Mitigation Techniques for Attacks on 1-Dimensional Databases that Support Range Queries

Evangelia Anna Markatou and Roberto Tamassia

Brown University, Providence RI 02912, USA,
markatou@brown.edu, rt@cs.brown.edu

Abstract. In recent years, a number of attacks have been developed that can reconstruct encrypted one-dimensional databases that support range queries under the persistent passive adversary model. These attacks allow an (honest but curious) adversary (such as the cloud provider) to find the order of the elements in the database and, in some cases, to even *reconstruct* the database itself.
In this paper we present two mitigation techniques to make it harder for the adversary to reconstruct the database. The first technique makes it impossible for an adversary to reconstruct the values stored in the database with an error smaller than $k/2$, for k chosen by the client. By fine-tuning k, the user can increase the adversary's error at will.
The second technique is targeted towards adversaries who have managed to learn the distribution of the queries issued. Such adversaries may be able to reconstruct most of the database after seeing a very small (i.e. poly-logarithmic) number of queries. To neutralize such adversaries, our technique turns the database to a circular buffer. All known techniques that exploit knowledge of distribution fail, and no technique can determine which record is first (or last) based on access pattern leakage.

Keywords: Searchable Encryption · Encrypted Databases · Leakage-Abuse Attacks · Mitigation

1 Introduction

Currently several organizations outsource their data, and sometimes their entire IT departments to the cloud. This is a reasonable choice as the cloud usually provides a reliable, inexpensive, and safe place to store an organization's data. However, although data stored on the cloud are usually safe from outside attackers, they are sometimes vulnerable to the prying eyes of curious insiders. To protect their data from curious cloud providers, some organizations choose to store their data encrypted, using searchable encryption. This extra encryption step makes sure that the data can not be seen (at least in plain text) by curious cloud providers.

Although, at a first glance, searchable encryption seems to ensure the confidentiality of the data, unfortunately, this is not true. Recent papers have

demonstrated several attacks against encrypted databases that allow for range queries on the data, attacks which leak various amounts of information ranging in some cases to full database reconstruction. For example, Kellaris et al. [14] have demonstrated that they can achieve full database reconstruction after observing $O(N^4 \log N)$ range queries where N is the range of the data. More recently, Grubbs et al. [11] have improved in this result and have shown that they can achieve full database reconstruction in $O(N^2 \log N)$ time. To make matters worse, they have also shown that approximate[1] database reconstruction can be done very fast: only in poly-logarithmic time. Although *approximate* database reconstruction reconstructs only a portion of the database, still this portion is usually large enough and reveals a lot of the data stored in the "encrypted" database.

The literature presents plenty of attacks on other various types of leakage as well. For example, Kornaropoulos, Papamanthou and Tamassia [15] developed an approximate reconstruction attack utilizing leakage from k-nearest neighborhood queries. Grubbs, Lacharité, Minaud, and Paterson [10] utilize volume leakage from responses to range queries to achieve full database reconstruction. Grubbs, Ristenpart, and Shmatikov [13] present a snapshot attack that can break the claimed security guarantees of encrypted databases.

All this previous research suggests that even if the database is stored encrypted, a database reconstruction may be possible in reasonable time.

Having realized that current searchable encryption approaches provide little protection against powerful attackers such as *honest but curious cloud providers*, in this paper we take a slightly different approach and explore whether it is possible to make the task of the attacker a bit more difficult by introducing some form of *noise* or some kind of error by *changing* the queries issued by the clients. For example, when a client issues query $[a, b]$, our methods issue query $[a', b']$ (where $a \neq a'$ and/or $b \neq b'$) or our methods issue multiple queries. The choice of a' and b' is done in such as way so as to *obfuscate* the real query that the clients want to issue and thus *confuse* the attacker.

So far, all attacks either assume that the client issues queries uniformly at random ([11], [14]), or that the adversary has access to all possible query responses ([10], [16]).

1.1 Contributions

In this paper we present two *obfuscation* techniques.

1. **Blocked Queries** or **BQ**.
   Our first technique modifies the queries that the client can issue.
   The client selects an integer k. When the client wants to issue a query $[a, b]$, **BQ** rounds a down to the nearest smaller multiple of k, and rounds b up to

[1] Reconstruction of most (but not all) database elements with a small (asymptotically small) error.

the nearest larger multiple of k. That is, instead of issuing query $[a, b]$, the client issues query

$$[k \cdot \lfloor a/k \rfloor , k \cdot \lceil b/k \rceil] .$$

This way, the adversary can only *approximately* reconstruct the database. Indeed, we will show that for each record r_i which corresponds to value v_i the adversary can estimate v_i up to an error of $k/2$.
The client can pick k as they desire. Note that k introduces a trade-off between communication complexity and security: the higher the value of k, the higher the error in the adversary's approximation, but also the larger the range being queried, which in denser databases can increase the communication complexity - i.e. the amount of data being transferred.

2. **Wrap-around Queries** or **WQ**.
Our second approach is focused towards cases where the adversary knows the distribution of the queries. Indeed, a large body of previous work on attacks (including [11] and [14]) assumes that clients issue queries *uniformly at random*. That is, under this *uniformly at random* assumption, all possible queries $[a, b]$ (where $a \leq b$) have the same probability of being issued. Using this assumption, previous work managed to achieve (almost) full database reconstruction in poly-logarithmic time with only a very small amount of error of $O(1/\log N)$.
Therefore, protecting the privacy of the data in settings when the adversary knows the distribution of the queries is of paramount importance.

