The primary goal of this work is to extend the semantics of oblivious storage. Previous work on OS has assumed that the client has some information about the keys that are present in the OS. An exception to this was the miss-tolerant solution in [7] where a client could perform lookups for non-existent keys. In this case the server would not learn that the key was a miss, and the client learns that the specific key does not exist. This interface makes it difficult to answer queries such as "give me all values where the key is in the range $[a, b]$", especially since it is possible that neither a nor b is in the dataset. This paper makes a significant step towards solving this problem, by providing an oblivious index that supports nearest neighbor queries including directional queries that are for nearest neighbor larger than (or smaller than) the query item. The non-directional version is simply: Given a key, find the keys that are closest to the given key. Note that the directional version can easily be used to find all keys in a range $[a, b]$, by finding the nearest successor to a (let it be x), then finding the nearest successor to x, etc. (In fact we can do much better than such a naive "follow the successor" approach, as will become apparent later in the paper.)

The rest of the paper is organized as follows: Section 2 describes related work. Section 3 gives the problem definition and defines the building blocks used in the paper. Section 4 describes the main result of this paper. Finally, section 5 concludes the paper.

2. Related Work

Oblivious RAM was introduced in [4]. In ORAM, the server has a sequence of values (pages in memory), $v_1, \ldots, v_n$. The client (who is also the data owner) can access an arbitrary value. Almost all of the solutions for ORAM provide an amortized performance guarantee. For example, in one solution proposed in [4] the cost of an access is $O(\sqrt{n})$ on average, but is $O(n)$ in the worst case. Many other schemes have been proposed to improve the efficiency of ORAM, including: [3, 5, 6, 10, 12, 13, 15]. The scheme in [6] is particularly interesting, because its worst case access time is sublinear.

In [2], a different model for oblivious outsourced storage was proposed called Oblivious Storage (OS), and this work was extended in [7]. In OS, the data store is a key-value store, which is a more natural framework than the RAM model. Another constraint of Oblivious Storage is to avoid increasing the server's storage by a multiplicative factor, as this will increase the cost of outsourcing significantly.

There is a growing list of papers in the framework of storage outsourcing (e.g.[8, 14], and others). [14] introduced the paradigm in which the service provider hosts the database as a service, and allows clients to store and access their own databases at the host site, which is similar to the framework in this paper. [8] describes several architectures that combine recent and non-standard cryptographic primitives in order to build a secure cloud storage service, and surveys the benefits such an architecture would provide to both customers and service providers.

The nearest neighbor search problem (also called the post-office problem by Knuth [9]) is a classic problem, and here we only review the related work of this problem in the secure outsourcing setting. Traditional encryption methods could hide the data from an untrusted server, but that would also prevent the client from doing queries like *nn* search or range queries, but prefix-preserving encryption (PPE) [11, 16] could help in handling *nn* search due to the fact that the longest common prefix of any two ciphertexts is of the same length as the longest common prefix of the corresponding plaintexts. However, the security is weakened since some prefix information is leaked to the server if PPE is used to encrypt the dataset.

Another recent work on similarity search [17] provides solutions for generic distance metrics (L_p norm) of multidimensional data with interesting trade-offs between query cost and accuracy, but it does not consider hiding the access patterns from the server. Several other related transformation-based techniques and hierarchy-based searches (using an encrypted R-tree to represent the database and then searching it for query point level by level) are proposed in location-based service (LBS) systems [18] which have the same issue of leaking access patterns.

3. Preliminaries

In this section, we begin by describing the notation used in this paper. The interval (x, y) includes all integers from x to y exclusive, and when the parenthesis are replaced by brackets (i.e., [or]) then the interval is inclusive. Given a value $x \in \{0,1\}^n$, we define $Prefix_m(x)$ to be the m most significant bits of x.

Our schemes utilize a pseudorandom function(PRF) $F : \{0,1\}^n \times \{0,1\}^n \rightarrow \{0,1\}^n$. We utilize the textbook definition for a PRF in that for all PPT distinguishing algorithms D, $|Pr[D^{F_k(\cdot)}(1^n) = 1] - Pr[D^{f(\cdot)}(1^n) = 1]$ is negligible in n where f is a random function. Our scheme utilizes a PRF that takes in variable length input tuples. This is easily accommodated by an encoding scheme that pads all messages to the same length. For example, all strings up to n bits long can be converted into a string of length $n + \log n$ by pre-pending the length and padding with 0's.

Finally, our scheme utilizes CPA-secure encryption schemes $(KeyGen, Enc, Dec)$ where an adversary cannot distinguish one of two ciphertexts given oracle access to Enc.

3.1. Framework/Problem Definition

We are assuming an honest but curious server, which means it will collaborate with the client and perform specified computations, but try to learn information about the client's data or access pattern. A data owner (client) that publishes a data set on the server, and the data owner wants to be able to query its own data while protecting the data from the server. This includes protecting both the content and the data access pattern. Previous work [2, 7] has introduced the concept of oblivious storage. In oblivious storage, the data owner publishes a key-value store on the server. More specifically, the server stores a set of tuples $S = \{(k_1, v_1), \ldots, (k_n, v_n)\}$ where $n \leq N$ for some size threshold N, the keys, k_i, are unique and are drawn from a key domain $[0, D-1]$, and the values, v_i, are drawn from a domain of values where each value has the same bit size (alternatively the values could be padded to have the same size).

The schemes in [2, 7] give protocol for functions: $get(k)$, $put(k, v)$, and $remove(k)$. In [7] oblivious stores are described as either miss-intolerant or miss-tolerant. In a miss-tolerant data store, the server does not learn whether a query is in the dataset or not, but this information is revealed to the server in a miss-intolerant oblivious store.

We seek to extend previous work in oblivious storage by adding the semantics of a nearest neighbors that returns the nearest predecessor and nearest successor of a key. Informally, this takes as input a value in $[0, D-1]$, and returns a tuple (np, ns) where np (resp. ns) is the largest (resp. smallest) key in S that is smaller (resp. no smaller) than the input. The efficiency goals are to minimize: i) the communication, ii) the computation, iii) the number of communication rounds, iv) the server storage, and v) the client storage.

