. This ensures that the server will only have access to the hashed indexes (which it needs obtain the encrypted rows) but will not have access to the decrypted rows.

SQL TABLE First, we define the SQL table object.

- Table defines a length for an attribute of the target column Table.xLen and a total length for all other attributes Table.yLen.
- Each Table defines the number of attributes that can be queried Table.num.
- Each data element is a set of tuples. Within the list of tuples **T**, each tuple has the target attributes $x_i \in \{0,1\}^{\mathsf{Table.xLen}}, i \in [1, \mathsf{Table.num}]$ as the first Table.num elements and all other attributes as the last element $y \in \{0,1\}^*$.

$$\mathsf{Table.DS} = \{\mathbf{T} : \mathbf{T} = \{(x_1, \ldots, x_{\mathsf{Table.num}}, y) : x \in \{0,1\}^{\mathsf{Table.xLen}}, y \in \{0,1\}^{\mathsf{Table.yLen}}\}\}$$

- For a given row r of **T**, we define $r[i]$ to refer to the value of attribute i stored in that row.

SQL DATABASE Now, we can define the SQL database data structure. We target queries of the type SELECT $*$ FROM tbl1 JOIN tbl2 WHERE attr1 $=$ attr2.

- Each data element is a tuple. The first element of the tuple is a set that contains all valid JOIN queries. The second element of the tuple is a set of tuples where the first element is a binary string and the second element is a table.

$$\mathsf{DbDt.DS} = \{\mathbf{DB} : \mathbf{DB} = (\mathbf{JC}, \{(i, \mathbf{T}) : i \in \{0,1\}^*, \mathbf{T} \in \mathsf{Table.DS}\})\}$$

- We assume there are primary keys for each table so that each row in the table is unique.

– The query set is $\{(i, q) : i \in [1, 2], q \in \mathbf{Q}_i\}$, where

$$\mathbf{Q}_1 = \{(j, attr, val) : j, val \in \{0,1\}^*, attr \in \mathbb{Z}^+\}$$
$$\mathbf{Q}_2 = \{(j_1, j_2, attr_1, attr_2) : j_1, j_2 \in \{0,1\}^*, attr_1, attr_2 \in \mathbb{Z}^+\}$$

– The query evaluation algorithm is defined in Figure 6

Alg DbDt.Qry$(i, q, \mathbf{DB})$
$(\mathbf{JC}, \{(x, \mathbf{T}_x) \dots\}) \leftarrow \mathbf{DB}$
If $i == 1$ then
  $(j, attr, val) \leftarrow q$
  Return $\{r : r[attr] = val, r \in \mathbf{T}_j\}$
else if $i == 2$ then
  $(j, k, attr_1, attr_2) \leftarrow q$
  If $q \notin \mathbf{JC}$ then return $\perp$
  Return $\{r||s : r[attr_1] = s[attr_2], r \in \mathbf{T}_j, s \in \mathbf{T}_k\}$
End if

Fig. 6. Definition of the query algorithm

SQL DATABASE ENCRYPTION SCHEME Now, we define the encryption scheme for SQL databases considering with queries of the following forms:

– SELECT $*$ FROM tbl WHERE attribute $=$ VALUE
– SELECT $*$ FROM tbl1 JOIN tbl2 WHERE attr1 $=$ attr2.
– The key for the encryption is defined as DbDt.KS $=$ RH.KS $\times$ RR.KS.
– It makes use of the structured encryption schemes for multimaps.

Alg $\mathsf{DbDt.Enc}((K_1, K_2), \mathbf{DB})$

$\{K_f, K_s\} \leftarrow K_1$
$(\mathbf{JC}, \{(id, \mathbf{T}_{id})\}) \leftarrow \mathbf{DB}$
For $(id, \mathbf{T}_{id}) \in \mathbf{DB}$ do
  For $\{(x^i_1, \ldots, x^i_{\mathbf{T}_{id}.\mathsf{num}}, y_i)\} \in \mathbf{T}_{id}$ do
    $\mathbf{M}[(id, i)] = (x^i_1, \ldots, x^i_{\mathbf{T}_{id}.\mathsf{num}}, y_i)$
    For $j \in 1 \ldots \mathbf{T}_{id}.\mathsf{num}$ do
      $\mathbf{N}[(id, j, x^i_j)] = \{\mathsf{F.Ev}(K_f, (id, i))\} \,||\, \mathbf{N}[(id, j, x^i_j)]$
    End for
  End for
End for
For $(i, j, p, q) \in \mathbf{JC}$ do
  For $\{(x^k_1, \ldots, x^k_{\mathbf{T}_i.\mathsf{num}}, y_k)\} \in \mathbf{T}_i$, $\{(w^l_1, \ldots, w^l_{\mathbf{T}_j.\mathsf{num}}, z_l)\} \in \mathbf{T}_j$ do
    If $x^k_p = w^l_q$ then
      $\mathbf{N}[(i, j, p, q)] \leftarrow \{\mathsf{F.Ev}(K_f, k))\} \,||\, \mathbf{N}[(i, j, p, q)]$
      $\mathbf{N}[(i, j, p, q)] \leftarrow \{\mathsf{F.Ev}(K_f, l))\} \,||\, \mathbf{N}[(i, j, p, q)]$
    End if
End for
$\mathbf{ED}_1 \leftarrow \mathsf{RH.Enc}(K_1, \mathbf{M})$
$\mathbf{ED}_2 \leftarrow \mathsf{RR.Enc}(K_2, \mathbf{N})$
Return $\mathbf{ED}_1, \mathbf{ED}_2$

Protocol $\mathsf{DbDt.Srch}\big((K_1, K_2), i, q\,;\ \mathbf{ED}_1, \mathbf{ED}_2\big)$

$(K_f, K_s) \leftarrow K_1$
$(\mathbf{JC}, \mathbf{DB}_1) \leftarrow \mathbf{DB}$
If $i = 1$ then
  $id, attr, val \leftarrow q$
  If $(id, \mathbf{T}_{id}) \notin \mathbf{DB}_1$ then
    Return $\perp$
  End if
else if $i = 2$ then
  If $q \notin \mathbf{JC}$ then
    Return $\perp$
  End if
End if
$K \leftarrow \mathsf{F.Ev}(K_f, q)$
Send K

$i \leftarrow 0$
While $\mathbf{ED}_2[\mathsf{F.Ev}(K, i)] \neq \perp$ do
$z_i \leftarrow \mathsf{SE.Dec}(K_2, \mathbf{ED}_2[\mathsf{F.Ev}(K, i)])\,;\, i \leftarrow i + 1$
End while
For $i \in 1 \ldots n$ do
$r_i \leftarrow \mathbf{ED}_1[z_i]$
End for
Send $\{r_1, \ldots, r_n\}$

For $j \in 1 \ldots n$ do
  If $i = 1$ then $C_j \leftarrow \mathsf{SE.Dec}(K_1, r_j)$
  else if $i = 2$ then
    $C^{(j-1) \bmod 2} \leftarrow \mathsf{SE.Dec}(K_1, r_j)$
    If $j \bmod 2 = 1$ then $C_{\frac{j}{2}} \leftarrow C^0 \,||\, C^1$
  End if
End for
Return $\{C_1, \ldots, C_n\}$

Fig. 7. Definition of SQL encryption scheme DbDt.

