Leakage-Abuse Attacks Against Structured Encryption for SQL

Alexander Hoover, *University of Chicago*; Ruth Ng, Daren Khu, Yao'An Li, Joelle Lim, and Derrick Ng, *DSO National Laboratories*; Jed Lim, *NUS High School of Mathematics and Science*; Yiyang Song, *Raffles Institution*

https://www.usenix.org/conference/usenixsecurity24/presentation/hoover
Abstract

Structured Encryption (StE) enables a client to securely store and query data stored on an untrusted server. Recent constructions of StE have moved beyond basic queries, and now support large subsets of SQL. However, the security of these constructions is poorly understood, and no systematic analysis has been performed.

We address this by providing the first leakage-abuse attacks against StE for SQL schemes. Our attacks can be run by a passive adversary on a server with access to some information about the distribution of underlying data, a common model in prior work. They achieve partial query recovery against select operations and partial plaintext recovery against join operations. We prove the optimality and near-optimality of two new attacks, in a Bayesian inference framework. We complement our theoretical results with an empirical investigation testing the performance of our attacks against real-world data and show they can successfully recover a substantial proportion of queries and plaintexts.

In addition to our new attacks, we provide proofs showing that the conditional optimality of a previously proposed leakage-abuse attack and that inference against join operations is NP-hard in general.

1 Introduction

Cloud storage is a ubiquitous and convenient solution for clients who want to store their data on remote servers. This service, running without encryption, can provide a client convenience by allowing the server to store large amounts of data and to process queries over the data. However, moving data on to third party servers comes with security and privacy risks to the information stored. Unfortunately, standard encryption techniques are not useful to clients wanting to outsource the storage and computation of a queryable database, because a server cannot perform queries over rows which are encrypted in standard fashion.

Early work on encrypted databases [2, 41] proposed solutions which had unclear security guarantees, using techniques that were not well understood. However, later work [6, 11, 26, 37] proposed a framework for what are now known as leakage-abuse attacks (LAAs) against these schemes. In their work, the authors give adversarial models and simulate attacks showing that plausible adversaries could infer or recover data, queries that were performed by the client, or other potentially sensitive information.

Structured Encryption (StE) [17] provides an attractive combination of efficiency and query support, and allows for servers to learn a quantified amount of information called leakage. Leakage captures the information (e.g. metadata, patterns) revealed about the StE data and queries within an abstract mathematical model. However, the implications of this leakage for cryptanalysis by a real-world adversary are not immediately clear. For many StE schemes, follow up work [10, 29, 42, 48] has shown sometimes devastating LAAs against the leakage.

A recent line of work has been to design schemes in the StE framework for SQL applications, with a focus on supporting select and join queries [13, 32, 49]. Because SQL is such a wide spread and expressive query language, these StE schemes would be very useful to a client wishing to outsource their database to a server. And, as these and similar schemes become increasingly used, it is important to consider the practical security of the systems in real world settings. For example, work has already shown other practical vulnerabilities in complex non-SQL systems like MongoDB’s queryable encryption [27], and many of these vulnerabilities could be avoided with a proper analysis before deployment.

However, the potential for leakage-abuse within StE for SQL schemes has not been studied at all. We seek to address this gap in the security analysis of StE for SQL schemes, by extending the established LAA framework to SQL select and join operations which compose more complex SQL queries. We then design the first LAAs against the above StE schemes,
and evaluate their effectiveness both theoretically and experimentally.

**Our contributions.** We evaluate the leakage of modern StE for SQL schemes [13, 32, 49] through leakage-abuse attacks. Our attacks assume a passive, honest-but-curious server with auxiliary data, similar to prior work [6, 26, 37]. Our attacks enable an adversary to infer the plaintext values in SQL select queries and to infer plaintext values within columns after a join operation. For complex or nested SQL queries, our attacks can be applied to any or all of the intermediate select or join operations.

Specifically, our evaluation of modern StE for SQL schemes includes the following contributions:

- We develop the first LAA for query inference against select queries in StE for SQL schemes and provide a proof that it is near-optimal. After selection queries are made, our attack can often recover the plaintext parts of those queries. This attack uses dynamic programming to generalize prior attacks against deterministic encryption, and in Section 3.1, we outline how it applies beyond StE for SQL to many searchable encryption schemes.

- We develop the first LAAs for plaintext inference against performing a many-many join in StE for SQL schemes. Whenever a join is performed, our attacks can exploit the leakage to recover the underlying values of the encrypted rows in the join. Our attacks work well in practice, but we show that optimal inference is NP-hard in general.

- We give an optimal LAA against many-many joins in the (common) case when the join output contains every row from both columns.

- We perform an empirical analysis of our attacks using a real data set, and show they almost always recover a substantial fraction of their targets. For many columns, before even 20% of unique values are selected, our select attack recovers 10-20% of the values and over 40% of the rows that were queried. Our best join attacks are arguably more devastating, recovering 30-40% of the unique values and over 60% of the rows in most experiments.

- In the appendix, we provide an algorithm to partition a set in a way that approximately maximizes a product of sums. This algorithm is useful as a subroutine in our attacks against the leakage of join operations and may be of independent interest.

Our attacks follow from two main technical insights. For our attack against selections, we show how to exploit partial frequency information, where prior attacks only worked with full information. For our attacks against joins, we observe that the leakage between the columns can give an adversary significantly more information to abuse. Whether it is partial or not, the server observes correlated frequency information across the columns, which allows them to make inferences more accurately.

In addition to the above, we identify how the type (i.e. one-one, one-many, many-many) of a join affects the leakage revealed to a passive adversary. Our work illustrates practical impacts of security in SQL systems that can depend both on the distribution of underlying data and the joins which can be performed in the database being queried. Our observations and attacks demonstrate that certain join patterns are more vulnerable to inference attacks than others, and we conclude that some care should be given to the database schema before deploying to a structured encryption scheme.

**Attack Evaluation.** For our theoretical results, our proofs measure the theoretical success of our attacks in the Bayesian inference framework, following prior work [6]. We consider an attack “optimal” if it returns a maximum likelihood estimator for the information it is trying to infer.

For our experimental results, we use public Chicago data and simulate a scenario where the city moves from public release to using a StE for SQL system. We use the year prior to the hypothetical change as auxiliary information for our adversaries who passively infer information for later years.

**Limitations and Mitigations.** Despite our use of real-world data, it is important to understand that the leakage that we attack is highly data-dependent in general, making it difficult to make broad conclusions about these scheme’s security. In general, we find that columns with few domain values or with highly skewed data is most vulnerable to our attacks. So, one way to mitigate is to study the distribution of data and queries for a particular database and identify potentially vulnerabilities. In such a case, one could modify the schema or implement specific mitigation techniques for the vulnerable information.

To generically mitigate against these new attacks, one could consider using some of the same mitigations proposed in prior work [45, 46], such as volume-hiding [33] or differentially private [1,40] structurally encrypted primitives. Additionally, one work [13] provides a Modern StE for SQL construction which circumvents the join leakage that we attack at the cost of client computation. While these kinds of solutions would render our attacks ineffective, it is unclear whether new similarly effective attacks could be developed.

To have stronger and more well-understood security, one should instead use more “heavy weight” cryptography such as functional encryption [8], fully homomorphic encryption [21], oblivious RAM [16, 24], or secure multiparty computation [5]. However, these will come at the cost of increased server and/or client computation.

**Organization.** In Section 2, we include preliminaries, including a high level overview of SQL queries, and review some of the related work. In Section 3, we explain the leak-
We follow an adversarial model that has been considered in Appendix A of the full version. In Section 5, we present our partial query recovery attack, including a formal attack model, optimality conditions, and experimental results. We differ the proof of the attack’s near-optimality to Section 6, we provide a summary of our findings and present alternatives and new ways forward to improve StE for SQL schemes.