   To obfuscate the database results, when the query distribution is known, we use a four-pronged approach:
   - We introduce the notion of *wrap-around* queries. In normal queries $[a, b]$ we always assume that $a \leq b$. In contrary, in *wrap-around* queries $[a, b]$ we assume that $a > b$.
     In such cases the result of a *wrap around* query $[a, b]$ is the union of the results of normal queries $[a, N]$ and $[1, b]$. That is, a *wrap-around* query, as the name goes, *wraps around* the end of the value range (i.e. N) and continues from the beginning (i.e. 1).
     One may imagine that wrap-around queries treat the data not as a vector (from 1 to N) but as a cyclic buffer. The size of the buffer is still N, but the start of the cyclic buffer is not known. Actually, in a cyclic buffer, much like in a circle, there is no start (or end for that matter).

   - Approximately each time a client issues query $[a, b]$, **WQ** issues a second query: $[a', b']$.[2] The purpose of this second query is to confuse the adversary who will not be able to say whether the original query were $[a, b]$

[2] Depending on the distribution, **WQ** may need to issue several queries. For the purposes of discussion, at this point we assume that just one extra query $[a', b']$ is issued.

or $[a', b']$. Note, that query $[a', b']$ has to be a bit more sophisticated. Indeed, queries $[a', b']$ are taken from a suitable distribution so that when one combines all queries $[a, b]$ and $[a', b']$, the probability of each value $v_i \in [1, N]$ being queried is the same for all i. In this way, no value v_i is more popular than the other values, removing asymmetries previously exploited by adversaries.[3]

- Range queries $[a, b]$ are issued as singleton queries $[a, a]$, $[a + 1, a + 1]$, $[a + 2, a + 2]$, ... $[b, b]$. In this way the attacker will not be able to distinguish normal queries (which would have been issued as a single query) from wrapped-around ones (which would otherwise have been issued as two queries: $[a, N]$ and $[1, b]$).

- We always issue range queries in pairs. We deconstruct all range queries into singleton queries as above, and shuffle the singleton queries of each range query. We then issue all the singleton queries to the server.

The **BQ** approach aims to introduce an error in the reconstruction of any database. This makes exact reconstruction impossible for any adversary exploiting access pattern leakage.

The **WQ** approach aims to render unusable a large number of current attacks on encrypted databases, that assume that they know the query distribution. Indeed, using **WQ** we break the main assumption that attacks have made so far: The assumption that when queries are issued uniformly at random, some database values appearing more frequently than others.

Kellaris et al. [14] and Grubbs et al. [11] present attacks that inherently depend on the client issuing queries uniformly at random. These attacks no longer work as **WQ** removes any asymmetries that were being exploited in [14], [11] and similar papers.

2 Model

We consider a client that stores information on a database hosted by a server. A client can issue queries to the server using tokens, and the server issues responses.

Consider a collection of n records in the *database*. Each record (r, x) contains a unique identifier r in some set R, and a value $x = val(r)$ from some ordered set of integers X, on which range queries are performed, $X = [1, ..., N]$. We assume N is a multiple of k.

Given r the client can retrieve x. We define normal queries, that the user can issue, and wrapped queries, that they can't.

[3] Actually, the same should be true for all combinations of values v_i and v_j as we will later show.

Definition 1. *Normal Query*
A range query $[a, b]$, where $a \leq b$, is called a normal query. It returns the set of all matching identifiers, $M = \{r \in R : val(r) = x, x \in [a, b]\}$.

Definition 2. *Wrap-around Query*
A range query $[a, b]$, where $a > b$, is called a wrap-around query. It returns the set of all matching identifiers, $M = \{r \in R : val(r) = x, x \in [b, N] \cup [1, a]\}$.

The *adversarial model* we consider is a persistent passive adversary, able to observe all communication between the client and the server. The adversary aims to recover information about $val(r)$ for the different $r \in R$. We assume that the encryption scheme used has access pattern leakage that the adversary can abuse.

Definition 3. *Access Pattern Leakage*
If whenever the server responds to a query, the adversary observes the set of all matching identifiers, M, we say that the scheme allows for access pattern leakage.

3 Related Work

In recent years, there have been a number of papers on attacks on encrypted databases that support range queries and allow for access pattern leakage. This work started with a seminal paper by Kellaris et al. [14]. A few papers followed improving this work by Lacharite et al. [16], and Grubbs et al. [11].

There are also other attacks that focus on slightly different types of leakage. For example, Kornaropoulos et al. [15] utilize leakage from k-nearest neighborhood queries and Grubbs et al. [10] utilize volume leakage. Grubbs et al. [13] also present a snapshot attack.

There are also more attacks in the area, but they either assume a more active adversary or they assume that the encryption scheme reveals more properties ([1], [3], [9], [12], [18], [19]).