Formally, our goal is to define an oblivious index structure that supports the following operations:

1. $insert(k)$ that takes a value $k \in [0, D-1]$ and inserts it into the structure.

2. $remove(k)$ that removes a value $k \in [0, D-1]$ from the structure.

3. $nn(k)$ returns (np, ns) where np is the largest value in the data set such that $np < k$, and ns is the smallest value such that $ns \geq k$.

In case there is no predecessor (resp. successor) we want to return special symbols that represent values $-\infty$ (resp. ∞), so that there is always an answer to this query. We also assume that there is an upper bound N on the number of keys in the oblivious store. The store should be oblivious in that the server should not learn which items are being accessed (that is the server should not be able to tell two access patterns apart). Furthermore, the $insert$ and $remove$ queries should be miss-tolerant, in that the server should not learn when an item is actually inserted or removed.

3.2. Details of Previous Protocol for Oblivious Storage

In this section we describe the high level details of previous work for miss-intolerant oblivious store. The previous work [7] has two phases: i) a query phase, and ii) a rebuilding phase. During the query phase, the client asks get, $insert$, and $remove$ queries, and after M queries, the rebuilding phase starts. During the rebuilding phase, query execution is suspended, and the server's storage is rebuilt.

The server stores N regular tuples and M dummy tuples. Each regular tuple corresponds to a key value pair, (k, v) and has the form $(F_{fk}(k), Enc_{ek}(v))$ where F is a pseudorandom function and Enc is a CPA-secure encryption scheme, and fk is a pseudorandom function key that changes during the rebuilding phase and ek is a key for the encryption scheme that does not change during the rebuilding phase. The dummy items are of the form $(F_{fk}(-i), Enc_{ek}(FAKE))$ for each value $i \in [1, M]$ where $FAKE$ is some padded dummy value. The items are stored in a random order.

The client has local storage of size $O(M)$ that keeps track of all queries made during the current query phase along with the answers to the queries (initially this local store starts out empty). These are stored in a data structure that allows $O(1)$ amortized insertions and searches by key.

To process a query $get(q)$, the client first searches its local store for q, and then:

- If q is in the local storage: The client sends a dummy query to the server. That is, the client sends $F_{fk}(-j)$ to the server where this is the jth dummy query sent to the server.

- Otherwise: The client sends $F_{fk}(q)$ to the server.

In either case the server obtains a value from the client that is in its dataset that has not been queried before. The server finds the value in its data set that matches the query, and sends the corresponding message back to the client. The server also removes this key-value pair from its data store. The client then stores this value in its local store and returns the result.

To process a query $insert(k, v)$, the client first issues a query $get(k)$ and then changes the value in its local store associated with key k to v. To process a query $remove(k)$, the client issues a query $get(k)$, and then removes the key-value pair from its local storage (note that it was already removed from the server). In both cases, the server only sees a get query and all other changes affect the client's local storage only. Note that this is for a miss-intolerant solution, and that this leaks to the server

when a value was replaced or inserted, and for removals the server learns when a value was actually removed.

After M queries, the rebuilding phase starts. In this phase the client reshuffles the values in the server storage, changes the pseudorandom function key, and re-encrypts the values. The values are randomly permuted to prevent the server from inferring information about the queries between two different query phases. The details of this shuffling process are in [7].

During the query phase the client and server perform $O(1)$ computation and communication per query. The server storage is $O(N)$. The cost of the rebuilding phase is $O(N)$, but the amortized cost per query is $O(\frac{N}{M})$. The client storage is $O(M)$. The number of communication rounds per query is $O(1)$. The base scheme in [7] sets M to $\sqrt{N}$, and thus the amortized cost is $O(\sqrt{N})$ and the client's storage is $O(\sqrt{N})$.

3.3. Observation

Providing a nearest neighbors oblivious index is at least as hard as providing miss-tolerance for *get* in the original data store. That is, suppose we have a miss-intolerant oblivious store, and a client queries $get(k)$. Simply call $(np, ns) \leftarrow nn(k)$ and $get(ns)$ (If ns is ∞ then use $get(np)$). The *get* will always be a hit, and the client can determine if k was a hit by testing if $k \stackrel{?}{=} ns$.

4. Nearest Neighbors Oblivious Index

In this section, we present the main result of this paper: an oblivious index structure for nearest neighbors. We utilize many of the ideas in the previous work on obvious storage. Let N denote the upper bound on the number of keys, let M denote the number of queries in the query phase, and let $[0, D-1]$ denote the key domain.

4.1. A straightforward protocol

The miss-intolerant data store in the previous section can be used to provide answers to *nn* queries. The client builds a balanced binary search tree over the key values to produce a tree with height h (Clearly $h = O(\log N)$). Each node in the tree is given a unique label, and the root node's label is a known constant. Each node in the tree has its children's labels along with the search value. The client can then perform a binary search to find the smallest key value that is not less than the query and the largest key that is not larger than the query. If the client finds the value in an intermediate node, or reaches a leaf node with height smaller than h, then the client performs the appropriate amount of extra queries to pad the number of queries to h (these extra queries can be repeated queries from before). This is necessary to make each query look identical to the server, otherwise the server would learn something about each query.

Furthermore, if insertions and removals can be performed while changing at most $O(h)$ nodes, then this can support insertion and removals using the *insert* and *remove* queries.

Suppose we set $M = \sqrt{N}$. The tree clearly has at most N nodes, and thus the server's storage is $O(N)$, the clients storage is $O(\sqrt{N})$, the query cost and communication is $O(\log N)$, and the number of rounds is $O(\log N)$. Finally, the amortized cost/communication is $O(\log N \sqrt{N})$. The main goal in the rest of this paper is to reduce the number of rounds to $O(1)$.

4.2. Server Storage

We are now ready to describe one of the main ideas of our proposed approach. Given S, the client partitions the key domain into a set of unique prefixes. Specifically a prefix, p, is interesting if all key values that share the prefix have the same nearest neighbors, but this is not true for any shorter prefix of p. More formally, p is interesting if $|\{nn(m) : Prefix_{|p|}(m) = p\}| = 1$ and $|\{nn(m) : Prefix_{|p|-1}(m) = Prefix_{|p|-1}(p)\}| > 1$. For each interesting prefix p with nearest predecessor np and nearest successor ns, the client creates a key value pair $(p, (np, ns))$. The client stores all such pairs on the server just as the key value pairs are stored in a miss-intolerant OS. That is the client stores $(F_{fk}(p), Enc_{ek}((p, np, ns))$ on the server where fk is a key for a pseudorandom function and ek is a key for a CPA-secure encryption scheme.