B Query Information

QUERY TYPE While it is not possible to be completely certain what is the type of query that is made since the query string is hashed, it is possible to guess what type of query is made using other information

- There are a few important differences between SELECT and JOIN queries.
- JOIN queries always return rows in pairs, thus the total number of rows returned must be even. This is not the case for SELECT queries.
- JOIN queries will also mostly always return duplicate rows. This will happen as long as there is more than 2 rows in the same column with the same value for the target attribute. On the other hand, SELECT queries always return unique rows.
- Thus, by checking if the number of rows returned is even / odd and if the rows are unique, the type of query made can be identified with a high degree of accuracy.

JOIN COLUMNS Using the number of rows returned from a JOIN query, it may be possible guess what columns the JOIN was run on.

- Let us define vectors a and b as the auxiliary data and c and d as the ciphertext frequencies for the first and second JOIN column respectively.
- c_0 and d_0 will represent the ciphertexts that did not appear in only the first or second column respectively. We will assume we know what these values are.
- Now, let us define

$$N_1 = \sum_i c_i, N_2 = \sum_i d_i$$

- Then, the approximate length of the JOIN query is

$$N = \sum_i N_1 N_2 a_i b_i$$

- If we find a JOIN query of similar length, it is probable that it correlates well with auxiliary data a and b.
- However, in reality, the adversary may not be aware of the value of c_0 and d_0 but is only aware of $c_0 + d_0$ as the other values in c and d are known and $N_1 + N_2$ is known.
- In this case, it is usually justified to assume that $c_0 = d_0$ as in real data, it is not likely that uncommon values are heavily skewed to either column.
- In many cases, c_0 and d_0 might also be quite small so may also be valid to ignore them.
- Finally, if the adversary is aware of a JOIN complete query that was run previously, N_1 and N_2 can be easily found so the exact value of c_0 and d_0 can be known.

We evaluated the accuracy of differentiating JOINs by length using the method above. We ran the attack on the synthetic data (Appendix C) with 1 million rows and attempted to differ-

entiate between 10 JOINs with the same distribution. We note that in real data, the distributions of different JOINs will likely differ and as a result, this attack will be even more accurate.

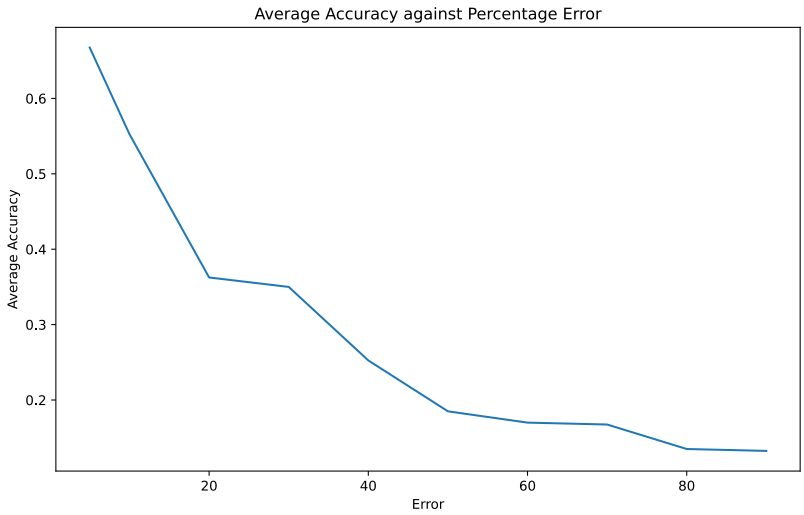

Fig. 8. A graph of the accuracy of JOIN identification against error in the auxiliary data

C Synthetic Data

The distributions used to generate the synthetic data are shown below. Random means that the order of ciphertexts was shuffled and inverse means that the order of the ciphertexts was inverted. For each pair of distributions, we generated 2 ciphertext columns from each distribution, with 1 million rows in each column. We generated a new pair of columns for each of the following values of distinct ciphertext numbers: 10, 50, 100, 200, 500. For every combination of (distribution, number of ciphertexts), we generated a pair of columns 10 times. Thus, all data we report on one combination of (distribution, number of ciphertexts) will be the average results on these 10 sets.

- Linear Distribution, Linear Distribution
- Linear Distribution, Slow Linear Distribution
- Linear Distribution, Inverse Linear Distribution
- Linear Distribution, Random Linear Distribution
- Zipfian Distribution, Linear Distribution
- Zipfian Distribution, Zipfian Distribution
- Zipfian Distribution, Inverse Zipfian Distribution
- Zipfian Distribution, Random Zipfian Distribution

The equations for the various distributions are shown below. $n = 1000000$ is the size of the distribution, i.e. the total number of rows.

- Linear: $\left\{\frac{2i}{n(n+1)}\right\}$
- Slow Linear: $\left\{\frac{1}{2n} + \frac{i}{n(n+1)}\right\}$
- Zipfian: $\left\{\frac{1/i}{\sum_{k=1}^{n} 1/k}\right\}$

To generate ciphertexts (and therefore their frequencies), we fixed $n = 1000000$, the number of rows to generate, then sampled n values from the probability distribution.

To generate the auxiliary probability distributions, we took the corresponding ciphertext distribution and introduced percentage errors of $e = 5$, 10 and 20. Specifically, for each probability x in a distribution, we scaled it to a random value between $(1 - \frac{e}{100})x$ and $(1 + \frac{e}{100})x$, then normalised the distribution to sum to 1. e represents the degree of random noise between the auxiliary data and ciphertexts, which corresponds to the variances in real-world data.

For Single Column, Incomplete, we simply looked at the pairs of columns as 2 individual columns.

D Leakage Abuse Attacks

D.1 Frequency Analysis

DESCRIPTION

– For this attack, we define the ciphertext frequencies as vector c and the auxiliary data as vector a. We take the auxiliary data a_i to mean that every ciphertext has a probability a_i of being associated with plaintext i.
– Now, we seek find a mapping f of plaintexts to ciphertexts that has the highest probability

$$\mathsf{Pr}[f] = \prod_i a_i^{c_{f(i)}}$$

– To optimise this probability, we sort a and c by value and pair them up accordingly such that the highest value in a is paired with the highest value in c, the 2nd highest value in a is paired with the 2nd highest value in c and so on.
– Since sorting is $O(n \log n)$, the time-complexity of this algorithm is also $O(n \log n)$. The pseudocode for this attack can be found in Figure 13.
– However, this attack requires that c is known. Thus, all possible SELECT queries on one column must have been run on only that column. In addition, there should be only 1 table so that this condition can be verified.