Additionally, there has been some work on mitigating attacks, but they don't prevent asymmetries caused by the client's query distribution [2]. Note that there are also much stronger mitigation techniques, like Oblivious Random Access Memory ([6], [7]) and Fully Homomorphic Encryption ([4], [5]). However these techniques are very costly.

4 Blocked Queries

In this section, we present our **BQ** technique (Figure 1). The goal of this technique is to introduce an error to any adversary's reconstruction of the database. To do so, when the client issues query $[a, b]$, our **BQ** system issues the superset query

$$[k \cdot \lfloor a/k \rfloor , k \cdot \lceil b/k \rceil] .$$

This way, no two queries can overlap in less than k positions.

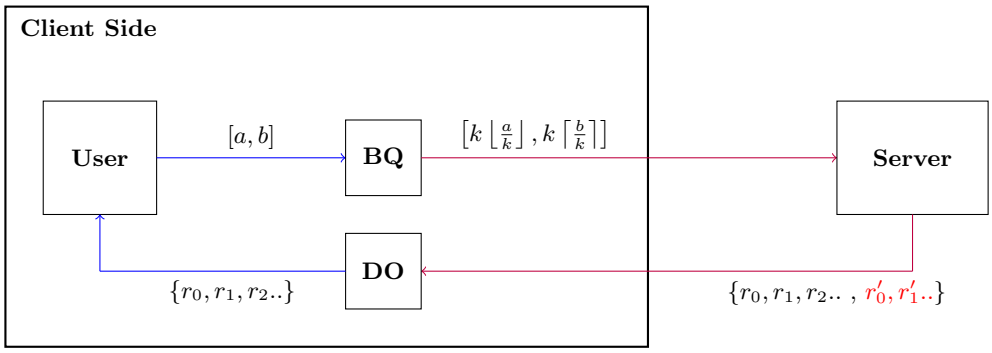


Fig. 1. The **BQ** technique: When the client wishes to issue a query **BQ** extends the query such as both endpoints are multiples of k. Once the server responds, the De-Obfuscation module (**DO**) removes any extra identifiers.

Algorithm 1 $BQ(a, b, k)$

1: Return $[k \cdot \lfloor a/k \rfloor , k \cdot \lceil b/k \rceil]$

Lemma 1. *Let R_a be the set of records with value $\in [a \cdot k, (a+1) \cdot k]$, for some integer a, $0 < a < N/k$. If one element of R_a is in a query response, all elements of R_a will also be in the query response.*

Proof. We will prove this by contradiction. Let r_1 and r_2 be two records whose values v_1 and v_2 are both in $[a \cdot k, (a+1) \cdot k]$. Suppose there exists some adversary that can deduce that $v_1 < v_2$.

The only leakage we consider here is access pattern leakage. In order for the adversary to distinguish between r_1 and r_2, they must observe some query response, which breaks the symmetry, that is it contains only one of the two values.

However, that is impossible as the client only issues queries that return all or none of the elements of $[a \cdot k, (a+1) \cdot k]$.

□

Definition 4. *Database D_k*

Let D be a database containing a set of records (r, x), that the user wishes to store. We construct database D_k by transforming the set of records (r, x) to $(r, \lfloor \frac{x}{k} \rfloor)$.

Theorem 1. *No adversary A can distinguish between the real database D, and D_k using access pattern leakage.*

Proof. Let A be an adversary that can distinguish between the real database D, and D_k using access pattern leakage. That means that there exists some query

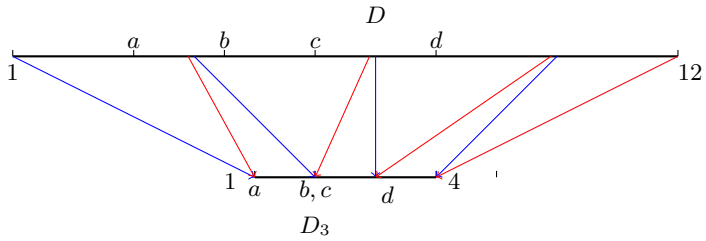


Fig. 2. Here we see D_3, for $N = 12$

response that can be observed when a user queries D that cannot be observed when they query D_k, or there exists some query response that can be observed when a user queries D_k that cannot be observed when they query D.

1. Let $[a, b]$ be a query whose results can be observed when a user queries D that cannot be observed when they query D_k.
   (a) Let a not be a multiple of k. Then, a will be turned into $k \cdot \lfloor a/k \rfloor$, which is a multiple of k. Thus, no query will be issued where a is not a multiple of k.
   (b) Let b not be a multiple of k. Similarly, for b, no query will be issued where b is not a multiple of k.
   (c) Let $a = k \cdot a'$ and $b = k \cdot b'$ be multiples of k. Then, query $[a, b]$ to D, results in the same response as query $[a', b']$ to D_k.

   It remains to show the other way. Let $[a, b]$ be a query whose results can be observed when a user queries D_k that cannot be observed when they query D. However, query $[a \cdot k, b \cdot k]$ to D results in the same response.

Thus, adversary A has no way of breaking the symmetry between database D and D_k using access pattern leakage.

□

Thus, no adversary can reconstruct the values of the database with an error smaller than $k/2$.

4.1 Overhead

An important point to address is how much this technique increases the communication complexity. Whenever the user wishes to make a query $[a, b]$, we issue the $[k \cdot \lfloor a/k \rfloor, k \cdot \lceil b/k \rceil]$.