Let d denote the number of bits used to represent a value in the dataset. The following theorem places an upper bound on the number of interesting prefixes (and hence on the size of server storage).

Theorem 1. There are at most $Nd + 1$ interesting prefixes.

Proof: Let $T(n, k)$ denote the maximum number of interesting prefixes if there are n values and k bits. First note that $T(n, k)$ is well defined if and only if $2^k \geq n$ (otherwise there are not n values with k bits).

Obviously, $T(0, k) = 1$ and the claim holds.

Now, we show that $T(1, k) \leq k + 1$. Obviously, this is true for $k = 0$. Now consider, $T(1, k)$. The value is either starts with a 0 or a 1. The half that does not contain the value all have the same nearest neighbors. Thus $T(1, k) \leq 1 + T(1, k-1)$, and the claim follows by induction.

Consider $T(2, k)$. Now, $T(2, 1) = 2$ and so the claim holds for the base case. Now either both values start with the same bit, or they are both different. Hence, $T(2, k) \leq \max\{2T(1, k-1), T(2, k-1) + 1\}$. By induction, $T(2, k) \leq \max\{2k, 2(k-1) + 1\} \leq 2k + 1$, and the claim holds.

Now consider $T(n,k)$ for $n \geq 3$. Now, $T(n, \lceil \log n \rceil) \leq 2^{\lceil \log n \rceil} < 2n+1$ (the last part assumes $n \geq 3$. Now considering larger values of k, for some constant c, there will be c values with a 0 prefix and $n-c$ values with a 1 prefix. Thus $T(n,k) = T(c,k-1) + T(n-c,k-1)$. By induction, $T(n,k) \leq c(k-1)+1+(n-c)(k-1)+1 = nk-n+2 \leq nk+1$. The claim follows. □

The server will thus store $Nd+1+2M$ tuples. If there are ℓ interesting prefixes, there will be ℓ tuples for these prefixes, $Nd+1-\ell$ dummy prefixes (so that the server does not learn how many interesting prefixes there are), and $2M$ dummy prefixes that will be used to generate fake hits (the full details are described in a later section).

The main idea to process a query q is to issue a query for each prefix of q (i.e., to issue the query $Prefix_1(q), \ldots, Prefix_d(q)$ i.e. by sending $F_{fk}(Prefix_1(q)), \ldots, F_{fk}(Prefix_d(q))$ to the server. Exactly one of these queries will result in a hit and thus revealing the number of hits to the server does not reveal anything. The server will find the one tuple that is a match and send the value back to the client. Note that the above interaction can be done in a single communication round.

Example Suppose $d = 4$ (i.e., the keys consists of 4 bit values) and that $N = 4$. Suppose that following four keys are in S: 2, 6, 7, and 11. In Figure 1 we show the tree based representation of the key space and have highlighted the nodes corresponding to the interesting prefixes. In this case, the server would store the following key-value pairs: $(000, (-\infty, 2))$, $(0010, (-\infty, 2))$, $(0011, (2, 6))$, $(010, (2, 6))$, $(0110, (2, 6))$, $(0111, (6, 7))$, $(10, (7, 11))$, and $(11, (11, \infty))$. The server would also store 9 dummy values so that the server is storing 17 values. If the client issued the query $nn(8)$, then the client would issue queries 1, 10, 100, and 1000. Notice that the query 10 is a match, and that the client would learn that prefix 11 is a match, and the nearest predecessor is 7 and the nearest successor is 11.

However, these values should be permuted before sending them to the server to prevent leaking which prefix length is a match. There are some complications including: i) over M queries many prefixes will be queried repeatedly and this will leak information to the server, ii) it is possible that two different queries will result in the same hit and thus we need to avoid this leakage, and iii) insertions and removals need to be handled.

We finish by giving the details of a pair of algorithms that will be used later. The first algorithm, *PREFIXSPLIT*(x,y), partitions the interval $[x,y]$ into its set of prefixes that minimally cover the entire interval. The main idea is that if you view the interval as part of a tree where the leaves range form $[0, D-1]$ then the minimum number of nodes in the tree that covers the interval correspond to the off path vertices on the paths from the nearest common ancestor of x and y to x and y. The straightforward details for splitting an interval into interesting prefixes are presented in Algorithm 1.

Algorithm 1 $SPLIT(x = x_1 \cdots x_d, y = y_1 \cdots y_d))$

```
P ← {}
c ← 0
{Find common prefix}
while x_c = y_c do
    c ← c + 1
end while
{Since x < y, x_{c+1} = 0 and y_{c+1} = 1}
i ← d + 1
while x_{i-1} = 0 do
    i ← i − 1
end while
I ← I ∪ {x_1 ··· x_{i-1}}
for j = i − 2 to c + 2 do
    if x_j = 0 then
        I ← I ∪ {x_1 ··· x_{j-1} 1}
    end if
end for
i ← d + 1
while y_{i-1} = 1 do
    i ← i − 1
end while
I ← I ∪ {y_1 ··· y_{i-1}}
for j = i − 2 to c + 2 do
    if y_j = 1 then
        I ← I ∪ {y_1 ··· y_{j-1} 0}
    end if
end for
return I
```

We now turn our attention to generating all interesting intervals for a set of key values $K = \{k_1, \ldots, k_n\}$. If we assume these keys are sorted, then this partitions the key space into intervals $[0, k_1], [k_1+1, k_2], \ldots, [k_{n-1}+1, k_n], [k_n+1, D-1]$. Notice that all points in the interval $[k_i+1, k_{i+1}]$ all share the same nearest predecessor and successor. This algorithm simply sorts the points and calls the previous algorithms to find all interesting prefixes. The details are in Algorithm 2.