1.1 Adversarial Models

We follow an adversarial model that has been considered in prior leakage-abuse attack work [6, 26, 37]. Throughout the paper, we assume that a server is running one of the modern StE for SQL schemes [13, 32, 49]. Unfortunately, these modern schemes are only provided formally and do not have implementations which can be directly attacked. However, our threat model has been practically considered for servers running other structurally encrypted systems or “smash-and-grab” adversaries that extract equivalent leakage from server logs [19, 27, 37].

We consider persistent passive adversaries on the servers which cannot influence or modify the queries or data. All of our attacks target leakage which is common in this setting for all schemes listed above. Our adversaries additionally have access to some prior knowledge, called auxiliary information, about the distribution of data in the database that’s being stored on the server. Finally, we assume the adversary knows which column(s) the client queries and, when the exact same query is performed, which is often either leaked directly by the protocol or can be inferred from deterministic tokens or query result sizes. In prior work [37], the adversary has to determine which column was queried through some additional information (i.e. the number of unique values in the column). For simplicity, we assume the adversary has access to this information, since the exact method to determine the targeted column may vary slightly depending on the underlying database and StE scheme chosen.

Query Scope. We narrow our scope to just attacking select and join operations. Modern StE for SQL schemes support more complicated queries by allowing a client to nest these operations and leave and projection or filtering to be performed client side after decryption. We emphasize that there are no proposed StE for SQL schemes which support the full generality and expressiveness of SQL. Modern StE for SQL schemes [13, 32, 49] only support encrypted query processing for the basic selection and join operations outlined in Section 2.

In these schemes, the client may also issue more complicated queries by nesting the basic operations, running them on intermediate tables. Our adversaries, with distributional information about the targeted columns, can calculate the auxiliary distributions for intermediate SQL tables. Then, using the access pattern of the encrypted query processing, selects and joins on the intermediate table(s) can be attacked. Furthermore, when we attack joins, we consider the (to be defined later) cross-column equality of the encrypted rows between two columns. If multiple joins are composed, then we could run even stronger attacks by considering the cross-column equality between each of the different joins together. So, because complex queries are still vulnerable to our attacks (and maybe to even stronger attacks), we focus on the most elementary operations for simplicity and generality.

Attack Goals. When attacking select operations, the goal of the adversary is to determine the values the user is selecting (i.e. query recovery). The adversary will observe the sizes of the query results for select operations. Then, the adversary uses this and the auxiliary information to infer the most likely queries performed by a client.

For attacks against join operations, the adversary observes the access pattern of the structured encryption algorithm for the duration of the join. The goal of the adversary is to infer the data in the columns that are joined upon (i.e. plaintext recovery) using its auxiliary information and the observed access pattern.

Despite two different attack goals, the leakage of volumes from selections and from access patterns of joins are very similar. For both selects and joins, our attacks take as input some auxiliary information and observed leakage pattern. The output is a guess of which plaintext values correspond to which groups in the observed pattern. This guess has the same output format but will correspond to the predicted set of queries (select) or to the data values in the columns (join).

2 Background

Notation. Given a positive integer $n$, let $[n] = \{1, \ldots, n\}$. Similarly, for two integers $m < n$, we write $[m, n] = \{m, m+1, \ldots, n\}$ (with the exception of $[0, 1]$, which represents the real numbers between 0 and 1). We write the image of a function $f : A \rightarrow B$ as $\text{Im}(f) = \{b \in B : \exists a \in A, f(a) = b\}$.

We use Dist($A$) to describe the set of distributions over a finite set $A$ (i.e. a function $p : A \rightarrow [0, 1]$ is in Dist($A$) if and only if $\sum_{a \in A} p(a) = 1$).

SQL Queries. We concern ourselves with two main SQL operations in this paper, selections and joins. In particular, we focus on how these operations work on one or two columns in a database, and we will define select and join operations abstractly, ignoring details such as projections, hierarchical joins, and returning columns not involved in the SQL query. This simplified view of joins and selects captures the necessary aspects of the operations to understand the leakage of encrypted SQL databases we attack. Figure 1 shows how these basic elements of SQL queries operate on the tables $T_1$ and $T_2$.
Before presenting our attacks, we take a closer look at some of the leakage-abuse attacks against those and other related schemes.

We are the first to consider attacks against Structured Encryption databases supporting certain SQL queries with cryptographic tools. We additionally recall some of the leakage-abuse attacks against those and other related schemes.

Meanwhile, the structured encryption framework was first introduced in [17]. The framework has since been used to create structurally encrypted databases supporting certain SQL queries [13, 32, 49]. Attacks against these systems, however, have only begun to be analyzed [19, 31], and the practical security of such systems remains an open question. However, other uses of the structured encryption framework are known to be vulnerable to other leakage-abuse attacks [10, 29, 42, 48].

Our select attacks are also applicable to searchable encryption schemes. This line of work focused on reducing leakage while maintaining efficiency. While early works had leakage equivalent to property-preserving schemes (and therefore are vulnerable to the same attacks), more recent work has given efficient schemes that leak only query and access patterns [12, 18]. Most leakage-abuse attacks introduced against such schemes have focused on specific schemes or different adversarial models and/or different leakage than what is considered in this work [4, 11, 22, 38, 42, 44, 46].

There is also a line of work considering generic leakage-abuse against such schemes which only exploit the search pattern, access pattern, or volume pattern without additional information [7, 20, 25, 34].

In addition to the structured encryption framework, there are approaches to encrypted search that make use of tools such as functional encryption [8], fully homomorphic encryption [21], oblivious RAM [24], or secure multiparty computation [5]. These schemes generally have less leakage but come at a significant theoretical and/or practical efficiency cost.

### 2.1 Related Work

We recall some alternative constructions for processing SQL queries with cryptographic tools. We additionally recall some of the leakage-abuse attacks against those and other related schemes.

Before using the structured encryption framework, some constructions used weaker or property-preserving primitives [41] to support a restricted class of SQL queries on a database. This sparked a line of work attacking these types of constructions [6, 7, 11, 26, 36, 37].

### 2.2 Comparison to Other Attacks

Before presenting our attacks, we take a closer look at some other approaches to leakage-abuse in prior work. This will identify some of our key differences and where we fit into the body of related literature.

We are the first to consider attacks against Structured Encryption (StE) for SQL; however, we will compare (mostly)
against attacks against Searchable Encryption (SSE), since there is some overlap between those attacks and attacks that apply to StE for SQL. For reference, a recent comparison of attacks against searchable encryption (SSE) is nicely summarized in prior work [45, 46]. We also discuss our own threat model in Section 1.1 in more depth and only focus on the differences between our work and other works.

First, many threat models against SSE or Dynamic SSE (DSSE) assume an active adversary [7, 47, 48], which allows for either data insertion or adversarial queries to be issued. In our work, we consider a weaker passive adversary who only tries to infer information about the underlying data or client activity.

Among the work on passive adversaries, there are attacks in which the adversary knows some or all of the underlying data [7, 11, 38, 46]. In our threat model, our adversaries only have access to only distributional information about the underlying data, a weaker assumption than partial knowledge. There are also attacks which work for adversaries without even this distributional assumption [7, 20, 25, 34].

The final difference that we consider is related to the type of leakage that is abused, which influences how present the adversary needs to be. For example, some prior work has considered “snapshot” or “smash-and-grab” adversaries [6, 19, 27, 37], who only access a server running an encrypted system once. In contrast, we consider a persistent adversary who observes the access pattern and activity of the server as queries are processed. We note, however, for some practical systems recent work has shown persistent adversary attacks may be possible from a single snapshot through the use of server logs [27].

The most similar work to our own are covered in a line of attacks such as [29, 39, 42]. These provide some of the most similar threat models to the ones that we consider. However, these attacks are against (sometimes specific) deployments of SSE rather than StE for SQL, which provides some key differences in the attack setting. For example, SSE deployed for keyword-document lookup will have the same document appear for different keywords, whereas two queries selecting on the same column with different values will never have rows overlap (for the restricted class of queries supported by StE for SQL schemes). This means that our attacks do not actually require access pattern leakage and could be run as a computer-in-the-middle with volume leakage.