PROOF OF OPTIMALITY We use an exchange argument to prove the optimality of frequency analysis. Assume there are 2 plaintexts $a_i \geq a_j$ and 2 ciphertexts $c_k \geq c_l$, such that $f(i) = l$ and $f(j) = k$. Let $P = \mathsf{Pr}[f]$. Consider a mapping f' that is identical to f, except $f'(i) = k$ and $f'(j) = l$. Then

$$\mathsf{Pr}[f'] = P \times \frac{a_i^{c_k} \times a_j^{c_l}}{a_i^{c_l} \times a_j^{c_k}} = P \times \frac{a_i}{a_j}^{c_k - c_l} > P$$

Thus, while there is such a "mismatch", we should perform a swap. Since there can only be at most $\frac{N(N-1)}{2}$ mismatches initially, and every such swap reduces the number of mismatches by at least 1, there exists a finite sequences of swaps for every initial mapping f' to the algorithm's mapping f. Since $\mathsf{Pr}[f] > \mathsf{Pr}[f']$ for every f', the algorithm is optimal. ■

D.2 Dynamic Programming

Note that the probability of a mapping is

$$\mathsf{Pr}[f] = \left(\sum_{f(i)=0} a_i \right)^{c_0} \times \prod_{f(i)\neq 0} a_i^{c_{f(i)}}$$

First, we preprocess the input, $\{a_1..a_n\}$ and $\{c_0..c_m\}$. We define $b = \mathsf{sorted}(\{a_1..a_n\})$ and $d = \{c_0, \mathsf{sorted}(\{c_1..c_m\})\}$. Now, we consider the new mapping from plaintexts in b to ciphertexts

in d. We claim that in the optimal mapping f, either $f(n) = m$ or $f(n) = 0$. Assume otherwise, that $f(n) = i, 0 < i < m$. Let x be such that $f(x) = m$. Because the arrays are sorted and $\mathsf{Pr}[f]$ is defined similarly, we can apply the same exchange argument from frequency analysis (Appendix D.1) here. Thus, $\mathsf{Pr}[f(n) = m] > \mathsf{Pr}[f(n) = i]$, so we only need to consider $f(n) = m$ or $f(n) = 0$. Then, we can remove b_n (and d_m if $f(n) = m$) from consideration, and repeat the same process on $f(n-1)$. Due to this self-similar nature, we can thus define this as a Dynamic Programming problem, and solve it recursively. The base case would be when all ciphertexts have been mapped, in which we map all remaining plaintexts to d_0.

The state of the Dynamic Programming is $dp(x, y, \delta)$, which returns a tuple $(f, \mathsf{Pr}[f])$, where $f : \{1..x\} \mapsto \{0..y\}$ is a mapping of the first x plaintexts onto the first y ciphertexts and d_0, and δ = the sum of all previous a_i which was mapped to d_0. The answer we seek is then $dp(n, m, 0)$. For the transition, refer to the pseudocode in Appendix D.8. For our implementation, due to the probabilities being very small, we applied $\log$ to it during all our calculations. To reduce memory usage and speed up computation, we used a slightly different implementation: instead of storing and returning the probability every time, we maintained the optimal $f(n)$ mapping and used backtracking to reconstruct f afterwards.

Note that in order for this attack to work, δ needs to be discretised to a terminating decimal, because the attack relies on δ repeating and thus saving computation space. If, for example, no 2 subsets of a_i have the same sum, this attack would be identical to brute forcing all 2^n mappings.

D.3 Graph Matching

CIPHERTEXT FREQUENCIES Before running the attack, we need to convert the returned pairs into ciphertext frequencies.

- To do this, we first initialise 2 lists of sets, **A** and **B**.
- When we encounter a new pair (x, y) where x and y are not found in any set in **A** or **B**, a new set is created in **A** and **B** and x and y are added to the new sets.
- If the new pair (x, y) has only either x or y element not in **A** or **B**, then the missing element is added to the corresponding set.
- This is done until all pairs have been looped through. The ciphertext frequencies c, d can thus be obtained by counting the number of elements in each set in **A** and **B**.

Alg PairsToFrequencies$((x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n))$
For $i \in [1, n]$ do
  $C \leftarrow 0$
  For $j \in [1, |\mathbf{A}|]$ do
    If $x_i \in \mathbf{A}_j \cap y_i \notin \mathbf{B}_j$ do
      $C \leftarrow 1$
      $\mathbf{B}_j = \mathbf{B}_j \cup \{y_i\}$
      Break
    Else if $x_i \notin \mathbf{A}_j \cap y_i \in \mathbf{A}_j$ do
      $\mathbf{A}_j = \mathbf{A}_j \cup \{x_i\}$
      $C \leftarrow 1$
      Break
    End if
  End for
  If $C == 0$
    $\mathbf{A} = \mathbf{A} \parallel [\{x_i\}]$
    $\mathbf{B} = \mathbf{B} \parallel [\{y_i\}]$
  End if
Return $\{|\mathbf{A}_i| : i \in [1, |\mathbf{A}|]\}$

Fig. 9. Pseudocode for the algorithm to convert pairs to frequencies

GRAPH MATCHING Now, we will describe the attack. The idea is to convert this problem into that of optimising the sum of edge weights in a bipartite graph which can be solved using the Hungarian algorithm.

– In this attack, the probability of a given mapping f of plaintexts to ciphertexts is

$$\mathsf{Pr}[f] = \prod_i a_i^{c_{f(i)}} b_i^{d_{f(i)}}$$

– Taking the log of this probability, we can obtain

$$\mathsf{Pr}[f] = \sum_i c_{f(i)} \log a_i + d_{f(i)} \log b_i$$

– To solve this problem, we present the mapping as a complete bipartite graph with $2N$ nodes. We define the weights between nodes i, j of the graph as

$$w_{ij} = c_j \log a_i + d_j \log b_i$$

– Then, finding the mapping with the high probability then becomes finding a matching (an example shown below) in the bipartite graph with the greatest sum of edge weights.
– This is a well known problem and can be solved by the Hungariam Algorithm.