4.3. Data Structure 1: Avoiding duplicate queries

Two problems with the previous approach involve the client asking duplicate queries when processing two distinct *nn* queries. To be able to overcome the problems, the client needs to be able to determine (for the current query): i) the longest common prefix with any previously issued *nn* query in the query phase, and ii) has the prefix group of the current query already been obtained.

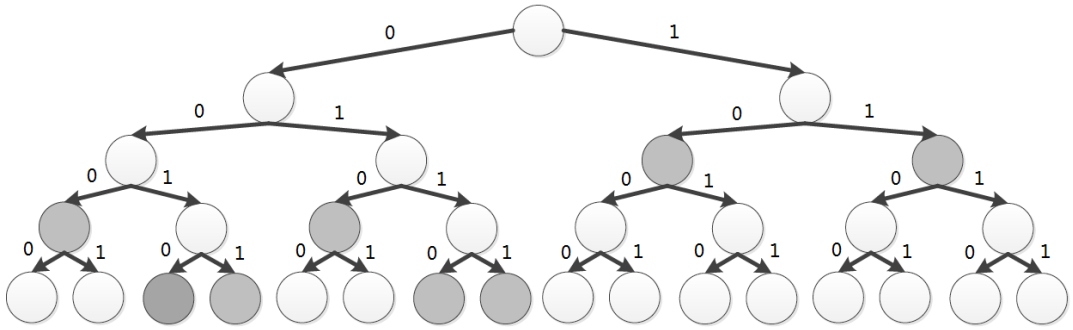

Figure 1. Interesting prefixes for {2,6,7,11} shown as the grey nodes

Algorithm 2 $ALLPREFIXES(\{k_1,\ldots,k_n\})$

1: $sk_1,\ldots,sk_n \leftarrow sort\{k_1,\ldots,k_n\}$
2: $A \leftarrow \{\}$
3: $S \leftarrow SPLIT(0, sk_1)$
4: $A \leftarrow \{(s,(-\infty,k_1)) : s \in S\}$
5: $S \leftarrow SPLIT(sk_n+1, D-1)$
6: $A \leftarrow \{(s,(sk_n,\infty)) : s \in S\}$
7: **for** $i = 1$ to $n-1$ **do**
8: $\quad S \leftarrow A \cup SPLIT(x_i+1, x_{i+1})$
9: $\quad A \leftarrow \{(s,(sk_i,sk_{i+1})) : s \in S\}$
10: **end for**
11: **return** A

Specifically, we desire a data structure with the following three operations:

1. $insert(q,mp,np,ns)$ this inserts the results of a previous query $nn(q)$. This stores: i) mp: the matching prefix of q, ii) np the nearest predecessor of q, and iii) ns the nearest successor of q.

2. $get(q)$ this returns a tuple (L,nn) where L is the length of the longest common prefix between q and any previous query, and nn is null if the prefix group containing q has not been queried, but is the nearest neighbors, (np,ns), for q if the query group has been found.

3. $initializeCommonQuery()$ This initializes the data structure to an empty structure.

The following theorem states this requires the ability to find the nearest successor and predecessor of the current query over all previously asked queries.

Theorem 2. Given a set of queries $S = \{s_1,\ldots,s_n\}$ and a query q, the longest common prefix of q to any query in S is either q's nearest predecessor or successor. Furthermore, if any query in S is in the same prefix group of q, then q's nearest successor or predecessor is also in the same prefix group as q.

Proof: Given two bit sequences x and y, let $LCP(x,y)$ denote the longest common prefix between x and y. First we show that if $x \le y < z$, that $|LCP(x,y)| \ge |LCP(x,z)|$. Denote $LCP(y,z)$ as $c_1\cdots c_p$. Now since $y < z$, it must be that y starts with $c_1\cdots c_p 0$ and z with $c_1\cdots c_p 1$. Since $x \le y$, x cannot start with the prefix $c_1\cdots c_p 1$, and thus $|LCP(x,z)| \le p$. Now consider, $LCP(x,y)$, there are two cases to consider: i) $|LCP(x,y)| \le p$, in which case $LCP(x,y) = LCP(x,z)$, and ii) $|LCP(x,y)| > p$, in which case $LCP(x,y) > LCP(x,z)$. In either case, the claim holds.

A symmetrical argument can be made that when $x < y \le z$, that $|LCP(y,z)| \ge |LCP(x,z)|$. Combining these two things together implies that the longest common prefix in the set is either the nearest successor or the nearest predecessor of the query. This proves the first part of the theorem. The second part follows because if two queries belong to the same prefix group, then the longest common prefix in the set must also be in the same prefix group. □

The client maintains a local data structure that stores values of the form (q,mp,np,ns) where q is the query, mp is the prefix of the q's prefix group, np is q's nearest predecessor, and ns is q's nearest successor. These values are stored in a balanced binary search tree organized by query. Given this structure the client can find the nearest neighbors of a specific query in $O(\log M)$ time.

Example Suppose that we use the example in Figure 1. Suppose that in a query phase a client first issues a query for 8, then the the client searches for prefixes 1, 10, 100, and 1000. It finds a match at prefix 10, and learns the nearest neighbors are $(7,11)$. Suppose that the client then asks query $ns(13)$. In this case the prefixes would be 1, 11, 110, and 1101. The client must avoid asking for the query 1, since the client has already asked this query in this phase. Thus the client asks for 11, 110, and 1101 along with a fake miss. The prefix 11 is the only match, and the client learns that 13's nearest neighbors are $(11,\infty)$. Finally, suppose that the client issues query $ns(9)$. The prefix groups would then be 1,10,100,1001. The first three of these groups have been asked by the client. Furthermore, the prefix group 10 is in the set S, and it indicates the nearest neighbors of 9 are $(7,11)$. Thus the client issues queries 1001, two fake

misses, and a fake hit. The fake hit is necessary to ensure that the server sees exactly one match in its dataset.

4.4. Data Structure 2: Handling Changes

The purpose of this data structure is to keep track of changes that have been made during a query phase. There are two main challenges: i) returning the correct answers in the rest of the query phase, ii) including the updates in the stored data in the rebuilding phase.