The user asked us to query $b - a$ values in the database. Thus at most, we will query $b - a + 2k$ values in the database.

Note that the overhead of this scheme depends a lot on the choice of k, and the query distribution that the user picks. For example, if the user usually queries ranges of size $O(N)$, then k = $O(N)$, would introduce an error of size $O(N)$, while only increasing the overhead by a constant factor.

Let's take for example a user who issues queries uniformly at random (a common victim in literature). They can issue N queries of size 1, $N-1$ queries of size 2, and so on. So, the average size of query is

$$\frac{2}{N(N+1)}\sum_{i=1}^{N} i(N-i) = \frac{2}{N(N+1)}\frac{(N-1)N(N+1)}{6} = \frac{N}{3}$$

Thus, this user can pick $k = O(N)$, and the technique would introduce a constant factor overhead for an error of size $O(N)$.

5 Wrap-Around Queries

One assumption frequently made in this literature is that the adversary knows the query distribution. In most of these papers the query distribution is assumed to be "uniform at random". That is, all queries $[a, b], a \leq b$ have the same probability of being issued. Our second mitigation scheme, *wrap-around queries*, illustrated in Figure 3, is geared towards these adversaries.

This scheme turns the linear range of possible database values into a cyclic range, without a discernible beginning and end in the eyes of the adversary. Also, the scheme assumes that the user issues queries according to a fixed probability distribution and adds fictitious queries that correspond to a "complementary" distribution. Thus, the scheme converts the original query stream into one that is uniformly distributed over the cyclic range, thus removing any asymmetries on the number of times database values are present in query answers. As a result, by adopting this mitigation scheme, we prevent an adversary from distinguishing among any two cyclic shifts of the database from the access pattern leakage.

5.1 Key Ideas

The wrap-around queries technique depends on three key ideas:

1. Suppose the client wants to issue query $[a, b]$. The client issues instead a series of *singleton* queries $[a, a], [a + 1, a + 1]...[b, b]$.
2. The client issues additional queries to introduce noise. Approximately, every time the client issues a query, it issues one or more additional fictitious queries. The fictitious queries may be normal queries (i.e., $[a', b']$ where $a' \leq b'$) or wrap-around queries (i.e., $[a', b']$ where $a' > b'$).
3. The original normal query and fictitious queries are issued in pairs.

This technique essentially changes the ordered vector of values $[1, N]$ into a circular buffer of N values, which being a circular buffer, has no start and no end. This change (from a vector to a circular buffer) renders a persistent passive

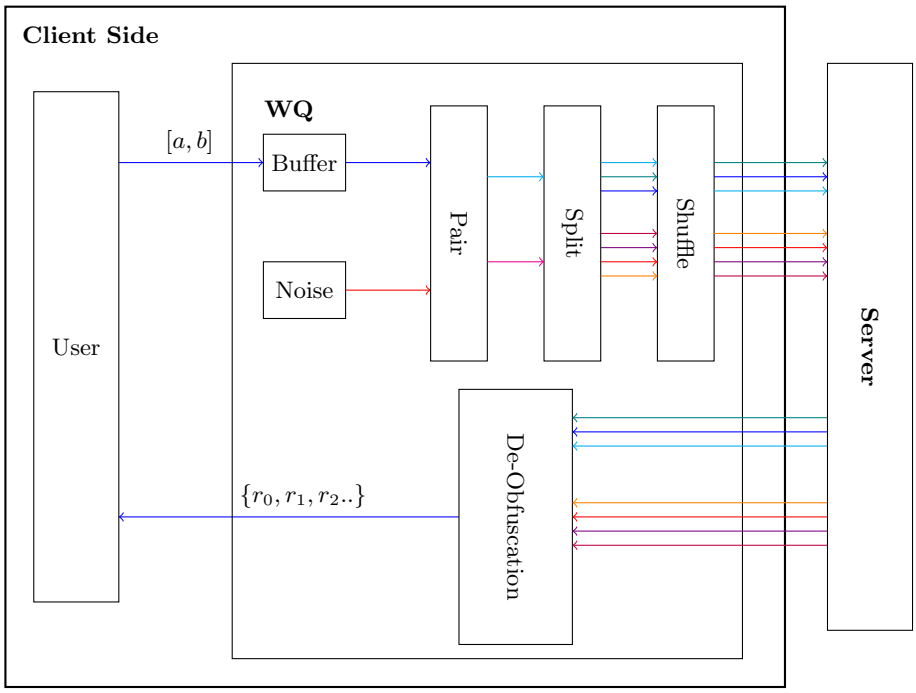


Fig. 3. The wrap-around queries module receives queries from the client and adds them to a buffer. It periodically constructs a pair of queries that contains both noise (i.e., fictitious queries) and real queries. Each query in the pair is further split into singleton queries (i.e., queries spanning a single value). Also, the singleton queries from each range query are shuffled. The shuffled queries are sent to the server. The de-obfuscation component receives the answers to the queries in the pair from the server, filters out the answers to noise, and reassembles the remaining singleton answers into answers to the original queries, which it forwards to the client.

adversary unable to determine which identifier corresponds to the minimum (or maximum) value.