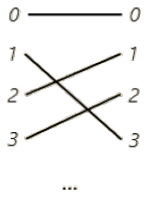

Fig. 10. A possible bipartite graph mapping

D.4 Partition Optimisation Algorithm

In this algorithm, we have two arrays of non-negative integers of length n, a and b. There are also 2 positive integers c and d. The partition optimisation algorithm seeks to find a set R of $\{1$ to n$\}$ such that the following is optimised.

$$S_1^c S_2^d = \left(\sum_{i \in R} a_i\right)^c \left(\sum_{i \notin R} b_i\right)^d$$

- Notice that the only thing that matters is the value of S_1 and S_2 and not which integers make up S_1 and S_2. Thus, in the worst case, we only need to search $(L+1)^2$ states.
- Thus, we will consider $L+1$ by $L+1$ grid and mark $(0,0)$ at the start. Now, we iterate through i from 1 to n and consider every marked square (x, y) and mark $(x + a_i, y)$ and $(x, y + b_i)$, then unmark (x, y). We also keep track of where $(x + a_i, y)$ and $(x, y + b_i)$ came from by keeping track of $path[(x+a_i, y)] = path[(x, y)]+A$, and $path[(x, y+b_i)] = path[(x, y)]+B$.
- After this, we can just look at every marked square (x, y) and find the square with the largest $x^c y^d$ and look at its path to reconstruct R.This algorithm runs in $O(nL^2)$ time and memory.
- However, we notice that for some i, if we have 2 marked squares (x, y) and (x', y') such that $x > x'$ and $y > y'$, we will never need to consider (x', y') as no matter what we do after this i, since using (x, y) will always result in a greater $S_1^c S_2^d$.
- Therefore, instead of a $L + 1$ by $L + 1$ grid, we only need $L + 1$ rows and 1 column (call this new grid G), where $G[x] =$ the marked square (x, y) with largest y. Then, at each i, we consider for each x, $(x + a_i, G[x])$ or $(x, G[x] + b_i)$. Then, we compare $(x + a_i, G[x + a_i])$ and $(x + a_i, G[x])$ and set $G[x + a_i]$ to the largest of those. Similar thing for $(x, G[x] + b_i)$, and for the path.

D.5 Probability Estimation

Here is a more detailed explanation of the calculation of $\mathsf{Pr}[\text{plaintext } i \text{ is in } Y]$.

$$\begin{aligned}
\mathsf{Pr}[i] &= \mathsf{Pr}[\text{plaintext } i \text{ appears in exactly one column}] + \mathsf{Pr}[\text{plaintext } i \text{ appears in neither column}] \\
&= \left(\left(1-(1-a_i)^{N_1}\right)(1-b_i)^{N_2} + (1-a_i)^{N_1}\left(1-(1-b_i)^{N_2}\right)\right) + \left((1-a_i)^{N_1}(1-b_i)^{N_2}\right) \\
&= \left((1-a_i)^{N_1} + (1-b_i)^{N_2} - 2(1-a_i)^{N_1}(1-b_i)^{N_2}\right) + \left((1-a_i)^{N_1}(1-b_i)^{N_2}\right) \\
&= (1-a_i)^{N_1} + (1-b_i)^{N_2} - (1-a_i)^{N_1}(1-b_i)^{N_2}
\end{aligned}$$

D.6 Gradient Descent

1. To get our initial guess, we assume every ciphertext found only in column 1 or found only in column 2 occurs the same number of times. We can then run the graph matching attack to find our initial mapping.
2. We can then proceed to try all possible swaps between **X** and **Y** and take the swap that results in the greatest improvement in the probability.
3. The Hungarian algorithm and Partition Optimisation algorithm are then re-run to obtain the new optimal mapping within **X** and **Y**.
4. This is done until no further improvement in the probability can be found.
5. We also formulate a variant of this single-step attack called "multi-step" gradient descent. Instead of making testing only 1 swap each time, we allow the algorithm to make multiple non-overlapping swaps. This is done using a modified Hungarian algorithm.

MODIFIED HUNGARIAN ALGORITHM The input to the Hungarian Algorithm is the following matrix where r_{ij} refers to the probability improvement obtained by swapping i and j.

$$\begin{bmatrix}
-\infty & \dots & -\infty & 0 & \dots & 0 \\
\vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
-\infty & \dots & -\infty & 0 & \dots & 0 \\
r_{ij} & \dots & r_{ij} & -\infty & \dots & -\infty \\
\vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
r_{ij} & \dots & r_{ij} & -\infty & \dots & -\infty
\end{bmatrix}$$

Fig. 11. Matrix to be solved by the modified Hungarian algorithm to make multiple non-overlapping swaps

D.7 Genetic Algorithm

1. We start by randomly generating a large number of possible X and Y. We call this our population, similar to a population of animals. We can then sort the list of possible subsets by their probability.

2. Now, to construct a new population, we randomly pick k subsets from the old population and the one with the highest probability is selected. This simulates survival of the fittest as subsets with a lower probability are less likely to be picked. We then make a random number of swaps between X and Y to simulate genetic mutations. These mutations help the algorithm to jump over local optima to find the global one.
3. 80% of the new population is filled this way. Another 10% is filled with the top 10% of the old population and the other 10% are random subsets in order to increase diversity.
4. We will repeat this process multiple times until the best probability no longer changes or we have exceeded a maximum number of iterations.

D.8 Pseudocode

Alg DPLAA($\{b_1..b_n\}, \{d_0..d_m\}$)
Initialise $\mathbb{D}, \mathbb{P}, f$
DP$(n, m, 0)$
$f(1 \leq i \leq n), \delta \leftarrow 0, 0$
While $\mathbb{P}[n, m, \delta] \neq \perp$ do
  If $\mathbb{P}[n, m, \delta] == (n-1, m-1, \delta)$, $f(n) = m$
  $(n, m, \delta) \leftarrow \mathbb{P}[n, m, \delta]$
End while
Return f

Alg DP(n, m, δ)
If $\mathbb{D}[n, m, \delta] \neq \perp$
  Return $\mathbb{D}[n, m, \delta]$
Else if $n < m$
  Return 0
Else if $m == 0$
  Return $\left(\delta + \sum_{i=1}^{n} b_i\right)^{d_0}$
Else
  $p_1 \leftarrow$ DP$(n-1, m-1, \delta) \times b_n^{d_m}$
  $p_2 \leftarrow$ DP$(n-1, m, \delta + b_n)$
  If $p_1 \geq p_2$
    $\mathbb{P}[n, m, \delta] \leftarrow (n-1, m-1, \delta)$
    Return $\mathbb{D}[n, m, \delta] \leftarrow p_1$
  Else
    $\mathbb{P}[n, m, \delta] \leftarrow (n-1, m, \delta + b_n)$
    Return $\mathbb{D}[n, m, \delta] \leftarrow p_2$
  End if
End if

Fig. 12. Pseudocode for Dynamic Programming LAA

Alg ProbabilityEstimation(a, b, c, d)

Initialise list $\mathbf{L} = [1 \dots n]$
Sort $\mathbf{L}$ by descending $\mathsf{Pr}[i]$
$a', b', c', d' \leftarrow a, b, c, d$ without top $n - m$ plaintexts in $\mathbf{L}$
$f \leftarrow \mathsf{GraphMatching}(a', b', c', d')$
$a'', b'' \leftarrow a, b$ with only top $n - m$ plaintexts in $\mathbf{L}$
$R \leftarrow \mathsf{PartitionOptimisation}(a'', b'', c_0, d_0)$
$f(i) \leftarrow m + 1$ for $i \in R$
$f(i) \leftarrow m + 2$ for $i \notin R \cap i \in \{\mathbf{L}_0, \mathbf{L}_1, \dots, \mathbf{L}_{n-m}\}$
Return f

Alg FrequencyAnalysis(a, c)