Additionally, we are the first paper to consider any attacks against cross-column equality leakage, because this leakage pattern is unique to StE for SQL schemes and not present in previous SSE literature.

3 Leakage of Encrypted Databases

Leakage of select and join operations among current encrypted SQL database solutions broadly fall into a handful of categories. In this section, we will outline the type of leakage that we will provide attacks against. Broadly, we categorize our attacks into attacks against selections and attacks against joins.

3.1 Select Leakage

We begin by focusing on the column equality leakage pattern, which our attacks exploit in Section 4.

As prior work has observed, some SQL supporting schemes such as CryptDB [41] or Microsoft’s Always Encrypted system [3] employ weakened primitives such as deterministic encryption to perform select operations by the server. These schemes will effectively reveal a deterministically created ciphertext of every row value in the column selected upon. The more formal leakage associated with this pattern is what we call the complete column equality (c-eq), because the server can check the equality between any two rows by checking if they are the same ciphertext.

This leakage pattern is well studied in prior work [6, 11, 26, 37]. However, we further this work by proving an optimality result and separation for one of the known attacks in Appendix A of the full version.

More recent work in structured encryption supports simple equality select operations without weakened primitives [13, 32, 49]. However, these encrypted SQL databases still leak the query equality and volume to an honest-but-curious server. Specifically, the server is able to know when a select is repeated (query equality) and which columns and rows are repeated between different queries.

For example, a client querying columns $C_1$ and $C_2$ from our example tables in Figure 1, may issues 4 select queries, corresponding to:

\[\text{SEL}(\text{Ari}, C_1), \text{SEL}(\text{Ari}, C_2), \text{SEL}(\text{Bob}, C_1), \text{SEL}(\text{Ari}, C_1)\]

and a server could learn that the first and last queries were the same and that the third query was on the same column as the first and last. However, they would not immediately know which specific values were queried. The final relevant aspect of this is that the server would know how many (and which) rows are returned to the client during each of the queries. In this toy example, this would allow a server to learn that there is some value in $C_1$ which appears 3 times (Ari), another that appears 1 time (Bob), and a value in $C_2$ which appears 2 times (Ari).

A server, after observing just a few queries, can determine the sizes of these equality groups, which are sets of row ciphertexts sharing the same underlying value.

After a client has queried every value in the relevant column, this leakage becomes equivalent to the complete equality leakage. However, before all values are queried, there can still be substantial leakage to the adversary. Especially in instances where a few values dominate most of the rows, this leakage could allow an adversary to infer a lot of the underlying information about the queries being performed.
Throughout this paper, we refer to this pattern as *incomplete column equality* or *partial equality* (i-eq). We explore this more fully and provide a new near-optimal attack against this leakage in Section 4.

**Searchable Encryption.** For simplicity, we discuss our attacks against this leakage in the context of encrypted SQL databases, but we observe that our attacks against encrypted SQL select queries can be applied directly to many searchable encryption (SSE) works. The majority of schemes from the SSE and encrypted multimap literature which achieve practical (i.e. linear) efficiency (e.g. [9,12,14,15,17,18,23,43]) and encrypted multimaps supporting simple equality predicates as a select operation will similarly leak the equality pattern of rows returned to an honest-but-curious server. Our attacks are novel in this setting provide a new approach to an existing line of work [29,39,42].

<table>
<thead>
<tr>
<th>Type</th>
<th>Complete</th>
<th>Incomplete</th>
</tr>
</thead>
<tbody>
<tr>
<td>One-One</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>One-Many</td>
<td>c-eq</td>
<td>i-eq</td>
</tr>
<tr>
<td>Many-Many</td>
<td>c-cross</td>
<td>i-cross</td>
</tr>
</tbody>
</table>

Figure 2: Table showing the leakage available to a server depending on the join type.

3.2 Join Leakage

There are a few techniques and constructions which provide support for server-computed joins on encrypted SQL databases. This includes the Adjustable Key Join technique used within CryptDB [41], the fully precomputed join indexes (SPX, OPX and Fps) [13,32,49]), and finally some structured encryption schemes with advanced security (Secure-Join and JXT [28,30]).

We identify generic and previously unanalyzed leakage present in every one of these schemes which we call the *cross-column equality* pattern. When issuing a join query, all of these schemes allow the server to observe pairs of rows which have matching values. This allows the server to observe the equality pattern of the rows which are output in the join. This reveals both the equality groups within each column (as with the column equality pattern), but additionally reveals the equality across the columns. For example, the join represented in Figure 1 would reveal to the server that there is some value (Ari) which occurs 3 times in C1 and 2 times in C2, another value (Bob) which occurs 1 time in C1 and 3 times in C2, and a third value (Cal) which occurs 3 times in C1 and 1 time in C2.

However, despite this generic leakage being shared across the proposed schemes, the join type of the join being performed informs how the leakage can be exploited, which we summarize in Figure 2. We distinguish join types by whether it is a one-one, one-many, or many-many join and whether the join is complete or incomplete.

The first set of terms inform the uniqueness of values in the columns joined on respectively. A one-one join has no repeated value in either column, a one-many join has no repeated value in one column, and a many-many join may have repeated values in either column. We call a join “complete” if every row has a matching row in the opposite column (i.e. the join outputs every row from both tables at least once), and otherwise the row is said to be incomplete. As an example, our join example in Figure 1, is an incomplete many-many join. It is incomplete because Eve does not appear in the join output, and it is many-many because both columns have values which appear multiple times (e.g. Ari).

For one-one joins (such as many primary key-foreign key joins), all the server observes is a set of encrypted row pairs, none of which repeat. So, there is no hope for a server, even with auxiliary information, to infer the underlying values.

During one-many joins, the server may observe that multiple rows, in the column allowing repeated values, match with the same value in the other column. This gives a server information about the sizes of equality groups in the column with the repeated values, since rows with the same value in the repeats column will be output with the same encrypted row in the unique column. When a join is complete and one-many, the cross-column equality pattern reduces to the complete column equality (c-eq) of the column with repeated values, and similarly, it reduces to the incomplete column equality (i-eq) when the join is incomplete.

Finally, while performing a many-many join, the server observes encrypted pairs of matching rows across the columns. This is a unique pattern which reveals more than just the equality within the columns. The adversary learns either the complete cross-column equality (c-cross) or the incomplete cross-column equality (i-cross), if the join is complete or incomplete respectively. This leakage is strictly more than the respective equality leakage of the two columns because the server learns both the equality group sizes within as well as across the columns in the join.

In this scenario, the server could run equality pattern attacks on the individual columns, but they may be able to outperform those attacks by knowing the rows which are equal between the two columns. In Section 5, we give new attacks which make use of this cross-column equality pattern and show that this additional leakage *does improve* the performance of inference attacks.

4 Select Query Recovery

In Section 3.1, we sketch how encrypted SQL systems leak the column equality pattern as queries are made. In this section, we consider this leakage more formally and outline a new attack against incomplete column equality and evaluate its performance against real-world data. In Appendix A of the full version, we prove that it is near-optimal as well.

**Attack Model.** The formalism for our leakage-abuse attacks follows that of prior work [6] but with some differences
in notation. The leakage input to our algorithms is a vector \( \vec{c} = (c_1, \ldots, c_m, u) \) which contains the sizes of each of the \( m \) leaked “equality groups” (in ascending order without loss of generality) and finally the number of rows without an observed equality group is included as the number \( u \), which can be computed from the leakage discussed in Section 3.1.

For example, if the a client made three queries on the column \( C_2 \) from Figure 1 corresponding to the sets:

\[
\text{SEL}(\text{Bob}, C_2), \text{SEL}(\text{Cal}, C_2), \text{SEL}(\text{Ali}, C_2),
\]

then the server would learn three equality groups sizes 3, 1, and 2 respectively and would know there is 1 row (row 7) which had not been returned on any query. So such an adversary would have the leakage input \( \vec{c} = (1, 2, 3, 1) \).