The main idea is that the client will keep track of all intervals where it knows the answer. That is, for every $nn(q)$ query, the client learns an interval $(np, ns]$ that contains q, and each value in $(np, ns]$ has the same nearest neighbors. Furthermore, all points outside of the interval $(np, ns]$ do not have the exact same nearest neighbors. This data structure will keep track of all such intervals that the client learns during the query phase. When modifying the dataset, the client will modify the local data structure, but leave the server's data (and its data from the first data structure unchanged). Hence, if this second data structure contains information about a specific interval. then this is considered more current than the values stored at the server. One could think of this data structure as a change log during the query phase.

We now give the details of insert and delete at a high level. To process a query $insert(q)$, first the client performs a $nn(q)$ query. Thus the interval containing this value will be in the local storage. Suppose that this interval is $(np, ns]$. The inserted value splits this interval into at most two intervals, and these intervals will replace the previous interval. That is, the process creates the intervals $(np, q]$ and $(q, ns]$.

To process $remove(q)$, the client will query the server for the removed value, and will thus have the interval containing the removed value in its local storage. Suppose this interval is $(np, ns]$. If $q \neq ns$, then nothing has to be done. However, if $q = ns$, then the client will have to mark the value ns as removed. A difficulty arises when a client queries a point inside of an interval where the end point has been removed, e.g., if the client later issued a query in the interval $(np, ns]$ after ns was removed. In this case, the client would know that the answer provided by the server is stale, but would not know the correct answer. To overcome this problem, the client first checks the second data structure to determine if this will be a problem. If so, then the client issued a query for the next interval. That is, in our example, the client would issue a query $nn(ns + 1)$ instead of $nn(q)$, and this will return the new nearest successor.

Specifically, the data structure will keep track of a set of intervals. An interval $[x, y]$ means that any point in the interval $(x, y]$ has x as its nearest predecessor and y as its successor. An interval is either marked valid or invalid. An interval is valid if y has not been removed and is invalid otherwise. Note that we ensure that the only points that are removed are endpoints of some interval in the client's structure. There are several operations that we want to perform with this structure, including:

1. $(ns, np, valid) \leftarrow lookup(x)$: This searches the interval list to find the interval containing x in the structure. If no such interval exists, then return null. Otherwise, if x is in a valid interval $[y, z]$, then return $(x, y, true)$. Otherwise, if x is in an invalid interval $[y, z]$, then return $(x, y, false)$.

2. $insertPoint(x)$: This has a precondition that there exists a valid interval containing x, let this interval be $(y, z]$. This interval is replaced with two valid intervals $(y, x]$ and $(x, z]$.

3. $removePoint(x)$: This has a precondition that there exists a valid interval containing x, let this interval be $(y, z]$. If $x \neq z$, then do nothing. Otherwise, mark this interval as invalid.

4. $insertInterval(x, y)$ This assumes that the interval $(x, y]$ does not overlap any current interval. If there is an invalid interval $(z, x]$, then this replaces this interval with a single valid interval $(z, y]$. Otherwise, this adds a single valid interval $(x, y]$.

5. The existence of an iterator function that allows us to iterate over all intervals in the structure (touching each interval once). This is encapsulated by the functions $first()$ which starts the iterator, and $next()$ which returns the next interval (and null if no such interval exists).

6. An initialization method, $initIntervalDS()$ that initializes an empty data structure.

The above data structure is straightforward to build using a balanced binary search tree (sorted by interval end point). If there are M intervals, then this structure has size $O(M)$. In this case lookup, insertions, and removals can be processed in $O(\log M)$ time. Furthermore, iterating over all intervals requires $O(M)$ time.

4.5. Putting Pieces Together

We are now ready to put all of the pieces together and give a detailed description of the system. We begin by highlighting the main ideas:

1. The data owner stores all interesting prefixes and their nearest predecessors and successors for that prefix, using similar techniques as [7]. That is, for the tuple $(p, (p, np, ns))$ we store $(F_{fk}(p), Enc_{ek}(p, np, ns))$ at the server. To process a

query, the data owner will query all prefixes of the query in parallel.

2. The data owner uses the data structure outlined in section 4.3 to maintain information about previous queries. This is used to prevent the data owner for asking about the same prefix multiple times, and to know when a dummy record needs to be queried (i.e., has the interesting prefix for the query already been queried).

3. The data owner uses the data structure outlined in section 4.4 to maintain information about the changes that have been made during the query phase. This is used during the query phase to ensure that the responses include the recent changes.

4. During the rebuild phase, all of the changes in the second data structure are stored on the server. Like [7] all of the values are randomly permuted (using the Buffer Shuffle techniques) to obfuscate the relationship between queries in different query phases.

Table 1 describes the notation used in the protocols.

It is worth discussing the various input values for the PRF that are used. Specifically, we use a PRF $F : \{0,1\}^\kappa \times \{REAL, DUMMY, MISS, PAD\} \times \cup_{i=1}^{Q}\{0,1\}^i \rightarrow \{0,1\}^\kappa$. That is, the PRF takes a message type and a variable length message (up to Q bits) as its second input. Here the value of Q is chosen such that $2^Q \geq \max\{D, N\log D, M, M\log D\}$. Such a PRF can be constructed with an appropriate encoding scheme. We assume that encryption pads message of variable length to the same size (in this case $3\log D$ is sufficient), and we assume the existence of a fake message *FAKE* that can be used for padding and dummy values.

We begin with the initialization algorithm. This algorithm is done once when the system is setup. The details are in Algorithm 3. The first steps (lines 1-2) is to set up the long-term encryption key and the query phase pseudorandom function key. In lines 3-13, the client generates the values that the server will store, which will consist of the PRF of a key and an encrypted message body. Specifically, lines 3-7, add all interesting prefixes to the server storage set. Since there will be at most N items, then there will be at most $N\log D + 1$ interesting prefixes (see Theorem 1), and thus lines 8-10 add padding to the list. Finally, $2M$ dummy values are added to the server set in lines 11-13. A random permutation of these values is stored in line 15. Finally, lines 15-18 initialize global variables used by the rest of the algorithms.