$b = \mathsf{Sort}(\{(i, a_i) : i \in [1, |a|]\})$
$d = \mathsf{Sort}(\{(j, c_j) : j \in [1, |c|]\})$
For $k \in [1, |c|]$ do
  $(i, a_i) \leftarrow b$
  $(j, c_j) \leftarrow d$
  $f(i) \leftarrow j$
End for
Return f

Alg GraphMatching(a, b, c, d)

Initialise matrix $\mathbf{W}$
$\mathbf{W}_{ij} \leftarrow c_j \log a_i + d_j \log b_i$
Return $\mathsf{HungarianAlgorithm}(\mathbf{W})$

Fig. 13. Pseudocode for, probability estimation (left), frequency analysis (top right) and graph matching (bottom right)

Alg Fitness$(a, b, c, d, (\mathbf{X}, \mathbf{Y}))$

$a', b', c', d' \leftarrow a, b, c, d$ with only plaintext i where $i \in \mathbf{X}$
$f \leftarrow \mathsf{GraphMatching}(a', b', c', d')$
$a'', b'' \leftarrow a, b$ with only plaintext i where $i \in \mathbf{Y}$
$R \leftarrow \mathsf{PartitionOptimisation}(a'', b'', c_0, d_0)$
$f(i) \leftarrow m + 1$ for $i \in R$
$f(i) \leftarrow m + 2$ for $i \notin R \cap i \in \mathbf{Y}$
Return f

Alg TournamentSelection(k, n)

Return $\min(\mathsf{RandomInteger}(1 \dots n), i \in [1, k])$

Alg Mutate$((\mathbf{X}, \mathbf{Y}))$

$C \leftarrow\!\!\$\ \mathsf{RandomInteger}(1 \dots 5)$
Swap C elements between $\mathbf{X}$ and $\mathbf{Y}$
Return $(\mathbf{X}, \mathbf{Y})$

Alg GeneticAlgorithm(a, b, c, d)

Initialise $\mathbf{P}$ with p random subsets of plaintexts not in the JOIN
For $i \in [1, g]$ do
  $\mathbf{F} = \mathsf{Pr}[\mathsf{Fitness}(a, b, c, d, \mathbf{P}_i)] : i \in [1, p]$
  Sort $\mathbf{F}$ by descending fitness
  $\mathbf{P}' \leftarrow \{\mathsf{Mutate}(\mathbf{P}_i) : i \leftarrow\!\!\$\ \mathsf{TournamentSelection}(k, p), j \in [1, \frac{4p}{5}]\}$
  $\mathbf{P}' \leftarrow \mathbf{P}' \cup \{\mathbf{P}_i : i \in [1, \frac{p}{10}]\}$
  Add $\frac{p}{10}$ random subsets to $\mathbf{P}'$
End for
Return $\mathsf{Fitness}(a, b, c, d, \mathbf{P}_0)$

Fig. 14. Pseudocode for genetic algorithm

Alg GradientDescent(a, b, c, d)

Initialise mapping f with buckets $\mathbf{X}$ and $\mathbf{Y}$
While true do
  If multistep
    Compute matrix $\mathbf{M}$
      If no possible score improvement
        Return f
      Else
        Make swaps according to HungarianAlgorithm$(\mathbf{M})$
      End if
  Else
    For $i \in \mathbf{X}, j \in \mathbf{Y}$ do
      $s_{ij} \leftarrow$ score improvement from swapping plaintext i and j
    End for
    $(i', \quad j') \leftarrow \underset{i \in \mathbf{X}, j \in \mathbf{Y}}{\operatorname{argmax}} s_{ij}$
    If $s_{i'j'} < 0$
      Return f
    Else
      Swap i and j
    End if
  End if
  $a', b', c', d' \leftarrow a, b, c, d$ with only plaintext i where $i \in \mathbf{X}$
  $f \leftarrow$ GraphMatching(a', b', c', d')
  $a'', b'' \leftarrow a, b$ with only plaintext i where $i \in \mathbf{Y}$
  $R \leftarrow$ PartitionOptimisation(a'', b'', c_0, d_0)
  $f(i) \leftarrow m + 1$ for $i \in R$
  $f(i) \leftarrow m + 2$ for $i \notin R \cap i \in \mathbf{Y}$
End while

Fig. 15. Pseudocode for gradient descent

E Results

E.1 Score Calculation

Consider the function $E(x, y)$ that returns 1 if $x = y$ and 0 otherwise. The formula for score calculation are as below.

	Single Column	**Double Column**
Complete	$v = \frac{1}{n}\sum_i E(A_i, C_{f(i)})$ $r = \frac{1}{N}\sum_i E(A_i, C_{f(i)}) \times c_{f(i)}$	$v = \frac{1}{n}\sum_i E(A_i, C_{f(i)})$ $r = \frac{\sum_i E(A_i, C_{f(i)}) \times c_{f(i)} d_{f(i)}}{\sum_{i>0} c_i d_i}$
Incomplete	$v = \frac{1}{m}\sum_i E(A_i, C_{f(i)})$ $r = \frac{1}{N}\sum_{f(i)\neq 0} E(A_i, C_{f(i)}) \times c_{f(i)}$	$v = \frac{1}{n}\sum_{f(i)\leq m} E(A_i, C_{f(i)})$ $r = \frac{\sum_{f(i)\leq m} E(A_i, C_{f(i)}) \times c_{f(i)} d_{f(i)}}{\sum_{i>0} c_i d_i}$

E.2 Single Column, Complete

Column Name	**v-score**	**r-score**
First name	0.000239	0.0846
Last name	0.0000489	0.019
Gender (Ohio)	1.00	1.00
Race (Ohio)	0.167	0.123
Age in years at admission	0.344	0.318
Neonatal age (first 28 days after birth) indicator	1.00	1.00
Admission month	0.769	0.758
Admission day is a weekend	1.00	1.00
Died during hospitalization	1.00	1.00
DRG in effect on discharge date	0.0367	0.256
MDC in effect on discharge date	0.769	0.979
ICD-10-CM Diagnosis 1	0.00197	0.223
Number of days from admission to I10_PR1	0.222	0.997
Gender (HCUP)	1.00	1.00
Race (HCUP)	1.00	1.00

Table 3. Single Column, Complete

E.3 Single Column, Incomplete

Number of distinct ciphertexts	$e = 5$		$e = 10$		$e = 20$	
	v-score	**r-score**	**v-score**	**r-score**	**v-score**	**r-score**
lin						
10	1.0	1.0	0.783	0.639	0.578	0.44
50	0.693	0.572	0.348	0.253	0.203	0.137
100	0.319	0.343	0.158	0.153	0.087	0.073
200	0.056	0.068	0.055	0.069	0.042	0.051
500	0.011	0.02	0.011	0.019	0.007	0.011
slin						
10	0.989	0.989	0.622	0.549	0.2	0.129
50	0.338	0.333	0.136	0.144	0.053	0.048
100	0.244	0.248	0.122	0.104	0.046	0.051
200	0.1	0.1	0.076	0.081	0.038	0.039
500	0.026	0.028	0.02	0.023	0.014	0.015
zipf						
10	1.0	1.0	1.0	1.0	0.42	0.68
50	0.383	0.776	0.313	0.704	0.164	0.481
100	0.19	0.659	0.15	0.598	0.074	0.397
200	0.093	0.581	0.071	0.514	0.047	0.38
500	Too many states, insufficient memory, omitted					