In addition to the leakage input, the adversary receives some auxiliary information in the form of a distribution over the probability that the rows in the equality groups take on the values \( V \). So, our attack outputs an injective function \( f: [m] \rightarrow V \) which represents the guessed value that underlies each equality group. Throughout, we call this kind of function a plaintext mapping.

**Optimality.** We consider an attack optimal if it returns a function that maximizes the likelihood of observing the given equality groups, assuming that the plaintexts are drawn from the distribution \( \rho \) and that \( f \) is the true mapping. More formally, we want an algorithm which returns

\[
\arg\max_f \Pr[\vec{c}|f = f, \rho],
\]

where \( f \) is the true plaintext mapping, modeled as a uniformly random injective function. This probability is shorthand, introduced in [6], for the probability that we would observe \( \vec{c} \) if \( f \) were the true plaintext mapping and rows were independently sampled from \( \rho \).

When analyzing this probability under these assumptions, we can write it in a convenient and simple form. Specifically, for the plaintext mapping \( f \), we want to calculate the probability that we observe the vector \( \vec{c} = (c_1, \ldots, c_m, u) \). So, we want the probability that \( f(i) \) appears exactly \( c_i \) times for each \( i \in [m] \).

First, we count the number of ways to arrange \( m \) values, \( f(1), \ldots, f(m) \), exactly \( c_1, \ldots, c_m \) times across the \( N = u + \sum c_i \) rows in the column respectively. This number is exactly the multinomial coefficient

\[
K_{\vec{c}} = \binom{N}{c_1, \ldots, c_m, u},
\]

that depends only on \( \vec{c} \).

**Alg SelAttack\( ^k \)(\( \vec{c}, \rho \))**

Let \((v_1, \ldots, v_n)\) be the elements of \( V \) in a ascending order of likelihood (i.e. \( \rho(v_i) \leq \rho(v_j) \) for all \( i < j \)).

Parse \((c_1, \ldots, c_m, u) \rightarrow \vec{c}\)

Define \( \text{rnd}_u(x) = x \cdot \frac{1}{u} \)

\((f, p) \leftarrow \text{helper}(n, m, 0)\)

Return \( f \)

**Alg helper\((i, j, \sigma)\)**

assert\((i \geq j)\)

If \( j = 0 \) then

- For \( k \in [i] \) do \( \sigma \leftarrow \sigma + \rho(v_k) \)

- Let \( f(k) = 1 \) for all \( k \in [m] \)

- Return \((f, u \log \sigma)\)

- \((f_1, p_1) \leftarrow \text{helper}(i - 1, j - 1, \sigma)\)

- \( p_1 \leftarrow p_1 + c_i \log \rho(v_i) \)

- If \( i > j \) then

- \((f_2, p_2) \leftarrow \text{helper}(i - 1, j, \text{rnd}_u(\sigma + \rho(v_i)))\)

- If \( p_1 < p_2 \) then Return \((f_2, p_2)\)

- Set \( f_1(j) = v_i \)

- Return \((f_1, p_1)\)

Figure 3: Pseudocode for an attack against column equality leakage.

Next, we consider the probability that of any fixed ordering. Since we assume rows are independent samples, the probability that the rows in the equality groups take on the values \( f \) assigned is \( \prod_i \rho(f(i))^{c_i} \), given our prior distribution. However, we also need that the \( u \) remaining values in the column are not in the observed equality groups, which happens with probability

\[
\left( \sum_{v \in V \setminus \text{Im}(f)} \rho(v) \right)^u
\]

Putting this together, we can write the expression

\[
\Pr[\vec{c}|f = f, \rho] = K_{\vec{c}} \prod_{i=1}^m \rho(f(i))^{c_i} \cdot \left( \sum_{v \in V \setminus \text{Im}(f)} \rho(v) \right)^u.
\]

**Complete Equality.** Notice that when \( m = |V| \), the summation drops out. Although this may seem insignificant, it simplifies the structure of the problem. In fact, this case is equivalent to when the complete column equality has been revealed and was directly studied in prior work [6, 36, 37]. In particular, prior work outlined three different attacks which use frequency analysis, norm-optimization, and bipartite matching respectively. The first and last of which were proven optimal in [36] and [6] respectively. We add to this line of work by proving in the full version that the attack using norm-optimization is optimal when using an \( \ell^p \)-norm with \( p > 1 \) (Theorem 3) and not optimal when \( p = 1 \) (Theorem 4) in Appendix A.
4.1 Incomplete Equality

Unfortunately, when a server does not observe every equality group in a column, they cannot run the attacks from prior work, which all rely on the existence of a one-to-one mapping between equality groups and plaintext values. For example, it is unclear how to use frequency analysis in such a setting. One can match the most common plaintexts (based on the distribution ρ) with the largest observed equality groups, but this strategy does not yield an optimal plaintext mapping.

Our attack remedies this issue and works for any \( m \leq |V| \). We present the pseudocode for SelAttack\( ^e \) in Figure 3, which details how we recursively compute a plaintext mapping. We sketch some of the intuition of the algorithm and informally why it is optimal.

This algorithm generalizes the frequency analysis attack from prior work [37]. In fact, when \( m = |V| \), SelAttack\( ^e \) always outputs the mapping \( f(i) = v_i \) where \( v_i \) is the \( i \)-th least frequent element according to ρ. This is the only case that never takes the \( i > j \) branch and therefore never rounds nor recurses more than once per call.

However when \( m < |V| \), the algorithm sometimes takes the \( i > j \) branch. And, each time it is taken, the algorithm is recursing and exploring assignments where \( v_i \) is not one of the observed equality groups (and adding this to the probability of the unobserved group). Whereas the normal branch is exploring paths where a \( c_j \) sized equality group corresponds to the \( v_i \) plaintext.

Without rounding, the end result would output an assignment and the logarithm of its corresponding probability (without the normalization constant),

\[
\log \left( \prod_{i=1}^{m} \rho(f(i)^{c_i} \cdot \left( \sum_{v \in V_i} \rho(v) \right)^u \right) = \log \left( \sum_{v \in V_i} \rho(v) \right) + \sum_{i=1}^{m} c_i \log \rho(f(i)).
\]

Additionally, we observe that any optimal plaintext mapping will never assign \( f(i) = v_1 \) and \( f(j) = v_2 \) when \( c_1 < c_j \) and \( \rho(v_1) > \rho(v_2) \). This is formalized by Definition 2 and Lemma 1 in Appendix A, which says any optimal mapping is non-crossing. And this exactly characterizes the mappings that helper recursively considers.

Combining these facts shows that SelAttack\( ^e \) (without rounding) actually explores each of these potential optimal mappings along its recursive branches and only keeps the one with the highest probability (because \( \log(\cdot) \) is an increasing function). So, SelAttack\( ^e \) returns an optimal plaintext mapping without rounding.

However when we introduce rounding, it allows us to bound the runtime of SelAttack\( ^e \) very easily. In practice, we supplement our recursive calls with memoization, a well known technique used often in dynamic programming. With memoization and rounding, helper can only receive at most \( m \cdot |V| / \epsilon \) inputs, giving us a runtime of \( O(m \cdot |V| / \epsilon) \). Without rounding, the input \( \sigma \) to helper could vary by just a little bit and still require more recursive calls, and there are many inputs that would run in exponential time (even with memoization). In Appendix A of the full version, we prove that this attack is still near optimal with some rounding (Theorem 5), so we benefit from both a bounded runtime and theoretical guarantees.

4.2 Experimental Evaluation

To supplement our theoretical bound on the performance of the algorithm, we simulate our attack against real-world data to better understand the practical implications of a near-optimal inference.