In addition, the noise queries follow such a distribution so that all individual values $val(r_i)$ have the same frequency of appearance.

5.2 Example

We illustrate how the wrap-around queries scheme works with an example. Suppose we have a database of size 5 and the user issues queries that are uniformly distributed.

Let S_n be the set of normal queries that the client can issue (i.e. $[a, b]$ where $a \leq b$) and let S_w be the set of wrap-around queries that the client cannot issue (i.e. $[a, b]$ where $a > b$). The set S_n contains 15 queries and the set S_w contains 10 queries. All these queries are shown in Table 1. We use a buffer, B, to store

(a) Normal					(b) Wrap-around				
[1, 1]	[2, 2]	[3, 3]	[4, 4]	[5, 5]	-	-	-	-	-
[1, 2]	[2, 3]	[3, 4]	[4, 5]	-	-	-	-	-	[5, 1]
[1, 3]	[2, 4]	[3, 5]	-	-	-	-	-	[4, 1]	[5, 2]
[1, 4]	[2, 5]	-	-	-	-	-	[3, 1]	[4, 2]	[5, 3]
[1, 5]	-	-	-	-	-	[2, 1]	[3, 2]	[4, 3]	[5, 4]

Table 1. All possible cyclic queries on five elements: (a) normal queries; (b) wrap-around queries

the queries that the user wishes to issue. Let's acquire a biased coin. This coin with probability $\frac{15}{25}$ returns heads, and with probability $\frac{10}{25}$ returns tails.

We now pick a pair of queries. We flip the biased coin 2 times. On heads, we pick a range query from buffer B,[4] and on tails we pick a noise query, which is a wrap-around query selected uniformly at random from S_w. Thus, after each coin flip, we add a query to the pair. Once the pair is formed, we deconstruct all queries into a series of singleton queries, shuffle the singletons, and issue them to the server. Once the server responds, we identify the responses corresponding to real queries and send them to the user.

In Table 2, we show how many times each single value (from 1 to 5) is expected to be included in a query in the pair. The probability of value 1 appearing in a query in the pair is $15/25 \cdot 5/15 + 10/25 \cdot 10/10 = 3/5$. The probability of value 3 appearing in a query in a given slot in the pair is $15/25 \cdot 9/15 + 10/25 \cdot 6/10 = 3/5$: exactly the same as the probability of value 1. The same happens for all values: the probability of a value i to appear in a query in a given slot is $3/5$:

[4] If the buffer is empty we choose a normal query from S_n uniformly at random.

the same probability for all values i.[5] Thus, all values have the same probability of being included in a query in a slot.

Value	Appearances	Normal Queries
1	5	[1,1], [1,2], [1,3], [1,4], [1,5]
2	8	[1,2], [1,3], [1,4], [1,5], [2,2], [2,3], [2,4], [2,5]
3	9	[1,3], [1,4] [1,5], [2,3], [2,4], [2,5], [3,3], [3,4], [3,5]
4	8	[1,4], [1,5], [2,4], [2,5], [3,4], [3,5], [4,4], [4,5]
5	5	[1,5], [2,5], [3,5], [4,5], [5,5]
Value	Appearances	Wrap-around Queries
1	10	$[5,1]$, $[4,1]$, $[3,1]$, $[2,1]$, $[5,2]$, $[4,2]$, $[3,2]$, $[5,3]$, $[4,3]$, $[5,4]$
2	7	$[5,2]$, $[4,2]$, $[3,2]$, $[5,3]$, $[4,3]$, $[5,4]$, $[2,1]$
3	6	$[2,1]$, $[3,1]$, $[3,2]$, $[4,3]$, $[5,3]$, $[5,4]$
4	7	$[2,1]$, $[3,1]$, $[4,1]$, $[3,2]$, $[4,2]$, $[4,3]$, $[5,4]$
5	10	$[2,1]$, $[3,1]$, $[4,1]$, $[5,1]$, $[3,2]$, $[4,2]$, $[5,2]$, $[4,3]$, $[5,3]$, $[5,4]$

Table 2. All values in the database appear with the same frequency in the query results. In this table we show in how many queries each value appears. That is, value "1" appears in 5 Normal Queries and 10 Wrap-around Queries (a total of 15). Similarly, value "2" appears 8 times in Normal Queries and 7 times in Wrap-around Queries (a total of 20). Each values appears 15 times in the Normal and Wrap-around Queries combined.

5.3 Algorithm

In this section, we describe algorithm **WQ**, which performs the wrap-around mitigation scheme for the case of a user that issues uniformly distributed range queries. **WQ** takes as input *buffer*, S_n, S_w, N. The *buffer* is a buffer that contains the queries that the client wishes to issue. S_w is the set of wrap-around queries and S_n is the set of normal queries. N is the size of the database.

As long as there are still queries in *buffer*, **WQ** will make a new pair of queries. For each slot in the pair, we flip a coin, and depending on the result either a normal or a wrap-around query takes up the slot.

If the coin instructs that a normal query takes up the slot, we pop one from the *buffer*. If the *buffer* is empty, we just pick a normal query uniformly at random. If the coin instructs that a wrap-around query takes up the slot, we pick one uniformly at random. Each query is split into singleton queries, and a random permutation of the singletons is added to the pair slot.