Table 4. Single Column, Incomplete

E.4 Double Column, Complete

$e =$ the percentage error introduced when generating auxiliary data

Number of distinct ciphertexts	$e = 5$		$e = 10$		$e = 20$	
	v-score	**r-score**	**v-score**	**r-score**	**v-score**	**r-score**
$\mathsf{lin} - \mathsf{lin}$						
10	1.0	1.0	0.98	0.953	0.92	0.857
50	0.782	0.608	0.546	0.333	0.308	0.161
100	0.488	0.272	0.312	0.156	0.182	0.073
200	0.289	0.119	0.176	0.064	0.109	0.0371
500	0.108	0.0524	0.066	0.0286	0.0472	0.0144
$\mathsf{lin} - \mathsf{slin}$						
10	1.0	1.0	0.98	0.965	0.86	0.792
50	0.676	0.508	0.47	0.337	0.244	0.135
100	0.395	0.211	0.258	0.138	0.141	0.0682
200	0.227	0.129	0.138	0.0696	0.0695	0.0285
500	0.0704	0.0407	0.0548	0.0243	0.0352	0.0162
$\mathsf{lin} - \mathsf{invlin}$						
10	1.0	1.0	1.0	1.0	0.96	0.95
50	0.924	0.9	0.684	0.598	0.428	0.329
100	0.617	0.522	0.385	0.276	0.226	0.153
200	0.294	0.21	0.201	0.135	0.11	0.064
500	0.112	0.0846	0.0772	0.048	0.0434	0.0246
$\mathsf{lin} - \mathsf{randlin}$						
10	1.0	1.0	1.0	1.0	0.96	0.927
50	0.992	0.992	0.914	0.887	0.692	0.593
100	0.937	0.88	0.819	0.681	0.525	0.323
200	0.874	0.816	0.644	0.49	0.332	0.172
500	0.667	0.517	0.393	0.22	0.178	0.0612

Table 5a. Double Column, Complete, Part 1

Number of distinct ciphertexts	$e = 5$		$e = 10$		$e = 20$	
	v-score	**r-score**	**v-score**	**r-score**	**v-score**	**r-score**
lin − zipf						
10	1.0	1.0	0.98	0.98	0.91	0.91
50	0.746	0.746	0.466	0.466	0.272	0.272
100	0.484	0.484	0.292	0.292	0.163	0.163
200	0.287	0.288	0.158	0.159	0.1	0.101
500	0.113	0.113	0.0802	0.0801	0.0408	0.0407
zipf − zipf						
10	1.0	1.0	1.0	1.0	0.86	0.984
50	0.804	0.995	0.556	0.983	0.338	0.95
100	0.503	0.99	0.32	0.975	0.198	0.946
200	0.294	0.986	0.188	0.971	0.116	0.945
500	0.129	0.984	0.0902	0.97	0.0526	0.941
zipf − randzipf						
10	1.0	1.0	1.0	1.0	1.0	1.0
50	1.0	1.0	0.928	0.977	0.696	0.906
100	0.957	0.993	0.821	0.961	0.509	0.871
200	0.855	0.986	0.643	0.954	0.33	0.824
500	0.561	0.965	0.382	0.934	0.185	0.863
zipf − invzipf						
10	1.0	1.0	1.0	1.0	0.92	0.946
50	0.894	0.951	0.672	0.837	0.374	0.636
100	0.658	0.847	0.436	0.711	0.239	0.552
200	0.384	0.723	0.252	0.613	0.142	0.476
500	0.19	0.617	0.11	0.501	0.063	0.41

Table 5b. Double Column, Complete, Part 2

E.5 Double Column, Incomplete, Probability Estimation

Number of distinct ciphertexts	$e = 5$		$e = 10$		$e = 20$	
	v-score	**r-score**	**v-score**	**r-score**	**v-score**	**r-score**
$\mathsf{lin} - \mathsf{lin}$						
10	1.0	1.0	0.831	0.631	0.664	0.491
50	0.531	0.545	0.36	0.318	0.17	0.0963
100	0.362	0.261	0.163	0.0976	0.102	0.0712
200	0.236	0.116	0.137	0.0569	0.0923	0.0362
500	0.0865	0.0292	0.0514	0.0162	0.0352	0.00924
$\mathsf{lin} - \mathsf{slin}$						
10	1.0	1.0	0.731	0.599	0.731	0.599
50	0.458	0.429	0.32	0.245	0.15	0.0712
100	0.305	0.225	0.195	0.111	0.102	0.0572
200	0.192	0.109	0.0978	0.0457	0.0711	0.0352
500	0.0593	0.0287	0.0397	0.0145	0.0266	0.0109
$\mathsf{lin} - \mathsf{invlin}$						
10	1.0	1.0	1.0	1.0	1.0	1.0
50	0.399	0.4	0.377	0.383	0.302	0.288
100	0.349	0.362	0.277	0.289	0.24	0.239
200	0.129	0.133	0.207	0.203	0.185	0.186
500	0.0706	0.0683	0.0786	0.0771	0.0854	0.0821
$\mathsf{lin} - \mathsf{randlin}$						
10	1.0	1.0	1.0	1.0	1.0	1.0
50	0.695	0.798	0.678	0.744	0.568	0.527
100	0.765	0.846	0.794	0.786	0.563	0.441
200	0.715	0.744	0.576	0.52	0.362	0.276
500	0.463	0.487	0.325	0.273	0.164	0.112

Table 6a. Double Column, Incomplete, Probability Estimation, Part 1

Number of distinct ciphertexts	$e = 5$		$e = 10$		$e = 20$	
	v-score	**r-score**	**v-score**	**r-score**	**v-score**	**r-score**
lin − zipf						
10	1.0	1.0	1.0	1.0	0.75	0.75
50	0.512	0.535	0.5	0.516	0.308	0.312
100	0.171	0.165	0.187	0.182	0.171	0.163
200	0.23	0.243	0.171	0.171	0.135	0.127
500	0.105	0.105	0.09	0.089	0.0571	0.0531
zipf − zipf						
10	1.0	1.0	1.0	1.0	0.6	0.928
50	0.578	0.988	0.39	0.973	0.228	0.896
100	0.422	0.981	0.234	0.96	0.145	0.837
200	0.231	0.974	0.137	0.961	0.0818	0.865
500	0.126	0.98	0.0748	0.966	0.0412	0.879
zipf − randzipf						
10	1.0	1.0	1.0	1.0	0.925	0.976
50	0.682	0.878	0.724	0.905	0.635	0.846
100	0.704	0.931	0.672	0.924	0.584	0.896
200	0.557	0.894	0.485	0.881	0.318	0.743
500	0.447	0.927	0.358	0.915	0.228	0.849
zipf − invzipf						
10	1.0	1.0	1.0	1.0	1.0	1.0
50	0.448	0.697	0.49	0.702	0.515	0.699
100	0.443	0.715	0.436	0.704	0.324	0.568
200	0.27	0.611	0.251	0.587	0.168	0.437
500	0.13	0.526	0.108	0.476	0.0741	0.37