We now turn to the server's main algorithm (we also require the server can stream all tuples in its data store to the data owner M at a time). This algorithm receives

Algorithm 3 $INIT(K = \{k_1, \ldots, k_n\})$

1: $fk \leftarrow \{0,1\}^\kappa$
2: $ek \leftarrow KeyGen(1^\kappa)$
3: $S \leftarrow \{\}$
4: {Store values on server}
5: Let $IP \leftarrow ALLPREFIXES(k_1, \ldots, k_n)$
6: **for all** $(p, (np, ns)) \in IP$ **do**
7: $\quad S \leftarrow S \cup \{F_{fk}(REAL, p), Enc_{ek}((p, np, ns))\}$
8: **end for**
9: **for** $i = 1$ to $N\log D + 1 - |IP|$ **do**
10: $\quad S \leftarrow S \cup \{F_{fk}(PAD, i), Enc_{ek}(FAKE)\}$
11: **end for**
12: **for** $i = 1$ to $2M$ **do**
13: $\quad S \leftarrow S \cup \{F_{fk}(DUMMY, i), Enc_{ek}(FAKE)\}$
14: **end for**
15: Permute S and send to server.
16: $DS1 \leftarrow InitalizeCommonQuery()$
17: $DS2 \leftarrow InitalizeIntervals()$
18: $queries, dumUsed, misUsed \leftarrow 0$

a set of keys (key values that have the PRF applied to it). The server simply looks up all matching keys that are in S, and returns the messages associated with the keys. The server also returns the query index for each match, so that the client knows which queries were a match. Note that this leaks to the server the number of hits, so this can only be used when the number of hits is controllable (i.e., always the same). The details are in Algorithm 4.

Algorithm 4 $SERVERPROCESS(\ell_1, \ldots, \ell_m)$

1: $R \leftarrow \{\}$
2: **for** $i = 1$ to m **do**
3: $\quad$ **if** $\exists(\ell_i, r_i) \in S$ **then**
4: $\quad\quad R \leftarrow R \cup \{(i, r_i)\}$
5: $\quad\quad$ remove (ℓ_i, r_i) from S
6: $\quad$ **end if**
7: **end for**
8: **return** R

We now turn to a nearest neighbor algorithm for static data. This is used as a building block by the actual nearest neighbor algorithm. The details are in Algorithm 5. The first step is to determine the longest common prefix with previous queries and to determine if the answer is known already. This is done using $DS1$ in line 1. Line 2 creates the list of the longest L prefixes that have not been queried before. Lines 4-8, handle the case where the prefix group containing the query is already known. In this case, one of the misses must be a hit (in order to ensure that the server always sees a single hit), and so a dummy is added to Q, and the number of misses is decremented. Lines 9-10 add the appropriate number of misses to the query set, so that

Table 1. Notation

Name	Description
N	Upper bound on number of keys (constant)
M	Queries in the query phase (constant)
κ	Security parameter (constant)
D	Key domain size (constant and power of 2)
$DS1$	Reference to Data Structure in section 4.3
$DS2$	Reference to Data Structure in section 4.4
fk	Key for pseudorandom function F
ek	Encryption key for CPA-secure encryption
$queries$	# of queries made during query phase.
$dumUsed$	# of dummy queries used during query phase.
$misUsed$	# of fake misses used during query phase.

$|Q| = \log D$. Lines 11-12 permute the queries and send the PRF values for each query in Q to the server. If the query answer was unknown before asking the query, then lines 15 sets the nearest neighbor as the decrypted result from the server. Otherwise, line 17, uses the previous value from $DS1$. In either case, $DS1$ is updated and the nearest neighbors are returned.

Algorithm 5 $LOOKUP(q)$

1: $(L, nn) \leftarrow DS1.get(q)$
2: $Q \leftarrow \{(REAL, Prefix_{d-i}(q)) : i \in [0, L-1]\}$
3: $misses \leftarrow \log D - L$
4: **if** $nn \neq null$ **then**
5: $\quad dumUsed \leftarrow dumUsed + 1$
6: $\quad Q \leftarrow Q \cup \{(DUMMY, dumUsed)\}$
7: $\quad misses \leftarrow misses - 1$
8: **end if**
9: $Q \leftarrow \{(MISS, misUsed + i) : i \in [1, misses]\}$
10: $misUsed \leftarrow misUsed + misses$
11: Permute Q
12: Send to server $\{F_{fk}(r) : r \in Q\}$ and receive QR.
13: {QR will contain one encrypted tuple, let it be (i, r).}
14: **if** $nn = null$ **then**
15: $\quad (p, np, ns) \leftarrow Dec_{ek}(r)$
16: **else**
17: $\quad$ Parse nn into (p, np, ns)
18: **end if**
19: $DS1.insert(q, p, np, ns)$
20: **return** (np, ns)

We now introduce the main algorithm, i.e., the nearest neighbor algorithm. This takes a query, and returns the nearest predecessor and successor of the query; the details are in Algorithm 6. In line 1, this looks up the query in the interval data structure to determine if the interval of the query is already known. Note that it may be that this value is more recent than the values stored in the server, since all updates affect only $DS2$ until the rebuild phase. If the interval is known, but the interval is invalid, then this means that the nearest successor has been removed. Thus, the answer returned from the server from q will be stale, and $DS2$ does not contain the correct nearest successor. To resolve this problem, line 3 changes the query to the one more than the stale nearest successor. Then line 5 either looks up the query or the modified query, using Algorithm 5. This new interval is added to $DS2$, which means that a valid interval containing q is now in $DS2$. Thus we lookup q in DS2 (in Line 7). Finally, we increment the number of queries and return the appropriate nearest predecessor and successor.

Algorithm 6 $NN(q)$

1: $(np, ns, valid) \leftarrow DS2.lookup(q)$
2: **if** $(np, ns, valid) \neq null$ AND $valid = false$ **then**
3: $\quad q \leftarrow ns + 1$
4: **end if**
5: $(np', ns') \leftarrow LOOKUP(q)$
6: $DS2.insertInterval(np', ns')$
7: $(np', ns', valid) \leftarrow DS2.lookup(q)$
8: $queries \leftarrow queries + 1$
9: **return** (np', ns')

The algorithm for insertion (resp. removal) are given in Algorithm 7 (resp. 8). In both algorithms, the client uses the nearest neighbor algorithm. Then insertion simply inserts the new point into $DS2$, and removal simply removes the query from $DS2$. Note that in both cases the precondition is met, because $NN(q)$ ensures that a valid interval containing q is in $DS2$.