Once the pair is full, we query the server.

Upon arrival of the results, we discard any results of wrap-around or fake queries, and return the rest to the client.

[5] The reader might wonder that since there are five values (1 to 5), then each value should have probability 1/5 (not 3/5) to appear in the query results. We should note however that these queries are *range* queries that return more than one value.

Algorithm 2 $WQ(buffer, S_n, S_w, N)$

Let c be a biased coin with $Pr(c = 1) = \frac{N(N+1)}{2N^2} = 1 - Pr(c = 0)$

while $|buffer| > 0$ **do**
  $pair = []$
  for 2 rounds **do**
    Flip c
    if $c = 1$ **then**
      $[a, b] = buffer.pop()$
      // If $|buffer| = 0$, pick a query from S_n uniformly at random
    else
      Pick $[a, b]$ uniformly at random from S_w

    $singletons = []$
    for $i \in range(a, b)$ **do**
      Add query $[i, i]$ to $singletons$
    Add a random permutation of $singletons$ to $pair$

  Issue all queries from $pair$ to server

  if $pair$ contained a user query **then**
    Return the relevant server's responses to the user.

5.4 Analysis

Lemma 2. *When using Algorithm 2, a slot of the pair contains each query with the same probability.*

Proof. Table 3 shows all possible queries.

Table 3. All possible queries: Note that the blue queries are normal, and the red ones are wrap-around.

$[1,1]$	$[1,2]$	...	$[1,N-1]$	$[1,N]$
$[2,1]$	$[2,2]$	...	$[2,N-1]$	$[2,N]$
$[3,1]$	$[3,2]$	...	$[3,N-1]$	$[3,N]$
...	...	...	...	...
$[N-1,1]$	$[N-1,2]$	...	$[N-1,N-1]$	$[N-1,N]$
$[N,1]$	$[N,2]$	...	$[N,N-1]$	$[N,N]$

Let q_1 be a normal query. Note that there are $\frac{N(N+1)}{2}$ normal queries. In a given pair slot, query q_1 is issued with probability

$$\frac{N(N+1)}{2N^2} \cdot \frac{1}{\frac{N(N+1)}{2}} = \frac{1}{N^2}.$$

Now, let q_2 be a wrap-around query. Note that there are $N^2 - \frac{N(N+1)}{2}$ wrap-around queries.

In a given pair slot, query q_2 is issued with probability

$$\frac{N^2 - \frac{N(N+1)}{2}}{N^2} \cdot \left(\frac{1}{N^2 - \frac{N(N+1)}{2}}\right) = \frac{N(N-1)}{2N^2} \cdot \left(\frac{2}{N(N-1)}\right) = \frac{1}{N^2}$$

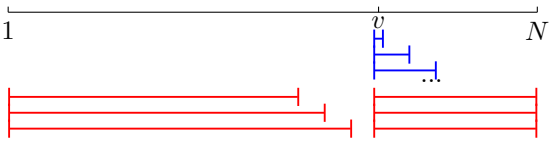
Thus, all queries are issued with the same probability $\frac{1}{N^2}$. □

Lemma 3. *When using Algorithm 2, each value in the database has the same probability of being in a given pair slot.*

Proof. Lemma 2 shows that all queries are issued with the same probability.

Let's look at all the queries that query some value, say v.

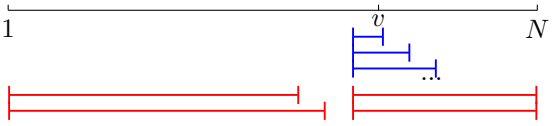
There are N queries that start at v and contain v.



$$[v, v], [v, v+1], [v, v+2]...[v, v-3], [v, v-2], [v, v-1] \text{ // N queries}$$

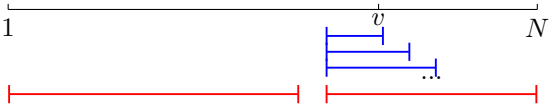
There are $N-1$ queries that start at $v-1$ and contain v:

$$[v-1, v], [v-1, v+1], [v-1, v+2]...[v-1, v-3], [v-1, v-2]$$



There are $N-2$ queries that start at $v-2$ and contain v:

$$[v-2, v], [v-2, v+1], [v-2, v+2]...[v-2, v-3]$$



Thus, there are

$$N + N - 1 + N - 2 + ... + 1 = \frac{N(N+1)}{2}$$

queries that query value v.

So, for any v there are $\frac{N(N+1)}{2}$ queries that query it.

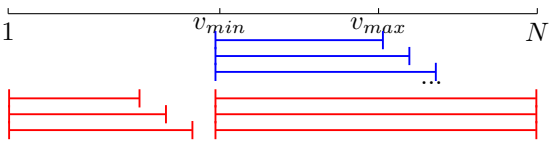
Since all queries are issued with the same probability (Lemma 2), all values v have the same probability of being queried in given a slot pair.

□

Lemma 4. *When using Algorithm 2, every contiguous set of points of size s has the same probability of being queried in a given pair slot.*

Proof. Let's look at some set of records S, and at the queries that return all elements of the set. Suppose that the smallest value of an element in S is v_{min} and the largest is v_{max}.