Table 6b. Double Column, Incomplete, Probability Estimation, Part 2

Number of distinct ciphertexts		$e = 5$		$e = 10$		$e = 20$	
		v-score	**r-score**	**v-score**	**r-score**	**v-score**	**r-score**
lin − lin							
	10	1.0	1.0	0.831	0.631	0.664	0.491
	50	0.456	0.53	0.336	0.312	0.174	0.0977
	100	0.335	0.258	0.166	0.0988	0.0908	0.0704
	200	0.221	0.115	0.136	0.0569	0.0866	0.0362
	500	0.0845	0.0292	0.0478	0.0164	0.0348	0.00941
lin − slin							
	10	1.0	1.0	0.731	0.599	0.731	0.599
	50	0.4	0.417	0.298	0.241	0.148	0.0717
	100	0.287	0.223	0.182	0.112	0.0952	0.0579
	200	0.165	0.107	0.0921	0.0456	0.07	0.0351
	500	0.0571	0.0283	0.0393	0.0146	0.0264	0.011
lin − invlin							
	10	1.0	1.0	1.0	1.0	1.0	1.0
	50	0.211	0.203	0.296	0.292	0.228	0.21
	100	0.118	0.118	0.243	0.242	0.243	0.234
	200	0.181	0.17	0.173	0.172	0.163	0.161
	500	0.0528	0.0485	0.07	0.0673	0.0818	0.0785
lin − randlin							
	10	1.0	1.0	1.0	1.0	1.0	1.0
	50	0.605	0.744	0.673	0.736	0.572	0.519
	100	0.734	0.821	0.783	0.772	0.563	0.441
	200	0.701	0.741	0.576	0.52	0.357	0.273
	500	0.412	0.448	0.307	0.269	0.159	0.11

Table 7a. Double Column, Incomplete, Probability Estimation $c_0 = d_0$, Part 1

Number of distinct ciphertexts	$e = 5$		$e = 10$		$e = 20$	
	v-score	**r-score**	**v-score**	**r-score**	**v-score**	**r-score**
lin − zipf						
10	1.0	1.0	1.0	1.0	0.75	0.75
50	0.459	0.479	0.452	0.471	0.327	0.332
100	0.225	0.233	0.2	0.201	0.176	0.164
200	0.165	0.174	0.148	0.146	0.12	0.116
500	0.11	0.113	0.0935	0.0934	0.0546	0.0519
zipf − zipf						
10	1.0	1.0	1.0	1.0	0.6	0.928
50	0.512	0.979	0.393	0.962	0.235	0.894
100	0.379	0.977	0.227	0.959	0.144	0.837
200	0.236	0.974	0.142	0.962	0.084	0.865
500	0.123	0.98	0.0743	0.966	0.0427	0.879
zipf − randzipf						
10	1.0	1.0	1.0	1.0	0.925	0.976
50	0.646	0.84	0.663	0.859	0.61	0.835
100	0.653	0.91	0.663	0.913	0.572	0.889
200	0.514	0.881	0.447	0.855	0.311	0.741
500	0.405	0.92	0.354	0.91	0.226	0.848
zipf − invzipf						
10	1.0	1.0	1.0	1.0	1.0	1.0
50	0.483	0.709	0.539	0.744	0.518	0.702
100	0.456	0.722	0.426	0.7	0.312	0.564
200	0.203	0.565	0.208	0.559	0.154	0.433
500	0.128	0.518	0.107	0.476	0.0748	0.372

Table 7b. Double Column, Incomplete, Probability Estimation $c_0 = d_0$, Part 2

E.6 Double Column, Incomplete, Gradient Descent

Number of distinct ciphertexts	$e = 5$		$e = 10$		$e = 20$	
	v-score	**r-score**	**v-score**	**r-score**	**v-score**	**r-score**
lin − lin						
10	1.0	1.0	0.831	0.631	0.664	0.491
50	0.752	0.572	0.486	0.323	0.268	0.0949
100	0.506	0.271	0.303	0.103	0.178	0.0731
200	0.284	0.116	0.162	0.057	0.104	0.0361
500	Omitted due to long runtime					
lin − slin						
10	1.0	1.0	0.731	0.599	0.731	0.599
50	0.65	0.459	0.433	0.251	0.227	0.071
100	0.422	0.233	0.262	0.114	0.149	0.0583
200	0.202	0.109	0.107	0.0469	0.0801	0.0361
500	Omitted due to long runtime					
lin − invlin						
10	1.0	1.0	1.0	1.0	1.0	1.0
50	0.439	0.447	0.421	0.428	0.734	0.718
100	0.522	0.539	0.377	0.372	0.427	0.428
200	0.193	0.194	0.196	0.188	0.123	0.129
500	Omitted due to long runtime					
lin − randlin						
10	1.0	1.0	1.0	1.0	1.0	1.0
50	0.906	0.96	0.876	0.907	0.669	0.617
100	0.752	0.866	0.706	0.785	0.47	0.433
200	0.627	0.74	0.466	0.502	0.26	0.254
500	Omitted due to long runtime					

Table 8a. Double Column, Incomplete, Single-step Gradient Descent, Part 1

Number of distinct ciphertexts	$e = 5$		$e = 10$		$e = 20$	
	v-score	**r-score**	**v-score**	**r-score**	**v-score**	**r-score**
$\mathsf{lin} - \mathsf{zipf}$						
10	1.0	1.0	1.0	1.0	0.75	0.75
50	0.791	0.782	0.667	0.651	0.426	0.393
100	0.47	0.468	0.335	0.327	0.231	0.211
200	0.247	0.251	0.179	0.175	0.116	0.111
500	Omitted due to long runtime					
$\mathsf{zipf} - \mathsf{zipf}$						
10	1.0	1.0	1.0	1.0	0.6	0.928
50	0.767	0.993	0.425	0.975	0.293	0.899
100	0.518	0.986	0.28	0.96	0.19	0.865
200	0.307	0.984	0.165	0.962	0.101	0.866
500	Omitted due to long runtime					
$\mathsf{zipf} - \mathsf{randzipf}$						
10	1.0	1.0	1.0	1.0	0.925	0.976
50	0.956	0.963	0.902	0.951	0.744	0.897
100	0.926	0.99	0.843	0.978	0.641	0.924
200	0.689	0.931	0.572	0.898	0.34	0.742
500	Omitted due to long runtime					
$\mathsf{zipf} - \mathsf{invzipf}$						
10	1.0	1.0	1.0	1.0	1.0	1.0
50	0.991	0.998	0.892	0.946	0.647	0.773
100	0.699	0.867	0.517	0.757	0.346	0.573
200	0.328	0.631	0.262	0.569	0.178	0.442
500	Omitted due to long runtime					