Algorithm 7 $Insert(q)$

1: $NN(q)$
2: $DS2.insertPoint(q)$

We now turn our attention to the rebuilding phase (this is triggered when *queries* = M). We first present

Algorithm 8 $Remove(q)$

1: $NN(q)$
2: $DS2.removePoint(q)$.

a helper algorithm that ensures all intervals in the interval structure, $DS2$, are valid. This is important, because any invalid interval corresponds to a situation where an endpoint has been removed, but the client doesn't know what the actual endpoint should be. Lines 1-6 iterate through all intervals in $DS2$ and for every invalid interval, it adds a query to Q that will make the interval valid (once we know the interval for the query). To hide the number of invalid intervals from the server, lines 8-10, pad the query set with to contain M points (there are at most M invalid intervals, because each remove can invalidate at most one interval). The padded points are dummy points, because they need to be hits on the server. This is the reason for needing $2M$ dummies, M to answer queries and and M for the rebuild phase. The client computes the PRF of all points in Q and sends them to the server in a random order in line 11. Lines 12-15, process each non-dummy return value by adding it to $DS2$. This will validate all intervals in $DS2$.

Algorithm 9 $ValidateAllIntervals()$

1: $DS2.first()$
2: $Q \leftarrow \{\}$
3: **while** $((x, y, valid) \leftarrow DS2.next()) \neq null$ **do**
4: **if** valid = false **then**
5: $Q \leftarrow Q \cup \{(REAL, y + 1)\}$
6: **end if**
7: **end while**
8: **for** $i = 1$ to $M - |Q|$ **do**
9: $Q \leftarrow Q \cup \{(DUMMY, dumUsed + i)\}$
10: **end for**
11: Send $\{F_{fk}(q) : q \in Q\}$ to server in random order.
12: **for all** Entry (i, r_i) corresponding to non-dummy **do**
13: $(p, np, ns) \leftarrow Dec_{ek}(r_i)$
14: $DS2.insertInterval(np, ns)$
15: **end for**

The main idea of the rebuild phase is to rewrite all $N \log D + 1 + 2M$ values to the server and then to reshuffle all of the buffers (the reshuffling is the same as in traditional OS). The client suspends execution of queries, and then chooses a new PRF key (lines 1-3). The client initializes some values, including: $prefixSet$ which is a set of prefixes to be written and $padWrite$ which is how much padding has been written (lines 4-5). The client then validates all intervals in DS2 (line 6). After this has been done, the client streams (by streams we mean that the client obtains M records at a time from the server, in order to prevent the client from having to store more than $O(M)$ things) the remaining $N \log D + 1$ entries from the server. For each entry, there are several cases: i) the interval specified by the prefix is not contained in $DS2$, i) the interval specified by the prefix is contained in $DS2$, iii) the tuple is a dummy or padding tuple. In the first case (line 27), the client simply re-encrypts the tuple (as it has not changed). In the other cases, the client throws the old tuple away, and builds a new tuple. To build this new tuple, the client first writes out all interesting prefixes in DS2. After all of these values have been written, then padding is written. After going through all $N \log D + 1$ tuples, the server has all interesting prefixes and the appropriate amount of padding. Then $2M$ dummy values are written (lines 31-33). After writing all of these entries, global variables are re-initialized (lines 36-38). All of the values are permuted using the techniques of [7], and then query processing is resumed.

4.6. Analysis

The client's storage is determined by the size of $DS1$ and $DS2$. For each nearest neighbor query, there is at most 1 thing in $DS1$, and thus its size is $O(M)$. Furthermore, $DS2$ has at most 2M intervals, and thus its size is $O(M)$. The client has to store $O(\log D)$ bits, and thus its total storage is $O(M \log D)$.

The server has to store $O(N \log D + M)$ items, and each has size $O(\max\{\kappa, \log D\})$. Since κ is a constant and $M << N$, then the server's total bit storage is $O(N \log^2 D)$.

The communication to process an insert, remove, or nn query is $O(1)$ for both the client and the server. Furthermore, the communication is $O(1)$.

The computational cost of the rebuilding phase is $O(N \log D)$, and the communication cost is $O(N \log^2 D)$.

It is worth comparing this solution to the original cost of OS, to determine the overhead of the nearest neighbor capabilities. Here V is the size of the messages associated with the keys. It is clear from the table that the overhead is dictated by the relationship between V and $\log^2 D$. There are many application with small key size (for example a key size of 8 bytes may be sufficient in many contexts). However, in many applications the sizes of the messages are large (the simulations used in [7] varied V from 1KB to 64 KB). In either case the $O(N \log^2 D)$ is dominated by $O(NV)$. Hence, the overhead added by the current approach is modest when compared to OS.

5. Summary

In this paper, we introduced an oblivious index that extends oblivious storage to support nearest neighbor queries. In realistic settings, the proposed index

Table 2. Comparison between the Original Scheme and the Index Overhead

Metric	Traditional	Index Overhead
Server Storage	$O(NV)$	$O(N \log^2 D)$
Client Storage	$O(MV)$	$O(M \log D)$
Online computation	$O(1)$	$O(1)$
Online communication	$O(V)$	$O(\log D)$
Rebuild Communication	$O(NV)$	$O(N \log^2 D)$

Algorithm 10 *rebuild*()