Setup. We run our experiments on public data from the city of Chicago.\(^2\) Our experiments focus on reconstruction attacks over the taxi, crime, rideshare, and vehicle crash tables in the data from the years 2018 to 2022.\(^3\) We attempt to reconstruct columns containing values for beats, community areas (CA), and speed limits. Further details about the data used in these and other experiments can be found in Appendix D.

We simulate a scenario where Chicago decides, in 2019, to keep their data more private. Specifically, they begin using a Structured Encryption for SQL scheme for their datasets and allow authorized users to make queries. We test the ability of an adversary observing the volumes of selections on the Chicago server to infer the values that a users is querying.

Our attacks use the already published 2018 Chicago tables as the auxiliary distribution ρ in our attacks. For each year 2019-2022, our adversary observes queries over some fraction of the values in the table. They then use SelAttack\( ^e \) to infer the value corresponding to each query. The equality group sizes the adversary observes and selected by either choosing a uniformly random subset of values from the value space or by sampling values randomly weighted according to the number of rows containing that value. For each year, fraction of equality groups revealed, and sampling method, we average over 10 independent trials.

Evaluation. To better understand the practical implications of a near-optimal plaintext mapping, we score our results using the value recovery rate (val-rec) and the row recovery rate (row-rec). These represent the fraction of values or rows inferred correctly in different ways.

More specifically, after \( m \) equality groups are revealed in \( \vec{c} = (c_1, \ldots, c_m, u) \), we let \( f : [m] \rightarrow V \) be the true mapping between the equality groups revealed and the underlying plaintext values.

Then, the value recovery rate (val-rec) for a plaintext mapping \( g \) is

\[
\frac{|\{i \in [m] : f(m) = g(m)\}|}{m}.
\]

\(^2\)https://data.cityofchicago.org/
\(^3\)Note that there is less (and different) data for the 2020 and 2021 years in some tables, due to the COVID pandemic, which may have an impact on the accuracy of our auxiliary distributions in the attacks.
the number of values inferred correctly to the exact equality group divided by the total number equality groups revealed.

The row recovery rate (row-rec) for a plaintext mapping \( g \) and column \( (r_1, \ldots, r_k) \in V^N \) is

\[
\frac{|\{i \in [N] : r_i \in \text{Im}(f) \land f^{-1}(r_i) = g^{-1}(r_i)\}|}{N - u},
\]

the number of rows correctly inferred divided by the total number of rows observed.\(^4\) This is similar to the value recovery rate but weighted toward values which appear frequently.

**Results.** In Figure 4, we present graphs charting how SelAttack\(^6\) performs as more equality groups are revealed. For each column we attack, the corresponding graph shows how our attack perform on average (for uniform and weighted sampling) as more equality groups are revealed. This corresponds to an adversary observing more and more queries from a client. One may expect that attacks would perform much worse when only a small subset of values are queried, but we find that our recovery rates are not heavily influenced by the fraction of values revealed.

Each attack we ran finished in under 40 seconds on an Intel i9 Macbook Pro, without significant optimization. Most would finish in just a few seconds, with the main exception being our Crimes Beat experiments when about 50% of the values were observed, which took 30-40 seconds. This was unsurprising, since this setting of many domain values with half being observed significantly increases the number of branches the algorithm must consider.

\(^4\)For values \( v \not\in \text{Im}(g) \), we let \( g^{-1}(v) = \perp \)

Our attack’s performance depends on the distribution of the underlying data. Since our attack uses the expected frequencies of certain values, distributions which are very flat and spread across many values (e.g. beats from the crimes table) perform worse, only achieving val-rec of less than 5%. Where as more skewed data on smaller domains typically perform much better, achieving val-rec around 10-40%. Also our attack is very good at recovering the most frequent values, evidenced by our attacks against speed limits, and taxi data, which often get row-rec over 50%, even when values are selected uniformly at random.

Overall, this level of query recovery could be concerning for specific deployments of modern SiE for SQL schemes. When a client performs selects on sensitive data, we show that a passive adversary may be able to infer key parts of the client’s queries. If a client wants to hide what they do with their data, these attacks indicate that the leakage from these schemes is insufficient, when an adversary might have some prior knowledge over the data distribution.

**5 Join Plaintext Recovery**

Section 3.2 illustrates how state of the art structured encryption for SQL schemes leak the equality group sizes across the columns during a join. Notice that this leakage is more than just the equality group sizes in both columns. The adversary additionally learns which groups have the same plaintext value across the columns. 

**Attack Model.** The formalism for this attack extends the ideas put forward in prior work [6], by using a Bayesian
inference framework. The leakage input to our algorithms attacking
cross-column equality are two vectors \( \vec{c}_1 = (c_1, \ldots, c_m, u_1) \)
and \( \vec{c}_2 = (d_1, \ldots, d_m, u_2) \) which contain the
sizes of each of the \( m \) observed equality groups across the
two columns. Unlike the vector in Section 4, we cannot
assume both of these are in ascending order without loss of
generality. Instead, we assume that the equality groups cor-
responding to \( c_i \) and \( d_i \) have the same underlying plaintext value and
that the observed group sizes of \( \vec{c}_1 \) are in ascending order.
The number of rows without an observed equality group is \( u_1 \)
and \( u_2 \) in the columns respectively, which can be computed
from the leakage discussed in Section 3.2.

As an example from Figure 1, if we were to observe the
leakage from the join corresponding to \( \text{JOIN}(C_1, C_2) \) then we
would run our attacks with the observed vectors

\[
\vec{c}_1 = (1, 3, 3, 0)
\]

and

\[
\vec{c}_2 = (3, 1, 2, 1),
\]

which correspond to the values Bob, Cal, and Ari (followed
by the number of unobserved rows), since we keep \( \vec{c}_1 \) in
ascending order.

In addition to the observed leakage vectors, our algorithms
take some auxiliary information as an input. Specifically they
take two distributions \( \rho_1 \) and \( \rho_2 \) over the set of plaintext values \( V \).
These distributions describe the plaintext distribution that
both columns have their values independently drawn from.
In this setting, our algorithms have the goal to find the most
likely mapping between the equality groups and the set of
plaintext values \( V \).

First Optimality Attempt. Unlike in Section 4, the optim-
ality of our attacks against cross-column equality is not
as straightforward. We put forward two possible theoretical
objectives of attacks and argue to adopt the latter definition.

First, we parallel Section 4 closely and consider attacks
which only output an injective function \( f : [m] \to V \), called a
plaintext mapping, which represents the guessed value that
underlies each equality group. This is not the most an adver-
sary could infer, since they additionally know that the rows
in the two columns, which were not in the observed equality
groups must be unequal from each other. However, this type
of attack may capture most of what an adversary would care
about and parallels its objective nicely with the attacks from
prior work.

An attack of this sort optimal if it returns a plaintext map-
ing that maximizes the likelihood of observing the given
equality groups, assuming that the columns’ values are drawn
from the distributions \( \rho_1 \) and \( \rho_2 \) respectively. More formally,
an algorithm is optimal if it returns

\[
\arg \max_f \Pr[\vec{c}_1, \vec{c}_2 | f = f, \rho_1, \rho_2],
\]

where \( f \) is the true plaintext mapping, modeled as a uniformly
random injective function.

Using an analogous probability analysis, as the one made
in Section 4, we can somewhat expand the expression for
the probability of the observed equality groups for a plain-
text mapping \( f \). For vectors \( \vec{c}_1 = (c_1, \ldots, c_m, u_1) \) and \( \vec{c}_2 =
(\ldots, d_m, u_2) \), we can expand the expression to

\[
\Pr[\vec{c}_1, \vec{c}_2 | f = f, \rho_1, \rho_2] =
K_{\vec{c}_1, \vec{c}_2} T_1^u T_2^{u_2} P \prod_{i=1}^m \rho_1(f(i))^{c_i} \cdot \rho_2(f(i))^{d_i},
\]

where \( K_{\vec{c}_1, \vec{c}_2} \) is a normalization constant that depends only on
the terms in the vectors and the terms

\[
T_i = \sum_{v \in V \setminus \text{Im}(f)} \rho_i(v),
\]

the total probability of a sample lying outside of the image of
\( f \) for the distribution \( \rho_i \). Unfortunately, \( P \) does not have a
clean expression in terms of our input variables, so we leave
it in terms of \( U_i \) which is the set values from \( u_i \) independent
samples of \( \rho_i \) conditioned on all \( u_i \) samples lying outside of
the image of \( f \). With that notation, we can write

\[
P = \Pr[U_1 \cap U_2 = \emptyset | f = f, \rho_1, \rho_2],
\]

the probability that the two unobserved rows in the two
columns are never equal (since otherwise they would’ve com-
prised another equality group).