There are $N - s$ queries that return all elements of S and start at v_{min}.



$[v_{min}, v_{max}], [v_{min}, v_{max}+1], [v_{min}, v_{max}+2]...[v_{min}, v_{min}-3], [v_{min}, v_{min}-2], [v_{min}, v_{min}-1]$

Similarly to Lemma 3, there are

$$\frac{(N-s)(N-s+1)}{2}$$

queries that return all points from S. Thus, for any set of size s there are $\frac{(N-s)(N-s+1)}{2}$ queries that return all its elements.

□

Definition 5. *Database D_{shift_s}*

Let database D be a set of pairs (r, x). We define database D_{shift_s} as the set of pairs $(r, x + s(mod\ N))$, for all $(r, x) \in D$.

Theorem 2. *Consider a user that issues uniformly distributed range queries and uses the wrap-around mitigation scheme realized by Algorithm 2 to create a noisy query stream. A passive persistent adversary who observes the noisy query stream cannot distinguish with probability greater than 1/2 between the original database D and D_{shift_s}, for any s mod $N \neq 0$ using access pattern leakage and asymmetries created by the user's query distribution,.*

Proof. Suppose there exists some adversary that can distinguish between D and D_{shift_s} with probability greater than 1/2. The adversary has two weapons (i) the fact that a user that issues uniformly distributed range queries, and (ii) access pattern leakage.

As long as there are queries that the user wants to send out, Algorithm 2 will create pairs of queries to send to the server.

The Algorithm issues range queries as a random permutation of singleton queries. Note that the random permutation is necessary, as otherwise we leak the order of the records and the distances between them on a single query.

We have to send the queries in pairs. If we sent only one query at a time, the adversary would know that the last query we sent was a normal query.

1. Let us first examine access pattern leakage:

   Algorithm 2 will eventually issue all queries $[a, b]$ for $a, b \in [1, N]$ regardless of whether the database stored is D or D_{shift_s}. Now, because Algorithm 2 issues both normal and wrap-around queries, we can map a query from D to a query from D_{shift_s} such that both queries have the same response. To be exact, suppose query $[a, b]$ produces response $resp$ when issued to database D. Query $[a + s(\text{mod } N), b + s(\text{mod } N)]$ produces the same response $resp$ when issued to database D_{shift_s}. Thus, the adversary cannot get any information out of the access pattern leakage, the databases produce the same responses.

2. Now, let us examine what the adversary can accomplish using the fact that a user that issues uniformly distributed range queries.

   The adversary might try to use that fact, as due to the uniformity of queries the frequency with which values of the database are queried varies. To be exact elements in the middle of the database have a higher chance of being queried than elements in the ends. However, Algorithm 2, ensures that any set of contiguous points of a certain size is equally likely to be in a pair slot (Lemma 4). Thus, there are no asymmetries that the adversary could exploit to deduce which database is which.

Thus, neither uniformity nor access pattern leakage can help the adversary break the symmetry between the two databases, and the best they can do is guess, which succeeds with probability 1/2.

□

Corollary 1. *The adversary is unable to infer which record has the minimum (or maximum) value with probability greater than* $1/N$*, where* N *is the size of the database.*

Notably, after all their observations the adversary gains no knowledge on which record is the first one.

5.5 Different Client Query Distributions

This technique can be extended to work with any previously known client query distribution. Knowing the query distribution, we can construct a complimentary

distribution, such that when the adversary observes queries, they observe the equivalent of a uniform distribution over the queries for a cyclic buffer (Table 3).

5.6 Overhead

In order to examine how much overhead this approach introduces, we need to consider both the singleton queries and the noisy wrap-around queries.

1. *Singleton Queries*
   We send each query as a set of singleton queries. In first glance, this might appear to be a lot of communication overhead, however let us consider dense databases, an important type of database studied a lot in literature. When the user queries $[a, b]$, the database responds with at least $b - a$ records. Thus, in this interaction at least $b - a$ records are communicated. By sending singleton queries, we're merely doubling the communication complexity in dense databases. However, in general, we do increase the communication complexity proportionally to the size of the interval queried.

2. *Wrap-Around Queries*
   Whenever there is a query in the buffer, **WQ** sends a pair of queries to the server.
   For every 1 normal query picked to be in the pair there are $\frac{N-1}{N+1}$ wrap-around queries also picked. However, that is approximately a 1:1 ratio.
   So the wrap-around queries double the communication complexity.

In our best case scenario, when the user has a constant supply of queries and the database is dense, the overhead of the **WQ** scheme is merely a factor of 4.

5.7 Current Attacks

The **WQ** technique makes the adversary's job more difficult in two ways:

1. It removes asymmetries due to the client's query distribution.
   Exploiting these asymmetries has been a focus of Kellaris et al. [14] in their seminal paper on attacks. Grubbs et al. [11] present a very elegant algorithm as well that exploits this uniformity assumption. As we remove the asymmetries, these attacks can no longer reconstruct the database.

2. Most importantly, it turns the database into a cyclic buffer in the eyes of any adversary.
   No adversary exploiting access pattern leakage can tell which element is the first (or last) in the database. So, even if an adversary can reconstruct the shape of the database, they won't be able to recover the actual record values, unless they have access to auxiliary information.