{Triggered when $QUERIES = M$}
Suspend query processing
$fk' \leftarrow PRF.Gen(1^\kappa)$
$prefixSet \leftarrow \{\}$
$padWrite \leftarrow 0$
$validateIntervals()$
$DS2.first()$
{Start streaming remaining records from server}
for all $(Enc_{ek}(p, np, ns)$ in Server storage **do**
  Decrypt to obtain (p, np, ns)
  if (p,np,ns)=*FAKE* OR DS2.containsPoint(ns) **then**
    if $prefixSet = \{\}$ **then**
      $interval \leftarrow DS2.next()$
      if $interval \neq null$ **then**
        $prefixSet \leftarrow SPLIT(interval)$
      end if
    end if
    if $prefixSet \neq \{\}$ **then**
      Pick $(p2, np2, ns2)$ from $prefixSet$
      $prefixSet \leftarrow prefixSet - \{(p2, np2, ns2)\}$
      $(k, v) \leftarrow ((REAL, p2), (p2, np2, ns2))$
    else
      $padWrite \leftarrow padWrite + 1$
      $(k, v) \leftarrow ((PAD, padWrite), FAKE)$
    end if
  else
    $(k, v) \leftarrow (p, np, ns)$
  end if
  Send server $(F_{fk'}(k), Enc_{ek}(v))$
end for
for $i = 1$ to $2M$ **do**
  $(k, v) \leftarrow ((DUMMY, i), FAKE)$
  Send $(F_{fk'}(k), Enc_{ek}(v))$ to server.
end for
$fk \leftarrow fk'$
$queries, dumUsed, misUsed \leftarrow 0$
$DS1 \leftarrow InitalizeCommonQuery()$
$DS2 \leftarrow InitalizeIntervals()$
Shuffle servers storage as in [7]
{Note the above changes the value of fk}
Resume query processing

introduces a small overhead, when compared to the original oblivious data store. Future work includes:

1. Implementing the index and determining actual overhead for realistic loads.

2. Extending a miss-intolerant OS to a miss-tolerant OS using these techniques. It is straightforward to do this for get, but less so for insert and remove.

3. Extending the semantics further to include range queries, range count and aggregate queries. A straightforward way to do range queries is based on *nn* query: the client partitions the key domain into $O(\sqrt{N})$ intervals, stores a key value pair for each one as $(left\_endpoint, (values\_inside\_interval, right\_endpoint))$, and builds the *nn* search index over all the left endpoints. To query for a range $[a, b]$, the client queries for the nearest left endpoint of a and gets all the values in the interval, and continues fetching the next interval by a *nn* query for the current interval's $right\_endpoint + 1$ until exceeding b. Range count and aggregate queries could also be done in a similar way. However, this will increase the local storage at the client to $O(N^{0.75})$, so future work could focus on these semantics without increasing the storage.

5.1. Acknowledgements

Portions of this work were supported by National Science Foundation Grants CPS-1329979, CNS-0915436, Science and Technology Center CCF-0939370; by an NPRP grant from the Qatar National Research Fund; and by sponsors of the Center for Education and Research in Information Assurance and Security. The statements made herein are solely the responsibility of the authors.

References

[1] Apache Software Foundation (2014) Apache HBase home. URL http://hbase.apache.org/.

[2] Boneh, D., Mazieres, D. and Popa., R.A. (2011) *Remote oblivious storage: Making oblivious RAM practical.* Tech. rep., CSAIL, MIT.

[3] Gentry, C., Goldman, K., Halevi, S., Julta, C., Raykova, M. and Wichs, D. (2013) Optimizing ORAM and using it efficiently for secure computation. In *Privacy Enhancing Technologies* (Springer Berlin Heidelberg), **7981**, 1–18.

[4] Goldreich, O. (1987) Towards a theory of software protection and simulation by oblivious rams. In *Proceedings of the 19th annual ACM Symposium on Theory of Computing*, STOC '87 (New York, NY, USA: ACM): 182–194.

[5] Goodrich, M.T. and Mitzenmacher, M. (2011) Privacy-preserving access of outsourced data via oblivious RAM simulation. In *Proceedings of the 38th International Conference on Automata, Languages and Programming - Volume Part II*, ICALP'11 (Berlin, Heidelberg: Springer-Verlag): 576–587.

[6] Goodrich, M.T., Mitzenmacher, M., Ohrimenko, O. and Tamassia, R. (2011) Oblivious ram simulation with efficient worst-case access overhead. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, CCSW '11 (New York, NY, USA: ACM): 95–100.

[7] Goodrich, M.T., Mitzenmacher, M., Ohrimenko, O. and Tamassia, R. (2012) Practical oblivious storage. In *Proceedings of the 2nd ACM Conference on Data and Application Security and Privacy*, CODASPY '12 (New York, NY, USA: ACM): 13–24.

[8] Hacigümüs, H., Mehrotra, S. and Iyer, B.R. (2002) Providing database as a service. In *18th International Conference on Data Engineering* (IEEE): 29–38.

[9] Knuth, D.E. (1973) *The Art of Computer Programming, Volume III: Sorting and Searching* (Addison-Wesley).

[10] Kushilevitz, E., Lu, S. and Ostrovsky, R. (2012) On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *Proceedings of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12 (SIAM): 143–156.

[11] Li, J. and Omiecinski, E. (2005) Efficiency and security trade-off in supporting range queries on encrypted databases. In *Data and Applications Security XIX* (Springer): 69–83.

[12] Shi, E., Chan, T., Stefanov, E. and Li, M. (2011) Oblivious RAM with $ø(log^3 n)$ worst-case cost. In *Advances in Cryptology-ASIACRYPT'11* (Springer), 197–214.

[13] Stefanov, E., van Dijk, M., Shi, E., Fletcher, C., Ren, L., Yu, X. and Devadas, S. (2013) Path ORAM: An extremely simple oblivious ram protocol. In *Proceedings of the 2013 ACM Conference on Computer and Communications Security*, CCS '13 (New York, NY, USA: ACM): 299–310.

[14] Vimercati, S.D.C.D., Foresti, S., Jajodia, S., Paraboschi, S. and Samarati, P. (2007) Over-encryption: Management of access control evolution on outsourced data. In *Proceedings of the 33rd International Conference on Very Large Data Bases* (VLDB endowment): 123–134.

[15] Williams, P., Sion, R. and Carbunar, B. (2008) Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08 (New York, NY, USA: ACM): 139–148.

[16] Xiao, L. and Yen, I.L. (2012) Security analysis and enhancement for prefix-preserving encryption schemes. *IACR Cryptology ePrint Archive* **2012**: 191.

[17] Yiu, M.L., A., I., Jensen, C.S. and Kalnis, P. (2012) Outsourced similarity search on metric data assets. *IEEE Trans. Knowl. Data Eng.* **24**(2): 338–352.

[18] Yiu, M.L., Ghinita, G., Jensen, C.S. and Kalnis, P. (2009) Outsourcing search services on private spatial data. In *25th International Conference on Data Engineering* (IEEE): 1140–1143.