Table 8b. Double Column, Incomplete, Single-step Gradient Descent, Part 2

Number of distinct ciphertexts		$e = 5$		$e = 10$		$e = 20$	
		v-score	**r-score**	**v-score**	**r-score**	**v-score**	**r-score**
lin − lin							
	10	1.0	1.0	0.831	0.631	0.664	0.491
	50	0.754	0.572	0.488	0.323	0.271	0.0949
	100	0.523	0.271	0.318	0.103	0.199	0.0733
	200	0.317	0.116	0.195	0.057	0.119	0.0362
	500	0.0857	0.029	0.0446	0.0163	0.0297	0.00926
lin − slin							
	10	1.0	1.0	0.731	0.599	0.731	0.599
	50	0.65	0.459	0.433	0.251	0.23	0.071
	100	0.436	0.233	0.277	0.114	0.156	0.0583
	200	0.265	0.111	0.145	0.0472	0.0999	0.0357
	500	0.0665	0.0289	0.0458	0.0146	0.0313	0.0109
lin − invlin							
	10	1.0	1.0	1.0	1.0	1.0	1.0
	50	0.809	0.8	0.88	0.872	0.83	0.81
	100	0.562	0.544	0.526	0.501	0.53	0.503
	200	0.372	0.332	0.337	0.306	0.312	0.283
	500	0.088	0.0935	0.0788	0.0847	0.0752	0.08
lin − randlin							
	10	1.0	1.0	1.0	1.0	1.0	1.0
	50	0.953	0.963	0.893	0.877	0.764	0.649
	100	0.871	0.868	0.813	0.789	0.562	0.441
	200	0.759	0.758	0.626	0.531	0.401	0.276
	500	0.532	0.545	0.36	0.293	0.189	0.122

Table 9a. Double Column, Incomplete, Multi-step Gradient Descent, Part 1

Number of distinct ciphertexts	$e = 5$		$e = 10$		$e = 20$	
	v-score	**r-score**	**v-score**	**r-score**	**v-score**	**r-score**
lin − zipf						
10	1.0	1.0	1.0	1.0	0.75	0.75
50	0.66	0.65	0.573	0.558	0.385	0.354
100	0.478	0.455	0.373	0.342	0.232	0.2
200	0.293	0.285	0.217	0.198	0.138	0.123
500	0.0923	0.0893	0.0735	0.0678	0.0582	0.056
zipf − zipf						
10	1.0	1.0	1.0	1.0	0.6	0.928
50	0.767	0.993	0.412	0.975	0.295	0.9
100	0.527	0.986	0.294	0.96	0.187	0.865
200	0.333	0.984	0.18	0.962	0.11	0.866
500	0.137	0.986	0.0735	0.967	0.0434	0.882
zipf − randzipf						
10	1.0	1.0	1.0	1.0	0.925	0.976
50	0.998	1.0	0.933	0.986	0.751	0.899
100	0.957	0.997	0.856	0.978	0.662	0.929
200	0.809	0.971	0.665	0.931	0.386	0.758
500	0.376	0.876	0.299	0.866	0.187	0.809
zipf − invzipf						
10	1.0	1.0	1.0	1.0	1.0	1.0
50	0.991	0.998	0.892	0.946	0.64	0.767
100	0.783	0.901	0.584	0.797	0.377	0.585
200	0.389	0.696	0.296	0.638	0.178	0.432
500	0.147	0.516	0.114	0.472	0.0723	0.374

Table 9b. Double Column, Incomplete, Multi-step Gradient Descent, Part 2

E.7 Double Column, Incomplete, Genetic Algorithm

Number of distinct ciphertexts	$e = 5$		$e = 10$		$e = 20$	
	v-score	**r-score**	**v-score**	**r-score**	**v-score**	**r-score**
lin − lin						
10	1.0	1.0	0.831	0.631	0.664	0.491
50	0.704	0.571	0.447	0.323	0.238	0.0958
100	0.39	0.267	0.215	0.107	0.136	0.0732
200	0.236	0.115	0.127	0.0562	0.0939	0.0381
500	0.0883	0.0294	0.0545	0.0163	0.037	0.00921
lin − slin						
10	1.0	1.0	0.731	0.599	0.731	0.599
50	0.621	0.459	0.404	0.251	0.218	0.0709
100	0.315	0.215	0.201	0.114	0.116	0.0578
200	0.159	0.104	0.0904	0.0454	0.0672	0.035
500	0.0608	0.0299	0.0356	0.0142	0.0284	0.0103
lin − invlin						
10	1.0	1.0	1.0	1.0	1.0	1.0
50	0.924	0.978	0.946	0.965	0.857	0.87
100	0.528	0.576	0.517	0.547	0.385	0.416
200	0.333	0.328	0.295	0.297	0.257	0.259
500	0.119	0.111	0.114	0.105	0.111	0.103
lin − randlin						
10	1.0	1.0	1.0	1.0	1.0	1.0
50	0.954	0.999	0.891	0.909	0.76	0.671
100	0.859	0.879	0.804	0.789	0.572	0.446
200	0.759	0.761	0.627	0.542	0.393	0.278
500	0.571	0.553	0.383	0.295	0.187	0.123

Table 10a. Double Column, Incomplete, Genetic Algorithm, Part 1

Number of distinct ciphertexts	$e = 5$		$e = 10$		$e = 20$	
	v-score	**r-score**	**v-score**	**r-score**	**v-score**	**r-score**
lin − zipf						
10	1.0	1.0	1.0	1.0	0.75	0.75
50	0.823	0.84	0.694	0.701	0.43	0.413
100	0.469	0.48	0.429	0.435	0.247	0.235
200	0.323	0.341	0.228	0.231	0.157	0.152
500	0.127	0.128	0.108	0.105	0.0645	0.0615
zipf − zipf						
10	1.0	1.0	1.0	1.0	0.6	0.928
50	0.738	0.993	0.404	0.975	0.291	0.9
100	0.42	0.982	0.263	0.96	0.173	0.864
200	0.29	0.983	0.147	0.962	0.0924	0.866
500	0.137	0.986	0.0763	0.967	0.0482	0.882
zipf − randzipf						
10	1.0	1.0	1.0	1.0	0.925	0.976
50	0.963	0.994	0.898	0.983	0.723	0.896
100	0.874	0.993	0.766	0.964	0.629	0.924
200	0.706	0.962	0.575	0.92	0.372	0.761
500	0.52	0.955	0.399	0.934	0.232	0.855
zipf − invzipf						
10	1.0	1.0	1.0	1.0	1.0	1.0
50	0.884	0.956	0.865	0.944	0.644	0.781
100	0.604	0.822	0.521	0.775	0.361	0.579
200	0.403	0.731	0.309	0.658	0.186	0.454
500	0.183	0.607	0.127	0.502	0.0789	0.386

Table 10b. Double Column, Incomplete, Genetic Algorithm, Part 2