Optimality. We also consider attacks which output \( f \) and sets
\( U_1 \) and \( U_2 \) with \( U_1 \cap U_2 = \emptyset \) and \( U_1 \cup U_2 \subseteq V \setminus \text{Im}(f) \),
which represent the guess at the most likely values among
rows in the unobserved equality groups in the columns. This
definition captures that maximum amount of information an
attack could infer about the underlying data from \( \vec{c}_1 \) and \( \vec{c}_2 \),
by determining the most likely values present in each column
respectively. Similar to the other definitions, an attack would
be optimal if it returns

\[
\arg \max_{f, U_1, U_2} \Pr[\vec{c}_1, \vec{c}_2 | f = f, U_1 = U_1, U_2 = U_2, \rho_1, \rho_2].
\]

For attacks which output these sets, we consider the prob-
ability of the triple \( (f, U_1, U_2) \) in a more explicit expression.
For vectors \( \vec{c}_1 = (c_1, \ldots, c_m, u_1) \) and \( \vec{c}_2 = (d_1, \ldots, d_m, u_2) \), we can write

\[
\Pr[\vec{c}_1, \vec{c}_2 | f = f, U_1 = U_1, U_2 = U_2, \rho_1, \rho_2] =
K_{\vec{c}_1, \vec{c}_2} P(U_1, U_2) \prod_{i=1}^m \rho_1(f(i))^{c_i} \cdot \rho_2(f(i))^{d_i},
\]

where \( K_{\vec{c}_1, \vec{c}_2} \) is a normalization constant that depends only on
the terms in the vectors, and

\[
P(U_1, U_2) = \left( \sum_{v \in U_1} \rho_1(v) \right)^{u_1} \cdot \left( \sum_{v \in U_2} \rho_2(v) \right)^{u_2}
\]
we show that is in fact NP-hard to make an inference which
For every n element set \( V \),
\[ \rho \]
and \( \text{BM} \)
and decode for
mapping maximizing the target probability. We give the pseu-
mapping is just the logarithm of the probability an attack
of equality groups to plaintext values.
us bipartite matching, similar to one of the attacks for com-
When \( m = |V| \), corresponding to attacking complete cross-
columns fall into \( U_1 \) and \( U_2 \) respectively.
When \( m < |V| \), the two optimality objectives differ. We find
that the latter definition is more useful from both theoretical
and practical perspectives. For example, our attacks make
use of efficient likelihood estimation, and in Appendix C,
we show that is in fact NP-hard to make an inference which
outputs the most likely sets \( U_1 \), \( U_2 \) of unassigned plaintext
values (Theorem 6). Additionally, this definition corresponds
to the most an adversary can infer, which is the overarching
goal of leakage-abuse attacks. Given these virtues, we stick
with this as the standard definition for optimality.

5.1 Complete Cross-Column Equality

Our main attack for attacking complete cross-column equality
uses bipartite matching, similar to one of the attacks for com-
plete equality in prior work [6]. The BM algorithm begins
by creating a bipartite graph based on the observed equality
groups and auxiliary distributional information. A perfect
matching in this graph corresponds directly to an assignment
of equality groups to plaintext values.

By weighting the edges appropriately, the weight of any
matching is just the logarithm of the probability a match-
ness to maximize (up to a constant). So, BM just constructs
such a graph and uses a subroutine to find a maximum weight
bipartite matching, which can be converted into a plaintext
mapping maximizing the target probability. We give the pseu-
docode for BM in Figure 5. Additionally, we prove the The-
orem 1 that this attack is optimal, as an almost immediate
corollary of the Hungarian Algorithm’s correctness.

**Theorem 1.** For every n element set \( V \), \( \bar{c}_1, \bar{c}_2 \in \mathbb{Z}^n \times \{0\} \)
and \( \rho_1, \rho_2 \in \text{Dist}(V) \), \( \text{BM}(\bar{c}_1, \bar{c}_2, \rho_1, \rho_2) \) (Figure 5) outputs

**Figure 5:** Bipartite matching algorithm BM for attacking complete cross-column equality.

is the probability that all the unobserved rows from the
columns fall into \( U_1 \) and \( U_2 \) respectively.

5.2 Incomplete Cross-Column Equality

The two potential targets for incomplete cross-column equal-
ity both have issues when trying to find an effective and
optimal leakage abuse attack. For attacks only returning a
plaintext mapping, there isn’t a very precise general formula
for the likelihood of the observed data. And for attacks which
maximize the amount they can infer, we show, in Theorem 6
that finding an optimal function is NP-hard in the worst case.

Despite these challenges, we give attacks which output a
plaintext mapping \( f \) as well as sets \( U_1 \) and \( U_2 \) to infer
the values in the unobserved rows. We do this because they are
more general, can just be modified to only output the plaintext

**Figure 6:** The algorithm Greedy\(^E \) shows one way to optimize over the MapFromSet\(^E \) function to find a likely plaintext mapping
for incomplete cross-column equality.

because the Hungarian Algorithm [35] always returns a
maximizing function. But, observing this expression is
\( \log(\text{Pr}[\bar{c}_1, \bar{c}_2|f = f, \rho_1, \rho_2]) \) up to a constant factor (de-
ending only on \( \bar{c}_1, \bar{c}_2 \)) and that \( \log(\cdot) \) is an increasing func-
tion leads us to conclude that such an \( f \) also maximizes
\( \text{Pr}[\bar{c}_1, \bar{c}_2|f = f, \rho_1, \rho_2] \).
mapping, and having a concrete formula allows us to adapt techniques used in our other leakage-abuse attack settings.

**Partitioning Approximation.** Throughout our attacks, we use our own Partitioning Optimization Approximation Algorithm (POAA) as a subroutine. The pseudocode for POAA is in Figure 11 (Appendix B). After a plaintext mapping is chosen, POAA approximately finds the best way to divide the remaining plaintext values between the two columns. It will then return sets $U_1, U_2$ which approximately maximize the value $P(U_1, U_2)$.

The algorithm is parameterized by a value $\epsilon \in (0, 1)$, which determines the precision of the approximation. As input, it takes the set of remaining plaintext values $R$, the distributions $\rho_i$, and the number of rows which were not in an equality group $u_i$ for both columns ($i = 1, 2$). It returns two sets, which are the respective guesses for the most likely split of the remaining plaintext values across the columns.

**Greedy Algorithm.** The first approach for solving incomplete cross-column equality is to give a function which can be optimized using generic techniques. In Figure 6, we present MapFromSet which takes a guess at which $m$ plaintexts are present in the join output and returns a guess at the plaintext mapping, the split between the unassigned values, and the probability estimate for the returned tuple.

It works by first running the BM algorithm on the equality groups, and restricted to the set $S$ to find a candidate plaintext mapping $f$. From there, it uses the POAA algorithm on the remaining plaintext values to figure out an approximately good split $U_1, U_2$. Finally, the algorithm returns the $(f, U_1, U_2)$ along with the approximate output probability of the tuple.

We use MapFromSet as a subroutine for Greedy in Figure 6. The algorithm Greedy, chooses an initial guess for the set and incrementally chooses a value in the set and a value out of the set to swap. On each iteration, it makes the swap which increases the likelihood of the set the most. Once it can find no more improving swaps, the function returns the mapping for the best set that it iterated to. In our experiments, we run Greedy with random starting points 5 times and take the most likely mapping returned, and we cap the number of iterations of the outer loop to $1/\epsilon = 1000$.