5.8 Complexity of Encryption Scheme

Encryption schemes that allow for range queries can be costly. It's interesting to note that when running Algorithm 2, the server is never queried for a range that is larger than 1. Thus, a lot of the formerly required machinery is no longer necessary, the server can just store an encrypted dictionary. The dictionary would have N keys in the range $[1, N]$, whose values would be the corresponding records in the database.

5.9 Reducing Storage Complexity

Algorithm 2 stores singleton queries from two range queries. This can require a lot of storage.

To reduce this complexity, we can run Algorithm 2 in a streaming fashion. More specifically, instead of constructing and storing *pair*, we can use techniques for block ciphers with arbitrary block size to perform the pseudorandom permutation in sub-linear (in N) space, see for example [8], [17].

6 Conclusion

In recent years, a number of attacks have been developed that can fully or approximately reconstruct encrypted databases that allow for range queries. In an effort to mitigate the attacks, this paper presents two approaches to help better protect user data.

Out first approach, **BQ**, introduces an error to the reconstruction of the database by any adversary. Our second approach, **WQ**, is aimed specifically at a large percentage of current attacks, attacks that assume that the user queries the database uniformly at random. Our approach removes exploitable asymmetries caused by the uniformity rendering current attacks not usable.

References

1. Cash, D., Grubbs, P., Perry, J., Ristenpart, T.: Leakage-abuse attacks against searchable encryption. In: Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security. CCS '15, ACM, New York, NY, USA (2015)
2. Demertzis, I., Papadopoulos, S., Papapetrou, O., Deligiannakis, A., Garofalakis, M.: Practical private range search revisited. In: Proceedings of the 2016 International Conference on Management of Data. pp. 185–198. ACM (2016)
3. Durak, F.B., DuBuisson, T.M., Cash, D.: What else is revealed by order-revealing encryption? In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. CCS '16, ACM, New York, NY, USA (2016)
4. Gentry, C.: Computing arbitrary functions of encrypted data. Commun. ACM **53** (2010)

5. Gentry, C., Boneh, D.: A fully homomorphic encryption scheme, vol. 20:09. Stanford university Stanford (2009)
6. Goldreich, O.: Towards a theory of software protection and simulation by oblivious rams. In: Proceedings of the nineteenth annual ACM symposium on Theory of computing. pp. 182–194. ACM (1987)
7. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious rams. Journal of the ACM (JACM) **43**(3), 431–473 (1996)
8. Granboulan, L., Pornin, T.: Perfect block ciphers with small blocks. In: International Workshop on Fast Software Encryption. pp. 452–465. Springer (2007)
9. Grubbs, P., Sekniqi, K., Bindschaedler, V., Naveed, M., Ristenpart, T.: Leakage-abuse attacks against order-revealing encryption. In: 2017 IEEE Symposium on Security and Privacy (SP) (May 2017)
10. Grubbs, P., Lacharite, M.S., Minaud, B., Paterson, K.G.: Pump up the volume: Practical database reconstruction from volume leakage on range queries. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. CCS '18, ACM, New York, NY, USA (2018)
11. Grubbs, P., Lacharité, M.S., Minaud, B., Paterson, K.G.: Learning to reconstruct: Statistical learning theory and encrypted database attacks. Cryptology ePrint Archive, Report 2019/011 (2019), https://eprint.iacr.org/2019/011
12. Grubbs, P., McPherson, R., Naveed, M., Ristenpart, T., Shmatikov, V.: Breaking web applications built on top of encrypted data. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. CCS '16, ACM, New York, NY, USA (2016)
13. Grubbs, P., Ristenpart, T., Shmatikov, V.: Why your encrypted database is not secure. In: Proceedings of the 16th Workshop on Hot Topics in Operating Systems. HotOS '17, ACM, New York, NY, USA (2017)
14. Kellaris, G., Kollios, G., Nissim, K., O'Neill, A.: Generic attacks on secure outsourced databases. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 1329–1340. ACM (2016)
15. Kornaropoulos, E.M., Papamanthou, C., Tamassia, R.: Data recovery on encrypted databases with k-nearest neighbor query leakage. In: Proc. IEEE Symposium on Security and Privacy. pp. 245–262 (2019)
16. Lacharité, M.S., Minaud, B., Paterson, K.G.: Improved reconstruction attacks on encrypted data using range query leakage. In: 2018 IEEE Symposium on Security and Privacy (SP). pp. 297–314. IEEE (2018)
17. Morris, B., Rogaway, P., Stegers, T.: Deterministic encryption with the thorp shuffle. J. Cryptology **31**(2), 521–536 (2018). https://doi.org/10.1007/s00145-017-9262-z, https://doi.org/10.1007/s00145-017-9262-z
18. Pouliot, D., Wright, C.V.: The shadow nemesis: Inference attacks on efficiently deployable, efficiently searchable encryption. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. CCS '16, ACM, New York, NY, USA (2016)
19. Zhang, Y., Katz, J., Papamanthou, C.: All your queries are belong to us: The power of file-injection attacks on searchable encryption. In: 25th USENIX Security Symposium (USENIX Security 16). USENIX Association, Austin, TX (2016)