**Genetic Algorithm.** We also use MapFromSet as a subroutine for Genetic in Figure 7. Our algorithm is inspired by genetic algorithms from machine learning. At a high level, the goal is to treat the probability as a “fitness function,” and like genetic selection, we keep “fit” sets and also randomly mutate sets across generations. Over many generations, the goal is to achieve a more fit population and to pick out the best plaintext mapping from the last generation.

In addition to the typical $\epsilon$, Genetic is parameterized by $k, g, t$, and $\gamma$, which correspond to the population size, generation number, tournament size, and mutation rate respectively. Once these are specified, the algorithm generates $k$ sets of candidate plaintext set guesses and then proceeds to run for $g$ generations, each time testing the likelihood of all candidates, keeping the top 10%, a tournament selected 80% (which get mutated), and finally randomly selecting 10%, to keep the population size at $k$. After $g$ generations, the function turns the most likely set into a solution for the incomplete cross-column problem using MapFromSet and returns the result.

**Split Algorithm.** Unlike our incremental optimization, we give a single shot attack in Figure 8 as well as a slightly generalized version, which just takes the best output of a few runs. Our main leakage-abuse attack however, is Split$^{\phi_1, \phi_2}$, which is parameterized by a parameter $\epsilon$ used for the POAA algorithm and distributions $\phi_1, \phi_2$ over the integers $[m + 1, n] = \{m, m + 1, \ldots, n\}$ (or any superset of this set).

The high level idea is to use the distributions $\phi_1, \phi_2$ as “guesses” of the underlying distribution of the data in the two columns. The algorithm will approximately partition (split) this guessed distribution first, then construct two new observation vectors with terms $c_i'$ and $d_i'$ respectively based on the partitioning returned. These new vectors create an instance of a complete cross-column equality pattern, and so we can run our optimal attack BM with them. Once, we get back a function $g$, we find the values corresponding to the equality groups $f$ and the plaintext values which correspond to the sets previously split sets.

Because this attack runs in one shot, it is also easy to enumerate over many distribution pair guesses and find one which
This is what we present as Split in some tables, due to the COVID pandemic, which may have an impact on the accuracy of our auxiliary distributions in the attacks. We test the ability of an adversary observing the join access pattern on the Chicago server to infer the values in the tables joined on, using the 2018 data as a prior distribution.

Between the taxi and crimes tables, we simulate a join between the pickup community area (CA) and the community area in which a crime occurred as well as the drop-off community area and the community area of a crime. Between the crime and crashes tables, we simulate a join between the beat in which a crime occurred and the beat in which a crash occurred.

For each of these potential joins and every year 2019-2022, we use the 2018 year table as an auxiliary distribution to estimate the column’s plaintext distribution for the year we are attacking. Then we calculate the cross-column leakage for the year and the columns we are joining. Finally, we take the auxiliary distributions and leakage and run our attacks Greedy\(^{\Phi}\), Genetic\(^{\Phi}\), and Split\(^{\Phi}\). We compare against a “no-cross” attack which runs SelAttack\(^{\Phi}\) on each column’s equality group leakage, runs POAA\(^{\Phi}\) to partition the remaining plaintext values, and returns the more likely of the two plaintext mappings. We include this to compare between cross-column and column equality more directly, to see how much more devastating an adversary is with the additional leakage between columns.

**Evaluation.** We evaluate our results using the value and row recovery rates just as in Section 4. For cross-column leakage-abuse attacks, the scoring formulas for the value and row recovery rates in an experiment with \(m\) observed values are
\[
\frac{|\{i \in [m] : f(m) = g(m)\}|}{m},
\]
and
\[
\frac{|\{i \in [N_1 + N_2] : r_i \in \text{Im}(f) \land f^{-1}(r_i) = g^{-1}(r_i)\}|}{N_1 + N_2 - (u_1 + u_2)},
\]
respectively, where \(g\) is the output plaintext mapping, \(f\) is the true plaintext mapping, and the join takes place on columns \((r_1, \ldots, r_{N_1})\) and \((r_{N_1+1}, \ldots, r_{N_1+N_2})\). These two formulae capture the fraction of values correctly identified and fraction of rows correctly identified respectively.

**Results.** We summarize some of our cross-column attacks in Figure 9. For all of our attacks we take \(\varepsilon = 0.001\). For our Genetic\(^{\Phi}\) algorithm, we set \(k = 30, g = 15, t = 5\), and \(\gamma = 0.2m\), and for Split\(^{\Phi}\) we try all pairs of the uniform distribution, zipfian distribution, and tail of the auxiliary distribution for each column (for a total of 9 calls to Split\(^{\Phi}\)). The “no-cross” attack which does not use the the cross-column equality pattern and instead just uses the column equality for each column. Specifically, it runs SelAttack\(^{\Phi}\) on each column and uses POAA\(^{\Phi}\) to partition any remaining values between the columns. It then returns the more likely triple between the results from the first and second columns.

For every experiment, except Crime vs Crash Beat, each attack we ran finished in under 30 seconds and usually in

\[\text{Alg Split}^{\Phi, \Phi_2}\left(\vec{c}_1, \vec{c}_2, \rho_1, \rho_2\right)\]

Parse \((c_1, \ldots, c_m, u_1) \leftarrow \vec{c}_1\) and \((d_1, \ldots, d_m, u_2) \leftarrow \vec{c}_2\)

\((S_1, S_2) \leftarrow \text{POAA}^{\Phi}(|m + 1, n|, \Phi_1, \Phi_2, u_1, u_2)\)

Let \(c'_i = \begin{cases} c_i & \text{for } 1 \leq i \leq m \\ (u_1 + u_2) \cdot \phi_1(i) & \text{if } i \in S_1 \\ 0 & \text{if } i \notin S_1 \end{cases}\)

Let \(d'_i = \begin{cases} d_i & \text{for } 1 \leq i \leq m \\ (u_1 + u_2) \cdot \phi_2(i) & \text{if } i \in S_2 \\ 0 & \text{if } i \notin S_2 \end{cases}\)

\(g \leftarrow \text{BM}(\vec{c}'_1, \ldots, \vec{c}'_n, 0, (d'_1, \ldots, d'_m, 0), \rho_1, \rho_2)\)

Let \(j(i) = g(i)\) for \(1 \leq i \leq m\)

Let \(U_j = \{j(i) : i \in S_j\}\) for \(j = 1, 2\)

Return \((f, U_1, U_2)\)

\[\text{Alg Split}^{\Phi}\left(\vec{c}_1, \vec{c}_2, \rho_1, \rho_2\right)\]

For each \((\Phi_1, \Phi_2) \in \Phi\) do

\((f, U_1, U_2) \leftarrow \text{Split}^{\Phi_1, \Phi_2}(\vec{c}_1, \vec{c}_2, \rho_1, \rho_2)\)

\(\rho_0, \rho_2 \leftarrow \text{Pr}[\vec{c}_1, \vec{c}_2 = f = f, U_1 = U_1, U_2 = U_2, \rho_1, \rho_2]\)

\((\Phi'_1, \Phi'_2) \leftarrow \text{arg max}_{(\Phi_1, \Phi_2) \in \Phi} \rho_0, \rho_2\)

Return \((\text{Split}^{\Phi'_1, \Phi'_2}(\vec{c}_1, \vec{c}_2, \rho_1, \rho_2)\)

Figure 8: The Split\(^{\Phi_1, \Phi_2}\) attack against incomplete cross column equality parameterized by distributions \(\Phi_1, \Phi_2\). The second Split\(^{\Phi}\) attack is parameterized by a set of distribution pairs \(\Phi\), and returns the best solution from among those pairs.

5.3 Experimental Evaluation

**Setup.** We test the practical performance of our leakage-abuse attacks against join leakage from real-world data. We use public data from the city of Chicago\(^{5}\) which has multiple datasets with columns of common types. We use the taxi, crime, rideshare, and vehicle crash tables from the years 2018-2022 to provide join leakage.\(^6\) Further details about the data used in these and other experiments can be found in Appendix D.

As in Section 4, we simulate a scenario where Chicago decides, in 2019, to keep their data more private. Specifically, they begin using a Structured Encryption for SQL scheme for the tables in question and allow authorized users to query the

\(^{5}\)https://data.cityofchicago.org/

\(^{6}\)Note that there is less (and different) data for the 2020 and 2021 years in some tables, due to the COVID pandemic, which may have an impact on the accuracy of our auxiliary distributions in the attacks.
less than a second on an Intel i9 Macbook Pro, without significant optimization. For Crime vs Crash Beat, our no-cross and Genetic\(ε, k, g, t\) algorithms took 3-4 minutes, and Greedy\(ε\) took about an hour. Fortunately, our (overall) best algorithm Split\(ε, Φ\) is also our most efficient! It only took about 5 seconds to run on Crime vs Crash Beat instances and was less than half a second on all of the other experiments.

Based on our experiments, the cross-column equality pattern allows an adversary to recover substantially more than attacks using only the equality leakage from each column. This is most highlighted by a simulated join between the “beats” column from the crimes table and the one from the crash table. The Split\(ε, Φ\) algorithm vastly out-performs all other others, recovering over 15% of values and rows on average, versus the others which are all under 5%. This pattern is displayed across our other joins as well, although often to a lesser extent. And, the baseline is generally good at recovering the values which occur in many rows, but additional cross-column information still improves that recovery.

In a deployment of a modern Ste for SQL scheme, this cross-column leakage is revealed entirely when a join is performed on the server, which only takes one query. So, when a database is stored with Ste for SQL, it is important to understand what a passive adversary could infer, given some prior distributional information. We conclude that if a client wants to keep adversaries from inferring underlying plaintext values, then Ste for SQL schemes may give insufficient protection, especially in settings where databases store sensitive information.

6 Conclusion

Based on our theoretical and empirical results, we find that modern Structured Encryption for SQL schemes \([13, 32, 49]\) are often not significantly more secure than older schemes supporting SQL with property-preserving encryption. In some contexts, this level of security carries acceptable risk and can still be used. However, one should take caution using these schemes for databases which contain sensitive information.

In our introduction, we also discuss some of the limitations of these findings and possible mitigations. Specifically, using certain primitives \([33, 40]\) or “heavy weight” cryptography \([5, 8, 21, 24]\) can be used to help with mitigation. However, all the solutions proposed come at some cost to the server or client computation and/or the bandwidth.

Acknowledgments

Support for this work’s researchers was provided primarily by DSO National Laboratories. We thank the high-school interns whose exploratory work helped inspired this project: Ming Kee Kang, Ananya Kharbanda, Allison Law, Jemma Lee, Megan Lee, Chelsea Ling Xinyi, Cadence Loh, and Naomi Wang. We thank Eileen Ee and Ryan Seah for their help supporting the interns, and thank David Cash for writing advice.

Availability

Our code can be found at https://github.com/ste4sql/LAA4STE4SQL. Appendix D contains details about data used.
References


[19] Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Saurabh Shintre. SEAL: Attack mitigation for encrypted databases via adjustable leakage. In Srdjan Capkun and Franziska Roesner, editors,


A Column Equality Theory

This section contains theorems and proofs omitted from the main text for brevity. We recall an attack from prior work in Figure 10. The following definition and lemma are useful in proofs below.

Definition 2. Let $V$ be an $n$ element set, $m \leq n$ be an integer, $\vec{c} = (c_1, \ldots, c_m, u) \in \mathbb{Z}^{m+1}$, and $\rho \in \text{Dist}(V)$. We call a function $f : [m] \to V$ non-crossing if for any $i, j \in [m]$,

$$\rho(f(i)) < \rho(f(j)) \implies c_i \leq c_j \quad \text{and} \quad c_i < c_j \implies \rho(f(i)) \leq \rho(f(j)).$$

Lemma 1. Let $V$ be an $n$ element set, $m \leq n$ be an integer, $\vec{c} = (c_1, \ldots, c_m, u) \in \mathbb{Z}^{m+1}$, and $\rho \in \text{Dist}(V)$. Then, if $f$ maximizes $\text{Pr}[\vec{c}|f = f, \rho]$, then $f$ is non-crossing.

We defer the proof of Lemma 1 to the full version.

Theorem 3. For every $n$ element set $V$, $\vec{c} \in \mathbb{Z}^n \times \{0\}$ and $\rho \in \text{Dist}(V)$, and $p > 1 \text{ L}_p(\vec{c}, \rho)$ outputs a plaintext mapping $g$ such that $\text{Pr}[\vec{c}|f = g, \rho] = \max_f \text{Pr}[\vec{c}|f = f, \rho]$.

We defer the proof of Theorem 3 to the full version.

Theorem 4. There exists a 4 element set $V$, $\vec{c} \in \mathbb{Z}^3$, and $\rho \in \text{Dist}(V)$, such that $L_1(\vec{c}, \rho)$ may output a plaintext mapping $g$ such that $\text{Pr}[\vec{c}|f = g, \rho] < \max_f \text{Pr}[\vec{c}|f = f, \rho]$.

We defer the proof of Theorem 4 to the full version.

Theorem 5. Let $\epsilon > 0$, $V$ be an $n$ element set, $m \leq n$ be an integer, $\vec{c} = (c_1, \ldots, c_m, u) \in \mathbb{Z}^{m+1}$ (with $u \geq 1$) and $\rho \in \text{Dist}(V)$ with $\rho(v) \geq \epsilon$ for all $v \in V$. Then, $\text{SelAttack}^k(\vec{c}, \rho)$ outputs a plaintext mapping $g : [m] \to V$ such that

$$\text{Pr}[\vec{c}|f = g, \rho] \geq (\max_f \text{Pr}[\vec{c}|f = f, \rho]) - O(u(n - m)\epsilon).$$
The product of two multinomial coefficients, with $N_1 = u_1 + \sum c_i$ and $N_2 = u_2 + \sum d_i$.

We go on to define the DICI problem below, which is strictly easier than finding the maximizing triple $(f, U_1, U_2)$.

**Name:** Decision Incomplete Cross Inference (DICI)

**Given:** Two integers with $m < n$, a number $t \in [0, 1]$, $\hat{c}_1 \in [n]^{m+1}$ and $\hat{c}_2 \in [n]^{m+1}$ and distributions $\mathbf{p}_1$ and $\mathbf{p}_2$ over $[n]$.

**Question:** Does there exist a triple $(f, U_1, U_2)$ such that $\Pr[\hat{c}_1, \hat{c}_2 | f = f, U_1 = U_1, U_2 = U_2, \mathbf{p}_1, \mathbf{p}_2] \geq t$?

**Theorem 6.** Decision Incomplete Cross Inference (DICI) is NP-hard.

We defer the proof of Theorem 6 to the full version.

## D Experimental Data and Results

In our experiments, we use data made public by the city of Chicago.² In particular, we use datasets of crimes,³ taxi rides,⁴ rideshare services,⁵ and car crashes⁶ for years between 2019 and 2022. Information about the number of rows can be found in Figure 12. The lack of data for the 2020 and 2021 years likely negatively impacted our experiments, but we were still able to attack these anomalous years.

---

²Formally, this problem only takes in $t$ and distributions which can be described using poly$(n)$ bits, rather than all real numbers.
³https://data.cityofchicago.org/
⁴https://data.cityofchicago.org/Public-Safety/Crimes-2001-to-Present/ijzp-q8t2
⁵https://data.cityofchicago.org/Transportation/Taxi-Trips/wrvz-psww
mödm-c72p
⁷https://data.cityofchicago.org/Transportation/Traffic-Crashes-Crashes/85ca-t31f